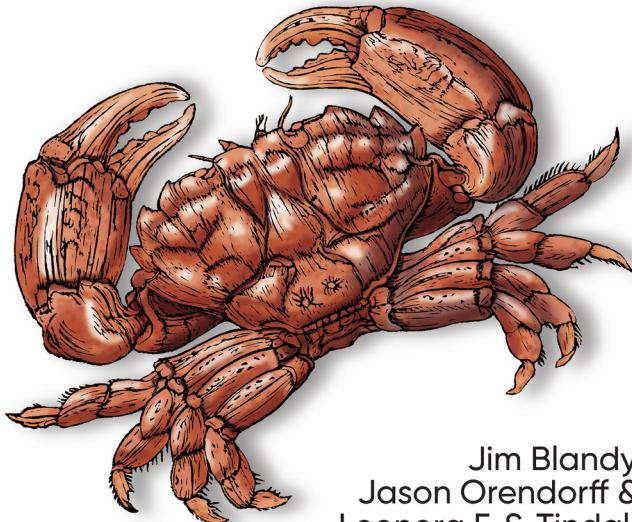


O'REILLY®

Programming Rust

Fast, Safe Systems Development

Revised 2nd Edition
Covers Rust 2021 Edition



Jim Blandy,
Jason Orendorff &
Leonora F. S. Tindall

Préface

Rust est un langage de programmation système.

Cela mérite une explication de nos jours, car la programmation système n'est pas familière à la plupart des programmeurs en activité. Pourtant, cela sous-tend tout ce que nous faisons.

Vous fermez votre ordinateur portable. Le système d'exploitation le détecte, suspend tous les programmes en cours d'exécution, éteint l'écran et met l'ordinateur en veille. Plus tard, vous ouvrez l'ordinateur portable : l'écran et les autres composants sont à nouveau alimentés, et chaque programme est capable de reprendre là où il s'était arrêté. Nous tenons cela pour acquis. Mais les programmeurs système ont écrit beaucoup de code pour que cela se produise.

Programmation des systèmes est pour :

- Systèmes d'exploitation
- Pilotes de périphériques de toutes sortes
- Systèmes de fichiers
- Bases de données
- Code qui s'exécute dans des appareils très bon marché, ou des appareils qui doivent être extrêmement fiables
- Cryptographie
- Codecs multimédias (logiciels de lecture et d'écriture de fichiers audio, vidéo et image)
- Traitement des médias (par exemple, logiciel de reconnaissance vocale ou de retouche photo)
- Gestion de la mémoire (par exemple, implémentation d'un ramasse-miettes)
- Rendu du texte (la conversion du texte et des polices en pixels)
- Implémentation de langages de programmation de niveau supérieur (comme JavaScript et Python)
- La mise en réseau
- Virtualisation et conteneurs logiciels
- Simulations scientifiques

- Jeux

En bref, la programmation système est *limitée en ressources* programmation. C'est de la programmation lorsque chaque octet et chaque cycle CPU comptent.

La quantité de code système impliquée dans la prise en charge d'une application de base est stupéfiante.

Ce livre ne vous apprendra pas la programmation système. En fait, ce livre couvre de nombreux détails sur la gestion de la mémoire qui peuvent sembler inutilement abstrus au premier abord, si vous n'avez pas déjà fait de programmation système par vous-même. Mais si vous êtes un programmeur système chevronné, vous constaterez que Rust est quelque chose d'exceptionnel : un nouvel outil qui élimine les problèmes majeurs et bien compris qui ont tourmenté toute une industrie pendant des décennies.

Qui devrait lire ce livre

Si vous êtes déjà un programmeur système et que vous êtes prêt pour une alternative au C++, ce livre est pour vous. Si vous êtes un développeur expérimenté dans n'importe quel langage de programmation, que ce soit C#, Java, Python, JavaScript ou autre chose, ce livre est également pour vous.

Cependant, vous n'avez pas seulement besoin d'apprendre Rust. Pour tirer le meilleur parti du langage, vous devez également acquérir une certaine expérience de la programmation système. Nous vous recommandons de lire ce livre tout en implémentant certains projets parallèles de programmation système dans Rust. Construisez quelque chose que vous n'avez jamais construit auparavant, quelque chose qui tire parti de la vitesse, de la simultanéité et de la sécurité de Rust. La liste des sujets au début de cette préface devrait vous donner quelques idées.

Pourquoi nous avons écrit ce livre

Nous avons entrepris d'écrire le livre que nous aurions aimé avoir lorsque nous avons commencé à apprendre Rust. Notre objectif était

d'aborder les nouveaux grands concepts de Rust de front et de front, en les présentant clairement et en profondeur afin de minimiser l'apprentissage par essais et erreurs.

Naviguer dans ce livre

Les deux premiers chapitres de ce livre présentent Rust et fournissent une brève visite guidée avant de passer aux types de données fondamentaux au [chapitre 3](#). Les chapitres [4](#) et [5](#) abordent les concepts fondamentaux de propriété et de références. Nous vous recommandons de lire ces cinq premiers chapitres dans l'ordre.

Les chapitres [6](#) à [10](#) couvrent les bases du langage : expressions ([chapitre 6](#)), gestion des erreurs ([chapitre 7](#)), crates et modules ([chapitre 8](#)), structures ([chapitre 9](#)) et énumérations et modèles ([chapitre 10](#)). C'est bien de survoler un peu ici, mais ne sautez pas le chapitre sur la gestion des erreurs. Fais nous confiance.

[Le chapitre 11](#) couvre les traits et les génériques, les deux derniers grands concepts que vous devez connaître. Les traits sont comme des interfaces en Java ou C#. Ils sont également le principal moyen pour Rust de prendre en charge l'intégration de vos types dans le langage lui-même. [Le chapitre 12](#) montre comment les traits prennent en charge la surcharge d'opérateurs, et le [chapitre 13](#) couvre de nombreux autres traits utilitaires.

Comprendre les traits et les génériques ouvre le reste du livre. Les fermes-tures et les itérateurs, deux outils puissants que vous ne voudrez pas manquer, sont traités respectivement dans les chapitres [14](#) et [15](#). Vous pouvez lire les chapitres restants dans n'importe quel ordre, ou simplement y plonger au besoin. Ils couvrent le reste du langage : collections ([Chapitre 16](#)), chaînes et texte ([Chapitre 17](#)), entrée et sortie ([Chapitre 18](#)), concurrence ([Chapitre 19](#)), programmation asynchrone ([Chapitre 20](#)), macros ([Chapitre 21](#)), unsafe code ([Chapitre 22](#)) et appeler des fonctions dans d'autres langages ([Chapitre 23](#)).

Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique de nouveaux termes, URL, adresses e-mail, noms de fichiers et extensions de fichiers.

Constant width

Utilisé pour les listes de programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que des noms de variables ou de fonctions, des bases de données, des types de données, des variables d'environnement, des instructions et des mots-clés.

Constant width bold

Affiche les commandes ou tout autre texte qui doit être tapé littéralement par l'utilisateur.

Constant width italic

Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

NOTER

Cette icône signifie une note générale.

Utiliser des exemples de code

Du matériel supplémentaire (exemples de code, exercices, etc.) est disponible en téléchargement sur <https://github.com/ProgrammingRust>.

Ce livre est là pour vous aider à faire votre travail. En général, si un exemple de code est proposé avec ce livre, vous pouvez l'utiliser dans vos programmes et votre documentation. Vous n'avez pas besoin de nous contacter pour obtenir une autorisation, sauf si vous reproduisez une partie importante du code. Par exemple, écrire un programme qui utilise plusieurs morceaux de code de ce livre ne nécessite pas d'autorisation. La vente ou la distribution d'exemples tirés des livres d'O'Reilly nécessite

une autorisation. Répondre à une question en citant ce livre et en citant un exemple de code ne nécessite pas d'autorisation. L'incorporation d'une quantité importante d'exemples de code de ce livre dans la documentation de votre produit nécessite une autorisation.

Nous apprécions, mais nous ne demandons pas d'attribution. Une attribution comprend généralement le titre, l'auteur, l'éditeur et l'ISBN. Par exemple : « *Programming Rust, deuxième édition* par Jim Blandy, Jason Orendorff et Leonora FS Tindall (O'Reilly). Copyright 2021 Jim Blandy, Leonora FS Tindall et Jason Orendorff, 978-1-492-05259-3.

Si vous pensez que votre utilisation d'exemples de code ne respecte pas l'utilisation équitable ou l'autorisation donnée ci-dessus, n'hésitez pas à nous contacter à l'[adresse permissions@oreilly.com](mailto:adresse_permissions@oreilly.com).

Apprentissage en ligne O'Reilly

NOTER

Depuis plus de 40 ans, [O'Reilly Media](#) fournit des formations, des connaissances et des connaissances en technologie et en affaires pour aider les entreprises à réussir.

Notre réseau unique d'experts et d'innovateurs partage leurs connaissances et leur expertise à travers des livres, des articles, des conférences et notre plateforme d'apprentissage en ligne. La plate-forme d'apprentissage en ligne d'O'Reilly vous donne un accès à la demande à des cours de formation en direct, des parcours d'apprentissage approfondis, des environnements de codage interactifs et une vaste collection de textes et de vidéos d'O'Reilly et de plus de 200 autres éditeurs. Pour plus d'informations, rendez-vous sur <http://oreilly.com>.

Comment nous contacter

Veuillez adresser vos commentaires et questions concernant ce livre à l'éditeur :

- O'Reilly Media, Inc.
- 1005, autoroute Gravenstein Nord
- Sébastopol, Californie 95472
- 800-998-9938 (aux États-Unis ou au Canada)
- 707-829-0515 (international ou local)
- 707-829-0104 (télécopieur)

Nous avons une page Web pour ce livre, où nous énumérons les errata, des exemples et toute information supplémentaire. Vous pouvez accéder à cette page à <https://oreil.ly/programming-rust-2e>.

Envoyez un e-mail à bookquestions@oreilly.com pour commenter ou poser des questions techniques sur ce livre.

Visitez <http://www.oreilly.com> pour plus d'informations sur nos livres et cours.

Retrouvez-nous sur Facebook : <http://facebook.com/oreilly>

Suivez-nous sur Twitter : <http://twitter.com/oreillymedia>

Regardez-nous sur YouTube : <http://youtube.com/oreillymedia>

Remerciements

Le livre que vous tenez a grandement bénéficié de l'attention de nos réviseurs techniques officiels : Brian Anderson, Matt Brubeck, J. David Eisenberg, Ryan Levick, Jack Moffitt, Carol Nichols et Erik Nordin ; et nos traducteurs : Hidemoto Nakada (中田 秀基) (japonais), M. Songfeng Li (chinois simplifié) et Adam Bochenek et Krzysztof Sawka (polonais).

De nombreux autres réviseurs non officiels ont lu les premières ébauches et ont fourni des commentaires inestimables. Nous tenons à remercier Eddy Bruel, Nick Fitzgerald, Graydon Hoare, Michael Kelly, Jeffrey Lim, Jakob Olesen, Gian-Carlo Pascutto, Larry Rabinowitz, Jaroslav Šnajdr, Joe Walker et Yoshua Wuyts pour leurs commentaires judicieux. Jeff Walden et Nicolas Pierron ont été particulièrement généreux de leur temps, révi-

sant presque tout le livre. Comme toute entreprise de programmation, un livre de programmation se nourrit de rapports de bogues de qualité.
Merci.

Mozilla a été extrêmement accommodant avec le travail de Jim et Jason sur ce projet, même s'il ne relevait pas de nos responsabilités officielles et leur faisait concurrence pour attirer notre attention. Nous sommes reconnaissants aux managers de Jim et Jason : Dave Camp, Naveed Ihsanullah, Tom Tromey et Joe Walker, pour leur soutien. Ils ont une vision à long terme de ce qu'est Mozilla ; nous espérons que ces résultats justifient la confiance qu'ils ont placée en nous.

Nous tenons également à exprimer notre gratitude à tous ceux d'O'Reilly qui ont contribué à la réalisation de ce projet, en particulier nos éditeurs étonnamment patients Jeff Bleiel et Brian MacDonald, et notre éditeur d'acquisitions Zan McQuade.

Surtout, nos sincères remerciements à nos familles pour leur amour indéfectible, leur enthousiasme et leur patience.

[Soutien](#) | [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) | [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 1. Les programmeurs système peuvent avoir de belles choses

Dans certains contextes, par exemple le contexte que Rust cible, être 10x ou même 2x plus rapide que la concurrence est une chose décisive. Il décide du sort d'un système sur le marché, autant qu'il le ferait sur le marché du matériel.

—[Graydon Hoare](#)

Tous les ordinateurs sont désormais parallèles...

La programmation parallèle, c'est de la programmation.

—Michael McCool et al., *Programmation parallèle structurée*

Faille de l'analyseur TrueType utilisée par l'attaquant de l'État-nation pour la surveillance ; tous les logiciels sont sensibles à la sécurité.

—[Andy Wingo](#)

Nous avons choisi d'ouvrir notre livre avec les trois citations ci-dessus pour une raison. Mais commençons par un mystère. Que fait le programme C suivant ?

```
int main(int argc, char **argv) {
    unsigned long a[1];
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

Sur l'ordinateur portable de Jim ce matin, ce programme a imprimé :

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Puis il s'est écrasé. Si vous l'essayez sur votre machine, il se peut qu'il fasse autre chose. Que se passe t-il ici?

Le programme est défectueux. Le tableau `a` n'a qu'un seul élément de long, donc l'utilisation `a[3]` est, selon la norme du langage de programmation C, *un comportement indéfini*:

Comportement, lors de l'utilisation d'une construction de programme non portable ou erronée ou de données erronées, pour lequel la présente Norme internationale n'impose aucune exigence

Un comportement indéfini n'a pas seulement un résultat imprévisible : la norme autorise explicitement le programme à faire *n'importe quoi*. Dans notre cas, le stockage de cette valeur particulière dans le quatrième élément de ce tableau particulier corrompt la pile d'appels de fonction de sorte que le retour de la `main` fonction, au lieu de quitter le programme avec élégance comme il se doit, saute au milieu du code du standard C bibliothèque pour récupérer un mot de passe à partir d'un fichier dans le répertoire personnel de l'utilisateur. Ça ne va pas bien.

C et C++ ont des centaines de règles pour éviter les comportements indéfinis. Ils relèvent principalement du bon sens : n'accédez pas à la mémoire que vous ne devriez pas, ne laissez pas les opérations arithmétiques déborder, ne divisez pas par zéro, etc. Mais le compilateur n'applique pas ces règles ; il n'a aucune obligation de détecter les violations, même flagrantes. En effet, le programme précédent se compile sans erreur ni avertissement. La responsabilité d'éviter les comportements indéfinis incombe entièrement à vous, le programmeur.

Empiriquement parlant, nous, les programmeurs, n'avons pas un excellent bilan à cet égard. Alors qu'il était étudiant à l'Université de l'Utah, le chercheur Peng Limodifié les compilateurs C et C++ pour que les programmes qu'ils traduisent signalent s'ils ont exécuté certaines formes de comportement indéfini. Il a constaté que presque tous les programmes le font, y compris ceux de projets très respectés qui maintiennent leur code à des normes élevées. Supposer que vous pouvez éviter un comportement indéfini en C et C++ revient à supposer que vous pouvez gagner une partie d'échecs simplement parce que vous connaissez les règles.

Le message étrange ou le plantage occasionnel peut être un problème de qualité, mais un comportement indéfini par inadvertance a également été une cause majeure de failles de sécurité depuis le Morris Worm de 1988.ont utilisé une variante de la technique présentée précédemment pour se propager d'un ordinateur à un autre sur l'Internet primitif.

Ainsi, C et C++ mettent les programmeurs dans une position délicate : ces langages sont les standards de l'industrie pour la programmation système, mais les exigences qu'ils imposent aux programmeurs garantissent pratiquement un flux constant de plantages et de problèmes de sécurité. Répondre à notre mystère soulève simplement une question plus importante : ne pouvons-nous pas faire mieux ?

Rust assume la charge pour vous

Notre réponse est encadré par nos trois citations d'ouverture. La troisième citation fait référence à des rapports selon lesquels Stuxnet, un ver informatique qui s'est introduit dans des équipements de contrôle industriels en 2010, a pris le contrôle des ordinateurs des victimes en utilisant, parmi de nombreuses autres techniques, un comportement indéfini dans le code qui analyse les polices TrueType intégrées dans les documents de traitement de texte. Il y a fort à parier que les auteurs de ce code ne s'attendaient pas à ce qu'il soit utilisé de cette façon, illustrant que ce ne sont pas seulement les systèmes d'exploitation et les serveurs qui doivent se soucier de la sécurité : tout logiciel susceptible de gérer des données provenant d'une source non fiable pourrait être la cible d'un exploit.

Le langage Rust vous fait une simple promesse : si votre programme passe les vérifications du compilateur, il est exempt de comportement indéfini. Les pointeurs suspendus, les doubles libérations et les déréférencements de pointeurs nuls sont tous interceptés au moment de la compilation. Les références de tableau sont sécurisées avec un mélange de vérifications à la compilation et à l'exécution, de sorte qu'il n'y a pas de dépassement de mémoire tampon : l'équivalent Rust de notre malheureux programme C se termine en toute sécurité avec un message d'erreur.

De plus, Rust se veut à la fois *sûr* et *agréable à utiliser*. Afin de garantir davantage le comportement de votre programme, Rust impose plus de restrictions sur votre code que C et C++, et ces restrictions nécessitent de la

pratique et de l'expérience pour s'y habituer. Mais le langage dans son ensemble est souple et expressif. Ceci est attesté par l'étendue du code écrit en Rust et la gamme de domaines d'application auxquels il est appliqué.

D'après notre expérience, pouvoir faire confiance à la langue pour déterminer plus d'erreurs nous encourage à essayer des projets plus ambitieux. La modification de programmes volumineux et complexes est moins risquée lorsque vous savez que les problèmes de gestion de la mémoire et de validité des pointeurs sont pris en charge. Et le débogage est beaucoup plus simple lorsque les conséquences potentielles d'un bogue n'incluent pas la corruption de parties non liées de votre programme.

Bien sûr, il y a encore beaucoup de bogues que Rust ne peut pas détecter. Mais dans la pratique, le fait de retirer de la table les comportements indéfinis modifie considérablement le caractère du développement pour le mieux.

La programmation parallèle est apprivoisée

Concurrence est notoirement difficile à utiliser correctement en C et C++. Les développeurs se tournent généralement vers la simultanéité uniquement lorsque le code à thread unique s'est avéré incapable d'atteindre les performances dont ils ont besoin. Mais la deuxième citation d'ouverture soutient que le parallélisme est trop important pour les machines modernes pour être considéré comme une méthode de dernier recours.

Il s'avère que les mêmes restrictions qui garantissent la sécurité de la mémoire dans Rust garantissent également que les programmes Rust sont exempts de courses de données. Vous pouvez partager librement des données entre les threads, tant qu'elles ne changent pas. Les données qui changent ne sont accessibles qu'à l'aide de primitives de synchronisation. Tous les outils de concurrence traditionnels sont disponibles : mutex, variables de condition, canaux, atomiques, etc. Rust vérifie simplement que vous les utilisez correctement.

Cela fait de Rust un excellent langage pour exploiter les capacités des machines multicœurs modernes. L'écosystème Rust propose des bibliothèques et des outils pour faciliter la programmation concurrente.

thèques qui vont au-delà des primitives de concurrence habituelles et vous aident à répartir uniformément les charges complexes entre les pools de processeurs, à utiliser des mécanismes de synchronisation sans verrouillage tels que Read-Copy-Update, etc.

Et pourtant la rouille est toujours rapide

Ceci, enfin, est notre première citation d'ouverture. Rust partage les ambitions que Bjarne Stroustrup exprime pour C++ dans son article "Abstraction and the C++ Machine Model" :

En général, les implémentations C++ obéissent au principe de zéro surcharge: Ce que vous n'utilisez pas, vous ne le payez pas. Et plus loin : ce que vous utilisez, vous ne pourriez pas coder mieux.

La programmation système consiste souvent à pousser la machine à ses limites. Pour les jeux vidéo, toute la machine devrait être consacrée à créer la meilleure expérience pour le joueur. Pour les navigateurs Web, l'efficacité du navigateur fixe le plafond de ce que les auteurs de contenu peuvent faire. Dans les limites inhérentes à la machine, autant d'attention mémoire et processeur que possible doit être laissée au contenu lui-même. Le même principe s'applique aux systèmes d'exploitation : le noyau doit mettre les ressources de la machine à la disposition des programmes utilisateurs, et non les consommer lui-même.

Mais quand nous disons que Rust est "rapide", qu'est-ce que cela signifie vraiment ? On peut écrire du code lent dans n'importe quel langage généraliste. Il serait plus précis de dire que, si vous êtes prêt à investir pour concevoir votre programme afin d'utiliser au mieux les capacités de la machine sous-jacente, Rust vous soutient dans cet effort. Le langage est conçu avec des valeurs par défaut efficaces et vous donne la possibilité de contrôler la façon dont la mémoire est utilisée et la façon dont l'attention du processeur est dépensée.

Rust facilite la collaboration

Nous avons caché une quatrième citation dans le titre de ce chapitre : "Les programmeurs systèmes peuvent avoir de belles choses." Cela fait référence à la prise en charge par Rust du partage et de la réutilisation du code.

Gestionnaire de paquets et outil de construction de Rust, Cargo, facilite l'utilisation des bibliothèques publiées par d'autres sur le référentiel de packages public de Rust, le site Web crates.io. Vous ajoutez simplement le nom de la bibliothèque et le numéro de version requis à un fichier, et Cargo s'occupe de télécharger la bibliothèque, ainsi que toutes les autres bibliothèques qu'elle utilise à son tour, et de relier le tout ensemble. Vous pouvez considérer Cargo comme la réponse de Rust à NPM ou RubyGems, en mettant l'accent sur une bonne gestion des versions et des versions reproductibles. Il existe des bibliothèques Rust populaires fournissant tout, de la sérialisation standard aux clients et serveurs HTTP et aux API graphiques modernes.

Pour aller plus loin, le langage lui-même est également conçu pour prendre en charge la collaboration : les traits et les génériques de Rust vous permettent de créer des bibliothèques avec des interfaces flexibles afin qu'elles puissent servir dans de nombreux contextes différents. Et la bibliothèque standard de Rust fournit un ensemble de types fondamentaux qui établissent des conventions partagées pour les cas courants, facilitant ainsi l'utilisation simultanée de différentes bibliothèques.

Le chapitre suivant vise à concrétiser les affirmations générales que nous avons faites dans ce chapitre, avec une visite de plusieurs petits programmes Rust qui montrent les atouts du langage..

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 2. Un tour de Rust

Rouillerlance un défi aux auteurs d'un livre comme celui-ci : ce qui donne à la langue son caractère n'est pas une caractéristique spécifique et étonnante que nous pouvons montrer sur la première page, mais plutôt la façon dont toutes ses parties sont conçues pour fonctionner ensemble en douceur au service des objectifs que nous avons définis dans le chapitre précédent : une programmation système sûre et performante. Chaque partie du langage est mieux justifiée dans le contexte de tout le reste.

Ainsi, plutôt que de s'attaquer à une fonctionnalité du langage à la fois, nous avons préparé une visite guidée de quelques petits programmes complets, dont chacun présente quelques fonctionnalités supplémentaires du langage, en contexte :

- En guise d'échauffement, nous avons un programme qui effectue un calcul simple sur ses arguments de ligne de commande, avec des tests unitaires. Cela montre les types de base de Rust et introduit les *traits*.
- Ensuite, nous construisons un serveur Web. Nous utiliserons une bibliothèque tierce pour gérer les détails de HTTP et introduire la gestion des chaînes, les fermetures et la gestion des erreurs.
- Notre troisième programme trace une belle fractale, répartissant le calcul sur plusieurs threads pour plus de rapidité. Cela inclut un exemple de fonction générique, illustre comment gérer quelque chose comme un tampon de pixels et montre le support de Rust pour la concurrence.
- Enfin, nous montrons un outil de ligne de commande robuste qui traite les fichiers à l'aide d'expressions régulières. Cela présente les fonctionnalités de la bibliothèque standard Rust pour travailler avec des fichiers et la bibliothèque d'expressions régulières tierce la plus couramment utilisée.

La promesse de Rust d'empêcher un comportement indéfini avec un impact minimal sur les performances influence la conception de chaque partie du système, des structures de données standard comme les vecteurs et les chaînes à la façon dont les programmes Rust utilisent des bibliothèques tierces. Les détails de la façon dont cela est géré sont couverts tout au long du livre. Mais pour l'instant, nous voulons vous montrer que Rust est un langage capable et agréable à utiliser.

Tout d'abord, bien sûr, vous devez installer Rust sur votre ordinateur.

Rustup et Cargo

Le meilleurLa façon d'installer Rust est d'utiliser `rustup`. Allez sur <https://rustup.rs> et suivez les instructions.

Vous pouvez également vous rendre sur le [site Web de Rust](#) pour obtenir des packages prédéfinis pour Linux, mac OS, et Windows. Rust est également inclus dans certaines distributions de systèmes d'exploitation. Nous préférons `rustup` car c'est un outil de gestion des installations Rust, comme RVM pour Ruby ou NVM pour Node. Par exemple, lorsqu'une nouvelle version de Rust est publiée, vous pourrez mettre à niveau sans aucun clic en tapant `rustup update`.

Dans tous les cas, une fois l'installation terminée, vous devriez avoir trois nouvelles commandes disponibles sur votre ligne de commande :

```
$cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

Ici, `$` est l'invite de commande ; sous Windows, ce serait `C:\>` ou quelque chose de similaire. Dans cette transcription, nous exécutons les trois commandes que nous avons installées, en demandant à chacune de signaler de quelle version il s'agit. Prenant chaque commande à tour de rôle :

- `cargo` est la compilation de Rustgestionnaire, gestionnaire de paquets et outil à usage général. Vous pouvez utiliser Cargo pour démarrer un nouveau projet, créer et exécuter votre programme et gérer toutes les bibliothèques externes dont dépend votre code.
- `rustc` est le compilateur Rust. Habituellement, nous laissons Cargo invoquer le compilateur pour nous, mais il est parfois utile de l'exécuter directement.
- `rustdoc` est la documentation Rustoutil. Si vous écrivez une documentation dans les commentaires de la forme appropriée dans le code source de votre programme, `rustdoc` vous pouvez créer du code HTML bien formaté à partir d'eux. Par exemple `rustc`, nous laissons habituellement Cargo courir `rustdoc` pour nous.

Pour plus de commodité, Cargo peut créer un nouveau package Rust pour nous, avec des métadonnées standard organisées de manière appropriée :

```
$cargo nouveau bonjour
Created binary (application) `hello` package
```

Cette commande crée un nouveau répertoire de package nommé `hello`, prêt à créer un exécutable en ligne de commande.

En regardant dans le répertoire de niveau supérieur du package :

```
$ cd bonjour
$ls
-latotal 24
drwxrwxr-x. 4 jimb jimb 4096 Sep 22 21:09 .
drwx----- 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x. 6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r-- 1 jimb jimb 7 Sep 22 21:09 .gitignore
-rw-rw-r-- 1 jimb jimb 88 Sep 22 21:09 Cargo.toml
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:09 src
```

Nous pouvons voir que Cargo a créé un fichier *Cargo.toml* pour contenir les métadonnées du paquet. Pour le moment ce fichier ne contient pas grand chose :

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Si jamais notre programme acquiert des dépendances sur d'autres bibliothèques, nous pouvons les enregistrer dans ce fichier, et Cargo se chargera de télécharger, construire et mettre à jour ces bibliothèques pour nous. Nous aborderons le fichier *Cargo.toml* en détail au [chapitre 8](#).

Cargo a configuré notre package pour une utilisation avec le `git` système de contrôle de version, en créant un sous-répertoire de métadonnées `.git` et un fichier `.gitignore`. Vous pouvez dire à Cargo de sauter cette étape en passant `--vcs none` à `cargo new` sur la ligne de commande.

Le sous-répertoire `src` contient le code Rust réel :

```
$ cd src
$ls -l
total 4
-rw-rw-r-- 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Il semble que Cargo ait commencé à écrire le programme en notre nom. Le fichier `main.rs` contient le texte :

```
fn main() {
    println!("Hello, world!");
}
```

Dans Rust, vous n'avez même pas besoin d'écrire votre propre "Hello, World!" programme. Et c'est l'étendue du passe-partout pour un nouveau

programme Rust : deux fichiers, totalisant treize lignes.

Nous pouvons invoquer la `cargo run` commande depuis n'importe quel répertoire du package pour construire et exécuter notre programme :

```
$course de fret
    Compiling hello v0.1.0 (/home/jimb/rust/hello)
        Finished dev [unoptimized + debuginfo] target(s) in 0.28s
            Running `~/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Ici, Cargo a appelé le compilateur Rust, `rustc`, puis a exécuté l'exécutable qu'il a produit. Cargo place l'exécutable dans le sous-répertoire *cible* en haut du package :

```
$ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb 198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb 68 Sep 22 21:37 incremental
$../target/debug/hello
Hello, world!
```

Lorsque nous avons terminé, Cargo peut nettoyer les fichiers générés pour nous:

```
$cargaison propre
$../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Fonctions de rouille

La syntaxe de Rust est délibérément sans originalité. Si vous êtes familier avec C, C++, Java ou JavaScript, vous pouvez probablement vous y retrouver dans la structure générale d'un programme Rust. Voici une fonction qui calcule le plus grand commun diviseur de deux entiers, en utilisant [l'algorithme d'Euclide](#). Vous pouvez ajouter ce code à la fin de `src/main.rs` :

```
fn gcd(mut n: u64, mut m: u64) ->u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = n % m;
    }
    return n;
}
```

```

    }
    m = m % n;
}

```

Le `fn` mot clé(prononcé "fun") introduit une fonction. Ici, nous définissons une fonction nommée `gcd`, qui prend deux paramètres `n` et `m`, dont chacun est de type `u64`, un entier 64 bits non signé. Le `->` jeton précède le type de retour : notre fonction retourne une `u64` valeur. L'indentation à quatre espaces est de style Rust standard.

Entier machine de Rust les noms de type reflètent leur taille et leur signature : `i32` est un entier 32 bits signé ; `u8` est un entier 8 bits non signé (utilisé pour les valeurs "octet"), et ainsi de suite. Les types `isize` et `usize` contiennent des entiers signés et non signés de la taille d'un pointeur, d'une longueur de 32 bits sur les plates-formes 32 bits et de 64 bits sur les plates-formes 64 bits. Rust a également deux types à virgule flottante, `f32` et `f64`, qui sont les types à virgule flottante simple et double précision IEEE, comme `float` et `double` en C et C++.

Par défaut, une fois qu'une variable est initialisée, sa valeur ne peut pas être modifiée, mais placer le mot- `mut` clé (prononcé "mute", abréviation de *mutable*) avant les paramètres `n` et `m` permet à notre corps de fonction de leur attribuer. En pratique, la plupart des variables ne sont pas affectées ; le `mut` mot clé sur ceux qui le font peut être un indice utile lors de la lecture du code.

Le corps de la fonction commence par un appel à la `assert!` macro, en vérifiant qu'aucun des arguments n'est nul. Le `!` caractère marque cela comme une invocation de macro, pas un appel de fonction. Comme la `assert` macro en C et C++, Rust `assert!` vérifie que son argument est vrai, et si ce n'est pas le cas, termine le programme avec un message utile incluant l'emplacement source de la vérification défaillante ; ce genre d'arrêt brutal s'appelle une *panique*. Contrairement à C et C++, dans lesquels les assertions peuvent être ignorées, Rust vérifie toujours les assertions, quelle que soit la manière dont le programme a été compilé. Il y a aussi une `debug_assert!` macro, dont les assertions sont ignorées lorsque le programme est compilé pour la vitesse.

Le cœur de notre fonction est une `while` boucle contenant une `if` déclaration et une affectation. Contrairement à C et C++, Rust ne nécessite pas de parenthèses autour des expressions conditionnelles, mais il nécessite des accolades autour des instructions qu'ils contrôlent.

Une `let` déclaration déclare une variable locale, comme `t` dans notre fonction. Nous n'avons pas besoin d'écrire `t` le type de , tant que Rust peut le déduire de la façon dont la variable est utilisée. Dans notre fonction, le seul type qui fonctionne pour `t` est `u64`, correspondant `m` à et `n`.

Rust ne déduit que les types dans les corps de fonction : vous devez écrire les types de paramètres de fonction et les valeurs de retour, comme nous l'avons fait auparavant. Si nous voulions épeler `t` le type de `m`, nous pourrions écrire :

```
let t:u64 = m;
```

Rust a une `return` déclaration, mais la `gcd` fonction n'en a pas besoin. Si le corps d'une fonction se termine par une expression *non* suivie d'un point-virgule, il s'agit de la valeur de retour de la fonction. En fait, tout bloc entouré d'accolades peut fonctionner comme une expression. Par exemple, il s'agit d'une expression qui imprime un message puis renvoie `x.cos()` comme valeur :

```
{
    println!("evaluating cos x");
    x.cos()
}
```

Il est typique dans Rust d'utiliser ce formulaire pour établir la valeur de la fonction lorsque le contrôle "tombe à la fin" de la fonction et d'utiliser des `return` instructions uniquement pour les premiers retours explicites au milieu d'une fonction.

Rédaction et exécution de tests unitaires

Rouillera un support simple pour les tests intégrés au langage. Pour tester notre `gcd` fonction, nous pouvons ajouter ce code à la fin de `src/main.rs` :

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                  3 * 7 * 11 * 13 * 19),
                  3 * 11);
}
```

Ici, nous définissons une fonction nommée `test_gcd`, qui appelle `gcd` et vérifie qu'elle renvoie des valeurs correctes. Le `#[test]` au-dessus de la définition marque `test_gcd` comme une fonction de test, à ignorer dans les compilations normales, mais incluse et appelée automatiquement si nous exécutons notre programme avec la `cargo test` commande. Nous pouvons avoir des fonctions de test dispersées dans notre arbre source, placées à côté du code qu'elles exécutent, et `cargo test` les rassembleront automatiquement et les exécuteront toutes.

Le `#[test]` marqueur est un exemple d'*attribut*. Les attributs sont un système ouvert pour marquer les fonctions et autres déclarations avec des informations supplémentaires, comme des attributs en C++ et C#, ou des annotations en Java. Ils sont utilisés pour contrôler les avertissements du compilateur et les vérifications de style de code, inclure le code de manière conditionnelle (comme `#ifdef` en C et C++), indiquer à Rust comment interagir avec le code écrit dans d'autres langages, etc. Nous verrons d'autres exemples d'attributs au fur et à mesure.

Avec nos définitions `gcd` et ajoutées au package `hello` que nous avons créé au début du chapitre, et notre répertoire courant quelque part dans la sous-arborescence du package, nous pouvons exécuter les tests comme suit : `test_gcd`

```
$ cargo test
Compiling hello v0.1.0 (/home/jimb/rust/hello)
Finished test [unoptimized + debuginfo] target(s) in 0.35s
Running unitests (/home/jimb/rust/hello/target/debug/deps/hello-2375...)

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Gestion des arguments de ligne de commande

Pour que notre programme prenne une série de nombres comme arguments de ligne de commande et afficher leur plus grand diviseur commun, nous pouvons remplacer la `main` fonction dans `src/main.rs` par ce qui suit :

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
```

```

    d = gcd(d, *m);
}

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

```

Il s'agit d'un gros bloc de code, alors prenons-le morceau par morceau :

```

use std::str::FromStr;
use std::env;

```

La première `use` déclaration apporte la bibliothèque standard *trait* `FromStr` dans la portée. Un trait est un ensemble de méthodes que les types peuvent implémenter. Tout type qui implémente le `FromStr` trait a une `from_str` méthode qui tente d'analyser une valeur de ce type à partir d'une chaîne. Le `u64` type implements `FromStr`, et nous appellerons `u64::from_str` pour analyser nos arguments de ligne de commande. Bien que nous n'utilisions jamais le nom `FromStr` ailleurs dans le programme, un trait doit être dans la portée afin d'utiliser ses méthodes. Nous couvrirons les traits en détail au [chapitre 11](#).

La deuxième `use` déclaration introduit le `std::env` module, qui fournit plusieurs fonctions et types utiles pour interagir avec l'environnement d'exécution, y compris la `args` fonction, qui nous donne accès aux arguments de ligne de commande du programme.

Passons à la `main` fonction du programme:

```
fn main() {
```

Notre `main` fonction ne renvoie pas de valeur, nous pouvons donc simplement omettre le `->` type de retour and qui devrait normalement suivre la liste des paramètres.

```
let mut numbers = Vec::new();
```

Nous déclarons une variable locale mutable `numbers` et l'initialisons à un vecteur vide. `Vec` est le vecteur de croissance de Rusttype, analogue à C++ `std::vector`, une liste Python ou un tableau JavaScript. Même si les vecteurs sont conçus pour être agrandis et rétrécis de manière dynamique, nous devons toujours marquer la variable `mut` pour que Rust nous permette d'ajouter des nombres à la fin.

Le type de `numbers` est `Vec<u64>`, un vecteur de `u64` valeurs, mais comme précédemment, nous n'avons pas besoin de l'écrire. Rust le déduira pour nous, en partie parce que ce que nous poussons sur le vecteur

sont des `u64` valeurs, mais aussi parce que nous passons les éléments du vecteur à `gcd`, qui n'accepte que `u64` des valeurs.

```
for arg in env::args().skip(1) {
```

Ici, nous utilisons une `for` boucle pour traiter nos arguments de ligne de commande, en définissant la variable `arg` sur chaque argument à tour de rôle et en évaluant le corps de la boucle.

Le `std::env` module `args` la fonction renvoie un *itérateur*, une valeur qui produit chaque argument à la demande et indique quand nous avons terminé. Les itérateurs sont omniprésents dans Rust ; la bibliothèque standard comprend d'autres itérateurs qui produisent les éléments d'un vecteur, les lignes d'un fichier, les messages reçus sur un canal de communication et presque tout ce qui a du sens à boucler. Les itérateurs de Rust sont très efficaces : le compilateur est généralement capable de les traduire dans le même code qu'une boucle manuscrite. Nous montrerons comment cela fonctionne et donnerons des exemples au [chapitre 15](#).

Au-delà de leur utilisation avec des `for` boucles, les itérateurs incluent une large sélection de méthodes que vous pouvez utiliser directement. Par exemple, la première valeur produite par l'itérateur renvoyé par `args` est toujours le nom du programme en cours d'exécution. Nous voulons ignorer cela, nous appelons donc la `skip` méthode de l'itérateur pour produire un nouvel itérateur qui omet cette première valeur.

```
numbers.push(u64::from_str(&arg)
              .expect("error parsing argument"));
```

Ici, nous appelons `u64::from_str` pour tenter d'analyser notre argument de ligne de commande `arg` en tant qu'entier 64 bits non signé. Plutôt qu'une méthode que nous invoquons sur une `u64` valeur que nous avons sous la main, il `u64::from_str` y a une fonction associée au `u64` type, semblable à une méthode statique en C++ ou Java. La `from_str` fonction ne renvoie pas `u64` directement, mais plutôt une `Result` valeur qui indique si l'analyse a réussi ou échoué. Une `Result` valeur est l'une des deux variantes suivantes :

- Une valeur écrite `Ok(v)`, indiquant que l'analyse a réussi et `v` est la valeur produite
- Une valeur écrite `Err(e)`, indiquant que l'analyse a échoué et `e` est une valeur d'erreur expliquant pourquoi

Les fonctions qui font tout ce qui pourrait échouer, comme effectuer une entrée ou une sortie ou interagir autrement avec le système d'exploitation, peuvent renvoyer des `Result` types dont les `Ok` variantes portent des résultats positifs (le nombre d'octets transférés, le fichier ouvert, etc.) et dont les `Err` variantes portent un code d'erreur indiquant ce qui s'est

mal passé. Contrairement à la plupart des langages modernes, Rust n'a pas d'exceptions : toutes les erreurs sont gérées à l'aide de `Result` ou panique, comme indiqué au [chapitre 7](#).

On utilise `Result` la `expect` méthode pour vérifier le succès de notre analyse. Si le résultat est un `Err(e)`, `expect` imprime un message qui inclut une description de `e` et quitte le programme immédiatement. Cependant, si le résultat est `Ok(v)`, `expect` retourne simplement `v` lui-même, que nous pouvons enfin pousser à la fin de notre vecteur de nombres.

```
if numbers.len() == 0 {  
    eprintln!("Usage: gcd NUMBER ...");  
    std::process::exit(1);  
}
```

Il n'y a pas de plus grand commun diviseur d'un ensemble vide de nombres, nous vérifions donc que notre vecteur a au moins un élément et quittons le programme avec une erreur si ce n'est pas le cas. On utilise la `eprintln!` macropour écrire notre message d'erreur dans le flux de sortie d'erreur standard.

```
let mut d = numbers[0];  
for m in &numbers[1..] {  
    d = gcd(d, *m);  
}
```

Cette boucle utilise `d` comme valeur courante, la mettant à jour pour rester le plus grand diviseur commun de tous les nombres que nous avons traités jusqu'à présent. Comme précédemment, nous devons marquer `d` comme mutable afin de pouvoir l'affecter dans la boucle.

La `for` boucle a deux parties surprenantes. D'abord, nous avons écrit `for m in &numbers[1..]`; à quoi sert l' `&` opérateur ? Deuxièmement, nous avons écrit `gcd(d, *m)`; à quoi sert le `*` in `*m` ? Ces deux détails sont complémentaires.

Jusqu'à présent, notre code n'a fonctionné que sur des valeurs simples comme des entiers qui tiennent dans des blocs de mémoire de taille fixe. Mais maintenant, nous sommes sur le point d'itérer sur un vecteur, qui pourrait être de n'importe quelle taille, peut-être très grand. Rust est prudent lorsqu'il gère de telles valeurs : il veut laisser le programmeur contrôler la consommation de mémoire, en indiquant clairement la durée de vie de chaque valeur, tout en s'assurant que la mémoire est libérée rapidement lorsqu'elle n'est plus nécessaire.

Ainsi, lorsque nous itérons, nous voulons dire à Rust que *la propriété* du vecteur doit rester avec `numbers` ; nous ne faisons *qu'emprunter* ses éléments pour la boucle. L' `&` opérateur en `&numbers[1..]` emprunte une

référenceaux éléments du vecteur à partir de la seconde. La `for` boucle itère sur les éléments référencés, laissant `m` emprunter chaque élément successivement. L'`*` opérateur dans `*m dereferences m`, donnant la valeur à laquelle il se réfère ; c'est la prochaine à laquelle `u64` nous voulons passer `gcd`. Enfin, puisque `numbers` possède le vecteur, Rust le libère automatiquement lorsqu'il `numbers` sort de la portée à la fin de `main`.

Les règles de Rust pour la propriété et les références sont essentielles à la gestion de la mémoire de Rust et à la concurrence sécurisée ; nous en discutons en détail dans le [chapitre 4](#) et son compagnon, le [chapitre 5](#). Vous devrez être à l'aise avec ces règles pour être à l'aise avec Rust, mais pour cette visite d'introduction, tout ce que vous devez savoir est qui `&x` emprunte une référence à `x`, et c'est `*r` la valeur à laquelle la référence fait référence `r`.

Continuons notre promenade à travers le programme :

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

Après avoir parcouru les éléments de `numbers`, le programme imprime les résultats dans le flux de sortie standard. La `println!` macro prend une chaîne de modèle, remplace les versions formatées des arguments restants pour les `{...}` formulaires tels qu'ils apparaissent dans la chaîne de modèle et écrit le résultat dans le flux de sortie standard.

Contrairement à C et C++, qui nécessitent `main` de renvoyer zéro si le programme s'est terminé avec succès, ou un état de sortie différent de zéro si quelque chose s'est mal passé, Rust suppose que s'il `main` revient du tout, le programme s'est terminé avec succès. Ce n'est qu'en appelant explicitement des fonctions comme `expect` ou `std::process::exit` que nous pouvons provoquer l'arrêt du programme avec un code d'état d'erreur.

La `cargo run` commande nous permet de passer des arguments à notre programme, afin que nous puissions essayer notre gestion de la ligne de commande :

```
$ parcours cargo 4256  
Compiling hello v0.1.0 (/home/jimb/rust/hello)  
Finished dev [unoptimized + debuginfo] target(s) in 0.22s  
    Running `~/home/jimb/rust/hello/target/debug/hello 42 56`  
The greatest common divisor of [42, 56] is 14  
$ parcours cargo 799459 2882327347  
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s  
    Running `~/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`  
The greatest common divisor of [799459, 28823, 27347] is 41  
$ parcours cargo 83  
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s  
    Running `~/home/jimb/rust/hello/target/debug/hello 83`  
The greatest common divisor of [83] is 83
```

```
$ parcours cargo
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
        Running `~/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

Nous avons utilisé quelques fonctionnalités de la bibliothèque standard de Rust dans cette section. Si vous êtes curieux de savoir ce qui est disponible, nous vous encourageons fortement à essayer la documentation en ligne de Rust. Il dispose d'une fonction de recherche en direct qui facilite l'exploration et inclut même des liens vers le code source. La `rustup` commande installe automatiquement une copie sur votre ordinateur lorsque vous installez Rust lui-même. Vous pouvez consulter la documentation de la bibliothèque standard sur le [site Web](#) de Rust ou dans votre navigateur avec la commande:

```
$doc rustup --std
```

Servir des pages sur le Web

Une des points forts de Rust est la collection de packages de bibliothèques disponibles gratuitement publiés sur le site Web [crates.io](#). La `cargo` commande permet à votre code d'utiliser facilement un package crates.io : il téléchargera la bonne version du package, le compilera et le mettra à jour comme demandé. Un paquet Rust, qu'il s'agisse d'une bibliothèque ou d'un exécutable, est appelé un *crate* ; Cargo et crates.io tirent tous deux leur nom de ce terme.

Pour montrer comment cela fonctionne, nous allons créer un serveur Web simple à l'aide du `actix-web` framework Webcrate, le `serde` crate de sérialisation et divers autres crates dont ils dépendent. Comme le montre la [figure 2-1](#), notre site Web demandera à l'utilisateur de saisir deux nombres et de calculer leur plus grand diviseur commun.

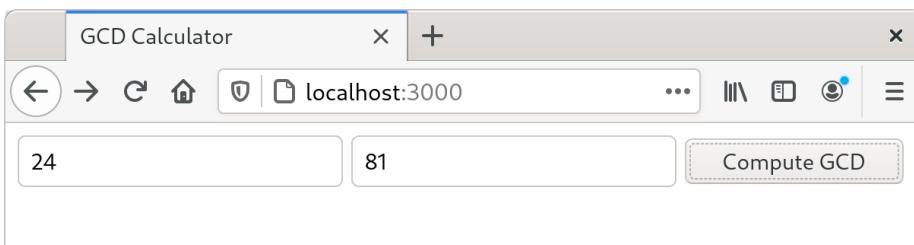


Illustration 2-1. Page Web proposant de calculer GCD

Tout d'abord, nous allons demander à Cargo de créer un nouveau package pour nous, nommé `actix-gcd` :

```
$cargo new actix-gcd
    Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

Ensuite, nous modifierons le fichier *Cargo.toml* de notre nouveau projet pour répertorier les packages que nous souhaitons utiliser. son contenu doit être le suivant :

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

Chaque ligne de la `[dependencies]` section de *Cargo.toml* donne le nom d'une caisse sur crates.io, et la version de cette caisse que nous aimeraisons utiliser. Dans ce cas, nous voulons la version `1.0.8` de la `actix-web` caisse et la version `1.0` de la `serde` caisse. Il peut bien y avoir des versions de ces caisses sur crates.io plus récentes que celles présentées ici, mais en nommant les versions spécifiques avec lesquelles nous avons testé ce code, nous pouvons nous assurer que le code continuera à se compiler même si de nouvelles versions des packages sont publiées. Nous aborderons la gestion des versions plus en détail au [chapitre 8](#).

Les caisses peuvent avoir des fonctionnalités facultatives : des parties de l'interface ou de l'implémentation dont tous les utilisateurs n'ont pas besoin, mais qu'il est néanmoins logique d'inclure dans cette caisse. La `serde` caisse offre une manière merveilleusement concise de gérer les données des formulaires Web, mais selon `serde` la documentation de , elle n'est disponible que si nous sélectionnons la `derive` fonctionnalité de la caisse, nous l'avons donc demandée dans notre fichier *Cargo.toml* comme indiqué.

Notez que nous n'avons qu'à nommer les caisses que nous utiliserons directement ; `cargo` se charge d'apporter les autres caisses dont ceux-ci ont besoin à leur tour.

Pour notre première itération, nous garderons le serveur Web simple : il ne servira que la page qui invite l'utilisateur à entrer des nombres avec lesquels calculer. Dans `actix-gcd/src/main.rs` , nous placerons le texte suivant :

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
```

```

    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                    <input type="text" name="n"/>
                    <input type="text" name="m"/>
                    <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}

```

Nous commençons par une `use` déclaration pour faciliter l'accès à certaines des `actix-web` définitions de la caisse. Lorsque nous écrivons `use actix_web::{ ... }`, chacun des noms listés à l'intérieur des accolades devient directement utilisable dans notre code ; au lieu d'avoir à épeler le nom complet à `actix_web::HttpResponse` chaque fois que nous l'utilisons, nous pouvons simplement nous y référer comme `HttpResponse`. (Nous arriverons à la `serde` caisse dans un instant.)

Notre `main` fonction est simple : il appelle `HttpServer::new` à créer un serveur qui répond aux requêtes d'un seul chemin, `"/"` ; imprime un message nous rappelant comment s'y connecter ; puis le met à l'écoute sur le port TCP 3000 sur la machine locale.

L'argument auquel nous passons est l' expression de `fermeture HttpServer::new Rust || { App::new() ... }`. Une fermeture est une valeur qui peut être appelée comme s'il s'agissait d'une fonction. Cette fermeture ne prend aucun argument, mais si c'était le cas, leurs noms apparaîtraient entre les `||` barres verticales. C'est `{ ... }` le corps de la fermeture. Lorsque nous démarrons notre serveur, Actix démarre un pool de threads pour gérer les requêtes entrantes. Chaque thread appelle notre fermeture pour obtenir une nouvelle copie de la `App` valeur qui lui indique comment acheminer et gérer les requêtes.

La fermeture appelle `App::new` à créer un nouveau vide `App` puis appelle sa `route` méthode pour ajouter une seule route pour le chemin `"/"`. Le gestionnaire fourni pour cette route, `web::get().to(get_index)`, traite les requêtes HTTP GET en appelant la fonction `get_index`. La `route` méthode renvoie la même `App` que

celle sur laquelle elle a été invoquée, maintenant améliorée avec la nouvelle route. Puisqu'il n'y a pas de point-virgule à la fin du corps de la fermeture, `App` est la valeur de retour de la fermeture, prête `HttpServer` à être utilisée par le thread.

La `get_index` fonction construit une `HttpResponse` valeur représentant la réponse à une `GET` / requête HTTP. `HttpResponse::Ok()` représente un statut HTTP `200 OK`, indiquant que la requête a réussi. Nous appelons ses méthodes `content_type` et `body` pour remplir les détails de la réponse ; chaque appel renvoie le `HttpResponse` auquel il a été appliqué, avec les modifications apportées. Enfin, la valeur de retour de `body` sert de valeur de retour de `get_index`.

Étant donné que le texte de la réponse contient de nombreux guillemets doubles, nous l'écrivons en utilisant la syntaxe "chaîne brute" de Rust : la lettre `r`, zéro ou plusieurs marques de hachage (c'est-à-dire le `#` caractère), un guillemet double, puis le contenu de la chaîne, terminé par un autre guillemet double suivi du même nombre de marques de hachage. Tout caractère peut apparaître dans une chaîne brute sans être échappé, y compris les guillemets doubles ; en fait, aucune séquence d'échappement comme `\"` n'est reconnue. Nous pouvons toujours nous assurer que la chaîne se termine là où nous le souhaitons en utilisant plus de marques de hachage autour des guillemets que jamais dans le texte.

Après avoir écrit `main.rs`, nous pouvons utiliser la `cargo run` commande pour faire tout ce qui est nécessaire pour le faire fonctionner : récupérer les caisses nécessaires, les compiler, créer notre propre programme, relier le tout et le démarrer :

```
$ course de fret
    Updating crates.io index
    Downloading crates ...
        Downloaded serde v1.0.100
        Downloaded actix-web v1.0.8
        Downloaded serde_derive v1.0.100
    ...
        Compiling serde_json v1.0.40
        Compiling actix-router v0.1.5
        Compiling actix-http v0.2.10
        Compiling awc v0.2.7
        Compiling actix-web v1.0.8
        Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
        Running `~/home/jimb/rust/actix-gcd/target/debug/actix-gcd`
    Serving on http://localhost:3000...
```

À ce stade, nous pouvons visiter l'URL donnée dans notre navigateur et voir la page illustrée précédemment à la [figure 2-1](#).

Malheureusement, cliquer sur Compute GCD ne fait rien, à part naviguer dans notre navigateur vers une page vierge. Corrigeons cela ensuite, en ajoutant une autre route à notre App pour gérer la POST demande de notre formulaire.

Il est enfin temps d'utiliser la `serde` caisse que nous avons répertoriée dans notre fichier `Cargo.toml` : elle fournit un outil pratique qui nous aidera à traiter les données du formulaire. Tout d'abord, nous devrons ajouter la `use` directive suivante en haut de `src/main.rs` :

```
use serde::Deserialize;
```

Les programmeurs Rust rassemblent généralement toutes leurs `use` déclarations vers le haut du fichier, mais ce n'est pas strictement nécessaire : Rust permet aux déclarations de se produire dans n'importe quel ordre, tant qu'elles apparaissent au niveau d'imbrication approprié.

Ensuite, définissons un type de structure Rust qui représente les valeurs que nous attendons de notre formulaire :

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

Cela définit un nouveau type nommé `GcdParameters` qui a deux champs, `n` et `m`, dont chacun est un `u64` type d'argument que notre `gcd` fonction attend.

L'annotation au-dessus de la `struct` définition est un attribut, comme l'`#[test]` attribut que nous avons utilisé précédemment pour marquer les fonctions de test. Placer un `#[derive(Deserialize)]` attribut au-dessus d'une définition de type indique au `serde` crate d'examiner le type lorsque le programme est compilé et de générer automatiquement du code pour analyser une valeur de ce type à partir de données au format utilisé par les formulaires HTML pour les `POST` requêtes. En fait, cet attribut est suffisant pour vous permettre d'analyser une `GcdParameters` valeur à partir de presque n'importe quel type de données structurées : JSON, YAML, TOML ou l'un des nombreux autres formats textuels et binaires. La `serde` caisse fournit également un `Serialize` attribut qui génère du code pour faire l'inverse, en prenant les valeurs Rust et en les écrivant dans un format structuré.

Avec cette définition en place, nous pouvons écrire notre fonction de gestionnaire assez facilement :

```

fn post_gcd(form: web:: Form<GcdParameters>) -> HttpResponse {
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Computing the GCD with zero is boring.");
    }

    let response =
        format!("The greatest common divisor of the numbers {} and {} \
                is <b>{}</b>\n",
                form.n, form.m, gcd(form.n, form.m));

    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}

```

Pour qu'une fonction serve de gestionnaire de requêtes Actix, ses arguments doivent tous avoir des types qu'Actix sait extraire d'une requête HTTP. Notre `post_gcd` fonction prend un argument, `form`, dont le type est `web::Form<GcdParameters>`. Actix sait extraire une valeur de n'importe quel type `web::Form<T>` d'une requête HTTP si, et seulement si, `T` peut être déserialisée à partir de données de formulaire HTML POST. Puisque nous avons placé l'`#[derive(Deserialize)]` attribut sur notre `GcdParameters` définition de type, Actix peut le déserialiser à partir des données de formulaire, de sorte que les gestionnaires de requêtes peuvent attendre une `web::Form<GcdParameters>` valeur en tant que paramètre. Ces relations entre les types et les fonctions sont toutes élaborées au moment de la compilation ; si vous écrivez une fonction de gestionnaire avec un type d'argument qu'Actix ne sait pas gérer, le compilateur Rust vous informe immédiatement de votre erreur.

En regardant à l'intérieur `post_gcd` de , la fonction renvoie d'abord une 400 BAD REQUEST erreur HTTP si l'un des paramètres est égal à zéro, car notre `gcd` fonction paniquera s'ils le sont. Ensuite, il construit une réponse à la requête à l'aide de la `format!` macro. La `format!` macro est comme la `println!` macro, sauf qu'au lieu d'écrire le texte sur la sortie standard, il le renvoie sous forme de chaîne. Une fois qu'il a obtenu le texte de la réponse, `post_gcd` il l'encapsule dans une réponse HTTP 200 OK, définit son type de contenu et le renvoie pour qu'il soit livré à l'expéditeur.

Nous devons également nous inscrire `post_gcd` en tant que gestionnaire du formulaire. Nous allons remplacer notre `main` fonction par cette version :

```

fn main() {
    let server = HttpServer:: new(|| {
        App:: new()
            .route("/", web:: get().to(get_index))

```

```

        .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}

```

Le seul changement ici est que nous avons ajouté un autre appel à `route`, établissant `web::post().to(post_gcd)` comme gestionnaire pour le chemin `"/gcd"`.

La dernière pièce restante est la `gcd` fonction nous avons écrit plus tôt, qui va dans le fichier `actix-gcd/src/main.rs`. Avec cela en place, vous pouvez interrompre tous les serveurs que vous pourriez avoir laissés en cours d'exécution et reconstruire et redémarrer le programme :

```

$course de fret
Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/actix-gcd`
Serving on http://localhost:3000...

```

Cette fois, en visitant `http://localhost:3000`, en saisissant quelques chiffres et en cliquant sur le bouton Compute GCD, vous devriez voir des résultats([Figure 2-2](#)).

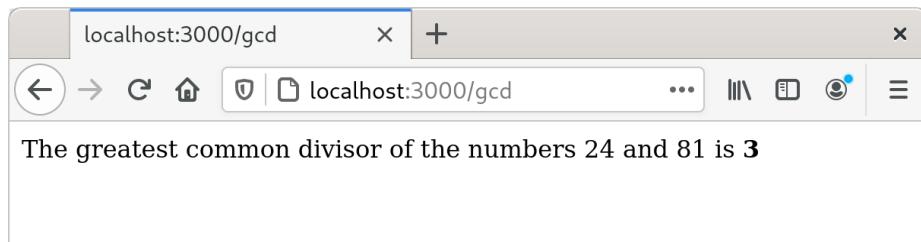


Illustration 2-2. Page Web montrant les résultats du calcul de GCD

Concurrence

UneL'une des grandes forces de Rust est sa prise en charge de la programmation simultanée. Les mêmes règles qui garantissent que les programmes Rust sont exempts d'erreurs de mémoire garantissent également que les threads ne peuvent partager la mémoire que de manière à éviter les courses de données.. Par exemple:

- Si vous utilisez un mutex pour coordonner les threads apportant des modifications à une structure de données partagée, Rust garantit que vous ne pouvez accéder aux données que lorsque vous maintenez le verrou et libère le verrou automatiquement lorsque vous avez termi-

né. En C et C++, la relation entre un mutex et les données qu'il protège est laissée aux commentaires.

- Si vous souhaitez partager des données en lecture seule entre plusieurs threads, Rust garantit que vous ne pouvez pas modifier les données accidentellement. En C et C++, le système de types peut aider à cela, mais il est facile de se tromper.
- Si vous transférez la propriété d'une structure de données d'un thread à un autre, Rust s'assure que vous avez bien renoncé à tout accès à celle-ci. En C et C++, c'est à vous de vérifier que rien sur le thread émetteur ne touchera plus jamais les données. Si vous ne le faites pas correctement, les effets peuvent dépendre de ce qui se trouve dans le cache du processeur et du nombre d'écritures en mémoire que vous avez effectuées récemment. Non pas que nous soyons amers.

Dans cette section, nous vous guiderons tout au long du processus d'écriture de votre deuxième programme multithread.

Vous avez déjà écrit votre premier: le framework Web Actix que vous avez utilisé pour implémenter le serveur Greatest Common Divisor utilise un pool de threads pour exécuter les fonctions du gestionnaire de requêtes. Si le serveur reçoit des requêtes simultanées, il peut exécuter les fonctions `get_form` et `post_gcd` dans plusieurs fils à la fois. Cela peut être un peu choquant, car nous n'avions certainement pas à l'esprit la concurrence lorsque nous avons écrit ces fonctions. Mais Rust garantit que cela peut être fait en toute sécurité, quelle que soit la complexité de votre serveur : si votre programme compile, il est exempt de courses de données. Toutes les fonctions Rust sont thread-safe.

Le programme de cette section trace le Mandelbrot ensemble, une fractale produite en itérant une fonction simple sur des nombres complexes. Tracer l'ensemble de Mandelbrot est souvent appelé un algorithme *parallèle embarrassant*, parce que le modèle de communication entre les threads est si simple ; nous couvrirons des modèles plus complexes au [chapitre 19](#), mais cette tâche démontre certains des éléments essentiels.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo new mandelbrot
     Created binary (application) `mandelbrot` package
$ cd mandelbrot
```

Tout le code ira dans `mandelbrot/src/main.rs`, et nous ajouterons quelques dépendances à `mandelbrot/Cargo.toml`.

Avant d'aborder l'implémentation Mandelbrot simultanée, nous devons décrire le calcul que nous allons effectuer.

Qu'est-ce que l'ensemble de Mandelbrot est réellement

Lors de la lecture de code, il est utile d'avoir une idée concrète de ce qu'il essaie de faire, alors faisons une petite excursion dans les mathématiques pures. Nous commencerons par un cas simple, puis ajouterons des détails compliqués jusqu'à ce que nous arrivions au calcul au cœur de l'ensemble de Mandelbrot.

Voici une boucle infinie, écrite en utilisant la syntaxe dédiée de Rust pour cela, une `loop` déclaration :

```
fn square_loop(mut x:f64) {  
    loop {  
        x = x * x;  
    }  
}
```

Dans la vraie vie, Rust peut voir qu'il `x` n'est jamais utilisé pour quoi que ce soit et ne prend donc pas la peine de calculer sa valeur. Mais pour le moment, supposons que le code s'exécute tel qu'il est écrit. Que devient la valeur de `x`? Mettre au carré tout nombre inférieur à 1 le rend plus petit, donc il se rapproche de zéro ; élire au carré 1 donne 1; mettre au carré un nombre plus grand que 1 le rend plus grand, donc il se rapproche de l'infini ; et la mise au carré d'un nombre négatif le rend positif, après quoi il se comporte comme l'un des cas précédents ([Figure 2-3](#)).

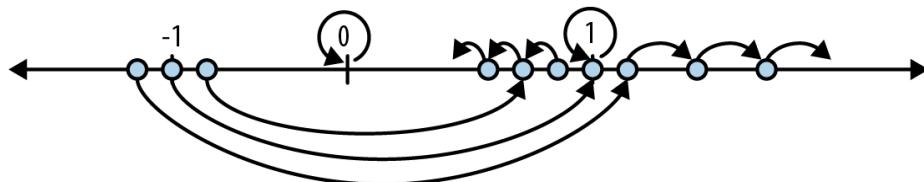


Illustration 2-3. Effets de la quadrature répétée d'un nombre

Ainsi, selon la valeur que vous transmettez à `square_loop`, `x` reste à zéro ou à un, s'approche de zéro ou s'approche de l'infini.

Considérons maintenant une boucle légèrement différente :

```
fn square_add_loop(c:f64) {  
    let mut x = 0.;  
    loop {  
        x = x * x + c;  
    }  
}
```

Cette fois, `x` commence à zéro, et nous ajustons sa progression à chaque itération en ajoutant `c` après l'avoir quadrillé. Cela rend plus difficile de voir comment `x` les tarifs, mais certaines expérimentations montrent que

si `c` est supérieur à 0,25 ou inférieur à -2,0, `x` devient finalement infiniment grand; sinon, il reste quelque part dans le voisinage de zéro.

L'astuce suivante : au lieu d'utiliser des `f64` valeurs, considérez la même boucle en utilisant des nombres complexes. La `num` caisse sur crates.io fournit un type de nombre complexe que nous pouvons utiliser, nous devons donc ajouter une ligne pour `num` la `[dependencies]` section dans le fichier `Cargo.toml` de notre programme. Voici le fichier complet, jusqu'à ce point (nous en ajouterons plus tard) :

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
num = "0.4"
```

Nous pouvons maintenant écrire l'avant-dernière version de notre boucle :

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```

Il est traditionnel de l'utiliser `z` pour les nombres complexes, nous avons donc renommé notre variable de bouclage. L'expression `Complex { re: 0.0, im: 0.0 }` est la façon dont nous écrivons le zéro complexe en utilisant le type `num` de la caisse `Complex`. `Complex` est une structure Rusttype (ou *struct*), défini comme ceci :

```
struct Complex<T> {
    /// Real portion of the complex number
    re:T,
    /// Imaginary portion of the complex number
    im:T,
}
```

Le code précédent définit une structure nommée `Complex`, avec deux champs, `re` et `im`. `Complex` est une structure *générique* : vous pouvez lire le `<T>` après le nom du type comme "pour tout type `T`". Par exemple,

`Complex<f64>` est un nombre complexe dont les champs `re` et `im` sont des valeurs, utiliseraient des flottants 32 bits, etc. Compte tenu de cette définition, une expression comme produit une valeur avec son champ initialisé à 0,24 et son champ initialisé à 0,3.

```
im f64 Complex<f32> Complex {  
    re: 0.24, im: 0.3 } Complex re im
```

La `num` caisse s'arrange pour que `*`, `+` et d'autres opérateurs arithmétiques fonctionnent sur des `Complex` valeurs, de sorte que le reste de la fonction fonctionne exactement comme la version précédente, sauf qu'elle opère sur des points sur le plan complexe, pas seulement sur des points le long de la ligne des nombres réels. Nous expliquerons comment vous pouvez faire fonctionner les opérateurs de Rust avec vos propres types au [chapitre 12](#).

Enfin, nous avons atteint la destination de notre excursion en mathématiques pures. L'ensemble de Mandelbrot est défini comme l'ensemble des nombres complexes `c` pour lesquels `z` ne s'envole pas vers l'infini. Notre boucle de mise au carré simple d'origine était suffisamment prévisible : tout nombre supérieur à 1 ou inférieur à -1 s'envole. Lancer un `+` `c` dans chaque itération rend le comportement un peu plus difficile à anticiper : comme nous l'avons dit plus tôt, les valeurs `c` supérieures à 0,25 ou inférieures à -2 font `z` s'envoler. Mais étendre le jeu à des nombres complexes produit des modèles vraiment bizarres et beaux, ce que nous voulons tracer.

Puisqu'un nombre complexe `c` a à la fois des composantes réelles et imaginaires `c.re` et `c.im`, nous les traiterons comme les coordonnées `x` et `y` d'un point sur le plan cartésien, et colorerons le point en noir s'il `c` se trouve dans l'ensemble de Mandelbrot, ou en une couleur plus claire sinon. Ainsi, pour chaque pixel de notre image, nous devons exécuter la boucle précédente sur le point correspondant du plan complexe, voir s'il s'échappe à l'infini ou orbite autour de l'origine pour toujours, et le colorer en conséquence.

La boucle infinie prend un certain temps à s'exécuter, mais il existe deux astuces pour les impatients. Premièrement, si nous renonçons à exécuter la boucle pour toujours et essayons simplement un nombre limité d'itérations, il s'avère que nous obtenons toujours une approximation décente de l'ensemble. Le nombre d'itérations dont nous avons besoin dépend de la précision avec laquelle nous voulons tracer la frontière. Deuxièmement, il a été démontré que, si `z` jamais une fois quitte le cercle de rayon 2 centré à l'origine, il s'envolera définitivement infiniment loin de l'origine. Voici donc la version finale de notre boucle, et le cœur de notre programme :

```
use num::Complex;  
  
/// Try to determine if `c` is in the Mandelbrot set, using at most `limit`  
/// iterations to decide.
```

```

/// If `c` is not a member, return `Some(i)` , where `i` is the number of
/// iterations it took for `c` to leave the circle of radius 2 centered on the
/// origin. If `c` seems to be a member (more precisely, if we reached the
/// iteration limit without being able to prove that `c` is not a member),
/// return `None`.
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}

```

Cette fonction prend le nombre complexe `c` que nous voulons tester pour l'appartenance à l'ensemble de Mandelbrot et une limite sur le nombre d'itérations à essayer avant d'abandonner et de déclarer `c` être probablement membre.

La valeur de retour de la fonction est un `Option<usize>`. La bibliothèque standard de Rust définit le `Option` type comme suit :

```

enum Option<T> {
    None,
    Some(T),
}

```

`Option` est un *type énuméré*, souvent appelé *enum*, car sa définition énumère plusieurs variantes qu'une valeur de ce type pourrait être : pour tout type `T`, une valeur de type `Option<T>` est soit `Some(v)`, où `v` est une valeur de type `T`, soit `None`, indiquant qu'aucune `T` valeur n'est disponible. Comme le `Complex` type dont nous avons parlé précédemment, `Option` est un type générique : vous pouvez l'utiliser `Option<T>` pour représenter une valeur facultative de n'importe quel type `T` que vous aimez.

Dans notre cas, `escape_time` renvoie un `Option<usize>` pour indiquer si `c` est dans l'ensemble de Mandelbrot - et si ce n'est pas le cas, combien de temps nous avons dû itérer pour le découvrir. Si `c` n'est pas dans l'ensemble, `escape_time` renvoie `Some(i)`, où `i` est le numéro de l'itération à laquelle `z` est sorti le cercle de rayon 2. Sinon, `c` est apparemment dans l'ensemble, et `escape_time` renvoie `None`.

```

for i in 0..limit {

```

Les exemples précédents montraient des `for` boucles itérant sur des arguments de ligne de commande et des éléments vectoriels ; cette `for` boucle itère simplement sur la plage d'entiers commençant par `0` et jusqu'à (mais non compris) `limit`.

L'`z.norm_sqr()` appel de méthode renvoie le carré de `z` la distance de à l'origine. Pour décider si `z` a quitté le cercle de rayon 2, au lieu de calculer une racine carrée, nous comparons simplement la distance au carré avec 4,0, ce qui est plus rapide.

Vous avez peut-être remarqué que nous utilisons `///` pour marquer les lignes de commentaire au-dessus de la définition de la fonction ; les commentaires au dessus des membres de la `Complex` structure commencent `///` aussi par. Ce sont *des commentaires de documentation* ; l'`rustdoc` utilitaire sait comment les analyser, ainsi que le code qu'ils décrivent, et produire une documentation en ligne. La documentation de la bibliothèque standard de Rust est écrite sous cette forme. Nous décrivons en détail les commentaires de la documentation au [chapitre 8](#).

Le reste du programme consiste à décider quelle partie de l'ensemble tracer à quelle résolution et à répartir le travail sur plusieurs threads pour accélérer le calcul ..

Analyse des arguments de ligne de commande de la paire

Le programme prend plusieurs arguments en ligne de commande pour contrôler la résolution de l'image que nous allons écrire et la partie de l'ensemble de Mandelbrot que l'image montre. Étant donné que ces arguments de ligne de commande suivent tous une forme commune, voici une fonction pour les analyser :

```
use std::str::FromStr;

/// Parse the string `s` as a coordinate pair, like `400x600` or `1.0,0.5`.
///
/// Specifically, `s` should have the form <left><sep><right>, where <sep> is
/// the character given by the `separator` argument, and <left> and <right> are
/// both strings that can be parsed by `T::from_str`. `separator` must be an
/// ASCII character.
///
/// If `s` has the proper form, return `Some<(x, y)>`. If it doesn't parse
/// correctly, return `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}
```

```

        }
    }

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",      ','), None);
    assert_eq!(parse_pair::<i32>("10,",    ','), None);
    assert_eq!(parse_pair::<i32>(",10",   ','), None);
    assert_eq!(parse_pair::<i32>("10,20", ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",   'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}

```

La définition de `parse_pair` est une *fonction générique*:

```
fn parse_pair<T: FromStr>(s: &str, separator: char) ->Option<(T, T)> {
```

Vous pouvez lire la clause `<T: FromStr>` à haute voix comme suit :

"Pour tout type `T` qui implémente le `FromStr` trait...". Cela nous permet effectivement de définir une famille entière de fonctions à la fois :

`parse_pair::<i32>` est une fonction qui analyse des paires de `i32` valeurs, `parse_pair::<f64>` analyse des paires de valeurs à virgule flottante, etc. Cela ressemble beaucoup à un modèle de fonction en C++. Un programmeur Rust appellerait `T` un *paramètre de type* de `parse_pair`. Lorsque vous utilisez une fonction générique, Rust sera souvent capable de déduire les paramètres de type pour vous, et vous n'aurez pas besoin de les écrire comme nous l'avons fait dans le code de test.

Notre type de retour est `Option<(T, T)>` : soit `None` ou une valeur `Some((v1, v2))`, où `(v1, v2)` est un tuple de deux valeurs, toutes deux de type `T`. La `parse_pair` fonction n'utilise pas d'instruction de retour explicite, donc sa valeur de retour est la valeur de la dernière (et la seule) expression de son corps :

```

match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}

```

Le `String` genre `find` méthode recherche dans la chaîne un caractère qui correspond à `separator`. Si `find` renvoie `None`, ce qui signifie que le caractère séparateur n'apparaît pas dans la chaîne, l'`match` expression entière est évalué à `None`, indiquant que l'analyse a échoué. Sinon, nous prenons `index` la position du séparateur dans la chaîne.

```

match (T:: from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}

```

Cela commence à montrer la puissance de l' `match` expression. L'argument de la correspondance est cette expression de tuple :

```
(T:: from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

Les expressions `&s[..index]` et `&s[index + 1..]` sont des tranches de la chaîne, précédent et suivant le séparateur. La fonction associée au paramètre de type `from_str` prend chacun d'entre eux et essaie de les analyser comme une valeur de type `T`, produisant un tuple de résultats. Voici ce contre quoi nous nous comparons :

```
(Ok(l), Ok(r)) => Some((l, r)),
```

Ce modèle correspond uniquement si les deux éléments du tuple sont `Ok` des variantes du `Result` type, indiquant que les deux analyses ont réussi. Si tel est le cas, `Some((l, r))` est la valeur de l'expression de correspondance et donc la valeur de retour de la fonction.

```
_ => None
```

Le modèle de caractère générique `_` correspond à n'importe quoi et ignore sa valeur. Si nous atteignons ce point, alors `parse_pair` a échoué, nous évaluons donc à `None`, en fournissant à nouveau la valeur de retour de la fonction.

Maintenant que nous avons `parse_pair`, il est facile d'écrire une fonction pour analyser une paire de coordonnées en virgule flottante et les renvoyer sous forme de `Complex<f64>` valeur:

```

/// Parse a pair of floating-point numbers separated by a comma as a complex
/// number.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im:-0.0625 }));
    assert_eq!(parse_complex(",,-0.0625"), None);
}

```

La `parse_complex` fonction appelle `parse_pair`, construit une `Complex` valeur si les coordonnées ont été analysées avec succès et transmet les échecs à son appelant.

Si vous lisiez attentivement, vous avez peut-être remarqué que nous avons utilisé une notation abrégée pour construire la `Complex` valeur. Il est courant d'initialiser les champs d'une structure avec des variables du même nom, donc plutôt que de vous forcer à écrire `Complex { re: re, im: im }`, Rust vous permet simplement d'écrire `Complex { re, im }`. Ceci est modélisé sur des notations similaires en JavaScript et Haskell.

Mappage des pixels aux nombres complexes

Le programme doit travailler dans deux espaces de coordonnées liés : chaque pixel de l'image de sortie correspond à un point sur le plan complexe. La relation entre ces deux espaces dépend de la portion du Mandelbrotset que nous allons tracer, et la résolution de l'image demandée, telle que déterminée par les arguments de la ligne de commande. La fonction suivante convertit à partir de l'*espace image* à l'*espace des nombres complexes* :

```
/// Given the row and column of a pixel in the output image, return the
/// corresponding point on the complex plane.
///
/// `bounds` is a pair giving the width and height of the image in pixels.
/// `pixel` is a (column, row) pair indicating a particular pixel in that image
/// The `upper_left` and `lower_right` parameters are points on the complex
/// plane designating the area our image covers.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                          upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // Why subtraction here? pixel.1 increases as we go down,
        // but the imaginary component increases as we go up.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 200), (25, 175),
                             Complex { re: -1.0, im: 1.0 },
                             Complex { re: 1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.75 });
}
```

[La figure 2-4](#) illustre le calcul effectué `pixel_to_point`.

Le code de `pixel_to_point` est simplement un calcul, nous ne l'expliquerons donc pas en détail. Cependant, il y a quelques points à souligner. Les expressions de cette forme font référence à des éléments de tuple :

```
pixel.0
```

Cela fait référence au premier élément du tuple `pixel`.

```
pixel.0 as f64
```

C'est la syntaxe de Rust pour une conversion de type : cela convertit `pixel.0` en une `f64` valeur. Contrairement à C et C++, Rust refuse généralement de convertir implicitement les types numériques ; vous devez écrire les conversions dont vous avez besoin. Cela peut être fastidieux, mais être explicite sur les conversions qui se produisent et quand est étonnamment utile. Les conversions implicites d'entiers semblent assez innocentes, mais historiquement, elles ont été une source fréquente de bogues et de failles de sécurité dans le code C et C++ du monde réel.

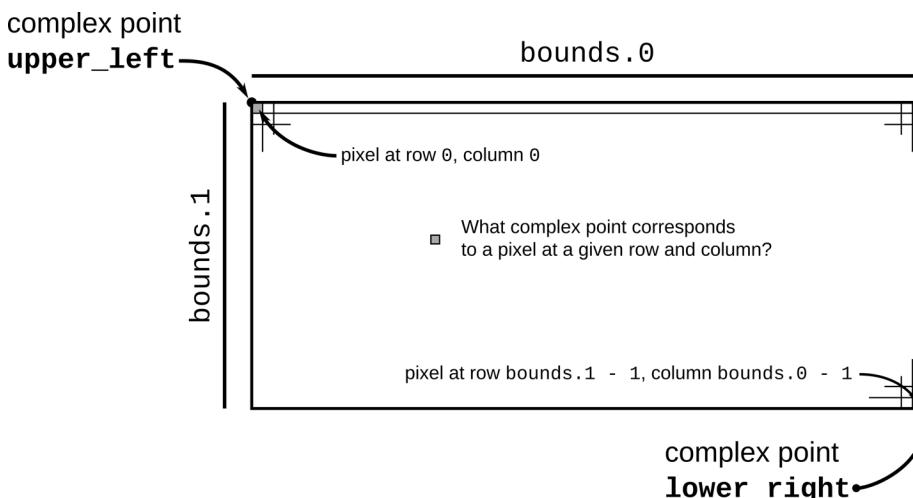


Illustration 2-4. La relation entre le plan complexe et les pixels de l'image

Tracer l'ensemble

Comploter l'ensemble de Mandelbrot, pour chaque pixel de l'image, on applique simplement `escape_time` au point correspondant sur le plan complexe, et on colore le pixel en fonction du résultat :

```
/// Render a rectangle of the Mandelbrot set into a buffer of pixels.  
///  
/// The `bounds` argument gives the width and height of the buffer `pixels`,  
/// which holds one grayscale pixel per byte. The `upper_left` and `lower_right`  
/// arguments specify points on the complex plane corresponding to the upper-  
/// left and lower-right corners of the pixel buffer.  
fn render(pixels: &mut [u8],  
          bounds: (usize, usize),
```

```

        upper_left: Complex<f64>,
        lower_right:Complex<f64>)
    }

    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                         upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}

```

Tout cela devrait vous sembler assez familier à ce stade.

```

pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };

```

Si `escape_time` dit que `point` appartient à l'ensemble, `render` colore le pixel correspondant en noir (0). Sinon, `render` attribue des couleurs plus foncées aux nombres qui ont mis plus de temps à sortir du cercle.

Écriture de fichiers image

La `image` caisse fournit des fonctions pour lire et écrire une grande variété de formats d'image, ainsi que certaines fonctions de manipulation d'image de base. En particulier, il comprend un encodeur pour le format de fichier image PNG, que ce programme utilise pour enregistrer les résultats finaux du calcul. Pour utiliser `image`, ajoutez la ligne suivante à la `[dependencies]` section de `Cargo.toml` :

```
image = "0.13.0"
```

Avec cela en place, nous pouvons écrire:

```

use image:: ColorType;
use image:: png:: PNGEncoder;
use std:: fs::File;

/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to the
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std:: io:: Error>

```

```

    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(pixels,
                   bounds.0 as u32, bounds.1 as u32,
                   ColorType::Gray(8))?;

    Ok(())
}

```

Le fonctionnement de cette fonction est assez simple : elle ouvre un fichier et essaie d'y écrire l'image. Nous transmettons à l'encodeur les données de pixel réelles de `pixels`, ainsi que sa largeur et sa hauteur de `bounds`, puis un argument final qui indique comment interpréter les octets dans `pixels` : la valeur `ColorType::Gray(8)` indique que chaque octet est une valeur en niveaux de gris de huit bits.

C'est tout simple. Ce qui est intéressant avec cette fonction, c'est la façon dont elle réagit quand quelque chose ne va pas. Si nous rencontrons une erreur, nous devons la signaler à notre appelant. Comme nous l'avons mentionné précédemment, les fonctions faillibles de Rust doivent renvoyer une `Result` valeur, qui est soit `Ok(s)` en cas de succès, où `s` est la valeur réussie, soit `Err(e)` en cas d'échec, où `e` est un code d'erreur. Quels sont donc `write_image` les types de réussite et d'erreur de ?

Quand tout va bien, notre `write_image` fonctionn'a aucune valeur utile à renvoyer ; il a écrit tout ce qui est intéressant dans le fichier. Son type de réussite est donc le type *d'unité* `()`, ainsi appelé car il n'a qu'une seule valeur, également écrite `()`. Le type d'unité est similaire à `void` en C et C++.

Lorsqu'une erreur se produit, c'est soit parce qu'il `File::create` n'a pas pu créer le fichier, soit parce qu'il `encoder.encode` n'a pas pu y écrire l'image ; l'opération d'E/S a renvoyé un code d'erreur. Le type de retour de `File::create` est `Result<std::fs::File, std::io::Error>`, tandis que celui de `encoder.encode` est `Result<(), std::io::Error>`, donc les deux partagent le même type d'erreur, `std::io::Error`. Il est logique que notre `write_image` fonction fasse de même. Dans les deux cas, l'échec doit entraîner un retour immédiat, en transmettant la `std::io::Error` valeur décrivant ce qui s'est mal passé.

Donc, pour gérer correctement `File::create` le résultat de , nous devons utiliser `match` sa valeur de retour, comme ceci :

```

let output = match File::create(filename) {
    Ok(f) => f,
    Err(e) => {
        return Err(e);
}

```

```
    }  
};
```

En cas de succès, laissez `output` être le `File` porté dans la `ok` valeur. En cas d'échec, transmettre l'erreur à notre propre appelant.

Ce type d'`match` instruction est un modèle si courant dans Rust que le langage fournit l'`? opérateur comme raccourci pour l'ensemble. Ainsi, plutôt que d'écrire explicitement cette logique chaque fois que nous tentons quelque chose qui pourrait échouer, vous pouvez utiliser l'énoncé équivalent suivant et beaucoup plus lisible :`

```
let output = File::create(filename)?;
```

En cas d'`File::create` échec, l'`? opérateur revient de write_image, transmettant l'erreur. Sinon, output contient le fichier File.`

NOTER

C'est une erreur courante de débutant d'essayer d'utiliser `? la main fonction. Cependant, puisque main lui-même ne renvoie pas de valeur, cela ne fonctionnera pas ; à la place, vous devez utiliser une match instruction, ou l'une des méthodes abrégées comme unwrap et expect. Il y a aussi la possibilité de simplement changer main pour retourner un Result, que nous aborderons plus tard.`

Un programme Mandelbrot simultané

Toutes les pièces sont en place, et nous pouvons vous montrer la `main` fonction, où nous pouvons mettre la simultanéité à notre service. Tout d'abord, une version non concurrente pour plus de simplicité :

```
use std::env;  
  
fn main() {  
    let args: Vec<String> = env::args().collect();  
  
    if args.len() != 5 {  
        eprintln!("Usage: {} FILE PIXELS UPPERLEFT LOWERRIGHT",  
                 args[0]);  
        eprintln!("Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",  
                 args[0]);  
        std::process::exit(1);  
    }  
  
    let bounds = parse_pair(&args[2], 'x')  
        .expect("error parsing image dimensions");  
    let upper_left = parse_complex(&args[3])  
        .expect("error parsing upper left corner point");  
    let lower_right = parse_complex(&args[4])  
        .expect("error parsing lower right corner point");
```

```

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

render(&mut pixels, bounds, upper_left, lower_right);

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

```

Après avoir collecté les arguments de ligne de commande dans un vecteur de `String`s, nous analysons chacun d'eux, puis commençons les calculs.

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

Un appel de macro `vec![v; n]` crée un vecteur `n` éléments long dont les éléments sont initialisés à `v`, donc le code précédent crée un vecteur de zéros dont la longueur est `bounds.0 * bounds.1`, où `bounds` est la résolution de l'image analysée à partir de la ligne de commande. Nous utiliserons ce vecteur comme un tableau rectangulaire de valeurs de pixels en niveaux de gris d'un octet, comme illustré à la [Figure 2-5](#).

La ligne d'intérêt suivante est celle-ci :

```
render(&mut pixels, bounds, upper_left, lower_right);
```

Cela appelle la `render` fonction pour calculer réellement l'image. L'expression `&mut pixels` emprunte une référence mutable à notre tampon de pixels, permettant `render` de le remplir avec des valeurs de niveaux de gris calculées, même s'il `pixels` reste le propriétaire du vecteur. Les arguments restants transmettent les dimensions de l'image et le rectangle du plan complexe que nous avons choisi de tracer.

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

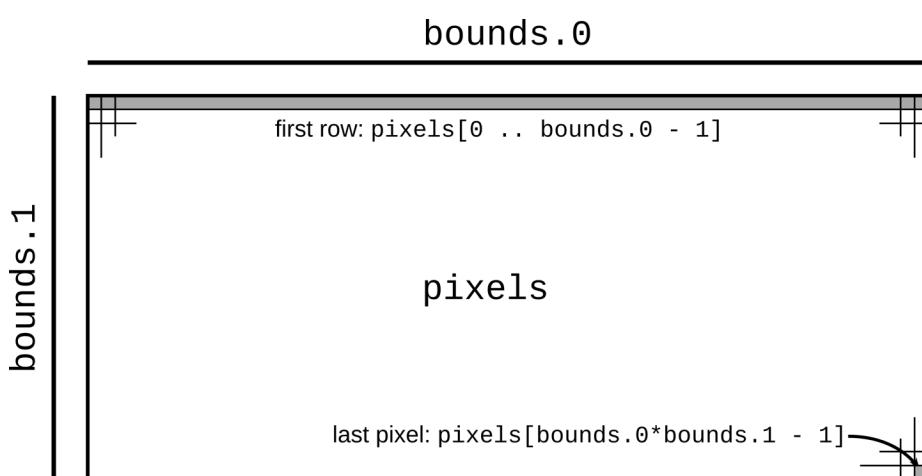


Illustration 2-5. Utilisation d'un vecteur comme tableau rectangulaire de pixels

Enfin, nous écrivons le tampon de pixels sur le disque sous forme de fichier PNG. Dans ce cas, nous passons une référence partagée (non modifiable) au tampon, car `write_image` nous n'avons pas besoin de modifier le contenu du tampon.

À ce stade, nous pouvons construire et exécuter le programme en mode `release`, ce qui permet de nombreuses optimisations puissantes du compilateur, et après quelques secondes, il écrira une belle image dans le fichier `mandel.png` :

```
$ cargo build --release
    Updating crates.io index
    Compiling autocfg v1.0.1
    ...
    Compiling image v0.13.0
    Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m4.678s
user    0m4.661s
sys     0m0.008s
```

Cette commande doit créer un fichier appelé `mandel.png`, que vous pouvez afficher avec le programme de visualisation d'images de votre système ou dans un navigateur Web. Si tout s'est bien passé, cela devrait ressembler à la [Figure 2-6](#).

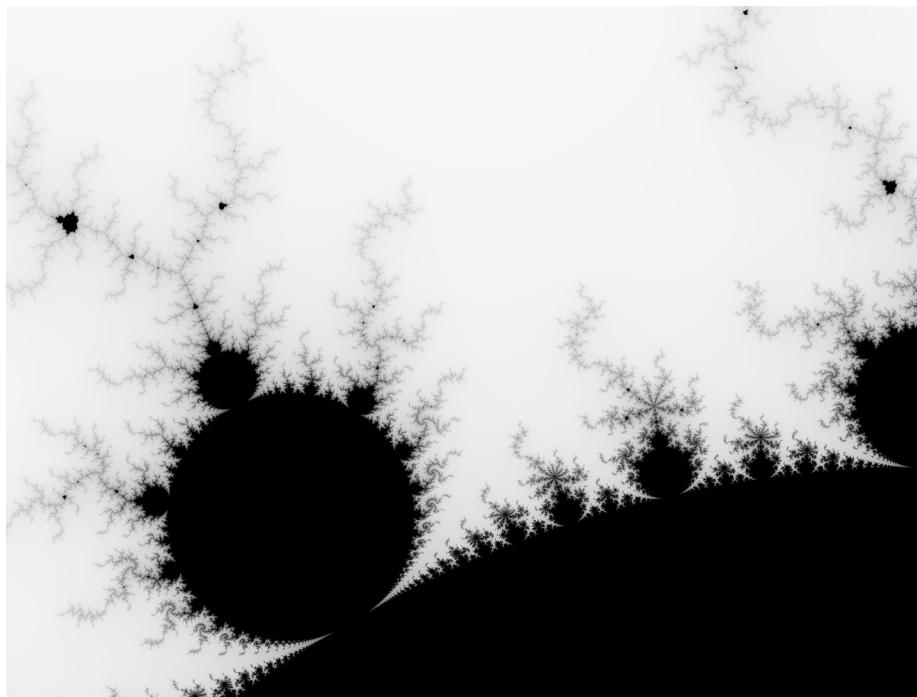


Illustration 2-6. Résultats du programme Mandelbrot parallèle

Dans la transcription précédente, nous avons utilisé le `time` programme Unix pour analyser le temps d'exécution du programme : il a fallu environ cinq secondes au total pour exécuter le calcul de Mandelbrot sur chaque pixel de l'image. Mais presque toutes les machines modernes ont plusieurs coeurs de processeur, et ce programme n'en utilisait qu'un seul.

Si nous pouvions répartir le travail sur toutes les ressources informatiques que la machine a à offrir, nous devrions pouvoir compléter l'image beaucoup plus rapidement.

À cette fin, nous allons diviser l'image en sections, une par processeur, et laisser chaque processeur colorer les pixels qui lui sont attribués. Pour plus de simplicité, nous allons le diviser en bandes horizontales, comme illustré à la [Figure 2-7](#). Lorsque tous les processeurs ont terminé, nous pouvons écrire les pixels sur le disque.

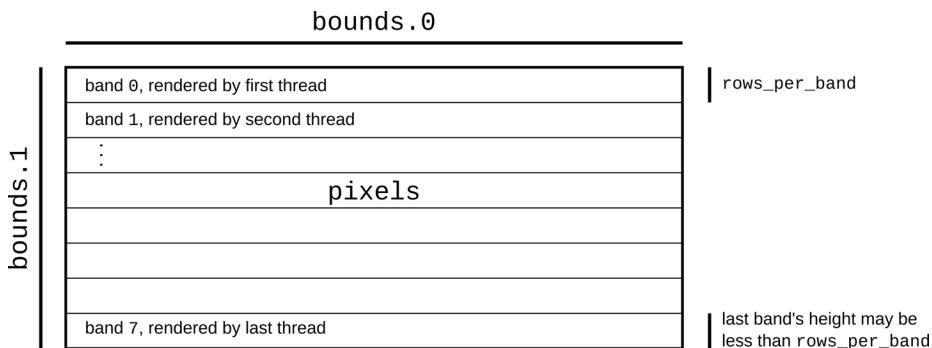


Illustration 2-7. Diviser le tampon de pixels en bandes pour un rendu parallèle

La `crossbeam` caisse fournit un certain nombre d'installations de simultanéité utiles, y compris une installation de `thread` à portée qui fait exactement ce dont nous avons besoin ici. Pour l'utiliser, nous devons ajouter la ligne suivante à notre fichier `Cargo.toml` :

```
crossbeam = "0.8"
```

Ensuite, nous devons supprimer l'appel à une seule ligne `render` et le remplacer par ce qui suit :

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                               upper_left, lower_right);

            spawner.spawn(move |_| {
                render(band, band_bounds, band_upper_left, band_lower_right);
            });
        }
    })
}
```

```
}).unwrap();  
}
```

Décomposer cela de la manière habituelle:

```
let threads = 8;  
let rows_per_band = bounds.1 / threads + 1;
```

Ici, nous décidons d'utiliser huit threads.¹ Ensuite, nous calculons le nombre de rangées de pixels que chaque bande doit avoir. Nous arrondissons la ligne en comptant vers le haut pour nous assurer que les bandes couvrent toute l'image même si la hauteur n'est pas un multiple de threads.

```
let bands:Vec<&mut [u8]> =  
    pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

Ici, nous divisons le tampon de pixels en bandes. La `chunks_mut` méthode du tampon renvoie un itérateur produisant des tranches modifiables et non superposées du tampon, chacune contenant `rows_per_band * bounds.0` des pixels, en d'autres termes, des lignes `rows_per_band` complètes de pixels. La dernière tranche qui `chunks_mut` produit peut contenir moins de lignes, mais chaque ligne contiendra le même nombre de pixels. `collect` Enfin, la méthode de l'itérateur construit un vecteur contenant ces tranches mutables et non superposées.

Maintenant, nous pouvons mettre la `crossbeam` bibliothèque au travail :

```
crossbeam::scope(|spawner| {  
    ...  
}).unwrap();
```

L'argument `|spawner| { ... }` est une fermeture Rust qui attend un seul argument, `spawner`. Notez que, contrairement aux fonctions déclarées avec `fn`, nous n'avons pas besoin de déclarer les types des arguments d'une fermeture ; Rust les déduira, ainsi que son type de retour. Dans ce cas, `crossbeam::scope` appelle la fermeture, en passant comme `spawner` argument une valeur que la fermeture peut utiliser pour créer de nouveaux threads. La `crossbeam::scope` fonction attend que tous ces threads aient terminé leur exécution avant de se retourner. Ce comportement permet à Rust de s'assurer que de tels threads n'accéderont pas à leurs parties `pixels` après qu'il soit sorti de la portée, et nous permet d'être sûr que lors `crossbeam::scope` du retour, le calcul de l'image est terminé. Si tout se passe bien, `crossbeam::scope` renvoie `Ok(())`, mais si l'un des threads que nous avons créés a paniqué, il renvoie un `Err`. Nous faisons appel `unwrap` à cela `Result` de sorte que, dans ce cas, nous paniquerons aussi et l'utilisateur recevra un rapport.

```
for (i, band) in bands.into_iter().enumerate() {
```

Ici, nous parcourons les bandes du tampon de pixels. L' `into_iter()` itérateur donne à chaque itération du corps de la boucle la propriété exclusive d'une bande, garantissant qu'un seul thread peut y écrire à la fois. Nous expliquons comment cela fonctionne en détail au [chapitre 5](#). Ensuite, l'`enumerate` adaptateur produit des tuples associant chaque élément vectoriel à son index.

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                    upper_left, lower_right);
```

Étant donné l'indice et la taille réelle de la bande (rappelons que la dernière peut être plus courte que les autres), nous pouvons produire une boîte englobante du type `render` requis, mais qui se réfère uniquement à cette bande du tampon, pas à l'ensemble image. `pixel_to_point` De même, nous réaffectons la fonction du moteur de rendu pour trouver où les coins supérieur gauche et inférieur droit de la bande tombent sur le plan complexe.

```
spawner.spawn(move |_| {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

Enfin, nous créons un thread, exécutant la fermeture `move |_| { ... }`. Le `move` mot clé au début indique que cette fermeture s'approprie les variables qu'elle utilise ; en particulier, seule la fermeture peut utiliser la tranche mutable `band`. La liste d'arguments `|_|` signifie que la fermeture prend un argument, qu'elle n'utilise pas (un autre générateur pour créer des threads imbriqués).

Comme nous l'avons mentionné précédemment, l'`crossbeam::scope` appel garantit que tous les threads sont terminés avant son retour, ce qui signifie qu'il est sûr d'enregistrer l'image dans un fichier, qui est notre prochaine action..

Exécution du traceur de Mandelbrot

Nous avons utilisé plusieurs caisses externes dans ce programme : `num` pour l'arithmétique des nombres complexes, `image` pour l'écriture de fichiers PNG et `crossbeam` pour les primitives de création de threads

délimitées. Voici le fichier *Cargo.toml* final incluant toutes ces dépendances :

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

Avec cela en place, nous pouvons construire et exécuter le programme :

```
$ cargo build --release
    Updating crates.io index
    Compiling crossbeam-queue v0.3.2
    Compiling crossbeam v0.8.1
    Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
        Finished release [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

Ici, nous avons `time` de nouveau utilisé pour voir combien de temps le programme a mis à s'exécuter ; notez que même si nous avons encore passé près de cinq secondes de temps processeur, le temps réel écoulé n'était que d'environ 1,5 seconde. Vous pouvez vérifier qu'une partie de ce temps est consacrée à l'écriture du fichier `image` en commentant le code qui le fait et en mesurant à nouveau. Sur l'ordinateur portable où ce code a été testé, la version concurrente réduit le temps de calcul de Mandelbrot proprement dit d'un facteur de près de quatre. Nous montrerons comment améliorer considérablement cela au [chapitre 19](#).

Comme précédemment, ce programme aura créé un fichier nommé `mandel.png`. Avec cette version plus rapide, vous pouvez explorer plus facilement l'ensemble Mandelbrot en modifiant les arguments de la ligne de commande à votre guise.

La sécurité est invisible

Au final, le programme parallèle nous nous sommes retrouvés avec n'est pas sensiblement différent de ce que nous pourrions écrire dans n'importe quel autre langage : nous répartissons des morceaux du tampon de pixels entre les processeurs, laissons chacun travailler sur son morceau séparément, et quand ils ont tous fini, présentons le résultat. Alors, qu'y a-t-il de si spécial dans le support de la simultanéité de Rust ?

Ce que nous n'avons pas montré ici, ce sont tous les programmes Rust que nous *ne pouvons pas* écrire. Le code que nous avons examiné dans ce chapitre partitionne correctement le tampon entre les threads, mais il existe de nombreuses petites variations sur ce code qui ne le font pas (et introduisent donc des courses de données); aucune de ces variantes ne passera les vérifications statiques du compilateur Rust. Le compilateur AC ou C++ vous aidera joyeusement à explorer le vaste espace des programmes avec des courses de données subtiles ; Rust vous dit, dès le départ, quand quelque chose pourrait mal tourner.

Dans les chapitres [4](#) et [5](#), nous décrirons les règles de Rust pour la sécurité de la mémoire. [Le chapitre 19](#) explique comment ces règles garantissent également une bonne hygiène de la concurrence.

Systèmes de fichiers et outils de ligne de commande

Rouillera trouvé un créneau important dans le monde des outils en ligne de commande. En tant que langage de programmation système moderne, sûr et rapide, il offre aux programmeurs une boîte à outils qu'ils peuvent utiliser pour assembler des interfaces de ligne de commande astucieuses qui reproduisent ou étendent les fonctionnalités des outils existants. Par exemple, la `bat` commandefournit une alternative sensible à la syntaxe `cat` avec prise en charge intégrée des outils de pagination et `hyperfine` peut évaluer automatiquement tout ce qui peut être exécuté avec une commande ou un pipeline.

Bien que quelque chose d'aussi complexe soit hors de portée de ce livre, Rust vous permet de vous plonger facilement dans le monde des applications ergonomiques en ligne de commande. Dans cette section, nous vous montrerons comment créer votre propre outil de recherche et de remplacement, avec une sortie colorée et des messages d'erreur conviviaux.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo nouveau remplacement
    rapide remplacement      Created binary (application) `quickreplace` package
$ cd rapide
```

Pour notre programme, nous aurons besoin de deux autres caisses : `text-colorizer` pour créer une sortie colorée dans le terminal et `regex` pour la fonctionnalité de recherche et de remplacement proprement dite. Comme précédemment, nous mettons ces caisses dans `Cargo.toml` pour dire `cargo` que nous en avons besoin :

```

[package]
name = "quickreplace"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
text-colorizer = "1"
regex = "1"

```

Les caisses de rouille qui ont atteint la version 1.0 , comme celles-ci, suivent les règles de «version sémantique»: jusqu'à ce que le numéro de version majeur 1 change, les nouvelles versions doivent toujours être des extensions compatibles de leurs prédecesseurs. Donc, si nous testons notre programme par rapport à la version 1.2 d'un certain crate, il devrait toujours fonctionner avec les versions 1.3 , 1.4 , etc. mais la version 2.0 pourrait introduire des modifications incompatibles. Lorsque nous demandons simplement la version "1" d'une caisse dans un fichier *Cargo.toml* , Cargo utilisera la dernière version disponible de la caisse avant 2.0 .

L'interface de ligne de commande

L'interface de ce programme est assez simple. Il prend quatre arguments : une chaîne (ou une expression régulière) à rechercher, une chaîne (ou une expression régulière) pour la remplacer, le nom d'un fichier d'entrée et le nom d'un fichier de sortie. Nous allons commencer notre fichier *main.rs* avec une structure contenant ces arguments :

```

#[derive(Debug)]
struct Arguments {
    target: String,
    replacement: String,
    filename: String,
    output: String,
}

```

L' `#[derive(Debug)]` attribut indique au compilateur de générer du code supplémentaire qui nous permet de formater la `Arguments` structure avec `{:?}` in `println!` .

Au cas où l'utilisateur entrerait le mauvais nombre d'arguments, il est d'usage d'imprimer une explication concise de la façon d'utiliser le programme. Nous allons le faire avec une fonction simple appelé `print_usage` et importez tout de `text-colorizer` sorte que nous puissions ajouter de la couleur :

```

use text_colorizer::*;

fn print_usage() {
    eprintln!("{} - change occurrences of one string into another",
              "quickreplace".green());
    eprintln!("Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>");
}

```

Le simple fait d'ajouter `.green()` à la fin d'un littéral de chaîne produit une chaîne enveloppée dans les codes d'échappement ANSI appropriés à afficher en vert dans un émulateur de terminal. Cette chaîne est ensuite interpolée dans le reste du message avant d'être imprimée.

Nous pouvons maintenant collecter et traiter les arguments du programme :

```

use std::env;

fn parse_args() -> Arguments {
    let args: Vec<String> = env::args().skip(1).collect();

    if args.len() != 4 {
        print_usage();
        eprintln!("{} wrong number of arguments: expected 4, got {}.",
                  "Error:".red().bold(), args.len());
        std::process::exit(1);
    }

    Arguments {
        target: args[0].clone(),
        replacement: args[1].clone(),
        filename: args[2].clone(),
        output: args[3].clone()
    }
}

```

Afin d'obtenir les arguments saisis par l'utilisateur, nous utilisons le même `args` itérateur que dans les exemples précédents.

`.skip(1)` ignore la première valeur de l'itérateur (le nom du programme en cours d'exécution) afin que le résultat n'ait que les arguments de la ligne de commande.

La `collect()` méthode produit un `vec` nombre d'arguments. Nous vérifions ensuite que le bon numéro est présent et, si ce n'est pas le cas, imprimons un message et sortons avec un code d'erreur. Nous colorons à nouveau une partie du message et l'utilisons `.bold()` également pour alourdir le texte. Si le bon nombre d'arguments est présent, nous les mettons dans une `Arguments` structure et la renvoyons.

Ensuite, nous ajouterons une `main` fonction qui appelle `parse_args` et imprime simplement la sortie :

```
fn main() {
    let args = parse_args();
    println!("{}:", args);
}
```

À ce stade, nous pouvons exécuter le programme et voir qu'il crache le bon message d'erreur :

```
$course de fret
Updating crates.io index
Compiling libc v0.2.82
Compiling lazy_static v1.4.0
Compiling memchr v2.3.4
Compiling regex-syntax v0.6.22
Compiling thread_local v1.1.0
Compiling aho-corasick v0.7.15
Compiling atty v0.2.14
Compiling text-colorizer v1.0.0
Compiling regex v1.4.3
Compiling quickreplace v0.1.0 (/home/jimb/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 6.98s
Running `target/debug/quickreplace`
quickreplace - change occurrences of one string into another
Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>
Error: wrong number of arguments: expected 4, got 0
```

Si vous donnez des arguments au programme, il affichera à la place une représentation de la `Arguments` structure :

```
$ "find" "replace" sortie du fichier d'
exécution de la cargaison    Finished dev [unoptimized + debuginfo] target(s) :
                               Running `target/debug/quickreplace find replace file output`
Arguments { target: "find", replacement: "replace", filename: "file", output: "o"
```

C'est un très bon début ! Les arguments sont correctement récupérés et placés dans les bonnes parties de la `Arguments` structure.

Lecture et écriture de fichiers

Prochain, nous avons besoin d'un moyen d'obtenir les données du système de fichiers afin de pouvoir les traiter et les réécrire lorsque nous avons terminé. Rust dispose d'un ensemble d'outils robustes pour l'entrée et la sortie, mais les concepteurs de la bibliothèque standard savent que la lecture et l'écriture de fichiers sont très courantes, et ils l'ont simplifié à dessein. Tout ce que nous avons à faire est d'importer un module, `std::fs` et nous avons accès aux fonctions `read_to_string` et `write`:

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

std::fs::read_to_string renvoie un `Result<String, std::io::Error>`. Si la fonction réussit, elle produit un `String`. S'il échoue, il produit un `std::io::Error`, le type de bibliothèque standard pour représenter les problèmes d'E/S. De même, `std::fs::write` renvoie un `Result<(), std::io::Error>`: rien en cas de succès, ou les mêmes détails d'erreur si quelque chose ne va pas.

```
fn main() {
    let args = parse_args();

    let data = match fs:: read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to read from file '{}': {:?}", "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    match fs:: write(&args.output, &data) {
        Ok(_) => {},
        Err(e) => {
            eprintln!("{} failed to write to file '{}': {:?}", "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };
}
```

Ici, nous utilisons la `parse_args()` fonction que nous avons écrit au préalable et transmis les noms de fichiers résultants à `read_to_string` et `write`. Les `match` instructions sur les sorties de ces fonctions gèrent les erreurs avec élégance, en affichant le nom du fichier, la raison fournie pour l'erreur et une petite touche de couleur pour attirer l'attention de l'utilisateur.

Avec cette `main` fonction mise à jour, nous pouvons exécuter le programme et voir que, bien sûr, le contenu des nouveaux et anciens fichiers est exactement le même :

```
$ cargo run "find" "replace"
Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/quickreplace find replace Cargo.toml Copy.toml`
```

Le programme lit dans le fichier d'entrée `Cargo.toml`, et il écrit dans le fichier de sortie `Copy.toml`, mais puisque nous n'avons pas écrit de code pour réellement rechercher et remplacer, rien dans la sortie n'a changé.

Nous pouvons facilement vérifier en exécutant la commande, qui ne détecte aucune différence : `diff`

```
$diff Cargo.toml Copie.toml
```

Trouver et remplacer

Le finaltouch pour ce programme consiste à implémenter sa fonctionnalité réelle : rechercher et remplacer. Pour cela, nous allons utiliser le `regex` crate, qui compile et exécute des expressions régulières. Il fournit une structure appelée `Regex`, qui représente une expression régulière compilée. `Regex` a une méthode `replace_all` qui fait exactement ce qu'il dit : il recherche dans une chaîne toutes les correspondances de l'expression régulière et remplace chacune par une chaîne de remplacement donnée. Nous pouvons extraire cette logique dans une fonction :

```
use regex:: Regex;
fn replace(target: &str, replacement: &str, text: &str)
    -> Result<String, regex:: Error>
{
    let regex = Regex::new(target)?;
    Ok(regex.replace_all(text, replacement).to_string())
}
```

Notez le type de retour de cette fonction. Tout comme les fonctions de bibliothèque standard que nous avons utilisées précédemment, `replace` renvoie un `Result`, cette fois avec un type d'erreur fourni par le `regex` crate.

`Regex::new` compile l'expression régulière fournie par l'utilisateur et peut échouer si une chaîne non valide lui est donnée. Comme dans le programme Mandelbrot, on ? court-circuite en cas d'`Regex::new` échec, mais dans ce cas la fonction renvoie un type d'erreur spécifique à la `regex` caisse. Une fois l'expression régulière compilée, sa `replace_all` méthode remplace toutes les correspondances `text` avec la chaîne de remplacement donnée.

Si `replace_all` trouve des correspondances, il renvoie un nouveau `String` avec ces correspondances remplacées par le texte que nous lui avons donné. Sinon, `replace_all` renvoie un pointeur vers le texte d'origine, évitant une allocation de mémoire et une copie inutiles. Dans ce cas, cependant, nous voulons toujours une copie indépendante, nous utilisons donc la `to_string` méthode pour obtenir un `String` dans les deux cas et renvoyer cette chaîne enveloppée dans `Result::Ok`, comme dans les autres fonctions.

Il est maintenant temps d'intégrer la nouvelle fonction dans notre `main` code :

```

fn main() {
    let args = parse_args();

    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to read from file '{}': {:?}", 
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    let replaced_data = match replace(&args.target, &args.replacement, &data) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to replace text: {:?}", 
                "Error:".red().bold(), e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &replaced_data) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to write to file '{}': {:?}", 
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };
}

```

Avec cette touche finale, le programme est prêt, et vous devriez pouvoir le tester :

```

$ echo "Hello, world"> test.txt
$cargo run "world" "Rust" test.txt test-modified.txt
Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 0.88s
Running `target/debug/quickreplace world Rust test.txt test-modified.txt` 

$chat test-modified.txt
Hello, Rust

```

Et, bien sûr, la gestion des erreurs est également en place, signalant gracieusement les erreurs à l'utilisateur :

```

$cargo run "[[a-z]]" "0" test.txt test-modified.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/quickreplace '[[a-z]]' 0 test.txt test-modified.txt` 
Error: failed to replace text: Syntax(
-----
regex parse error:
[[a-z]]

```

```
^  
error: unclosed character class  
~~~~~  
)
```

Il manque bien sûr de nombreuses fonctionnalités à cette simple démonstration, mais les fondamentaux sont là. Vous avez vu comment lire et écrire des fichiers, propager et afficher des erreurs et coloriser la sortie pour une meilleure expérience utilisateur dans le terminal.

Les prochains chapitres exploreront des techniques plus avancées pour le développement d'applications, des collections de données et de la programmation fonctionnelle avec des itérateurs aux techniques de programmation asynchrone pour une concurrence extrêmement efficace, mais d'abord, vous aurez besoin des bases solides du chapitre suivant dans les données fondamentales de Rust.¹ les types.

¹ La `num_cpus` caisse fournit une fonction qui renvoie le nombre de processeurs disponibles sur le système actuel.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 3. Types fondamentaux

Il existe de nombreux types de livres dans le monde, ce qui est logique, car il existe de très nombreux types de personnes et tout le monde veut lire quelque chose de différent.

—Snicket citronné

Dans une large mesure, le langage Rust est conçu autour de ses types. Sa prise en charge d'un code haute performance découle du fait qu'il permet aux développeurs de choisir la représentation des données qui correspond le mieux à la situation, avec le bon équilibre entre simplicité et coût. Les garanties de sécurité de la mémoire et des threads de Rust reposent également sur la solidité de son système de types, et la flexibilité de Rust découle de ses types et traits génériques.

Ce chapitre couvre les types fondamentaux de Rust pour représenter les valeurs. Ces types au niveau de la source ont des équivalents concrets au niveau de la machine avec des coûts et des performances prévisibles. Bien que Rust ne promette pas qu'il représentera les choses exactement comme vous l'avez demandé, il prend soin de ne s'écartez de vos demandes que lorsqu'il s'agit d'une amélioration fiable.

Comparé à un langage typé dynamiquement comme JavaScript ou Python, Rust nécessite plus de planification de votre part. Vous devez préciser les types d'arguments de fonction et les valeurs de retour, les champs de structure et quelques autres constructions. Cependant, deux fonctionnalités de Rust rendent cela moins problématique que prévu :

- Étant donné les types que vous épelez, l'*inférence de type* de Rust trouvera la plupart du reste pour vous. En pratique, il n'y a souvent qu'un seul type qui fonctionnera pour une variable ou une expression donnée ; lorsque c'est le cas, Rust vous permet de laisser de côté, ou d'*elide*, le type. Par exemple, vous pouvez épeler chaque type dans une fonction, comme ceci :

```
fn build_vector() -> Vec<i16> {
    let mut v: Vec<i16> = Vec::new();
    v.push(10i16);
    v.push(20i16);
    v
}
```

Mais c'est encombré et répétitif. Étant donné le type de retour de la fonction, il est évident que `v` doit être à `Vec<i16>`, un vecteur d'entiers signés 16 bits ; aucun autre type ne fonctionnerait. Et de là, il s'ensuit que chaque élément du vecteur doit être un `i16`. C'est exactement le genre de raisonnement que l'inférence de type de Rust applique, vous permettant d'écrire à la place :

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v
}
```

Ces deux définitions sont exactement équivalentes et Rust générera le même code machine dans les deux cas. L'inférence de type redonne une grande partie de la lisibilité des langages à typage dynamique, tout en capturant les erreurs de type au moment de la compilation.

- Les fonctions peuvent être *génériques*: une seule fonction peut travailler sur des valeurs de plusieurs types différents.

En Python et JavaScript, toutes les fonctions fonctionnent naturellement de cette façon : une fonction peut opérer sur n'importe quelle valeur qui possède les propriétés et les méthodes dont la fonction aura besoin. (C'est la caractéristique souvent appelée *duck typing*: s'il canne comme un canard, c'est un canard.) Mais c'est précisément cette flexibilité qui rend si difficile pour ces langages la détection précoce des erreurs de type ; les tests sont souvent le seul moyen de détecter de telles erreurs. Les fonctions génériques de Rust donnent au langage le même degré de flexibilité, tout en captant toutes les erreurs de type au moment de la compilation.

Malgré leur flexibilité, les fonctions génériques sont tout aussi efficaces que leurs homologues non génériques. Il n'y a aucun avantage inhérent en termes de performances à tirer de l'écriture, par exemple, d'une `sum` fonction spécifique pour chaque entier par rapport à l'écriture d'une fonction générique qui gère tous les entiers. Nous aborderons les fonctions génériques en détail au [chapitre 11](#).

Le reste de ce chapitre couvre les types de Rust de bas en haut, en commençant par les types numériques simples comme les entiers et les valeurs à virgule flottante, puis en passant aux types qui contiennent plus de données : boîtes, tuples, tableaux et chaînes.

Voici un résumé des types de types que vous verrez dans Rust. [Le tableau 3-1](#) montre les types primitifs de Rust, certains types très courants de la bibliothèque standard et quelques exemples de types définis par l'utilisateur.

Tableau 3-1. Exemples de types en Rust

Taper	La description	Valeurs
i8 , i16 , i 32 , i64 , i1 28 u8 , u16 , u 32 , u64 , u12 8	Entiers signés et non signés, de largeur de bit donnée	42 , -5i8 , 0x40 0u16 , 0o10 0i16 , 20_922_78 9_888_000 u64 , b'*'(
		u8 octet littéral)
isize , usiz e	Entiers signés et non signés, de même taille qu'une adresse sur la machine (32 ou 64 bits)	137 , -0b0101_0 010isize , 0xffff_fc 00usize
f32 , f64	Nombres à virgule flottante IEEE, simple et double précision	1.61803 , 3.14f32 , 6.0221e23 f64
bool	booléen	true , fals e
char	Caractère Unicode, largeur 32 bits	'*' , '\n' , '字' , '\x7 f' , '\u{CA 0}'
(char , u8 , i32)	Tuple : types mixtes autorisés	('%' , 0x7 f , -1)
()	"Unité" (tuple vide)	()
struct S { x: f32 , y: f32 }	Structure de champ nommé	S { x: 12 0.0 , y: 20 9.0 }

Taper	La description	Valeurs
struct T (i32, char);	Structure de type tuple	T(120, 'x')
struct E;	Structure de type unité ; n'a pas de champs	E
enum Attended { OnTime, Late(u32) }	Énumération, type de données algébrique	Attended::Late(5), Attended::OnTime
Box<Attended>	Boîte : posséder un pointeur vers la valeur dans le tas	Box::new(Late(1))
&i32, &mut i32	Références partagées et mutables : pointeurs non propriétaires qui ne doivent pas survivre à leur référent	&s.y, &mut v
String	Chaîne UTF-8, dimensionnée dynamiquement	"ラーメン：ラーメン".to_string()
&str	Référence à str : pointeur non propriétaire vers le texte UTF-8	"そば：そば", &s[0..12]
[f64; 4], [u8; 256]	Array, longueur fixe ; éléments tous du même type	[1.0, 0.0, 0.0, 1.0], [b' '; 256]
Vec<f64>	Vecteur, longueur variable ; éléments tous du même type	vec![0.367, 2.718, 7.389]

Taper	La description	Valeurs
<code>&[u8], &mut [u8]</code>	Référence à la tranche : référence à une partie d'un tableau ou d'un vecteur, comprenant un pointeur et une longueur	<code>&v[10..20], &mut a[..]</code>
<code>Option<&str r></code>	Valeur facultative : soit <code>None</code> (absent), soit <code>Some(v)</code> (présent, avec la valeur <code>v</code>)	<code>Some("Dr.")</code> , <code>None</code>
<code>Result<u64, Error></code>	Résultat de l'opération qui peut échouer : soit une valeur de succès <code>Ok(v)</code> , soit une erreur <code>Err(e)</code>	<code>Ok(4096)</code> , <code>Err(Error::last_os_error())</code>
<code>&dyn Any, &mut dyn Read</code>	Objet trait : référence à toute valeur qui implémente un ensemble donné de méthodes	<code>value as &dyn Any, &mut file as &mut dyn Read</code>
<code>fn(&str) -> bool</code>	Pointeur vers la fonction	<code>str::is_empty</code>
(Les types de fermeture n'ont pas de forme écrite)	Fermeture	<code> a, b { a*a + b*b }</code>

La plupart de ces types sont traités dans ce chapitre, à l'exception des suivants :

- Nous donnons aux `struct` types leur propre chapitre, [Chapitre 9](#).
- Nous donnons aux types énumérés leur propre chapitre, le [chapitre 10](#).
- Nous décrivons les objets trait au [chapitre 11](#).
- Nous décrivons l'essentiel de `String` et `&str` ici, mais fournissons plus de détails au [chapitre 17](#).
- Nous couvrons les types de fonctions et de fermetures au [chapitre 14](#).

Types numériques à largeur fixe

La base du système de types de Rust est une collection de types numériques à largeur fixe, choisis pour correspondre aux types que presque tous les processeurs modernes implémentent directement dans le matériel.

Les types numériques à largeur fixe peuvent déborder ou perdre en précision, mais ils conviennent à la plupart des applications et peuvent être des milliers de fois plus rapides que des représentations telles que des entiers à précision arbitraire et des rationnels exacts. Si vous avez besoin de ces types de représentations numériques, elles sont prises en charge dans la `num` caisse.

Les noms des types numériques de Rust suivent un modèle régulier, épelant leur largeur en bits et la représentation qu'ils utilisent ([Tableau 3-2](#)).

Tableau 3-2. Types numériques de rouille

Taille (bits)	Entier non signé	Entier signé	Point flottant
8	<code>u8</code>	<code>i8</code>	
16	<code>u16</code>	<code>i16</code>	
32	<code>u32</code>	<code>i32</code>	<code>f32</code>
64	<code>u64</code>	<code>i64</code>	<code>f64</code>
128	<code>u128</code>	<code>i128</code>	
Mot machine	<code>usize</code>	<code>isize</code>	

Ici, un *mot machine* est une valeur de la taille d'une adresse sur la machine sur laquelle le code s'exécute, 32 ou 64 bits.

Types entiers

Rust n'est pas signé types entiers utiliser leur plage complète pour représenter les valeurs positives et zéro ([tableau 3-3](#)).

Tableau 3-3. Rust types entiers non signés

Taper	Intervalle
u8	0 à $2^8 - 1$ (0 à 255)
u16	0 à $2^{16} - 1$ (0 à 65 535)
u32	0 à $2^{32} - 1$ (0 à 4 294 967 295)
u64	0 à $2^{64} - 1$ (0 à 18 446 744 073 709 551 615 ou 18 quintillions)
u128	0 à $2^{128} - 1$ (0 à environ $3,4 \times 10^{38}$)
usize	0 à $2^{32} - 1$ ou $2^{64} - 1$

Rust est signé les types entiers utilisent la représentation en complément à deux, en utilisant les mêmes modèles de bits que le type non signé correspondant pour couvrir une plage de valeurs positives et négatives ([Tableau 3-4](#)).

Tableau 3-4. Types entiers signés Rust

Taper	Intervalle
i8	-2^7 à $2^7 - 1$ (-128 à 127)
i16	-2^{15} à $2^{15} - 1$ (-32 768 à 32 767)
i32	-2^{31} à $2^{31} - 1$ (-2 147 483 648 à 2 147 483 647)
i64	-2^{63} à $2^{63} - 1$ (-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807)
i128	-2^{127} à $2^{127} - 1$ (environ $-1,7 \times 10^{38}$ à $+1,7 \times 10^{38}$)
isize	Soit -2^{31} à $2^{31} - 1$, soit -2^{63} à $2^{63} - 1$

Rust utilise le `u8` type pour les valeurs d'octets. Par exemple, la lecture de données à partir d'un fichier binaire ou d'un socket génère un flux de `u8` valeurs.

Contrairement à C et C++, Rust traite les caractères comme distincts des types numériques : a `char` n'est pas un `u8`, ni un `u32` (bien qu'il fasse 32 bits). Nous décrivons le type de Rust `char` dans "[Personnages](#)".

Les types `usize` et `isize` sont analogues à `size_t` et `ptrdiff_t` en C et C++. Leur précision correspond à la taille de l'espace d'adressage sur la machine cible : elles ont une longueur de 32 bits sur les architectures 32 bits et de 64 bits sur les architectures 64 bits. Rust nécessite que les indices de tableau soient des `usize` valeurs. Les valeurs représentant les tailles de tableaux ou de vecteurs ou le nombre d'éléments dans certaines structures de données ont également généralement le `usize` type.

Littéraux entiers en Rust peuvent prendre un suffixe indiquant leur type : `42u8` est une `u8` valeur, et `1729isize` est un `isize`. Si un littéral entier n'a pas de suffixe de type, Rust reporte la détermination de son type jusqu'à ce qu'il trouve la valeur utilisée d'une manière qui la fixe : stockée dans une variable d'un type particulier, transmise à une fonction qui attend un type particulier, comparée avec une autre valeur d'un type particulier, ou quelque chose comme ça. En fin de compte, si plusieurs types peuvent fonctionner, Rust utilise par défaut `i32` si cela fait partie des possibilités. Sinon, Rust signale l'ambiguïté comme une erreur.

Les préfixes `0x`, `0o` et `0b` désignent des littéraux hexadécimaux, octaux et binaires.

Pour rendre les nombres longs plus lisibles, vous pouvez insérer des traits de soulignement entre les chiffres. Par exemple, vous pouvez écrire la plus grande `u32` valeur sous la forme `4_294_967_295`. L'emplacement exact des traits de soulignement n'est pas significatif, vous pouvez donc diviser les nombres hexadécimaux ou binaires en groupes de quatre chiffres au lieu de trois, comme dans `0xffff_ffff`, ou définir le suffixe de type à partir des chiffres, comme dans `127_u8`. Quelques exemples de littéraux entiers sont illustrés dans [le Tableau 3-5](#).

Tableau 3-5. Exemples de littéraux entiers

Littéral	Taper	Valeur décimale
116i8	i8	116
0xcafeu32	u32	51966
0b0010_1010	Inféré	42
0o106	Inféré	70

Bien que les types numériques et le `char` type soient distincts, Rust fournit *des littéraux d'octets*, littéraux de type caractère pour les `u8` valeurs : `b'x'` représente le code ASCII du caractère `x`, sous forme de `u8` valeur. Par exemple, puisque le code ASCII pour `A` est 65, les littéraux `b'A'` et `65u8` sont exactement équivalents. Seuls les caractères ASCII peuvent apparaître dans les littéraux d'octets.

Il y a quelques caractères que vous ne pouvez pas simplement placer après le guillemet simple, car cela serait soit syntaxiquement ambigu, soit difficile à lire. Les caractères du [tableau 3-6](#) ne peuvent être écrits qu'à l'aide d'une notation de remplacement, introduite par une barre oblique inverse.

Tableau 3-6. Caractères nécessitant une notation de remplacement

Personnage	Octet littéral	Équivalent numérique
Simple citation, '	<code>b'\ ''</code>	<code>39u8</code>
barre oblique inverse, \	<code>b'\\\'</code>	<code>92u8</code>
Nouvelle ligne	<code>b'\n'</code>	<code>10u8</code>
Retour chariot	<code>b'\r'</code>	<code>13u8</code>
Languette	<code>b'\t'</code>	<code>9u8</code>

Pour les caractères difficiles à écrire ou à lire, vous pouvez écrire leur code en hexadécimal à la place. Un littéral d'octet de la forme `b'\xHH'`, où `HH` est un nombre hexadécimal à deux chiffres, représente l'octet dont la valeur est `HH`. Par exemple, vous pouvez écrire un littéral d'octet pour

le caractère de contrôle « échappement » ASCII sous `b'\x1b'` la forme , puisque le code ASCII pour « échappement » est 27, ou 1B en hexadécimal. Étant donné que les littéraux d'octets ne sont qu'une autre notation pour les `u8` valeurs, demandez-vous si un simple littéral numérique pourrait être plus lisible : il est probablement judicieux de l'utiliser `b'\x1b'` plutôt que simplement `27` uniquement lorsque vous souhaitez souligner que la valeur représente un code ASCII.

Vous pouvez convertir d'un type entier à un autre à l'aide de l' `as` opérateur. Nous expliquons comment fonctionnent les conversions dans "[Type Casts](#)" , mais voici quelques exemples :

```
assert_eq!( 10_i8 as u16,    10_u16); // in range
assert_eq!( 2525_u16 as i16,  2525_i16); // in range

assert_eq!( -1_i16 as i32,   -1_i32); // sign-extended
assert_eq!( 65535_u16 as i32, 65535_i32); // zero-extended

// Conversions that are out of range for the destination
// produce values that are equivalent to the original modulo 2^N,
// where N is the width of the destination in bits. This
// is sometimes called "truncation."
assert_eq!( 1000_i16 as u8,   232_u8);
assert_eq!( 65535_u32 as i16, -1_i16);

assert_eq!( -1_i8   as u8,    255_u8);
assert_eq!( 255_u8  as i8,    -1_i8);
```

La bibliothèque standard fournit certaines opérations sous forme de méthodes sur des nombres entiers. Par exemple:

```
assert_eq!(2_u16.pow(4), 16);           // exponentiation
assert_eq!((-4_i32).abs(), 4);          // absolute value
assert_eq!(0b101101_u8.count_ones(), 4); // population count
```

Vous pouvez les trouver dans la documentation en ligne. Notez cependant que la documentation contient des pages séparées pour le type lui-même sous " `i32` (type primitif)" , et pour le module dédié à ce type (recherchez " `std::i32` ").

Dans le code réel, vous n'aurez généralement pas besoin d'écrire les suffixes de type comme nous l'avons fait ici, car le contexte déterminera le type. Quand ce n'est pas le cas, cependant, les messages d'erreur peuvent être surprenants. Par exemple, ce qui suit ne compile pas :

```
println!("{}" , (-4).abs());
```

La rouille se plaint :

```
error: can't call method `abs` on ambiguous numeric type `{integer}`
```

Cela peut être un peu déconcertant : tous les types d'entiers signés ont une `abs` méthode, alors quel est le problème ? Pour des raisons techniques, Rust veut savoir exactement quel type d'entier a une valeur avant d'appeler les propres méthodes du type. La valeur par défaut de `i32` s'applique uniquement si le type est toujours ambigu après la résolution de tous les appels de méthode, il est donc trop tard pour vous aider ici. La solution consiste à préciser le type souhaité, soit avec un suffixe, soit en utilisant la fonction d'un type spécifique :

```
println!("{}" , (-4_i32).abs());  
println!("{}" , i32::abs(-4));
```

Notez que les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe unaire, soyez donc prudent lorsque vous appliquez des méthodes à des valeurs négatives. Sans les parenthèses autour `-4_i32` de la première instruction, `-4_i32.abs()` appliquerait la `abs` méthode à la valeur positive `4`, produisant positive `4`, puis nierait cela, produisant `-4`.

Arithmétique vérifiée, enveloppante, saturée et débordante

Lorsqu'une opération arithmétique entière déborde, Rust panique dans une version de débogage. Dans une version de version, l'opération *se termine*: il produit la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur. (Dans aucun des deux cas, le comportement de débordement n'est défini, comme c'est le cas en C et C++.)

Par exemple, le code suivant panique dans une version de débogage :

```
let mut i = 1;  
loop {  
    i *= 10; // panic: attempt to multiply with overflow  
             // (but only in debug builds!)  
}
```

Dans une version de version, cette multiplication revient à un nombre négatif et la boucle s'exécute indéfiniment.

Lorsque ce comportement par défaut n'est pas ce dont vous avez besoin, les types entiers fournissent des méthodes qui vous permettent d'épeler exactement ce que vous voulez. Par exemple, les paniques suivantes dans n'importe quel build :

```
let mut i:i32 = 1;
loop {
    // panic: multiplication overflowed (in any build)
    i = i.checked_mul(10).expect("multiplication overflowed");
}
```

Ces méthodes d'arithmétique entière se répartissent en quatre catégories générales :

- Opérations vérifiées renvoie un Option du résultat : Some(v) si le résultat mathématiquement correct peut être représenté comme une valeur de ce type, ou None s'il ne le peut pas. Par exemple:

```
// The sum of 10 and 20 can be represented as a u8.
assert_eq!(10_u8.checked_add(20), Some(30));

// Unfortunately, the sum of 100 and 200 cannot.
assert_eq!(100_u8.checked_add(200), None);

// Do the addition; panic if it overflows.
let sum = x.checked_add(y).unwrap();

// Oddly, signed division can overflow too, in one particular case.
// A signed n-bit type can represent  $-2^{n-1}$ , but not  $2^{n-1}$ .
assert_eq!((-128_i8).checked_div(-1), None);
```

- Opérations d'*emballage* renvoie la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur :

```
// The first product can be represented as a u16;
// the second cannot, so we get 250000 modulo  $2^{16}$ .
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// Operations on signed types may wrap to negative values.
assert_eq!(500_i16.wrapping_mul(500), -12144);
```

```

// In bitwise shift operations, the shift distance
// is wrapped to fall within the size of the value.
// So a shift of 17 bits in a 16-bit type is a shift
// of 1.
assert_eq!(5_i16.wrapping_shl(17), 10);

```

Comme expliqué, c'est ainsi que les opérateurs arithmétiques ordinaires se comportent dans les versions de version. L'avantage de ces méthodes est qu'elles se comportent de la même manière dans toutes les versions.

- Opérations *saturantes* renvoie la valeur représentable la plus proche du résultat mathématiquement correct. En d'autres termes, le résultat est "bridé" aux valeurs maximales et minimales que le type peut représenter :

```

assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);

```

Il n'y a pas de méthode de division saturante, de reste ou de décalage au niveau du bit.

- Opérations *débordantes* renvoie un tuple `(result, overflowed)`, où `result` est ce que la version d'encapsulation de la fonction renverrait, et indique si un débordement s'est produit `overflowed : bool`

```

assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(255_u8.overflowing_add(2), (1, true));

```

`overflowing_shl` et `overflowing_shr` s'écartent un peu du motif : ils ne renvoient `true` `overflowed` que si la distance de décalage était aussi grande ou supérieure à la largeur en bits du type lui-même. Le décalage réel appliqué est le décalage demandé modulo la largeur de bit du type :

```

// A shift of 17 bits is too large for `u16`, and 17 modulo 16 is 1.
assert_eq!(5_u16.overflowing_shl(17), (10, true));

```

Les noms d'opération qui suivent le préfixe `,`, ou sont indiqués `checked_` dans `wrapping_` le `saturating_` Tableau [3-7](#)
`.overflowing_`

Tableau 3-7. Noms d'opération

Opération	Suffixe de nom	Exemple
Ajout	add	<code>100_i8.checked_add(27) = = Some(127)</code>
Soustraction	sub	<code>10_u8.checked_sub(11) == None</code>
Multiplication	mul	<code>128_u8.saturating_mul(3) == 255</code>
Division	div	<code>64_u16.wrapping_div(8) = = 8</code>
Reste	rem	<code>(-32768_i16).wrapping_re m(-1) == 0</code>
Négation	neg	<code>(-128_i8).checked_neg() == None</code>
Valeur absolue	abs	<code>(-32768_i16).wrapping_ab s() == -32768</code>
Exponentiation	pow	<code>3_u8.checked_pow(4) == S ome(81)</code>
Décalage à gauche au niveau du bit	shl	<code>10_u32.wrapping_shl(34) == 40</code>
Décalage à droite au niveau du bit	shr	<code>40_u64.wrapping_shr(66) == 10</code>

Types à virgule flottante

Rouiller fournit des types à virgule flottante simple et double précision IEEE. Ces types incluent des infinis positifs et négatifs, des valeurs zéro positives et négatives distinctes et une valeur *non numérique* ([Tableau 3-8](#)).

Tableau 3-8. Types à virgule flottante simple et double précision IEEE

Taper	Précision	Intervalle
f32	Simple précision IEEE (au moins 6 chiffres décimaux)	Environ $-3,4 \times 10^{-38}$ à $+3,4 \times 10^{38}$
f64	Double précision IEEE (au moins 15 chiffres décimaux)	Environ $-1,8 \times 10^{-308}$ à $+1,8 \times 10^{308}$

Rust f32 et f64 correspondent aux types float et double en C et C++ (dans les implémentations prenant en charge la virgule flottante IEEE) ainsi qu'en Java (qui utilise toujours la virgule flottante IEEE).

Littéraux à virgule flottante avoir la forme générale schématisée à la [Figure 3-1](#).

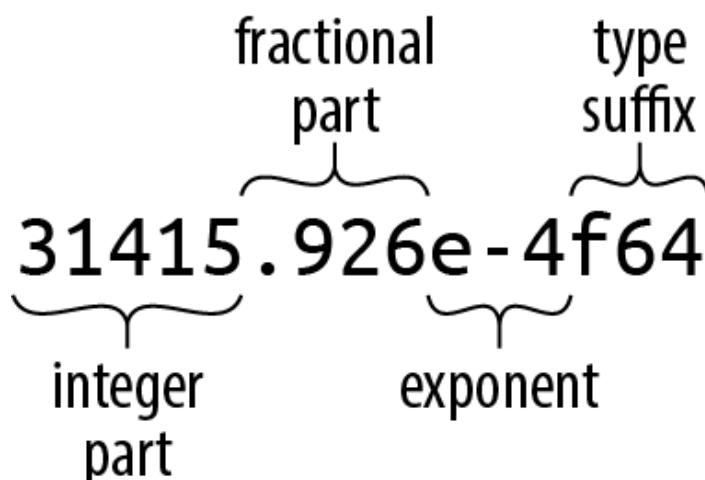


Illustration 3-1. Un littéral à virgule flottante

Chaque partie d'un nombre à virgule flottante après la partie entière est facultative, mais au moins une partie fractionnaire, un exposant ou un suffixe de type doit être présent, pour le distinguer d'un littéral entier. La partie fractionnaire peut être constituée d'un seul point décimal, c'est donc 5. une constante à virgule flottante valide.

Si un littéral à virgule flottante n'a pas de suffixe de type, Rust vérifie le contexte pour voir comment les valeurs sont utilisées, tout comme il le fait pour les littéraux entiers. S'il trouve finalement que l'un ou l'autre des types à virgule flottante peut convenir, il choisit f64 par défaut.

Aux fins de l'inférence de type, Rust traite les littéraux entiers et les littéraux à virgule flottante en tant que classes distinctes : il n'inférera jamais un type à virgule flottante pour un littéral entier, ou vice versa. [Le tableau 3-9](#) montre quelques exemples de littéraux à virgule flottante.

Tableau 3-9. Exemples de littéraux à virgule flottante

Littéral	Taper	Valeur mathématique
-1.5625	Inféré	$-(1 \frac{9}{16})$
2.	Inféré	2
0.25	Inféré	$\frac{1}{4}$
1e4	Inféré	10 000
40f32	f32	40
9.109_383_56e-31f64	f64	Environ $9,10938356 \times 10^{-31}$

Les types `f32` et `f64` ont des constantes associées pour les valeurs spéciales requises par l'IEEE telles que `INFINITY`, `NEG_INFINITY` (infini négatif), `NAN` (la valeur non numérique) et `MIN` et `MAX` (les valeurs finies les plus grandes et les plus petites) :

```
assert!((-1. / f32:: INFINITY).is_sign_negative());
assert_eq!(-f32:: MIN, f32::MAX);
```

Les types `f32` et `f64` fournissent un complément complet de méthodes pour les calculs mathématiques ; par exemple, `2f64.sqrt()` est la racine carrée double précision de deux. Quelques exemples:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // exactly 5.0, per IEEE
assert_eq!((-1.01f64).floor(), -2.0);
```

Encore une fois, les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe, alors assurez-vous de bien mettre entre parenthèses les appels de méthode sur les valeurs négatives.

Les modules `std::f32::consts` et `std::f64::consts` fournissent diverses constantes mathématiques couramment utilisées telles que `E`, `PI` et la racine carrée de deux.

Lors de la recherche dans la documentation, n'oubliez pas qu'il existe des pages pour les types eux-mêmes, nommés " `f32` (type primitif)" et " `f64` (type primitif)", et les modules pour chaque type, `std::f32` et `std::f64`.

Comme pour les entiers, vous n'aurez généralement pas besoin d'écrire des suffixes de type sur des littéraux à virgule flottante dans du code réel, mais lorsque vous le ferez, mettre un type sur le littéral ou sur la fonction suffira :

```
println!("{}", (2.0_f64).sqrt());  
println!("{}", f64::sqrt(2.0));
```

Contrairement à C et C++, Rust n'effectue presque aucune conversion numérique implicite. Si une fonction attend un `f64` argument, c'est une erreur de passer une `i32` valeur comme argument. En fait, Rust ne convertera même pas implicitement une `i16` valeur en une `i32` valeur, même si chaque `i16` valeur est également une `i32` valeur. Mais vous pouvez toujours écrire des conversions *explicites* `as` en utilisant l'opérateur : `i as f64`, ou `x as i32`.

L'absence de conversions implicites rend parfois une expression Rust plus détaillée que ne le serait le code C ou C++ analogue. Cependant, les conversions implicites d'entiers ont un historique bien établi de bogues et de failles de sécurité, en particulier lorsque les entiers en question représentent la taille de quelque chose en mémoire et qu'un débordement imprévu se produit. D'après notre expérience, le fait d'écrire des conversions numériques dans Rust nous a alertés sur des problèmes que nous aurions autrement manqués.

Nous expliquons exactement comment les conversions se comporter dans ["Type Casts"](#).

Le type booléen

Booléen de Rusttype, `bool`, a les deux valeurs habituelles pour de tels types, `true` et `false`. Opérateurs de comparaison `==` et `<` produisent des `bool` résultats : la valeur de `2 < 5` est `true`.

De nombreux langages sont indulgents quant à l'utilisation de valeurs d'autres types dans des contextes qui nécessitent une valeur booléenne : C et C++ convertissent implicitement les caractères, les entiers, les nombres à virgule flottante et les pointeurs en valeurs booléennes, afin qu'ils puissent être utilisés directement comme condition dans un `if` ou `while` déclaration. Python autorise les chaînes, les listes, les dictionnaires et même les ensembles dans des contextes booléens, traitant ces valeurs comme vraies si elles ne sont pas vides. Rust, cependant, est très

strict : les structures de contrôle comme `if` et `while` exigent que leurs conditions soient `bool` des expressions, tout comme les opérateurs logiques de court-circuit `&&` et `||`. Vous devez écrire `if x != 0 { ... }`, pas simplement `if x { ... }`.

`as` L'opérateur de Rust peut convertir `bool` des valeurs en types entiers :

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

Cependant, `as` ne convertira pas dans l'autre sens, des types numériques vers `bool`. Au lieu de cela, vous devez écrire une comparaison explicite telle que `x != 0`.

Bien que a n'ait `bool` besoin que d'un seul bit pour le représenter, Rust utilise un octet entier pour une `bool` valeur en mémoire, vous pouvez donc créer un pointeur vers celui-ci.

Personnages

Le personnage de Rust type `char` représente un seul caractère Unicode, sous la forme d'une valeur 32 bits.

Rust utilise le `char` type pour les caractères uniques de manière isolée, mais utilise l'encodage UTF-8 pour les chaînes et les flux de texte. Ainsi, a `String` représente son texte comme une séquence d'octets UTF-8, pas comme un tableau de caractères.

Personnageles littéraux sont des caractères entre guillemets simples, comme `'8'` ou `'!'`. Vous pouvez utiliser toute l'étendue d'Unicode : `'鏽'` est un `char` littéral représentant le kanji japonais pour *sabi* (rouille).

Comme pour les littéraux d'octets, des échappements par barre oblique inverse sont requis pour quelques caractères ([Tableau 3-10](#)).

Tableau 3-10. Caractères nécessitant des échappements par barre oblique inverse

Personnage	Littéral de caractère de rouille
Simple citation, '	'\\'
barre oblique inverse, \	'\\\\'
Nouvelle ligne	'\\n'
Retour chariot	'\\r'
Languette	'\\t'

Si vous préférez, vous pouvez écrire le point de code Unicode d'un caractère en hexadécimal:

- Si le point de code du caractère est compris entre U+0000 et U+007F (c'est-à-dire s'il provient du jeu de caractères ASCII), vous pouvez écrire le caractère sous la forme '\xHH', où HH est un nombre hexadécimal à deux chiffres. Par exemple, les caractères littéraux '*' et '\x2A' sont équivalents, car le point de code du caractère * est 42, ou 2A en hexadécimal.
- Vous pouvez écrire n'importe quel caractère Unicode sous la forme '\u{HHHHHH}', où HHHHHH est un nombre hexadécimal jusqu'à six chiffres, avec des traits de soulignement autorisés pour le regroupement comme d'habitude. Par exemple, le caractère littéral '\u{CA0}' représente le caractère "ಠ", un caractère Kannada utilisé dans l'Unicode Look of Disapproval, "ಠ_ಠ". Le même littéral pourrait aussi être simplement écrit comme 'ಠ' .

A `char` contient toujours un point de code Unicode dans la plage 0x0000 à 0xD7FF, ou 0xE000 à 0x10FFFF. A `char` n'est jamais une moitié de paire de substitution (c'est-à-dire un point de code compris entre 0xD800 et 0xDFFF) ou une valeur en dehors de l'espace de code Unicode (c'est-à-dire supérieure à 0x10FFFF). Rust utilise le système de type et les vérifications dynamiques pour s'assurer que `char` les valeurs sont toujours dans la plage autorisée.

Rust ne convertit jamais implicitement entre `char` et tout autre type. Vous pouvez utiliser l' `as` opérateur de conversion pour convertir a `char` en un type entier ; pour les types inférieurs à 32 bits, les bits supérieurs de la valeur du caractère sont tronqués :

```
assert_eq!('*' as i32, 42);
assert_eq!(‘ø’ as u16, 0xca0);
assert_eq!(‘ø’ as i8, -0x60); // U+0CA0 truncated to eight bits, signed
```

Dans l'autre sens, `u8` est le seul type `as` vers lequel l'opérateur convertira `char`: Rust a l'intention que l'`as` opérateur n'effectue que des conversions bon marché et infaillibles, mais chaque type d'entier autre que `u8` comprend des valeurs qui ne sont pas autorisées. points de code Unicode, donc ces conversions nécessiteraient run -vérifications horaires. Au lieu de cela, la fonction de bibliothèque standard

`std::char::from_u32` prend n'importe quelle `u32` valeur et renvoie un `Option<char>`: si `u32` n'est pas un point de code Unicode autorisé, alors `from_u32` renvoie `None`; sinon, elle renvoie `Some(c)`, où `c` est le `char` résultat.

La bibliothèque standard fournit des méthodes utiles sur les caractères, que vous pouvez rechercher dans la documentation en ligne sous « `char` (type primitif) » et le module « » `std::char`. Par exemple:

```
assert_eq!(‘*’.is_alphabetic(), false);
assert_eq!(‘β’.is_alphabetic(), true);
assert_eq!(‘8’.to_digit(10), Some(8));
assert_eq!(‘ø’.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

Naturellement, les caractères isolés ne sont pas aussi intéressants que les chaînes et les flux de texte. Nous décrirons le `String` type standard de Rust et la gestion du texte en général dans "[Types de chaînes](#)".

Tuples

Un *tuple* est une paire, ou triple, quadruple, quintuple, etc. (donc, *n-tuple*, ou *tuple*), de valeurs de types assortis. Vous pouvez écrire un tuple comme une séquence d'éléments, séparés par des virgules et entourés de parenthèses. Par exemple, `("Brazil", 1985)` est un tuple dont le premier élément est une chaîne allouée statiquement et dont le second est un entier ; son type est `(&str, i32)`. Étant donné une valeur de tuple `t`, vous pouvez accéder à ses éléments en tant que `t.0`, `t.1`, etc.

Dans une certaine mesure, les tuples ressemblent à des tableaux: les deux types représentent une séquence ordonnée de valeurs. De nombreux langages de programmation confondent ou combinent les deux concepts,

mais dans Rust, ils sont complètement séparés. D'une part, chaque élément d'un tuple peut avoir un type différent, alors que les éléments d'un tableau doivent tous être du même type. De plus, les tuples n'autorisent que les constantes comme indices, comme `t.4`. Vous ne pouvez pas écrire `t.i` ou `t[i]` pour obtenir le `i`ème élément.

Le code Rust utilise souvent des types tuple pour renvoyer plusieurs valeurs à partir d'une fonction. Par exemple, la `split_at` méthode sur les tranches de chaîne, qui divise une chaîne en deux moitiés et les renvoie toutes les deux, est déclarée comme ceci :

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

Le type de retour `(&str, &str)` est un tuple de deux tranches de chaîne. Vous pouvez utiliser la syntaxe de correspondance de modèle pour affecter chaque élément de la valeur de retour à une variable différente :

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

C'est plus lisible que l'équivalent :

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Vous verrez également des tuples utilisés comme une sorte de type de structure à drame minimal. Par exemple, dans le programme Mandelbrot du [chapitre 2](#), nous devions transmettre la largeur et la hauteur de l'image aux fonctions qui la tracent et l'écrivent sur le disque. Nous pourrions déclarer une structure avec `width` et `height` membres, mais c'est une notation assez lourde pour quelque chose d'aussi évident, donc nous avons juste utilisé un tuple :

```
/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
```

```
-> Result<(), std::io::Error>
{ ... }
```

Le type du `bounds` paramètre est `(usize, usize)`, un tuple de deux `usize` valeurs. Certes, nous pourrions tout aussi bien écrire des paramètres `width` et séparés `height`, et le code machine serait à peu près le même dans les deux cas. C'est une question de clarté. Nous considérons la taille comme une valeur, pas deux, et l'utilisation d'un tuple nous permet d'écrire ce que nous voulons dire.

L'autre type de tuple couramment utilisé est le zéro-tuple `()`. Ceci est traditionnellement appelé le *type d'unité* car il n'a qu'une seule valeur, également écrite `()`. Rust utilise le type d'unité où il n'y a pas de valeur significative à transporter, mais le contexte nécessite néanmoins une sorte de type.

Par exemple, une fonction qui ne renvoie aucune valeur a un type de retour de `()`. La `std::mem::swap` fonction de la bibliothèque standard n'a pas de valeur de retour significative ; il échange juste les valeurs de ses deux arguments. La déclaration pour `std::mem::swap` se lit comme suit :

```
fn swap<T>(x: &mut T, y: &mut T);
```

Les `<T>` moyens c'est générique `swap` : vous pouvez l'utiliser sur des références à des valeurs de n'importe quel type `T`. Mais la signature omet `swap` complètement le type de retour de `,` qui est un raccourci pour renvoyer le type d'unité :

```
fn swap<T>(x: &mut T, y: &mut T) ->();
```

De même, l'`write_image` exemple que nous avons mentionné précédemment a un type de retour de `Result<(), std::io::Error>`, ce qui signifie que la fonction renvoie une `std::io::Error` valeur si quelque chose ne va pas, mais ne renvoie aucune valeur en cas de succès.

Si vous le souhaitez, vous pouvez inclure une virgule après le dernier élément d'un tuple : les types `(&str, i32,)` et `(&str, i32)` sont équivalents, tout comme les expressions `("Brazil", 1985,)` et `("Brazil", 1985)`. Rust autorise systématiquement une virgule de fin supplémentaire partout où des virgules sont utilisées : arguments de fonction, tableaux, définitions de structure et d'énumération, etc. Cela peut sembler

étrange aux lecteurs humains, mais cela peut faciliter la lecture des différences lorsque des entrées sont ajoutées et supprimées à la fin d'une liste.

Par souci de cohérence, il existe même des tuples qui contiennent une seule valeur. Le littéral ("lonely hearts",) est un tuple contenant une seule chaîne ; son type est (&str,) . Ici, la virgule après la valeur est nécessaire pour distinguer le tuple singleton d'une simple expression entre parenthèses.

Types de pointeur

Rust a plusieurs types qui représentent des adresses mémoire.

c'est une grande différence entre Rust et la plupart des langages avec ramasse-miettes. En Java, si `class Rectangle` contient un champ `Vector2D upperLeft;`, alors `upperLeft` est une référence à un autre `Vector2D` objet créé séparément. Les objets ne contiennent jamais physiquement d'autres objets en Java.

La rouille est différente. Le langage est conçu pour aider à maintenir les allocations au minimum. Les valeurs s'imbriquent par défaut. La valeur `((0, 0), (1440, 900))` est stockée sous la forme de quatre nombres entiers adjacents. Si vous le stockez dans une variable locale, vous avez une variable locale large de quatre entiers. Rien n'est alloué dans le tas.

C'est excellent pour l'efficacité de la mémoire, mais par conséquent, lorsqu'un programme Rust a besoin de valeurs pour pointer vers d'autres valeurs, il doit utiliser explicitement les types de pointeur. La bonne nouvelle est que les types de pointeurs utilisés dans Rust sécurisé sont contraints d'éliminer les comportements indéfinis, de sorte que les pointeurs sont beaucoup plus faciles à utiliser correctement dans Rust qu'en C++.

Nous aborderons ici trois types de pointeurs : les références, les boîtes et les pointeurs non sécurisés.

Références

Une valeur de type `&string` (prononcé "ref String") est une référence à une `String` valeur, `a &i32` est une référence à un `i32`, et ainsi de suite.

Il est plus facile de commencer en considérant les références comme le type de pointeur de base de Rust. Au moment de l'exécution, une réfé-

rence à un `i32` est un seul mot machine contenant l'adresse du `i32`, qui peut être sur la pile ou dans le tas. L'expression `&x` produit une référence à `x`; dans la terminologie Rust, nous disons qu'il *emprunte une référence* à `x`. Étant donné une référence `r`, l'expression `*r` fait référence à la valeur `r` vers laquelle pointe. Ils ressemblent beaucoup aux opérateurs `&` et `*` en C et C++. * Et comme un pointeur C, une référence ne libère pas automatiquement de ressources lorsqu'elle sort de la portée.

Contrairement aux pointeurs C, cependant, les références Rust ne sont jamais nulles : il n'y a tout simplement aucun moyen de produire une valeur nulle.référence en toute sécurité Rust. Et contrairement à C, Rust suit la propriété et la durée de vie des valeurs, de sorte que les erreurs telles que les pointeurs perdus, les doubles libérations et l'invalidation des pointeurs sont exclues au moment de la compilation.

Les références de rouille se déclinent en deux saveurs :

`&T`

Un immuable, référence partagée. Vous pouvez avoir plusieurs références partagées à une valeur donnée à la fois, mais elles sont en lecture seule : la modification de la valeur vers laquelle elles pointent est interdite, comme `const T*` en C.

`&mut T`

Une mutable, référence exclusive. Vous pouvez lire et modifier la valeur vers laquelle elle pointe, comme avec `a T*` en C. Mais tant que la référence existe, vous ne pouvez avoir aucune autre référence d'aucune sorte à cette valeur. En fait, la seule façon d'accéder à la valeur est de passer par la référence mutable.

Rust utilise cette dichotomie entre les références partagées et mutables pour appliquer une règle « auteur unique *ou* lecteurs multiples » : soit vous pouvez lire et écrire la valeur, soit elle peut être partagée par n'importe quel nombre de lecteurs, mais jamais les deux en même temps. Cette séparation, renforcée par des vérifications au moment de la compilation, est au cœur des garanties de sécurité de Rust. [Le chapitre 5](#) explique les règles de Rust pour une utilisation sûre des références.

Des boites

Le plus simplefaçon d'allouer une valeur dans le tas est d'utiliser

`Box::new:`

```
let t = (12, "eggs");
let b = Box::new(t); // allocate a tuple in the heap
```

Le type de `t` est `(i32, &str)`, donc le type de `b` est `Box<(i32, &str)>`. L'appel à `Box::new` alloue suffisamment de mémoire pour contenir le tuple sur le tas. Lorsqu'il `b` sort de la portée, la mémoire est immédiatement libérée, sauf si `b` elle a été *déplacée*, en la retournant, par exemple. Les mouvements sont essentiels à la façon dont Rust gère les valeurs allouées par tas ; nous expliquons tout cela en détail au [chapitre 4](#).

Pointeurs bruts

Rouillera également les types de pointeurs bruts `*mut T` et `*const T`. Les pointeurs bruts sont vraiment comme les pointeurs en C++. L'utilisation d'un pointeur brut n'est pas sûre, car Rust ne fait aucun effort pour suivre ce vers quoi il pointe. Par exemple, les pointeurs bruts peuvent être nuls, ou ils peuvent pointer vers de la mémoire qui a été libérée ou qui contient maintenant une valeur d'un type différent. Toutes les erreurs de pointage classiques du C++ sont proposées pour votre plus grand plaisir.

Cependant, vous ne pouvez que déréférencer pointeurs bruts dans un `unsafe` bloc. Un `unsafe` bloc est le mécanisme d'acceptation de Rust pour les fonctionnalités de langage avancées dont la sécurité dépend de vous. Si votre code n'a pas de `unsafe` blocs (ou si ceux qu'il contient sont écrits correctement), alors les garanties de sécurité sur lesquelles nous insistons tout au long de ce livre sont toujours valables. Pour plus de détails, reportez-vous au [chapitre 22](#).

Tableaux, vecteurs et tranches

Rust a trois types pour représenter une séquence de valeurs en mémoire:

- Le type `[T; N]` représente un tableau de `N` valeurs, chacune de type `T`. La taille d'un tableau est une constante déterminée au moment de la compilation et fait partie du type ; vous ne pouvez pas ajouter de nouveaux éléments ou réduire un tableau.
- Le type `Vec<T>`, appelé *vecteur de Ts*, est une séquence de valeurs de type évolutive allouée dynamiquement `T`. Les éléments d'un vecteur vivent sur le tas, vous pouvez donc redimensionner les vecteurs à volonté.

lonté : mettez-y de nouveaux éléments, ajoutez-y d'autres vecteurs, supprimez des éléments, etc.

- Les types `&[T]` et `&mut [T]`, appelés *partagetranche de* `T`s et *mutabletranche de* `T`s, sont des références à une série d'éléments qui font partie d'une autre valeur, comme un tableau ou un vecteur. Vous pouvez considérer une tranche comme un pointeur vers son premier élément, avec un décompte du nombre d'éléments auxquels vous pouvez accéder à partir de ce point. Une tranche modifiable `&mut [T]` vous permet de lire et de modifier des éléments, mais ne peut pas être partagée ; une tranche partagée `&[T]` permet de partager l'accès entre plusieurs lecteurs, mais ne permet pas de modifier les éléments.

Étant donné une valeur `v` de l'un de ces trois types, l'expression `v.len()` donne le nombre d'éléments dans `v` et `v[i]` fait référence au `i`ème élément de `v`. Le premier élément est `v[0]`, et le dernier élément est `v[v.len() - 1]`. Contrôles de rouille qui `i` se situent toujours dans cette plage ; si ce n'est pas le cas, l'expression panique. La longueur de `v` peut être zéro, auquel cas toute tentative d'indexation paniquera. `i` doit être une `usize` valeur ; vous ne pouvez pas utiliser d'autre type entier comme index.

Tableaux

Il existe plusieurs façons d'écrire un tableau de valeurs. Le plus simple est d'écrire une suite de valeurs entre crochets :

```
let lazy_caterer:[u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

Pour le cas courant d'un long tableau rempli d'une certaine valeur, vous pouvez écrire `, où est la valeur que chaque élément doit avoir et est la longueur. Par exemple, est un tableau de 10 000 éléments, tous définis sur : [V; N] V N [true; 10000] bool true`

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}
```

```
        }
    }

    assert!(sieve[211]);
    assert!(!sieve[9876]);
}
```

Vous verrez cette syntaxe utilisée pour les tampons de taille fixe : [0u8; 1024] peut être un tampon d'un kilo-octet, rempli de zéros. Rust n'a pas de notation pour un tableau non initialisé. (En général, Rust garantit que le code ne peut jamais accéder à une sorte de valeur non initialisée.)

La longueur d'un tableau fait partie de son type et est fixée au moment de la compilation. Si `n` est une variable, vous ne pouvez pas écrire [`true`; `n`] pour obtenir un tableau d' `n` éléments. Lorsque vous avez besoin d'un tableau dont la longueur varie au moment de l'exécution (et vous le faites généralement), utilisez plutôt un vecteur.

Les méthodes utiles que vous aimeriez voir sur les tableaux (itération sur les éléments, recherche, tri, remplissage, filtrage, etc.) sont toutes fournies en tant que méthodes sur les tranches, et non sur les tableaux. Mais Rust convertit implicitement une référence à un tableau en une tranche lors de la recherche de méthodes, vous pouvez donc appeler directement n'importe quelle méthode de tranche sur un tableau :

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

Ici, la `sort` méthode est en fait définie sur des tranches, mais puisqu'elle prend son opérande par référence, Rust produit implicitement une `&mut [i32]` tranche faisant référence à l'ensemble du tableau et la transmet à `sort` pour opérer dessus. En fait, la `len` méthode que nous avons mentionnée précédemment est également une méthode de tranche. Nous couvrons les tranches plus en détail dans "["Slices"](#)" .

Vecteurs

Un vecteur `Vec<T>` est un tableau redimensionnable d'éléments de type `T`, alloués sur le tas.

Il existe plusieurs façons de créer des vecteurs. Le plus simple est d'utiliser la `vec!` macro, ce qui nous donne une syntaxe pour les vecteurs qui

ressemble beaucoup à un littéral de tableau :

```
let mut primes = vec![2, 3, 5, 7];
assert_eq!(primes.iter().product::<i32>(), 210);
```

Mais bien sûr, il s'agit d'un vecteur, pas d'un tableau, nous pouvons donc y ajouter des éléments dynamiquement :

```
primes.push(11);
primes.push(13);
assert_eq!(primes.iter().product::<i32>(), 30030);
```

Vous pouvez également créer un vecteur en répétant une valeur donnée un certain nombre de fois, toujours en utilisant une syntaxe qui imite les littéraux de tableau :

```
fn new_pixel_buffer(rows: usize, cols: usize) ->Vec<u8> {
    vec![0; rows * cols]
}
```

La `vec!` macro équivaut à appeler `Vec::new` pour créer un nouveau vecteur vide, puis à pousser les éléments dessus, ce qui est un autre idiome :

```
let mut pal = Vec::new();
pal.push("step");
pal.push("on");
pal.push("no");
pal.push("pets");
assert_eq!(pal, vec!["step", "on", "no", "pets"]);
```

Une autre possibilité est de construire un vecteur à partir des valeurs produites par un itérateur :

```
let v:Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

Vous aurez souvent besoin de fournir le type lors de l'utilisation `collect` (comme nous l'avons fait ici), car il peut créer de nombreux types de collections, pas seulement des vecteurs. En spécifiant le type de `v`, nous avons précisé le type de collection que nous voulons.

Comme pour les tableaux, vous pouvez utiliser les méthodes slice sur les vecteurs :

```
// A palindrome!
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
palindrome.reverse();
// Reasonable yet disappointing:
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a man"]);
```

Ici, la `reverse` méthode est en fait défini sur les tranches, mais l'appel emprunte implicitement une `&mut [&str]` tranche au vecteur et l'invoque `reverse` sur celle-ci.

`Vec` est un type essentiel à Rust - il est utilisé presque partout où l'on a besoin d'une liste de taille dynamique - il existe donc de nombreuses autres méthodes qui construisent de nouveaux vecteurs ou étendent ceux qui existent déjà. Nous les aborderons au [chapitre 16](#).

A `Vec<T>` se compose de trois valeurs: un pointeur vers le tampon alloué par `tas` pour les éléments, qui est créé et détenu par le `Vec<T>` ; le nombre d'éléments que la mémoire tampon a la capacité de stocker ; et le nombre qu'il contient actuellement (en d'autres termes, sa longueur). Lorsque le tampon a atteint sa capacité, ajouter un autre élément au vecteur implique d'allouer un tampon plus grand, d'y copier le contenu actuel, de mettre à jour le pointeur du vecteur et sa capacité à décrire le nouveau tampon, et enfin de libérer l'ancien.

Si vous connaissez le nombre d'éléments dont un vecteur aura besoin à l'avance, au lieu de `Vec::new` vous pouvez appeler `Vec::with_capacity` pour créer un vecteur avec un tampon suffisamment grand pour les contenir tous, dès le début ; ensuite, vous pouvez ajouter les éléments au vecteur un par un sans provoquer de réallocation. La `vec!` macro utilise une astuce comme celle-ci, car elle sait combien d'éléments le vecteur final aura. Notez que cela n'établit que la taille initiale du vecteur ; si vous dépassez votre estimation, le vecteur agrandit simplement son stockage comme d'habitude.

De nombreuses fonctions de bibliothèque recherchent la possibilité d'utiliser à la `Vec::with_capacity` place de `Vec::new`. Par exemple, dans l'`collect` exemple, l'itérateur `0..5` sait à l'avance qu'il donnera cinq valeurs, et la `collect` fonction en profite pour pré-allouer le vecteur qu'il renvoie avec la capacité correcte. Nous verrons comment cela fonctionne au [chapitre 15](#).

`len` Tout comme la méthode d'un vecteur renvoie le nombre d'éléments qu'il contient maintenant, sa `capacity` méthode renvoie le nombre d'éléments qu'il pourrait contenir sans réallocation :

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// Typically prints "capacity is now 4":
println!("capacity is now {}", v.capacity());
```

La capacité imprimée à la fin n'est pas garantie d'être exactement 4, mais elle sera d'au moins 3, puisque le vecteur contient trois valeurs.

Vous pouvez insérer et supprimer des éléments où vous le souhaitez dans un vecteur, bien que ces opérations déplacent tous les éléments après la position affectée vers l'avant ou vers l'arrière, elles peuvent donc être lentes si le vecteur est long :

```
let mut v = vec![10, 20, 30, 40, 50];

// Make the element at index 3 be 35.
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// Remove the element at index 1.
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

Vous pouvez utiliser la `pop` méthode pour supprimer le dernier élément et le renvoyer. Plus précisément, extraire une valeur de a `Vec<T>` renvoie un `Option<T>` : `None` si le vecteur était déjà vide, ou `Some(v)` si son dernier élément avait été `v` :

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

Vous pouvez utiliser une `for` boucle pour parcourir un vecteur :

```
// Get our command-line arguments as a vector of Strings.  
let languages: Vec<String> = std::env::args().skip(1).collect();  
for l in languages {  
    println!("{}: {}", l,  
            if l.len() % 2 == 0 {  
                "functional"  
            } else {  
                "imperative"  
            });  
}
```

L'exécution de ce programme avec une liste de langages de programmation est éclairante :

```
$ cargo run Lisp Schéma C C++ Fortran  
Compiling proglangs v0.1.0 (/home/jimb/rust/proglangs)  
Finished dev [unoptimized + debuginfo] target(s) in 0.36s  
Running `target/debug/proglangs Lisp Scheme C C++ Fortran`  
Lisp: functional  
Scheme: functional  
C: imperative  
C++: imperative  
Fortran: imperative  
$
```

Enfin, une définition satisfaisante du terme *langage fonctionnel*.

Malgré son rôle fondamental, `Vec` est un type ordinaire défini dans Rust, non intégré au langage. Nous couvrirons les techniques nécessaires pour implémenter de tels types au [chapitre 22](#).

Tranches

Une tranche, écrit `[T]` sans spécifier la longueur, est une région d'un tableau ou vecteur. Étant donné qu'une tranche peut avoir n'importe quelle longueur, les tranches ne peuvent pas être stockées directement dans des variables ou transmises en tant qu'arguments de fonction. Les tranches sont toujours passées par référence.

Une référence à une tranche est un *pointeur gras*: une valeur de deux mots comprenant un pointeur vers le premier élément de la tranche et le nombre d'éléments dans la tranche.

Supposons que vous exécutiez le code suivant :

```
let v: Vec<f64> = vec![0.0, 0.707, 1.0, 0.707];
let a:[f64; 4] = [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa:&[f64] = &a;
```

Dans les deux dernières lignes, Rust convertit automatiquement la `&Vec<f64>` référence et la `&[f64; 4]` référence en références de tranche qui pointent directement vers les données.

À la fin, la mémoire ressemble à la [figure 3-2](#).

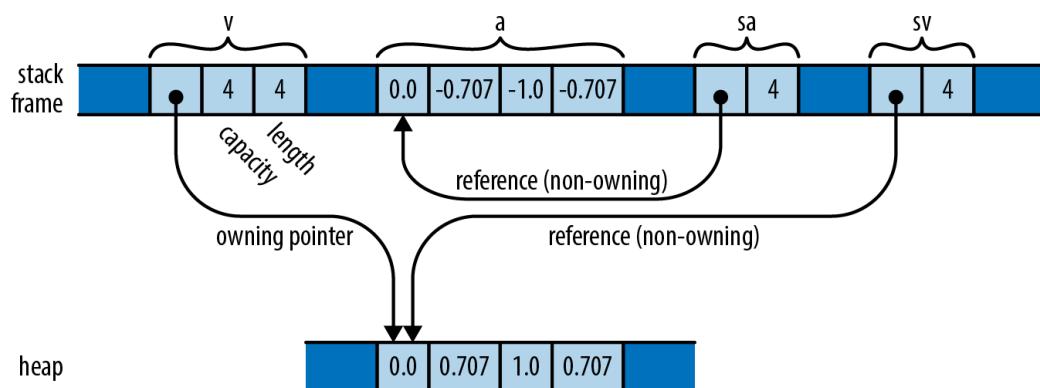


Illustration 3-2. Un vecteur `v` et un tableau `a` en mémoire, avec des tranches `sa` et `sv` se référant à chacun

Alors qu'une référence ordinaire est un pointeur non propriétaire vers une valeur unique, une référence à une tranche est un pointeur non propriétaire vers une plage de valeurs consécutives en mémoire. Cela fait des références de tranche un bon choix lorsque vous souhaitez écrire une fonction qui opère sur un tableau ou un vecteur. Par exemple, voici une fonction qui imprime une tranche de nombres, un par ligne :

```
fn print(n:&[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&a); // works on arrays
print(&v); // works on vectors
```

Étant donné que cette fonction prend une référence de tranche comme argument, vous pouvez l'appliquer à un vecteur ou à un tableau, comme indiqué. En fait, de nombreuses méthodes que vous pourriez considérer

comme appartenant à des vecteurs ou à des tableaux sont des méthodes définies sur des tranches : par exemple, les méthodes `sort` et `reverse`, qui trient ou inversent une séquence d'éléments en place, sont en fait des méthodes sur le type de tranche `[T]`.

Vous pouvez obtenir une référence à une tranche d'un tableau ou d'un vecteur, ou à une tranche d'une tranche existante, en l'indexant avec une plage :

```
print(&v[0..2]);      // print the first two elements of v
print(&a[2..]);      // print elements of a starting with a[2]
print(&sv[1..3]);    // print v[1] and v[2]
```

Comme pour les accès aux tableaux ordinaires, Rust vérifie que les indices sont valides. Essayer d'emprunter une tranche qui s'étend au-delà de la fin des données entraîne une panique.

Étant donné que les tranches apparaissent presque toujours derrière les références, nous nous référerons souvent à des types comme `&[T]` ou `&str` en tant que "tranches", en utilisant le nom le plus court pour le concept le plus courant..

Types de chaînes

Programmeurs familier avec C++ se souviendra qu'il existe deux types de chaînes dans le langage. Les littéraux de chaîne ont le type de pointeur `const char *`. La bibliothèque standard propose également une classe, `std::string`, pour créer dynamiquement des chaînes au moment de l'exécution.

Rust a un design similaire. Dans cette section, nous montrerons toutes les façons d'écrire des littéraux de chaîne, puis nous présenterons les deux types de chaîne de Rust. Nous fournissons plus de détails sur les chaînes et la gestion du texte au [chapitre 17](#).

Littéraux de chaîne

Chaîne de caractères les littéraux sont entourés de guillemets doubles. Ils utilisent les mêmes séquences d'échappement antislash que les `char` littéraux :

```
let speech = "\\"Ouch!\\\" said the well.\n";
```

Dans les littéraux de chaîne, contrairement aux `char` littéraux, les guillemets simples n'ont pas besoin d'une barre oblique inverse, contrairement aux guillemets doubles.

Une chaîne peut s'étendre sur plusieurs lignes :

```
println!("In the room the women come and go,  
        Singing of Mount Abora");
```

Le caractère de saut de ligne dans ce littéral de chaîne est inclus dans la chaîne et donc dans la sortie. Il en va de même pour les espaces au début de la deuxième ligne.

Si une ligne d'une chaîne se termine par une barre oblique inverse, le caractère de saut de ligne et l'espace de tête sur la ligne suivante sont supprimés :

```
println!("It was a bright, cold day in April, and \  
        there were four of us—\  
        more or less.");
```

Cela imprime une seule ligne de texte. La chaîne contient un seul espace entre "et" et "là" car il y a un espace avant la barre oblique inverse dans le programme, et aucun espace entre le tiret cadratin et "plus".

Dans quelques cas, la nécessité de doubler chaque antislash dans une chaîne est une nuisance. (Les exemples classiques sont les expressions régulières et les chemins Windows.) Pour ces cas, Rust propose *des chaînes brutes*. Une chaîne brute est étiquetée avec la lettre minuscule `r`. Toutes les barres obliques inverses et les espaces blancs à l'intérieur d'une chaîne brute sont inclus textuellement dans la chaîne. Aucune séquence d'échappement n'est reconnue :

```
let default_win_install_path = r"C:\Program Files\Gorillas";  
  
let pattern = Regex::new(r"\d+(\.\d+)*");
```

Vous ne pouvez pas inclure un guillemet double dans une chaîne brute simplement en mettant une barre oblique inverse devant - rappelez-vous, nous avons dit *qu'aucune* séquence d'échappement n'est reconnue. Cependant, il existe également un remède à cela. Le début et la fin d'une chaîne brute peuvent être marqués par des signes dièse :

```
println!(r###"  
    This raw string started with 'r###' .  
    Therefore it does not end until we reach a quote mark ('')  
    followed immediately by three pound signs ('##'):   
"##");
```

Vous pouvez ajouter aussi peu ou autant de signes dièse que nécessaire pour indiquer clairement où se termine la chaîne brute.

Chaînes d'octets

Un string littéral avec le `b` préfixe est une *chaîne d'octets*. Une telle chaîne est une tranche de `u8` valeurs, c'est-à-dire des octets, plutôt qu'un texte Unicode :

```
let method = b"GET";  
assert_eq!(method, &[b'G', b'E', b'T']);
```

Le type de `method` est `&[u8; 3]` : c'est une référence à un tableau de trois octets. Il n'a aucune des méthodes de chaîne dont nous parlerons dans une minute. La chose qui ressemble le plus à une chaîne est la syntaxe que nous avons utilisée pour l'écrire.

Les chaînes d'octets peuvent utiliser toutes les autres syntaxes de chaîne que nous avons présentées : elles peuvent s'étendre sur plusieurs lignes, utiliser des séquences d'échappement et utiliser des barres obliques inverses pour joindre des lignes. Les chaînes d'octets brutes commencent par `br"`.

Les chaînes d'octets ne peuvent pas contenir de caractères Unicode arbitraires. Ils doivent se contenter d'ASCII et de `\xHH` séquences d'échappement.

Chaînes en mémoire

Rouiller Les chaînes sont des séquences de caractères Unicode, mais elles ne sont pas stockées en mémoire sous forme de tableaux de `char`s. Au lieu de cela, ils sont stockés en utilisant UTF-8, un codage à largeur variable. Chaque caractère ASCII d'une chaîne est stocké dans un octet. Les autres caractères occupent plusieurs octets.

[La figure 3-3](#) montre les valeurs `String` et `&str` créées par le code suivant :

```

let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ø_ø";

```

A `String` a un tampon redimensionnable contenant du texte UTF-8. Le tampon est alloué sur le tas, il peut donc redimensionner son tampon selon les besoins ou à la demande. Dans l'exemple, `noodles` est a `String` qui possède un tampon de huit octets, dont sept sont en cours d'utilisation. Vous pouvez considérer a `String` comme un `Vec<u8>` qui est garanti pour contenir un UTF-8 bien formé ; en fait, c'est ainsi qu'il `String` est mis en œuvre.

A `&str` (prononcé « stir » ou « string slice ») est une référence à une série de texte UTF-8 appartenant à quelqu'un d'autre : il "emprunte" le texte. Dans l'exemple, `oodles` est une `&str` référence aux six derniers octets du texte appartenant à `noodles`, il représente donc le texte "oodles". Comme les autres références de tranche, a `&str` est un pointeur gras, contenant à la fois l'adresse des données réelles et leur longueur. Vous pouvez considérer a `&str` comme n'étant rien de plus qu'un `&[u8]` qui est garanti pour contenir un UTF-8 bien formé.

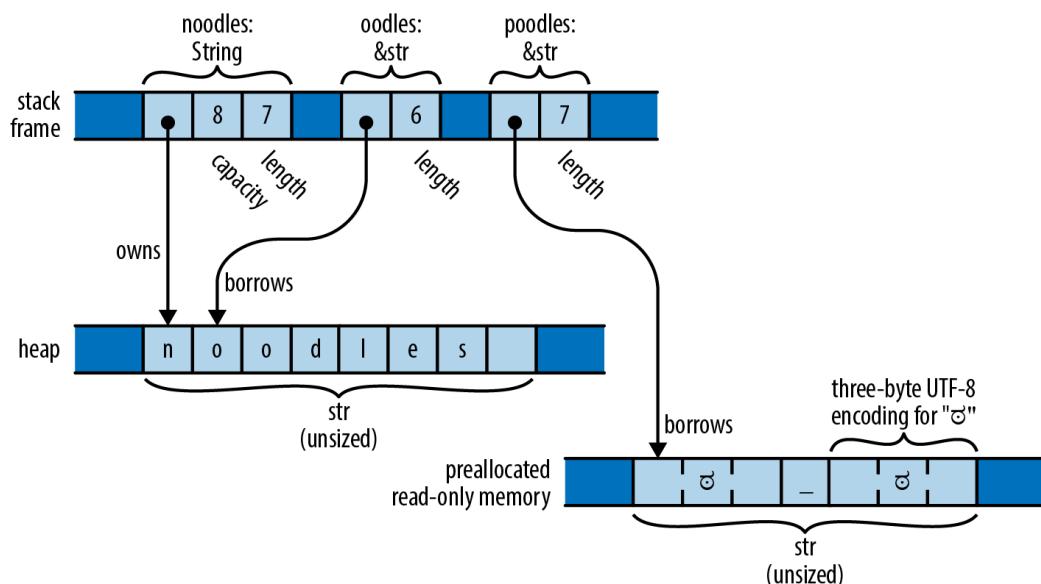


Illustration 3-3. `String`, `&str`, et `str`

Un littéral de chaîne est un `&str` qui fait référence à du texte préalloué, généralement stocké dans une mémoire en lecture seule avec le code machine du programme. Dans l'exemple précédent, `poodles` est un littéral de chaîne, pointant vers sept octets qui sont créés lorsque le programme commence l'exécution et qui durent jusqu'à ce qu'il se termine.

Méthode A `String` ou `&str_.len()` renvoie sa longueur. La longueur est mesurée en octets, pas en caractères :

```
assert_eq!("ø_ø".len(), 7);
assert_eq!("ø_ø".chars().count(), 3);
```

Il est impossible de modifier un `&str`:

```
let mut s = "hello";
s[0] = 'c';      // error: `&str` cannot be modified, and other reasons
s.push('\n');   // error: no method named `push` found for reference `&str`
```

Pour créer de nouvelles chaînes au moment de l'exécution, utilisez `String`.

Le type `&mut str` existe, mais il n'est pas très utile, car presque toutes les opérations sur UTF-8 peuvent modifier sa longueur globale en octets, et une tranche ne peut pas réallouer son référent. En fait, les seules opérations disponibles sur `&mut str` sont `make_ascii_uppercase` et `make_ascii_lowercase`, qui modifient le texte en place et n'affectent que les caractères à un octet, par définition.

Chaîne de caractères

`&str` ressemble beaucoup à `&[T]` : un gros pointeur vers des données. `String` est analogue à `Vec<T>`, comme décrit dans le [Tableau 3-11](#).

Tableau 3-11. Vec <T> et string comparaison

	Vec<T>	Chaîne de caractères
Libère automatiquement les tampons	Oui	Oui
Cultivable	Oui	Oui
<code>::new()</code> et <code>::with_capacity()</code> fonctions associées au type	Oui	Oui
<code>.reserve()</code> et <code>.capacity()</code> méthodes	Oui	Oui
<code>.push()</code> et <code>.pop()</code> méthodes	Oui	Oui
Syntaxe de plage <code>v[start..stop]</code>	Oui, retours & [T]	Oui, retours & <code>str</code>
Conversion automatique	<code>&Vec<T></code> à <code>&[T]</code>	<code>&String</code> à & <code>str</code>
Hérite des méthodes	De <code>&[T]</code>	De <code>&str</code>

Comme un `Vec`, chacun `String` a son propre tampon alloué par `tas` qui n'est partagé avec aucun autre `String`. Lorsqu'une `String` variable sort de la portée, le tampon est automatiquement libéré, sauf si `String` elle a été déplacée.

Il existe plusieurs façons de créer des `String`s :

- La `.to_string()` méthode convertit un `&str` à un `String`. Cela copie la chaîne :

```
let error_message = "too many pets".to_string();
```

La `.to_owned()` méthode fait la même chose, et vous pouvez le voir utilisé de la même manière. Cela fonctionne également pour d'autres types, comme nous le verrons au [chapitre 13](#).

- La `format!()` macrofonctionne comme `println!()`, saufqu'il renvoie un nouveau `String` au lieu d'écrire du texte sur `stdout`, et qu'il n'ajoute pas automatiquement une nouvelle ligne à la fin :

```
assert_eq!(format!("{}{:02}{:02}"N, 24, 5, 23),
           "24°05'23"N".to_string());
```

- Les tableaux, les tranches et les vecteurs de chaînes ont deux méthodes, `.concat()` et `.join(sep)`, qui forment un nouveau `String` à partir de plusieurs chaînes :

```
let bits = vec![ "veni", "vidi", "vici"];
assert_eq!(bits.concat(), "venividivici");
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

Le choix se pose parfois du type à utiliser : `&str` ou `String`. [Le chapitre 5](#) aborde cette question en détail. Pour l'instant, il suffira de souligner que `a &str` peut faire référence à n'importe quelle tranche de n'importe quelle chaîne, qu'il s'agisse d'un littéral de chaîne(stocké dans l'exécutable) ou a `String` (alloué et libéré à l'exécution). Cela signifie qu'il `&str` est plus approprié pour les arguments de fonction lorsque l'appelant doit être autorisé à transmettre l'un ou l'autre type de chaîne.

Utilisation de chaînes

Les chaînes prennent en charge les opérateurs `==` et `!=`. Deux chaînes sont égales si elles contiennent les mêmes caractères dans le même ordre (qu'elles pointent ou non vers le même emplacement en mémoire) :

```
assert!("ONE".to_lowercase() == "one");
```

Les chaînes prennent également en charge la comparaisonopérateurs `<`, `<=`, `>`, et `>=`, ainsi que de nombreuses méthodes et fonctions utiles que vous pouvez trouver dans la documentation en ligne sous « `str` (type primitif) » ou le `std::str` module « » (ou passez simplement au [chapitre 17](#)). Voici quelques exemples:

```
assert!("peanut".contains("nut"));
assert_eq!("θ_θ".replace("θ", "█"), "█_█");
assert_eq!("      clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
```

```
    assert!(word.starts_with("v"));
}
```

Gardez à l'esprit que, compte tenu de la nature d'Unicode, une simple `char` comparaison par `char` comparaison ne donne *pas* toujours les réponses attendues. Par exemple, les chaînes Rust "`th\u{e9}`" et "`the\u{301}`" sont toutes deux des représentations Unicode valides pour thé, le mot français pour thé. Unicode indique qu'ils doivent être affichés et traités de la même manière, mais Rust les traite comme deux chaînes complètement distinctes. De même, les opérateurs de commande de Rust < utilisent un ordre lexicographique simple basé sur des valeurs de points de code de caractères. Cet ordre ne ressemble que parfois à l'ordre utilisé pour le texte dans la langue et la culture de l'utilisateur. Nous abordons ces questions plus en détail au [chapitre 17](#).

Autres types de type chaîne

Rust garantit que les chaînes sont valides UTF-8. Parfois, un programme doit vraiment être capable de gérer des chaînes qui ne sont *pas* valides en Unicode. Cela se produit généralement lorsqu'un programme Rust doit interagir avec un autre système qui n'applique pas de telles règles. Par exemple, dans la plupart des systèmes d'exploitation, il est facile de créer un fichier avec un nom de fichier qui n'est pas Unicode valide. Que doit-il se passer lorsqu'un programme Rust rencontre ce type de nom de fichier ?

La solution de Rust consiste à proposer quelques types de type chaîne pour ces situations :

- Tenez-vous en à `String` et `&str` pour le texte Unicode.
- Lorsque vous travaillez avec des noms de fichiers, utilisez `std::path::PathBuf` et à la `&Path` place.
- Lorsque vous travaillez avec des données binaires qui ne sont pas du tout encodées en UTF-8, utilisez `Vec<u8>` et `&[u8]`.
- Lorsque vous travaillez avec des noms de variables d'environnement et des arguments de ligne de commande sous la forme native présentée par le système d'exploitation, utilisez `OsString` et `&OsStr`.
- Lors de l'interopérabilité avec des bibliothèques C qui utilisent des chaînes terminées par un caractère nul, utilisez `std::ffi::CString` et `&CStr`.

Tapez les alias

Le `type` mot clé peut être utilisé comme `typedef` en C++ pour déclarer un nouveau nom pour un type existant :

```
type Bytes = Vec<u8>;
```

Le type `Bytes` que nous déclarons ici est un raccourci pour ce type particulier de `Vec` :

```
fn decode(data:&Bytes) {  
    ...  
}
```

Au-delà des bases

Les types sont une partie centrale de Rust. Nous continuerons à parler des types et à en introduire de nouveaux tout au long du livre. En particulier, les types définis par l'utilisateur de Rust donnent au langage une grande partie de sa saveur, car c'est là que les méthodes sont définies. Il existe trois types de types définis par l'utilisateur, et nous les aborderons dans trois chapitres successifs : les structures au [chapitre 9](#), les énumérations au [chapitre 10](#) et les traits au [chapitre 11](#).

Les fonctions et les fermetures ont leurs propres types, traités au [chapitre 14](#). Et les types qui composent la bibliothèque standard sont couverts tout au long du livre. Par exemple, le [chapitre 16](#) présente les types de collection standard.

Tout cela devra cependant attendre. Avant de poursuivre, il est temps d'aborder les concepts qui sont au cœur de la sécurité de Rustrègles.

Soutien Se déconnecter

Chapitre 4. Propriété et déménagements

Quand il s'agit de gérer la mémoire, il y a deux caractéristiques que nous aimerais avoir dans nos langages de programmation :

- Nous aimerais la mémoire être libérée rapidement, au moment de notre choix. Cela nous donne le contrôle sur la consommation de mémoire du programme.
- Nous ne voulons jamais utiliser un pointeur vers un objet après qu'il a été libéré. Ce serait un comportement indéfini, entraînant des plantages et des failles de sécurité.

Mais ceux-ci semblent s'exclure mutuellement : libérer une valeur alors que des pointeurs existent sur celle-ci laisse nécessairement ces pointeurs en suspens. Presque tous les principaux langages de programmation appartiennent à l'un des deux camps, en fonction de laquelle des deux qualités ils renoncent :

- Le camp "Safety First" utilise la collecte des ordures pour gérer la mémoire, en libérant automatiquement les objets lorsque tous les pointeurs accessibles vers eux ont disparu. Cela élimine les pointeurs pendents en gardant simplement les objets autour jusqu'à ce qu'il ne reste plus de pointeurs vers eux. Presque tous les langages modernes appartiennent à ce camp, de Python, JavaScript et Ruby à Java, C# et Haskell. Mais s'appuyer sur le ramasse-miettes signifie renoncer au contrôle sur le moment exact où les objets sont remis au collecteur. En général, les ramasse-miettes sont des bêtes surprises, et comprendre pourquoi la mémoire n'a pas été libérée quand on s'y attendait peut être un défi.
- Le camp "Control First" vous laisse le soin de libérer la mémoire. La consommation de mémoire de votre programme est entièrement entre vos mains, mais éviter les pointeurs pendents devient également entièrement votre préoccupation. C et C++ sont les seuls langages traditionnels de ce camp.

C'est très bien si vous ne faites jamais d'erreurs, mais les preuves suggèrent que vous finirez par le faire. L'utilisation abusive de pointeurs a été un coupable courant dans les problèmes de sécurité signalés aussi longtemps que ces données ont été collectées.

Rust vise à être à la fois sûr et performant, donc aucun de ces compromis n'est acceptable. Mais si la réconciliation était facile, quelqu'un l'aurait fait bien avant maintenant. Quelque chose de fondamental doit changer.

Rust sort de l'impasse d'une manière surprenante : en limitant la façon dont vos programmes peuvent utiliser des pointeurs. Ce chapitre et le suivant sont consacrés à expliquer exactement ce que sont ces restrictions et pourquoi elles fonctionnent. Pour l'instant, il suffit de dire que certaines structures courantes que vous avez l'habitude d'utiliser peuvent ne pas respecter les règles, et vous devrez rechercher des alternatives. Mais l'effet net de ces restrictions est d'apporter juste assez d'ordre dans le chaos pour permettre aux contrôles de compilation de Rust de vérifier que votre programme est exempt d'erreurs de sécurité mémoire : pointeurs suspendus, doubles libérations, utilisation de mémoire non initialisée, etc. Au moment de l'exécution, vos pointeurs sont de simples adresses en mémoire, comme ils le seraient en C et C++. La différence est qu'il a été prouvé que votre code les utilise en toute sécurité.

Ces mêmes règles constituent également la base de la prise en charge par Rust de la sécurité simultanée programmation. En utilisant les primitives de threading soigneusement conçues de Rust, les règles qui garantissent que votre code utilise correctement la mémoire servent également à prouver qu'il est exempt de courses de données. Un bogue dans un programme Rust ne peut pas amener un thread à corrompre les données d'un autre, introduisant des défaillances difficiles à reproduire dans des parties non liées du système. Le comportement non déterministe inhérent au code multithread est isolé des fonctionnalités conçues pour le gérer (mutex, canaux de messages, valeurs atomiques, etc.) plutôt que d'apparaître dans les références mémoire ordinaires. Le code multithread en C et C++ a gagné sa mauvaise réputation, mais Rust le réhabilite assez bien.

Le pari radical de Rust, la revendication sur laquelle il mise son succès et qui constitue la racine du langage, est que même avec ces restrictions en place, vous trouverez le langage plus qu'assez flexible pour presque toutes les tâches et que les avantages - les l'élimination de grandes classes de bogues de gestion de la mémoire et de simultanéité - justifiera les adaptations que vous devrez apporter à votre style. Les auteurs de ce livre sont optimistes sur Rust précisément à cause de notre vaste expérience avec C et C++. Pour nous, l'accord de Rust est une évidence.

Les règles de Rust sont probablement différentes de ce que vous avez vu dans d'autres langages de programmation. Apprendre à travailler avec eux et à les transformer à votre avantage est, à notre avis, le défi central de l'apprentissage de Rust. Dans ce chapitre, nous allons d'abord donner un aperçu de la logique et de l'intention derrière les règles de Rust en montrant comment les mêmes problèmes sous-jacents se produisent dans d'autres langages. Ensuite, nous expliquerons en détail les règles de Rust, en examinant ce que signifie la propriété au niveau conceptuel et mécanique, comment les changements de propriété sont suivis dans divers scé-

narios et les types qui contournent ou enfreignent certaines de ces règles afin de fournir plus de flexibilité.

La possession

Si vous avez lu beaucoup de code C ou C++, vous avez probablement rencontré un commentaire indiquant qu'une instance d'une classe *possède* un autre objet vers lequel elle pointe. Cela signifie généralement que l'objet propriétaire décide quand libérer l'objet possédé : lorsque le propriétaire est détruit, il détruit ses possessions avec lui.

Par exemple, supposons que vous écriviez le code C++ suivant :

```
std::string s = "frayed knot";
```

La chaîne `s` est généralement représentée en mémoire, comme illustré à [la Figure 4-1](#).

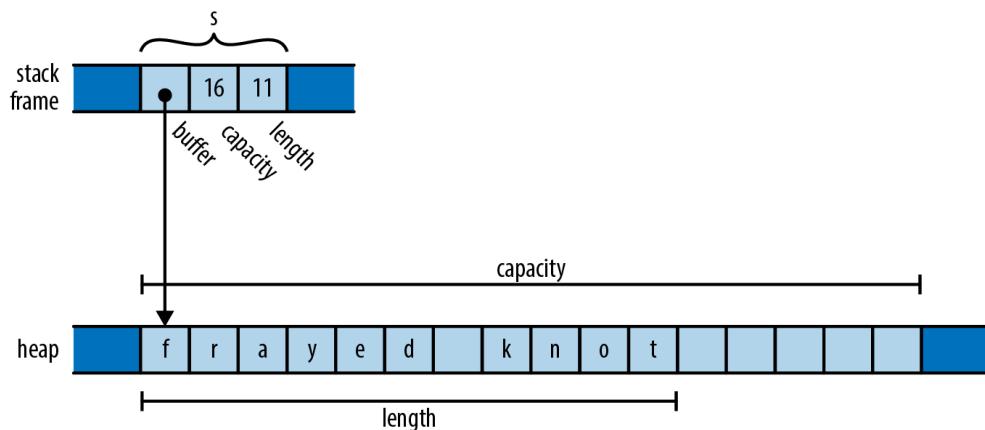


Figure 4-1. Une valeur C++ `std::string` sur la pile, pointant vers son tampon alloué par tas

Ici, l'objet réel `std::string` lui-même fait toujours exactement trois mots, comprenant un pointeur vers un tampon alloué par tas, la capacité globale du tampon (c'est-à-dire la taille du texte peut croître avant que la chaîne doive allouer un tampon plus grand pour le contenir), et la longueur du texte qu'il contient maintenant. Ce sont des champs privés à la `std::string` classe, non accessibles aux utilisateurs de la chaîne.

A `std::string` possède son tampon : lorsque le programme détruit la chaîne, le destructeur de la chaîne libère le tampon. Dans le passé, certaines bibliothèques C++ partageaient un seul tampon entre plusieurs `std::string` valeurs, en utilisant un compteur de références pour décliner quand le tampon devait être libéré. Les versions plus récentes de la spécification C++ excluent effectivement cette représentation ; toutes les bibliothèques C++ modernes utilisent l'approche présentée ici.

Dans ces situations, il est généralement entendu que bien qu'il soit acceptable pour un autre code de créer des pointeurs temporaires vers la mémoire possédée, il est de la responsabilité de ce code de s'assurer que ses pointeurs ont disparu avant que le propriétaire ne décide de détruire l'objet possédé. Vous pouvez créer un pointeur vers un caractère vivant dans `std::string` le tampon d'un , mais lorsque la chaîne est détruite, votre pointeur devient invalide, et c'est à vous de vous assurer que vous ne l'utilisez plus. Le propriétaire détermine la durée de vie de la propriété et tous les autres doivent respecter ses décisions.

Nous avons utilisé `std::string` ici un exemple de ce à quoi ressemble la propriété en C++ : c'est juste une convention que la bibliothèque standard suit généralement, et bien que le langage vous encourage à suivre des pratiques similaires, la façon dont vous concevez vos propres types dépend finalement de vous.

Dans Rust, cependant, le concept de propriété est intégré au langage lui-même et appliqué par des vérifications au moment de la compilation. Chaque valeur a un propriétaire unique qui détermine sa durée de vie. Lorsque le propriétaire est libéré— *abandonné*, dans la terminologie Rust - la valeur possédée est également supprimée. Ces règles sont destinées à vous permettre de trouver facilement la durée de vie d'une valeur donnée simplement en inspectant le code, vous donnant le contrôle sur sa durée de vie qu'un langage système devrait fournir.

Une variable possède sa valeur. Lorsque le contrôle quitte le bloc dans lequel la variable est déclarée, la variable est supprimée, donc sa valeur est supprimée avec elle. Par exemple:

```
fn print_padovan() {
    let mut padovan = vec![1, 1, 1]; // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
} // dropped here
```

Le type de la variable `padovan` est `Vec<i32>` , un vecteur d'entiers 32 bits. En mémoire, la valeur finale de `padovan` ressemblera à la [Figure 4-2](#).

.

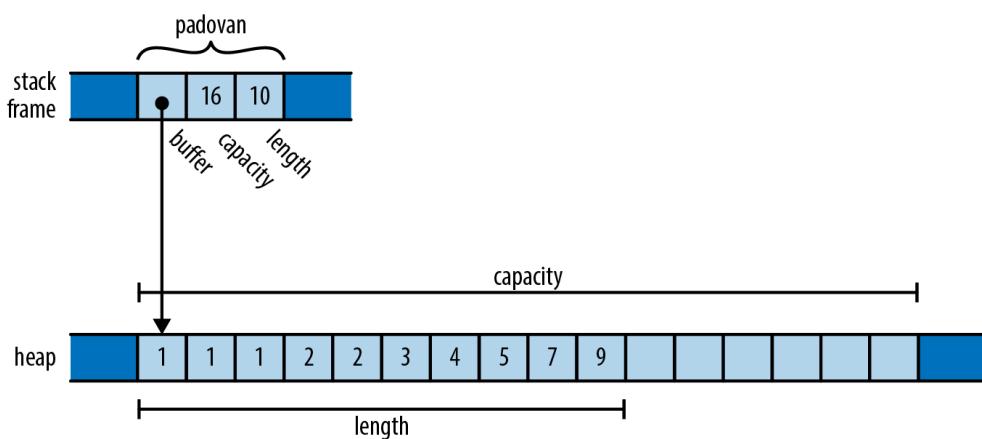


Illustration 4-2. A `Vec<i32>` sur la pile, pointant vers son tampon dans le tas

Ceci est très similaire au C++ `std::string` que nous avons montré précédemment, sauf que les éléments du tampon sont des valeurs 32 bits, pas des caractères. Notez que les mots contenant `padovan` le pointeur, la capacité et la longueur de 's vivent directement dans le cadre de pile de la `print_padovan` fonction; seul le tampon du vecteur est alloué sur le tas.

Comme pour la chaîne `s` précédente, le vecteur possède le tampon contenant ses éléments. Lorsque la variable `padovan` sort de la portée à la fin de la fonction, le programme supprime le vecteur. Et puisque le vecteur possède son tampon, le tampon va avec.

`Box` Type de rouilleest un autre exemple de propriété. A `Box<T>` est un pointeur vers une valeur de type `T` stockée sur le tas. L'appel `Box::new(v)` alloue de l'espace de tas, `v` y déplace la valeur et renvoie un `Box` pointage vers l'espace de tas. Puisque a `Box` possède l'espace vers lequel il pointe, lorsque le `Box` est supprimé, il libère également l'espace.

Par exemple, vous pouvez allouer un tuple dans le tas comme ceci :

```
{
    let point = Box::new((0.625, 0.5)); // point allocated here
    let label = format!("{}:", point); // label allocated here
    assert_eq!(label, "(0.625, 0.5)");
} // both dropped here
```

Lorsque le programme appelle `Box::new`, il alloue de l'espace pour un tuple de deux `f64` valeurs sur le tas, déplace son argument `(0.625, 0.5)` dans cet espace et renvoie un pointeur vers celui-ci. Au moment où le contrôle atteint l'appel à `assert_eq!`, le cadre de la pile ressemble à la [figure 4-3](#).

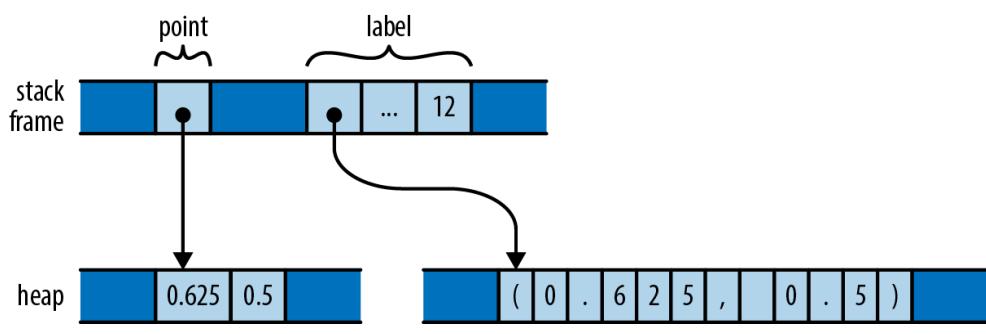


Figure 4-3. Deux variables locales, chacune possédant de la mémoire dans le tas

Le cadre de pile lui-même contient les variables `point` et `label`, chacune faisant référence à une allocation de tas qu'elle possède. Lorsqu'ils sont supprimés, les allocations qu'ils possèdent sont libérées avec eux.

Tout comme les variables possèdent leurs valeurs, les structures possèdent leurs champs et les tuples, les tableaux et les vecteurs possèdent leurs éléments :

```
struct Person { name: String, birth:i32 }

let mut composers = Vec:: new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{} , born {}", composer.name, composer.birth);
}
```

Ici, `composers` est un `Vec<Person>`, un vecteur de structures, chacune contenant une chaîne et un nombre. En mémoire, la valeur finale de `composers` ressemble à la [Figure 4-4](#).

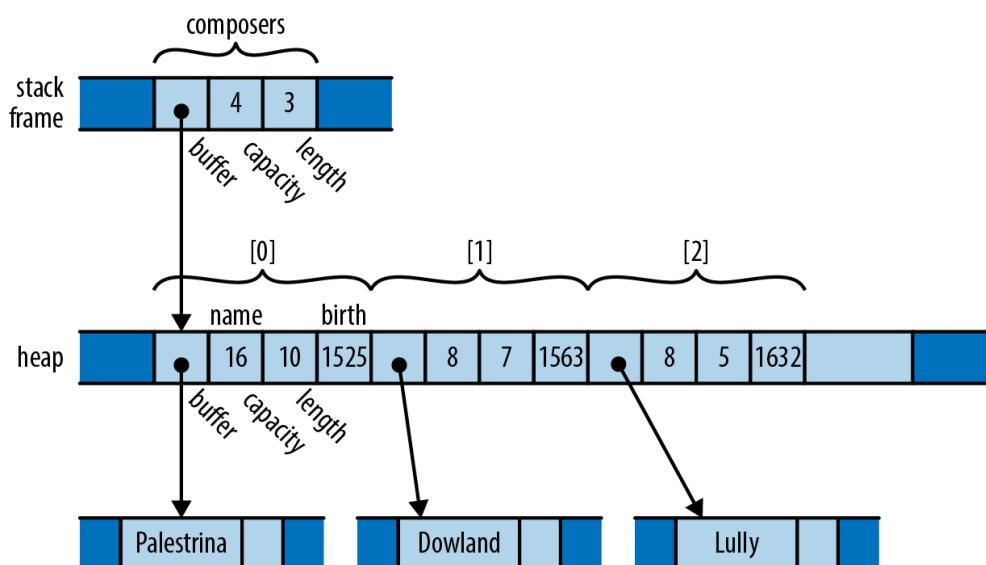


Illustration 4-4. Un arbre de propriété plus complexe

Il existe de nombreuses relations de propriété ici, mais chacune est assez simple : `composers` possède un vecteur ; le vecteur possède ses éléments, dont chacun est une `Person` structure ; chaque structure possède ses terrains ; et le champ de chaîne possède son texte. Lorsque le contrôle quitte la portée dans laquelle `composers` il est déclaré, le programme supprime sa valeur et emporte l'ensemble de l'arrangement avec lui. S'il y avait d'autres sortes de collections dans l'image-un `HashMap`, peut-être, ou un `BTreeSet` l'histoire serait la même.

À ce stade, prenez du recul et considérez les conséquences des relations de propriété que nous avons présentées jusqu'à présent. Chaque valeur a un propriétaire unique, ce qui permet de décider facilement quand la supprimer. Mais une seule valeur peut posséder plusieurs autres valeurs : par exemple, le vecteur `composers` possède tous ses éléments. Et ces valeurs peuvent posséder d'autres valeurs à leur tour : chaque élément de `composers` possède une chaîne, qui possède son texte.

Il s'ensuit que les propriétaires et leurs valeurs possédées forment *des arbres*: votre propriétaire est votre parent, et les valeurs que vous possédez sont vos enfants. Et à la racine ultime de chaque arbre se trouve une variable ; lorsque cette variable sort de la portée, l'arborescence entière l'accompagne. Nous pouvons voir un tel arbre de propriété dans le diagramme pour `composers` : ce n'est pas un "arbre" au sens d'une structure de données d'arbre de recherche, ou un document HTML fait à partir d'éléments DOM. Nous avons plutôt un arbre construit à partir d'un mélange de types, la règle du propriétaire unique de Rust interdisant tout regroupement de structure qui pourrait rendre l'arrangement plus complexe qu'un arbre. Chaque valeur d'un programme Rust est membre d'un arbre, enraciné dans une variable.

Les programmes Rust ne suppriment généralement pas explicitement les valeurs du tout, de la même manière que les programmes C et C++ utiliseraient `free` et `delete`. La façon de supprimer une valeur dans Rust est de la supprimer de l'arbre de propriété d'une manière ou d'une autre : en quittant la portée d'une variable, ou en supprimant un élément d'un vecteur, ou quelque chose du genre. À ce stade, Rust s'assure que la valeur est correctement supprimée, ainsi que tout ce qu'elle possède.

Dans un certain sens, Rust est moins puissant que les autres langages : tous les autres langages de programmation pratiques vous permettent de construire des graphiques arbitraires d'objets qui pointent les uns vers les autres de la manière que vous jugez appropriée. Mais c'est justement parce que Rust est moins puissant que les analyses que le langage peut effectuer sur vos programmes peuvent être plus puissantes. Les garanties de sécurité de Rust sont possibles exactement parce que les relations qu'il peut rencontrer dans votre code sont plus traitables. Cela fait partie du

« pari radical » de Rust que nous avons mentionné plus tôt : dans la pratique, affirme Rust, il y a généralement plus qu'assez de flexibilité dans la façon dont on s'y prend pour résoudre un problème pour s'assurer qu'au moins quelques solutions parfaitement fines respectent les restrictions imposées par le langage.

Cela dit, le concept de propriété tel que nous l'avons expliqué jusqu'ici est encore beaucoup trop rigide pour être utile. Rust étend cette idée simple de plusieurs façons :

- Vous pouvez déplacer des valeurs d'un propriétaire à un autre. Cela vous permet de construire, de réorganiser et de démolir l'arbre.
- Les types très simples comme les entiers, les nombres à virgule flottante et les caractères sont exemptés des règles de propriété. Ceux-ci sont appelés `Copy` types.
- La bibliothèque standard fournit les types de pointeurs à référence comptée `Rc` et `Arc`, qui permettent aux valeurs d'avoir plusieurs propriétaires, sous certaines restrictions.
- Vous pouvez « emprunter une référence » à une valeur ; les références sont des pointeurs non propriétaires, avec des durées de vie limitées.

Chacune de ces stratégies apporte de la flexibilité au modèle de propriété, tout en respectant les promesses de Rust. Nous expliquerons chacun à son tour, avec des références couvertes dans le chapitre suivant.

Se déplace

En rouille, pour la plupart des types, les opérations telles que l'affectation d'une valeur à une variable, sa transmission à une fonction ou son retour à partir d'une fonction ne copient pas la valeur : elles la *déplacent*. La source abandonne la propriété de la valeur à la destination et devient non initialisée ; la destination contrôle désormais la durée de vie de la valeur. Les programmes Rust construisent et détruisent des structures complexes une valeur à la fois, un mouvement à la fois.

Vous pourriez être surpris que Rust change le sens de telles opérations fondamentales ; l'affectation est sûrement quelque chose qui devrait être assez bien défini à ce stade de l'histoire. Cependant, si vous regardez attentivement la façon dont les différentes langues ont choisi de gérer les devoirs, vous verrez qu'il existe en fait des variations significatives d'une école à l'autre. La comparaison permet également de mieux comprendre la signification et les conséquences du choix de Rust.

Considérez le code Python suivant :

```

s = [ 'udon' , 'ramen' , 'soba' ]
t = s
u = s

```

Chaque PythonL'objet contient un décompte de références, qui suit le nombre de valeurs qui s'y réfèrent actuellement. Ainsi, après l'affectation à `s`, l'état du programme ressemble à la [Figure 4-5](#) (notez que certains champs sont omis).

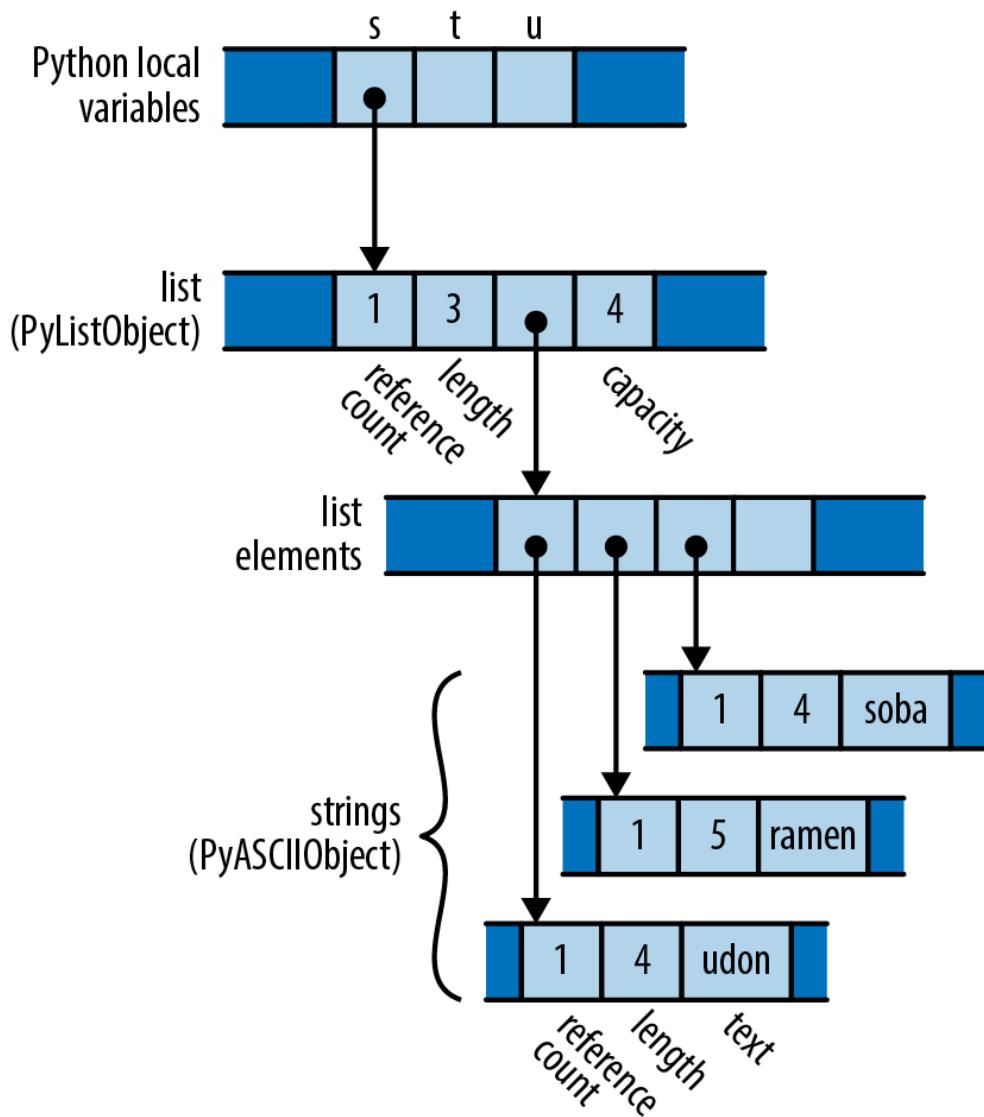


Figure 4-5. Comment Python représente une liste de chaînes en mémoire

Puisque `only s` pointe vers la liste, le nombre de références de la liste est 1 ; et puisque la liste est le seul objet pointant vers les chaînes, chacune de leurs décomptes de références vaut également 1.

Que se passe-t-il lorsque le programme exécute les affectations à `t` et `u` ? Python implémente l'affectation simplement en faisant pointer la destination sur le même objet que la source et en incrémentant le nombre de références de l'objet. L'état final du programme ressemble donc à la [figure 4-6](#).

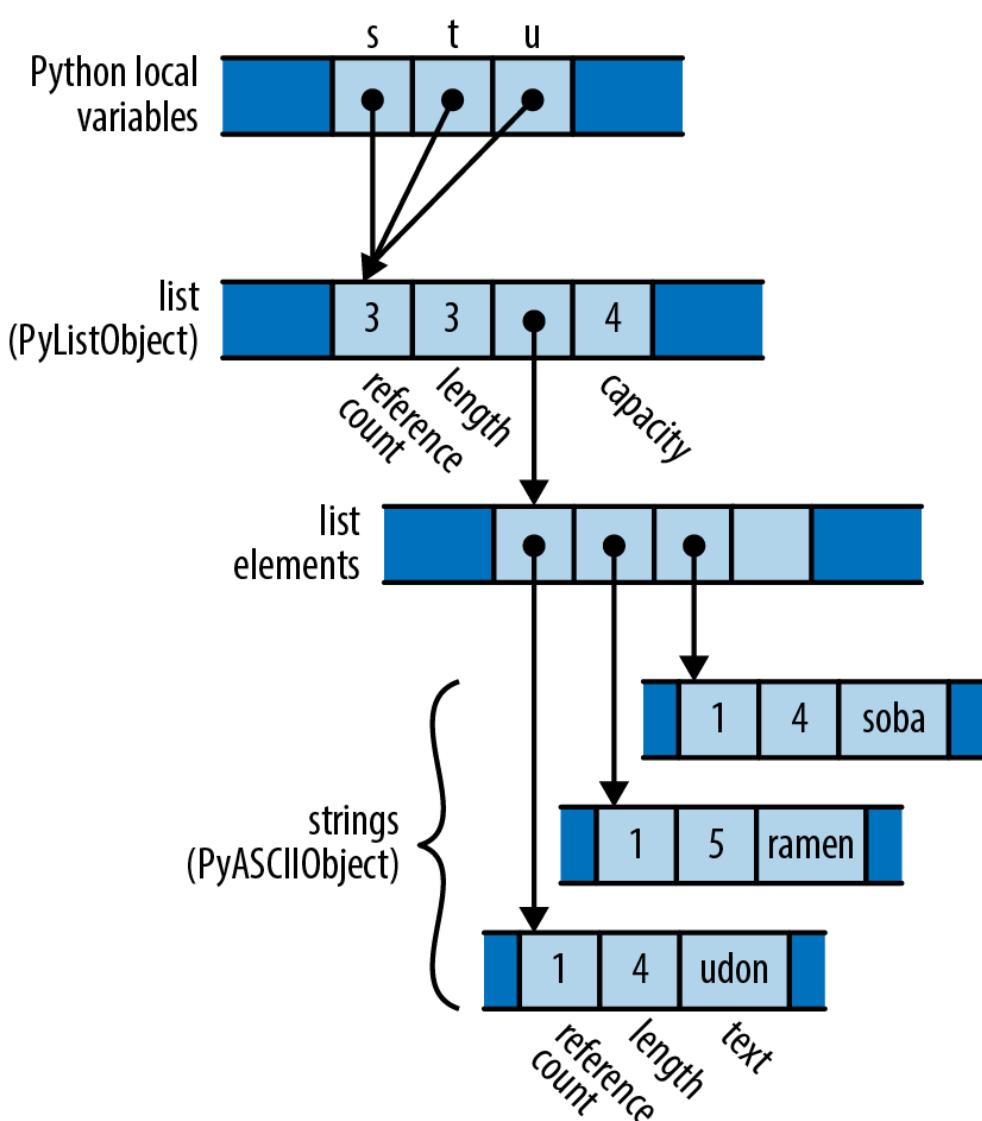


Figure 4-6. Le résultatat de l'affectation `s` à la fois `t` et `u` en Python

Python a copié le pointeur de `s` dans `t` et `u` et a mis à jour le nombre de références de la liste à 3. L'affectation en Python est bon marché, mais comme elle crée une nouvelle référence à l'objet, nous devons maintenir le nombre de références pour savoir quand nous pouvons libérer la valeur.

Considérons maintenant le code C++ analogue:

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

La valeur d'origine de `s` ressemble à la [Figure 4-7](#) en mémoire.

Que se passe-t-il lorsque le programme affecte `s` à `t` et `u`? L'affectation de `a std::vector` produit une copie du vecteur en C++ ; `std::string` se comporte de manière similaire. Ainsi, au moment où le programme atteint la fin de ce code, il a en fait alloué trois vecteurs et neuf chaînes ([Figure 4-8](#)).

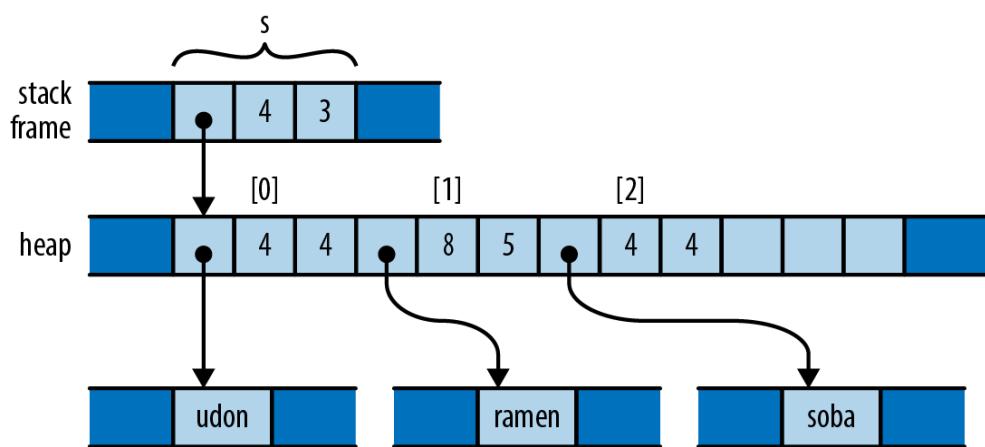


Illustration 4-7. Comment C++ représente un vecteur de chaînes en mémoire

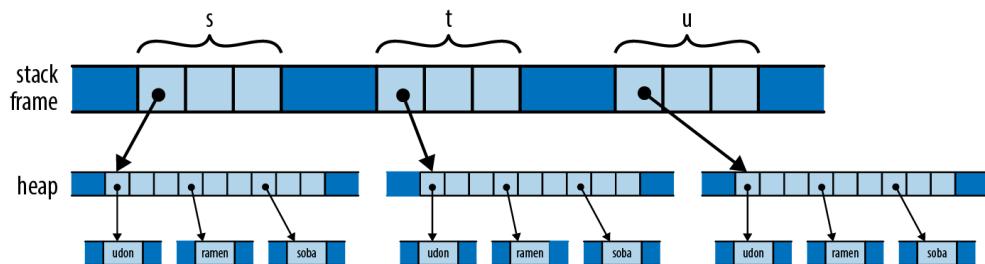


Illustration 4-8. Le résultat de l'affectation *s* à la fois *t* et *u* en C++

Selon les valeurs impliquées, l'affectation en C++ peut consommer des quantités illimitées de mémoire et de temps processeur. L'avantage, cependant, est qu'il est facile pour le programme de décider quand libérer toute cette mémoire : lorsque les variables sortent de la portée, tout ce qui est alloué ici est automatiquement nettoyé.

Dans un sens, C++ et Python ont choisi des compromis opposés : Python rend l'affectation bon marché, au prix d'un comptage de références (et dans le cas général, d'un ramasse-miettes). C++ conserve clairement la propriété de toute la mémoire, au détriment de l'attribution d'une copie complète de l'objet. Les programmeurs C++ sont souvent peu enthousiasmés par ce choix : les copies complètes peuvent être coûteuses et il existe généralement des alternatives plus pratiques.

Alors, que ferait le programme analogue dans Rust ? Voici le code :

```
let s = vec![ "udon".to_string(), "ramen".to_string(), "soba".to_string() ];
let t = s;
let u = s;
```

Comme C et C++, Rust met des littéraux de chaîne simples comme "udon" dans la mémoire en lecture seule, donc pour une comparaison plus claire avec les exemples C++ et Python, nous appelons ici pour obtenir des valeurs `to_string` allouées au tas `.String`

Après avoir effectué l'initialisation de *s*, étant donné que Rust et C++ utilisent des représentations similaires pour les vecteurs et les chaînes, la si-

tuation ressemble exactement à ce qu'elle était en C++ ([Figure 4-9](#)).

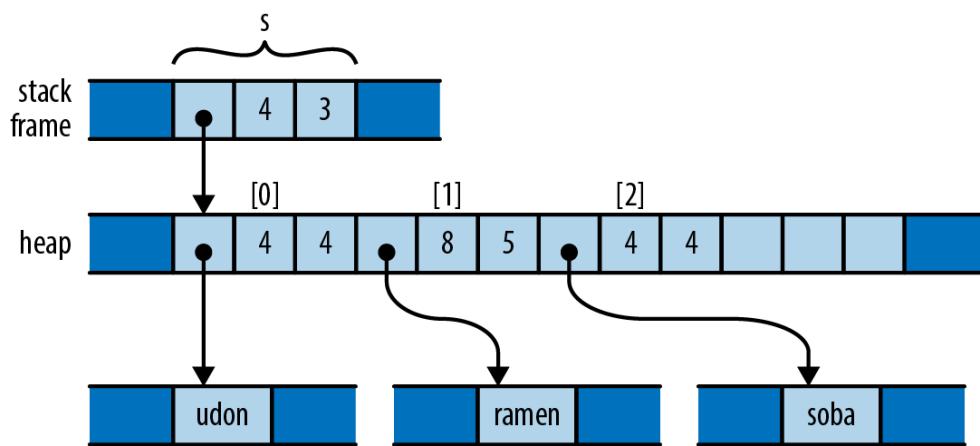


Illustration 4-9. Comment Rust représente un vecteur de chaînes en mémoire

Mais rappelez-vous que, dans Rust, les affectations de la plupart des types *déplacent* la valeur de la source vers la destination, laissant la source non initialisée. Ainsi, après l'initialisation `t`, la mémoire du programme ressemble à la [Figure 4-10](#).

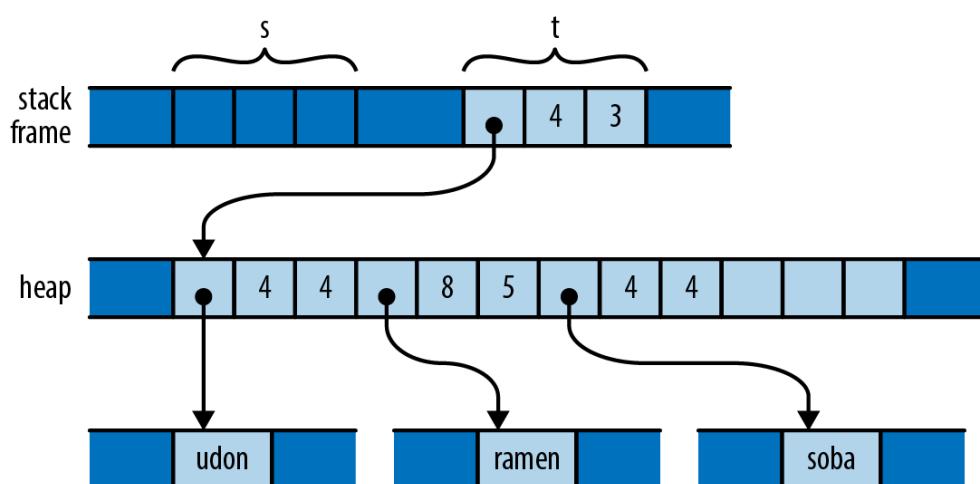


Figure 4-10. Le résultat de l'affectation `s` à `t` dans Rust

Que s'est-il passé ici ? L'initialisation `let t = s;` a déplacé les trois champs d'en-tête du vecteur de `s` vers `t`; possède maintenant `t` le vecteur. Les éléments du vecteur sont restés là où ils étaient et rien n'est arrivé aux chaînes non plus. Chaque valeur a toujours un propriétaire unique, même si l'un d'entre eux a changé de mains. Il n'y avait pas de comptages de référence à ajuster. Et le compilateur considère maintenant `s` non initialisé.

Alors que se passe-t-il lorsque nous atteignons l'initialisation `let u = s;` ? Cela affecterait la valeur non initialisée `s` à `u`. Rust interdit prudemment l'utilisation de valeurs non initialisées, donc le compilateur rejette ce code avec l'erreur suivante :

```
error: use of moved value: `s`
```

|

```

7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
|         - move occurs because `s` has type `Vec<String>`,
|             which does not implement the `Copy` trait
8 |     let t = s;
|             - value moved here
9 |     let u = s;
|             ^ value used here after move

```

Considérez les conséquences de l'utilisation d'un mouvement par Rust ici. Comme Python, l'affectation est bon marché : le programme déplace simplement l'en-tête de trois mots du vecteur d'un endroit à un autre. Mais comme en C++, la propriété est toujours claire : le programme n'a pas besoin de comptage de références ou de récupération de place pour savoir quand libérer les éléments vectoriels et le contenu des chaînes.

Le prix à payer est que vous devez explicitement demander des copies quand vous en avez besoin. Si vous voulez vous retrouver dans le même état que le programme C++, avec chaque variable contenant une copie indépendante de la structure, vous devez appeler la `clone` méthode du vecteur, qui effectue une copie complète du vecteur et de ses éléments :

```

let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s.clone();
let u = s.clone();

```

Vous pouvez également recréer le comportement de Python en utilisant les types de pointeurs comptés par référence de Rust; nous en discuterons sous peu dans [« Rc et Arc : Propriété partagée »](#).

Plus d'opérations qui bougent

Dans les exemples jusqu'à présent, nous avons montré des initialisations, en fournissant des valeurs pour les variables lorsqu'elles entrent dans la portée d'une `let` instruction. Affectation à une variable est légèrement différente, en ce sens que si vous déplacez une valeur dans une variable qui a déjà été initialisée, Rust supprime la valeur précédente de la variable. Par exemple:

```

let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // value "Govinda" dropped here

```

Dans ce code, lorsque le programme affecte la chaîne "Siddhartha" à `s`, sa valeur précédente "Govinda" est supprimée en premier. Mais considérez ce qui suit :

```

let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // nothing is dropped here

```

Cette fois, `t` a pris possession de la chaîne d'origine de `s`, de sorte qu'au moment où nous l'attribuons à `s`, elle n'est pas initialisée. Dans ce scénario, aucune chaîne n'est supprimée.

Nous avons utilisé des initialisations et des affectations dans les exemples ici parce qu'elles sont simples, mais Rust applique la sémantique de déplacement à presque toutes les utilisations d'une valeur. Passer des arguments aux fonctions déplace la propriété des paramètres de la fonction ; renvoyer une valeur à partir d'une fonction déplace la propriété vers l'appelant. Construire un tuple déplace les valeurs dans le tuple. Etc.

Vous avez peut-être maintenant une meilleure idée de ce qui se passe réellement dans les exemples que nous avons proposés dans la section précédente. Par exemple, lorsque nous construisions notre vecteur de compositeurs, nous écrivions :

```

struct Person { name: String, birth:i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth:1525 });

```

Ce code montre plusieurs endroits où se produisent les déplacements, au-delà de l'initialisation et de l'affectation :

Renvoyer des valeurs d'une fonction

La call `Vec::new()` construit un nouveau vecteur et retourne, non pas un pointeur vers le vecteur, mais le vecteur lui-même : sa propriété se déplace de `Vec::new` vers la variable `composers`. De même, l'`to_string` appel renvoie une nouvelle `String` instance.

Construire de nouvelles valeurs

Le `name` champ de la nouvelle `Person` structure est initialisée avec la valeur de retour de `to_string`. La structure s'approprie la chaîne.

Passer des valeurs à une fonction

La structure entière `Person`, pas un pointeur vers elle, est passée à la méthode du vecteur `push`, qui la déplace à la fin de la structure. Le vecteur s'approprie le `Person` et devient ainsi également le propriétaire indirect du nom `String`.

Valeurs mobiles autour comme cela peut sembler inefficace, mais il y a deux choses à garder à l'esprit. Tout d'abord, les déplacements s'appliquent toujours à la valeur proprement dite, et non au stockage de tas qu'ils possèdent. Pour les vecteurs et les chaînes, la *valeur propre* est l'en-tête de trois mots seul ; les tableaux d'éléments potentiellement volumineux et les tampons de texte restent là où ils se trouvent dans le tas. Deuxièmement, la génération de code du compilateur Rust est bonne pour « voir à travers » tous ces mouvements ; en pratique, le code machine stocke souvent la valeur directement là où elle appartient.

Mouvements et flux de contrôle

La précédente les exemples ont tous un flux de contrôle très simple ; comment les mouvements interagissent-ils avec un code plus compliqué ? Le principe général est que, s'il est possible qu'une variable ait vu sa valeur éloignée et qu'elle n'ait pas définitivement reçu une nouvelle valeur depuis, elle est considérée comme non initialisée. Par exemple, si une variable a toujours une valeur après avoir évalué `if` la condition d'une expression, nous pouvons l'utiliser dans les deux branches :

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x); // bad: x is uninitialized here if either path uses it
```

Pour des raisons similaires, le déplacement d'une variable dans une boucle est interdit :

```
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first iteration,
           // uninitialized in second
}
```

Autrement dit, à moins que nous ne lui ayons définitivement donné une nouvelle valeur à la prochaine itération :

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);
```

Déplacements et contenu indexé

Nous avons mentionné qu'un mouvement laisse sa source non initialisée, car la destination prend possession de la valeur. Mais tous les types de propriétaires de valeur ne sont pas prêts à devenir non initialisés. Par exemple, considérez le code suivant :

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2]; // error: Cannot move out of index of Vec
let fifth = v[4]; // here too
```

Pour que cela fonctionne, Rust devrait en quelque sorte se souvenir que les troisième et cinquième éléments du vecteur sont devenus non initialisés et suivre ces informations jusqu'à ce que le vecteur soit supprimé. Dans le cas le plus général, les vecteurs devraient transporter des informations supplémentaires avec eux pour indiquer quels éléments sont actifs et lesquels ne sont plus initialisés. Ce n'est clairement pas le bon comportement pour un langage de programmation système ; un vecteur ne devrait être rien d'autre qu'un vecteur. En fait, Rust rejette le code précédent avec l'erreur suivante :

```
error: cannot move out of index of `Vec<String>`  
|  
14 |     let third = v[2];
      ^^^^  
|  
|     move occurs because value has type `String`,  
|     which does not implement the `Copy` trait  
|     help: consider borrowing here: `&v[2]`
```

Il formule également une plainte similaire concernant le passage à `fifth`. Dans le message d'erreur, Rust suggère d'utiliser une référence, au cas où vous souhaiteriez accéder à l'élément sans le déplacer. C'est souvent ce que vous voulez. Mais que se passe-t-il si vous voulez vraiment déplacer un élément hors d'un vecteur ? Vous devez trouver une méthode qui le fait d'une manière qui respecte les limites du type. Voici trois possibilités :

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
```

```

    v.push(i.to_string());
}

// 1. Pop a value off the end of the vector:
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 2. Move a value out of a given index in the vector,
// and move the last element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. Swap in another value for the one we're taking out:
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// Let's see what's left of our vector.
assert_eq!(v, vec!["101", "104", "substitute"]);

```

Chacune de ces méthodes déplace un élément hors du vecteur, mais le fait d'une manière qui laisse le vecteur dans un état entièrement rempli, voire plus petit.

Les types de collection comme `vec` proposent aussi généralement des méthodes pour consommer tous leurs éléments en boucle :

```

let v = vec![
    "liberté".to_string(),
    "égalité".to_string(),
    "fraternité".to_string()];
for mut s in v {
    s.push('!');
    println!("{}!", s);
}

```

Lorsque nous passons directement le vecteur à la boucle, comme dans `for ... in v`, cela *déplace* le vecteur hors de `v`, le laissant `v` non initialisé. La `for` machinerie interne de la boucle s'approprie le vecteur et le dissèque en ses éléments. A chaque itération, la boucle déplace un autre élément vers la variable `s`. Puisque `s` maintenant possède la chaîne, nous pouvons la modifier dans le corps de la boucle avant de l'imprimer. Et puisque le vecteur lui-même n'est plus visible pour le code, rien ne peut l'observer au milieu de la boucle dans un état partiellement vidé.

Si vous avez besoin de déplacer une valeur hors d'un propriétaire que le compilateur ne peut pas suivre, vous pouvez envisager de changer le type du propriétaire en quelque chose qui peut suivre dynamiquement s'il a

une valeur ou non. Par exemple, voici une variante de l'exemple précédent :

```
struct Person { name: Option<String>, birth:i32 }

let mut composers = Vec:: new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth:1525 });
```

Vous ne pouvez pas faire ceci :

```
let first_name = composers[0].name;
```

Cela provoquera simplement la même erreur "Impossible de sortir de l'index" indiquée précédemment. Mais parce que vous avez changé le type du `name` champ de `String` à `Option<String>`, cela signifie qu'il `None` s'agit d'une valeur légitime pour le champ, donc cela fonctionne :

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

L'`replace` appel déplace la valeur de `composers[0].name`, laissant `None` à sa place, et transmet la propriété de la valeur d'origine à son appelant. En fait, l'utilisation de `Option` cette méthode est suffisamment courante pour que le type fournit une `take` méthode dans ce but précis. Vous pourriez écrire plus lisiblement la manipulation précédente comme suit:

```
let first_name = composers[0].name.take();
```

Cet appel à `take` a le même effet que l'appel précédent à `replace`.

Types de copie : l'exception aux déménagements

Les exemples nous avons montré jusqu'à présent que les valeurs déplacées impliquent des vecteurs, des chaînes et d'autres types qui pourraient potentiellement utiliser beaucoup de mémoire et être coûteux à copier. Les mouvements gardent la propriété de ces types claire et l'affectation bon marché. Mais pour les types plus simples comme les entiers ou les caractères, ce genre de manipulation prudente n'est vraiment pas nécessaire.

Comparez ce qui se passe en mémoire lorsque nous attribuons à `String` avec ce qui se passe lorsque nous attribuons une `i32` valeur :

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1:i32 = 36;
let num2 = num1;
```

Après avoir exécuté ce code, la mémoire ressemble à la [Figure 4-11](#) .

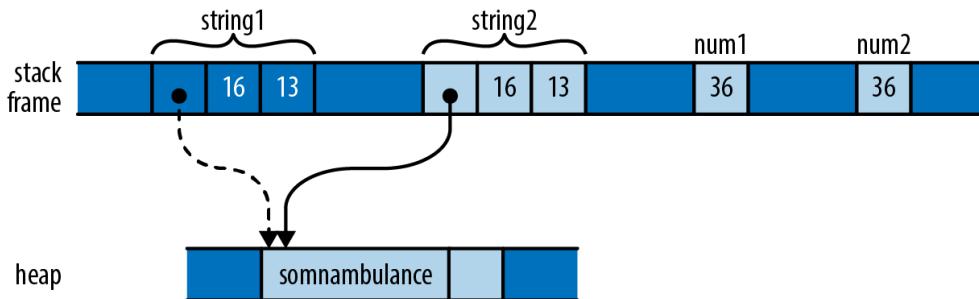


Figure 4-11. L'attribution d'un `String` déplace la valeur, tandis que l'attribution d'un `i32` la copie

Comme pour les vecteurs précédents, l'affectation *se déplace* `string1` vers `string2` afin que nous ne nous retrouvions pas avec deux chaînes responsables de la libération du même tampon. Cependant, la situation avec `num1` et `num2` est différente. Un `i32` est simplement un motif de bits en mémoire ; il ne possède aucune ressource de tas ou ne dépend vraiment d'autre chose que des octets qu'il comprend. Au moment où nous avons déplacé ses bits vers `num2`, nous avons créé une copie complètement indépendante de `num1`.

Le déplacement d'une valeur laisse la source du déplacement non initialisée. Mais alors qu'il sert un objectif essentiel de traiter `string1` comme sans valeur, traiter de `num1` cette façon est inutile ; aucun dommage ne pourrait résulter de la poursuite de son utilisation. Les avantages d'un déménagement ne s'appliquent pas ici, et c'est gênant.

Précédemment, nous avons pris soin de dire que *la plupart des types* sont déplacés ; nous en sommes maintenant aux exceptions, les types que Rust désigne comme *copy types*. L'attribution d'une valeur d'un *copy type* copie la valeur, plutôt que de la déplacer. La source de l'affectation reste initialisée et utilisable, avec la même valeur qu'elle avait auparavant. Passer *Copy* des types aux fonctions et aux constructeurs se comporte de la même manière.

Les *copy types* standard incluent tous les types entiers et numériques à virgule flottante de la machine, les types `char` et `bool` et quelques

autres. Un tuple ou un tableau de `Copy` types de taille fixe est lui-même un `Copy` type.

Seuls les types pour lesquels une simple copie bit à bit suffit peuvent être `Copy`. Comme nous l'avons déjà expliqué, `String` n'est pas un `Copy` type, car il possède un tampon alloué par tas. Pour des raisons similaires, `Box<T>` n'est pas `Copy`; il possède son référent alloué par tas. Le `File` type, représentant un handle de fichier du système d'exploitation, n'est pas `Copy`; dupliquer une telle valeur impliquerait de demander au système d'exploitation un autre descripteur de fichier. De même, le `MutexGuard` type, représentant un mutex verrouillé, ne l'est pas `Copy`: ce type n'a aucun sens à copier, car un seul thread peut contenir un mutex à la fois.

En règle générale, tout type qui doit faire quelque chose de spécial lorsqu'une valeur est supprimée ne peut pas l'être `Copy`: `a_vec` doit libérer ses éléments, `a_File` doit fermer son descripteur de fichier, `a_MutexGuard` doit déverrouiller son mutex, etc. La duplication bit à bit de ces types ne permettrait pas de savoir quelle valeur était désormais responsable des ressources de l'original.

Qu'en est-il des types que vous définissez vous-même ? Par défaut, `struct` et `enum` les types ne sont pas `Copy`:

```
struct Label { number:u32 }

fn print(l:Label) { println!("STAMP: {}", l.number); }

let l = Label { number:3 };
print(l);
println!("My label number is: {}", l.number);
```

Cela ne compilera pas; La rouille se plaint :

```
error: borrow of moved value: `l`
|
10 |     let l = Label { number: 3 };
|         - move occurs because `l` has type `main::Label`,
|             which does not implement the `Copy` trait
11 |     print(l);
|         - value moved here
12 |     println!("My label number is: {}", l.number);
|                                         ^
|                                         value borrowed here after move
```

Puisque `Label` n'est pas `Copy`, le transmettre à `a_print` transférera la propriété de la valeur à la `print` fonction, qui l'a ensuite supprimée avant

de revenir. Mais c'est idiot; a `Label` n'est rien d'autre qu'un `u32` avec des prétentions. Il n'y a aucune raison de passer `1` à `print` devrait déplacer la valeur.

Mais les types définis par l'utilisateur étant non-`Copy` n'est que la valeur par défaut. Si tous les champs de votre struct sont eux-mêmes `Copy`, vous pouvez également créer le type `Copy` en plaçant l'attribut `#[derive(Copy, Clone)]` au-dessus de la définition, comme ceci :

```
#[derive(Copy, Clone)]
struct Label { number:u32 }
```

Avec ce changement, le code précédent se compile sans problème. Cependant, si nous essayons cela sur un type dont les champs ne sont pas `all Copy`, cela ne fonctionne pas. Supposons que nous compilions le code suivant :

```
#[derive(Copy, Clone)]
struct StringLabel { name:String }
```

Il provoque cette erreur :

```
error: the trait `Copy` may not be implemented for this type
|
7 | #[derive(Copy, Clone)]
|      ^^^^
8 | struct StringLabel { name: String }
|                         ----- this field does not implement `Copy`
```

Pourquoi les types définis par l'utilisateur ne sont-ils pas automatiquement `Copy`, en supposant qu'ils sont éligibles ? Qu'un type soit `Copy` ou non a un effet important sur la façon dont le code est autorisé à l'utiliser : `Copy` les types sont plus flexibles, car l'affectation et les opérations associées ne laissent pas l'original non initialisé. Mais pour l'implémenteur d'un type, le contraire est vrai : `Copy` les types sont très limités dans les types qu'ils peuvent contenir, tandis que les non `Copy`-types peuvent utiliser l'allocation de tas et posséder d'autres types de ressources. Ainsi, la création d'un type `Copy` représente un engagement sérieux de la part de l'implémenteur : s'il est nécessaire de le changer pour une version non `Copy` ultérieure, une grande partie du code qui l'utilise devra probablement être adaptée.

Alors que C++ vous permet de surcharger les opérateurs d'affectation et de définir des constructeurs de copie et de déplacement spécialisés, Rust ne permet pas ce type de personnalisation. Dans Rust, chaque mouve-

ment est une copie superficielle octet par octet qui laisse la source non initialisée. Les copies sont identiques, sauf que la source reste initialisée. Cela signifie que les classes C++ peuvent fournir des interfaces pratiques que les types Rust ne peuvent pas, où le code d'apparence ordinaire ajuste implicitement le nombre de références, reporte les copies coûteuses pour plus tard ou utilise d'autres astuces d'implémentation sophistiquées.

Mais l'effet de cette flexibilité sur C++ en tant que langage est de rendre moins prévisibles les opérations de base telles que l'affectation, le passage de paramètres et le retour de valeurs à partir de fonctions. Par exemple, plus tôt dans ce chapitre, nous avons montré comment l'assignation d'une variable à une autre en C++ peut nécessiter des quantités arbitraires de mémoire et de temps processeur. L'un des principes de Rust est que les coûts doivent être évidents pour le programmeur. Les opérations de base doivent rester simples. Les opérations potentiellement coûteuses doivent être explicites, comme les appels à `clone` dans l'exemple précédent qui font des copies complètes des vecteurs et des chaînes qu'ils contiennent.

Dans cette section, nous avons parlé de `Copy` et `Clone` en termes vagues comme des caractéristiques qu'un type pourrait avoir. Ce sont en fait des exemples de *traits*, la fonction ouverte de Rust pour classer les types en fonction de ce que vous pouvez en faire. Nous décrivons les traits en général au [chapitre 11](#), `Copy` et `Clone` en particulier au [chapitre 13](#).

Rc et Arc : Propriété partagée

Bien quela plupart des valeurs ont des propriétaires uniques dans le code Rust typique, dans certains cas, il est difficile de trouver chaque valeur d'un seul propriétaire qui a la durée de vie dont vous avez besoin ; vous aimeriez que la valeur vive simplement jusqu'à ce que tout le monde ait fini de l'utiliser. Pour ces cas, Rust fournit le pointeur compté par référencetypes `Rc` et `Arc`. Comme on peut s'y attendre de Rust, ceux-ci sont entièrement sûrs à utiliser : vous ne pouvez pas oublier d'ajuster le nombre de références, de créer d'autres pointeurs vers le référent que Rust ne remarque pas, ou de tomber sur l'un des autres types de problèmes qui accompagnent la référence. compter les types de pointeurs en C++.

Les types `Rc` et sont très similaires ; `Arc` la seule différence entre eux est qu'un `Arc` peut être partagé directement entre les threads en toute sécurité - le nom `Arc` est l'abréviation de *décompte de références atomiques* - alors qu'un simple `Rc` utilise un code non sécurisé pour les threads plus rapide pour mettre à jour son décompte de références. Si vous n'avez pas besoin de partager les pointeurs entre les threads, il n'y a aucune raison

de payer la pénalité de performance d'un `Arc`, vous devez donc utiliser `Rc`; La rouille vous empêchera d'en passer accidentellement un à travers une limite de fil. Les deux types sont par ailleurs équivalents, donc pour le reste de cette section, nous ne parlerons que de `Rc`.

Plus tôt, nous avons montré comment Python utilise le nombre de références pour gérer la durée de vie de ses valeurs. Vous pouvez utiliser `Rc` pour obtenir un effet similaire dans Rust. Considérez le code suivant :

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

// Rust can infer all these types; written out for clarity
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

Pour tout type `T`, une `Rc<T>` valeur est un pointeur vers un tas alloué `T` auquel un nombre de références a été attaché. Le clonage d'une `Rc<T>` valeur ne copie pas le `T`; à la place, il crée simplement un autre pointeur vers celui-ci et incrémente le nombre de références. Ainsi, le code précédent produit la situation illustrée à la [figure 4-12](#) en mémoire.

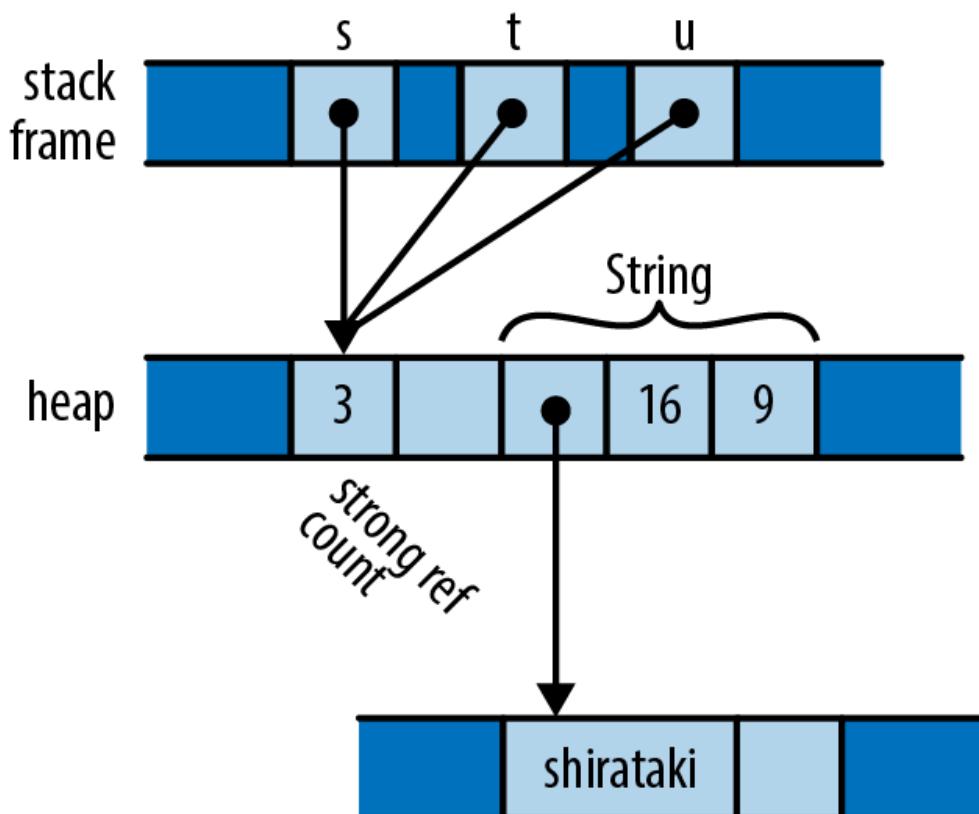


Illustration 4-12. Une chaîne comptée par référence avec trois références

Chacun des trois `Rc<String>` pointeurs fait référence au même bloc de mémoire, qui contient un nombre de références et un espace pour le fichier `String`. Les règles de propriété habituelles s'appliquent aux `Rc` pointeurs eux-mêmes, et lorsque le dernier existant `Rc` est supprimé, Rust supprime `String` également.

Vous pouvez utiliser n'importe laquelle des `String` méthodes habituelles de directement sur un `Rc<String>`:

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

Une valeur détenue par un `Rc` pointeur est immuable. Supposons que vous essayez d'ajouter du texte à la fin de la chaîne :

```
s.push_str(" noodles");
```

La rouille diminuera :

```
error: cannot borrow data in an `Rc` as mutable
|
13 |     s.push_str(" noodles");
|     ^ cannot borrow as mutable
|
```

Les garanties de sécurité de la mémoire et des threads de Rust dépendent de la garantie qu'aucune valeur n'est simultanément partagée et modifiable. Rust suppose que le référent d'un `Rc` pointeur peut en général être partagé, il ne doit donc pas être modifiable. Nous expliquons pourquoi cette restriction est importante au [chapitre 5](#).

Un problème bien connu avec l'utilisation des décomptes de références pour gérer la mémoire est que, s'il y a jamais deux valeurs comptées par référence qui pointent l'une vers l'autre, chacune maintiendra le décompte de référence de l'autre au-dessus de zéro, de sorte que les valeurs ne seront jamais libérées ([Figure 4-13](#)).

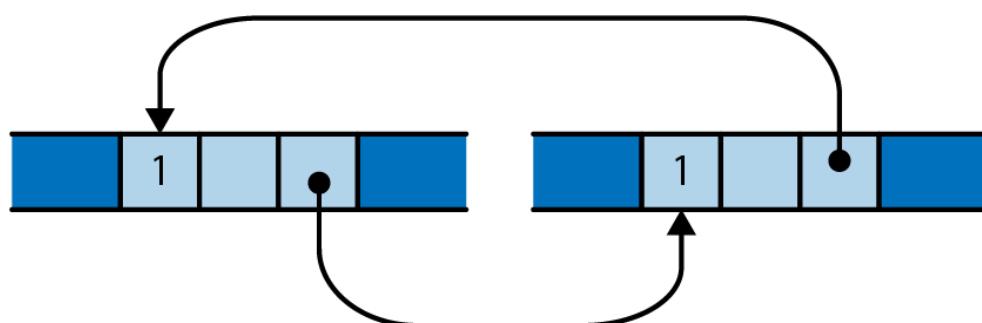


Figure 4-13. Une boucle de comptage de références ; ces objets ne seront pas libérés

Il est possible de divulguer des valeurs dans Rust de cette façon, mais de telles situations sont rares. Vous ne pouvez pas créer un cycle sans, à un moment donné, faire pointer une valeur plus ancienne vers une valeur plus récente. Cela nécessite évidemment que la valeur la plus ancienne

soit modifiable. Puisque `Rc` les pointeurs maintiennent leurs référents immuables, il n'est normalement pas possible de créer un cycle. Cependant, Rust fournit des moyens de créer des portions mutables de valeurs autrement immuables ; c'est ce qu'on appelle *la mutabilité intérieure*, et nous le couvrons dans « [Mutabilité intérieure](#) ». Si vous combinez ces techniques avec des `Rc` pointeurs, vous pouvez créer un cycle et une fuite de mémoire.

Vous pouvez parfois éviter de créer des cycles de `Rc` pointeurs en utilisant *des pointeurs faibles*, `std::rc::Weak`, pour certains des liens à la place. Cependant, nous ne les aborderons pas dans ce livre ; consultez la documentation de la bibliothèque standard pour plus de détails.

Les déplacements et les pointeurs comptés par référence sont deux façons d'assouplir la rigidité de l'arbre de propriété. Dans le chapitre suivant, nous examinerons une troisième voie : emprunter des références à des valeurs. Une fois que vous serez à l'aise avec la propriété et l'emprunt, vous aurez gravi la partie la plus raide de la courbe d'apprentissage de Rust et vous serez prêt à tirer parti des atouts uniques de Rust..

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 5. Références

Les bibliothèques ne peuvent pas fournir de nouvelles incapacités.

—Marc Miller

Tout le pointeur les types que nous avons vus jusqu'à présent (le simple `Box<T>` pointeur de tas et les pointeurs internes à `String` et aux `Vec` valeurs) possèdent des pointeurs : lorsque le propriétaire est supprimé, le référent l'accompagne. Rust a également des types de pointeurs non propriétaires appelés *références*, qui n'ont aucun effet sur la durée de vie de leurs référents.

En fait, c'est plutôt le contraire : les références ne doivent jamais survivre à leurs référents. Vous devez faire apparaître dans votre code qu'aucune référence ne peut survivre à la valeur vers laquelle elle pointe. Pour souligner cela, Rust se réfère à la création d'une référence à une certaine valeur en tant qu'*emprunt* la valeur : ce que vous avez emprunté, vous devez éventuellement le rendre à son propriétaire.

Si vous avez ressenti un moment de scepticisme en lisant la phrase « Vous devez le rendre apparent dans votre code », vous êtes en excellente compagnie. Les références elles-mêmes n'ont rien de spécial - sous le capot, ce ne sont que des adresses. Mais les règles qui assurent leur sécurité sont nouvelles pour Rust ; en dehors des langages de recherche, vous n'aurez jamais rien vu de tel auparavant. Et bien que ces règles soient la partie de Rust qui nécessite le plus d'efforts à maîtriser, l'ampleur des bogues classiques et absolument quotidiens qu'elles empêchent est surprenante, et leur effet sur la programmation multithread est libérateur. C'est encore le pari radical de Rust.

Dans ce chapitre, nous verrons comment les références fonctionnent dans Rust ; montrer comment les références, les fonctions et les types définis par l'utilisateur intègrent tous des informations de durée de vie pour garantir qu'ils sont utilisés en toute sécurité ; et illustrent certaines catégories courantes de bogues que ces efforts évitent, au moment de la compilation et sans pénalités de performances à l'exécution.

Références aux valeurs

Par exemple, supposons que nous allons construire une table d'artistes meurtriers de la Renaissance et des œuvres pour lesquelles ils sont connus. La bibliothèque standard de Rust inclut un type de table de hachage, nous pouvons donc définir notre type comme ceci :

```
use std:: collections::HashMap;

type Table = HashMap<String, Vec<String>>;
```

En d'autres termes, il s'agit d'une table de hachage qui mappe des `String` valeurs à des `Vec<String>` valeurs, prenant le nom d'un artiste dans une liste des noms de leurs œuvres. Vous pouvez parcourir les entrées de `a HashMap` avec une `for` boucle, nous pouvons donc écrire une fonction pour imprimer `a Table`:

```
fn show(table:Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Construire et imprimer le tableau est simple :

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                vec![ "many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                vec![ "The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                vec![ "Perseus with the head of Medusa".to_string(),
                      "a salt cellar".to_string()]);
    show(table);
}
```

Et tout fonctionne bien :

```

$course de fret
    Running `/home/jimb/rust/book/fragments/target/debug/fragments` 
works by Gesualdo:
    many madrigals
    Tenebrae Responsoria
works by Cellini:
    Perseus with the head of Medusa
    a salt cellar
works by Caravaggio:
    The Musicians
    The Calling of St. Matthew
$
```

Mais si vous avez lu la section du chapitre précédent sur les mouvements, cette définition de `show` devrait soulever quelques questions. En particulier, `HashMap` n'est pas `Copy` — il ne peut pas l'être, puisqu'il possède une table allouée dynamiquement. Ainsi, lorsque le programme appelle `show(table)`, toute la structure est déplacée vers la fonction, laissant la variable `table` non initialisée. (Il parcourt également son contenu sans ordre spécifique, donc si vous avez un ordre différent, ne vous inquiétez pas.) Si le code appelant essaie d'utiliser `table` maintenant, il rencontrera des problèmes :

```

...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust se plaint qu'il `table` n'est plus disponible :

```

error: borrow of moved value: `table`
|
20 |     let mut table = Table::new();
|     ----- move occurs because `table` has type
|           `HashMap<String, Vec<String>>`,
|           which does not implement the `Copy` trait
...
31 |     show(table);
|     ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
|           ^^^^^ value borrowed here after move
```

En fait, si nous examinons la définition de `show`, la boucle externe `for` s'approprie la table de hachage et la consomme entièrement ; et la boucle interne `for` fait de même pour chacun des vecteurs. (Nous avons

vu ce comportement plus tôt, dans l'exemple "liberté, égalité, fraternité"). En raison de la sémantique des mouvements, nous avons complètement détruit la structure entière simplement en essayant de l'imprimer. Merci Rust!

La bonne façon de gérer cela est d'utiliser des références. Une référence vous permet d'accéder à une valeur sans affecter sa propriété. Les références sont de deux sortes :

- Une *référence partagée* permet de lire mais pas de modifier son référent. Cependant, vous pouvez avoir autant de références partagées à une valeur particulière à la fois que vous le souhaitez. L'expression `&e` produit une référence partagée à `e` la valeur de ; si `e` a le type `T`, alors `&e` a le type `&T`, prononcé « ref `T` ». Les références partagées sont `Copy`.
- Si vous avez une *référence mutable* à une valeur, vous pouvez à la fois lire et modifier la valeur. Cependant, vous ne pouvez avoir aucune autre référence d'aucune sorte à cette valeur active en même temps. L'expression `&mut e` donne une référence mutable à `e` la valeur de ; vous écrivez son type comme `&mut T`, qui se prononce « ref muet `T` ». Les références mutables ne le sont pas `Copy`.

Vous pouvez considérer la distinction entre les références partagées et mutables comme un moyen d'imposer une *multiplicité de lecteurs ou auuteur unique* règle au moment de la compilation. En fait, cette règle ne s'applique pas uniquement aux références ; elle couvre également le propriétaire de la valeur empruntée. Tant qu'il existe des références partagées à une valeur, même son propriétaire ne peut pas la modifier ; la valeur est verrouillée. Personne ne peut modifier `table` pendant qu'il `show` travaille avec. De même, s'il existe une référence mutable à une valeur, elle a un accès exclusif à la valeur ; vous ne pouvez pas du tout utiliser le propriétaire, jusqu'à ce que la référence mutable disparaisse. Garder le partage et la mutation complètement séparés s'avère essentiel à la sécurité de la mémoire, pour des raisons que nous aborderons plus loin dans ce chapitre.

La fonction d'impression dans notre exemple n'a pas besoin de modifier la `table`, il suffit de lire son contenu. L'appelant doit donc pouvoir lui transmettre une référence partagée à la `table`, comme suit :

```
show(&table);
```

Les références ne sont pas propriétaires des pointeurs, de sorte que la table variable reste propriétaire de toute la structure ; show vient de l'emprunter un peu. Naturellement, nous devrons ajuster la définition de show pour qu'elle corresponde, mais vous devrez regarder attentivement pour voir la différence :

```
fn show(table:&Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Le type du show paramètre table de est passé de Table à &Table : au lieu de passer la table par valeur (et donc de déplacer la propriété dans la fonction), nous passons maintenant une référence partagée. C'est le seul changement textuel. Mais comment cela se passe-t-il lorsque nous travaillons à travers le corps ?

for Alors que notre boucle externe d'origine s'est emparée du HashMap et l'a consommé, dans notre nouvelle version, elle reçoit une référence partagée au HashMap . L'itération sur une référence partagée à a HashMap est définie pour produire des références partagées à la clé et à la valeur de chaque entrée : artist a changé de a String à a &String , et works de a Vec<String> à a &Vec<String> .

La boucle intérieure est modifiée de la même manière. L'itération sur une référence partagée à un vecteur est définie pour produire des références partagées à ses éléments, il en work va de même maintenant pour un &String . Aucune propriété ne change de mains dans cette fonction ; il ne s'agit que de faire circuler des références non propriétaires.

Maintenant, si nous voulions écrire une fonction pour classer par ordre alphabétique les œuvres de chaque artiste, une référence partagée ne suffit pas, car les références partagées ne permettent pas de modification. Au lieu de cela, la fonction de tri doit prendre une référence mutable à la table :

```
fn sort_works(table:&mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

```
}
```

Et nous devons en passer un :

```
sort_works(&mut table);
```

Cet emprunt mutable donne `sort_works` la possibilité de lire et de modifier notre structure, comme l'exige la méthode des vecteurs `sort`.

Lorsque nous passons une valeur à une fonction d'une manière qui transfère la propriété de la valeur à la fonction, nous disons que nous l'avons transmise *par valeur*. Si nous passons à la place à la fonction une référence à la valeur, nous disons que nous avons passé la valeur *par référence*. Par exemple, nous avons corrigé notre `show` fonction en la modifiant pour accepter la table par référence plutôt que par valeur. De nombreuses langues établissent cette distinction, mais elle est particulièrement importante dans Rust, car elle explique comment la propriété est affectée.

Travailler avec des références

L'exemple précédent montre une utilisation assez typique des références : permettre aux fonctions d'accéder ou de manipuler une structure sans en prendre possession. Mais les références sont plus flexibles que cela, alors regardons quelques exemples pour avoir une vue plus détaillée de ce qui se passe.

Références Rust versus références C++

Si vous êtes familier avec des références en C++, elles ont quelque chose en commun avec les références Rust. Plus important encore, ce ne sont que des adresses au niveau de la machine. Mais en pratique, les références de Rust ont une sensation très différente.

En C++, les références sont créées implicitement par conversion et déréférées implicitement également :

```
// C++ code!
int x = 10;
int &r = x;           // initialization creates reference implicitly
```

```
assert(r == 10);           // implicitly dereference r to see x's value
r = 20;                   // stores 20 in x, r itself still points to x
```

Dans Rust, les références sont créées explicitement avec l' `&` opérateur et déréférencées explicitement avec l' `*` opérateur:

```
// Back to Rust code from this point onward.
let x = 10;
let r = &x;               // &x is a shared reference to x
assert!(*r == 10);       // explicitly dereference r
```

Pour créer une référence mutable, utilisez l' `&mut` opérateur :

```
let mut y = 32;
let m = &mut y;          // &mut y is a mutable reference to y
*m += 32;                // explicitly dereference m to set y's value
assert!(*m == 64);       // and to see y's new value
```

Mais vous vous souviendrez peut-être que, lorsque nous avons fixé la `show` fonction pour prendre la table des artistes par référence plutôt que par valeur, nous n'avons jamais eu à utiliser l' `*` opérateur. Pourquoi donc?

Étant donné que les références sont si largement utilisées dans Rust, l' `.` opérateur déréférence implicitement son opérande gauche, si nécessaire :

```
struct Anime { name: &'static str, bechdel_pass: bool }
let aria = Anime { name: "Aria: The Animation", bechdel_pass:true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Equivalent to the above, but with the dereference written out:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

La `println!` macro utilisée dans la `show` fonction s'étend au code qui utilise l' `.` opérateur, elle tire donc également parti de ce déréférence-ment implicite.

L' `.` opérateur peut également emprunter implicitement une référence à son opérande gauche, si nécessaire pour un appel de méthode. Par exemple, avec la `sort` méthode de prend une référence mutable au vecteur, donc ces deux appels sont équivalents :

```

let mut v = vec![1973, 1968];
v.sort();           // implicitly borrows a mutable reference to v
(&mut v).sort();   // equivalent, but more verbose

```

En un mot, alors que C++ convertit implicitement entre les références et les lvalues (c'est-à-dire des expressions faisant référence à des emplacements en mémoire), ces conversions apparaissent partout où elles sont nécessaires, dans Rust vous utilisez les opérateurs `&` et `*` pour créer et suivre des références, à l'exception de l' `.` opérateur, qui emprunte et déréférence implicitement.

Affectation de références

Affectation une référence à une variable fait pointer cette variable quelque part de nouveau :

```

let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);

```

La référence `r` pointe initialement vers `x`. Mais si `b` est vrai, le code le pointe à la `y` place, comme illustré à la [Figure 5-1](#) .

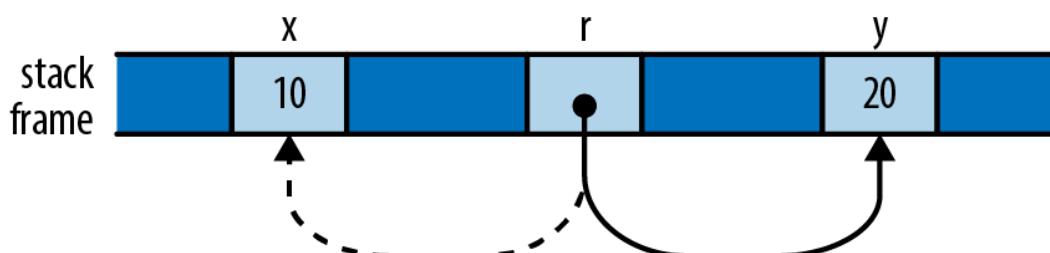


Figure 5-1. La référence `r`, pointant maintenant vers au `y` lieu de `x`

Ce comportement peut sembler trop évident pour mériter d'être mentionné : bien sûr `r` pointe maintenant vers `y` , puisque nous y avons stocké `&y` . Mais nous le soulignons car les références C++ se comportent très différemment : comme indiqué précédemment, l'attribution d'une valeur à une référence en C++ stocke la valeur dans son référent. Une fois qu'une référence C++ a été initialisée, il n'y a aucun moyen de la faire pointer vers autre chose.

Références aux références

Rust autorise les références aux références:

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr:&&&Point = &rr;
```

(Nous avons écrit les types de référence pour plus de clarté, mais vous pouvez les omettre ; il n'y a rien ici que Rust ne puisse déduire par lui-même.) L' `.` opérateur suit autant de références qu'il en faut pour trouver sa cible :

```
assert_eq!(rrr.y, 729);
```

En mémoire, les références sont disposées comme illustré à la [Figure 5-2](#) .

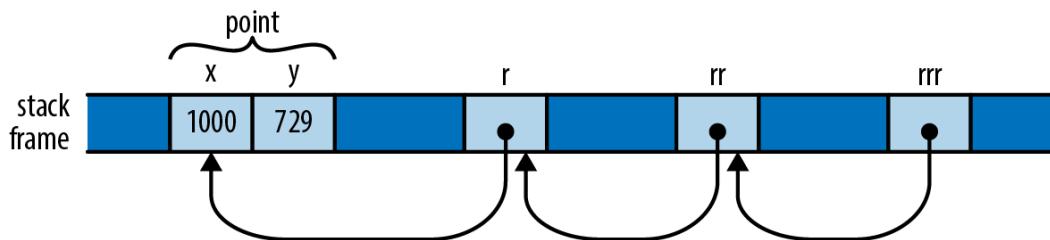


Figure 5-2. Une chaîne de références à références

Ainsi, l'expression `rrr.y`, guidée par le type de `rrr`, traverse en fait trois références pour atteindre le `Point` avant de récupérer son `y` champ.

Comparer des références

Comme l' `.` opérateur, la comparaison de Rust les opérateurs "voient à travers" n'importe quel nombre de références :

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;
```

```
assert!(rrx <= rry);  
assert!(rrx == rry);
```

L'assertion finale réussit ici, même si `rrx` et `rry` pointent vers des valeurs différentes (à savoir, `rx` et `ry`), car l'`==` opérateur suit toutes les références et effectue la comparaison sur leurs cibles finales, `x` et `y`. C'est presque toujours le comportement que vous souhaitez, en particulier lors de l'écriture de fonctions génériques. Si vous voulez réellement savoir si deux références pointent vers la même mémoire, vous pouvez utiliser `std::ptr::eq`, qui les compare en tant qu'adresses :

```
assert!(rx == ry);           // their referents are equal  
assert!(!std::ptr::eq(rx, ry)); // but occupy different addresses
```

Notez que les opérandes d'une comparaison doivent avoir exactement le même type, y compris les références :

```
assert!(rx == rrx);      // error: type mismatch: `&i32` vs `&&i32`  
assert!(rx == *rrx);     // this is okay
```

Les références ne sont jamais nulles

Les références de rouille ne sont jamais nulles. Il n'y a pas d'analogie au C `NULL` ou au C++ `nullptr`. Il n'y a pas de valeur initiale par défaut pour une référence (vous ne pouvez utiliser aucune variable tant qu'elle n'a pas été initialisée, quel que soit son type) et Rust ne convertira pas les entiers en références (en dehors du `unsafe` code), vous ne pouvez donc pas convertir zéro en une référence.

Le code C et C++ utilise souvent un pointeur nul pour indiquer l'absence de valeur : par exemple, la `malloc` fonction renvoie soit un pointeur vers un nouveau bloc de mémoire, soit `nullptr` s'il n'y a pas assez de mémoire disponible pour satisfaire la requête. Dans Rust, si vous avez besoin d'une valeur qui soit une référence à quelque chose ou non, utilisez le type `Option<&T>`. Au niveau de la machine, Rust représente `None` un pointeur nul et `Some(r)`, où `r` est une `&T` valeur, une adresse différente de zéro, `Option<&T>` est donc tout aussi efficace qu'un pointeur nullable en C ou C++, même s'il est plus sûr : son type vous oblige à vérifier s'il est `None` avant de pouvoir l'utiliser.

Emprunter des références à des expressions arbitraires

Alors que C et C++ ne vous permettent d'appliquer l'opérateur qu'à certains types d'expressions, Rust vous permet d'emprunter une référence à la valeur de n'importe quel type d'expression :

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

Dans des situations comme celle-ci, Rust crée simplement une variable anonyme pour contenir la valeur de l'expression et en fait le point de référence. La durée de vie de cette variable anonyme dépend de ce que vous faites avec la référence :

- Si vous affectez immédiatement la référence à une variable dans une `let` instruction (ou en faites une partie d'une structure ou d'un tableau qui est immédiatement affecté), alors Rust fait vivre la variable anonyme tant que la variable `let` s'initialise. Dans l'exemple précédent, Rust ferait cela pour le référent de `r`.
- Sinon, la variable anonyme vit jusqu'à la fin de l'instruction englobante. Dans notre exemple, la variable anonyme créée pour contenir `1009` ne dure que jusqu'à la fin de l'`assert_eq!` instruction.

Si vous êtes habitué à C ou C++, cela peut sembler source d'erreurs. Mais rappelez-vous que Rust ne vous laissera jamais écrire de code qui produirait une référence pendante. Si la référence peut être utilisée au-delà de la durée de vie de la variable anonyme, Rust vous signalera toujours le problème au moment de la compilation. Vous pouvez ensuite corriger votre code pour conserver le référent dans une variable nommée avec une durée de vie appropriée.

Références aux tranches et aux objets de trait

Les references nous avons montré jusqu'ici sont toutes des adresses simples. Cependant, Rust comprend également deux types de *pointeurs gras*, des valeurs de deux mots portant l'adresse d'une certaine valeur,

ainsi que d'autres informations nécessaires pour mettre la valeur à utiliser.

Une référence à une tranche est un pointeur gras, portant l'adresse de départ de la tranche et sa longueur. Nous avons décrit les tranches en détail au [chapitre 3](#).

L'autre type de pointeur gras de Rust est un *objet de trait*, une référence à une valeur qui implémente un certain trait. Un objet de trait porte l'adresse d'une valeur et un pointeur vers l'implémentation du trait appropriée à cette valeur, pour invoquer les méthodes du trait. Nous couvrirons les objets de trait en détail dans "[Objets de trait](#)".

En plus de transporter ces données supplémentaires, les références d'objets slice et trait se comportent exactement comme les autres sortes de références que nous avons montrées jusqu'ici dans ce chapitre : elles ne possèdent pas leurs référents, elles ne sont pas autorisées à survivre à leurs référents, elles peuvent être mutable ou partagé, et ainsi de suite.

Sécurité de référence

Comme nous les avons présentées jusqu'ici, les références ressemblent à peu près à des pointeurs ordinaires en C ou C++. Mais ceux-ci ne sont pas sûrs; comment Rust garde-t-il ses références sous contrôle ? Peut-être que la meilleure façon de voir les règles en action est d'essayer de les enfreindre.

Pour transmettre les idées fondamentales, nous commencerons par les cas les plus simples, montrant comment Rust garantit que les références sont utilisées correctement dans un corps de fonction unique. Ensuite, nous verrons comment passer des références entre les fonctions et les stocker dans des structures de données. Il s'agit de donner auxdites fonctions et types de données *des paramètres de durée de vie*, que nous allons vous expliquer. Enfin, nous présenterons quelques raccourcis fournis par Rust pour simplifier les modèles d'utilisation courants. Tout au long, nous montrerons comment Rust signale le code défectueux et suggère souvent des solutions.

Emprunter une variable locale

Voici un cas assez évident. Vous ne pouvez pas emprunter une référence à une variable locale et sortez-la de la portée de la variable :

```

{
    let r;
{
    let x = 1;
    r = &x;
}
assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
}

```

Le compilateur Rust rejette ce programme, avec un message d'erreur détaillé :

```

error: `x` does not live long enough
|
7 |         r = &x;
|             ^^^ borrowed value does not live long enough
8 |     }
|     - `x` dropped here while still borrowed
9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
10| }

```

La plainte de Rust est qu'il `x` ne vit que jusqu'à la fin du bloc intérieur, alors que la référence reste vivante jusqu'à la fin du bloc extérieur, ce qui en fait un pointeur suspendu, ce qui est verboten.

Bien qu'il soit évident pour un lecteur humain que ce programme est cassé, il vaut la peine de regarder comment Rust lui-même est parvenu à cette conclusion. Même cet exemple simple montre les outils logiques que Rust utilise pour vérifier un code beaucoup plus complexe.

Rust essaie d'attribuer à chaque type de référence de votre programme une *durée* de vie qui respecte les contraintes imposées par son utilisation. Une durée de vie est une partie de votre programme pour laquelle une référence peut être utilisée en toute sécurité : une instruction, une expression, la portée d'une variable, etc. Les durées de vie sont entièrement le fruit de l'imagination de Rust au moment de la compilation. Au moment de l'exécution, une référence n'est rien d'autre qu'une adresse ; sa durée de vie fait partie de son type et n'a pas de représentation à l'exécution.

Dans cet exemple, il y a trois vies dont nous devons établir les relations. Les variables `r` et `x` les deux ont une durée de vie, s'étendant du moment où elles sont initialisées jusqu'au moment où le compilateur peut prouver qu'elles ne sont plus utilisées. La troisième durée de vie est celle d'un type

de référence : le type de la référence que nous empruntons `x` et stockons dans `r`.

Voici une contrainte qui devrait sembler assez évidente : si vous avez une variable `x`, alors une référence à `x` ne doit pas survivre à `x` elle-même, comme le montre la [figure 5-3](#).

Au-delà du point où `x` sort de la portée, la référence serait un pointeur pendant. On dit que la durée de vie de la variable doit *contenir* ou *enfermer* celle de la référence qui lui est empruntée.

```
{  
    let r;  
    {  
        let x = 1;  
        ...  
        r = &x;  
        ...  
    }  
    assert_eq!(*r, 1);  
}
```

lifetime of &x must not exceed this range

Figure 5-3. Durées de vie autorisées pour `&x`

Voici un autre type de contrainte : si vous stockez une référence dans une variable `r`, le type de la référence doit être bon pour toute la durée de vie de la variable, depuis son initialisation jusqu'à sa dernière utilisation, comme illustré à la [Figure 5-4](#).

Si la référence ne peut pas vivre au moins aussi longtemps que la variable, alors à un moment donné `r`, il y aura un pointeur pendant. On dit que la durée de vie de la référence doit contenir ou enfermer celle de la variable.

```

{
    let r;
{
    let x = 1;
    ...
    r = &x;
    ...
}
assert_eq!(*r, 1);
}

```

lifetime of anything stored in
r must cover at least this range

Figure 5-4. Durées de vie admissibles pour la référence stockée dans r

Le premier type de contrainte limite la durée de vie d'une référence, tandis que le second limite sa taille. Rust essaie simplement de trouver une durée de vie pour chaque référence qui satisfait toutes ces contraintes. Dans notre exemple, cependant, cette durée de vie n'existe pas, comme le montre la [Figure 5-5](#).

```

{
    let r;
{
    let x = 1;
    ...
    r = &x;
    ...
}
assert_eq!(*r, 1);
}

```

There is no lifetime that lies
entirely within this range...

...but also fully encloses this range.

Figure 5-5. Une référence aux contraintes contradictoires sur sa durée de vie

Considérons maintenant un exemple différent où les choses fonctionnent. Nous avons les mêmes types de contraintes : la durée de vie de la référence doit être contenue par `x`'s, mais enfermer complètement `r`'s. Mais comme `r` la durée de vie de `x` est maintenant plus petite, il existe une durée de vie qui respecte les contraintes, comme le montre la [figure 5-6](#).

```

{
    let x = 1;
    {
        let r = &x;
        ...
        assert_eq! (*r, 1);
        ...
    }
}

```

The inner lifetime covers the lifetime of `r`, but is fully enclosed by the lifetime of `x`.

Figure 5-6. Une référence avec une durée de vie englobant `r` la portée de , mais dans `x` la portée de

Ces règles s'appliquent de manière naturelle lorsque vous empruntez une référence à une partie d'une structure de données plus large, comme un élément d'un vecteur :

```

let v = vec![1, 2, 3];
let r = &v[1];

```

Puisque `v` possède le vecteur, qui possède ses éléments, la durée de vie de `v` doit contenir celle du type de référence de `&v[1]`. De même, si vous stockez une référence dans une structure de données, sa durée de vie doit contenir celle de la structure de données. Par exemple, si vous construisez un vecteur de références, toutes doivent avoir des durées de vie renfermant celle de la variable qui possède le vecteur.

C'est l'essence du processus que Rust utilise pour tout le code. Apporter plus de fonctionnalités de langage dans l'image - par exemple, les structures de données et les appels de fonctions - introduit de nouvelles sortes de contraintes, mais le principe reste le même : premièrement, comprendre les contraintes résultant de la façon dont le programme utilise les références ; puis, trouvez des durées de vie qui les satisfont. Ce n'est pas si différent du processus que les programmeurs C et C++ s'imposent à eux-mêmes ; la différence est que Rust connaît les règles et les applique.

Recevoir des références en tant qu'arguments de fonction

Quand on passe une référence à une fonction, comment Rust s'assure-t-il que la fonction l'utilise en toute sécurité ? Supposons que nous ayons une fonction `f` qui prend une référence et la stocke dans une variable globale. Nous devrons apporter quelques révisions à cela, mais voici une première coupe :

```
// This code has several problems, and doesn't compile.  
static mut STASH: &i32;  
fn f(p:&i32) { STASH = p; }
```

L'équivalent de Rust d'une variable globale est appelé un *statique*: c'est une valeur créée au démarrage du programme et qui dure jusqu'à ce qu'il se termine. ([Comme toute autre déclaration, le système de module de Rust contrôle où les statiques sont visibles](#), donc elles ne sont que "globales" dans leur durée de vie, pas leur visibilité.) règles que le code qui vient d'être affiché ne suit pas :

- Chaque statique doit être initialisé.
- Les statiques mutables ne sont par nature pas thread-safe (après tout, n'importe quel thread peut accéder à un statique à tout moment), et même dans les programmes à un seul thread, ils peuvent être la proie d'autres types de problèmes de réentrance. Pour ces raisons, vous ne pouvez accéder à un statique mutable qu'à l'intérieur d'un `unsafe` bloc. Dans cet exemple, nous ne sommes pas concernés par ces problèmes particuliers, nous allons donc ajouter un `unsafe` bloc et passer à autre chose.

Avec ces révisions effectuées, nous avons maintenant ce qui suit :

```
static mut STASH: &i32 = &128;  
fn f(p:&i32) { // still not good enough  
    unsafe {  
        STASH = p;  
    }  
}
```

Nous avons presque terminé. Pour voir le problème restant, nous devons écrire quelques éléments que Rust nous laisse utilement omettre. La signature de `f` tel qu'écrit ici est en fait un raccourci pour ce qui suit :

```
fn f<'a>(p:&'a i32) { ... }
```

Ici, la durée de vie '`a` (prononcez "tick A") est un *paramètre de durée de vie* de `f`. Vous pouvez lire `<'a>` "pour toute durée de vie '`a`", donc lorsque nous écrivons `fn f<'a>(p: &'a i32)`, nous définissons une fonction qui prend une référence à un `i32` avec une durée de vie donnée '`a`.

Puisque nous devons autoriser 'a n'importe quelle durée de vie, il vaudrait mieux que les choses se passent si c'est la plus petite durée de vie possible : une qui enferme juste l'appel à `f`. Cette affectation devient alors un point de discorde :

```
STASH = p;
```

Puisque `STASH` vit pour toute l'exécution du programme, le type de référence qu'il contient doit avoir une durée de vie de la même longueur ; Rust appelle cela la '`static` durée de vie'. Mais la durée de vie de `p` la référence est `some 'a`, qui peut être n'importe quoi, tant qu'elle contient l'appel à `f`. Ainsi, Rust rejette notre code :

```
error: explicit lifetime required in the type of `p`  
|  
5 |     STASH = p;  
|           ^ lifetime `static` required
```

À ce stade, il est clair que notre fonction ne peut pas accepter n'importe quelle référence comme argument. Mais comme le souligne Rust, il devrait être capable d'accepter une référence qui a une '`static` durée de vie' : stocker une telle référence dans `STASH` ne peut pas créer de pointeur pendant. Et en effet, le code suivant se compile parfaitement :

```
static mut STASH:&i32 = &10;  
  
fn f(p:&'static i32) {  
    unsafe {  
        STASH = p;  
    }  
}
```

Cette fois, `f` la signature de précise qu'il `p` doit s'agir d'une référence avec `life 'static`, il n'y a donc plus de problème à stocker cela dans `STASH`. Nous ne pouvons appliquer que `f` des références à d'autres statiques, mais c'est la seule chose qui ne restera pas en `STASH` suspens de toute façon. Alors on peut écrire :

```
static WORTH_POINTING_AT:i32 = 1000;  
f(&WORTH_POINTING_AT);
```

Étant donné `WORTH_POINTING_AT` que est un statique, le type de `&WORTH_POINTING_AT` est `&'static i32`, qui peut être transmis en toute sécurité à `f`.

Prenez du recul, cependant, et remarquez ce qui est arrivé à `f` la signature de lorsque nous avons modifié notre chemin vers l'exactitude : l'original `f(p: &i32)` s'est terminé par `f(p: &'static i32)`. En d'autres termes, nous étions incapables d'écrire une fonction qui stockait une référence dans une variable globale sans refléter cette intention dans la signature de la fonction. Dans Rust, la signature d'une fonction expose toujours le comportement du corps.

Inversement, si nous voyons une fonction avec une signature comme `g(p: &i32)` (ou avec les durées de vie écrites, `g<'a>(p: &'a i32)`), nous pouvons dire qu'elle *ne cache pas* son argument `p` n'importe où qui survivra à l'appel. Il n'est pas nécessaire d'examiner `g` la définition de ; la signature seule nous dit ce qui `g` peut et ne peut pas faire avec son argument. Ce fait finit par être très utile lorsque vous essayez d'établir la sécurité d'un appel à la fonction.

Passer des références aux fonctions

À présentque nous avons montré comment la signature d'une fonction est liée à son corps, examinons comment elle est liée aux appelants de la fonction. Supposons que vous ayez le code suivant :

```
// This could be written more briefly: fn g(p: &i32),
// but let's write out the lifetimes for now.
fn g<'a>(p:&'a i32) { ... }

let x = 10;
g(&x);
```

À partir `g` de la seule signature de , Rust sait qu'il n'enregistrera `p` aucun élément susceptible de survivre à l'appel : toute durée de vie qui entoure l'appel doit fonctionner pour `'a`. Donc Rust choisit la plus petite durée de vie possible pour `&x` : celle de l'appel à `g`. Cela répond à toutes les contraintes : il ne survit pas à `x`, et il contient l'intégralité de l'appel à `g`. Donc, ce code passe le cap.

Notez que bien que `g` prenant un paramètre de durée de vie `'a`, nous n'avons pas besoin de le mentionner lors de l'appel `g`. Vous n'avez qu'à vous soucier des paramètres de durée de vie lors de la définition des

fonctions et des types ; lors de leur utilisation, Rust déduit les durées de vie pour vous.

Et si nous essayons de passer `&x` à notre fonction `f` précédente qui stocke son argument dans un statique ?

```
fn f(p:&'static i32) { ... }

let x = 10;
f(&x);
```

Cela échoue à compiler : la référence `&x` ne doit pas survivre à `x`, mais en la passant à `f`, nous la contraignons à vivre au moins aussi longtemps que `'static`. Il n'y a aucun moyen de satisfaire tout le monde ici, donc Rust rejette le code.

Renvoyer des références

C'est courant pour qu'une fonction prenne une référence à une structure de données, puis renvoie une référence dans une partie de cette structure. Par exemple, voici une fonction qui renvoie une référence au plus petit élément d'une tranche :

```
// v should have at least one element.
fn smallest(v: &[i32]) ->&i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

Nous avons omis les durées de vie de la signature de cette fonction de la manière habituelle. Lorsqu'une fonction prend une seule référence comme argument et renvoie une seule référence, Rust suppose que les deux doivent avoir la même durée de vie. Écrire cela explicitement nous donnerait :

```
fn smallest<'a>(v: &'a [i32]) ->&'a i32 { ... }
```

Supposons que nous appelons `smallest` ainsi :

```

let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // bad: points to element of dropped array

```

À partir de la signature de `smallest`, nous pouvons voir que son argument et sa valeur de retour doivent avoir la même durée de vie, 'a.

Dans notre appel, l'argument `¶bola` ne doit pas survivre à `parabola` lui-même, mais `smallest` la valeur de retour de doit vivre au moins aussi longtemps que `s`. Il n'y a pas de durée de vie possible 'a qui puisse satisfaire les deux contraintes, donc Rust rejette le code :

```

error: `parabola` does not live long enough
|
11 |         s = smallest(&parabola);
|                         ----- borrow occurs here
12 |     }
|     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // bad: points to element of dropped array
|                         - borrowed value needs to live until here
14 | }

```

Déplacer `s` de sorte que sa durée de vie soit clairement contenue dans `parabola`'s résout le problème :

```

{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}

```

Les durées de vie dans les signatures de fonction permettent à Rust d'évaluer les relations entre les références que vous transmettez à la fonction et celles que la fonction renvoie, et elles garantissent qu'elles sont utilisées en toute sécurité.

Structures contenant des références

Comment Rust gère-t-il les références stocké dans des structures de données ? Voici le même programme erroné que nous avons vu plus tôt, sauf que nous avons placé la référence dans une structure :

```
// This does not compile.

struct S {
    r:&i32
}

let s;
{
    let x = 10;
    s = S { r:&x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`
```

Les contraintes de sécurité que Rust place sur les références ne peuvent pas disparaître comme par magie simplement parce que nous avons caché la référence dans une structure. D'une manière ou d'une autre, ces contraintes doivent finir par s'appliquer `s` également. En effet, Rust est sceptique :

```
error: missing lifetime specifier
|
7 |     r: &i32
|           ^ expected lifetime parameter
```

Chaque fois qu'un type de référence apparaît dans la définition d'un autre type, vous devez écrire sa durée de vie. Vous pouvez écrire ceci :

```
struct S {
    r:&'static i32
}
```

Cela signifie que `r` cela ne peut faire référence qu'à des `i32` valeurs qui dureront pendant toute la durée de vie du programme, ce qui est plutôt limitatif. L'alternative est de donner au type un paramètre de durée de vie '`a` et de l'utiliser pour `r`:

```
struct S<'a> {
    r:&'a i32
}
```

Désormais, le `s` type a une durée de vie, tout comme les types de référence. Chaque valeur que vous créez de type `s` obtient une nouvelle durée de vie '`a`', qui devient contrainte par la façon dont vous utilisez la valeur. La durée de vie de toute référence dans laquelle vous stockez `r` doit

contenir '`a`', et '`a`' doit durer plus longtemps que la durée de vie de l'endroit où vous stockez le fichier `s`.

En revenant au code précédent, l'expression `s { r: &x }` crée une nouvelle `s` valeur avec une durée de vie '`a`'. Lorsque vous stockez `&x` sur le `r` terrain, vous vous engagez '`a`' à mentir entièrement pendant `x` la durée de vie de `.`

L'affectation `s = s { ... }` stocke ceci `s` dans une variable dont la durée de vie s'étend jusqu'à la fin de l'exemple, contraignant '`a`' à durer plus longtemps que la durée de vie de `s`. Et maintenant, Rust est arrivé aux mêmes contraintes contradictoires qu'auparavant : '`a`' ne doit pas survivre à `x`, mais doit vivre au moins aussi longtemps que `s`. Aucune durée de vie satisfaisante n'existe et Rust rejette le code. Catastrophe évitée !

Comment un type avec un paramètre de durée de vie se comporte-t-il lorsqu'il est placé dans un autre type ?

```
struct D {
    s:S // not adequate
}
```

Rust est sceptique, tout comme il l'était lorsque nous avons essayé de placer une référence dans `s` sans spécifier sa durée de vie :

```
error: missing lifetime specifier
|
8 |     s: S // not adequate
|         ^ expected named lifetime parameter
|
```

Nous ne pouvons pas laisser de côté `s` le paramètre de durée de vie de : Rust a besoin de savoir comment `D` la durée de vie de `s` est liée à celle de la référence dans son `s` afin d'appliquer les mêmes vérifications `D` que pour `s` les références simples.

On pourrait donner `s` la '`static`' durée de vie. Cela marche:

```
struct D {
    s:S<'static>
}
```

Avec cette définition, le `s` champ ne peut emprunter que des valeurs qui vivent pendant toute l'exécution du programme. C'est quelque peu restrictif, mais cela signifie qu'il `D` est impossible d'emprunter une variable locale ; il n'y a pas de contraintes particulières sur `D` la durée de vie de `s`.

Le message d'erreur de Rust suggère en fait une autre approche, plus générale :

```
help: consider introducing a named lifetime parameter
|
7 | struct D<'a> {
8 |     s: S<'a>
| }
```

Ici, nous donnons `D` son propre paramètre de durée de vie et le passons à `s` :

```
struct D<'a> {
    s:S<'a>
}
```

En prenant un paramètre de durée de vie `'a` et en l'utilisant dans `s` le type de `s`, nous avons permis à Rust de relier `D` la durée de vie de la valeur à celle de la référence qu'elle `s` contient.

Nous avons montré précédemment comment la signature d'une fonction expose ce qu'elle fait avec les références que nous lui passons. Maintenant, nous avons montré quelque chose de similaire à propos des types : les paramètres de durée de vie d'un type révèlent toujours s'il contient des références avec des durées de vie intéressantes (c'est-à-dire non `'static`) et quelles peuvent être ces durées de vie.

Par exemple, supposons que nous ayons une fonction d'analyse qui prend une tranche d'octets et renvoie une structure contenant les résultats de l'analyse :

```
fn parse_record<'i>(input: &'i [u8]) ->Record<'i> { ... }
```

Sans regarder du tout dans la définition du `Record` type, nous pouvons dire que, si nous recevons un `Record` from `parse_record`, toutes les références qu'il contient doivent pointer vers le tampon d'entrée que nous

avons transmis, et nulle part ailleurs (sauf peut-être aux `'static` valeurs).

En fait, cette exposition du comportement interne est la raison pour laquelle Rust exige que les types contenant des références prennent des paramètres de durée de vie explicites. Il n'y a aucune raison pour que Rust ne puisse pas simplement créer une durée de vie distincte pour chaque référence dans la structure et vous éviter d'avoir à les écrire. Les premières versions de Rust se comportaient en fait de cette façon, mais les développeurs trouvaient cela déroutant : il est utile de savoir quand une valeur emprunte quelque chose à une autre valeur, en particulier lorsque l'on travaille sur des erreurs.

Ce ne sont pas seulement les références et les types comme `s` ceux-là qui ont des durées de vie. Chaque type dans Rust a une durée de vie, y compris `i32` et `String`. La plupart sont simplement `'static`, ce qui signifie que les valeurs de ces types peuvent vivre aussi longtemps que vous le souhaitez ; par exemple, un `Vec<i32>` est autonome et n'a pas besoin d'être supprimé avant qu'une variable particulière ne sorte de la portée. Mais un type comme `Vec<&'a i32>` a une durée de vie qui doit être entourée par `'a` : il doit être supprimé tant que ses référents sont encore vivants.

Paramètres de durée de vie distincts

Supposons que vous ayez défini une structure contenant deux références comme celle-ci :

```
struct S<'a> {
    x: &'a i32,
    y:&'a i32
}
```

Les deux références utilisent la même durée de vie `'a`. Cela pourrait être un problème si votre code veut faire quelque chose comme ceci :

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y:&y };
        r = s.x;
```

```

    }
    println!("{}", r);
}

```

Ce code ne crée aucun pointeur pendant. La référence à `y` reste dans `s`, qui sort de la portée avant `y`. La référence à `x` se termine par `r`, qui ne survit `x` pas à .

Si vous essayez de compiler cela, cependant, Rust se plaindra de `y` ne pas vivre assez longtemps, même si c'est clairement le cas. Pourquoi Rust est-il inquiet ? Si vous parcourez attentivement le code, vous pouvez suivre son raisonnement :

- Les deux champs de `s` sont des références avec la même durée de vie '`a`', donc Rust doit trouver une seule durée de vie qui fonctionne pour `s.x` et `s.y`.
- Nous attribuons `r = s.x`, nécessitant '`a`' de joindre `r` la durée de vie de .
- Nous avons initialisé `s.y` avec `&y`, nécessitant '`a`' de ne pas dépasser la durée de `y` vie de .

Ces contraintes sont impossibles à satisfaire : aucune durée de vie n'est inférieure à `y` la portée de mais supérieure à celle `r` de . Boucliers de rouille.

Le problème survient parce que les deux références `s` ont la même durée de vie '`a`'. Changer la définition de `s` pour que chaque référence ait une durée de vie distincte corrige tout :

```

struct S<'a, 'b> {
    x: &'a i32,
    y:&'b i32
}

```

Avec cette définition, `s.x` et `s.y` ont des durées de vie indépendantes. Ce que nous faisons `s.x` n'a aucun effet sur ce que nous stockons dans `s.y`, il est donc facile de satisfaire les contraintes maintenant : '`a` peut simplement être `r` la durée de vie de , et '`b` peut être celle `s` de . (`y` La durée de vie de fonctionnerait aussi pour '`b`', mais Rust essaie de choisir la plus petite durée de vie qui fonctionne.) Tout finit bien.

Les signatures de fonction peuvent avoir des effets similaires. Supposons que nous ayons une fonction comme celle-ci :

```
fn f<'a>(r: &'a i32, s: &'a i32) ->&'a i32 { r } // perhaps too tight
```

Ici, les deux paramètres de référence utilisent la même durée de vie '`'a`', ce qui peut inutilement contraindre l'appelant de la même manière que nous l'avons montré précédemment. Si cela pose problème, vous pouvez laisser les durées de vie des paramètres varier indépendamment :

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) ->&'a i32 { r } // looser
```

L'inconvénient est que l'ajout de durées de vie peut rendre les types et les signatures de fonction plus difficiles à lire. Vos auteurs ont tendance à essayer d'abord la définition la plus simple possible, puis à assouplir les restrictions jusqu'à ce que le code soit compilé. Étant donné que Rust ne permettra pas au code de s'exécuter à moins qu'il ne soit sûr, attendre simplement d'être informé lorsqu'il y a un problème est une tactique parfaitement acceptable.

Omission des paramètres de durée de vie

Jusqu'à présent, nous avons montré de nombreuses fonctions dans ce livre qui renvoient des références ou les prennent comme paramètres, mais nous n'avons généralement pas eu besoin de préciser quelle durée de vie est laquelle. Les durées de vie sont là; Rust nous laisse simplement les omettre quand il est raisonnablement évident de savoir ce qu'ils devraient être.

Dans les cas les plus simples, vous n'aurez peut-être jamais besoin d'écrire des durées de vie pour vos paramètres. Rust attribue simplement une durée de vie distincte à chaque endroit qui en a besoin. Par exemple:

```
struct S<'a, 'b> {
    x: &'a i32,
    y:&'b i32
}

fn sum_r_xy(r: &i32, s: S) ->i32 {
    r + s.x + s.y
}
```

La signature de cette fonction est un raccourci pour :

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) ->i32
```

Si vous renvoyez des références ou d'autres types avec des paramètres de durée de vie, Rust essaie toujours de faciliter les cas non ambigus. S'il n'y a qu'une seule durée de vie qui apparaît parmi les paramètres de votre fonction, alors Rust suppose que toutes les durées de vie dans votre valeur de retour doivent être celle-là :

```
fn first_third(point: &[i32; 3]) ->(&i32, &i32) {
    (&point[0], &point[2])
}
```

Avec toutes les durées de vie écrites, l'équivalent serait :

```
fn first_third<'a>(point: &'a [i32; 3]) ->(&'a i32, &'a i32)
```

S'il y a plusieurs durées de vie parmi vos paramètres, il n'y a aucune raison naturelle de préférer l'un à l'autre pour la valeur de retour, et Rust vous oblige à préciser ce qui se passe.

Si votre fonction est une méthode sur un type et prend son `self` paramètre par référence, cela rompt le lien : Rust suppose que `self` la durée de vie est celle qui donne tout dans votre valeur de retour. (Un `self` paramètre fait référence à la valeur sur laquelle la méthode est appelée, l'équivalent de Rust `this` en C++, Java ou JavaScript, ou `self` en Python. Nous couvrirons les méthodes dans ["Définir des méthodes avec impl"](#).)

Par exemple, vous pouvez écrire ce qui suit :

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

La `find_by_prefix` signature de la méthode est un raccourci pour :

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) ->Option<&'a String>
```

Rust suppose que tout ce que vous empruntez, vous empruntez à `self`.

Encore une fois, ce ne sont que des abréviations, destinées à être utiles sans introduire de surprises. Quand ils ne sont pas ce que vous voulez, vous pouvez toujours écrire explicitement les durées de vie.

Partage contre mutation

Jusqu'à présent, nous avons discuté comment Rust garantit qu'aucune référence ne pointera jamais vers une variable qui est sortie de la portée. Mais il existe d'autres façons d'introduire des pointeurs pendant. Voici un cas facile :

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v; // move vector to aside
r[0];          // bad: uses `v`, which is now uninitialized
```

L'affectation à `aside` déplace le vecteur, le laissant `v` non initialisé, et se transforme `r` en un pointeur pendant, comme illustré à la [Figure 5-7](#).

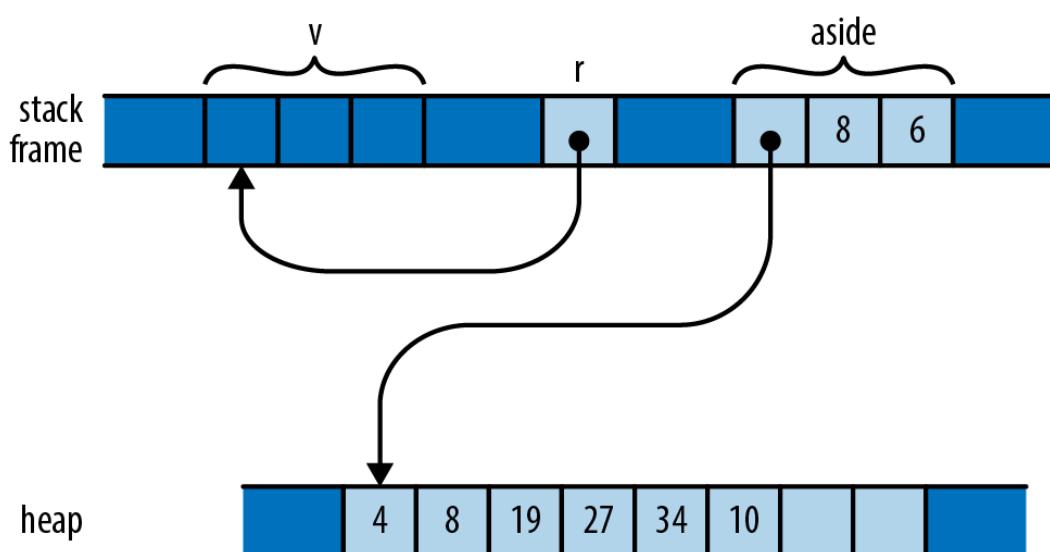


Figure 5-7. Une référence à un vecteur qui a été éloigné

Bien qu'il `v` reste dans la portée pendant `r` toute la durée de vie de , le problème ici est que `v` la valeur de est déplacée ailleurs, laissant `v` non initialisée tout `r` en s'y référant. Naturellement, Rust détecte l'erreur :

```

error: cannot move out of `v` because it is borrowed
|
9 |     let r = &v;
|         - borrow of `v` occurs here
10 |     let aside = v; // move vector to aside
|         ^^^^^ move out of `v` occurs here

```

Pendant toute sa durée de vie, une référence partagée rend son référent en lecture seule : vous ne pouvez pas l'affecter au référent ou déplacer sa valeur ailleurs. Dans ce code, `r` la durée de vie de contient la tentative de déplacement du vecteur, donc Rust rejette le programme. Si vous modifiez le programme comme indiqué ici, il n'y a pas de problème :

```

let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];           // ok: vector is still there
}
let aside = v;

```

Dans cette version, `r` sort plus tôt du champ d'application, la durée de vie de la référence se termine avant qu'elle ne `v` soit mise de côté, et tout va bien.

Voici une autre façon de faire des ravages. Supposons que nous ayons une fonction pratique pour étendre un vecteur avec les éléments d'une tranche :

```

fn extend(vec: &mut Vec<f64>, slice:&[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}

```

`extend_from_slice` Il s'agit d'une version moins flexible (et beaucoup moins optimisée) de la méthode de la bibliothèque standard sur les vecteurs. Nous pouvons l'utiliser pour construire un vecteur à partir de tranches d'autres vecteurs ou tableaux :

```

let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

```

```

extend(&mut wave, &head);    // extend wave with another vector
extend(&mut wave, &tail);    // extend wave with an array

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);

```

Nous avons donc construit une période d'une onde sinusoïdale ici. Si nous voulons ajouter une autre ondulation, pouvons-nous ajouter le vecteur à lui-même ?

```

extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                      0.0, 1.0, 0.0, -1.0]);

```

Cela peut sembler correct lors d'une inspection occasionnelle. Mais rappelez-vous que lorsque nous ajoutons un élément à un vecteur, si son tampon est plein, il doit allouer un nouveau tampon avec plus d'espace. Supposons `wave` qu'il commence par un espace pour quatre éléments et qu'il doive donc allouer un tampon plus grand quand `extend` essaie d'ajouter un cinquième. La mémoire finit par ressembler à la [figure 5-8](#).

L'argument `extend` de la fonction `vec` emprunte `wave` (appartenant à l'appelant), qui s'est alloué un nouveau tampon avec de l'espace pour huit éléments. Mais `slice` continue de pointer vers l'ancien tampon à quatre éléments, qui a été abandonné.

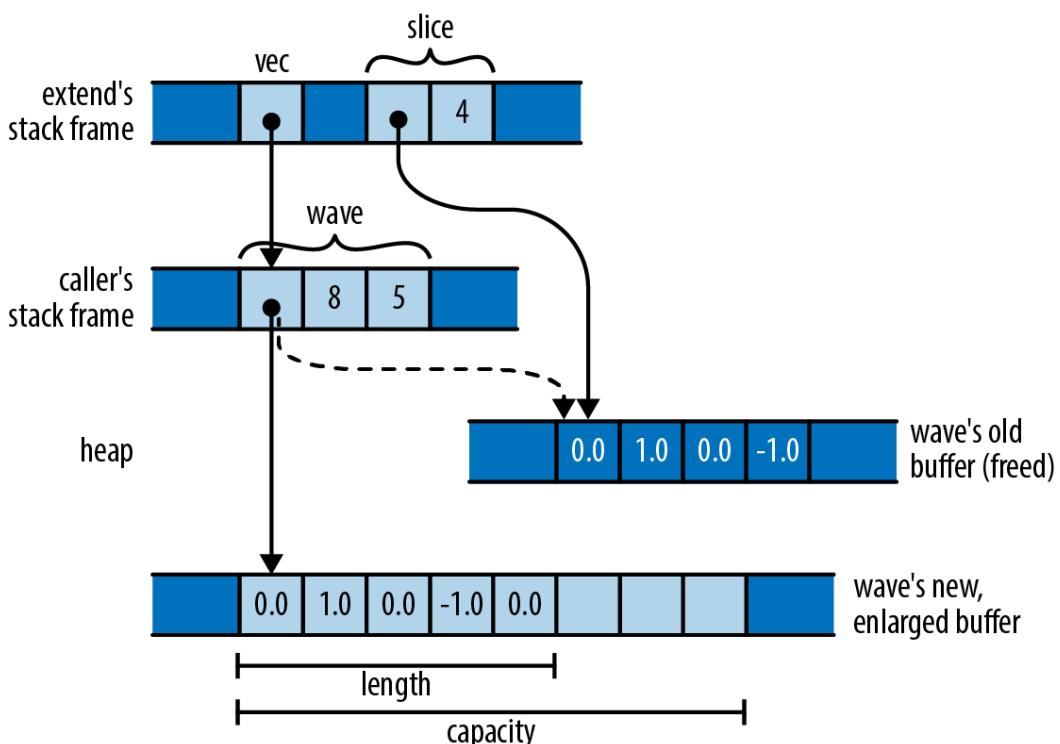


Figure 5-8. Une tranche transformée en un pointeur pendant par une réallocation de vecteur

Ce genre de problème n'est pas propre à Rust : modifier des collections tout en pointant dessus est un territoire délicat dans de nombreuses langues. En C++, la `std::vector` spécification vous avertit que "la réallocation [du tampon du vecteur] invalide toutes les références, pointeurs et itérateurs faisant référence aux éléments de la séquence". De même, dit Java, de modifier un `java.util.Hashtable` objet :

Si la table de hachage est structurellement modifiée à tout moment après la création de l'itérateur, de quelque manière que ce soit, sauf via la propre méthode remove de l'itérateur, l'itérateur lèvera un ConcurrentModificationException.

Ce qui est particulièrement difficile avec ce genre de bogue, c'est qu'il n'arrive pas tout le temps. Lors des tests, votre vecteur peut toujours disposer de suffisamment d'espace, le tampon peut ne jamais être réalloué et le problème peut ne jamais apparaître.

Rust, cependant, signale le problème avec notre appel à `extend` au moment de la compilation :

```
error: cannot borrow `wave` as immutable because it is also
      borrowed as mutable
|
9 |     extend(&mut wave, &wave);
|           ----- ^^^^-- mutable borrow ends here
|           |
|           |           |           immutable borrow occurs here
|           |           |           mutable borrow occurs here
```

En d'autres termes, nous pouvons emprunter une référence mutable au vecteur, et nous pouvons emprunter une référence partagée à ses éléments, mais les durées de vie de ces deux références ne doivent pas se chevaucher. Dans notre cas, les durées de vie des deux références contiennent l'appel à `extend`, donc Rust rejette le code.

Ces erreurs proviennent toutes deux de violations des règles de Rust pour la mutation et le partage :

L'accès partagé est un accès en lecture seule.

Valeurs empruntés par des références partagées sont en lecture seule. Tout au long de la durée de vie d'une référence partagée, ni son référent, ni quoi que ce soit d'accessible à partir de ce référent, ne peut être modifié *par quoi que ce soit*. Il n'existe aucune référence mutable en direct à quoi que ce soit dans cette

structure, son propriétaire est en lecture seule, et ainsi de suite. C'est vraiment gelé.

L'accès modifiable est un accès exclusif.

Une valeur empruntée par une référence mutable est accessible exclusivement via cette référence. Au cours de la durée de vie d'une référence mutable, il n'y a pas d'autre chemin utilisable vers son référent ou vers une valeur accessible à partir de là. Les seules références dont les durées de vie peuvent chevaucher une référence mutable sont celles que vous empruntez à la référence mutable elle-même.

Rust a signalé l' `extend` exemple comme une violation de la deuxième règle : puisque nous avons emprunté une référence mutable à `wave`, cette référence mutable doit être le seul moyen d'atteindre le vecteur ou ses éléments. La référence partagée à la tranche est elle-même un autre moyen d'atteindre les éléments, en violation de la deuxième règle.

Mais Rust aurait également pu traiter notre bogue comme une violation de la première règle : puisque nous avons emprunté une référence partagée aux `wave` éléments de `vec`, les éléments et le `vec` lui-même sont tous en lecture seule. Vous ne pouvez pas emprunter une référence mutable à une valeur en lecture seule.

Chaque type de référence affecte ce que nous pouvons faire avec les valeurs le long du chemin propriétaire vers le référent et les valeurs accessibles à partir du référent ([Figure 5-9](#)).

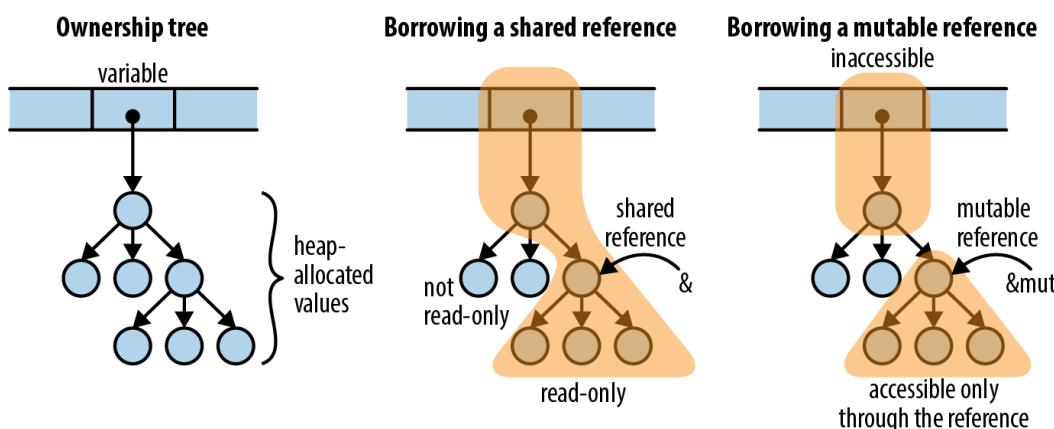


Figure 5-9. L'emprunt d'une référence affecte ce que vous pouvez faire avec d'autres valeurs dans le même arbre de propriété

Notez que dans les deux cas, le chemin de propriété menant au référent ne peut pas être modifié pendant la durée de vie de la référence. Pour un emprunt partagé, le chemin est en lecture seule ; pour un emprunt mutable, c'est complètement inaccessible. Il n'y a donc aucun moyen pour le programme de faire quoi que ce soit qui invalide la référence.

Réduisant ces principes aux exemples les plus simples possibles :

```
let mut x = 10;
let r1 = &x;
let r2 = &x;      // ok: multiple shared borrows permitted
x += 10;          // error: cannot assign to `x` because it is borrowed
let m = &mut x;  // error: cannot borrow `x` as mutable because it is
                 // also borrowed as immutable
println!("{}", {}, {}", r1, r2, m); // the references are used here,
                                     // so their lifetimes must last
                                     // at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y; // error: cannot borrow as mutable more than once
let z = y;        // error: cannot use `y` because it was mutably borrowed
println!("{}", {}, {}, m1, m2, z); // references are used here
```

Il est acceptable de réemprunter une référence partagée à partir d'une référence partagée :

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;      // ok: reborrowing shared as shared
let m1 = &mut r.1;  // error: can't reborrow shared as mutable
println!("{}", r0); // r0 gets used here
```

Vous pouvez réemprunter à partir d'une référence mutable :

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;    // ok: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;        // ok: reborrowing shared from mutable,
                     // and doesn't overlap with m0
v.1;                  // error: access through other paths still forbid
println!("{}", r1);   // r1 gets used here
```

Ces restrictions sont assez strictes. Pour en revenir à notre tentative d'appel `extend(&mut wave, &wave)`, il n'y a pas de moyen rapide et facile de corriger le code pour qu'il fonctionne comme nous le souhaitons. Et Rust applique ces règles partout : si nous empruntons, disons, une référence partagée à une clé dans un `HashMap`, nous ne pouvons pas emprunter une référence mutable à la `HashMap` tant que la durée de vie de la référence partagée n'est pas terminée.

Mais il y a une bonne justification à cela : concevoir des collections pour prendre en charge une itération et une modification simultanées et illimitées est difficile et empêche souvent des implémentations plus simples et plus efficaces. Java `Hashtable` et C++ `vector` ne dérangent pas, et ni les dictionnaires Python ni les objets JavaScript ne définissent exactement comment se comporte un tel accès. D'autres types de collections en JavaScript le font, mais nécessitent par conséquent des implémentations plus lourdes. La promesse de C++ `std::map` que l'insertion de nouvelles entrées n'invaliderait pas les pointeurs vers d'autres entrées de la carte, mais en faisant cette promesse, la norme empêche les conceptions plus efficaces en termes de cache comme celle de Rust `BTreeMap`, qui stocke plusieurs entrées dans chaque nœud de l'arborescence.

Voici un autre exemple du type de bogue que ces règles attrapent. Considérez le code C++ suivant, destiné à gérer un descripteur de fichier. Pour simplifier les choses, nous allons seulement montrer un constructeur et un opérateur d'affectation de copie, et nous allons omettre la gestion des erreurs :

```
struct File {
    int descriptor;

    File(int d) : descriptor(d) { }

    File& operator=(const File &rhs) {
        close(descriptor);
        descriptor = dup(rhs.descriptor);
        return *this;
    }
};
```

L'opérateur d'affectation est assez simple, mais échoue mal dans une situation comme celle-ci :

```
File f(open("foo.txt", ...));
...
f = f;
```

Si nous attribuons à `File` à lui-même, les deux `rhs` et `*this` sont le même objet, alors `operator=` ferme le descripteur de fichier auquel il est sur le point de passer `dup`. Nous détruisons la même ressource que nous étions censés copier.

Dans Rust, le code analogue serait :

```
struct File {
    descriptor:i32
}

fn new_file(d: i32) -> File {
    File { descriptor:d }
}

fn clone_from(this: &mut File, rhs:&File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

(Ce n'est pas Rust idiomatique. Il existe d'excellents moyens de donner aux types Rust leurs propres fonctions et méthodes de constructeur, que nous décrivons au [chapitre 9](#), mais les définitions précédentes fonctionnent pour cet exemple.)

Si on écrit le code Rust correspondant à l'utilisation de `File`, on obtient :

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

Rust, bien sûr, refuse même de compiler ce code :

```
error: cannot borrow `f` as immutable because it is also
      borrowed as mutable
      |
18 |     clone_from(&mut f, &f);
      |             ^-- mutable borrow ends here
      |             |
      |             |     immutable borrow occurs here
      |             |
      |             |     mutable borrow occurs here
```

Cela devrait vous sembler familier. Il s'avère que deux bogues C++ classiques - l'incapacité à gérer l'auto-assignation et l'utilisation d'itérateurs invalidés - sont le même type de bogue sous-jacent ! Dans les deux cas, le code suppose qu'il modifie une valeur tout en en consultant une autre, alors qu'en fait, il s'agit de la même valeur. Si vous avez déjà accidentellement laissé la source et la destination d'un appel vers `memcpy` ou `strcpy` se chevaucher en C ou C++, c'est encore une autre forme que le

bogue peut prendre. En exigeant que l'accès mutable soit exclusif, Rust a repoussé une large classe d'erreurs quotidiennes.

L'immiscibilité des références partagées et mutables démontre vraiment sa valeur lors de l'écriture de code concurrent. Une course aux données n'est possible que lorsqu'une valeur est à la fois modifiable et partagée entre les threads, ce qui est exactement ce que les règles de référence de Rust éliminent. Un programme Rust concurrent qui évite `unsafe` le code est exempt de courses de données *par construction*. Nous aborderons cet aspect plus en détail lorsque nous parlerons de la concurrence au [chapitre 19](#), mais en résumé, la concurrence est beaucoup plus facile à utiliser dans Rust que dans la plupart des autres langages..

Au premier contrôle `const`, les références partagées de Rust semblent ressembler étroitement aux pointeurs vers les valeurs de C et C++ . Cependant, les règles de Rust pour les références partagées sont beaucoup plus strictes. Par exemple, considérons le code C suivant :

```
int x = 42;           // int variable, not const
const int *p = &x;    // pointer to const int
assert(*p == 42);
x++;                // change variable directly
assert(*p == 43);   // "constant" referent's value has changed
```

Le fait que `p` soit un `const int *` signifie que vous ne pouvez pas modifier son référent via `p` lui-même : `(*p)++` est interdit. Mais vous pouvez également accéder directement au référent en tant que `x`, qui n'est pas `const`, et modifier sa valeur de cette façon. Le mot-clé de la famille C `const` a ses utilisations, mais il ne l'est pas.

En Rust, une référence partagée interdit toute modification de son référent, jusqu'à la fin de sa durée de vie :

```
let mut x = 42;        // non-const i32 variable
let p = &x;            // shared reference to i32
assert_eq!(*p, 42);
x += 1;               // error: cannot assign to x because it is borrowed
assert_eq!(*p, 42);   // if you take out the assignment, this is true
```

Pour garantir qu'une valeur est constante, nous devons garder une trace de tous les chemins possibles vers cette valeur et nous assurer qu'ils ne permettent pas de modification ou qu'ils ne peuvent pas être utilisés du tout. Les pointeurs C et C++ sont trop illimités pour que le compilateur puisse vérifier cela. Les références de Rust sont toujours liées à une durée de vie particulière, ce qui permet de les vérifier au moment de la compilation.

Prendre les armes contre une mer d'objets

Depuis l'essor de la gestion automatique de la mémoire dans les années 1990, l'architecture par défaut de tous les programmes est la *mer d'objets*,

illustré à la [Figure 5-10](#) .

C'est ce qui arrive si vous avez un ramasse-miettes et que vous commencez à écrire un programme sans rien concevoir. Nous avons tous construit des systèmes qui ressemblent à ceci.

Cette architecture présente de nombreux avantages qui n'apparaissent pas dans le schéma : les premiers progrès sont rapides, il est facile de pirater des éléments et, quelques années plus tard, vous n'aurez aucune difficulté à justifier une réécriture complète. (Cue "Highway to Hell" d'AC/DC.)

Bien sûr, il y a aussi des inconvénients. Quand tout dépend de tout le reste comme ça, il est difficile de tester, d'évoluer ou même de penser à un composant isolément.

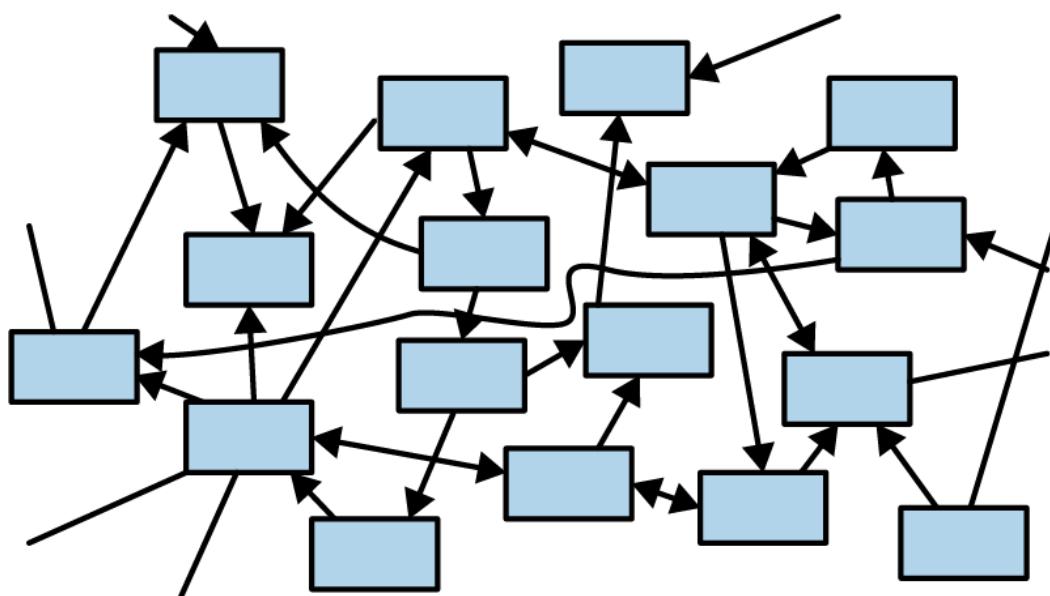


Figure 5-10. Une mer d'objets

Une chose fascinante à propos de Rust est que le modèle de propriété met un ralentisseur sur l'autoroute de l'enfer. Il faut un peu d'effort pour créer un cycle dans Rust - deux valeurs telles que chacune contient une référence pointant vers l'autre. Vous devez utiliser un type de pointeur intelligent, tel que `Rc`, et [la mutabilité intérieure](#), un sujet que nous n'avons même pas encore abordé. Rust préfère que les pointeurs, la propriété et le flux de données traversent le système dans une seule direction, comme illustré à la [Figure 5-11](#) .

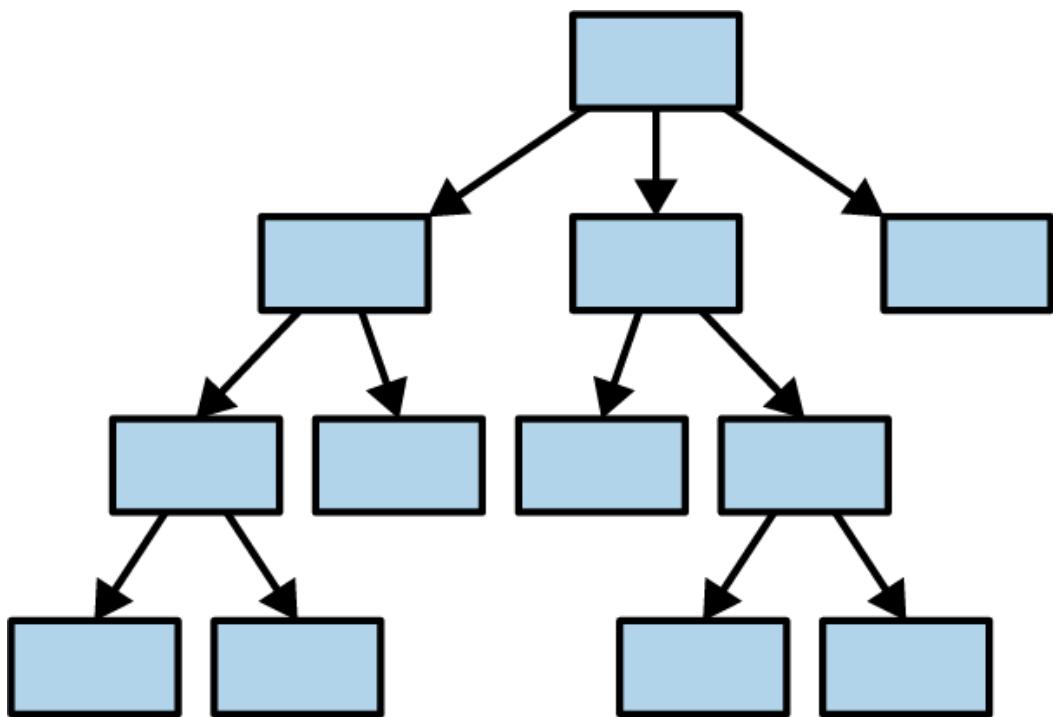


Figure 5-11. Un arbre de valeurs

La raison pour laquelle nous évoquons cela maintenant est qu'il serait naturel, après avoir lu ce chapitre, de vouloir se précipiter et de créer une "mer de structures", toutes liées avec `Rc` des pointeurs intelligents, et de recréer tous les objets- les anti-modèles orientés que vous connaissez. Cela ne fonctionnera pas pour vous tout de suite. Le modèle de propriété de Rust vous causera quelques problèmes. Le remède est de faire une conception initiale et de créer un meilleur programme.

Rust consiste à transférer la difficulté de comprendre votre programme du futur au présent. Cela fonctionne déraisonnablement bien : non seulement Rust peut vous forcer à comprendre pourquoi votre programme est thread-safe, mais il peut même nécessiter une certaine quantité d'architecture de haut niveau.motif.

Chapitre 6. Expressions

Les programmeurs LISP connaissent la valeur de tout, mais le coût de rien.

—Alan Perlis, épigramme #55

Dans ce chapitre, nous couvrirons les *expressions* de Rust, les blocs de construction qui composent le corps des fonctions Rust et donc la majorité du code Rust. La plupart des choses dans Rust sont des expressions. Dans ce chapitre, nous allons explorer la puissance que cela apporte et comment travailler avec ses limites. Nous couvrirons le flux de contrôle, qui dans Rust est entièrement orienté expression, et comment les opérateurs fondamentaux de Rust fonctionnent de manière isolée et combinée.

Quelques concepts qui entrent techniquement dans cette catégorie, tels que les fermetures et les itérateurs, sont suffisamment profonds pour que nous leur consacrons un chapitre entier plus tard. Pour l'instant, notre objectif est de couvrir autant de syntaxe que possible en quelques pages.

Un langage d'expression

Rouillerressemble visuellement à la famille des langages C, mais c'est un peu une ruse. En C, il y a une nette distinction entre *les expressions*, des morceaux de code qui ressemblent à ceci :

```
5 * (fahr-32) / 9
```

et *des déclarations*, qui ressemblent plus à ceci :

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

Expressionsont des valeurs. Les déclarations ne le font pas.

Rust est ce qu'on appelle un *langage d'expression*. Cela signifie qu'il suit une tradition plus ancienne, remontant à Lisp, où les expressions font tout le travail.

En C, `if` et `switch` sont des instructions. Ils ne produisent pas de valeur et ne peuvent pas être utilisés au milieu d'une expression. Dans Rust,

`if` et `match` peut produire des valeurs. Nous avons déjà vu une `match` expression qui produit une valeur numérique au [chapitre 2](#) :

```
pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im:point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

Une `if` expression peut être utilisée pour initialiser une variable :

```
let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus:: Ok
    } else {
        HttpStatus::ServerError // server melted
    };
```

Une `match` expression peut être passée en argument à une fonction ou une macro :

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest"
});
```

Cela explique pourquoi Rust n'a pas l'opérateur ternaire de C(). En C, il s'agit d'un analogue pratique au niveau de l'expression de l'instruction. Ce serait redondant en Rust : l'expression gère les deux cas. `expr1 ? expr2 : expr3` if if

La plupart des outils de flux de contrôle en C sont des instructions. Dans Rust, ce sont toutes des expressions.

Priorité et associativité

[Le tableau 6-1](#) résume la syntaxe des expressions Rust. Nous aborderons tous ces types d'expressions dans ce chapitre. Les opérateurs sont classés par ordre de priorité, du plus élevé au plus bas. (Comme la plupart des langages de programmation, Rust a la *priorité des opérateurs* pour déterminer l'ordre des opérations lorsqu'une expression contient plusieurs opérateurs adjacents. Par exemple, dans `limit < 2 * broom.size + 1`, l'opérateur a la priorité la plus élevée, donc l'accès au champ a lieu en premier.)

Tableau 6-1. Expressions

Type d'expression	Exemple	Traits associés
Littéral de tableau	[1, 2, 3]	
Répéter le littéral du tableau	[0; 50]	
Tuple	(6, "crullers")	
Regroupement	(2 + 2)	
Bloquer	{ f(); g() }	
Expressions de flux de contrôle	<pre>if ok { f() } if ok { 1 } else { 0 }</pre>	
	<pre>if let Some(x) = f() { x } else { 0 }</pre>	
	<pre>match x { None => 0, => 1 }</pre>	
	<pre>for v in e { f(v); }</pre>	<u>std::iter::IntoIterator</u>
	<pre>while ok { ok = f(); }</pre>	
	<pre>while let Some(x) = i t.next() { f(x); }</pre>	
	<pre>loop { next_event(); }</pre>	
	break	
	continue	
	return 0	

Type d'expression	Exemple	Traits associés
Appel de macro	<code>println! ("ok")</code>	
Chemin	<code>std::f64::consts::PI</code>	
Littéral de structure	<code>Point {x: 0, y: 0}</code>	
Accès au champ Tuple	<code>pair.0</code>	<u>Deref</u> , <u>DerefM</u> <u>ut</u>
Accès au champ de structure	<code>point.x</code>	<u>Deref</u> , <u>DerefM</u> <u>ut</u>
Appel de méthode	<code>point.translate(50, 50)</code>	<u>Deref</u> , <u>DerefM</u> <u>ut</u>
Appel de fonction	<code>stdin()</code>	<u>Fn(Arg0, ... -> T,</u> <u>FnMut(Arg0, ... -> T,</u> <u>FnOnce(Arg0, ... -> T</u>
Indice	<code>arr[0]</code>	<u>Index</u> , , <u>Index</u> <u>Mut</u> <u>Deref DerefMu</u> <u>t</u>
Vérification des erreurs	<code>create_dir("tmp")?</code>	
NON logique/au niveau du bit	<code>!ok</code>	<u>Not</u>
Négation	<code>-num</code>	<u>Neg</u>
Déréférence	<code>*ptr</code>	<u>Deref</u> , <u>DerefM</u> <u>ut</u>
Emprunter	<code>&val</code>	
Fonte de type	<code>x as u32</code>	

Type d'expression	Exemple	Traits associés
Multiplication	n * 2	Mul
Division	n / 2	Div
Reste (module)	n % 2	Rem
Ajout	n + 1	Add
Soustraction	n - 1	Sub
Décalage à gauche	n << 1	Shl
Décalage à droite	n >> 1	Shr
ET au niveau du bit	n & 1	BitAnd
OU exclusif au niveau du bit	n ^ 1	BitXor
OU au niveau du bit	n 1	BitOr
Moins que	n < 1	std::cmp::PartialOrd
Inférieur ou égal	n <= 1	std::cmp::PartialOrd
Plus grand que	n > 1	std::cmp::PartialOrd
Meilleur que ou égal	n >= 1	std::cmp::PartialOrd
Égal	n == 1	std::cmp::PartialEq
Inégal	n != 1	std::cmp::PartialEq

Type d'expression	Exemple	Traits associés
ET logique	<code>x.ok && y.ok</code>	
OU logique	<code>x.ok backup.ok</code>	
Fin de gamme exclusive	<code>start .. stop</code>	
Fin de gamme inclusive	<code>start ..= stop</code>	
Mission	<code>x = val</code>	
Affectation composée	<code>x *= 1</code>	<u>MulAssign</u>
	<code>x /= 1</code>	<u>DivAssign</u>
	<code>x %= 1</code>	<u>RemAssign</u>
	<code>x += 1</code>	<u>AddAssign</u>
	<code>x -= 1</code>	<u>SubAssign</u>
	<code>x <= 1</code>	<u>ShlAssign</u>
	<code>x >= 1</code>	<u>ShrAssign</u>
	<code>x &= 1</code>	<u>BitAndAssign</u>
	<code>x ^= 1</code>	<u>BitXorAssign</u>
	<code>x = 1</code>	<u>BitOrAssign</u>
Fermeture	<code> x, y x + y</code>	

Tous les opérateurs qui peuvent utilement être chaînés sont associatifs à gauche. Autrement dit, une chaîne d'opérations telle que `a - b - c` est regroupée en tant que `(a - b) - c`, et non `a - (b - c)`. Les opérateurs qui peuvent être chaînés de cette manière sont tous ceux auxquels vous pourriez vous attendre :

`* / % + - << >> & ^ | && ||` comme

Les opérateurs de comparaison, les opérateurs d'affectation et les opérateurs de plage .. et ..= ne peuvent pas du tout être chaînés.

Blocs et points-virgules

Blocssont le type d'expression le plus général. Un bloc produit une valeur et peut être utilisé partout où une valeur est nécessaire :

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

Le code après `Some(author) =>` est l'expression simple `author.name()`. Le code après `None =>` est une expression de bloc. Cela ne fait aucune différence pour Rust. La valeur du bloc est la valeur de sa dernière expression, `ip.to_string()`.

Notez qu'il n'y a pas de point-virguleaprès l'`ip.to_string()` appel de la méthode. La plupart des lignes de code Rust se terminent par un point-virgule ou des accolades, tout comme C ou Java. Et si un bloc ressemble à du code C, avec des points-virgules à tous les endroits familiers, il fonctionnera comme un bloc C et sa valeur sera `()`. Comme nous l'avons mentionné au [chapitre 2](#), lorsque vous laissez le point-virgule hors de la dernière ligne d'un bloc, cela fait de la valeur du bloc la valeur de son expression finale, plutôt que l'habituel `()`.

Dans certains langages, en particulier JavaScript, vous êtes autorisé à omettre les points-virgules, et le langage les remplit simplement pour vous, une commodité mineure. Ceci est différent. Dans Rust, le point-virgule signifie en fait quelque chose :

```
let msg = {
    // let-declaration: semicolon is always required
    let dandelion_control = puffball.open();

    // expression + semicolon: method is called, return value dropped
    dandelion_control.release_all_seeds(launch_codes);

    // expression with no semicolon: method is called,
    // return value stored in `msg`
    dandelion_control.get_status()
};
```

Cette capacité des blocs à contenir des déclarations et à produire également une valeur à la fin est une fonctionnalité intéressante, qui devient rapidement naturelle. Le seul inconvénient est que cela conduit à un message d'erreur étrange lorsque vous omettez un point-virgule par accident :

```
...
if preferences.changed() {
    page.compute_size() // oops, missing semicolon
}
...
```

Si vous faites cette erreur dans un programme C ou Java, le compilateur vous signalera simplement qu'il vous manque un point-virgule. Voici ce que dit Rust :

```
error: mismatched types
22 |         page.compute_size() // oops, missing semicolon
|         ^^^^^^^^^^^^^^^^^^- help: try adding a semicolon: `;`  
|         |
|         expected (), found tuple
|
= note: expected unit type `()`  
       found tuple `(u32, u32)`
```

Sans le point-virgule, la valeur du bloc serait ce qui `page.compute_size()` revient, mais un `if` sans `else` doit toujours revenir `()`. Heureusement, Rust a déjà vu ce genre de chose et suggère d'ajouter le point-virgule.

Déclarations

En plus des expressions et des points-virgules, un bloc peut contenir n'importe quel nombre de déclarations. Les plus courantes sont les `let` déclarations, qui déclarent des variables locales :

```
let name : type = expr ;
```

Le type et l'initialiseur sont facultatifs. Le point-virgule est obligatoire. Comme tous les identifiants dans Rust, les noms de variables doivent commencer par une lettre ou un trait de soulignement, et ne peuvent contenir des chiffres qu'après ce premier caractère. Rust a une définition large de «lettre»: elle comprend les lettres grecques, les caractères latins accentués et bien d'autres symboles, tout ce que l'annexe standard Unicode # 31 déclare appropriée. Les emoji ne sont pas autorisés.

Une `let` déclaration peut déclarer une variable sans l'initialiser. La variable peut alors être initialisée avec une affectation ultérieure. Ceci est parfois utile, car parfois une variable doit être initialisée à partir du milieu d'une sorte de construction de flux de contrôle :

```
let name;
if user.hasNickname() {
    name = user.nickname();
} else {
    name = generateUniqueName();
    user.register(&name);
}
```

Ici, il existe deux manières différentes d' initialiser la variable locale, mais dans les deux cas, elle sera initialisée exactement une fois, et `name` n'a donc pas besoin d'être déclarée `mut`.

C'est une erreur d'utiliser une variable avant qu'elle ne soit initialisée. (Ceci est étroitement lié à l'erreur d'utilisation d'une valeur après qu'elle ait été déplacée. Rust veut vraiment que vous n'utilisiez les valeurs que tant qu'elles existent !)

Vous pouvez occasionnellement voir du code qui semble redéclarer une variable existante, comme ceci :

```
for line in file.lines() {
    let line = line?;
    ...
}
```

La `let` déclaration crée une nouvelle, deuxième variable, d'un type différent. Le type de la première variable `line` est `Result<String, io::Error>`. Le second `line` est un `String`. Sa définition remplace la première pour le reste du bloc. C'est ce qu'on appelle l'*ombrage* et est très courant dans les programmes Rust. Le code est équivalent à :

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

Dans ce livre, nous nous en tiendrons à l'utilisation d'un `_result` suffixe dans de telles situations afin que les variables aient des noms distincts.

Un bloc peut également contenir *des déclarations d'éléments*. Un élément est simplement n'importe quelle déclaration qui pourrait apparaître glo-

galement dans un programme ou un module, comme un `fn`, `struct` ou `use`.

Les chapitres suivants couvriront les éléments en détail. Pour l'instant, `fn` fait un exemple suffisant. Tout bloc peut contenir un `fn`:

```
use std::io;
use std::cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp.cmp(&b.timestamp) // first, compare timestamps
            .reverse() // newest file first
            .then(a.path.cmp(&b.path)) // compare paths to break ties
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

Lorsqu'un `fn` est déclaré à l'intérieur d'un bloc, sa portée est le bloc entier, c'est-à-dire qu'il peut être utilisé dans tout le bloc englobant. Mais un imbriqué `fn` ne peut pas accéder aux variables locales ou aux arguments qui se trouvent dans la portée. Par exemple, la fonction

`cmp_by_timestamp_then_name` ne pouvait pas utiliser `v` directement. (Rust a également des fermetures, qui voient dans les portées englobantes. Voir le [chapitre 14](#).)

Un bloc peut même contenir un module entier. Cela peut sembler un peu long - avons-nous vraiment besoin de pouvoir imbriquer *chaque* élément du langage dans chaque autre élément ? - mais les programmeurs (et en particulier les programmeurs utilisant des macros) ont un moyen de trouver des utilisations pour chaque morceau d'orthogonalité fourni par le langage..

si et correspondre

La forme d'une `if` expression est familier :

```
si condition1 {
    bloc1
} sinon si condition2 {
    bloc2
} sinon {
```

```
bloc_n  
}
```

Chacun `condition` doit être une expression de type `bool` ; fidèle à la forme, Rust ne convertit pas implicitement les nombres ou les pointeurs en valeurs booléennes.

Contrairement à C, les parenthèses ne sont pas nécessaires autour des conditions. En fait, `rustc` émettra un avertissement si des parenthèses inutiles sont présentes. Les accolades sont cependant obligatoires.

Les `else if` blocs, ainsi que le final `else`, sont facultatifs. Une `if` expression sans bloc se comporte exactement comme si elle avait un bloc `else vide . else`

`match` expressionssont quelque chose comme l' `switch` instruction C, mais plus flexible. Un exemple simple :

```
match code {  
    0 => println!("OK"),  
    1 => println!("Wires Tangled"),  
    2 => println!("User Asleep"),  
    _ => println!("Unrecognized Error {}", code)  
}
```

C'est quelque chose qu'une `switch` déclaration pourrait faire. Exactement l'un des quatre bras de cette `match` expression s'exécutera, en fonction de la valeur de `code`. Le modèle générique `_` correspond à tout. C'est comme le `default`: cas dans une `switch` instruction, sauf qu'elle doit venir en dernier ; placer un `_` motif avant d'autres motifs signifie qu'il aura priorité sur eux. Ces modèles ne correspondront jamais à rien (et le compilateur vous en avertira).

Le compilateur peut optimiser ce type d' `match` utilisation d'une table de saut, tout comme une `switch` instruction en C++. Une optimisation similaire est appliquée lorsque chaque bras de a `match` produit une valeur constante. Dans ce cas, le compilateur construit un tableau de ces valeurs, et le `match` est compilé dans un accès au tableau. Hormis une vérification des limites, il n'y a pas de branchement du tout dans le code compilé.

La polyvalence de `match` découle de la variété des *modèles pris en charge* qui peut être utilisé à gauche de `=>` dans chaque bras. Ci-dessus, chaque motif est simplement un entier constant. Nous avons également montré `match` des expressions qui distinguent les deux types de `Option` valeur :

```

match params.get("name") {
    Some(name) => println!("Hello, {}!", name),
    None => println!("Greetings, stranger.")
}

```

Ce n'est qu'un aperçu de ce que les modèles peuvent faire. Un modèle peut correspondre à une plage de valeurs. Il peut décompresser les tuples. Il peut correspondre à des champs individuels de structures. Il peut rechercher des références, emprunter des parties d'une valeur, etc. Les motifs de Rust sont un mini-langage à part entière. Nous leur consacrerons plusieurs pages dans le [chapitre 10](#).

La forme générale d'une `match` expression est :

```

valeur de correspondance {
    motif => expr ,
    ...
}

```

La virgule après un bras peut être supprimée si le `expr` est un bloc.

Rust vérifie les données `value` par rapport à chaque motif à tour de rôle, en commençant par le premier. Lorsqu'un modèle correspond, le correspondant `expr` est évalué et l' `match` expression est complète ; aucun autre motif n'est vérifié. Au moins un des motifs doit correspondre. Rust interdit les `match` expressions qui ne couvrent pas toutes les valeurs possibles :

```

let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
}; // error: nonexhaustive patterns

```

Tous les blocs d'une `if` expression doivent produire des valeurs du même type :

```

let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena }; // ok

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 }; // error

let best_sports_team =
    if is_hockey_season() { "Predators" }; // error

```

(Le dernier exemple est une erreur car en juillet, le résultat serait `()`.)

De même, tous les bras d'une `match` expression doivent avoir le même type :

```
let suggested_pet =  
    match favorites.element {  
        Fire => Pet:: RedPanda,  
        Air => Pet:: Buffalo,  
        Water => Pet::Orca,  
        _ => None // error: incompatible types  
    };
```

si laissé

Là est une `if` forme de plus, l'`if let` expression :

```
if let pattern = expr {  
    block1  
} else {  
    block2  
}
```

Le donné `expr` correspond soit `pattern` à , auquel cas `block1` s'exécute, soit ne correspond pas, et `block2` s'exécute. Parfois, c'est un bon moyen d'extraire des données d'un `Option` ou `Result` :

```
if let Some(cookie) = request.session_cookie {  
    return restore_session(cookie);  
}  
  
if let Err(err) = show_cheesy_anti_robot_task() {  
    log_robot_attempt(err);  
    politely_accuse_user_of_being_a_robot();  
} else {  
    session.mark_as_human();  
}
```

Il n'est jamais strictement nécessaire d'utiliser `if let`, car `match` peut tout `if let` faire. Une `if let` expression est un raccourci pour un `match` avec un seul motif:

```
match expr {  
    motif => { bloc1 }  
    _ => { bloc2 }  
}
```

Boucles

Il y a quatre boucles expressions:

```
tant que condition {
    bloc
}

while let pattern = expr {
    bloc
}

boucle {
    bloc
}

pour motif dans iterable {
    bloc
}
```

Les boucles sont des expressions dans Rust, mais la valeur d'une boucle `while` ou `for` est toujours `()`, donc leur valeur n'est pas très utile. Une `loop` expression peut produire une valeur si vous en spécifiez une.

Une `while` boucle se comporte exactement comme l'équivalent C, sauf que, encore une fois, le `condition` doit être du type exact `bool`.

La `while let` boucle est analogue à `if let`. Au début de chaque itération de boucle, la valeur de `expr` correspond à la valeur donnée `pattern`, auquel cas le bloc s'exécute, ou non, auquel cas la boucle se termine.

Utiliser `loop` pour écrire des boucles infinies. Il exécute le `block` à plusieurs reprises pour toujours (ou jusqu'à ce qu'un `break` or `return` soit atteint ou que le thread panique).

Une `for` boucle évalue l' `iterable` expression, puis évalue `block` une fois pour chaque valeur dans l'itérateur résultant. De nombreux types peuvent être itérés, y compris toutes les collections standard telles que `Vec` et `HashMap`. `for` La boucle C standard :

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

s'écrit ainsi en Rust :

```
for i in 0..20 {
    println!("{}" , i);
```

```
}
```

Comme en C, le dernier nombre imprimé est 19.

L'`..` opérateur produit une *gamme*, une structure simple avec deux champs : `start` et `end`. 0..20 est le même que `std::ops::Range { start: 0, end: 20 }`. Les plages peuvent être utilisées avec des `for` boucles car `Range` il s'agit d'un type itérable : il implémente le `std::iter::IntoIterator` trait, dont nous parlerons au [chapitre 15](#). Les collections standard sont toutes itérables, tout comme les tableaux et les tranches.

Conformément à la sémantique de déplacement de Rust, une `for` boucle sur une valeur consomme la valeur :

```
let strings:Vec<String> = error_messages();
for s in strings {                                // each String is moved into s here...
    println!("{}", s);
}                                              // ...and dropped here
println!("{} error(s)", strings.len()); // error: use of moved value
```

Cela peut être gênant. Le remède simple consiste à boucler sur une référence à la collection à la place. La variable de boucle sera alors une référence à chaque élément de la collection :

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

Ici, le type de `&strings` est `&Vec<String>`, et le type de `rs` est `&String`.

L'itération sur une `mut` référence fournit une `mut` référence à chaque élément :

```
for rs in &mut strings { // the type of rs is &mut String
    rs.push('\n'); // add a newline to each string
}
```

[Le chapitre 15](#) couvre les `for` boucles plus en détail et montre de nombreuses autres façons d'utiliser les itérateurs.

Flux de contrôle dans les boucles

Une `break` expression sort d'une boucle englobante. (Dans Rust, `break` ne fonctionne que dans les boucles. Ce n'est pas nécessaire dans les `match` expressions, qui sont différentes des `switch` déclarations à cet égard.)

Dans le corps d'un `loop`, vous pouvez donner `break` une expression, dont la valeur devient celle de la boucle :

```
// Each call to `next_line` returns either `Some(line)`, where
// `line` is a line of input, or `None`, if we've reached the end of
// the input. Return the first line that starts with "answer: ".
// Otherwise, return "answer: nothing".
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

Naturellement, toutes les `break` expressions de la `loop` doivent produire des valeurs de même type, qui devient le type de la `loop` elle-même.

Une `continue` expression saute à l'itération de boucle suivante :

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

En `for` boucle, `continue` avance à la valeur suivante de la collection. S'il n'y a plus de valeurs, la boucle se termine. De la même manière, dans une `while` boucle, `continue` vérifie la condition de la boucle. Si c'est maintenant faux, la boucle se termine.

Une boucle peut être étiquetée avec une durée de vie. Dans l'exemple suivant, '`search:`' est une étiquette pour la `for` boucle externe. Ainsi, `break 'search'` quitte cette boucle, pas la boucle interne :

```
'search:
for room in apartment {
```

```

        for spot in room.hiding_spots() {
            if spot.contains(keys) {
                println!("Your keys are {} in the {}.", spot, room);
                break 'search;
            }
        }
    }
}

```

A `break` peut avoir à la fois une étiquette et une expression de valeur:

```

// Find the square root of the first perfect square
// in the series.
let sqrt = 'outer:loop {
    let n = next_number();
    for i in 1.. {
        let square = i * i;
        if square == n {
            // Found a square root.
            break 'outer i;
        }
        if square > n {
            // `n` isn't a perfect square, try the next
            break;
        }
    }
};

```

Les étiquettes peuvent également être utilisées avec `continue`.

expression de retour

Une `return` expression quitte la fonction en cours, renvoyant une valeur à l'appelant.

`return` sans valeur est un raccourci pour `return ()`:

```

fn f() {      // return type omitted: defaults to ()
    return;   // return value omitted: defaults to ()
}

```

Les fonctions n'ont pas besoin d'avoir une `return` expression explicite. Le corps d'une fonction fonctionne comme une expression de bloc : si la dernière expression n'est pas suivie d'un point-virgule, sa valeur est la valeur de retour de la fonction. En fait, c'est le moyen préféré de fournir la valeur de retour d'une fonction dans Rust.

Mais cela ne signifie pas que `return` c'est inutile, ou simplement une concession aux utilisateurs qui ne sont pas expérimentés avec les lan-

gages d'expression. Comme une `break` expression, `return` peut abandonner un travail en cours. Par exemple, au [chapitre 2](#), nous avons utilisé l' `? opérateur` pour vérifier les erreurs après avoir appelé une fonction qui peut échouer :

```
let output = File::create(filename)?;
```

Nous avons expliqué qu'il s'agit d'un raccourci pour une `match` expression :

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

Ce code commence par appeler `File::create(filename)`. Si cela revient `Ok(f)`, alors l' `match` expression entière est évaluée à `f`, donc `f` est stockée dans `output`, et nous continuons avec la ligne de code suivante après le `match`.

Sinon, nous allons faire correspondre `Err(err)` et frapper l' `return` expression. Lorsque cela se produit, peu importe que nous soyons en train d'évaluer une `match` expression pour déterminer la valeur de la variable `output`. Nous abandonnons tout cela et quittons la fonction englobante, renvoyant l'erreur que nous avons obtenue de `File::create()`.

Nous couvrirons l' `? opérateur` plus complètement dans "[Propagation des erreurs](#)".

Pourquoi Rust a une boucle

Plusieurs éléments du compilateur Rust analysent le flux de contrôle dans votre programme :

- Rust vérifie que chaque chemin à travers une fonction renvoie une valeur du type de retour attendu. Pour le faire correctement, il doit savoir s'il est possible d'atteindre la fin de la fonction.
- Rust vérifie que les variables locales ne sont jamais utilisées non initialisées. Cela implique de vérifier chaque chemin à travers une fonction pour s'assurer qu'il n'y a aucun moyen d'atteindre un endroit où une variable est utilisée sans avoir déjà traversé le code qui l'initialise.
- Rust met en garde contre un code inaccessible. Le code est inaccessible si *aucun* chemin à travers la fonction ne l'atteint.

Ceux-ci sont dits *sensibles au flux* analyses. Ils n'ont rien de nouveau; Java a eu une analyse «d'affectation définie», similaire à celle de Rust, pendant des années.

Lors de l'application de ce type de règle, un langage doit trouver un équilibre entre la simplicité, qui permet aux programmeurs de comprendre plus facilement de quoi le compilateur parle parfois, et l'intelligence, qui peut aider à éliminer les faux avertissements et les cas où le compilateur rejette un programme sécuritaire. Rust a opté pour la simplicité. Ses analyses sensibles au flux n'examinent pas du tout les conditions de boucle, mais supposent simplement que n'importe quelle condition dans un programme peut être vraie ou fausse.

Cela amène Rust à rejeter certains programmes sûrs :

```
fn wait_for_process(process: &mut Process) ->i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
} // error: mismatched types: expected i32, found ()
```

L'erreur ici est fausse. Cette fonction ne sort que via l'`return` instruction, donc le fait que la `while` boucle ne produise pas un `i32` n'est pas pertinent.

L'`loop` expression est proposée comme une solution "dire ce que vous voulez dire" à ce problème.

Le système de type de Rust est également affecté par le flux de contrôle. Nous avons dit précédemment que toutes les branches d'une `if` expression doivent avoir le même type. Mais il serait idiot d'appliquer cette règle aux blocs qui se terminent par une expression `break` ou , un infini ou un appel à ou . Ce que toutes ces expressions ont en commun, c'est qu'elles ne se terminent jamais de la manière habituelle, en produisant une valeur. A ou sort brusquement du bloc courant, un infini ne finit jamais du tout, et ainsi de suite. `return loop panic!`

```
() std::process::exit() break return loop
```

Ainsi, dans Rust, ces expressions n'ont pas de type normal. Les expressions qui ne se terminent pas normalement se voient attribuer le type spécial `!` et sont exemptées des règles concernant les types devant correspondre. Vous pouvez voir `!` dans la signature de fonction de `std::process::exit()`:

```
fn exit(code: i32) ->!
```

Le ! moyen qui `exit()` ne revient jamais. C'est une *fonction divergente*.

Vous pouvez écrire vos propres fonctions divergentes en utilisant la même syntaxe, et c'est parfaitement naturel dans certains cas :

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) ->! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

Bien sûr, Rust considère alors qu'il s'agit d'une erreur si la fonction peut revenir normalement.

Avec ces blocs de construction de flux de contrôle à grande échelle en place, nous pouvons passer aux expressions plus fines généralement utilisées dans ce flux, comme les appels de fonction et les opérateurs arithmétiques.

Appels de fonction et de méthode

La syntaxe pour appeler des fonctions et des méthodes est la même dans Rust que dans de nombreux autres langages :

```
let x = gcd(1302, 462); // function call

let room = player.location(); // method call
```

Dans le deuxième exemple ici, `player` est une variable de type composé `Player`, qui a une `.location()` méthode composée. (Nous montrerons comment définir vos propres méthodes lorsque nous commencerons à parler des types définis par l'utilisateur au [chapitre 9](#).)

Rust fait généralement une distinction nette entre les références et les valeurs auxquelles elles se réfèrent. Si vous passez `a &i32` à une fonction qui attend un `i32`, c'est une erreur de type. Vous remarquerez que l'`.` opérateur assouplit un peu ces règles. Dans l'appel de méthode `player.location()`, `player` il peut s'agir de `a Player`, d'une référence de type `&Player` ou d'un pointeur intelligent de type `Box<Player>` or `Rc<Player>`. La `.location()` méthode peut prendre le joueur soit par valeur, soit par référence. La même `.location()` syntaxe fonctionne dans tous les cas, car l'`.` opérateur de Rust déréférence automatiquement `player` ou lui emprunte une référence selon les besoins.

Une troisième syntaxe est utilisée pour appeler les fonctions associées au type, comme `Vec::new()` :

```
let mut numbers = Vec::new(); // type-associated function call
```

Celles-ci sont similaires aux méthodes statiques dans les langages orientés objet : les méthodes ordinaires sont appelées sur des valeurs (comme `my_vec.len()`) et les fonctions associées au type sont appelées sur des types (comme `Vec::new()`).

Naturellement, les appels de méthode peuvent être chaînés :

```
// From the Actix-based web server in Chapter 2:  
server  
    .bind("127.0.0.1:3000").expect("error binding server to address")  
    .run().expect("error running server");
```

Une bizarrerie de la syntaxe Rust est que dans un appel de fonction ou un appel de méthode, la syntaxe habituelle pour les types génériques, `Vec<T>`, ne fonctionne pas :

```
return Vec<i32>::with_capacity(1000); // error: something about chained compa  
  
let ramp = (0 .. n).collect<Vec<i32>>(); // same error
```

Le problème est que dans les expressions, `<` est l'opérateur inférieur à. Le compilateur Rust suggère utilement d'écrire à la `::<T>` place de `<T>` dans ce cas, et cela résout le problème :

```
return Vec:: <i32>::with_capacity(1000); // ok, using ::<  
  
let ramp = (0 .. n).collect::<Vec<i32>>(); // ok, using ::<
```

Le symbole `::<...>` est affectueusement connu dans la communauté de Rust sous le nom de *turbofish*.

Alternativement, il est souvent possible de supprimer les paramètres de type et de laisser Rust les déduire :

```
return Vec::with_capacity(10); // ok, if the fn return type is Vec<i32>  
  
let ramp:Vec<i32> = (0 .. n).collect(); // ok, variable's type is given
```

Il est considéré comme bon style d'omettre les types chaque fois qu'ils peuvent être déduits.

Champs et éléments

Les champs d'une structure sont accessibles à l'aide d'une syntaxe familière. Les tuples sont les mêmes sauf que leurs champs ont des noms plutôt que des noms :

```
game.black_pawns    // struct field
coords.1            // tuple element
```

Si la valeur à gauche du point est une référence ou un type de pointeur intelligent, elle est automatiquement déréférencée, comme pour les appels de méthode.

Les crochets accèdent aux éléments d'un tableau, d'une tranche ou d'un vecteur :

```
pieces[i]           // array element
```

La valeur à gauche des parenthèses est automatiquement déréférencée.

Des expressions comme ces trois sont appelées *lvalues*, car ils peuvent apparaître sur le côté gauche d'un devoir :

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Bien sûr, cela n'est autorisé que si `game`, `coords`, et `pieces` sont déclarés comme `mut` variables.

Extraire une tranche d'un tableau ou d'un vecteur est simple :

```
let second_half = &game_moves[midpoint .. end];
```

Ici, `game_moves` il peut s'agir d'un tableau, d'une tranche ou d'un vecteur ; le résultat, quoi qu'il en soit, est une tranche empruntée de longueur `end - midpoint`. `game_moves` est considéré comme emprunté pour la durée de vie de `second_half`.

L' `..` opérateur permet d'omettre l'un ou l'autre des opérandes ; il produit jusqu'à quatre types d'objets différents selon les opérandes présents :

```
..      // RangeFull
a ..    // RangeFrom { start: a }
```

```
.. b      // RangeTo { end: b }
a .. b    // Range { start: a, end: b }
```

Les deux dernières formes sont *exclusives à la fin*(ou *semi-ouvert*) : la valeur finale n'est pas comprise dans la plage représentée. Par exemple, la plage `0 .. 3` comprend les nombres `0`, `1` et `2`.

L'`..=` opérateur produit des gammes *inclusives* (ou *fermées*), qui incluent la valeur finale :

```
..= b      // RangeToInclusive { end: b }
a ..= b    // RangeInclusive::new(a, b)
```

Par exemple, la plage `0 ..= 3` comprend les nombres `0`, `1`, `2` et `3`.

Seules les plages qui incluent une valeur de départ sont itérables, car une boucle doit avoir un point de départ. Mais dans le découpage en tableaux, les six formes sont utiles. Si le début ou la fin de la plage est omis, il s'agit par défaut du début ou de la fin des données découpées.

Ainsi, une implémentation de quicksort, l'algorithme de tri classique diviser pour mieux régner, pourrait ressembler, en partie, à ceci :

```
fn quicksort<T: Ord>(slice:&mut [T]) {
    if slice.len() <= 1 {
        return; // Nothing to sort.
    }

    // Partition the slice into two parts, front and back.
    let pivot_index = partition(slice);

    // Recursively sort the front half of `slice`.
    quicksort(&mut slice[.. pivot_index]);

    // And the back half.
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

Opérateurs de référence

L'adresse des opérateurs, `&` et `&mut`, sont traités au [chapitre 5](#).

* L'opérateur `unary &` permet d'accéder à la valeur pointée par une référence. Comme nous l'avons vu, Rust suit automatiquement les références lorsque vous utilisez l'`operator .` opérateur pour accéder à un champ ou à une méthode, de sorte que l'`*` opérateur n'est nécessaire que lorsque nous voulons lire ou écrire la valeur entière vers laquelle pointe la référence.

Par exemple, parfois un itérateur produit des références, mais le programme a besoin des valeurs sous-jacentes :

```
let padovan:Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

Dans cet exemple, le type de `elem` est `&u64`, tout `*elem` comme a `u64`.

Opérateurs arithmétiques, binaires, de comparaison et logiques

Le binaire de Rust les opérateurs sont comme ceux de beaucoup d'autres langages. Pour gagner du temps, nous supposons la familiarité avec l'une de ces langues et nous nous concentrerons sur les quelques points où Rust s'écarte de la tradition.

Rust a l'arithmétique habituelle opérateurs, `+`, `-`, `*`, `/` et `%`. Comme mentionné au [chapitre 3](#), un débordement d'entier est détecté et provoque une panique dans les versions de débogage. La bibliothèque standard fournit des méthodes telles `a.wrapping_add(b)` que l'arithmétique non vérifiée.

Entier la division arrondit vers zéro et diviser un entier par zéro déclenche une panique même dans les versions de version. Les entiers ont une méthode `a.checked_div(b)` qui renvoie un `Option` (`None` si `b` est égal à zéro) et ne panique jamais.

Unaire `-` nie un nombre. Il est pris en charge pour tous les types numériques à l'exception des entiers non signés. Il n'y a pas d'opérateur unary `+`.

```
println!("{}" , -100);      // -100
println!("{}" , -100u32);   // error: can't apply unary `--` to type `u32`
println!("{}" , +100);      // error: expected expression, found `+`
```

Comme en C, `a % b` calcule le reste signé, ou module, de la division arrondie vers zéro. Le résultat a le même signe que l'opérande de gauche. Notez que cela `%` peut être utilisé sur les nombres à virgule flottante ainsi que sur les entiers :

```
let x = 1234.567 % 10.0;  // approximately 4.567
```

Rust hérite également des C au niveau du bitopérateurs entiers, `&`, `|`, `^`, `<<` et `>>`. Cependant, Rust utilise à la `!` place de `~` pour NOT au niveau du bit :

```
let hi:u8 = 0xe0;
let lo = !hi; // 0x1f
```

Cela signifie que `!n` cela ne peut pas être utilisé sur un nombre entier `n` pour signifier "n est égal à zéro". Pour cela, écrivez `n == 0`.

Le décalage de bits est toujours une extension de signe sur les types d'entiers signés et une extension de zéro sur les types d'entiers non signés. Puisque Rust a des entiers non signés, il n'a pas besoin d'un opérateur de décalage non signé, comme l'`>>>` opérateur de Java.

Les opérations au niveau du bit ont une priorité plus élevée que les comparaisons, contrairement au C, donc si vous écrivez `x & BIT != 0`, cela signifie `(x & BIT) != 0`, comme vous l'aviez probablement prévu. C'est bien plus utile que l'interprétation de C, `x & (BIT != 0)`, qui teste le mauvais bit !

La comparaison de Rust les opérateurs sont `==`, `!=`, `<`, `<=`, `>` et `>=`. Les deux valeurs comparées doivent avoir le même type.

Rust possède également les deux logiques de court-circuitopérateurs `&&` et `||`. Les deux opérandes doivent avoir le type exact `bool`.

Mission

L'`=` opérateur peut être utilisé pour affecter des `mut` variables et leurs champs ou éléments. Mais l'affectation n'est pas aussi courant dans Rust que dans d'autres langages, car les variables sont immuables par défaut.

Comme décrit au [chapitre 4](#), si la valeur a un non-`Copy` type, l'affectation la *déplace* vers la destination. La propriété de la valeur est transférée de la source à la destination. La valeur précédente de la destination, le cas échéant, est supprimée.

L'affectation composée est prise en charge :

```
total += item.price;
```

Cela équivaut à `total = total + item.price;`. D'autres opérateurs sont également pris en charge : `-=`, `*=`, etc. La liste complète est donnée dans le [Tableau 6-1](#), plus haut dans ce chapitre.

Contrairement à C, Rust ne prend pas en charge l'affectation de chaînage : vous ne pouvez pas écrire `a = b = 3` pour affecter la valeur 3 à la fois à `a` et `b`. L'affectation est suffisamment rare dans Rust pour que vous ne manquiez pas ce raccourci.

Rust n'a pas les opérateurs d'incrémentation et de décrémentation de C `++` et `--`.

Moulages de type

La conversion d'une valeur d'un type à un autre nécessite généralement un cast explicite à Rust. Les casts utilisent le mot-`as` clé :

```
let x = 17;           // x is type i32
let index = x as usize; // convert to usize
```

Plusieurs types de moulages sont autorisés :

- Les nombres peuvent être convertis de n'importe lequel des types numériques intégrés en n'importe quel autre.

La conversion d'un entier en un autre type d'entier est toujours bien définie. La conversion en un type plus étroit entraîne une troncature. Un entier signé converti en un type plus large est étendu par un signe, un entier non signé est étendu par zéro, et ainsi de suite. Bref, pas de surprise.

La conversion d'un type à virgule flottante en un type entier arrondit vers zéro : la valeur de `-1.99 as i32` est `-1`. Si la valeur est trop grande pour tenir dans le type entier, le cast produit la valeur la plus proche que le type entier peut représenter : la valeur de `1e6 as u8` est `255`.

- Les valeurs de type `bool` ou `char`, ou d'un type de `enum` type C, peuvent être converties en n'importe quel type entier. (Nous couvrirons les énumérations au [chapitre 10](#).)

La conversion dans l'autre sens n'est pas autorisée, car les types `bool`, `char` et `enum` ont tous des restrictions sur leurs valeurs qui devraient être appliquées avec des vérifications à l'exécution. Par exemple, la conversion de `a u16` en type `char` est interdite car certaines `u16` valeurs, telles que , correspondent à des points de code de substitution Unicode et ne constituerait donc pas des valeurs `0xd800` valides . `char` Il existe une méthode standard, `std::char::from_u32()`, qui effectue la vérification à l'exécution et renvoie un `Option<char>` ; mais plus précisément, le besoin de ce type de conversion est devenu rare. Nous convertissons généralement des chaînes ou des flux entiers à la fois, et les algorithmes sur le texte

Unicode sont souvent non triviaux et il vaut mieux les laisser aux bibliothèques.

Exceptionnellement, `a` `u8` peut être converti en type `char`, puisque tous les entiers de 0 à 255 sont des points de code Unicode valides pour `char` tenir.

- Certains transtypages impliquant des types de pointeurs non sécurisés sont également autorisés. Voir [« Pointeurs bruts »](#).

Nous avons dit qu'une conversion nécessite *généralement un casting*.

Quelques conversions impliquant des types de référence sont si simples que le langage les exécute même sans transtypage. Un exemple trivial est la conversion d'une `mut` référence en une non- `mut` référence.

Cependant, plusieurs conversions automatiques plus importantes peuvent se produire :

- Les valeurs de type `&string` sont automatiquement converties en type `&str` sans transtypage.
- Les valeurs de type `&Vec<i32>` se convertissent automatiquement en `&[i32]`.
- Les valeurs de type `&Box<Chessboard>` se convertissent automatiquement en `&Chessboard`.

Celles-ci sont appelées *coercitions de deref*, car ils s'appliquent aux types qui implémentent le `Deref` trait intégré. Le but de `Deref` la coercition est de faire en sorte que les types de pointeurs intelligents, comme `Box`, se comportent autant que possible comme la valeur sous-jacente. Utiliser `a Box<Chessboard>` est la plupart du temps comme utiliser un plain `Chessboard`, grâce à `Deref`.

Les types définis par l'utilisateur peuvent `Deref` également implémenter le trait. Lorsque vous devez écrire votre propre type de pointeur intelligent, consultez [« Deref et DerefMut »](#).

Fermetures

La rouille a des *fermetures*, des valeurs de type fonction légères. Une fermeture consiste généralement en une liste d'arguments, donnée entre des barres verticales, suivie d'une expression :

```
let is_even = |x| x % 2 == 0;
```

Rust déduit les types d'arguments et le type de retour. Vous pouvez également les écrire explicitement, comme vous le feriez pour une fonction. Si vous spécifiez un type de retour, le corps de la fermeture doit être un bloc, par souci de cohérence syntaxique :

```
let is_even = |x: u64| ->bool x % 2 == 0; // error  
  
let is_even = |x: u64| ->bool { x % 2 == 0 }; // ok
```

L'appel d'une fermeture utilise la même syntaxe que l'appel d'une fonction :

```
assert_eq!(is_even(14), true);
```

Les fermetures sont l'une des caractéristiques les plus délicieuses de Rust, et il y a beaucoup plus à dire à leur sujet. Nous le dirons au [chapitre 14](#).

En avant

Les expressions sont ce que nous considérons comme du "code en cours d'exécution". Ils font partie d'un programme Rust qui se compile en instructions machine. Pourtant, ils ne représentent qu'une petite fraction de l'ensemble de la langue.

Il en va de même dans la plupart des langages de programmation. La première tâche d'un programme est de s'exécuter, mais ce n'est pas sa seule tâche. Les programmes doivent communiquer. Ils doivent être testables. Ils doivent rester organisés et flexibles pour pouvoir continuer à évoluer. Ils doivent interagir avec le code et les services créés par d'autres équipes. Et même juste pour fonctionner, les programmes dans un language typé statiquement comme Rust ont besoin de plus d'outils pour organiser les données que de simples tuples et tableaux.

À venir, nous passerons plusieurs chapitres à parler des fonctionnalités dans ce domaine : les modules et les caisses, qui donnent la structure de votre programme, puis les structures et les énumérations, qui font la même chose pour vos données..

Tout d'abord, nous consacrerons quelques pages au sujet important de ce qu'il faut faire lorsque les choses tournent mal.

Chapitre 7. Gestion des erreurs

Je savais que si je restais assez longtemps, quelque chose comme ça arriverait.

—George Bernard Shaw sur la mort

L'approche de Rust pour la gestion des erreurs est suffisamment inhabituel pour justifier un court chapitre sur le sujet. Il n'y a pas d'idées difficiles ici, juste des idées qui pourraient être nouvelles pour vous. Ce chapitre couvre les deux différents types de gestion des erreurs dans Rust : panique et `Results`.

Les erreurs ordinaires sont gérées à l'aide du `Result` type. `Results` représentent généralement des problèmes causés par des éléments extérieurs au programme, comme une entrée erronée, une panne de réseau ou un problème d'autorisations. Que de telles situations se produisent ne dépend pas de nous ; même un programme sans bogue les rencontrera de temps en temps. La majeure partie de ce chapitre est consacrée à ce type d'erreur. Nous couvrirons d'abord la panique, car c'est la plus simple des deux.

La panique concerne l'autre type d'erreur, celle qui *ne devrait jamais arriver*.

Panique

Un programme panique lorsqu'il rencontre quelque chose de tellement confus qu'il doit y avoir un bogue dans le programme lui-même. Quelque chose comme :

- Accès à la baie hors limites
- Division entière par zéro
- Faire appel `.expect()` à un `Result` qui se trouve être `Err`
- Échec de l'assertion

(Il y a aussi la macro `panic!()`, pour les cas où votre propre code découvre qu'il a mal tourné, et vous devez donc déclencher directement une panique. `panic!()` accepte `println!()` arguments facultatifs de style `-`, pour la construction d'un message d'erreur.)

Ce que ces conditions ont en commun, c'est qu'elles sont toutes, sans trop insister là-dessus, la faute du programmeur. Une bonne règle d'or est : « Ne paniquez pas.

Mais nous faisons tous des erreurs. Lorsque ces erreurs qui ne devraient pas se produire se produisent, que se passe-t-il alors ? Remarquablement, Rust vous donne le choix. Rust peut soit dérouler la pile en cas de panique, soit interrompre le processus. Le déroulement est la valeur par défaut.

Se détendre

Lorsque les pirates se partagent le butin d'un raid, le capitaine reçoit la moitié du butin. Les membres d'équipage ordinaires gagnent des parts égales de l'autre moitié. (Les pirates détestent les fractions, donc si l'une ou l'autre des divisions n'est pas égale, le résultat est arrondi à l'inférieur, le reste allant au perroquet du navire.)

```
fn pirate_share(total: u64, crew_size: usize) ->u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

Cela peut bien fonctionner pendant des siècles jusqu'au jour où il s'avère que le capitaine est le seul survivant d'un raid. Si nous passons a `crew_size` de zéro à cette fonction, elle divisera par zéro. En C++, ce serait un comportement indéfini. Dans Rust, cela déclenche une panique, qui se déroule généralement comme suit :

- Un message d'erreur est imprimé sur le terminal :

```
thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Si vous définissez la `RUST_BACKTRACE` variable d'environnement, comme le suggèrent les messages, Rust videra également la pile à ce stade.

- La pile est déroulée. Cela ressemble beaucoup à la gestion des exceptions C++.

Toutes les valeurs temporaires, variables locales ou arguments que la fonction actuelle utilisait sont supprimés, dans l'ordre inverse de leur création. Supprimer une valeur signifie simplement nettoyer après elle : tous les `String`s ou `Vec`s que le programme utilisait sont libé-

rés, tous les `File`s ouverts sont fermés, et ainsi de suite. drop Les méthodes définies par l'utilisateur sont également appelées ; voir "[Déposer](#)". Dans le cas particulier de `pirate_share()`, il n'y a rien à nettoyer.

Une fois l'appel de fonction en cours nettoyé, nous passons à son appellant, en supprimant ses variables et ses arguments de la même manière. Ensuite, nous passons à l'appelant de *cette* fonction, et ainsi de suite dans la pile.

- Enfin, le fil se termine. Si le thread paniqué était le thread principal, alors tout le processus se termine (avec un code de sortie différent de zéro).

La panique est peut-être un nom trompeur pour ce processus ordonné. Une panique n'est pas un crash. Ce n'est pas un comportement indéfini. C'est plus comme un `RuntimeException` en Java ou un `std::logic_error` en C++. Le comportement est bien défini ; ça ne devrait pas arriver.

La panique est sans danger. Il ne viole aucune des règles de sécurité de Rust ; même si vous parvenez à paniquer au milieu d'une méthode de bibliothèque standard, elle ne laissera jamais un pointeur pendant ou une valeur à moitié initialisée en mémoire. L'idée est que Rust intercepte l'accès invalide au tableau, ou quoi que ce soit, *avant* que quelque chose de mal ne se produise. Il serait dangereux de continuer, donc Rust déroule la pile. Mais le reste du processus peut continuer à s'exécuter.

La panique est par thread. Un thread peut paniquer tandis que d'autres threads poursuivent leurs activités normales. Au [chapitre 19](#), nous montrerons comment un thread parent peut savoir quand un thread enfant panique et gérer l'erreur avec élégance.

Il existe également un moyen d'*attraper* le déroulement de la pile, permettant au thread de survivre et de continuer à fonctionner. La fonction de bibliothèque standard `std::panic::catch_unwind()` est ce que ça. Nous n'expliquerons pas comment l'utiliser, mais c'est le mécanisme utilisé par le harnais de test de Rust pour récupérer lorsqu'une assertion échoue dans un test. (Cela peut également être nécessaire lors de l'écriture de code Rust pouvant être appelé depuis C ou C++, car le déroulement dans du code non-Rust est un comportement indéfini ; voir le [chapitre 22](#).)

Idéalement, nous aurions tous un code sans bogue qui ne panique jamais. Mais personne n'est parfait. Vous pouvez utiliser des threads et `catch_unwind()` gérer la panique, ce qui rend votre programme plus ro-

buste. Une mise en garde importante est que ces outils n'attrapent que les paniques qui déroulent la pile. Toutes les paniques ne se déroulent pas de cette façon.

Abandon

Empiler le déroulement est le comportement de panique par défaut, mais il existe deux circonstances dans lesquelles Rust n'essaie pas de dérouler la pile.

Si une `.drop()` méthode déclenche une deuxième panique alors que Rust essaie toujours de nettoyer après la première, cela est considéré comme fatal. Rust arrête de se dérouler et interrompt tout le processus.

De plus, le comportement de panique de Rust est personnalisable. Si vous编译 avec `-C panic=abort`, la première panique de votre programme interrompt immédiatement le processus. (Avec cette option, Rust n'a pas besoin de savoir comment dérouler la pile, ce qui peut réduire la taille de votre code compilé.)

Ceci conclut notre discussion sur la panique dans Rust. Il n'y a pas grand-chose à dire, car le code Rust ordinaire n'a aucune obligation de gérer la panique. Même si vous utilisez des threads ou `catch_unwind()`, tout votre code de gestion de panique sera probablement concentré à quelques endroits. Il est déraisonnable de s'attendre à ce que chaque fonction d'un programme anticipe et gère les bogues dans son propre code. Les erreurs causées par d'autres facteurs sont une autre paire de manches.

Résultat

Rust n'a pas d'exceptions. Au lieu de cela, les fonctions qui peuvent échouer ont un type de retour qui le dit :

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

Le `Result` type indique une panne possible. Lorsque nous appelons la `get_weather()` fonction, elle renverra soit un *résultat de réussite* `Ok(weather)`, où `weather` est une nouvelle `WeatherReport` valeur, soit un *résultat d'erreur* `Err(error_value)`, où `error_value` est une `io::Error` explication de ce qui s'est mal passé.

Rust nous oblige à écrire une sorte de gestion des erreurs chaque fois que nous appelons cette fonction. Nous ne pouvons pas accéder au `WeatherReport` sans faire *quelque chose* au `Result`, et vous recevrez un avertissement du compilateur si une `Result` valeur n'est pas utilisée.

Au [chapitre 10](#), nous verrons comment la bibliothèque standard définit `Result` et comment vous pouvez définir vos propres types similaires. Pour l'instant, nous allons adopter une approche de « livre de recettes » et nous concentrer sur la façon d'utiliser `Results` pour obtenir le comportement de gestion des erreurs que vous souhaitez. Nous verrons comment intercepter, propager et signaler les erreurs, ainsi que les modèles courants d'organisation et d'utilisation des `Result` types.

Attraper les erreurs

La façon la plus complète de traiter a `Result` est celle que nous avons montrée au [chapitre 2](#) : utiliser une `match` expression.

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

C'est l'équivalent de Rust `try/catch` dans d'autres langages. C'est ce que vous utilisez lorsque vous voulez gérer les erreurs de front, et non les transmettre à votre interlocuteur.

`match` est un peu verbeux, `Result<T, E>` offre donc une variété de méthodes qui sont utiles dans des cas courants particuliers. Chacune de ces méthodes a une `match` expression dans son implémentation. (Pour la liste complète des `Result` méthodes, consultez la documentation en ligne. Les méthodes répertoriées ici sont celles que nous utilisons le plus.)

```
result.is_ok(), result.is_err()
```

Revenir un `bool` dire si `result` est un résultat de succès ou un résultat d'erreur.

```
result.ok()
```

Retourne la valeur de réussite, le cas échéant, en tant que `Option<T>`. Si `result` est un résultat de réussite, cela renvoie `Some(success_value)` ;

sinon, il renvoie `None`, en supprimant la valeur d'erreur.

`result.err()`

Retourne la valeur d'erreur, le cas échéant, sous forme de `Option<E>`.

`result.unwrap_or(fallback)`

Retourne la valeur de réussite, si `result` est un résultat de réussite.

Sinon, elle renvoie `fallback`, en supprimant la valeur d'erreur.

```
// A fairly safe prediction for Southern California.  
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);  
  
// Get a real weather report, if possible.  
// If not, fall back on the usual.  
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);  
display_weather(los_angeles, &report);
```

C'est une bonne alternative à `.ok()` parce que le type de retour est `T`, pas `Option<T>`. Bien sûr, cela ne fonctionne que lorsqu'il existe une valeur de repli appropriée.

`result.unwrap_or_else(fallback_fn)`

C'est le même, mais au lieu de transmettre directement une valeur de secours, vous transmettez une fonction ou une fermeture. C'est pour les cas où il serait inutile de calculer une valeur de repli si vous n'allez pas l'utiliser. Le `fallback_fn` n'est appelé que si nous avons un résultat d'erreur.

```
let report =  
    get_weather(hometown)  
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

([Le chapitre 14](#) couvre les fermetures en détail.)

`result.unwrap()`

Retourne également la valeur de réussite, si `result` est un résultat de réussite. Cependant, s'il `result` y a un résultat d'erreur, cette méthode panique. Cette méthode a ses utilisations ; nous en reparlerons plus tard.

`result.expect(message)`

Cette identique à `.unwrap()`, mais vous permet de fournir un message qu'il imprime en cas de panique.

Enfin, les méthodes pour travailler avec des références dans un `Result` :

```
result.as_ref()

Convertit un Result<T, E> à un Result<&T, &E>.
```

```
result.as_mut()

Cette est le même, mais emprunte une référence mutable. Le type de retour est
Result<&mut T, &mut E>.
```

L'une des raisons pour lesquelles ces deux dernières méthodes sont utiles est que toutes les autres méthodes répertoriées ici, à l'exception de `.is_ok()` et `.is_err()`, consomment le `result` sur lequel elles opèrent. Autrement dit, ils prennent l' `self` argument par valeur. Parfois, il est assez pratique d'accéder aux données à l'intérieur d'un `result` sans les détruire, et c'est ce `.as_ref()` que `.as_mut()` nous faisons pour nous. Par exemple, supposons que vous vouliez appeler `result.ok()`, mais que vous deviez `result` rester intact. Vous pouvez écrire `result.as_ref().ok()`, qui emprunte simplement `result`, renvoyant un `Option<&T>` plutôt qu'un `Option<T>`.

Alias de type de résultat

quelquefois vous verrez la documentation de Rust qui semble omettre le type d'erreur de `a Result`:

```
fn remove_file(path: &Path) ->Result<()>
```

Cela signifie qu'un `Result` alias de type est utilisé.

Un alias de type est une sorte de raccourci pour les noms de type. Les modules définissent souvent un `Result` alias de type pour éviter d'avoir à répéter un type d'erreur qui est utilisé de manière cohérente par presque toutes les fonctions du module. Par exemple, le module de la bibliothèque standard `std::io` inclut cette ligne de code :

```
pub type Result<T> = result::Result<T, Error>;
```

Cela définit un type public `std::io::Result<T>`. C'est un alias pour `Result<T, E>`, mais il est codé en dur `std::io::Error` comme type d'erreur. Concrètement, cela signifie que si vous écrivez `use std::io;`, alors Rust comprendra `io::Result<String>` comme un raccourci pour `Result<String, io::Error>`.

Lorsque quelque chose comme `Result<()>` apparaît dans la documentation en ligne, vous pouvez cliquer sur l'identifiant `Result` pour voir quel

alias de type est utilisé et connaître le type d'erreur. En pratique, c'est généralement évident d'après le contexte.

Erreurs d'impression

quelquefois la seule façon de gérer une erreur est de la vider dans le terminal et de passer à autre chose. Nous avons déjà montré une façon de procéder :

```
println!("error querying the weather: {}", err);
```

La bibliothèque standard définit plusieurs types d'erreurs avec des noms ennuyeux : `std::io::Error`, `std::fmt::Error`, `std::str::Utf8Error`, etc. Tous implémentent une interface commune `std::error::Error`, le trait, ce qui signifie qu'ils partagent les fonctionnalités et méthodes suivantes :

```
println!()
```

Toutes les erreurs les types sont imprimables en utilisant ceci. L'impression d'une erreur avec le `{}` spécificateur de format n'affiche généralement qu'un bref message d'erreur. Vous pouvez également imprimer avec le `{:?}` spécificateur de format pour obtenir une Debug vue de l'erreur. Ceci est moins convivial, mais comprend des informations techniques supplémentaires.

```
// result of `println!("error: {}", err);`  
error: failed to look up address information: No address associated w  
hostname  
  
// result of `println!("error: {:?}", err);`  
error: Error { repr: Custom(Custom { kind: Other, error: StringError(  
"failed to look up address information: No address associated with  
hostname") }) }
```

```
err.to_string()
```

Retourne un message d'erreur sous forme de `String`.

```
err.source()
```

Retourne une `Option` de l'erreur sous-jacente, le cas échéant, qui a causé `err`. Par exemple, une erreur de réseau peut entraîner l'échec d'une transaction bancaire, ce qui pourrait entraîner la reprise de possession de votre bateau. Si `err.to_string()` est "boat was reposessed", alors `err.source()` peut renvoyer une erreur concernant la transaction ayant

échoué. Cette erreur `.to_string()` peut être , et il peut s'agir d'un avec des détails sur la panne de réseau spécifique qui a causé tout ce remue-ménage.

Cette troisième erreur est la cause première, donc sa méthode renverrait . Étant donné que la bibliothèque standard ne comprend que des fonctionnalités plutôt de bas niveau, la source des erreurs renvoyées par la bibliothèque standard est généralement . "failed to transfer \$300 to United Yacht Supply" .source() io::Error .source() None None

L'impression d'une valeur d'erreur n'imprime pas également sa source. Si vous voulez être sûr d'imprimer toutes les informations disponibles, utilisez cette fonction :

```
use std::error:: Error;
use std::io::{Write, stderr};

/// Dump an error message to `stderr`.
///
/// If another error happens while building the error message or
/// writing to `stderr`, it is ignored.
fn print_error(mut err:&dyn Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {}", source);
        err = source;
    }
}
```

La `writeln!` macrofonctionne comme `println!`, sauf qu'il écrit les données dans un flux de votre choix. Ici, nous écrivons les messages d'erreur dans le flux d'erreur standard, `std::io::stderr`. Nous pourrions utiliser la `eprintln!` macro pour faire la même chose, mais `eprintln!` panique si une erreur se produit. Dans `print_error`, nous voulons ignorer les erreurs qui surviennent lors de l'écriture du message ; nous expliquons pourquoi dans [« Ignorer les erreurs »](#), plus loin dans le chapitre.

Les types d'erreur de la bibliothèque standard n'incluent pas de trace de pile, mais le `anyhow` crate populaire fournit un type d'erreur prêt à l'emploi qui le fait, lorsqu'il est utilisé avec une version instable du compilateur Rust. (A partir de Rust 1.56, les fonctions de la bibliothèque standard pour capturer les backtraces n'étaient pas encore stabilisées.)

Propagation des erreurs

Dans la plupart des endroits où nous essayons quelque chose qui pourrait échouer, nous ne voulons pas attraper et gérer l'erreur immédiatement. C'est tout simplement trop de code pour utiliser une `match` instruction de 10 lignes à chaque endroit où quelque chose pourrait mal tourner.

Au lieu de cela, si une erreur se produit, nous voulons généralement laisser notre appelant s'en occuper. Nous voulons que les erreurs se *propagent* dans la pile des appels.

Rust a un `? opérateur` qui fait ça. Vous pouvez ajouter un `?` à toute expression qui produit un `Result`, comme le résultat d'un appel de fonction :

```
let weather = get_weather(hometown)?;
```

Le comportement de `?` dépend du fait que cette fonction renvoie un résultat de réussite ou un résultat d'erreur :

- En cas de succès, il déballe le `Result` pour obtenir la valeur de succès à l'intérieur. Le type d'`weather` ici n'est pas `Result<WeatherReport, io::Error>` mais simplement `WeatherReport`.
- En cas d'erreur, il revient immédiatement de la fonction englobante, transmettant le résultat de l'erreur dans la chaîne d'appel. Pour s'assurer que cela fonctionne, `?` ne peut être utilisé que sur `Result` des fonctions qui ont un `Result` type de retour.

? L'opérateur n'a rien de magique. Vous pouvez exprimer la même chose en utilisant une `match` expression, bien qu'elle soit beaucoup plus verbosse :

```
let weather = match get_weather(hometown) {  
    Ok(success_value) => success_value,  
    Err(err) => return Err(err)  
};
```

Les seules différences entre ceci et l'`?` opérateur sont quelques petits détails concernant les types et les conversions. Nous couvrirons ces détails dans la section suivante.

Dans le code plus ancien, vous pouvez voir la `try!()` macro, qui était la manière habituelle de propager les erreurs jusqu'à ce que l'`?` opérateur soit introduit dans Rust 1.13 :

```
let weather = try!(get_weather(hometown));
```

La macro se développe en une `match` expression, comme la précédente.

Il est facile d'oublier à quel point la possibilité d'erreurs est omniprésente dans un programme, en particulier dans le code qui s'interface avec le système d'exploitation. L' `? opérateur` apparaît parfois sur presque toutes les lignes d'une fonction :

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

fn move_all(src: &Path, dst: &Path) -> io:: Result<()> {
    for entry_result in src.read_dir()? { // opening dir could fail
        let entry = entry_result?;           // reading dir could fail
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?; // renaming could fail
    }
    Ok(()) // phew!
}
```

`?` fonctionne également de la même manière avec le `Option` type. Dans une fonction qui retourne `Option`, vous pouvez utiliser `?` pour déballer une valeur et revenir tôt dans le cas de `None`:

```
let weather = get_weather(hometown).ok()?;

```

Travailler avec plusieurs types d'erreurs

Souvent, plusqu'une chose pourrait mal tourner. Supposons que nous lisons simplement des nombres à partir d'un fichier texte :

```
use std:: io::{self, BufRead};

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;           // reading lines can fail
        numbers.push(line.parse()?)?;     // parsing integers can fail
    }
    Ok(numbers)
}
```

Rust nous renvoie une erreur de compilation :

```
error: `?` couldn't convert the error to `std::io::Error`

    numbers.push(line.parse()?);      // parsing integers can fail
                                         ^
the trait `std::convert::From<std::num::ParseIntError>`
is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait
```

Les termes de ce message d'erreur auront plus de sens lorsque nous atteindrons le [chapitre 11](#), qui couvre les traits. Pour l'instant, notez simplement que Rust se plaint que l' `? opérateur` ne peut pas convertir une `std::num::ParseIntError` valeur en type `std::io::Error`.

Le problème ici est que la lecture d'une ligne d'un fichier et l'analyse d'un entier produisent deux types d'erreurs potentiels différents. Le type de `line_result` est `Result<String, std::io::Error>`. Le type de `line.parse()` est `Result<i64, std::num::ParseIntError>`. Le type de retour de notre `read_numbers()` fonction n'accepte que `io::Error`s. Rust essaie de gérer le `ParseIntError` en le convertissant en un `io::Error`, mais il n'y a pas une telle conversion, nous obtenons donc une erreur de type.

Il y a plusieurs façons de traiter cela. Par exemple, le `image` crate que nous avons utilisé au [chapitre 2](#) pour créer les fichiers image de l'ensemble de Mandelbrot définit son propre type d'erreur, `ImageError`, et implémente les conversions de `io::Error` et plusieurs autres types d'erreur vers `ImageError`. Si vous souhaitez emprunter cette voie, essayez la `thiserror` caisse, qui est conçue pour vous aider à définir de bons types d'erreurs avec seulement quelques lignes de code.

Une approche plus simple consiste à utiliser ce qui est intégré à Rust. Tous les types d'erreur de bibliothèque standard peuvent être convertis en type `Box<dyn std::error::Error + Send + Sync + 'static>`. C'est un peu long, mais `dyn std::error::Error` cela représente "n'importe quelle erreur" et `Send + Sync + 'static` permet de passer en toute sécurité entre les threads, ce que vous voudrez souvent.¹ Pour plus de commodité, vous pouvez définir des alias de type :

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

Ensuite, changez le type de retour de `read_numbers()` en `GenericResult<Vec<i64>>`. Avec ce changement, la fonction compile. L' `? opérateur` convertit automatiquement l'un ou l'autre type d'erreur en à `GenericError` selon les besoins.

Incidemment, l' `? opérateur` effectue cette conversion automatique en utilisant une méthode standard que vous pouvez utiliser vous-même.

Pour convertir toute erreur en `GenericError` type, appelez

```
GenericError::from():
```

```
let io_error = io:: Error:: new(           // make our own io::Error
    io:: ErrorKind:: Other, "timed out");
return Err(GenericError::from(io_error)); // manually convert to GenericError
```

Nous couvrirons entièrement le `From` trait et sa `from()` méthode au [chapitre 13](#).

L'inconvénient de l' `GenericError` approche est que le type de retour ne communique plus précisément à quels types d'erreurs l'appelant peut s'attendre. L'appelant doit être prêt à tout.

Si vousappelez une fonction qui renvoie à `GenericResult` et que vous souhaitez gérer un type particulier d'erreur mais laisser toutes les autres se propager, utilisez la méthode générique `error.downcast_ref::<ErrorType>()`. Il emprunte une référence à l'erreur, si l'il s'agit du type particulier d'erreur que vous recherchez :

```
loop {
    match compile_project() {
        Ok(() ) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>()
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue; // try again!
            }
            return Err(err);
        }
    }
}
```

De nombreux langages ont une syntaxe intégrée pour ce faire, mais cela s'avère rarement nécessaire. Rust a une méthode pour cela à la place.

Traiter les erreurs qui « ne peuvent pas se produire »

quelquefois nous savons juste qu'une erreur ne peut pas arriver. Par exemple, supposons que nous écrivions du code pour analyser un fichier de configuration et qu'à un moment donné, nous découvrions que la prochaine chose dans le fichier est une chaîne de chiffres :

```
if next_char.is_digit(10) {  
    let start = current_index;  
    current_index = skip_digits(&line, current_index);  
    let digits = &line[start..current_index];  
    ...  
}
```

Nous voulons convertir cette chaîne de chiffres en un nombre réel. Il existe une méthode standard qui fait cela:

```
let num = digits.parse::<u64>();
```

Maintenant le problème : la `str.parse::<u64>()` méthode ne renvoie pas un `u64`. Il renvoie un `Result`. Cela peut échouer, car certaines chaînes ne sont pas numériques :

```
"bleen".parse::<u64>() // ParseIntError: invalid digit
```

Mais il se trouve que nous savons que dans ce cas, `digits` se compose entièrement de chiffres. Que devrions nous faire?

Si le code que nous écrivons renvoie déjà un `GenericResult`, nous pouvons ajouter un `?` et l'oublier. Sinon, nous sommes confrontés à la perspective irritante de devoir écrire du code de gestion des erreurs pour une erreur qui ne peut pas se produire. Le meilleur choix serait alors d'utiliser `.unwrap()`, une `Result` méthode qui panique si le résultat est un `Err`, mais renvoie simplement la valeur de succès d'un `Ok`:

```
let num = digits.parse::<u64>().unwrap();
```

C'est exactement comme `?` sauf que si nous nous trompons sur cette erreur, si cela *peut* arriver, alors dans ce cas, nous paniquerions.

En fait, nous nous trompons sur ce cas particulier. Si l'entrée contient une chaîne de chiffres suffisamment longue, le nombre sera trop grand pour tenir dans un `u64` :

```
"999999999999999999999999".parse::<u64>()      // overflow error
```

L'utiliser `.unwrap()` dans ce cas particulier serait donc un bug. Une fausse entrée ne devrait pas provoquer de panique.

Cela dit, des situations surviennent où une `Result` valeur ne peut vraiment pas être une erreur. Par exemple, au [chapitre 18](#), vous verrez que le `Write` trait définit un ensemble commun de méthodes (`.write()` et d'autres) pour le texte et la sortie binaire. Toutes ces méthodes renvoient `io::Result`s, mais si vous écrivez dans un `Vec<u8>`, elles ne peuvent pas échouer. Dans de tels cas, il est acceptable d'utiliser `.unwrap()` ou `.expect(message)` de renoncer à l'`Result` art.

Ces méthodes sont également utiles lorsqu'une erreur indique une condition si grave ou bizarre que la panique est exactement la façon dont vous voulez la gérer :

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
    ...
}
```

Ici, la `.elapsed()` méthode peut échouer que si l'heure système est *antérieure* à la date de création du fichier. Cela peut se produire si le fichier a été créé récemment et que l'horloge système a été ajustée à l'envers pendant l'exécution de notre programme. Selon la façon dont ce code est utilisé, il est raisonnable de paniquer dans ce cas, plutôt que de gérer l'erreur ou de la propager à l'appelant.

Ignorer les erreurs

Parfois, nous voulons juste ignorer une erreur en somme. Par exemple, dans notre `print_error()` fonction, nous avons dû gérer la situation improbable où l'impression de l'erreur déclenche une autre erreur. Cela peut arriver, par exemple, si `stderr` est redirigé vers un autre processus et que ce processus est tué. L'erreur d'origine que nous essayions de signaler est probablement plus importante à propager, nous voulons donc simplement ignorer les problèmes avec `stderr`, mais le compilateur Rust avertit des `Result` valeurs inutilisées :

```
writeln!(stderr(), "error: {}", err); // warning: unused result
```

L'idiome `let _ = ...` est utilisé pour faire taire cet avertissement :

```
let _ = writeln!(stderr(), "error: {}", err); // ok, ignore result
```

Gestion des erreurs dans main()

Dans la plupart des endroits où `a` `Result` est produit, laisser l'erreur remonter jusqu'à l'appelant est le bon comportement. C'est pourquoi `? est un seul personnage dans Rust. Comme nous l'avons vu, dans certains programmes, il est utilisé sur plusieurs lignes de code à la suite.`

Mais si vous propagez une erreur assez longtemps, elle finit par atteindre `main()`, et quelque chose doit être fait avec. Normalement, `main()` ne peut pas utiliser `?` car son type de retour n'est pas `Result` :

```
fn main() {
    calculate_tides()?;
    // error: can't pass the buck any further
}
```

La manière la plus simple de gérer les erreurs en `main()` est d'utiliser `.expect()` :

```
fn main() {
    calculate_tides().expect("error");
    // the buck stops here
}
```

Si `calculate_tides()` revient un résultat d'erreur, la `.expect()` méthode panique. Paniquer dans le thread principal imprime un message d'erreur, puis se termine avec un code de sortie différent de zéro, ce qui correspond à peu près au comportement souhaité. Nous l'utilisons tout le temps pour de petits programmes. C'est un début.

Le message d'erreur est un peu intimidant, cependant :

```
$tidecalc --planète mercure
thread 'main' panicked at 'error: "moon not found"', src/main.rs:2:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrac
```

Le message d'erreur se perd dans le bruit. Aussi, `RUST_BACKTRACE=1` est un mauvais conseil dans ce cas particulier.

Cependant, vous pouvez également modifier la signature de type de `main()` pour renvoyer un `Result` type, vous pouvez donc utiliser `?:`:

```
fn main() ->Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

Cela fonctionne pour tout type d'erreur qui peut être imprimé avec le `{::?}` formateur, ce que peuvent être tous les types d'erreur standard, comme `std::io::Error`. Cette technique est facile à utiliser et donne un message d'erreur un peu plus agréable, mais ce n'est pas idéal :

```
$tidecalc --planète mercure
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

Si vous avez des types d'erreurs plus complexes ou si vous souhaitez inclure plus de détails dans votre message, il est préférable d'imprimer vous-même le message d'erreur :

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

Ce code utilise une `if let` expression pour imprimer le message d'erreur uniquement si l'appel à `calculate_tides()` renvoie un résultat d'erreur. Pour plus de détails sur `if let` les expressions, voir [Chapitre 10](#). La `print_error` fonction est répertoriée dans "[Erreurs d'impression](#)".

Maintenant, la sortie est belle et bien rangée :

```
$tidecalc --planète mercure
error: moon not found
```

Déclarer un type d'erreur personnalisé

Supposons que vous écrivez un nouvel analyseur JSON et vous souhaitez qu'il ait son propre type d'erreur. (Nous n'avons pas encore couvert les types

définis par l'utilisateur ; cela arrivera dans quelques chapitres. Mais les types d'erreur sont pratiques, nous allons donc inclure un petit aperçu ici.)

Approximativement, le code minimum que vous écririez est :

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

Cette structure s'appellera `json::error::JsonError`, et lorsque vous voudrez lever une erreur de ce type, vous pourrez écrire :

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

Cela fonctionnera bien. Cependant, si vous voulez que votre type d'erreur fonctionne comme les types d'erreur standard, comme les utilisateurs de votre bibliothèque s'y attendent, alors vous avez un peu plus de travail à faire :

```
use std::fmt;

// Errors should be printable.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// Errors should implement the std::error::Error trait,
// but the default definitions for the Error methods are fine.
impl std::error::Error for JsonError { }
```

Encore une fois, la signification du mot- `impl` clé, `self`, et tout le reste seront expliqués dans les prochains chapitres.

Comme pour de nombreux aspects du langage Rust, les caisses existent pour rendre la gestion des erreurs beaucoup plus facile et plus concise. Il en existe une grande variété, mais l'un des plus utilisés est `thiserror`, qui fait tout le travail précédent pour vous, vous permettant d'écrire des erreurs comme celle-ci :

```
use thiserror:: Error;
#[derive(Error, Debug)]
#[error("{message}: {line}, {column}")]
pub struct JsonError {
    message: String,
    line: usize,
    column: usize,
}
```

La `#[derive(Error)]` directive indique `thiserror` de générer le code présenté précédemment, ce qui peut économiser beaucoup de temps et d'efforts.

Pourquoi Résultats ?

Maintenant, nous en savons assez pour comprendre ce que Rust est en train de choisir `Result`s plutôt que des exceptions. Voici les points clés de la conception :

- Rust oblige le programmeur à prendre une sorte de décision et à l'enregistrer dans le code, à chaque point où une erreur pourrait se produire. C'est une bonne chose car sinon, il est facile de se tromper dans la gestion des erreurs par négligence.
- La décision la plus courante est de permettre aux erreurs de se propager, et c'est écrit avec un seul caractère, `? .` Ainsi, la plomberie d'erreur n'encombre pas votre code comme elle le fait en C and Go. Pourtant, il est toujours visible : vous pouvez regarder un morceau de code et voir d'un coup d'œil tous les endroits où les erreurs se propagent.
- Étant donné que la possibilité d'erreurs fait partie du type de retour de chaque fonction, il est clair quelles fonctions peuvent échouer et lesquelles ne le peuvent pas. Si vous modifiez une fonction pour qu'elle soit faillible, vous modifiez son type de retour, de sorte que le compilateur vous obligera à mettre à jour les utilisateurs en aval de cette fonction.
- Rust vérifie que `Result` les valeurs sont utilisées, vous ne pouvez donc pas laisser passer accidentellement une erreur en silence (une erreur courante en C).

- Puisqu'il `Result` s'agit d'un type de données comme les autres, il est facile de stocker les résultats de réussite et d'erreur dans la même collection. Cela facilite la modélisation d'un succès partiel. Par exemple, si vous écrivez un programme qui charge des millions d'enregistrements à partir d'un fichier texte et que vous avez besoin d'un moyen de faire face au résultat probable que la plupart réussiront, mais que certains échoueront, vous pouvez représenter cette situation en mémoire à l'aide d'un vecteur de `Result` l'art.

Le coût est que vous vous retrouverez à penser et à gérer les erreurs d'ingénierie plus dans Rust que vous ne le feriez dans d'autres langages.

Comme dans de nombreux autres domaines, la gestion des erreurs de Rust est un peu plus stricte que ce à quoi vous êtes habitué. Pour la programmation système, cela vaut la peinece.

¹ Vous devriez également envisager d'utiliser la caisse populaire `anyhow`, qui fournit des types d'erreur et de résultat très similaires à nos `GenericError` et `GenericResult`, mais avec quelques fonctionnalités supplémentaires intéressantes.

Chapitre 8. Caisses et modules

C'est une remarque dans un thème Rust : les programmeurs systèmes peuvent avoir de belles choses.

—Robert O'Callahan, "[Réflexions aléatoires sur Rust : crates.io et les IDE](#)"

Supposons que vous écriviez un programme qui simule la croissance des fougères, du niveau des cellules individuelles vers le haut. Votre programme, comme une fougère, commencera très simplement, avec tout le code, peut-être, dans un seul fichier - juste la spore d'une idée. Au fur et à mesure de sa croissance, il commencera à avoir une structure interne. Différentes pièces auront des objectifs différents. Il se ramifiera en plusieurs fichiers. Il peut couvrir toute une arborescence de répertoires. Avec le temps, il peut devenir une partie importante de tout un écosystème logiciel. Pour tout programme qui dépasse quelques structures de données ou quelques centaines de lignes, une certaine organisation est nécessaire.

Ce chapitre couvre les fonctionnalités de Rust qui aident à garder votre programme organisé : crates et modules. Nous couvrirons également d'autres sujets liés à la structure et à la distribution d'une caisse Rust, y compris comment documenter et tester le code Rust, comment faire taire les avertissements indésirables du compilateur, comment utiliser Cargo pour gérer les dépendances et les versions du projet, comment publier l'open source bibliothèques sur le référentiel de caisses public de Rust, crates.io, comment Rust évolue à travers les éditions de langage, et plus encore, en utilisant le simulateur de fougère comme exemple courant.

Caisses

Les programmes de rouille sont faits de *caisses*. Chaque caisse est une unité complète et cohérente : tout le code source d'une seule bibliothèque ou exécutable, ainsi que tous les tests, exemples, outils, configurations et autres éléments inutiles associés. Pour votre simulateur de fougère, vous pouvez utiliser des bibliothèques tierces pour les graphiques 3D, la bioinformatique, le calcul parallèle, etc. Ces bibliothèques sont distribuées sous forme de caisses (voir [Figure 8-1](#)).

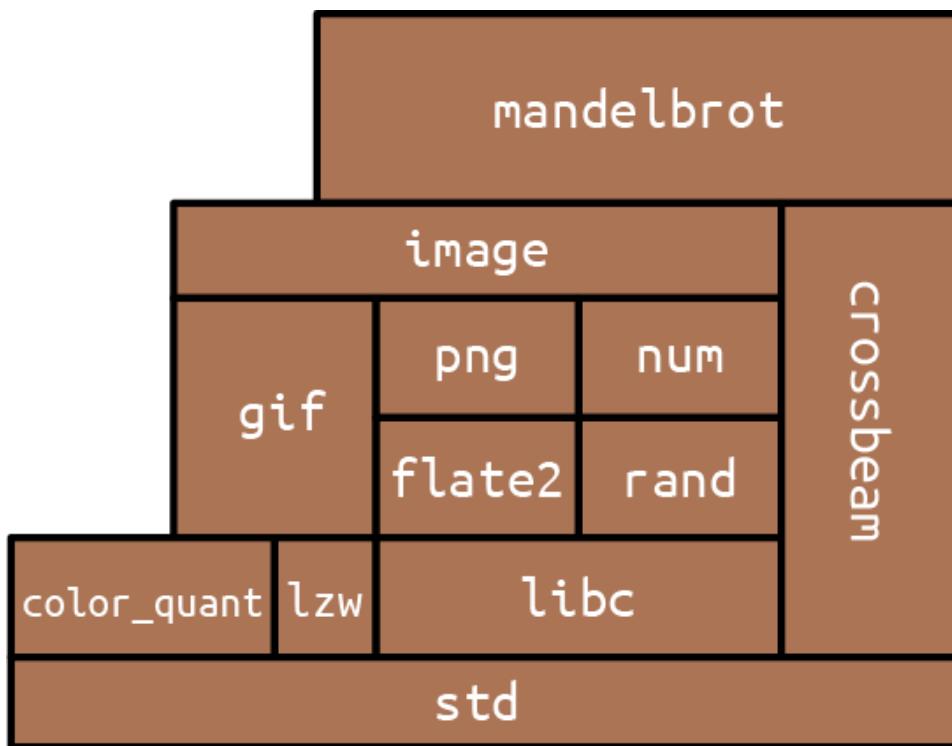


Illustration 8-1. Une caisse et ses dépendances

La façon la plus simple de voir ce que sont les caisses et comment elles fonctionnent ensemble est d'utiliser `cargo build` avec le `--verbose` drapeau pour construire un projet existant qui a des dépendances. Nous l'avons fait en utilisant "[Un programme Mandelbrot simulé](#)" comme exemple. Les résultats sont affichés ici :

```

$ cd cargaison/de/mandelbrot
     construction $ de cargaison propre      # delete previously compiled code
$ --verbose
     Updating registry `https://github.com/rust-lang/crates.io-index`
     Downloading autocfg v1.0.0
     Downloading semver-parser v0.7.0
     Downloading gif v0.9.0
     Downloading png v0.7.0

     ... (downloading and compiling many more crates)

Compiling jpeg-decoder v0.1.18
     Running `rustc
             --crate-name jpeg_decoder
             --crate-type lib
             ...
             --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
             ...
     Compiling image v0.13.0
     Running `rustc
             --crate-name image
             --crate-type lib
             ...
             --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
             --extern gif=.../libgif-a7006d35f1b58927.rmeta

```

```
--extern jpeg_decoder=.../libjpeg_decoder-5c10558d0d57d300.rmeta
Compiling mandelbrot v0.1.0 (/tmp/rustbook-test-files/mandelbrot)
    Running `rustc
        --edition=2021
        --crate-name mandelbrot
        --crate-type bin
        ...
        --extern crossbeam=.../libcrossbeam-f87b4b3d3284acc2.rlib
        --extern image=.../libimage-b5737c12bd641c43.rlib
        --extern num=.../libnum-1974e9a1dc582ba7.rlib -C link-arg=-fuse-ld=1
Finished dev [unoptimized + debuginfo] target(s) in 16.94s
$
```

Nous avons reformaté la `rustc` commandelignes pour plus de lisibilité, et nous avons supprimé de nombreuses options du compilateur qui ne sont pas pertinentes pour notre discussion, en les remplaçant par des points de suspension (...).

Vous vous souviendrez peut-être qu'au moment où nous avions terminé, le `main.rs` du programme Mandelbrot contenait plusieurs `use` déclarations pour des éléments provenant d'autres caisses :

```
use num:: Complex;
// ...
use image:: ColorType;
use image:: png::PNGEncoder;
```

Nous avons également précisé dans notre fichier `Cargo.toml` quelle version de chaque caisse nous voulions :

```
[dépendances]
nombre = "0.4"
image="0.13"
traverse = "0.8"
```

Le mot `dépendances`ici signifie simplement d'autres caisses utilisées par ce projet : le code dont nous dépendons. Nous avons trouvé ces caisses sur [crates.io](#), le site de la communauté Rust pour les caisses open source. Par exemple, nous avons découvert la `image` bibliothèque en allant sur crates.io et en recherchant une bibliothèque d'images. La page de chaque caisse sur crates.io affiche son fichier `README.md` et des liens vers la documentation et la source, ainsi qu'une ligne de configuration comme `image = "0.13"` celle que vous pouvez copier et ajouter à votre `Cargo.toml`. Les numéros de version indiqués ici sont simplement les dernières versions de ces trois packages au moment où nous avons écrit le programme.

La transcription de Cargo raconte comment ces informations sont utilisées. Lorsque nous exécutons `cargo build`, Cargo commence par télécharger le code source pour les versions spécifiées de ces caisses à partir de crates.io. Ensuite, il lit les fichiers `Cargo.toml` de ces caisses, téléchargent leurs dépendances, etc. de manière récursive. Par exemple, le code source de la version 0.13.0 de la `image` caisse contient un fichier `Cargo.toml` qui inclut ceci :

```
[dépendances]
byteorder = "1.0.0"
num-iter = "0.1.32"
numérique-rationnel = "0.1.32"
num-traits = "0.1.32"
enum_primitive = "0.1.0"
```

Voyant cela, Cargo sait qu'avant de pouvoir utiliser `image`, il doit également récupérer ces caisses. Plus tard, nous verrons comment dire à Cargo de récupérer le code source d'un référentiel Git ou du système de fichiers local plutôt que de crates.io.

Puisque `mandelbrot` dépend de ces caisses indirectement, par son utilisation de la `image` caisse, nous les appelons *transitives dépendances* de `mandelbrot`. La collection de toutes ces relations de dépendance, qui indique à Cargo tout ce qu'il doit savoir sur les caisses à construire et dans quel ordre, est connue sous le nom *de graphique de dépendance* de la caisse. La gestion automatique par Cargo du graphe de dépendances et des dépendances transitives est une énorme victoire en termes de temps et d'efforts pour le programmeur.

Une fois qu'il a le code source, Cargo compile toutes les caisses. Il exécute `rustc`, le compilateur Rust, une fois pour chaque caisse dans le graphe de dépendance du projet. Lors de la compilation des bibliothèques, Cargo utilise l'`--crate-type lib` option. Cela indique `rustc` de ne pas rechercher une `main()` fonction mais plutôt de produire un fichier `.rlib` contenant du code compilé pouvant être utilisé pour créer des fichiers binaires et d'autres fichiers `.rlib`.

Lors de la compilation d'un programme, Cargo utilise `--crate-type bin`, et le résultat est un exécutable binaire pour la plate-forme cible : `mandelbrot.exe` sous Windows, par exemple.

Avec chaque `rustc` commande, Cargo passe `--extern` des options, donnant le nom de fichier de chaque bibliothèque que la caisse utilisera. De cette façon, lorsqu'il `rustc` voit une ligne de code comme `use image::png::PNGEncoder`, il peut comprendre qu'il `image` s'agit du nom d'un autre crate, et grâce à Cargo, il sait où trouver ce crate compilé

sur le disque. Le compilateur Rust a besoin d'accéder à ces fichiers *.rlib* car ils contiennent le code compilé de la bibliothèque. Rust liera statiquement ce code dans l'exécutable final. Le *.rlib* contient également des informations de type afin que Rust puisse vérifier que les fonctionnalités de la bibliothèque que nous utilisons dans notre code existent réellement dans la caisse et que nous les utilisons correctement. Il contient également une copie des fonctions publiques en ligne, des génériques et des macros de la caisse, des fonctionnalités qui ne peuvent pas être entièrement compilées en code machine tant que Rust ne voit pas comment nous les utilisons.

`cargo build` prend en charge toutes sortes d'options, dont la plupart sortent du cadre de ce livre, mais nous en mentionnerons une ici : `cargo build --release` produit une construction optimisée. Les versions de version s'exécutent plus rapidement, mais elles prennent plus de temps à compiler, elles ne vérifient pas le dépassement d'entier, elles ignorent les `debug_assert!()` assertions et les traces de pile qu'elles générèrent en cas de panique sont généralement moins fiables..

Éditions

Rouillera des garanties de compatibilité extrêmement fortes. Tout code compilé sur Rust 1.0 doit tout aussi bien se compiler sur Rust 1.50 ou, s'il sort un jour, Rust 1.900.

Mais parfois, il existe des propositions convaincantes d'extensions du langage qui empêcheraient la compilation d'anciens codes. Par exemple, après de nombreuses discussions, Rust a opté pour une syntaxe pour la prise en charge de la programmation asynchrone qui réutilise les identifiants `async` et `await` comme mots-clés (voir le [chapitre 20](#)). Mais ce changement de langage casserait tout code existant qui utilise `async` ou `await` comme nom de variable.

Pour évoluer sans casser le code existant, Rust utilise les *éditions*. L'édition 2015 de Rust est compatible avec Rust 1.0. L'édition 2018 a changé `async` et `await` en mots-clés et rationalisé le système de modules, tandis que l'édition 2021 a amélioré l'ergonomie des tableaux et rendu certaines définitions de bibliothèques largement utilisées disponibles partout par défaut. Ce sont toutes des améliorations importantes du langage, mais qui auraient cassé le code existant. Pour éviter cela, chaque caisse indique dans quelle édition de Rust elle est écrite avec une ligne comme celle-ci dans la `[package]` section au-dessus de son fichier `Cargo.toml` :

```
édition = "2021"
```

Si ce mot-clé est absent, l'édition 2015 est supposée, donc les anciennes caisses n'ont pas à changer du tout. Mais si vous souhaitez utiliser les fonctions asynchrones ou le nouveau système de module, vous aurez besoin `edition = "2018"` de ou plus tard dans votre fichier `Cargo.toml`.

Rust promet que le compilateur acceptera toujours toutes les éditions existantes du langage, et les programmes peuvent librement mélanger des caisses écrites dans différentes éditions. C'est même bien qu'une caisse édition 2015 dépende d'une caisse édition 2021. En d'autres termes, l'édition d'un crate n'affecte que la manière dont son code source est interprété ; les distinctions d'édition ont disparu au moment où le code a été compilé. Cela signifie qu'il n'y a aucune pression pour mettre à jour les anciennes caisses juste pour continuer à participer à l'écosystème Rust moderne. De même, il n'y a aucune pression pour garder votre caisse sur une ancienne édition pour éviter de gêner ses utilisateurs. Vous n'avez besoin de changer d'édition que lorsque vous souhaitez utiliser de nouvelles fonctionnalités de langage dans votre propre code.

Les éditions ne sortent pas chaque année, uniquement lorsque le projet Rust décide qu'une est nécessaire. Par exemple, il n'y a pas d'édition 2020. Le réglage `edition` sur "2020" provoque une erreur. Le [Guide de l'édition Rust](#) couvre les changements introduits dans chaque édition et fournit de bonnes informations sur le système d'édition.

C'est presque toujours une bonne idée d'utiliser la dernière édition, en particulier pour le nouveau code. `cargo new` crée de nouveaux projets sur la dernière édition par défaut. Ce livre utilise l'édition 2021 tout au long.

Si vous avez une caisse écrite dans une ancienne édition de Rust, la `cargo fix` commande peut vous aider à mettre à jour automatiquement votre code vers la nouvelle édition. Le Guide Rust Edition explique la `cargo fix` commande en détail.

Créer des profils

Il existe plusieurs configurations paramètres que vous pouvez mettre dans votre fichier `Cargo.toml` qui affectent les `rustc` lignes de commande `cargo` générées ([Tableau 8-1](#)).

Ligne de commande	Section Cargo.toml utilisée
cargo build	[profile.dev]
cargo build --release	[profile.release]
cargo test	[profile.test]

Les valeurs par défaut sont généralement bonnes, mais une exception que nous avons trouvée est lorsque vous souhaitez utiliser un profileur—un outil qui mesure où votre programme passe son temps CPU. Pour obtenir les meilleures données d'un profileur, vous avez besoin à la fois d'optimisations (généralement activées uniquement dans les versions de version) et de symboles de débogage (généralement activés uniquement dans les versions de débogage). Pour activer les deux, ajoutez ceci à votre *Cargo.toml*:

```
[profil.release]
debug = true # activer les symboles de débogage dans les versions de version
```

Le `debug` paramètre contrôle l'`-g` option de `rustc`. Avec cette configuration, lorsque vous tapez `cargo build --release`, vous obtenez un binaire avec des symboles de débogage. Les paramètres d'optimisation ne sont pas affectés.

[La documentation Cargo](#) répertorie de nombreux autres paramètres que vous pouvez régler dans *Cargo.toml*.

Modules

Alors que les caisses concernent le partage de code entre les projets, les *modules* concernent l'organisation du code au sein d'un projet. Ils agissent comme des espaces de noms de Rust, des conteneurs pour les fonctions, les types, les constantes, etc. qui composent votre programme ou votre bibliothèque Rust. Un module ressemble à ceci :

```
mod spores {
    use cells::{Cell, Gene};

    /// A cell made by an adult fern. It disperses on the wind as part of
    /// the fern life cycle. A spore grows into a prothallus -- a whole
    /// separate organism, up to 5mm across -- which produces the zygote
    /// that grows into a new fern. (Plant sex is complicated.)
```

```

pub struct Spore {
    ...
}

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(factory: &mut Sporangium) ->Spore {
    ...
}

/// Extract the genes in a particular spore.
pub(crate) fn genes(spore: &Spore) ->Vec<Gene> {
    ...
}

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent:&mut Cell) {
    ...
}

...
}

```

Un module est une collection d' *éléments*, des fonctionnalités nommées comme la `Spore` structure et les deux fonctions dans cet exemple. Le `pub` mot-clé rend un élément public, il est donc accessible depuis l'extérieur du module.

Une fonction est marquée `pub(crate)` , ce qui signifie qu'elle est disponible n'importe où à l'intérieur de cette caisse, mais n'est pas exposée dans le cadre de l'interface externe. Il ne peut pas être utilisé par d'autres caisses et il n'apparaîtra pas dans la documentation de cette caisse.

Tout ce qui n'est pas marqué `pub` est privé et ne peut être utilisé que dans le même module dans lequel il est défini, ou dans n'importe quel module enfant :

```

let s = spores::produce_spore(&mut factory); // ok

spores::recombine(&mut cell); // error: `recombine` is private

```

Marquer un article comme on l' `pub` appelle souvent "exporter" cet article.

Le reste de cette section couvre les détails que vous devez connaître pour tirer pleinement parti des modules :

- Nous montrons comment imbriquer des modules et les distribuer dans différents fichiers et répertoires, si nécessaire.

- Nous expliquons la syntaxe de chemin que Rust utilise pour faire référence aux éléments d'autres modules et montrons comment importer des éléments afin que vous puissiez les utiliser sans avoir à écrire leurs chemins complets.
- Nous abordons le contrôle fin de Rust pour les champs de structure.
- Nous introduisons *le préludemodules*, qui réduisent le passe-partout en rassemblant les importations courantes dont presque tous les utilisateurs auront besoin.
- Nous présentons *des constanteset statique*, deux façons de définir des valeurs nommées, pour plus de clarté et de cohérence.

Modules imbriqués

Les modules peuvent s'imbriquer, et il est assez courant de voir un module qui n'est qu'une collection de sous-modules:

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

Si vous souhaitez qu'un élément d'un module imbriqué soit visible par d'autres crates, assurez-vous de le marquer, *ainsi que tous les modules englobants*, comme publics. Sinon, vous pourriez voir un avertissement comme celui-ci :

```
warning: function is never used: `is_square`
|
23 | /         pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |         }
| |_____^
|
```

Peut-être que cette fonction est vraiment du code mort pour le moment. Mais si vous vouliez l'utiliser dans d'autres caisses, Rust vous fait savoir qu'il n'est pas réellement visible pour eux. Vous devez vous assurer que ses modules englobants le sont `pub` également.

Il est également possible de spécifier `pub(super)`, rendant un élément visible uniquement au module parent, et `pub(in <path>)`, qui le rend visible dans un module parent spécifique et ses descendants. Ceci est particulièrement utile avec les modules profondément imbriqués :

```
mod plant_structures {
    pub mod roots {
        pub mod products {
            pub(in crate::: plant_structures::roots) struct Cytokinin {
                ...
            }
        }

        use products::Cytokinin; // ok: in `roots` module
    }

    use roots:: products::Cytokinin; // error: `Cytokinin` is private
}

// error: `Cytokinin` is private
use plant_structures:: roots:: products::Cytokinin;
```

De cette façon, nous pourrions écrire un programme entier, avec une énorme quantité de code et toute une hiérarchie de modules, liés de toutes les manières que nous voulions, le tout dans un seul fichier source.

En fait, travailler de cette façon est pénible, il existe donc une alternative.

Modules dans des fichiers séparés

Un module peut aussi s'écrire ainsi :

```
mod spores;
```

Auparavant, nous avons inclus le corps du `spores` module, entouré d'accolades. Ici, nous disons plutôt au compilateur Rust que le `spores` module vit dans un fichier séparé, appelé `spores.rs` :

```
// spores.rs

/// A cell made by an adult fern...
pub struct Spore {
    ...
}

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
```

```

    ...
}

/// Extract the genes in a particular spore.
pub(crate) fn genes(spore: &Spore) ->Vec<Gene> {
    ...
}

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent:&mut Cell) {
    ...
}

```

spores.rs contient uniquement les éléments qui composent le module. Il n'a besoin d'aucun type de passe-partout pour déclarer qu'il s'agit d'un module.

L'emplacement du code est la *seule* différence entre ce *spores* module et la version que nous avons montrée dans la section précédente. Les règles concernant ce qui est public et ce qui est privé sont exactement les mêmes dans les deux cas. Et Rust ne compile jamais les modules séparément, même s'ils sont dans des fichiers séparés : lorsque vous construisez un crate Rust, vous recompilez tous ses modules.

Un module peut avoir son propre répertoire. Lorsque Rust voit `mod spores;`, il recherche à la fois *spores.rs* et *spores/mod.rs* ; si aucun fichier n'existe, ou les deux existent, c'est une erreur. Pour cet exemple, nous avons utilisé *spores.rs*, car le *spores* module n'avait aucun sous-module. Mais considérez le *plant_structures* module que nous avons écrit plus tôt. Si nous décidons de scinder ce module et ses trois sous-modules dans leurs propres fichiers, le projet résultant ressemblera à ceci :

```

fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs

```

Dans *main.rs*, nous déclarons le *plant_structures* module :

```
pub mod plant_structures;
```

Cela amène Rust à charger *plant_structures/mod.rs* , qui déclare les trois sous- modules :

```
// in plant_structures/mod.rs
pub mod roots;
pub mod stems;
pub mod leaves;
```

Le contenu de ces trois modules est stocké dans des fichiers séparés nommés *leaves.rs* , *roots.rs* et *stems.rs* , situés à côté de *mod.rs* dans le répertoire *plant_structures* .

Il est également possible d'utiliser un fichier et un répertoire portant le même nom pour constituer un module. Par exemple, si *stems* nécessaire pour inclure des modules appelés *xylem* et *phloem* , nous pourrions choisir de conserver *stems* dans *plant_structures/stems.rs* et d'ajouter un répertoire *stems* :

```
fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    └── spores.rs
        └── plant_structures/
            ├── mod.rs
            ├── leaves.rs
            ├── roots.rs
            ├── stems/
            │   ├── phloem.rs
            │   └── xylem.rs
            └── stems.rs
```

Ensuite, dans *stems.rs* , nous déclarons les deux nouveaux sous-modules :

```
// in plant_structures/stems.rs
pub mod xylem;
pub mod phloem;
```

Ces trois options - modules dans leur propre fichier, modules dans leur propre répertoire avec un *mod.rs* , et modules dans leur propre fichier avec un répertoire supplémentaire contenant des sous-modules - donnent au système de modules suffisamment de flexibilité pour prendre en charge presque toutes les structures de projet que vous pourriez souhaiter.

Chemins et importations

L' `::` opérateur est utilisé pour accéder aux fonctionnalités d'un module. Le code n'importe où dans votre projet peut faire référence à n'importe quelle fonctionnalité de bibliothèque standard en écrivant son chemin :

```
if s1 > s2 {  
    std:: mem::swap(&mut s1, &mut s2);  
}
```

`std` est le nom de la norme bibliothèque. Le chemin `std` fait référence au module de niveau supérieur de la bibliothèque standard. `std::mem` est un sous-module dans la bibliothèque standard et `std::mem::swap` est une fonction publique dans ce module.

Vous pourriez écrire tout votre code de cette façon, en épelant

`std::f64::consts::PI` et à `std::collections::HashMap::new` chaque fois que vous voulez un cercle ou un dictionnaire, mais ce serait fastidieux à taper et difficile à lire. L'alternative est d'*importer* fonctionnalités dans les modules où elles sont utilisées :

```
use std::mem;  
  
if s1 > s2 {  
    mem::swap(&mut s1, &mut s2);  
}
```

La `use` déclaration fait du nom `mem` un alias local pour `std::mem` tout le bloc ou module englobant.

Nous pourrions écrire `use std::mem::swap;` pour importer la `swap` fonction elle-même au lieu du `mem` module. Cependant, ce que nous avons fait plus tôt est généralement considéré comme le meilleur style : importer des types, des traits et des modules (comme `std::mem`), puis utiliser des chemins relatifs pour accéder aux fonctions, constantes et autres membres qu'ils contiennent.

Plusieurs noms peuvent être importés à la fois :

```
use std:: collections::{HashMap, HashSet}; // import both  
  
use std:: fs::{self, File}; // import both `std::fs` and `std::fs::File`.  
  
use std:: io:: prelude::*;

// import everything
```

Ceci est juste un raccourci pour écrire toutes les importations individuelles :

```
use std:: collections:: HashMap;
use std:: collections:: HashSet;

use std:: fs;
use std:: fs:: File;

// all the public items in std::io::prelude:
use std:: io:: prelude:: Read;
use std:: io:: prelude:: Write;
use std:: io:: prelude:: BufRead;
use std:: io:: prelude:: Seek;
```

Vous pouvez utiliser `as` pour importer un élément mais lui donner un nom différent localement :

```
use std:: io::Result as IOResult;

// This return type is just another way to write `std::io::Result<()>`:
fn save_spore(spore: &Spore) ->IOResult<()>
...
...
```

Les modules n'héritent *pas* automatiquement des noms de leurs modules parents. Par exemple, supposons que nous ayons ceci dans notre *proteines/mod.rs* :

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

Ensuite, le code dans *synthesis.rs* ne voit pas automatiquement le type `AminoAcid`:

```
// proteins/synthesis.rs
pub fn synthesize(seq:&[AminoAcid]) // error: can't find type `AminoAcid`
...
...
```

Au lieu de cela, chaque module commence par une ardoise vierge et doit importer les noms qu'il utilise :

```
// proteins/synthesis.rs
use super::AminoAcid; // explicitly import from parent
```

```
pub fn synthesize(seq:&[AminoAcid]) // ok
    ...

```

Par défaut, les chemins sont relatifs au module courant :

```
// in proteins/mod.rs

// import from a submodule
use synthesis::synthesize;
```

`self` est aussi un synonyme du module courant, on pourrait donc écrire soit :

```
// in proteins/mod.rs

// import names from an enum,
// so we can write `Lys` for lysine, rather than `AminoAcid::Lys`
use self:: AminoAcid::*;


```

ou simplement:

```
// in proteins/mod.rs

use AminoAcid::*;


```

(L' `AminoAcid` exemple ici est, bien sûr, un écart par rapport à la règle de style que nous avons mentionnée précédemment concernant l'importation de types, de traits et de modules uniquement. Si notre programme comprend de longues séquences d'acides aminés, cela est justifié par la sixième règle d'Orwell : "Briser l'un de ces règles plutôt que de dire quoi que ce soit de carrément barbare. »)

Les mots-clés `super` et `crate` ont une signification particulière dans les chemins : `super` fait référence au module parent, et `crate` fait référence au crate contenant le module courant.

L'utilisation de chemins relatifs à la racine du crate plutôt qu'au module actuel facilite le déplacement du code dans le projet, car toutes les importations ne seront pas interrompues si le chemin du module actuel change. Par exemple, nous pourrions écrire `synthesis.rs` en utilisant `crate` :

```
// proteins/synthesis.rs
use crate:: proteins::AminoAcid; // explicitly import relative to crate root
```

```
pub fn synthesize(seq:&[AminoAcid]) // ok
    ...

```

Les sous-modules peuvent accéder aux éléments privés de leurs modules parents avec `use super::*`.

Si vous avez un module portant le même nom qu'une caisse que vous utilisez, il faut être prudent en se référant à son contenu. Par exemple, si votre programme répertorie la `image` caisse comme une dépendance dans son fichier `Cargo.toml`, mais a également un module nommé `image`, alors les chemins commençant par `image` sont ambigus :

```
mod image {
    pub struct Sampler {
        ...
    }
}

// error: Does this refer to our `image` module, or the `image` crate?
use image::Pixels;
```

Même si le `image` module n'a pas de `Pixels` type, l'ambiguïté est toujours considérée comme une erreur : ce serait déroutant si l'ajout ultérieur d'une telle définition pouvait modifier silencieusement les chemins auxquels se réfèrent ailleurs dans le programme.

Pour résoudre l'ambiguïté, Rust a un type spécial de chemin appelé *chemin absolu*, commençant par `::`, qui fait toujours référence à une caisse externe. Pour faire référence au `Pixels` type dans la `image` caisse, vous pouvez écrire :

```
use ::image::Pixels;           // the `image` crate's `Pixels`
```

Pour faire référence au `Sampler` type de votre propre module, vous pouvez écrire :

```
use self::image::Sampler;     // the `image` module's `Sampler`
```

Les modules ne sont pas la même chose que les fichiers, mais il existe une analogie naturelle entre les modules et les fichiers et répertoires d'un système de fichiers Unix. Le `use` mot clé crée des alias, tout comme la `ln` commande crée des liens. Les chemins, comme les noms de fichiers, se présentent sous des formes absolues et relatives. `self` et `super` sont comme les répertoires `.` et `..` spéciaux.

Le prélude standard

Nous avons dit tout à l'heure que chaque module commence par une « ardoise vierge », en ce qui concerne les noms importés. Mais l'ardoise n'est pas *complètement* vierge.

D'une part, la bibliothèque standard `std` est automatiquement liée à chaque projet. Cela signifie que vous pouvez toujours utiliser `use std::whatever` ou faire référence à `std` des éléments par leur nom, comme `std::mem::swap()` en ligne dans votre code. De plus, quelques noms particulièrement pratiques, comme `Vec` et `Result`, sont inclus dans le *prélude standard* et automatiquement importés. Rust se comporte comme si chaque module, y compris le module racine, avait démarré avec l'import suivant :

```
use std:: prelude:: v1::*;


```

Le prélude standard contient quelques dizaines de traits et de types couramment utilisés.

Au [chapitre 2](#), nous avons mentionné que les bibliothèques fournissent parfois des modules nommés `prelude`. Mais `std::prelude::v1` c'est le seul prélude jamais importé automatiquement. Nommer un module `prelude` n'est qu'une convention qui indique aux utilisateurs qu'il est censé être importé à l'aide de `*`.

Faire usage des déclarations pub

Même si `use` les déclarationsne sont que des pseudonymes, ils peuvent être publics :

```
// in plant_structures/mod.rs
...
pub use self:: leaves:: Leaf;
pub use self:: roots:: Root;
```

Cela signifie que `Leaf` et `Root` sont des éléments publics du `plant_structures` module. Ce sont encore de simples alias pour `plant_structures::leaves::Leaf` et `plant_structures::roots::Root`.

Le prélude standard est écrit comme une telle série d' `pub` importations.

Création d'un pub Structu Fields

Un module peut inclure des structuretypes, introduits à l'aide du mot-`struct` clé. Nous les couvrons en détail au [chapitre 9](#), mais c'est un bon point pour mentionner comment les modules interagissent avec la visibilité des champs struct.

Une structure simple ressemble à ceci :

```
pub struct Fern {
    pub roots: RootSet,
    pub stems: StemSet
}
```

Les champs d'une structure, même les champs privés, sont accessibles dans tout le module où la structure est déclarée, et ses sous-modules. En dehors du module, seuls les champs publics sont accessibles.

Il s'avère que l'application du contrôle d'accès par module, plutôt que par classe comme en Java ou C++, est étonnamment utile pour la conception de logiciels. Il réduit les méthodes passe-partout « getter » et « setter », et il élimine en grande partie le besoin de quelque chose comme les `friend` déclarations C++. Un seul module peut définir plusieurs types qui fonctionnent étroitement ensemble, comme peut-être `frond::LeafMap` et `frond::LeafMapIter`, accédant aux champs privés de l'autre selon les besoins, tout en cachant ces détails d'implémentation au reste de votre programme.

Statique et constantes

En plus des fonctions, des types et des modules imbriqués, les modules peuvent également définir *des constantes et statique*.

Le `const` mot cléintroduit une constante. La syntaxe est la même `let` sauf qu'elle peut être marquée `pub`, et le type est obligatoire. De plus, `UPPERCASE_NAMES` sont conventionnels pour les constantes :

```
pub const ROOM_TEMPERATURE:f64 = 20.0; // degrees Celsius
```

Le `static` mot cléintroduit un élément statique, qui est presque la même chose :

```
pub static ROOM_TEMPERATURE:f64 = 68.0; // degrees Fahrenheit
```

Une constante est un peu comme un C++ `#define` : la valeur est compilée dans votre code à chaque endroit où elle est utilisée. Un statique est une

variable configurée avant le démarrage de votre programme et qui dure jusqu'à sa fermeture. Utilisez des constantes pour les nombres magiques et les chaînes dans votre code. Utilisez la statique pour de plus grandes quantités de données ou chaque fois que vous avez besoin d'emprunter une référence à la valeur constante.

Il n'y a pas de `mut` constantes. Statique peut être marqué `mut`, mais comme indiqué au [chapitre 5](#), Rust n'a aucun moyen d'appliquer ses règles concernant l'accès exclusif aux `mut` statiques. Ils sont donc intrinsèquement non-thread-safe, et le code sécurisé ne peut pas du tout les utiliser :

```
static mut PACKETS_SERVED:usize = 0;

println!("{} served", PACKETS_SERVED); // error: use of mutable static
```

La rouille décourage l'état mutable global. Pour une discussion des alternatives, voir [« Variables globales »](#).

Transformer un programme en bibliothèque

Comme votre simulateur de fougère recommence à décoller, vous décidez que vous avez besoin de plus d'un programme. Supposons que vous disposez d'un programme en ligne de commande qui exécute la simulation et enregistre les résultats dans un fichier. Maintenant, vous voulez écrire d'autres programmes pour effectuer une analyse scientifique des résultats enregistrés, afficher des rendus 3D des plantes en croissance en temps réel, rendre des images photoréalistes, etc. Tous ces programmes doivent partager le code de base de simulation de fougère. Vous devez créer une bibliothèque.

La première étape consiste à diviser votre projet existant en deux parties : une caisse de bibliothèque, qui contient tout le code partagé, et un exécutable, qui contient le code qui n'est nécessaire que pour votre programme de ligne de commande existant.

Pour montrer comment vous pouvez faire cela, utilisons un exemple de programme grossièrement simplifié :

```
struct Fern {
    size: f64,
    growth_rate:f64
}
```

```

impl Fern {
    /// Simulate a fern growing for one day.
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Run a fern simulation for some number of days.
fn run_simulation(fern: &mut Fern, days:usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate:0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}

```

Nous supposerons que ce programme a un fichier *Cargo.toml* trivial :

```

[forfait]
nom = "fougère_sim"
version = "0.1.0"
auteurs = ["Vous <vous@example.com>"]
édition = "2021"

```

Transformer ce programme en une bibliothèque est facile. Voici les étapes :

1. Renommez le fichier *src/main.rs* en *src/lib.rs* .
2. Ajoutez le mot- `pub` clé aux éléments de *src/lib.rs* qui seront des fonctionnalités publiques de notre bibliothèque.
3. Déplacez la `main` fonction vers un fichier temporaire quelque part.
Nous y reviendrons dans une minute.

Le fichier *src/lib.rs* résultant ressemble à ceci :

```

pub struct Fern {
    pub size: f64,
    pub growth_rate:f64
}

impl Fern {

```

```

    /// Simulate a fern growing for one day.
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }

    /// Run a fern simulation for some number of days.
    pub fn run_simulation(fern: &mut Fern, days: usize) {
        for _ in 0 .. days {
            fern.grow();
        }
    }
}

```

Notez que nous n'avons rien eu à changer dans *Cargo.toml*. En effet, notre fichier *Cargo.toml* minimal laisse Cargo à son comportement par défaut. Par défaut, `cargo build` regarde les fichiers dans notre répertoire source et détermine ce qu'il faut construire. Quand il voit le fichier *src/lib.rs*, il sait construire une bibliothèque.

Le code dans *src/lib.rs* forme le *module racine* de la bibliothèque. Les autres caisses qui utilisent notre bibliothèque ne peuvent accéder qu'aux éléments publics de ce module racine.

Le répertoire src/bin

Obtenir l'originalle programme en ligne de commande `fern_sim` fonctionne à nouveau est également simple: Cargo a un support intégré pour les petits programmes qui vivent dans la même caisse qu'une bibliothèque.

En fait, Cargo lui-même est écrit de cette façon. La majeure partie du code se trouve dans une bibliothèque Rust. Le `cargo` programme de ligne de commande que nous avons utilisé tout au long de ce livre est un programme d'encapsulation léger qui appelle la bibliothèque pour tout le travail lourd. La bibliothèque et le programme de ligne de commande [vivent dans le même référentiel source](#).

Nous pouvons également conserver notre programme et notre bibliothèque dans la même caisse. Mettez ce code dans un fichier nommé *src/bin/fern.rs*:

```

use fern_sim::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,

```

```

        growth_rate:0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}

```

La `main` fonction est celle que nous avons mise de côté plus tôt. Nous avons ajouté une `use` déclaration pour certains articles de la `fern_sim` caisse, `Fern` et `run_simulation`. En d'autres termes, nous utilisons cette caisse comme une bibliothèque.

Parce que nous avons mis ce fichier dans `src/bin`, Cargo compilera à la fois la `fern_sim` bibliothèque et ce programme la prochaine fois que nous exécuterons `cargo build`. Nous pouvons exécuter le `efern` programme en utilisant `cargo run --bin efern`. Voici à quoi cela ressemble, en utilisant `--verbose` pour afficher les commandes que Cargo exécute :

```

$cargo build --verbose
Compiling fern_sim v0.1.0 (file:///.../fern_sim)
  Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...`
  Running `rustc src/bin/efern.rs --crate-name efern --crate-type bin ...`
$cargo run --bin efern --verbose
  Fresh fern_sim v0.1.0 (file:///.../fern_sim)
  Running `target/debug/efern`
final fern size: 2.7169239322355985

```

Nous n'avons toujours pas eu à apporter de modifications à `Cargo.toml`, car, encore une fois, la valeur par défaut de Cargo est de regarder vos fichiers source et de comprendre les choses. Il traite automatiquement les fichiers `.rs` dans `src/bin` comme des programmes supplémentaires à construire.

Nous pouvons également créer des programmes plus volumineux dans le répertoire `src/bin` en utilisant des sous-répertoires. Supposons que nous souhaitions fournir un deuxième programme qui dessine une fougère à l'écran, mais que le code de dessin est volumineux et modulaire, il appartient donc à son propre fichier. Nous pouvons donner au deuxième programme son propre sous-répertoire :

```

fern_sim/
├── Cargo.toml
└── src/
    └── bin/
        ├── efern.rs
        └── draw_fern/

```

```
└── main.rs  
└── draw.rs
```

Cela a l'avantage de laisser des binaires plus grands avoir leurs propres sous-modules sans encombrer ni le code de la bibliothèque ni le répertoire `src/bin`.

Bien sûr, maintenant que `fern_sim` c'est une bibliothèque, nous avons aussi une autre option. Nous aurions pu mettre ce programme dans son propre projet isolé, dans un répertoire complètement séparé, avec sa propre liste `Cargo.toml` `fern_sim` comme dépendance :

```
[dépendances]  
fougère_sim = { chemin = "../fougère_sim" }
```

C'est peut-être ce que vous ferez pour d'autres programmes de simulation de fougères sur la route. Le répertoire `src/bin` est parfait pour des programmes simples comme `efern` et `draw_fern`.

Les attributs

N'importe quel article dans un programme Rust peut être décoré avec des *attributs*. Les attributs sont la syntaxe fourre-tout de Rust pour écrire diverses instructions et conseils au compilateur. Par exemple, supposons que vous receviez cet avertissement :

```
libgit2.rs: warning: type `git_revspeс` should have a camel case name  
such as `GitRevspeс`, #[warn(non_camel_case_types)] on by default
```

Mais vous avez choisi ce nom pour une raison, et vous souhaiteriez que Rust se taise à ce sujet. Vous pouvez désactiver l'avertissement en ajoutant un `#[allow]` attribut sur le genre :

```
#[allow(non_camel_case_types)]  
pub struct git_revspeс {  
    ...  
}
```

La compilation conditionnelle est une autre fonctionnalité écrite à l'aide d'un attribut, à savoir `#[cfg]` :

```
// Only include this module in the project if we're building for Android.  
#[cfg(target_os = "android")]  
mod mobile;
```

La syntaxe complète de `#[cfg]` est spécifiée dans la [Rust Reference](#) ; les options les plus couramment utilisées sont répertoriées dans le [Tableau 8-2](#).

Tableau 8-2. `#[cfg]` Options les plus couramment utilisées

<code>#[cfg</code>	Activé lorsque
<code>(...)] option</code>	
<code>test</code>	Les tests sont activés (compilation avec <code>cargo test</code> ou <code>rustc --test</code>).
<code>debug_</code>	Les assertions de débogage sont activées (généralement dans les versions non optimisées).
<code>assert</code>	
<code>ions</code>	
<code>unix</code>	Compilation pour Unix, y compris macOS.
<code>window</code>	Compilation pour Windows.
<code>s</code>	
<code>target</code>	Cibler une plate-forme 64 bits. L'autre valeur possible est
<code>_pointe</code>	"32".
<code>r_width</code>	
<code>= "64"</code>	
<code>target</code>	Ciblant x86-64 en particulier. Autres valeurs : "x86",
<code>_arch =</code>	"arm", "aarch64", "powerpc", "powerpc64", "mip
<code>"x86_6</code>	s".
<code>4"</code>	
<code>target</code>	Compilation pour macOS. Autres valeurs : "windows",
<code>_os =</code>	"ios", "android", "linux", "freebsd", "openbs
<code>"maco</code>	d", "netbsd", "dragonfly".
<code>s"</code>	
<code>featur</code>	La fonctionnalité définie par l'utilisateur nommée "rob
<code>e = "ro</code>	ots" est activée (compilation avec <code>cargo build --fea</code>
<code>bots"</code>	ture robots ou <code>rustc --cfg feature='robots'</code>).
	Les fonctionnalités sont déclarées dans la
	<u>[features]. section Cargo.toml</u> .
<code>not(U</code>	A n'est pas satisfait. Pour fournir deux implémentations
<code>N)</code>	différentes d'une fonction, marquez l'une avec <code>#[cfg</code>
	<code>(X)]</code> et l'autre avec <code>#[cfg(not(X))]</code> .
<code>all(Un</code>	A et B sont satisfais (l'équivalent de <code>&&</code>).
<code>, B)</code>	

```
#cfg  
Activé lorsque  
(...) option
```

```
any( Un A ou B est satisfait (l'équivalent de || ).  
, B )
```

Parfois, nous devons microgérer l'expansion en ligne des fonctions, une optimisation que nous sommes généralement heureux de laisser au compilateur. Nous pouvons utiliser l' `#[inline]` attribut pour ça:

```
// Adjust levels of ions etc. in two adjacent cells  
// due to osmosis between them.  
#[inline]  
fn do_osmosis(c1: &mut Cell, c2:&mut Cell) {  
    ...  
}
```

Il y a une situation où l'inlining ne se produira *pas* sans `#[inline]`. Lorsqu'une fonction ou une méthode définie dans un crate est appelée dans un autre crate, Rust ne l'inline pas à moins qu'elle ne soit générique (elle a des paramètres de type) ou qu'elle soit explicitement marquée `#[inline]`.

Sinon, le compilateur traite `#[inline]` comme une suggestion. Rust prend également en charge le plus insistant `#[inline(always)]`, pour demander qu'une fonction soit développée en ligne sur chaque site d'appel, et `#[inline(never)]`, pour demander qu'une fonction ne soit jamais en ligne.

Certains attributs, comme `#[cfg]` et `#[allow]`, peuvent être attachés à un module entier et s'appliquer à tout ce qu'il contient. Les autres, comme `#[test]` et `#[inline]`, doivent être attachés à des éléments individuels. Comme on peut s'y attendre pour une fonctionnalité fourre-tout, chaque attribut est personnalisé et possède son propre ensemble d'arguments pris en charge. La référence Rust documente en détail [l'ensemble complet des attributs pris en charge](#).

Pour attacher un attribut à une caisse entière, ajoutez-le en haut du fichier `main.rs` ou `lib.rs`, avant tout élément, et écrivez à la `#!` place de `#`, comme ceci :

```
// libgit2_sys/lib.rs  
#![allow(non_camel_case_types)]  
  
pub struct git_revspec {
```

```

    ...
}

pub struct git_error {
    ...
}

```

Le `#!` dit à Rust d'attacher un attribut à l'élément englobant plutôt qu'à ce qui vient ensuite : dans ce cas, l'`#![allow]` attribut s'attache à l'ensemble de la `libgit2_sys` caisse, pas seulement `struct git_revspec`.

`#!` peut également être utilisé dans des fonctions, des structures, etc., mais il n'est généralement utilisé qu'au début d'un fichier, pour attacher un attribut à l'ensemble du module ou de la caisse. Certains attributs utilisent toujours la `#!` syntaxe car ils ne peuvent être appliqués qu'à un tout entier.

Par exemple, l'`#![feature]` attribut est utilisé pour activer les fonctionnalités *instables* du langage et des bibliothèques Rust, des fonctionnalités qui sont expérimentales et qui peuvent donc avoir des bogues ou être modifiées ou supprimées à l'avenir. Par exemple, au moment où nous écrivons ceci, Rust a un support expérimental pour tracer l'expansion des macros comme `assert!`, mais comme ce support est expérimental, vous ne pouvez l'utiliser qu'en (1) installant la version nocturne de Rust et (2) en déclarant explicitement que votre caisse utilise le traçage de macro :

```

#[feature(trace_macros)]

fn main() {
    // I wonder what actual Rust code this use of assert_eq!
    // gets replaced with!
    trace_macros!(true);
    assert_eq!(10*10*10 + 9*9*9, 12*12*12 + 1*1*1);
    trace_macros!(false);
}

```

Au fil du temps, l'équipe Rust *stabilise* parfois une fonctionnalité expérimentale afin qu'elle devienne une partie standard du langage. L'`#![feature]` attribut devient alors superflu, et Rust génère un avertissement vous conseillant de le supprimer.

Essais et documentation

Comme nous l'avons vu dans "[Writing and Running Unit Tests](#)" , un test unitaire simpleframework est intégré à Rust. Les tests sont des fonctions ordinaires marquées de l' `#[test]` attribut :

```
#[test]
fn math_works() {
    let x:i32 = 1;
    assert!(x.is_positive());
    assert_eq!(x + 1, 2);
}
```

`cargo test` court tous les tests de votre projet :

```
$ cargo test
Compiling math_test v0.1.0 (file:///.../math_test)
Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

(Vous verrez également des sorties sur les "doc-tests", auxquelles nous reviendrons dans une minute.)

Cela fonctionne de la même manière que votre crate soit un exécutable ou une bibliothèque. Vous pouvez exécuter des tests spécifiques en passant des arguments à Cargo : `cargo test math` exécute tous les tests qui contiennent `math` quelque part dans leur nom.

Les tests utilisent couramment les macros `assert!` et `assert_eq!` de la bibliothèque standard Rust. `assert!(expr)` réussit si `expr` est vrai. Si non, il panique, ce qui fait échouer le test. `assert_eq!(v1, v2)` est identique `assert!(v1 == v2)` sauf que si l'assertion échoue, le message d'erreur affiche les deux valeurs.

Vous pouvez utiliser ces macros dans du code ordinaire pour vérifier les invariants, mais notez que `assert!` et `assert_eq!` sont inclus même dans les versions de version. Utilisez `debug_assert!` et `debug_assert_eq!` à la place pour écrire des assertions qui ne sont vérifiées que dans les versions de débogage.

Pour tester les cas d'erreur, ajoutez l' `#[should_panic]` attribut à ton essai :

```

/// This test passes only if division by zero causes a panic,
/// as we claimed in the previous chapter.
#[test]
#[allow(unconditional_panic, unused_must_use)]
#[should_panic(expected="divide by zero")]
fn test_divide_by_zero_error() {
    1 / 0; // should panic!
}

```

Dans ce cas, nous devons également ajouter un `allow` attribut pour dire au compilateur de nous laisser faire des choses dont il peut prouver statiquement qu'elles vont paniquer, et effectuer des divisions et simplement jeter la réponse, car normalement, il essaie d'arrêter ce genre de bêtises.

Vous pouvez également renvoyer un `Result<(), E>` de vos tests. Tant que la variante d'erreur est `Debug`, ce qui est généralement le cas, vous pouvez simplement renvoyer à `Result` en utilisant `?` pour supprimer la `Ok` variante :

```

use std::num::ParseIntError;

/// This test will pass if "1024" is a valid number, which it is.
#[test]
fn explicit_radix() -> Result<(), ParseIntError> {
    i32::from_str_radix("1024", 10)?;
    Ok(())
}

```

Les fonctions marquées d'un `#[test]` sont compilées conditionnellement. Un simple `cargo build` ou `cargo build --release` saute le code de test. Mais lorsque vous exécutez `cargo test`, Cargo construit votre programme deux fois : une fois de manière ordinaire et une fois avec vos tests et le harnais de test activé. Cela signifie que vos tests unitaires peuvent coexister avec le code qu'ils testent, en accédant aux détails d'implémentation internes s'ils en ont besoin, et pourtant il n'y a aucun coût d'exécution. Cependant, cela peut entraîner certains avertissements. Par exemple :

```

fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}

#[test]
fn trig_works() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}

```

Dans les versions qui omettent le code de test, `roughly_equal` apparaît inutilisé et Rust se plaindra :

```
$construction de cargaison
    Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
|
7 | / fn roughly_equal(a: f64, b: f64) -> bool {
8 | |     (a - b).abs() < 1e-6
9 | |
| |_-
|
= note: #[warn(dead_code)] on by default
```

Ainsi, la convention, lorsque vos tests deviennent suffisamment substantiels pour nécessiter du code de support, est de les mettre dans un `tests` module et de déclarer l'ensemble du module comme test uniquement en utilisant l' `#[cfg]` attribut :

```
#[cfg(test)] // include this module only when testing
mod tests {
    fn roughly_equal(a: f64, b: f64) ->bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        use std::f64::consts::PI;
        assert!(roughly_equal(PI.sin(), 0.0));
    }
}
```

Le harnais de test de Rust utilise plusieurs threads pour exécuter plusieurs tests à la fois, un avantage secondaire intéressant du fait que votre code Rust est thread-safe par défaut. Pour désactiver cela, exécutez un seul test, ou exécutez . (Le premier garantit que passe l' option à l'exécutable de test.) Cela signifie que, techniquement, le programme Mandelbrot que nous avons montré au [chapitre 2](#) n'était pas le deuxième programme multithread de ce chapitre, mais le troisième ! L'exécution de "[Writing and Running Unit Tests](#)" a été la première `cargo test`

```
testname cargo test -- --test-threads 1 -- cargo test --
test-threads cargo test.
```

Normalement, le faisceau de test affiche uniquement la sortie des tests qui ont échoué. Pour afficher la sortie des tests qui réussissent également, exécutez `cargo test -- --no-capture`.

Essais d'intégration

Votre simulateur de fougère continue à grandir. Vous avez décidé de mettre toutes les fonctionnalités principales dans une bibliothèque qui peut être utilisée par plusieurs exécutables. Ce serait bien d'avoir des tests liés à la bibliothèque comme le ferait un utilisateur final, en utilisant `fern_sim.rlib` comme caisse externe. De plus, vous avez des tests qui commencent par charger une simulation enregistrée à partir d'un fichier binaire, et il est gênant d'avoir ces gros fichiers de test dans votre répertoire `src`. Les tests d'intégration aident à résoudre ces deux problèmes.

Les tests d'intégration sont des fichiers `.rs` qui résident dans un répertoire de tests à côté du répertoire `src` de votre projet. Lorsque vous exécutez `cargo test`, Cargo compile chaque test d'intégration en tant que caisse distincte et autonome, liée à votre bibliothèque et au harnais de test Rust. Voici un exemple:

```
// tests/unfurl.rs - Fiddleheads unfurl in sunlight

use fern_sim:: Terrarium;
use std:: time::Duration;

#[test]
fn test_fiddlehead_unfurling() {
    let mut world = Terrarium:: load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
```

Les tests d'intégration sont précieux en partie parce qu'ils voient votre caisse de l'extérieur, tout comme le ferait un utilisateur. Ils testent l'API publique de la caisse.

`cargo test` exécute à la fois des tests unitaires et des tests d'intégration. Pour exécuter uniquement les tests d'intégration dans un fichier particulier, par exemple, `tests/unfurl.rs`, utilisez la commande `cargo test --test unfurl`.

Documentation

La commande `cargo doc` crée des documents HTML pour votre bibliothèque :

```
$document de fret --no-deps --open  
Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

L' `--no-deps` option indique à Cargo de générer une documentation uniquement pour `fern_sim` lui-même, et non pour toutes les caisses dont il dépend.

L' `--open` option indique à Cargo d'ouvrir ensuite la documentation dans votre navigateur.

Vous pouvez voir le résultat dans la [Figure 8-2](#). Cargo enregistre les nouveaux fichiers de documentation dans `target/doc`. La page de démarrage est `target/doc/fern_sim/index.html`.

Click or press 'S' to search, '?' for more options...

Crate `fern_sim`

[[-](#)] [[src](#)]

[[-](#)] Simulate the growth of ferns, from the level of individual cells on up.

Reexports

```
pub use plant_structures::Fern;  
pub use simulation::Terrarium;
```

Modules

<code>cells</code>	The simulation of biological cells, which is as low-level as we go.
<code>plant_structures</code>	Higher-level biological structures.
<code>simulation</code>	Overall simulation control.
<code>spores</code>	Fern reproduction.

Illustration 8-2. Exemple de documentation générée par `rustdoc`

La documentation est générée à partir des `pub` fonctionnalités de votre bibliothèque, ainsi que des *commentaires de documentation* vous y êtes attaché. Nous avons déjà vu quelques commentaires de doc dans ce chapitre. Ils ressemblent à des commentaires :

```
// Simulate the production of a spore by meiosis.  
pub fn produce_spore(factory: &mut Sporangium) ->Spore {  
    ...  
}
```

Mais lorsque Rust voit des commentaires commençant par trois barres obliques, il les traite `#[doc]` plutôt comme un attribut. Rust traite l'exemple précédent exactement de la même manière :

```
#[doc = "Simulate the production of a spore by meiosis."]  
pub fn produce_spore(factory: &mut Sporangium) ->Spore {  
    ...  
}
```

Lorsque vous compilez une bibliothèque ou un binaire, ces attributs ne changent rien, mais lorsque vous générez de la documentation, les commentaires de la documentation sur les fonctionnalités publiques sont inclus dans la sortie.

De même, les commentaires commençant par `//!` sont traités comme des `#![doc]` attributs et sont attachés à la fonction englobante, généralement un module ou une caisse. Par exemple, votre fichier `fern_sim/src/lib.rs` pourrait commencer ainsi :

```
//! Simulate the growth of ferns, from the level of  
//! individual cells on up.
```

Le contenu d'un commentaire de document est traité comme Markdown, une notation abrégée pour un formatage HTML simple. Les astérisques sont utilisés pour `*italics*` et `**bold type**`, une ligne vide est traitée comme un saut de paragraphe, et ainsi de suite. Vous pouvez également inclure des balises HTML, qui sont copiées textuellement dans la documentation formatée.

Une particularité des commentaires de documentation dans Rust est que les liens Markdown peuvent utiliser des chemins d'accès aux éléments Rust, comme `leaves::Leaf`, au lieu d'URL relatives, pour indiquer à quoi ils se réfèrent. Cargo recherchera à quoi le chemin fait référence et substituera un lien au bon endroit dans la bonne page de documentation. Par exemple, la documentation générée à partir de ce code renvoie aux pages de documentation pour `VascularPath`, `Leaf` et `Root`:

```
/// Create and return a [`VascularPath`] which represents the path of  
/// nutrients from the given [`Root`][r] to the given [`Leaf`][leaves::Leaf]  
///
```

```
/// [r]: roots::Root
pub fn trace_path(leaf: &leaves::Leaf, root: &roots::Root) ->VascularPath
...
}
```

Vous pouvez également ajouter des alias de recherche pour faciliter la recherche d'éléments à l'aide de la fonction de recherche intégrée. La recherche de "chemin" ou "route" dans la documentation de cette caisse conduira à `VascularPath` :

```
#[doc(alias = "route")]
pub struct VascularPath {
    ...
}
```

Pour des blocs de documentation plus longs ou pour rationaliser votre flux de travail, vous pouvez inclure des fichiers externes dans votre documentation. Par exemple, si le fichier `README.md` de votre référentiel contient le même texte que vous souhaitez utiliser comme documentation de niveau supérieur de votre caisse, vous pouvez le mettre en haut de `lib.rs` ou `main.rs`:

```
#![doc = include_str!("../README.md")]
```

Vous pouvez utiliser `backticks` pour déclencher des morceaux de code au milieu d'un texte en cours d'exécution. Dans la sortie, ces extraits seront formatés dans une police à largeur fixe. Des exemples de code plus grands peuvent être ajoutés en indentant quatre espaces :

```
/// A block of code in a doc comment:
///
///     if samples::everything().works() {
///         println!("ok");
///     }
```

Vous pouvez également utiliser des blocs de code délimités par Markdown. Cela a exactement le même effet :

```
/// Another snippet, the same code, but written differently:
///
/// ``
/// if samples::everything().works() {
///     println!("ok");
/// }
/// ``
```

Quel que soit le format que vous utilisez, une chose intéressante se produit lorsque vous incluez un bloc de code dans un commentaire de doc. Rust le transforme automatiquement en test.

Doc-Tests

Quand tu courstests dans une caisse de bibliothèque Rust, Rust vérifie que tout le code qui apparaît dans votre documentation s'exécute et fonctionne réellement. Pour ce faire, il prend chaque bloc de code qui apparaît dans un commentaire de documentation, le compile en tant que caisse exécutable distincte, le relie à votre bibliothèque et l'exécute.

Voici un exemple autonome de doc-test. Créez un nouveau projet en exécutant `cargo new --lib ranges` (le `--lib` drapeau indique à Cargo que nous créons une caisse de bibliothèque, pas une caisse exécutable) et placez le code suivant dans `ranges/src/lib.rs` :

```
use std::ops::Range;

/// Return true if two ranges overlap.
///
///     assert_eq!(ranges::overlap(0..7, 3..10), true);
///     assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// If either range is empty, they don't count as overlapping.
///
///     assert_eq!(ranges::overlap(0..0, 0..10), false);
///

pub fn overlap(r1: Range<usize>, r2: Range<usize>) ->bool {
    r1.start < r1.end && r2.start < r2.end &&
    r1.start < r2.end && r2.start < r1.end
}
```

Les deux petits blocs de code dans le commentaire doc apparaissent dans la documentation générée par `cargo doc`, comme illustré à la [figure 8-3](#).

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

[-] Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

Illustration 8-3. Documentation montrant quelques doc-tests

Ils deviennent également deux tests distincts :

```
$cargaisonstest
Compiling ranges v0.1.0 (file:///.../ranges)
...
Doc-tests ranges

running 2 tests
test overlap_0 ... ok
test overlap_1 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Si vous passez le `--verbose` drapeau à Cargo, vous verrez qu'il est utilisé `rustdoc --test` pour exécuter ces deux tests. `rustdoc` stocke chaque échantillon de code dans un fichier séparé, en ajoutant quelques lignes de code passe-partout, pour produire deux programmes. Voici le premier :

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
    assert_eq!(ranges::overlap(1..5, 101..105), false);
}
```

Et voici le deuxième :

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..0, 0..10), false);
}
```

Les tests réussissent si ces programmes se compilent et s'exécutent correctement.

Ces deux exemples de code contiennent des assertions, mais c'est simplement parce que dans ce cas, les assertions constituent une documentation décente. L'idée derrière les doc-tests n'est pas de mettre tous vos tests en commentaires. Au lieu de cela, vous écrivez la meilleure documentation possible, et Rust s'assure que les exemples de code de votre documentation se compilent et s'exécutent réellement.

Très souvent, un exemple de travail minimal inclut certains détails, tels que les importations ou le code de configuration, qui sont nécessaires pour faire compiler le code, mais qui ne sont tout simplement pas assez importants pour être affichés dans la documentation. Pour masquer une ligne d'un exemple de code, placez un `#` suivi d'un espace au début de cette ligne :

```
// Let the sun shine in and run the simulation for a given
// amount of time.
//
// # use fern_sim::Terrarium;
// # use std::time::Duration;
// # let mut tm = Terrarium::new();
// tm.apply_sunlight(Duration::from_secs(60));
//
pub fn apply_sunlight(&mut self, time:Duration) {
    ...
}
```

Parfois, il est utile de montrer un exemple de programme complet dans la documentation, y compris une `main` fonction. Évidemment, si ces morceaux de code apparaissent dans votre exemple de code, vous ne voulez pas non plus `rustdoc` les ajouter automatiquement. Le résultat ne serait pas compilé. `rustdoc` traite donc tout bloc de code contenant la chaîne exacte `fn main` comme un programme complet et n'y ajoute rien .

Les tests peuvent être désactivés pour des blocs de code spécifiques. Pour dire à Rust de compiler votre exemple, mais de ne pas l'exécuter réellement, utilisez un bloc de code clôturé avec l'`no_run` annotation :

```
// Upload all local terrariums to the online gallery.
//
// `no_run
// let mut session = fern_sim::connect();
// session.upload_all();
//
// `no_run
pub fn upload_all(&mut self) {
    ...
}
```

Si le code n'est même pas censé être compilé, utilisez à la place `ignore` de `no_run`. Les blocs marqués par `ignore` n'apparaissent pas dans la sortie de `cargo run`, mais `no_run` les tests s'affichent comme ayant réussi si les se compilent. Si le bloc de code n'est pas du tout du code Rust, utilisez le nom du langage, comme `c++` ou `sh`, ou `text` pour du texte brut. `rustdoc` ne connaît pas les noms de certaines de langages de programmation ; il traite plutôt toute annotation qu'il ne reconnaît pas comme indiquant que le bloc de code n'est pas Rust. Cela désactive la mise en surbrillance du code ainsi que les tests de documentation.

Spécification des dépendances

Nous avons vu une façon de dire à Cargo où obtenir le code source pour les crates dont dépend votre projet : par `versionNuméro`.

```
image = "0.6.1"
```

Il existe plusieurs façons de spécifier les dépendances, et certaines choses plutôt nuancées que vous voudrez peut-être dire sur les versions à utiliser, il vaut donc la peine de consacrer quelques pages à ce sujet.

Tout d'abord, vous voudrez peut-être utiliser des dépendances qui ne sont pas du tout publiées sur crates.io. Pour ce faire, vous pouvez notamment spécifier une URL et une révision du référentiel Git :

```
image = { git = "https://github.com/Piston/image.git", rev = "528f19c" }
```

Cette caisse particulière est open source, hébergée sur GitHub, mais vous pouvez tout aussi bien pointer vers un référentiel Git privé hébergé sur votre réseau d'entreprise. Comme indiqué ici, vous pouvez spécifier le particulier `rev`, `tag` ou `branch` à utiliser. (Ce sont toutes des façons d'indiquer à Git quelle révision du code source vérifier.)

Une autre alternative consiste à spécifier un répertoire contenant le code source du crate :

```
image = { chemin = "fournisseur/image" }
```

Ceci est pratique lorsque votre équipe dispose d'un référentiel de contrôle de version unique qui contient le code source de plusieurs caisses, ou peut-être l'intégralité du graphique de dépendance. Chaque caisse peut spécifier ses dépendances à l'aide de chemins relatifs.

Avoir ce niveau de contrôle sur vos dépendances est puissant. Si jamais vous décidez que l'un des crates open source que vous utilisez n'est pas exactement à votre goût, vous pouvez le bifurquer de manière triviale : appuyez simplement sur le bouton Fork sur GitHub et modifiez une ligne dans votre fichier *Cargo.toml*. Votre prochain `cargo build` utilisera de manière transparente votre fork of the crate au lieu de la version officielle.

Versions

Lorsque vous écrivez quelque chose comme `image = "0.13.0"` dans votre fichier *Cargo.toml*, Cargo interprète cela de manière assez vague. Il utilise la version la plus récente de `image` qui est considérée comme compatible avec la version 0.13.0.

Les règles de compatibilité sont adaptées de [Semantic Versioning](#).

- Un numéro de version qui commence par 0.0 est si brut que Cargo ne suppose jamais qu'il est compatible avec une autre version.
- Un numéro de version qui commence par 0. *x*, où *x* est différent de zéro, est considéré comme compatible avec les autres versions ponctuelles de la série 0. *x*. Nous avons spécifié `image` la version 0.6.1, mais Cargo utiliserait la version 0.6.3 si disponible. (Ce n'est pas ce que dit la norme Semantic Versioning à propos des numéros de version 0. *x*, mais la règle s'est avérée trop utile pour être omise.)
- Une fois qu'un projet atteint la version 1.0, seules les nouvelles versions majeures rompent la compatibilité. Donc, si vous demandez la version 2.0.1, Cargo utilisera peut-être la 2.17.99 à la place, mais pas la 3.0.

Les numéros de version sont flexibles par défaut car sinon le problème de la version à utiliser deviendrait rapidement surcontraint. Supposons qu'une bibliothèque, `libA`, utilise `num = "0.1.31"` tandis qu'une autre, `libB`, utilise `num = "0.1.29"`. Si les numéros de version nécessitaient des correspondances exactes, aucun projet ne pourrait utiliser ces deux bibliothèques ensemble. Permettre à Cargo d'utiliser n'importe quelle version compatible est une valeur par défaut beaucoup plus pratique.

Pourtant, différents projets ont des besoins différents en matière de dépendances et de gestion des versions. Vous pouvez spécifier une version exacte ou une plage de versions à l'aide d'opérateurs, comme illustré dans le [Tableau 8-3](#).

Ligne Cargo.toml	Sens
<code>image = "=0.1.0.0"</code>	Utilisez uniquement la version exacte 0.10.0
<code>image = ">=1.0.5"</code>	Utilisez 1.0.5 ou <i>toute</i> version supérieure (même 2.9, si elle est disponible)
<code>image = ">1.0.5 <1.1.9"</code>	Utilisez une version supérieure à 1.0.5, mais inférieure à 1.1.9
<code>image = "<=2.7.10"</code>	Utilisez n'importe quelle version jusqu'à 2.7.10

Une autre spécification de version que vous verrez occasionnellement est le caractère générique `*`. Cela indique à Cargo que n'importe quelle version fera l'affaire. À moins qu'un autre fichier *Cargo.toml* ne contienne une contrainte plus spécifique, Cargo utilisera la dernière version disponible. [La documentation Cargo sur doc.crates.io](#) couvre les spécifications de version de manière encore plus détaillée.

Notez que les règles de compatibilité signifient que les numéros de version ne peuvent pas être choisis uniquement pour des raisons de marketing. Ils veulent vraiment dire quelque chose. Il s'agit d'un contrat entre les responsables d'une caisse et ses utilisateurs. Si vous maintenez un crate qui est à la version 1.7 et que vous décidez de supprimer une fonction ou d'apporter toute autre modification qui n'est pas entièrement rétrocompatible, vous devez faire passer votre numéro de version à 2.0. Si vous deviez l'appeler 1.8, vous prétendriez que la nouvelle version est compatible avec la 1.7, et vos utilisateurs pourraient se retrouver avec des versions cassées.

Cargo.lock

La version les numéros dans *Cargo.toml* sont délibérément flexibles, mais nous ne voulons pas que Cargo nous mette à niveau vers les dernières versions de la bibliothèque à chaque fois que nous construisons. Imaginez être au milieu d'une session de débogage intense lorsque `cargo build` vous êtes soudainement mis à niveau vers une nouvelle version d'une bibliothèque. Cela pourrait être incroyablement perturbateur. Tout ce qui change au milieu du débogage est mauvais. En fait, lorsqu'il s'agit de bibliothèques, il n'y a jamais de bon moment pour un changement inattendu.

Cargo dispose donc d'un mécanisme intégré pour empêcher cela. La première fois que vous créez un projet, Cargo génère un fichier *Cargo.lock* qui enregistre la version exacte de chaque caisse utilisée. Les versions ultérieures consulteront ce fichier et continueront à utiliser les mêmes versions. Cargo met à niveau vers des versions plus récentes uniquement lorsque vous le lui demandez, soit en augmentant manuellement le numéro de version dans votre fichier *Cargo.toml*, soit en exécutant `cargo update`:

```
$ mise à jour de la cargaison
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating libc v0.2.7 -> v0.2.11
    Updating png v0.4.2 -> v0.4.3
```

`cargo update` uniquement les mises à niveau vers les dernières versions compatibles avec ce que vous avez spécifié dans *Cargo.toml*. Si vous avez spécifié `image = "0.6.1"` et que vous souhaitez mettre à niveau vers la version 0.10.0, vous devrez modifier cela dans *Cargo.toml*. La prochaine fois que vous compilerez, Cargo mettra à jour la nouvelle version de la `image` bibliothèque et stockera le nouveau numéro de version dans *Cargo.lock*.

L'exemple précédent montre Cargo mettant à jour deux caisses hébergées sur crates.io. Quelque chose de très similaire se produit pour les dépendances stockées dans Git. Supposons que notre fichier *Cargo.toml* contienne ceci :

```
image = { git = "https://github.com/Piston/image.git", branche = "maître" }
```

`cargo build` ne tirera pas de nouvelles modifications du référentiel Git s'il voit que nous avons un fichier *Cargo.lock*. Au lieu de cela, il lit *Cargo.lock* et utilise la même révision que la dernière fois. Mais `cargo update` tirera de `master` sorte que notre prochaine version utilise la dernière révision.

Cargo.lock est généré automatiquement pour vous et vous ne le modifiez normalement pas à la main. Néanmoins, si votre projet est un exécutable, vous devez valider *Cargo.lock* pour le contrôle de version. De cette façon, tous ceux qui construisent votre projet obtiendront systématiquement les mêmes versions. L'historique de votre fichier *Cargo.lock* enregistrera vos mises à jour de dépendance.

Si votre projet est une bibliothèque Rust ordinaire, ne vous embêtez pas à valider *Cargo.lock*. Les utilisateurs en aval de votre bibliothèque auront des fichiers *Cargo.lock* contenant des informations de version pour l'en-

semble de leur graphique de dépendance ; ils ignoreront le fichier *Cargo.lock* de votre bibliothèque. Dans les rares cas où votre projet est une bibliothèque partagée (c'est-à-dire que la sortie est un fichier *.dll*, *.dylib* ou *.so*), il n'y a pas d'utilisateur en aval de ce type `cargo` et vous devez donc valider *Cargo.lock*.

Les spécificateurs de version flexibles de Cargo.toml facilitent l'utilisation des bibliothèques Rust dans votre projet et maximisent la compatibilité entre les bibliothèques. *La comptabilité de Cargo.lock* prend en charge des constructions cohérentes et reproductibles sur toutes les machines. Ensemble, ils contribuent grandement à vous aider à éviter l'enfer de la dépendance.

Publier des caisses sur crates.io

Vous avez décidé de publier votre bibliothèque de simulation de fougères en tant que logiciel open source. Toutes nos félicitations! Cette partie est facile.

Tout d'abord, assurez-vous que Cargo peut emballer la caisse pour toi.

```
$ colis de fret
warning: manifest has no description, license, license-file, documentation,
homepage or repository. See http://doc.crates.io/manifest.html#package-metadata
for more info.

Packaging fern_sim v0.1.0 (file:///.../fern_sim)
Verifying fern_sim v0.1.0 (file:///.../fern_sim)
Compiling fern_sim v0.1.0 (file:///.../fern_sim/target/package/fern_sim-0.1.0.crate)
```

La `cargo package` commande crée un fichier (dans ce cas, *target/package/fern_sim-0.1.0.crate*) contenant tous les fichiers sources de votre bibliothèque, y compris *Cargo.toml*. C'est le fichier que vous téléchargerez sur crates.io pour le partager avec le monde. (Vous pouvez utiliser `cargo package --list` pour voir quels fichiers sont inclus.) Cargo revérifie ensuite son travail en créant votre bibliothèque à partir du fichier *.crate*, tout comme vos utilisateurs éventuels le feront.

Cargo avertit qu'il manque à la `[package]` section de *Cargo.toml* certaines informations qui seront importantes pour les utilisateurs en aval, telles que la licence sous laquelle vous distribuez le code. L'URL dans l'avertissement est une excellente ressource, nous n'expliquerons donc pas tous les champs en détail ici. En bref, vous pouvez corriger l'avertissement en ajoutant quelques lignes à *Cargo.toml*:

```
[forfait]
nom = "fougère_sim"
version = "0.1.0"
édition = "2021"
auteurs = ["Vous <vous@example.com>"]
licence = "MIT"
page d'accueil = "https://fernsm.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsm.example.com/docs"
descriptif = """
Simulation de fougère, du niveau cellulaire vers le haut.
"""

```

NOTER

Une fois que vous avez publié cette caisse sur crates.io, toute personne qui télécharge votre caisse peut voir le fichier *Cargo.toml*. Donc, si le `authors` champ contient une adresse e-mail que vous préférez garder privée, il est maintenant temps de la changer.

Un autre problème qui survient parfois à ce stade est que votre fichier *Cargo.toml* peut spécifier l'emplacement d'autres caisses par `path`, comme indiqué dans [« Spécification des dépendances »](#) :

```
image = { chemin = "fournisseur/image" }
```

Pour vous et votre équipe, cela pourrait bien fonctionner. Mais naturellement, lorsque d'autres personnes téléchargent la `fern_sim` bibliothèque, elles n'auront pas les mêmes fichiers et répertoires sur leur ordinateur que vous. *Cargo ignore* donc la `path` clé dans les bibliothèques téléchargées automatiquement, ce qui peut entraîner des erreurs de construction. Le correctif, cependant, est simple : si votre bibliothèque doit être publiée sur crates.io, ses dépendances doivent également l'être sur crates.io. Spécifiez un numéro de version au lieu d'un `path` :

```
image="0.13.0"
```

Si vous préférez, vous pouvez spécifier à la fois un `path`, qui est prioritaire pour vos propres builds locaux, et une `version` pour tous les autres utilisateurs :

```
image = { chemin = "fournisseur/image", version = "0.13.0" }
```

Bien sûr, dans ce cas, il est de votre responsabilité de vous assurer que les deux restent synchronisés.

Enfin, avant de publier un crate, vous devrez vous connecter à crates.io et obtenir une clé API. Cette étape est simple : une fois que vous avez un compte sur crates.io, votre page "Paramètres du compte" affichera une `cargo login` commande, comme celle-ci :

```
$ connexion cargo 5j0dv54Bj1XBpUUbfIj7G9DvNl1vsWW1
```

Cargo enregistre la clé dans un fichier de configuration, et la clé API doit être gardée secrète, comme un mot de passe. Exécutez donc cette commande uniquement sur un ordinateur que vous contrôlez.

Ceci fait, la dernière étape consiste à lancer `cargo publish`:

```
$ cargo publier
  Updating registry `https://github.com/rust-lang/crates.io-index`
    Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

Avec cela, votre bibliothèque rejoint des milliers d'autres sur crates.io.

Espaces de travail

Comme votre projet continue de croître, vous finissez par écrire de nombreuses caisses. Ils vivent côté à côté dans un référentiel source unique :

```
fernsoft/
  -- .git/...
  -- fern_sim/
    -- Cargo.toml
    -- Cargo.lock
    -- src/...
    -- target/...
  -- fern_img/
    -- Cargo.toml
    -- Cargo.lock
    -- src/...
    -- target/...
  -- fern_video/
    -- Cargo.toml
    -- Cargo.lock
    -- src/...
    -- target/...
```

De la manière dont Cargo fonctionne, chaque caisse a son propre répertoire de construction, `target`, qui contient une version distincte de toutes les dépendances de cette caisse. Ces répertoires de construction sont complètement indépendants. Même si deux crates ont une dépendance commune, ils ne peuvent partager aucun code compilé. C'est du gaspillage.

Vous pouvez économiser du temps de compilation et de l'espace disque en utilisant un espace de travail Cargo, une collection de caisses qui partagent un répertoire de construction commun et le fichier `Cargo.lock`.

Tout ce que vous avez à faire est de créer un fichier `Cargo.toml` dans le répertoire racine de votre référentiel et d'y mettre ces lignes :

```
[espace de travail]
membres = ["fern_sim", "fern_img", "fern_video"]
```

Voici `fern_sim` etc. sont les noms des sous-répertoires contenant vos caisses. Supprimez tous les fichiers `Cargo.lock` restants et les répertoires cibles qui existent dans ces sous-répertoires.

Une fois que vous avez fait cela, `cargo build` dans n'importe quel crate créera et utilisera automatiquement un répertoire de construction partagé sous le répertoire racine (dans ce cas, `fernsoft/target`). La commande `cargo build --workspace` construit toutes les caisses dans l'espace de travail actuel. `cargo test` et `cargo doc` acceptez `--workspace` également l'option.

Plus de belles choses

Au cas où vous ne seriez pas encore ravi, la communauté Rust a quelques autres bric et de broc pour vous :

- Lorsque vous publiez un crate open source sur [crates.io](#), votre documentation est automatiquement rendue et hébergée sur `docs.rs` grâce à Onur Aslan.
- Si votre projet est sur GitHub, Travis CI peut créer et tester votre code à chaque poussée. Il est étonnamment facile à configurer ; voir [travis-ci.org](#) pour plus de détails. Si vous connaissez déjà Travis, ce fichier `.travis.yml` vous aidera à démarrer :

```
language: rust
rust:
  - stable
```

- Vous pouvez générer un fichier *README.md* à partir du commentaire doc de niveau supérieur de votre crate. Cette fonctionnalité est proposée en tant que plug-in Cargo tiers par Livio Ribeiro. Exécutez `cargo install cargo-readme` pour installer le plug-in, puis `cargo readme --help` pour apprendre à l'utiliser.

Nous pourrions continuer.

Rust est nouveau, mais il est conçu pour soutenir de grands projets ambitieux. Il a d'excellents outils et une communauté active. Les programmeurs système *peuvent* avoir de belles choses.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 9. Structures

Autrefois, lorsque les bergers voulaient voir si deux troupeaux de moutons étaient isomorphes, ils recherchaient un isomorphisme explicite.

—John C. Baez et James Dolan, « [Catégorisation](#) »

RouillerLes structures, parfois appelées *structures*, ressemblent aux `struct` types en C et C++, aux classes en Python et aux objets en JavaScript. Une structure assemble plusieurs valeurs de types assortis en une seule valeur afin que vous puissiez les traiter comme une unité. Étant donné une structure, vous pouvez lire et modifier ses composants individuels. Et une structure peut être associée à des méthodes qui fonctionnent sur ses composants.

Rust a trois types de types de structure, *named-field*, *tuple-like* et *unit-like*, qui diffèrent dans la façon dont vous vous référez à leurs composants : une structure de champ nommé donne un nom à chaque composant, tandis qu'une structure de type tuple identifie par l'ordre dans lequel ils apparaissent. Les structures de type unité n'ont aucun composant ; ce ne sont pas courants, mais plus utiles que vous ne le pensez.

Dans ce chapitre, nous expliquerons chaque type en détail et montrerons à quoi ils ressemblent en mémoire. Nous verrons comment leur ajouter des méthodes, comment définir des types de structures génériques qui fonctionnent avec de nombreux types de composants différents et comment demander à Rust de générer des implémentations de traits pratiques courants pour vos structures.

Structures de champ nommé

La définition d'une structure de champ nommé le type ressemble à ceci :

```
// A rectangle of eight-bit grayscale pixels.
struct GrayscaleMap {
    pixels: Vec<u8>,
    size:(usize, usize)
}
```

Ceci déclare un type `GrayscaleMap` avec deux champs nommés `pixels` et `size`, des types donnés. La convention dans Rust est que tous les types, structures incluses, aient des noms qui mettent en majuscule la première lettre de chaque mot, comme `GrayscaleMap`, une convention appelée *CamelCase* (ou *PascalCase*). Les champs et les méthodes sont en minuscules, les mots étant séparés par des traits de soulignement. Ceci s'appelle *snake_case*.

Vous pouvez construire une valeur de ce type avec une *expression struct*, comme ceci :

```
let width = 1024;
let height = 576;
let image = GrayscaleMap {
    pixels: vec![0; width * height],
    size:(width, height)
};
```

Une expression structurée commence par le nom du type (`GrayscaleMap`) et répertorie le nom et la valeur de chaque champ, le tout entre accolades. Il existe également un raccourci pour remplir les champs à partir de variables locales ou d'arguments portant le même nom :

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) ->GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

L'expression de structure `GrayscaleMap { pixels, size }` est l'abréviation de `GrayscaleMap { pixels: pixels, size: size }`. Vous pouvez utiliser `key: value` la syntaxe pour certains champs et la sténo-graphie pour d'autres dans la même expression de structure.

Pour accéder aux champs d'une structure, utilisez l' `.` opérateur familier :

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

Comme tous les autres éléments, les structures sont privées par défaut, visibles uniquement dans le module où elles sont déclarées et ses sous-modules. Vous pouvez rendre une structure visible en dehors de son mo-

dule en préfixant sa définition avec `pub`. Il en va de même pour chacun de ses champs, qui sont également privés par défaut :

```
// A rectangle of eight-bit grayscale pixels.  
pub struct GrayscaleMap {  
    pub pixels: Vec<u8>,  
    pub size:(usize, usize)  
}
```

Même si une structure est déclarée `pub`, ses champs peuvent être privés :

```
// A rectangle of eight-bit grayscale pixels.  
pub struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size:(usize, usize)  
}
```

D'autres modules peuvent utiliser cette structure et toutes les fonctions publiques associées qu'elle pourrait avoir, mais ne peuvent pas accéder aux champs privés par leur nom ou utiliser des expressions de structure pour créer de nouvelles `GrayscaleMap` valeurs. Autrement dit, la création d'une valeur de structure nécessite que tous les champs de la structure soient visibles. C'est pourquoi vous ne pouvez pas écrire une expression de structure pour créer un nouveau `String` ou `Vec`. Ces types standard sont des structures, mais tous leurs champs sont privés. Pour en créer un, vous devez utiliser des fonctions publiques associées à un type telles que `Vec::new()`.

Lors de la création d'une valeur de structure de champ nommé, vous pouvez utiliser une autre structure du même type pour fournir des valeurs pour les champs que vous omettez. Dans une expression `struct`, si les champs nommés sont suivis de `... EXPR`, tous les champs non mentionnés tirent leurs valeurs de `EXPR`, qui doit être une autre valeur du même type `struct`. Supposons que nous ayons une structure représentant un monstre dans un jeu :

```
// In this game, brooms are monsters. You'll see.  
struct Broom {  
    name: String,  
    height: u32,  
    health: u32,  
    position: (f32, f32, f32),  
    intent:BroomIntent  
}
```

```

/// Two possible alternatives for what a `Broom` could be working on.
#[derive(Copy, Clone)]
enum BroomIntent { FetchWater, DumpWater }

```

Le meilleur conte de fées pour programmeurs est *L'apprenti sorcier* : un magicien novice enchante un balai pour qu'il fasse son travail à sa place, mais ne sait pas comment l'arrêter une fois le travail terminé. Couper le balai en deux avec une hache ne produit que deux balais, chacun de la moitié de la taille, mais poursuivant la tâche avec le même dévouement aveugle que l'original :

```

// Receive the input Broom by value, taking ownership.
fn chop(b: Broom) -> (Broom, Broom) {
    // Initialize `broom1` mostly from `b`, changing only `height`. Since
    // `String` is not `Copy`, `broom1` takes ownership of `b`'s name.
    let mut broom1 = Broom { height:b.height / 2, .. b };

    // Initialize `broom2` mostly from `broom1`. Since `String` is not
    // `Copy`, we must clone `name` explicitly.
    let mut broom2 = Broom { name:broom1.name.clone(), .. broom1 };

    // Give each fragment a distinct name.
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}

```

Avec cette définition en place, nous pouvons créer un balai, le couper en deux et voir ce que nous obtenons :

```

let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    health: 100,
    position: (100.0, 200.0, 0.0),
    intent: BroomIntent::FetchWater
};

let (hokey1, hokey2) = chop(hokey);
assert_eq!(hokey1.name, "Hokey I");
assert_eq!(hokey1.height, 30);
assert_eq!(hokey1.health, 100);

assert_eq!(hokey2.name, "Hokey II");

```

```
assert_eq!(hokey2.height, 30);
assert_eq!(hokey2.health, 100);
```

Les nouveaux `hokey1` et `hokey2` les balais ont reçu des noms ajustés, la moitié de la hauteur et toute la santé de l'original.

Structures de type tuple

Le deuxième type de structure type est appelé un *tuple-like struct*, car il ressemble à un tuple :

```
struct Bounds(usize, usize);
```

Vous construisez une valeur de ce type comme vous le feriez pour un tuple, sauf que vous devez inclure le nom de la structure :

```
let image_bounds = Bounds(1024, 768);
```

Les valeurs détenues par une structure de type tuple sont appelées *éléments*, tout comme le sont les valeurs d'un tuple. Vous y accédez comme vous le feriez pour un tuple :

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

Les éléments individuels d'une structure de type tuple peuvent être publics ou non :

```
pub struct Bounds(pub usize, pub usize);
```

L'expression `Bounds(1024, 768)` ressemble à un appel de fonction, et en fait c'est le cas : la définition du type définit également implicitement une fonction :

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

Au niveau le plus fondamental, les structures de champ nommé et de type tuple sont très similaires. Le choix de celui à utiliser se résume à des questions de lisibilité, d'ambiguïté et de brièveté. Si vous utilisez l' `.` opérateur pour accéder aux composants d'une valeur, l'identification des champs par leur nom fournit au lecteur plus d'informations et est proba-

lement plus robuste contre les fautes de frappe. Si vous utilisez généralement la correspondance de modèles pour trouver les éléments, les structures de type tuple peuvent bien fonctionner.

Les structures de type tuple sont bonnes pour les *nouveaux types*, structures avec un seul composant que vous définissez pour obtenir une vérification de type plus stricte. Par exemple, si vous travaillez uniquement avec du texte ASCII, vous pouvez définir un nouveau type comme ceci :

```
struct Ascii(Vec<u8>);
```

L'utilisation de ce type pour vos chaînes ASCII est bien meilleure que de simplement passer des `Vec<u8>` tampons et d'expliquer ce qu'ils sont dans les commentaires. Le newtype aide Rust à détecter les erreurs lorsqu'un autre tampon d'octets est passé à une fonction attendant du texte ASCII. Nous donnerons un exemple d'utilisation de newtypes pour des conversions de type efficaces au [chapitre 22](#).

Structures de type unité

Le troisième type de structure est un peu obscur : il déclare un type struct sans aucun élément :

```
struct Onesuch;
```

Une valeur d'un tel type n'occupe pas de mémoire, tout comme le type d'unité `()`. Rust ne prend pas la peine de stocker en mémoire des valeurs de structure de type unité ou de générer du code pour les exploiter, car il peut dire tout ce dont il a besoin de savoir sur la valeur à partir de son seul type. Mais logiquement, une structure vide est un type avec des valeurs comme les autres, ou plus précisément, un type dont il n'y a qu'une seule valeur :

```
let o = Onesuch;
```

Vous avez déjà rencontré une structure de type unité lors de la lecture de l' `..` opérateur de plage dans ["Fields and Elements"](#). Alors qu'une expression like `3..5` est un raccourci pour la struct value `Range { start: 3, end: 5 }`, l'expression `..`, une plage omettant les deux points de terminaison, est un raccourci pour la struct value `RangeFull`.

Les structures de type unité peuvent également être utiles lorsque vous travaillez avec des traits, que nous décrirons au [chapitre 11](#).

Disposition de la structure

En mémoire, les structures de champ nommé et de type tuplesont la même chose : un ensemble de valeurs, de types éventuellement mixtes, agencées d'une manière particulière en mémoire. Par exemple, plus tôt dans le chapitre, nous avons défini cette structure :

```
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size:(usize, usize)  
}
```

Une `GrayscaleMap` valeur est disposée en mémoire comme illustré à la [Figure 9-1](#).

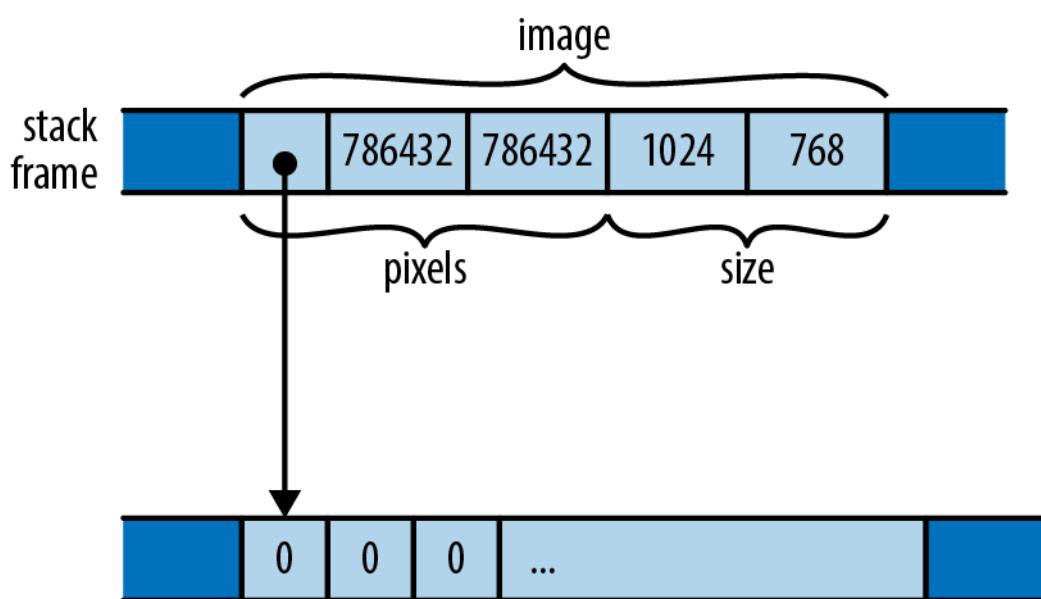


Illustration 9-1. Une `GrayscaleMap` structure en mémoire

Contrairement à C et C++, Rust ne fait pas de promesses spécifiques sur la manière dont il ordonnera les champs ou les éléments d'une structure en mémoire ; ce schéma ne montre qu'un seul agencement possible. Cependant, Rust promet de stocker les valeurs des champs directement dans le bloc de mémoire de la structure. Alors que JavaScript, Python et Java placeraient les valeurs `pixels` et `size` chacun dans leurs propres blocs alloués par tas et feraient `GrayscaleMap` pointer les champs vers eux, Rust intègre `pixels` et `size` directement dans la `GrayscaleMap` valeur. Seul le tampon alloué au tas appartenant au `pixels` vecteur reste dans son propre bloc.

Vous pouvez demander à Rust de disposer les structures d'une manière compatible avec C et C++, en utilisant l' `#[repr(C)]` attribut. Nous aborderons cela en détail au [chapitre 23](#).

Définir des méthodes avec `impl`

À travers le livre que nous appelons des méthodes sur toutes sortes de valeurs. Nous avons poussé des éléments sur des vecteurs avec `v.push(e)`, récupéré leur longueur avec `v.len()`, vérifié les `Result` valeurs pour les erreurs avec `r.expect("msg")`, etc. Vous pouvez également définir des méthodes sur vos propres types de structure. Plutôt que d'apparaître dans la définition de structure, comme en C++ ou Java, les méthodes Rust apparaissent dans un `impl` bloc séparé.

Un `impl` bloc est simplement une collection de `fn` définitions, dont chacune devient une méthode sur le type de structure nommé en haut du bloc. Ici, par exemple, nous définissons un public struct `Queue`, puis lui donnons deux méthodes publiques, `push` et `pop`:

```
// A first-in, first-out queue of characters.
pub struct Queue {
    older: Vec<char>, // older elements, eldest last.
    younger: Vec<char> // younger elements, youngest last.
}

impl Queue {
    // Push a character onto the back of a queue.
    pub fn push(&mut self, c: char) {
        self.younger.push(c);
    }

    // Pop a character off the front of a queue. Return `Some(c)` if there
    // was a character to pop, or `None` if the queue was empty.
    pub fn pop(&mut self) -> Option<char> {
        if self.oldier.is_empty() {
            if self.younger.is_empty() {
                return None;
            }
        }

        // Bring the elements in younger over to older, and put them in
        // the promised order.
        use std::mem::swap;
        swap(&mut self.oldier, &mut self.younger);
        self.oldier.reverse();
    }
}
```

```

        // Now older is guaranteed to have something. Vec's pop method
        // already returns an Option, so we're set.
        self.older.pop()
    }
}

```

Les fonctions définies dans un `impl` bloc sont appelées *fonctions associées*, puisqu'ils sont associés à un type spécifique. L'opposé d'une fonction associée est une *fonction libre*, celui qui n'est pas défini comme `impl` élément d'un bloc.

Rust transmet à une méthode la valeur sur laquelle elle est appelée comme premier argument, qui doit avoir le nom spécial `self`. Étant donné `self` que le type de est évidemment celui nommé en haut du `impl` bloc, ou une référence à celui-ci, Rust vous permet d'omettre le type et d'écrire `self`, `&self` ou `&mut self` comme raccourci pour `self: Queue`, `self: &Queue` ou `self: &mut Queue`. Vous pouvez utiliser les formulaires longs si vous le souhaitez, mais presque tout le code Rust utilise le raccourci, comme indiqué précédemment.

Dans notre exemple, les méthodes `push` et `pop` font référence aux champs de `older` et `younger` comme et . Contrairement à C++ et Java, où les membres de l'objet "this" sont directement visibles dans les corps de méthode en tant qu'identifiants non qualifiés, une méthode Rust doit explicitement utiliser pour faire référence à la valeur sur laquelle elle a été appelée, de la même manière que les méthodes Python utilisent `,` et le façon dont les méthodes JavaScript utilisent `this`

```
.pop Queue self.older self.younger self self this
```

Depuis `push` et `pop` doivent modifier le `Queue`, ils prennent tous les deux `&mut self`. Cependant, lorsque vous appelez une méthode, vous n'avez pas besoin d'emprunter vous-même la référence mutable ; la syntaxe d'appel de méthode ordinaire s'en charge implicitement. Donc, avec ces définitions en place, vous pouvez utiliser `Queue` comme ceci :

```

let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('0');
q.push('1');
assert_eq!(q.pop(), Some('0'));

q.push('∞');
assert_eq!(q.pop(), Some('1'));
assert_eq!(q.pop(), Some('∞'));
assert_eq!(q.pop(), None);

```

L'écriture simple `q.push(...)` emprunte une référence mutable à `q`, comme si vous aviez écrit `(&mut q).push(...)`, puisque c'est ce que la `push` méthode `self` exige.

Si une méthode n'a pas besoin de modifier son `self`, vous pouvez la définir pour prendre une référence partagée à la place. Par exemple:

```
impl Queue {  
    pub fn is_empty(&self) ->bool {  
        self.older.is_empty() && self.younger.is_empty()  
    }  
}
```

Encore une fois, l'expression d'appel de méthode sait quel type de référence emprunter :

```
assert!(q.is_empty());  
q.push('o');  
assert!(!q.is_empty());
```

Ou, si une méthode veut s'approprier `self`, elle peut prendre `self` par valeur :

```
impl Queue {  
    pub fn split(self) ->(Vec<char>, Vec<char>) {  
        (self.older, self.younger)  
    }  
}
```

L'appel de cette `split` méthode ressemble aux autres appels de méthode :

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };  
  
q.push('P');  
q.push('D');  
assert_eq!(q.pop(), Some('P'));  
q.push('X');  
  
let (older, younger) = q.split();  
// q is now uninitialized.  
assert_eq!(older, vec!['D']);  
assert_eq!(younger, vec!['X']);
```

Mais notez que, puisque `split` prend sa `self` par valeur, cela déplace le `Queue` hors de `q`, laissant `q` non initialisé. Étant donné `split` que `self` possède maintenant la file d'attente, il est capable d'en retirer les vecteurs individuels et de les renvoyer à l'appelant.

Parfois, prendre `self` par valeur comme celle-ci, ou même par référence, ne suffit pas, donc Rust vous permet également de passer `self` par des types de pointeurs intelligents.

Se faire passer pour une boîte, un Rc ou un Arc

`self` L' argument d'une méthode peut également être un `Box<Self>`, `Rc<Self>`, ou `Arc<Self>`. Une telle méthode peut être appelée que sur une valeur du type de pointeur donné. L'appel de la méthode lui transmet la propriété du pointeur.

Vous n'aurez généralement pas besoin de le faire. Une méthode qui attend `self` par référence fonctionne correctement lorsqu'elle est appelée sur l'un de ces types de pointeurs :

```
let mut bq = Box::new(Queue::new());  
  
// `Queue::push` expects a `&mut Queue`, but `bq` is a `Box<Queue>`.  
// This is fine: Rust borrows a `&mut Queue` from the `Box` for the  
// duration of the call.  
bq.push('■');
```

Pour les appels de méthode et l'accès aux champs, Rust emprunte automatiquement une référence à partir de types de pointeurs tels que `Box`, `Rc`, et `Arc`, donc `&self` et `&mut self` sont presque toujours la bonne chose dans une signature de méthode, avec occasionnellement `self`.

Mais s'il arrive qu'une méthode nécessite la propriété d'un pointeur sur `Self`, et que ses appelants disposent d'un tel pointeur, Rust vous laissera le passer comme `self` argument de la méthode. Pour ce faire, vous devez épeler le type de `self`, comme s'il s'agissait d'un paramètre ordinaire :

```
impl Node {  
    fn append_to(self: Rc<Self>, parent:&mut Node) {  
        parent.children.push(self);  
    }  
}
```

Fonctions associées au type

Un `impl` bloc pour un type donné peut aussi définir des fonctions qui ne prennent pas `self` du tout comme argument. Ce sont toujours des fonctions associées, puisqu'elles sont dans un `impl` bloc, mais ce ne sont pas des méthodes, puisqu'elles ne prennent pas d'`self` argument. Pour les distinguer des méthodes, nous les appelons *fonctions associées au type*.

Ils sont souvent utilisés pour fournir des fonctions constructeur, comme ceci :

```
impl Queue {
    pub fn new() -> Queue {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
}
```

Pour utiliser cette fonction, nous nous y référons comme `Queue::new` : le nom du type, un double deux-points, puis le nom de la fonction. Maintenant, notre exemple de code devient un peu plus svelte :

```
let mut q = Queue::new();
q.push('*');
...
```

Il est conventionnel dans Rust que les fonctions constructeur soient nommées `new` ; nous avons déjà vu `Vec::new`, `Box::new`, `HashMap::new`, et d'autres. Mais il n'y a rien de spécial dans le nom `new`. Ce n'est pas un mot-clé, et les types ont souvent d'autres fonctions associées qui servent de constructeurs, comme `Vec::with_capacity`.

Bien que vous puissiez avoir plusieurs blocs distincts `impl` pour un seul type, ils doivent tous se trouver dans la même caisse qui définit ce type. Cependant, Rust vous permet d'attacher vos propres méthodes à d'autres types ; nous expliquerons comment au [chapitre 11](#).

Si vous êtes habitué à C++ ou Java, séparer les méthodes d'un type de sa définition peut sembler inhabituel, mais il y a plusieurs avantages à le faire :

- Il est toujours facile de trouver les membres de données d'un type.

Dans les grandes définitions de classe C++, vous devrez peut-être par-

courir des centaines de lignes de définitions de fonctions membres pour vous assurer que vous n'avez manqué aucun des membres de données de la classe ; à Rust, ils sont tous au même endroit.

- Bien que l'on puisse imaginer intégrer des méthodes dans la syntaxe des structures de champ nommé, ce n'est pas si simple pour les structures de type tuple et de type unité. L'extraction de méthodes dans un `impl` bloc permet une syntaxe unique pour les trois. En fait, Rust utilise cette même syntaxe pour définir des méthodes sur des types qui ne sont pas du tout des structs, tels que des `enum` types et des types primatifs comme `i32`. (Le fait que n'importe quel type puisse avoir des méthodes est l'une des raisons pour lesquelles Rust n'utilise pas beaucoup le terme *objet*, préférant appeler tout une *valeur*.)
- La même `impl` syntaxe sert également parfaitement à implémenter des traits, que nous aborderons au [chapitre 11](#).

Const associés

Une autre caractéristique des langages comme C# et Java que Rust adopte dans son système de types est l'idée de valeurs associées à un type, plutôt qu'une instance spécifique de ce type. Dans Rust, ceux-ci sont connus sous le nom de *consts associés*.

Comme son nom l'indique, les constantes associées sont des valeurs constantes. Ils sont souvent utilisés pour spécifier les valeurs couramment utilisées d'un type. Par exemple, vous pouvez définir un vecteur bidimensionnel à utiliser en algèbre linéaire avec un vecteur unitaire associé :

```
pub struct Vector2 {
    x: f32,
    y:f32,
}

impl Vector2 {
    const ZERO: Vector2 = Vector2 { x: 0.0, y: 0.0 };
    const UNIT: Vector2 = Vector2 { x: 1.0, y:0.0 };
}
```

Ces valeurs sont associées au type lui-même et vous pouvez les utiliser sans faire référence à une autre instance de `vector2`. Tout comme les fonctions associées, elles sont accessibles en nommant le type auquel elles sont associées, suivi de leur nom :

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

Un const associé ne doit pas non plus être du même type que le type auquel il est associé ; nous pourrions utiliser cette fonctionnalité pour ajouter des identifiants ou des noms aux types. Par exemple, s'il y avait plusieurs types similaires à ceux `Vector2` qui devaient être écrits dans un fichier puis chargés en mémoire plus tard, un const associé pourrait être utilisé pour ajouter des noms ou des identifiants numériques qui pourraient être écrits à côté des données pour identifier son type :

```
impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID:u32 = 18;
}
```

Structures génériques

Notre plus tôt La définition de `Queue` n'est pas satisfaisante : il est écrit pour stocker des caractères, mais il n'y a rien dans sa structure ou ses méthodes qui soit spécifique aux caractères. Si nous devions définir une autre structure contenant, par exemple, `String` des valeurs, le code pourrait être identique, sauf qu'il `char` serait remplacé par `String`. Ce serait une perte de temps.

Heureusement, les structures Rust peuvent être *génériques*, ce qui signifie que leur définition est un modèle dans lequel vous pouvez insérer les types de votre choix. Par exemple, voici une définition pour `Queue` qui peut contenir des valeurs de n'importe quel type :

```
pub struct Queue<T> {
    older: Vec<T>,
    younger:Vec<T>
}
```

Vous pouvez lire le `<T>` dans `Queue<T>` comme "pour tout type d'élément `T`...". Donc, cette définition se lit comme suit : "Pour tout type `T`, a `Queue<T>` est deux champs de type `Vec<T>`." Par exemple, dans `Queue<String>`, `T` is `String`, so `older` et `younger` have type `Vec<String>`. Dans `Queue<char>`, `T` est `char`, et nous obtenons une structure identique à la `char` définition spécifique avec laquelle nous

avons commencé. En fait, `Vec` elle-même est une structure générique, définie exactement de cette manière.

Dans les définitions de structure génériques, les noms de type utilisés < entre crochets > sont appelés *paramètres de type*. Un `impl` bloc pour une structure générique ressemble à ceci :

```
impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t:T) {
        self.younger.push(t);
    }

    pub fn is_empty(&self) ->bool {
        self.older.is_empty() && self.younger.is_empty()
    }

    ...
}
```

Vous pouvez lire la ligne `impl<T> Queue<T>` comme quelque chose comme « pour tout type `T`, voici quelques fonctions associées disponibles sur `Queue<T>` ». Ensuite, vous pouvez utiliser le paramètre de type `T` comme type dans les définitions de fonctions associées.

La syntaxe peut sembler un peu redondante, mais `impl<T>` il est clair que le `impl` bloc couvre n'importe quel type `T`, ce qui le distingue d'un `impl` bloc écrit pour un type spécifique de `Queue`, comme celui-ci :

```
impl Queue<f64> {
    fn sum(&self) ->f64 {
        ...
    }
}
```

Cet `impl` en-tête de bloc se lit comme suit : "Voici quelques fonctions associées spécifiquement pour `Queue<f64>` ." Cela donne `Queue<f64>` une `sum` méthode, disponible sur aucun autre type de `Queue`.

Nous avons utilisé le raccourci de Rust pour les `self` paramètres dans le code précédent ; écrire `Queue<T>` partout devient une bouchée et une distraction. Comme autre raccourci, chaque `impl` bloc, générique ou non,

définit le paramètre de type spécial `Self` (notez le CamelCase nom) comme étant le type auquel nous ajoutons des méthodes. Dans le code précédent, `Self` serait `Queue<T>`, nous pouvons donc abréger `Queue::new` un peu plus la définition de :

```
pub fn new() -> Self {  
    Queue { older: Vec::new(), younger: Vec::new() }  
}
```

Vous avez peut-être remarqué que, dans le corps de `new`, nous n'avions pas besoin d'écrire le paramètre de type dans l'expression de construction ; simplement écrire `Queue { ... }` suffisait. C'est l'inférence de type de Rust à l'œuvre : puisqu'il n'y a qu'un seul type qui fonctionne pour la valeur de retour de cette fonction, à savoir, `Queue<T>` —Rust nous fournit le paramètre. Cependant, vous devrez toujours fournir des paramètres de type dans les signatures de fonction et les définitions de type. Rust ne les déduit pas ; à la place, il utilise ces types explicites comme base à partir de laquelle il déduit les types dans les corps de fonction.

`Self` peut également être utilisé de cette manière; nous aurions pu écrire `Self { ... }` à la place. C'est à vous de décider ce que vous trouvez le plus facile à comprendre.

Pour les appels de fonction associés, vous pouvez fournir le paramètre de type explicitement en utilisant la `::<>` notation (turbofish) :

```
let mut q = Queue::<char>::new();
```

Mais en pratique, vous pouvez généralement laisser Rust le découvrir pour vous :

```
let mut q = Queue::new();  
let mut r = Queue::new();  
  
q.push("CAD"); // apparently a Queue<&'static str>  
r.push(0.74); // apparently a Queue<f64>  
  
q.push("BTC"); // Bitcoins per USD, 2019-6  
r.push(13764.0); // Rust fails to detect irrational exuberance
```

En fait, c'est exactement ce que nous avons fait avec `vec`, un autre type de structure générique, tout au long du livre.

Il n'y a pas que les structures qui peuvent être génériques. Les énumérations peuvent également prendre des paramètres de type, avec une syntaxe très similaire. Nous montrerons cela en détail dans [« Enums »](#).

Structures génériques avec paramètres de durée de vie

Comme nous l'avons vu dans [« Structures contenant des références »](#), si un type de structure contient des références, vous devez nommer les durées de vie de ces références. Par exemple, voici une structure qui peut contenir des références aux éléments les plus grands et les plus petits d'une tranche :

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}
```

Plus tôt, nous vous avons invité à penser à une déclaration comme `struct Queue<T>` signifiant que, étant donné n'importe quel type spécifique `T`, vous pouvez faire un `Queue<T>` contenant ce type. De même, vous pouvez penser `struct Extrema<'elt>` que, compte tenu de toute durée de vie spécifique `'elt`, vous pouvez créer un `Extrema<'elt>` qui contient des références avec cette durée de vie.

Voici une fonction pour analyser une tranche et renvoyer une `Extrema` valeur dont les champs font référence à ses éléments :

```
fn find_extrema<'s>(slice: &'s [i32]) ->Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
        if slice[i] < *least { least = &slice[i]; }
        if slice[i] > *greatest { greatest = &slice[i]; }
    }
    Extrema { greatest, least }
}
```

Ici, puisque `find_extrema` emprunte des éléments de `slice`, qui a life `'s`, la `Extrema` structure que nous retournons utilise également `'s` comme durée de vie de ses références. Rust déduit toujours les para-

mètres de durée de vie des appels, donc les appels `find_extrema` n'ont pas besoin de les mentionner :

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

Parce qu'il est si courant que le type de retour utilise la même durée de vie comme argument, Rust nous permet d'omettre les durées de vie lorsqu'il y a un candidat évident. Nous aurions aussi pu écrire

`find_extrema` la signature de comme ceci, sans changement de sens :

```
fn find_extrema(slice: &[i32]) ->Extrema {
    ...
}
```

Certes, nous *aurions* pu vouloir dire `Extrema<'static>`, mais c'est assez inhabituel. Rust fournit un raccourci pour le cas courant.

Structures génériques avec paramètres constants

Une structure générique peut également prendre des paramètres qui sont des valeurs constantes. Par exemple, vous pouvez définir un type représentant des polynômes de degré arbitraire comme ceci :

```
/// A polynomial of degree N - 1.
struct Polynomial<const N: usize> {
    /// The coefficients of the polynomial.
    ///
    /// For a polynomial a + bx + cx2 + ... + zxn-1,
    /// the `i`'th element is the coefficient of xi.
    coefficients:[f64; N]
}
```

Avec cette définition, `Polynomial<3>` est un polynôme quadratique, par exemple. La `<const N: usize>` clause indique que le `Polynomial` type attend une `usize` valeur comme paramètre générique, qu'il utilise pour décider du nombre de coefficients à stocker.

Contrairement à `Vec`, qui a des champs contenant sa longueur et sa capacité et stocke ses éléments dans le tas, `Polynomial` stocke ses coefficients directement dans la valeur, et rien d'autre. La longueur est donnée par le type. (La capacité n'est pas nécessaire, car `Polynomial`s ne peut pas croître de manière dynamique.)

Nous pouvons utiliser le paramètre `N` dans les fonctions associées au type :

```
impl<const N: usize> Polynomial<N> {
    fn new(coefficients: [f64; N]) -> Polynomial<N> {
        Polynomial { coefficients }
    }

    /// Evaluate the polynomial at `x`.
    fn eval(&self, x: f64) -> f64 {
        // Horner's method is numerically stable, efficient, and simple:
        // c0 + x(c1 + x(c2 + x(c3 + ... x(c[n-1] + x c[n]))))
        let mut sum = 0.0;
        for i in (0..N).rev() {
            sum = self.coefficients[i] + x * sum;
        }

        sum
    }
}
```

Ici, la `new` fonction accepte un tableau de longueur `N` et prend ses éléments comme coefficients d'une nouvelle `Polynomial` valeur. La `eval` méthode itère sur la plage `0..N` pour trouver la valeur du polynôme à un point donné `x`.

Comme pour les paramètres de type et de durée de vie, Rust peut souvent déduire les bonnes valeurs pour les paramètres constants :

```
use std::f64::consts::FRAC_PI_2;      // π/2

// Approximate the `sin` function: sin x ≈ x - 1/6 x3 + 1/120 x5
// Around zero, it's pretty accurate!
let sine_poly = Polynomial::new([0.0, 1.0, 0.0, -1.0/6.0, 0.0,
                                 1.0/120.0]);
assert_eq!(sine_poly.eval(0.0), 0.0);
assert!((sine_poly.eval(FRAC_PI_2) - 1.).abs() < 0.005);
```

Puisque nous passons `Polynomial::new` un tableau avec six éléments, Rust sait que nous devons construire un `Polynomial<6>`. La `eval` méthode sait combien d'itérations la `for` boucle doit exécuter simplement en consultant son `Self` type. Comme la longueur est connue au moment de la compilation, le compilateur remplacera probablement entièrement la boucle par du code linéaire.

Un `const` paramètre générique peut être n'importe quel type entier, `char`, ou `bool`. Les nombres à virgule flottante, les énumérations et autres types ne sont pas autorisés.

Si la structure prend d'autres types de paramètres génériques, les paramètres de durée de vie doivent venir en premier, suivis des types, suivis de toutes les `const` valeurs. Par exemple, un type contenant un tableau de références pourrait être déclaré comme ceci :

```
struct LumpOfReferences<'a, T, const N: usize> {
    the_lump:[&'a T; N]
}
```

Les paramètres génériques constants sont un ajout relativement nouveau à Rust, et leur utilisation est quelque peu restreinte pour le moment. Par exemple, il aurait été plus agréable de définir `Polynomial` comme ceci :

```
/// A polynomial of degree N.
struct Polynomial<const N: usize> {
    coefficients:[f64; N + 1]
}
```

Cependant, Rust rejette cette définition :

```
error: generic parameters may not be used in const operations
|
6 |     coefficients: [f64; N + 1]
|                         ^ cannot perform const operation using `N`
|
= help: const parameters may only be used as standalone arguments, i.e.
```

Bien que ce soit bien de le dire `[f64; N]`, un type comme `[f64; N + 1]` est apparemment trop risqué pour Rust. Mais Rust impose cette restriction pour le moment pour éviter de se confronter à des problèmes comme celui-ci :

```

struct Ketchup<const N: usize> {
    tomayto: [i32; N & !31],
    tomahto:[i32; N - (N % 32)],
}

```

En fait, `N & !31` et `N - (N % 32)` sont égaux pour toutes les valeurs de `N`, donc `tomato` et `tomahto` ont toujours le même type. Il devrait être permis d'attribuer l'un à l'autre, par exemple. Mais enseigner au vérificateur de type de Rust l'algèbre de manipulation de bits dont il aurait besoin pour être en mesure de reconnaître ce fait risque d'introduire des cas d'angle déroutants dans un aspect du langage qui est déjà assez compliqué. Bien sûr, des expressions simples comme `N + 1` sont beaucoup plus sages, et des travaux sont en cours pour apprendre à Rust à les gérer en douceur.

Étant donné que le problème ici concerne le comportement du vérificateur de type, cette restriction ne s'applique qu'aux paramètres constants apparaissant dans les types, comme la longueur d'un tableau. Dans une expression ordinaire, vous pouvez utiliser `N` comme bon vous semble : `N + 1` et `N & !31` sont parfaitement acceptables .

Si la valeur que vous souhaitez fournir pour un `const` paramètre générique n'est pas simplement un littéral ou un identifiant unique, vous devez l'entourer d'accolades, comme dans `Polynomial<{5 + 1}>`. Cette règle permet à Rust de signaler les erreurs de syntaxe avec plus de précision.

Dérivation de traits communs pour les types de structure

Structures peut être très simple à écrire :

```

struct Point {
    x: f64,
    y:f64
}

```

Cependant, si vous deviez commencer à utiliser ce `Point` type, vous remarqueriez rapidement que c'est un peu pénible. Comme écrit, `Point` n'est pas copiable ou clonable. Vous ne pouvez pas l'imprimer

```
avec println!("{}:{}", point); et il ne prend pas en charge les opérateurs == et . !=
```

Chacune de ces fonctionnalités a un nom dans Rust— `Copy`, `Clone`, `Debug` et `PartialEq`. On les appelle *traits*. Au [chapitre 11](#), nous montrerons comment implémenter des traits à la main pour vos propres structures. Mais dans le cas de ces traits standard, et de plusieurs autres, vous n'avez pas besoin de les implémenter à la main, sauf si vous souhaitez une sorte de comportement personnalisé. Rust peut les implémenter automatiquement pour vous, avec une précision mécanique. Ajoutez simplement un `#[derive]` attribut à la structure :

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Point {
    x: f64,
    y:f64
}
```

Chacun de ces traits peut être implémenté automatiquement pour une structure, à condition que chacun de ses champs implémente le trait. Nous pouvons demander à Rust de dériver `PartialEq` car `Point` ses deux champs sont tous les deux de type `f64`, qui implémente déjà `PartialEq`.

Rust peut également dériver `PartialOrd`, ce qui ajouterait la prise en charge des opérateurs de comparaison `<`, `>`, `<=` et `>=`. Nous ne l'avons pas fait ici, car comparer deux points pour voir si l'un est "inférieur" à l'autre est en fait une chose assez étrange à faire. Il n'y a pas d'ordre conventionnel sur les points. Nous choisissons donc de ne pas prendre en charge ces opérateurs pour les `Point` valeurs. Des cas comme celui-ci sont l'une des raisons pour lesquelles Rust nous fait écrire l' `#[derive]` attribut plutôt que de dériver automatiquement tous les traits possibles. Une autre raison est que l'implémentation d'un trait est automatiquement une fonctionnalité publique, donc la copiabilité, la clonage, etc. font toutes partie de l'API publique de votre structure et doivent être choisies délibérément.

Nous décrirons en détail les traits standard de Rust et expliquerons les- quels sont `#[derive]` capables au [chapitre 13](#).

Mutabilité intérieure

Mutabilité c'est comme n'importe quoi d'autre : en excès, ça cause des problèmes, mais on en veut souvent juste un peu. Par exemple, supposons que votre système de contrôle de robot araignée ait une structure centrale, `SpiderRobot`, qui contient des paramètres et des poignées d'E/S. Il est configuré au démarrage du robot et les valeurs ne changent jamais :

```
pub struct SpiderRobot {
    species: String,
    web_enabled: bool,
    leg_devices: [fd::FileDesc; 8],
    ...
}
```

Chaque système majeur du robot est géré par une structure différente, et chacun a un pointeur vers `SpiderRobot` :

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

pub struct SpiderSenses {
    robot: Rc<SpiderRobot>, // <-- pointer to settings and I/O
    eyes: [Camera; 32],
    motion: Accelerometer,
    ...
}
```

Les structures pour la construction de sites Web, la préation, le contrôle du flux de venin, etc. ont également toutes un `Rc<SpiderRobot>` pointeur intelligent. Rappel qui `Rc` signifie [comptage de références](#), et une valeur dans une `Rc` boîte est toujours partagée et donc toujours immuable.

Supposons maintenant que vous souhaitez ajouter un peu de journalisation à la `SpiderRobot` structure, en utilisant le `File` type standard. Il y a un problème : `File` a doit être `mut`. Toutes les méthodes pour y écrire nécessitent une `mut` référence.

Ce genre de situation revient assez souvent. Ce dont nous avons besoin, c'est d'un peu de données modifiables (à `File`) à l'intérieur d'une valeur autrement immuable (la `SpiderRobot` structure). C'est ce qu'on appelle *la mutabilité intérieure*. La rouille en offre plusieurs saveurs ; dans cette section, nous aborderons les deux types les plus simples : `Cell<T>` et `RefCell<T>`, tous deux dans le `std::cell` module.

A `Cell<T>` est une structure qui contient une seule valeur privée de type `T`. La seule particularité de la `Cell` est que vous pouvez obtenir et définir le champ même si vous n'avez pas `mut` accès à lui- `Cell` même :

`Cell::new(value)`

Crée un nouveau `Cell`, en y déplaçant le donné `value`.

`cell.get()`

Renvoie une copie de la valeur dans le fichier `cell`.

`cell.set(value)`

Stocke le donné `value` dans le `cell`, en supprimant la valeur précédemment stockée.

Cette méthode prend `self` comme non `mut` référence :

```
fn set(&self, value:T)      // note: not `&mut self`
```

Ceci est, bien sûr, inhabituel pour les méthodes nommées `set`. À l'heure actuelle, Rust nous a appris à nous attendre à ce que nous ayons besoin d'`mut` un accès si nous voulons apporter des modifications aux données. Mais du même coup, ce détail inhabituel est tout l'intérêt de l'`Cell` art. Ils sont simplement un moyen sûr de contourner les règles d'immuabilité, ni plus, ni moins.

Les cellules ont également quelques autres méthodes, que vous pouvez lire [dans la documentation](#).

A `Cell` serait pratique si vous ajoutiez un simple compteur à votre fichier `SpiderRobot`. Vous pourriez écrire :

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count:Cell<u32>,
    ...
}
```

Ensuite, même les non-`mut` méthodes de `SpiderRobot` peuvent accéder à cela `u32`, en utilisant les méthodes `.get()` et `.set()`

```
impl SpiderRobot {
    /// Increase the error count by 1.
```

```

pub fn add_hardware_error(&self) {
    let n = self.hardware_error_count.get();
    self.hardware_error_count.set(n + 1);
}

/// True if any hardware errors have been reported.
pub fn has_hardware_errors(&self) ->bool {
    self.hardware_error_count.get() > 0
}
}

```

C'est assez simple, mais cela ne résout pas notre problème de journalisation. `Cell` ne vous permet *pas* d'appeler des `mut` méthodes sur une valeur partagée. La `.get()` méthode renvoie une copie de la valeur dans la cellule, donc cela ne fonctionne que si `T` implémente [le Copy trait](#). Pour la journalisation, nous avons besoin d'un mutable `File`, et `File` n'est pas copiable.

Le bon outil dans ce cas est un `RefCell`. Comme `Cell<T>`, `RefCell<T>` est un type générique qui contient une seule valeur de type `T`. Contrairement à `Cell`, prend en `RefCell` charge les références d'emprunt à sa `T` valeur :

`RefCell::new(value)`

Crée un nouveau `RefCell`, emménageant `value` dedans.

`ref_cell.borrow()`

Retourne `Ref<T>`, qui est essentiellement juste une référence partagée à la valeur stockée dans `ref_cell`.

Cette méthode panique si la valeur est déjà empruntée de manière mutable ; voir les détails à suivre.

`ref_cell.borrow_mut()`

Retourne `RefMut<T>`, essentiellement une référence mutable à la valeur dans `ref_cell`.

Cette méthode panique si la valeur est déjà empruntée ; voir les détails à suivre.

`ref_cell.try_borrow(), ref_cell.try_borrow_mut()`

Travailler comme `borrow()` et `borrow_mut()`, mais renvoie un `Result`. Au lieu de paniquer si la valeur est déjà empruntée de manière mutable, ils renvoient une `Err` valeur.

Encore une fois, `RefCell` a quelques autres méthodes, que vous pouvez trouver [dans la documentation](#).

Les deux `borrow` méthodes ne paniquent que si vous essayez d'enfreindre la règle de Rust selon laquelle `mut` les références sont des références exclusives. Par exemple, cela ferait paniquer :

```
use std::cell::RefCell;

let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();           // ok, returns a Ref<String>
let count = r.len();                // ok, returns "hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();   // panic: already borrowed
w.push_str(" world");
```

Pour éviter de paniquer, vous pouvez mettre ces deux emprunts dans des blocs séparés. De cette façon, `r` serait abandonné avant d'essayer d'emprunter `w`.

Cela ressemble beaucoup au fonctionnement des références normales. La seule différence est que normalement, lorsque vous empruntez une référence à une variable, Rust vérifie *au moment de la compilation* pour s'assurer que vous utilisez la référence en toute sécurité. Si les vérifications échouent, vous obtenez une erreur de compilation. `RefCell` applique la même règle à l'aide de contrôles d'exécution. Donc, si vous enfreignez les règles, vous obtenez une panique (ou un `Err`, pour `try_borrow` et `try_borrow_mut`).

Nous sommes maintenant prêts `RefCell` à travailler dans notre `SpiderRobot` type :

```
pub struct SpiderRobot {

    ...
    log_file:RefCell<File>,
    ...
}

impl SpiderRobot {
    /// Write a line to the log file.
    pub fn log(&self, message:&str) {
        let mut file = self.log_file.borrow_mut();
        // `writeln!` is like `println!`, but sends
```

```
// output to the given file.  
writeln!(file, "{}", message).unwrap();  
}  
}
```

La variable `file` est de type `RefMut<File>`. Il peut être utilisé comme une référence mutable à un fichier `File`. Pour plus de détails sur l'écriture dans des fichiers, voir [Chapitre 18](#).

Les cellules sont faciles à utiliser. Devoir appeler `.get()` et `.set()` ou `.borrow()` et `.borrow_mut()` est un peu gênant, mais c'est juste le prix à payer pour contourner les règles. L'autre inconvénient est moins évident et plus grave : les cellules (et tous les types qui en contiennent) ne sont pas thread-safe. Rust [ne permettra](#) donc pas à plusieurs threads d'y accéder à la fois. Nous décrirons les variantes thread-safe de la mutabilité intérieure au [chapitre 19](#), lorsque nous aborderons « [Mutex<T>](#) », « [Aomics](#) » et « [Global Variables](#) ».

Qu'une structure ait des champs nommés ou qu'elle ressemble à un tuple, il s'agit d'une agrégation d'autres valeurs : si j'ai une `SpiderSenses` structure, alors j'ai un `Rc` pointeur vers une structure partagée `SpiderRobot`, et j'ai des yeux, et j'ai un accéléromètre, etc.. Ainsi, l'essence d'une structure est le mot « et » : j'ai un X *et* un Y. Mais que se passerait-il s'il y avait un autre type de type construit autour du mot « ou » ? Autrement dit, lorsque vous avez une valeur d'un tel type, vous auriez *soit* un X, *soit* un Y ? De tels types s'avèrent si utiles qu'ils sont omniprésents dans Rust, et ils font l'objet du chapitre suivant.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 10. Énumérations et modèles

Étonnant à quel point les trucs informatiques ont du sens vus comme une privation tragique de types de somme (cf. privation de lambdas).

—[Graydon Hoare](#)

Le premier sujet de ce chapitre est puissant, aussi vieux que les collines, heureux de vous aider à accomplir beaucoup de choses en peu de temps (pour un prix), et connu sous de nombreux noms dans de nombreuses cultures. Mais ce n'est pas le diable. C'est une sorte de type de données défini par l'utilisateur, connu depuis longtemps des pirates ML et Haskell sous le nom de types de somme, d'unions discriminées ou de types de données algébriques. Dans Rust, on les appelle *des énumérations* ou simplement *des énumérations*. Contrairement au diable, ils sont tout à fait en sécurité, et le prix qu'ils demandent n'est pas une grande privation.

C++ et C# ont des énumérations; vous pouvez les utiliser pour définir votre propre type dont les valeurs sont un ensemble de constantes nommées. Par exemple, vous pouvez définir un type nommé `Color` avec les valeurs `Red`, `Orange`, `Yellow`, etc. Ce type d'énumération fonctionne également dans Rust. Mais Rust va beaucoup plus loin dans les énumérations. Une énumération Rust peut également contenir des données, même des données de différents types. Par exemple, le type de Rust `Result<String, io::Error>` est une énumération ; une telle valeur est soit une `Ok` valeur contenant un `String` soit une `Err` valeur contenant un `io::Error`. C'est au-delà de ce que les énumérations C++ et C# peuvent faire. Cela ressemble plus à un C `union` - mais contrairement aux unions, les énumérations Rust sont de type sécurisé.

Les énumérations sont utiles chaque fois qu'une valeur peut être une chose ou une autre. Le « prix » de leur utilisation est que vous devez accéder aux données en toute sécurité, en utilisant le pattern matching, notre sujet pour la seconde moitié de ce chapitre.

Les modèles peuvent également vous être familiers si vous avez utilisé la décompression en Python ou la déstructuration en JavaScript, mais Rust va plus loin. Les modèles de rouille sont un peu comme des expressions régulières pour toutes vos données. Ils sont utilisés pour tester si oui ou non une valeur a une forme particulière souhaitée. Ils peuvent extraire simultanément plusieurs champs d'une structure ou d'un tuple dans des

variables locales. Et comme les expressions régulières, elles sont concises, faisant généralement tout cela en une seule ligne de code.

Ce chapitre commence par les bases des énumérations, montrant comment les données peuvent être associées à des variantes d'énumération et comment les énumérations sont stockées en mémoire. Ensuite, nous montrerons comment les modèles et les `match` instructions de Rust peuvent spécifier de manière concise une logique basée sur des énumérations, des structures, des tableaux et des tranches. Les modèles peuvent également inclure des références, des mouvements et `if` des conditions, ce qui les rend encore plus performants.

Énumérations

Énumérations simples de style C sont simples :

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

Cela déclare un type `Ordering` avec trois valeurs possibles, appelées *variantes* ou *constructeurs* : `Ordering::Less`, `Ordering::Equal` et `Ordering::Greater`. Cette énumération particulière fait partie de la bibliothèque standard, donc le code Rust peut l'importer, soit par lui-même :

```
use std::cmp::Ordering;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Ordering:: Less
    } else if n > m {
        Ordering:: Greater
    } else {
        Ordering:: Equal
    }
}
```

ou avec tous ses constructeurs :

```
use std::cmp:: Ordering::{self, *};      // `*` to import all children

fn compare(n: i32, m: i32) -> Ordering {
```

```

if n < m {
    Less
} else if n > m {
    Greater
} else {
    Equal
}
}

```

Après avoir importé les constructeurs, nous pouvons écrire à la Less place de `Ordering::Less`, et ainsi de suite, mais comme c'est moins explicite, il est généralement préférable de *ne pas* les importer, sauf lorsque cela rend votre code beaucoup plus lisible.

Pour importer les constructeurs d'une énumération déclarée dans le module courant, utilisez un `self` import :

```

enum Pet {
    Orca,
    Giraffe,
    ...
}

use self:: Pet::*;

```

En mémoire, les valeurs des énumérations de style C sont stockées sous forme d'entiers. Parfois, il est utile d'indiquer à Rust quels entiers utiliser :

```

enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}

```

Sinon, Rust attribuera les numéros pour vous, en commençant par 0.

Par défaut, Rust stocke les énumérations de style C en utilisant le plus petit type entier intégré qui peut les accueillir. La plupart tiennent dans un seul octet :

```

use std:: mem:: size_of;
assert_eq!(size_of:: <Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2); // 404 doesn't fit in a u8

```

Vous pouvez remplacer le choix de Rust de représentation en mémoire en ajoutant un `#[repr]` attribut à l'énumération. Pour plus de détails, voir « [Recherche de représentations de données communes](#) ».

La conversion d'une énumération de style C en un entier est autorisée :

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

Cependant, la conversion dans l'autre sens, de l'entier à l'énumération, ne l'est pas. Contrairement à C et C++, Rust garantit qu'une valeur enum n'est jamais qu'une des valeurs énoncées dans la `enum` déclaration. Un transty-page non vérifié d'un type entier vers un type enum pourrait rompre cette garantie, il n'est donc pas autorisé. Vous pouvez soit écrire votre propre conversion vérifiée :

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None,
    }
}
```

ou utilisez [la enum primitive caisse](#). Il contient une macro qui génère automatiquement ce type de code de conversion pour vous.

Comme pour les structures, le compilateur implémentera des fonctionnalités telles que l' `==` opérateur pour vous, mais vous devez demander :

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years,
}
```

Les énumérations peuvent avoir des méthodes, tout comme les structures :

```
impl TimeUnit {
    /// Return the plural noun for this time unit.
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
            TimeUnit::Days => "days",
            TimeUnit::Months => "months",
            TimeUnit::Years => "years",
        }
    }
}
```

```

        TimeUnit:: Hours => "hours",
        TimeUnit:: Days => "days",
        TimeUnit:: Months => "months",
        TimeUnit::Years => "years",
    }
}

/// Return the singular noun for this time unit.
fn singular(self) ->&'static str {
    self.plural().trim_end_matches('s')
}
}

```

Voilà pour les énumérations de style C. Le type d'énumération Rust le plus intéressant est celui dont les variantes contiennent des données. Nous montrerons comment ceux-ci sont stockés en mémoire, comment les rendre génériques en ajoutant des paramètres de type et comment construire des structures de données complexes à partir d'énumérations..

Énumérations avec des données

Certains programmes toujours besoin d'afficher des dates et des heures complètes jusqu'à la milliseconde, mais pour la plupart des applications, il est plus convivial d'utiliser une approximation approximative, comme "il y a deux mois". Nous pouvons écrire une énumération pour aider à cela, en utilisant l'énumération définie précédemment :

```

/// A timestamp that has been deliberately rounded off, so our program
/// says "6 months ago" instead of "February 9, 2016, at 9:49 AM".
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}

```

Deux des variantes de cette énumération, `InThePast` et `InTheFuture`, prennent des arguments. Celles-ci sont appelées *variantes de tuple*. Comme les structures de tuple, ces constructeurs sont des fonctions qui créent de nouvelles `RoughTime` valeurs :

```

let four_score_and_seven_years_ago =
    RoughTime:: InThePast(TimeUnit::Years, 4 * 20 + 7);

let three_hours_from_now =
    RoughTime:: InTheFuture(TimeUnit::Hours, 3);

```

Les énumérations peuvent également avoir des *variantes* de structure , qui contiennent des champs nommés, tout comme les structures ordinaires :

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2:Point3d },
}

let unit_sphere = Shape:: Sphere {
    center: ORIGIN,
    radius:1.0,
};
```

En tout, Rust a trois types de variantes enum, faisant écho aux trois types de struct que nous avons montrés dans le chapitre précédent. Les variantes sans données correspondent à des structures de type unité. Les variantes de tuple ressemblent et fonctionnent comme des structures de tuple. Les variantes de structure ont des accolades et des champs nommés. Une seule énumération peut avoir des variantes des trois types :

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr:EarlyModernistPoem,
    },
}
```

Tous les constructeurs et champs d'une énumération partagent la même visibilité que l'énumération elle-même.

Énumérations en mémoire

En mémoire, les énumérations contenant des données sont stockées sous la forme d'une petite *balise* entière , plus suffisamment de mémoire pour contenir tous les champs de la plus grande variante. Le champ tag est destiné à l'usage interne de Rust. Il indique quel constructeur a créé la valeur et donc quels champs elle a.

Depuis Rust 1.56, RoughTime tient dans 8 octets, comme illustré à la [Figure 10-1](#) .

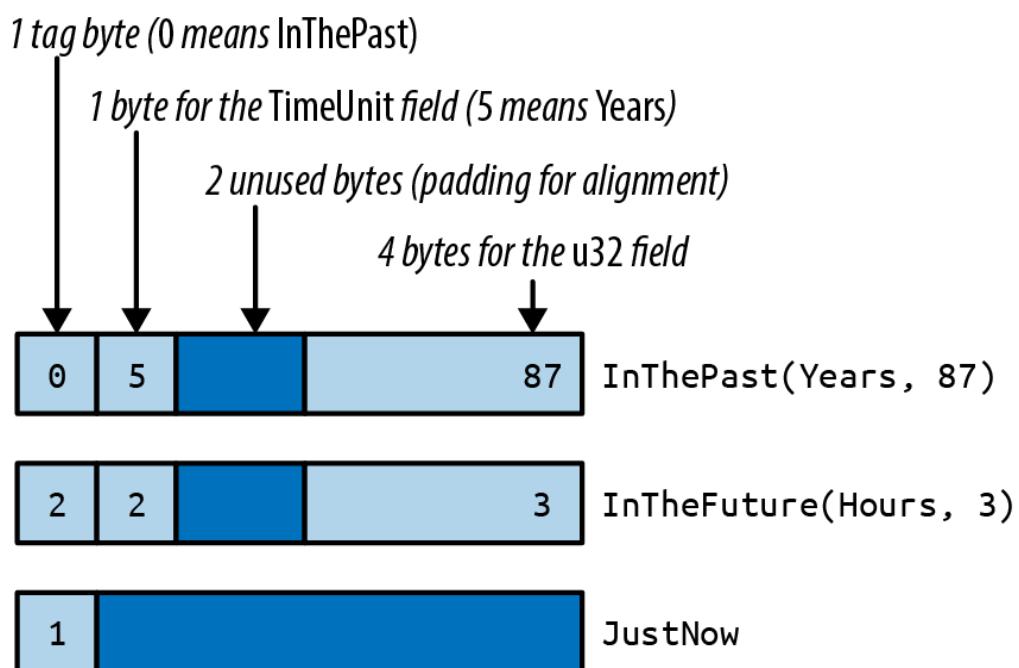


Illustration 10-1. RoughTime valeurs en mémoire

Cependant, Rust ne fait aucune promesse concernant la disposition des enums, afin de laisser la porte ouverte à de futures optimisations. Dans certains cas, il serait possible d'emballer une énumération plus efficacement que ne le suggère la figure. Par exemple, certaines structures générées peuvent être stockées sans balise, comme nous le verrons plus tard.

Structures de données riches à l'aide d'énumérations

Énumérations sont également utiles pour implémenter rapidement des structures de données arborescentes. Par exemple, supposons qu'un programme Rust doive travailler avec des données JSON arbitraires. En mémoire, tout JSON document peut être représenté comme une valeur de ce type Rust :

```
use std::collections::HashMap;

enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>),
}
```

L'explication de cette structure de données en anglais ne peut pas beaucoup améliorer le code Rust. La norme JSON spécifie les différents types

de données qui peuvent apparaître dans un document JSON : `null`, valeurs booléennes, nombres, chaînes, tableaux de valeurs JSON et objets avec des clés de chaîne et des valeurs JSON. L' `Json` énumération énonce simplement ces types.

Ceci n'est pas un exemple hypothétique. Une énumération très similaire peut être trouvée dans `serde_json`, une sérialisation bibliothèque pour les structures Rust qui est l'une des caisses les plus téléchargées sur crates.io.

Le `Box` autour de `HashMap` qui représente un `Object` sert uniquement à rendre toutes les `Json` valeurs plus compactes. En mémoire, les valeurs de type `Json` occupent quatre mots machine. `String` et `Vec` les valeurs sont trois mots, et Rust ajoute un octet de balise. `Null` et `Boolean` les valeurs ne contiennent pas suffisamment de données pour utiliser tout cet espace, mais toutes les `Json` valeurs doivent avoir la même taille. L'espace supplémentaire n'est pas utilisé. [La figure 10-2](#) montre quelques exemples de l'`Json` apparence réelle des valeurs en mémoire.

A `HashMap` est encore plus grand. Si nous devions lui laisser de la place dans chaque `Json` valeur, elles seraient assez grandes, huit mots environ. Mais un `Box<HashMap>` est un mot unique : c'est juste un pointeur vers des données allouées par tas. On pourrait rendre `Json` encore plus compact en boxant plus de champs.

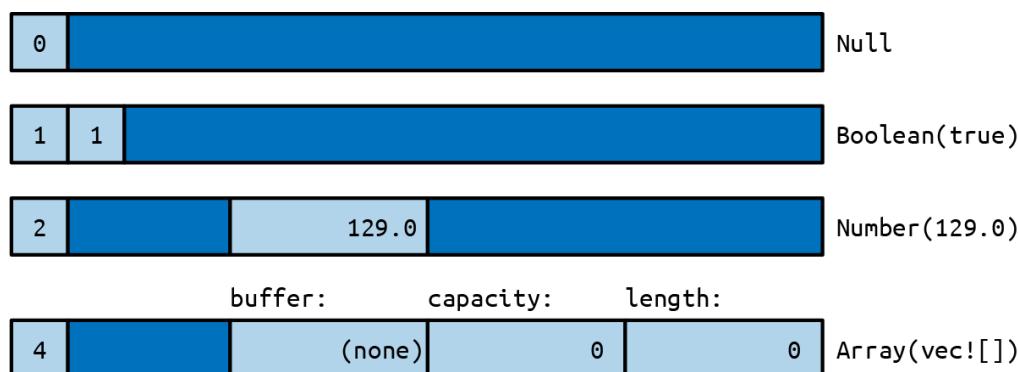


Illustration 10-2. `Json` valeurs en mémoire

Ce qui est remarquable ici, c'est la facilité avec laquelle il a été mis en place. En C++, on pourrait écrire une classe pour cela :

```
class JSON {
private:
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
```

```

        double number;
        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;

    Data() {}
    ~Data() {}
    ...
};

Tag tag;
Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON(); // clean up string/array/object value
        tag = Boolean;
        data.boolean = value;
    }
    ...
};

```

A 30 lignes de code, nous avons à peine commencé le travail. Cette classe aura besoin de constructeurs, d'un destructeur et d'un opérateur d'affectation. Une alternative serait de créer une hiérarchie de classes avec une classe de base `JSON` et des sous-classes `JSONBoolean`, `JSONString`, etc. Quoi qu'il en soit, quand ce sera fait, notre bibliothèque C++ JSON aura plus d'une douzaine de méthodes. Il faudra un peu de lecture pour que d'autres programmeurs le prennent et l'utilisent. L'énumération complète de Rust est de huit lignes de code.

Énumérations génériques

Énumérations peuvent être générique. Deux exemples de la bibliothèque standard font partie des types de données les plus utilisés dans le langage :

```

enum Option<T> {
    None,
    Some(T),
}

```

```

enum Result<T, E> {
    Ok(T),
    Err(E),
}

```

Ces types sont déjà assez familiers et la syntaxe des énumérations génériques est la même que celle des structures génériques.

Un détail non évident est que Rust peut éliminer le champ de balise `Option<T>` lorsque le type `T` est une référence, `Box` ou un autre type de pointeur intelligent. Puisqu'aucun de ces types de pointeurs n'est autorisé à être nul, Rust peut représenter `Option<Box<i32>>`, par exemple, comme un seul mot machine : 0 pour `None` et différent de zéro pour `Some` pointeur. Cela fait de ces `Option` types des analogues proches des valeurs de pointeur C ou C++ qui pourraient être nulles. La différence est que le système de type de Rust vous oblige à vérifier que `an Option` est `Some` avant de pouvoir utiliser son contenu. Cela élimine efficacement les déréférencements de pointeur nul.

Les structures de données génériques peuvent être construites avec seulement quelques lignes de code :

```

// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}

```

Ces quelques lignes de code définissent un `BinaryTree` type qui peut stocker n'importe quel nombre de valeurs de type `T`.

Une grande quantité d'informations est contenue dans ces deux définitions, nous prendrons donc le temps de traduire mot à mot le code en anglais. Chaque `BinaryTree` valeur est soit `Empty` ou `NonEmpty`. Si c'est `Empty`, alors il ne contient aucune donnée. Si `NonEmpty`, alors il a un `Box`, un pointeur vers un tas alloué `TreeNode`.

Chaque `TreeNode` valeur contient un élément réel, ainsi que deux autres `BinaryTree` valeurs. Cela signifie qu'un arbre peut contenir des sous-

arbres, et donc un `NonEmpty` arbre peut avoir n'importe quel nombre de descendants.

Une esquisse d'une valeur de type `BinaryTree<&str>` est illustrée à la [Figure 10-3](#). Comme avec `Option<Box<T>>`, Rust élimine le champ de balise, donc une `BinaryTree` valeur n'est qu'un mot machine.

Construire un nœud particulier dans cet arbre est simple :

```
use self:: BinaryTree:: *;
let jupiter_tree = NonEmpty(Box:: new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

De plus grands arbres peuvent être construits à partir de plus petits :

```
let mars_tree = NonEmpty(Box:: new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
    right: mercury_tree,
}));
```

Naturellement, cette affectation transfère la propriété de `jupiter_node` et `mercury_node` à leur nouveau nœud parent.

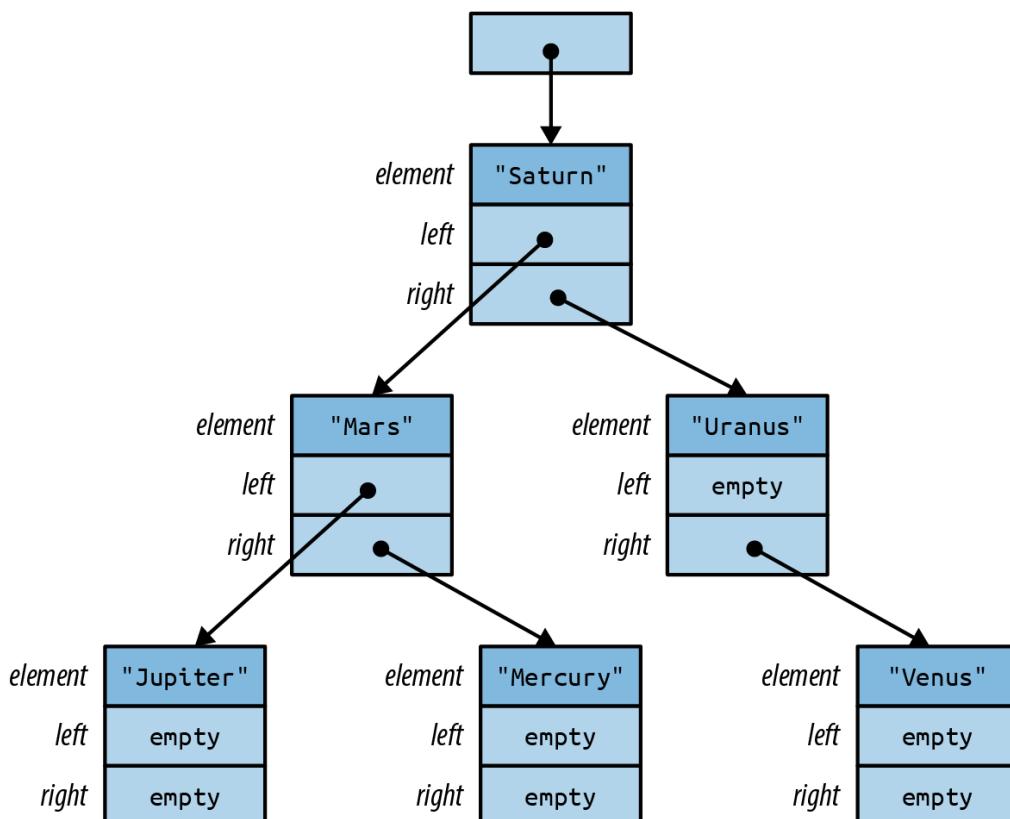


Illustration 10-3. A `BinaryTree` contenant six cordes

Les autres parties de l'arbre suivent les mêmes modèles. Le nœud racine n'est pas différent des autres :

```
let tree = NonEmpty(Box:: new(TreeNode {  
    element: "Saturn",  
    left: mars_tree,  
    right: uranus_tree,  
}));
```

Plus loin dans ce chapitre, nous montrerons comment implémenter une `add` méthode sur le `BinaryTree` type pour que l'on puisse plutôt écrire :

```
let mut tree = BinaryTree::Empty;  
for planet in planets {  
    tree.add(planet);  
}
```

Quelle que soit la langue d'où vous venez, la création de structures de données comme `BinaryTree` dans Rust nécessitera probablement un peu de pratique. Il ne sera pas évident au début où mettre les `Box` es. Une façon de trouver une conception qui fonctionnera consiste à dessiner une image comme la [figure 10-3](#) qui montre comment vous voulez que les choses soient disposées en mémoire. Revenez ensuite de l'image au code. Chaque collection de rectangles est une structure ou un tuple ; chaque flèche est un `Box` ou un autre pointeur intelligent. Déterminer le type de chaque champ est un peu un casse-tête, mais gérable. La récompense pour résoudre le puzzle est le contrôle de l'utilisation de la mémoire de votre programme.

Vient maintenant le "prix" que nous avons mentionné dans l'introduction. Le champ `tag` d'une énumération coûte un peu de mémoire, jusqu'à huit octets dans le pire des cas, mais c'est généralement négligeable. Le véritable inconvénient des énumérations (si on peut l'appeler ainsi) est que le code Rust ne peut pas faire preuve de prudence et essayer d'accéder aux champs, qu'ils soient ou non réellement présents dans la valeur :

```
let r = shape.radius; // error: no field `radius` on type `Shape`
```

Le seul moyen d'accéder aux données d'une énumération est le moyen sûr : utiliser des modèles.

Motifs

Rappel de la définition de notre `RoughTime` type au début de ce chapitre :

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

Supposons que vous ayez une `RoughTime` valeur et que vous souhaitez l'afficher sur une page Web. Vous devez accéder aux champs `TimeUnit` et `u32` à l'intérieur de la valeur. Rust ne vous permet pas d'y accéder directement, en écrivant `rough_time.0` et `rough_time.1`, car après tout, la valeur pourrait être `RoughTime::JustNow`, qui n'a pas de champs. Mais alors, comment sortir les données ?

Vous avez besoin d'une `match` expression:

```
1 fn rough_time_to_english(rt: RoughTime) -> String {
2     match rt {
3         RoughTime:: InThePast(units, count) =>
4             format!("{} {} ago", count, units.plural()),
5         RoughTime:: JustNow =>
6             format!("just now"),
7         RoughTime::InTheFuture(units, count) =>
8             format!("{} {} from now", count, units.plural()),
9     }
10 }
```

`match` effectue une correspondance de modèle ; dans cet exemple, les *modèles* sont les parties qui apparaissent avant le `=>` symbole sur les lignes 3, 5 et 7. Les modèles qui correspondent aux `RoughTime` valeurs ressemblent aux expressions utilisées pour créer des `RoughTime` valeurs. Ce n'est pas un hasard. Les expressions *produisent des* valeurs ; les modèles *consomment des* valeurs. Les deux utilisent beaucoup de la même syntaxe.

Passons en revue ce qui se passe lorsque cette `match` expression s'exécute. Supposons que `rt` la valeur

`RoughTime::InTheFuture(TimeUnit::Months, 1)`. Rust essaie d'abord de faire correspondre cette valeur avec le modèle de la ligne 3. Comme vous pouvez le voir sur la [figure 10-4](#), cela ne correspond pas.

```
value: RoughTime::InTheFuture(TimeUnit::Months, 1)
```



```
pattern: RoughTime::InThePast(units, count)
```

Illustration 10-4. Une RoughTime valeur et un modèle qui ne correspondent pas

Le motif correspondant à une énumération, une structure ou un tuple fonctionne comme si Rust effectuait un simple balayage de gauche à droite, vérifiant chaque composant du motif pour voir si la valeur lui correspond. Si ce n'est pas le cas, Rust passe au motif suivant.

Les motifs des lignes 3 et 5 ne correspondent pas. Mais le motif de la ligne 7 réussit ([Figure 10-5](#)).

```
value: RoughTime::InTheFuture(TimeUnit::Months, 1)
```



```
pattern: RoughTime::InTheFuture(
```



```
        units, count)
```

Illustration 10-5. Un match réussi

Lorsqu'un modèle contient des identifiants simples comme `units` et `count`, ceux-ci deviennent des variables locales dans le code suivant le modèle. Tout ce qui est présent dans la valeur est copié ou déplacé dans les nouvelles variables. Rust stocke `TimeUnit::Months` dans `units` et 1 dans `count`, exécute la ligne 8 et renvoie la chaîne "1 months from now".

Cette sortie a un problème grammatical mineur, qui peut être résolu en ajoutant un autre bras au `match`:

```
RoughTime::InTheFuture(unit, 1) =>
    format!("a {} from now", unit.singular()),
```

Ce bras ne correspond que si le `count` champ est exactement 1. Notez que ce nouveau code doit être ajouté avant la ligne 7. Si nous l'ajoutons à la fin, Rust n'y arrivera jamais, car le modèle de la ligne 7 correspond à toutes les `InTheFuture` valeurs. Le compilateur Rust vous avertira d'un "modèle inaccessible" si vous faites ce genre d'erreur.

Même avec le nouveau code,

`RoughTime::InTheFuture(TimeUnit::Hours, 1)` présente toujours un problème : le résultat "a hour from now" n'est pas tout à fait cor-

rect. Telle est la langue anglaise. Cela aussi peut être corrigé en ajoutant un autre bras au `match`.

Comme le montre cet exemple, la correspondance de modèles fonctionne de pair avec les énumérations et peut même tester les données qu'elles contiennent, ce qui constitue `match` un remplacement puissant et flexible de l'`switch` instruction C.. Jusqu'à présent, nous n'avons vu que des modèles qui correspondent à des valeurs enum. Il y a plus que cela. Les motifs de rouille sont leur propre petit langage, résumé dans le [ta-bleau 10-1](#). Nous consacrerons la majeure partie du reste du chapitre aux fonctionnalités présentées dans ce tableau.

Tableau 10-1. Motifs

Type de motif	Exemple	Remarques
Littéral	100 "name"	Correspond à une valeur exacte ; le nom d'un <code>const</code> est également autorisé
Intervalle	0 ..= 100 'a' ..= 'k' 256..	Correspond à n'importe quelle valeur dans la plage, y compris la valeur de fin si elle est donnée
Caractère générique	_	Correspond à n'importe quelle valeur et l'ignore
Variable	name <code>mut count</code>	Comme <code>_</code> mais déplace ou copie la valeur dans une nouvelle variable locale
ref variable	ref field ref mut fi eld	Emprunte une référence à la valeur correspondante au lieu de la déplacer ou de la copier
Reliure avec sous-motif	val @ 0 .. = 99 ref circle @ Shape::Ci rcle { .. }	Correspond au modèle à droite de <code>@</code> , en utilisant le nom de la variable à gauche
Modèle d'énumération	Some(valu e) None Pet::Orca	
Modèle de tuple	(key, valu e) (r, g, b)	
Modèle de tableau	[a, b, c, d, e, f, g] [heading, carom, corr ection]	

Type de motif	Exemple	Remarques
Modèle de tranche	[first, se cond] [first, _, third] [first, ..., nth] []	
Modèle de structure	Color(r, g, b) Point { x, y } Card { suit: Clubs, rank: n } Account { id, name, ... }	
Référence	&value &(k, v)	Ne correspond qu'aux valeurs de référence
Ou des motifs	'a' 'A' Some("left" "right")	
Expression de garde	x if x * x <= r2	En match seulement (non valide en let, etc.)

Littéraux, variables et caractères génériques dans les modèles

Jusqu'à présent, nous avons montré `match` des expressions fonctionnant avec des énumérations. D'autres types peuvent également être assortis. Lorsque vous avez besoin de quelque chose comme une `switch` instruction C, à utiliser `match` avec une valeur entière. Littéraux entiersaiement 0 et 1 peuvent servir de motifs :

```
match meadow.count_rabbits() {
  0 => {} // nothing to say
  1 => println!("A rabbit is nosing around in the clover."),
  n => println!("There are {} rabbits hopping about in the meadow", n),
}
```

Le motif `0` correspond s'il n'y a pas de lapins dans le pré. `1` correspond s'il n'y en a qu'un. S'il y a deux lapins ou plus, nous atteignons le troisième modèle, `n`. Ce modèle n'est qu'une variableNom. Il peut correspondre à n'importe quelle valeur et la valeur correspondante est déplacée ou copiée dans une nouvelle variable locale. Donc dans ce cas, la valeur de `meadow.count_rabbits()` est stockée dans une nouvelle variable locale `n`, que nous imprimons ensuite.

D'autres littéraux peuvent également être utilisés comme modèles, y compris les booléens, les caractères et même les chaînes :

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar:: Gregorian,
    "chinese" => Calendar:: Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other => return parse_error("calendar", other),
};
```

Dans cet exemple, `other` sert de modèle fourre-tout comme `n` dans l'exemple précédent. Ces modèles jouent le même rôle qu'un default cas dans une `switch` instruction, correspondant à des valeurs qui ne correspondent à aucun des autres modèles.

Si vous avez besoin d'un modèle fourre-tout, mais que vous ne vous souciez pas de la valeur correspondante, vous pouvez utiliser un seul trait de soulignement `_` comme modèle, le *modèle générique*:

```
let caption = match photo.tagged_pet() {
    Pet:: Tyrannosaur => "RRRAAAAHHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // generic caption, works for any pet
};
```

Le modèle de caractère générique correspond à n'importe quelle valeur, mais sans le stocker nulle part. Étant donné que Rust nécessite que chaque `match` expression gère toutes les valeurs possibles, un caractère générique est souvent requis à la fin. Même si vous êtes certain que les cas restants ne peuvent pas se produire, vous devez au moins ajouter un bras de secours, peut-être un qui panique :

```
// There are many Shapes, but we only support "selecting"
// either some text, or everything in a rectangular area.
// You can't select an ellipse or trapezoid.
match document.selection() {
    Shape:: TextSpan(start, end) => paint_text_selection(start, end),
```

```

    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type"),
}

```

Modèles de tuple et de structure

Modèles de tuple correspondent à des tuples. Ils sont utiles chaque fois que vous souhaitez impliquer plusieurs éléments de données dans un seul `match` :

```

fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp:: Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else",
    }
}

```

Structureles modèles utilisent des accolades, tout comme les expressions `struct`. Ils contiennent un sous-modèle pour chaque champ :

```

match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y:y } =>
        println!("at ({}m, {}m)", x, y),
}

```

Dans cet exemple, si le premier bras correspond, alors `balloon.location.y` est stocké dans la nouvelle variable locale `height`.

Supposons `balloon.location` que `Point { x: 30, y: 40 }`. Comme toujours, Rust vérifie tour à tour chaque composant de chaque motif [Figure 10-6](#).

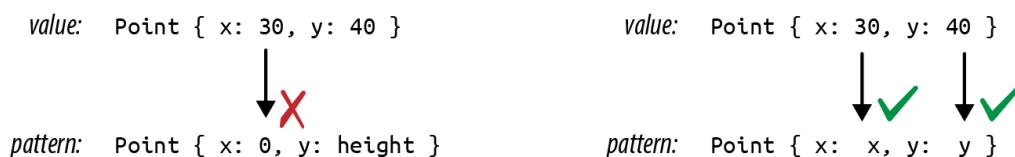


Illustration 10-6. Correspondance de modèles avec des structures

Le deuxième bras correspond, donc la sortie serait at (30m, 40m) .

Les modèles comme `Point { x: x, y: y }` sont courants lors de la correspondance des structures, et les noms redondants sont un encombrement visuel, donc Rust a un raccourci pour cela : `Point {x, y}` . Le sens est le même. Ce modèle stocke toujours le `x` champ d'un point dans un nouveau local `x` et son `y` champ dans un nouveau local `y` .

Même avec la sténographie, il est fastidieux de faire correspondre une grande structure lorsque nous ne nous soucions que de quelques champs :

```
match get_account(id) {
    ...
    Some(Account {
        name, language, // <--- the 2 things we care about
        id: _, status: _, address: _, birthday: _, eye_color: _,
        pet: _, security_question: _, hashed_innermost_secret: _,
        is_adamantium_preferred_customer: _, }) =>
        language.show_custom_greeting(name),
    }
}
```

Pour éviter cela, utilisez `..` pour indiquer à Rust que vous ne vous souciez d'aucun des autres champs :

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name),
```

Modèles de tableau et de tranche

Déployer les motifs correspondent aux tableaux. Ils sont souvent utilisés pour filtrer certaines valeurs de cas particuliers et sont utiles chaque fois que vous travaillez avec des tableaux dont les valeurs ont une signification différente en fonction de la position.

Par exemple, lors de la conversion des valeurs de couleur de teinte, de saturation et de luminosité (HSL) en valeurs de couleur rouge, vert, bleu (RVB), les couleurs avec une luminosité nulle ou totale sont simplement noires ou blanches. Nous pourrions utiliser une `match` expression pour traiter ces cas simplement.

```
fn hsl_to_rgb(hsl: [u8; 3]) ->[u8; 3] {
    match hsl {
        [_, _, 0] => [0, 0, 0],
        [_, _, 255] => [255, 255, 255],
```

```
    ...
}
```

Tranches les modèles sont similaires, mais contrairement aux tableaux, les tranches ont des longueurs variables, de sorte que les modèles de tranche correspondent non seulement sur les valeurs mais aussi sur la longueur.

.. dans un modèle de tranche correspond à n'importe quel nombre d'éléments :

```
fn greet_people(names:&[&str]) {
    match names {
        [] => { println!("Hello, nobody.") },
        [a] => { println!("Hello, {}.", a) },
        [a, b] => { println!("Hello, {} and {}.", a, b) },
        [a, .., b] => { println!("Hello, everyone from {} to {}.", a, b) }
    }
}
```

Modèles de référence

Rouiller les modèles prennent en charge deux fonctionnalités pour travailler avec des références. `ref` les modèles empruntent des parties d'une valeur correspondante. `&` les modèles correspondent aux références. Nous couvrirons `ref` d'abord les modèles.

La correspondance d'une valeur non copiable déplace la valeur. En continuant avec l'exemple de compte, ce code serait invalide :

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // error: borrow of moved value: `acc
    }
}
```

Ici, les champs `account.name` et `account.language` sont déplacés dans des variables locales `name` et `language`. Le reste `account` est abandonné. C'est pourquoi nous ne pouvons pas lui emprunter une référence par la suite.

Si `name` et `language` étaient tous deux des valeurs copiables, Rust copierait les champs au lieu de les déplacer, et ce code irait bien. Mais supposons qu'il s'agisse de l'`String` art. Que pouvons-nous faire?

Nous avons besoin d'une sorte de modèle qui *emprunte* les valeurs correspondantes au lieu de les déplacer. Le `ref` mot-clé fait exactement cela :

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // ok
    }
}
```

Désormais, les variables locales `name` et `language` sont des références aux champs correspondants dans `account`. Puisque `account` n'est qu'emprunté, pas consommé, vous pouvez continuer à appeler des méthodes dessus.

Vous pouvez utiliser `ref mut` pour emprunter `mut` des références :

```
match line_result {
    Err(ref err) => log_error(err), // `err` is &Error (shared ref)
    Ok(ref mut line) => {
        trim_comments(line); // modify the String in place
        handle(line);
    }
}
```

Le modèle `Ok(ref mut line)` correspond à tout résultat de réussite et emprunte une `mut` référence à la valeur de réussite stockée à l'intérieur.

Le type opposé de modèle de référence est le `&` modèle. Un modèle commençant par `&` correspond à une référence :

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

Dans cet exemple, supposons `sphere.center()` qu'il renvoie une référence à un champ privé de `sphere`, un modèle courant dans Rust. La valeur renournée est l'adresse d'un `Point3d`. Si le centre est à l'origine, alors `sphere.center()` renvoie `&Point3d { x: 0.0, y: 0.0, z: 0.0 }`.

La correspondance de motifs se déroule comme illustré à la [Figure 10-7](#).

```
value: &Point3d { x: 0.0, y: 0.0, z: 0.0}
```



```
pattern: &Point3d { x, y, z }
```

Illustration 10-7. Correspondance de modèle avec des références

C'est un peu délicat car Rust suit un pointeur ici, une action que nous associons généralement à l'`*` opérateur, pas l'`&` opérateur. La chose à retenir est que les motifs et les expressions sont des opposés naturels. L'expression `(x, y)` transforme deux valeurs en un nouveau tuple, mais le modèle `(x, y)` fait le contraire : il correspond à un tuple et décompose les deux valeurs. C'est pareil avec `&`. Dans une expression, `&` crée une référence. Dans un motif, `&` correspond à une référence.

Faire correspondre une référence suit toutes les règles auxquelles nous nous attendons. Les durées de vie sont imposées. Vous ne pouvez pas y mut accéder via une référence partagée. Et vous ne pouvez pas déplacer une valeur hors d'une référence, même une `mut` référence. Lorsque nous faisons correspondre `&Point3d { x, y, z }`, les variables `x`, `y`, et `z` reçoivent des copies des coordonnées, laissant la `Point3d` valeur d'origine intacte. Cela fonctionne parce que ces champs sont copiables. Si nous essayons la même chose sur une structure avec des champs non copiables, nous aurons une erreur :

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) => // error: can't move out of borrow
    ...
    None => {}
}
```

Mettre au rebut une voiture empruntée pour des pièces n'est pas agréable, et Rust ne le supportera pas. Vous pouvez utiliser un `ref` motif pour emprunter une référence à une pièce. Vous ne le possédez tout simplement pas :

```
Some(&Car { ref engine, .. }) => // ok, engine is a reference
```

Regardons un autre exemple de `&` modèle. Supposons que nous ayons un itérateur `chars` sur les caractères d'une chaîne, et qu'il ait une méthode `chars.peek()` qui renvoie un `Option<&char>` : une référence au caractère suivant, le cas échéant. (Les itérateurs `Peekable` renvoient en fait un `Option<&ItemType>`, comme nous le verrons au [chapitre 15](#).)

Un programme peut utiliser un `&` motif pour obtenir le caractère pointé:

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars"),
}
```

Gardes de match

quelquefoisun bras de match a des conditions supplémentaires qui doivent être remplies avant de pouvoir être considéré comme un match. Supposons que nous implémentions un jeu de société avec des espaces hexagonaux et que le joueur clique simplement pour déplacer une pièce. Pour confirmer que le clic était valide, nous pourrions essayer quelque chose comme ceci :

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) => // try to match if user clicked the current_hex
                                // (it doesn't work: see explanation below)
            Err("You are already there! You must click somewhere else."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

Cela échoue car les identifiants dans les modèles introduisent de *nouvelles* variables. Le modèle `Some(current_hex)` ici crée une nouvelle variable locale `current_hex`, occultant l' argument `current_hex`. Rust émet plusieurs avertissements à propos de ce code, en particulier le dernier bras du `match` est inaccessible. Une façon de résoudre ce problème consiste simplement à utiliser une `if` expression dans le bras de correspondance :

```
match point_to_hex(click) {
    None => Err("That's not a game space."),
    Some(hex) => {
        if hex == current_hex {
            Err("You are already there! You must click somewhere else")
        } else {
            Ok(hex)
        }
    }
}
```

Mais Rust fournit également des *match guards*, des conditions supplémentaires qui doivent être vraies pour qu'un bras de match s'applique, écrit comme `if CONDITION`, entre le motif et le `=>` jeton du bras :

```
match point_to_hex(click) {
    None => Err("That's not a game space."),
    Some(hex) if hex == current_hex =>
        Err("You are already there! You must click somewhere else"),
    Some(hex) => Ok(hex)
}
```

Si le modèle correspond, mais que la condition est fausse, la correspondance se poursuit avec le bras suivant.

Faire correspondre plusieurs possibilités

Un motif du formulaire correspond si l'un ou l'autre des sous-modèles correspond : `pat1 | pat2`

```
let at_end = match chars.peek() {
    Some(&'\r' | &'\n') | None => true,
    _ => false,
};
```

Dans une expression, `|` est l'opérateur OU au niveau du bit, mais ici il fonctionne plus comme le `|` symbole dans une expression régulière.
`at_end` est défini sur `true` si `chars.peek()` est `None`, ou a `Some` contenant un retour chariot ou un saut de ligne.

Utilisez `..=` pour faire correspondre toute une plage de valeurs. Les modèles de plage incluent les valeurs de début et de fin, donc `'0' ..= '9'` correspondent à tous les chiffres ASCII :

```
match next_char {
    '0' ..= '9' => self.read_number(),
    'a' ..= 'z' | 'A' ..= 'Z' => self.read_word(),
    ' ' | '\t' | '\n' => self.skip_whitespace(),
    _ => self.handle_punctuation(),
}
```

Rust autorise également des modèles de plage comme `x..`, qui correspondent à n'importe quelle valeur `x` jusqu'à la valeur maximale du type. Cependant, les autres variétés de plages exclusives de fin, comme

0..100 ou ..100 , et les plages illimitées comme .. ne sont pas encore autorisées dans les modèles.

Reliure avec @ Patterns

Enfin, les matchs `x @ pattern` exactement comme le `given pattern`, mais en cas de succès, au lieu de créer des variables pour des parties de la valeur correspondante, il crée une seule variable `x` et y déplace ou copie la valeur entière. Par exemple, disons que vous avez ce code :

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) => {
        optimized_paint(&Shape::Rect(top_left, bottom_right))
    }
    other_shape => {
        paint_outline(other_shape.get_outline())
    }
}
```

Notez que le premier cas déballe une `Shape::Rect` valeur, uniquement pour reconstruire une valeur identique `Shape::Rect` sur la ligne suivante. Cela peut être réécrit pour utiliser un `@` modèle :

```
rect @ Shape::Rect(..) => {
    optimized_paint(&rect)
}
```

`@` les motifs sont également utiles avec les plages :

```
match chars.next() {
    Some(digit @ '0'...'9') => read_number(digit, chars),
    ...
},
```

Où les modèles sont autorisés

Bien que les modèles soient les plus importants dans les `match` expressions, ils sont également autorisés à plusieurs autres endroits, généralement à la place d'un identifiant. La signification est toujours la même : au lieu de simplement stocker une valeur dans une seule variable, Rust utilise la correspondance de modèle pour séparer la valeur.

Cela signifie que les modèles peuvent être utilisés pour...

```

// ...unpack a struct into three new local variables
let Track { album, track_number, title, .. } = song;

// ...unpack a function argument that's a tuple
fn distance_to((x, y): (f64, f64)) ->f64 { ... }

// ...iterate over keys and values of a HashMap
for (id, document) in &cache_map {
    println!("Document #{:}: {}", id, document.title);
}

// ...automatically dereference an argument to a closure
// (handy because sometimes other code passes you a reference
// when you'd rather have a copy)
let sum = numbers.fold(0, |a, &num| a + num);

```

Chacun d'entre eux enregistre deux ou trois lignes de code passe-partout. Le même concept existe dans d'autres langages : en JavaScript, cela s'appelle *déstructuration*, tandis qu'en Python, c'est *unpacking*.

Notez que dans les quatre exemples, nous utilisons des modèles dont la correspondance est garantie. Le modèle `Point3d { x, y, z }` correspond à toutes les valeurs possibles du `Point3d` type de structure, `(x, y)` correspond à n'importe quelle `(f64, f64)` paire, etc. Les motifs qui correspondent toujours sont spéciaux dans Rust. Ils sont appelés *modèles irréfutables*, et ce sont les seuls modèles autorisés dans les quatre endroits indiqués ici (après `let`, dans les arguments de fonction, après `for` et dans les arguments de fermeture).

Un *modèle réfutable* est un qui ne correspond pas, comme `ok(x)`, qui ne correspond pas à un résultat d'erreur, ou `'0' ..= '9'`, qui ne correspond pas au caractère `'Q'`. Des modèles réfutables peuvent être utilisés dans `match` les armes, car `match` ils sont conçus pour eux : si un modèle ne correspond pas, ce qui se passe ensuite est clair. Les quatre exemples précédents sont des endroits dans les programmes Rust où un modèle peut être pratique, mais le langage ne permet pas l'échec de la correspondance.

Les modèles réfutables sont également autorisés dans les expressions `if let` et, qui peuvent être utilisées pour... `while let`

```

// ...handle just one enum variant specially
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

```

```

// ...run some code only if a table lookup succeeds
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ...repeatedly try something until it succeeds
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // let the user try again (it might still be a human)
}

// ...manually loop over an iterator
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}

```

Pour plus de détails sur ces expressions, voir [« if let »](#) et [« Loops »](#).

Remplir un arbre binaire

Plus tôt nous avons promis de montrer comment mettre en œuvre une méthode, `BinaryTree::add()`, qui ajoute un nœud à un `BinaryTree` de ce type :

```

// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right:BinaryTree<T>,
}

```

Vous en savez maintenant assez sur les modèles pour écrire cette méthode. Une explication des arbres de recherche binaires dépasse le cadre de ce livre, mais pour les lecteurs déjà familiarisés avec le sujet, cela vaut la peine de voir comment cela se passe dans Rust.

```

1  impl<T: Ord> BinaryTree<T> {
2      fn add(&mut self, value: T) {
3          match *self {
4              BinaryTree:: Empty => {
5                  *self = BinaryTree:: NonEmpty(Box:: new(TreeNode {

```

```

6             element: value,
7             left: BinaryTree:: Empty,
8             right: BinaryTree:: Empty,
9         })
10    }
11    BinaryTree::NonEmpty(ref mut node) => {
12        if value <= node.element {
13            node.left.add(value);
14        } else {
15            node.right.add(value);
16        }
17    }
18}
19}
20}

```

La ligne 1 indique à Rust que nous définissons une méthode sur `BinaryTree`s de types ordonnés. C'est exactement la même syntaxe que nous utilisons pour définir des méthodes sur des structures génériques, expliquée dans ["Définir des méthodes avec impl"](#).

Si l'arborescence existante `*self` est vide, c'est le cas le plus simple. Les lignes 5 à 9 s'exécutent, changeant l'`Empty` arbre en `NonEmpty` un. L'appel à `Box::new()` here alloue un nouveau `TreeNode` dans le tas. Lorsque nous avons terminé, l'arbre contient un élément. Ses sous-arborescences gauche et droite sont toutes les deux `Empty`.

Si `*self` n'est pas vide, nous faisons correspondre le motif de la ligne 11 :

```
BinaryTree::NonEmpty(ref mut node) => {
```

Ce modèle emprunte une référence mutable au `Box<TreeNode<T>>`, afin que nous puissions accéder et modifier les données dans ce nœud d'arbre. Cette référence est nommée `node`, et elle est dans la portée de la ligne 12 à la ligne 16. Puisqu'il y a déjà un élément dans ce nœud, le code doit appeler de manière récursive pour ajouter le nouvel élément au sous-arbre `.add()` gauche ou droit.

La nouvelle méthode peut être utilisée comme ceci :

```

let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...

```

La grande image

Les énumérations de Rust sont peut-être nouvelles pour la programmation système, mais elles ne sont pas une idée nouvelle. Voyageant sous divers noms à consonance académique, comme *les types de données algébriques*, ils sont utilisés dans les langages de programmation fonctionnels depuis plus de quarante ans. On ne sait pas pourquoi si peu d'autres langages de la tradition C en ont jamais eu. C'est peut-être simplement que pour un concepteur de langage de programmation, combiner les variantes, les références, la mutabilité et la sécurité de la mémoire est extrêmement difficile. Les langages de programmation fonctionnels se passent de mutabilité. Les `C union`, en revanche, ont des variantes, des pointeurs et une mutabilité, mais sont si spectaculairement dangereux que même en C, ils sont un dernier recours. Le vérificateur d'emprunt de Rust est la magie qui permet de combiner les quatre sans compromis.

La programmation est un traitement de données. Mettre les données dans la bonne forme peut faire la différence entre un petit programme rapide et élégant et un enchevêtement lent et gigantesque de ruban adhésif et d'appels de méthodes virtuelles.

Il s'agit de l'adresse des énumérations d'espace problématique. Ils sont un outil de conception pour mettre les données dans la bonne forme. Pour les cas où une valeur peut être une chose, ou une autre chose, ou peut-être rien du tout, les énumérations sont meilleures que les hiérarchies de classes sur chaque axe : plus rapides, plus sûres, moins de code, plus faciles à documenter.

Le facteur limitant est la flexibilité. Les utilisateurs finaux d'une énumération ne peuvent pas l'étendre pour ajouter de nouvelles variantes. Des variantes peuvent être ajoutées uniquement en modifiant la déclaration `enum`. Et lorsque cela se produit, le code existant se brise. Chaque `match` expression qui correspond individuellement à chaque variante de l'énumération doit être revisitée - elle a besoin d'un nouveau bras pour gérer la nouvelle variante. Dans certains cas, troquer la flexibilité contre la simplicité relève du bon sens. Après tout, la structure de JSON ne devrait pas changer. Et dans certains cas, revoir toutes les utilisations d'une énumération lorsqu'elle change est exactement ce que nous voulons. Par exemple, lorsqu'un `enum` est utilisé dans un compilateur pour représenter les différents opérateurs d'un langage de programmation, l'ajout d'un nouvel opérateur *doit* impliquer de toucher tout le code qui gère les opérateurs.

Mais parfois, plus de flexibilité est nécessaire. Pour ces situations, Rust a des traits, le sujet de notre prochain chapitre.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 11. Traits et génériques

[Un] informaticien a tendance à être capable de traiter des structures non uniformes - cas 1, cas 2, cas 3 - tandis qu'un mathématicien aura tendance à vouloir un axiome unificateur qui régit tout un système.

—Donald Knuth

Une des grandes découvertes en programmation, c'est qu'il est possible d'écrire du code qui opère sur des valeurs de nombreux types différents, *même des types qui n'ont pas encore été inventés*. Voici deux exemples :

- `Vec<T>` est générique : vous pouvez créer un vecteur de n'importe quel type de valeur, y compris des types définis dans votre programme que les auteurs `Vec` n'avaient jamais anticipés.
- Beaucoup de choses ont des `.write()` méthodes, y compris `File`s et `TcpStream`s. Votre code peut prendre un écrivain par référence, n'importe quel écrivain, et lui envoyer des données. Votre code n'a pas à se soucier de quel type d'écrivain il s'agit. Plus tard, si quelqu'un ajoute un nouveau type de rédacteur, votre code le supportera déjà.

Bien sûr, cette capacité n'est pas nouvelle avec Rust. C'est ce qu'on appelle le *polymorphisme*, et c'était la nouvelle technologie de langage de programmation en vogue des années 1970. À présent, il est effectivement universel. Rust prend en charge le polymorphisme avec deux fonctionnalités connexes : les traits et les génériques. Ces concepts seront familiers à de nombreux programmeurs, mais Rust adopte une nouvelle approche inspirée des classes de types de Haskell.

Les traits sont le point de vue de Rust sur les interfaces ou les classes de base abstraites. Au début, elles ressemblent à des interfaces en Java ou en C#. Le trait pour écrire des octets s'appelle `std::io::Write`, et sa définition dans la bibliothèque standard commence comme ceci :

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

Ce trait propose plusieurs méthodes; nous n'avons montré que les trois premiers.

Les types standard `File` et `TcpStream` les deux implémentent `std::io::Write`. Tout comme `Vec<u8>`. Les trois types fournissent des méthodes nommées `.write()`, `.flush()`, etc. Le code qui utilise un écrivain sans se soucier de son type ressemble à ceci :

```
use std:: io::Write;

fn say_hello(out: &mut dyn Write) -> std:: io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Le type de `out` est `&mut dyn Write`, ce qui signifie "une référence mutable à toute valeur qui implémente le `Write` trait". Nous pouvons passer `say_hello` une référence mutable à une telle valeur :

```
use std:: fs:: File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // works

let mut bytes = vec![];
say_hello(&mut bytes)?; // also works
assert_eq!(bytes, b"hello world\n");
```

Ce chapitre commence par montrer comment les traits sont utilisés, comment ils fonctionnent et comment définir les vôtres. Mais il y a plus dans les traits que ce que nous avons laissé entendre jusqu'à présent. Nous les utiliserons pour ajouter des méthodes d'extension aux types existants, même aux types intégrés comme `str` et `bool`. Nous expliquerons pourquoi l'ajout d'un trait à un type ne coûte pas de mémoire supplémentaire et comment utiliser des traits sans surcoût d'appel de méthode virtuelle. Nous verrons que les traits intégrés sont le crochet dans le langage fourni par Rust pour la surcharge des opérateurs et d'autres fonctionnalités. Et nous couvrirons le `Self` type, les fonctions associées et les types associés, trois fonctionnalités que Rust a extraites de Haskell qui résolvent élégamment les problèmes que d'autres langages traitent avec des solutions de contournement et des hacks.

Génériques sont l'autre saveur du polymorphisme dans Rust. Comme un modèle C++, une fonction ou un type générique peut être utilisé avec des valeurs de nombreux types différents :

```

/// Given two values, pick whichever one is less.
fn min<T: Ord>(value1: T, value2: T) ->T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}

```

Le `<T: Ord>` dans cette fonction signifie que `min` peut être utilisé avec des arguments de n'importe quel type `T` qui implémente le `Ord` trait, c'est-à-dire n'importe quel type ordonné. Une exigence comme celle-ci est appelée une *limite*, car elle définit des limites sur les types qui `T` pourraient éventuellement l'être. Le compilateur génère un code machine personnalisé pour chaque type `T` que vous utilisez réellement.

Les génériques et les traits sont étroitement liés : les fonctions génériques utilisent des traits dans des limites pour préciser à quels types d'arguments ils peuvent être appliqués. Nous parlerons donc également de la façon dont `&mut dyn Write` et `<T: Write>` sont similaires, de la façon dont ils sont différents et de la manière de choisir entre ces deux façons d'utiliser les traits.

Utilisation des caractéristiques

Un trait est une fonctionnalité qu'un type donné peut ou non prendre en charge. Le plus souvent, un trait représente une capacité : quelque chose qu'un type peut faire.

- Une valeur qui implémente `std::io::Write` peut écrire des octets.
- Une valeur qui implémente `std::iter::Iterator` peut produire une séquence de valeurs.
- Une valeur qui implémente `std::clone::Clone` peut créer des clones d'elle-même en mémoire.
- Une valeur qui implémente `std::fmt::Debug` peut être imprimée à l'aide `println!()` du `{:?}` spécificateur de format.

Ces quatre traits font tous partie de la bibliothèque standard de Rust, et de nombreux types standard les implémentent. Par exemple:

- `std::fs::File` implémente le `Write` trait ; il écrit des octets dans un fichier local. `std::net::TcpStream` écrit sur une connexion réseau. `Vec<u8>` implémente également `Write`. Chaque `.write()` appel sur un vecteur d'octets ajoute des données à la fin.

- Range<i32> (le type de 0 .. 10) implémente le `Iterator` trait, tout comme certains types d'itérateurs associés aux tranches, aux tables de hachage, etc.
- La plupart des types de bibliothèques standard implémentent `Clone`. Les exceptions sont principalement des types comme `TcpStream` celui qui représentent plus que de simples données en mémoire.
- De même, la plupart des types de bibliothèques standard prennent en charge `Debug`.

Il existe une règle inhabituelle concernant les méthodes de trait : le trait lui-même doit être dans la portée. Sinon, toutes ses méthodes sont masquées :

```
let mut buf:Vec<u8> = vec![];
buf.write_all(b"hello")?; // error: no method named `write_all`
```

Dans ce cas, le compilateur affiche un message d'erreur convivial qui suggère d'ajouter `use std::io::Write;` et qui résout effectivement le problème :

```
use std:: io::Write;

let mut buf:Vec<u8> = vec![];
buf.write_all(b"hello")?; // ok
```

Rust a cette règle car, comme nous le verrons plus loin dans ce chapitre, vous pouvez utiliser des traits pour ajouter de nouvelles méthodes à n'importe quel type, même les types de bibliothèque standard comme `u32` et `str`. Les caisses tierces peuvent faire la même chose. Évidemment, cela pourrait entraîner des conflits de nommage ! Mais puisque Rust vous oblige à importer les traits que vous prévoyez d'utiliser, les caisses sont libres de profiter de cette superpuissance. Pour obtenir un conflit, vous devez importer deux traits qui ajoutent une méthode portant le même nom au même type. C'est rare en pratique. (Si vous rencontrez un conflit, vous pouvez préciser ce que vous voulez en utilisant la [syntaxe de méthode entièrement qualifiée](#), abordée plus loin dans le chapitre.)

La raison `Clone` et `Iterator` les méthodes fonctionnent sans aucune importation spéciale, c'est qu'ils sont toujours dans la portée par défaut : ils font partie du prélude standard, des noms que Rust importe automatiquement dans chaque module. En fait, le prélude est surtout une sélection soigneusement choisie de traits. Nous en couvrirons beaucoup au [chapitre 13](#).

Les programmeurs C++ et C# auront déjà remarqué que les méthodes de trait sont comme des méthodes virtuelles. Pourtant, les appels comme celui montré ci-dessus sont rapides, aussi rapides que n'importe quel autre appel de méthode. Autrement dit, il n'y a pas de polymorphisme ici. Il est évident qu'il buf s'agit d'un vecteur, pas d'un fichier ou d'une connexion réseau. Le compilateur peut émettre un simple appel à

`Vec<u8>::write()`. Il peut même intégrer la méthode. (C++ et C# feront souvent la même chose, bien que la possibilité de sous-classement l'empêche parfois.) Seuls les appels traversants `&mut dyn Write` entraînent la surcharge d'une répartition dynamique, également connue sous le nom d'appel de méthode virtuelle, qui est indiquée par le mot-`dyn` clé dans le type. `dyn Write` est connu comme un *objet trait*; nous examinerons les détails techniques des objets trait, et comment ils se comparent aux fonctions génériques, dans les sections suivantes.

Objets de trait

Il y a deux façons d'utiliser les traitsécrire du code polymorphe en Rust : objets traits et génériques. Nous présenterons d'abord les objets de trait et passerons aux génériques dans la section suivante.

Rust n'autorise pas les variables de type `dyn Write`:

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf; // error: `Write` does not have a constant size
```

La taille d'une variable doit être connue au moment de la compilation, et les types qui l'implémentent `write` peuvent être de n'importe quelle taille.

Cela peut être surprenant si vous venez de C# ou Java, mais la raison est simple. En Java, une variable de type `OutputStream` (l'interface standard Java analogue à `std::io::Write`) est une référence à tout objet qui implémente `OutputStream`. Le fait qu'il s'agisse d'une référence va sans dire. C'est la même chose avec les interfaces en C# et la plupart des autres langages.

Ce que nous voulons dans Rust, c'est la même chose, mais dans Rust, les références sont explicites :

```
let mut buf: Vec<u8> = vec![];
let writer:&mut dyn Write = &mut buf; // ok
```

Une référence à un type de trait, comme `writer`, est appelée un *objet de trait*. Comme toute autre référence, un objet de trait pointe vers une valeur, il a une durée de vie et il peut être `mut` partagé ou partagé.

Ce qui rend un objet trait différent, c'est que Rust ne connaît généralement pas le type du référent au moment de la compilation. Ainsi, un objet de trait inclut un peu d'informations supplémentaires sur le type du référent. C'est strictement pour l'usage propre de Rust dans les coulisses : lorsque vous appelez `writer.write(data)`, Rust a besoin des informations de type pour appeler dynamiquement la bonne `write` méthode en fonction du type de `*writer`. Vous ne pouvez pas interroger directement les informations de type et Rust ne prend pas en charge la conversion descendante de l'objet de trait `&mut dyn Write` en un type concret tel que `Vec<u8>`.

Disposition des objets de trait

En mémoire, un objet traitest un pointeur gras composé d'un pointeur vers la valeur, plus un pointeur vers une table représentant le type de cette valeur. Chaque objet trait occupe donc deux mots machine, comme le montre la [figure 11-1](#).

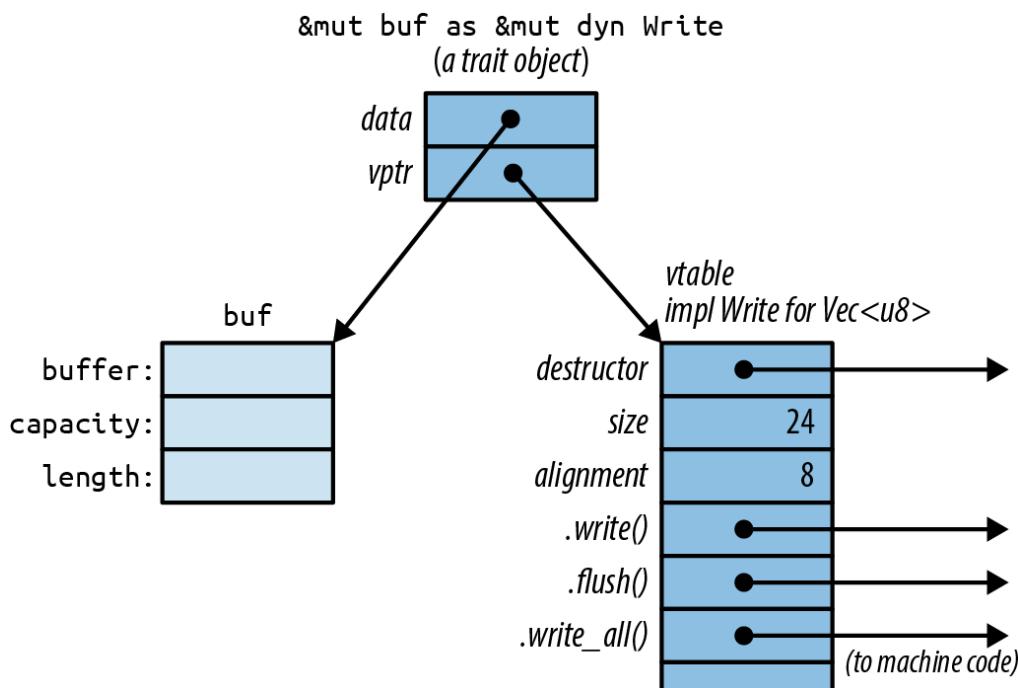


Illustration 11-1. Objets de trait en mémoire

C++ possède également ce type d'informations de type à l'exécution. C'est ce qu'on appelle une *table virtuelle*, ou *vtable*. En Rust, comme en C++, la vtable est générée une seule fois, au moment de la compilation, et partagée par tous les objets du même type. Tout ce qui apparaît dans la teinte plus foncée de la [figure 11-1](#), y compris la vtable, est un détail d'implémentation privé de Rust. Encore une fois, ce ne sont pas des champs et

des structures de données auxquels vous pouvez accéder directement. Au lieu de cela, le langage utilise automatiquement la vtable lorsque vous appelez une méthode d'un objet trait, pour déterminer quelle implémentation appeler.

Les programmeurs C++ chevronnés remarqueront que Rust et C++ utilisent la mémoire un peu différemment. En C++, le pointeur vtable, ou *vptr*, est stocké dans le cadre de la structure. Rust utilise des pointeurs gras à la place. La structure elle-même ne contient que ses champs. De cette façon, une structure peut implémenter des dizaines de traits sans contenir des dizaines de vptrs. Même des types comme `i32`, qui ne sont pas assez grands pour accueillir un vptr, peuvent implémenter des traits.

Rust convertit automatiquement les références ordinaires en objets trait si nécessaire. C'est pourquoi nous pouvons passer `&mut local_file` à `say_hello` dans cet exemple :

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file);
```

Le type de `&mut local_file` est `&mut File` et le type de l'argument de `say_hello` est `&mut dyn Write`. Puisque `a File` est une sorte d'écrivain, Rust le permet, convertissant automatiquement la référence simple en un objet trait.

De même, Rust se fera un plaisir de convertir `a Box<File>` en `a Box<dyn Write>`, une valeur qui possède un écrivain dans le tas :

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>`, comme `&mut dyn Write`, est un gros pointeur : il contient l'adresse du rédacteur lui-même et l'adresse de la vtable. Il en va de même pour les autres types de pointeurs, comme `Rc<dyn Write>`.

Ce type de conversion est le seul moyen de créer un objet trait. Ce que le compilateur fait réellement ici est très simple. Au moment où la conversion se produit, Rust connaît le vrai type du référent (dans ce cas, `File`), il ajoute donc simplement l'adresse de la vtable appropriée, transformant le pointeur normal en un pointeur gras.

Fonctions génériques et paramètres de type

Au début de ce chapitre, nous avons montré une `say_hello()` fonction qui a pris un objet trait comme argument. Réécrivons cette fonction comme une fonction générique :

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Seule la signature de type a changé :

```
fn say_hello(out:&mut dyn Write)      // plain function

fn say_hello<W: Write>(out:&mut W)    // generic function
```

La phrase `<W: Write>` est ce qui rend la fonction générique. Ceci est un *paramètre de type*. Cela signifie que dans tout le corps de cette fonction, `w` représente un type qui implémente le `Write` trait. Les paramètres de type sont généralement des lettres majuscules simples, par convention.

Le type `w` correspond à dépend de la façon dont la fonction générique est utilisée :

```
say_hello(&mut local_file)?; // calls say_hello::<File>
say_hello(&mut bytes)?;     // calls say_hello::<Vec<u8>>
```

Lorsque vous passez `&mut local_file` à la fonction générique `say_hello()`, vous appelez `say_hello::<File>()`. Rust génère du code machine pour cette fonction qui appelle `File::write_all()` et `File::flush()`. Quand vous passez `&mut bytes`, vousappelez `say_hello::<Vec<u8>>()`. Rust génère un code machine séparé pour cette version de la fonction, appelant les `Vec<u8>` méthodes correspondantes. Dans les deux cas, Rust déduit le type `w` du type de l'argument. Ce processus est connu sous le nom de *monomorphisation*, et le compilateur gère tout cela automatiquement.

Vous pouvez toujours épeler les paramètres de type :

```
say_hello::<File>(&mut local_file);
```

Ceci est rarement nécessaire, car Rust peut généralement déduire les paramètres de type en examinant les arguments. Ici, la `say_hello` fonction

générique attend un `&mut` w argument, et nous lui transmettons un `&mut File`, donc Rust en déduit que `w = File`.

Si la fonction générique que vous appelez n'a pas d'arguments qui fournissent des indices utiles, vous devrez peut-être l'épeler :

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect(); // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

Parfois, nous avons besoin de plusieurs capacités à partir d'un paramètre de type. Par exemple, si nous voulons imprimer les dix valeurs les plus courantes dans un vecteur, nous aurons besoin que ces valeurs soient imprimables :

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values:&Vec<T>) { ... }
```

Mais ce n'est pas assez bon. Comment envisageons-nous de déterminer quelles valeurs sont les plus courantes ? La méthode habituelle consiste à utiliser les valeurs comme clés dans une table de hachage. Cela signifie que les valeurs doivent prendre en charge les opérations `Hash` et `Eq`. Les bornes sur `T` doivent inclure celles-ci ainsi que `Debug`. La syntaxe pour cela utilise le `+` signe :

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values:&Vec<T>) { ... }
```

Certains types implement `Debug`, d'autres implement `Hash`, d'autres support `Eq`, et quelques-uns, comme `u32` et `String`, implémentent les trois, comme le montre la [figure 11-2](#).

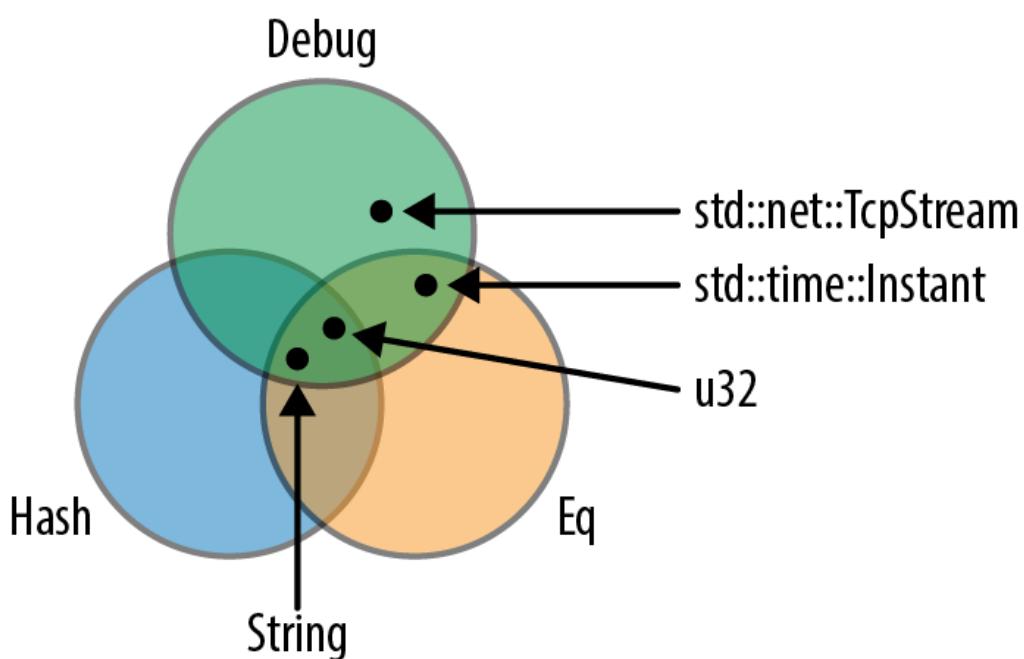


Illustration 11-2. Les traits comme ensembles de types

Il est également possible qu'un paramètre de type n'ait aucune limite, mais vous ne pouvez pas faire grand-chose avec une valeur si vous n'avez spécifié aucune limite pour celle-ci. Vous pouvez le déplacer. Vous pouvez le mettre dans une boîte ou un vecteur. C'est à peu près ça.

Les fonctions génériques peuvent avoir plusieurs paramètres de type :

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) ->Results
{ ... }
```

Comme le montre cet exemple, les limites peuvent devenir si longues qu'elles sont pénibles pour les yeux. Rust fournit une syntaxe alternative en utilisant le mot-clé `where` :

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

Les paramètres de type `M` et `R` sont toujours déclarés au début, mais les limites sont déplacées sur des lignes séparées. Ce type de `where` clause est également autorisé sur les structures génériques, les énumérations, les alias de type et les méthodes, partout où les limites sont autorisées.

Bien sûr, une alternative aux `where` clauses est de rester simple : trouver un moyen d'écrire le programme sans utiliser les génériques de manière

aussi intensive.

« [Recevoir des références en tant qu'arguments de fonction](#) » a introduit la syntaxe pour la durée de vie des paramètres. Une fonction générique peut avoir à la fois des paramètres de durée de vie et des paramètres de type. Les paramètres de durée de vie viennent en premier :

```
// Return a reference to the point in `candidates` that's
// closest to the `target` point.
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P:MeasureDistance
{
    ...
}
```

Cette fonction prend deux arguments, `target` et `candidates`. Les deux sont des références, et nous leur donnons des durées de vie distinctes '`t`' et '`c`' (comme indiqué dans "[Paramètres de durée de vie distincts](#)"). De plus, la fonction fonctionne avec n'importe quel type `P` qui implémente le `MeasureDistance` trait, nous pouvons donc l'utiliser sur des `Point2d` valeurs dans un programme et `Point3d` des valeurs dans un autre.

Les durées de vie n'ont jamais d'impact sur le code machine. Deux appels `nearest()` utilisant le même type `P`, mais des durées de vie différentes, appellent la même fonction compilée. Seuls des types différents amènent Rust à compiler plusieurs copies d'une fonction générique.

En plus des types et des durées de vie, les fonctions génériques peuvent également prendre des paramètres constants, comme la `Polynomial` structure que nous avons présentée dans "[Structures génériques à paramètres constants](#)" :

```
fn dot_product<const N: usize>(a: [f64; N], b: [f64; N]) -> f64 {
    let mut sum = 0.;
    for i in 0..N {
        sum += a[i] * b[i];
    }
    sum
}
```

Ici, la phrase `<const N: usize>` indique que la fonction `dot_product` attend un paramètre générique `N`, qui doit être un `usize`. Étant donné `N`, la fonction prend deux arguments de type `[f64; N]`, et additionne les produits de leurs éléments correspondants. Ce qui

se distingue d'un argument `N` ordinaire, c'est que vous pouvez l'utiliser dans les types dans la signature ou le corps de `.usize dot_product`

Comme pour les paramètres de type, vous pouvez soit fournir explicitement des paramètres constants, soit laisser Rust les déduire :

```
// Explicitly provide `3` as the value for `N`.
dot_product::<3>([0.2, 0.4, 0.6], [0., 0., 1.])

// Let Rust infer that `N` must be `2`.
dot_product([3., 4.], [-5., 1.])
```

Bien entendu, les fonctions ne sont pas le seul type de code générique dans Rust :

- Nous avons déjà couvert les types génériques dans ["Generic Structs"](#) et ["Generic Enums"](#).
- Une méthode individuelle peut être générique, même si le type sur lequel elle est définie n'est pas générique :

```
impl PancakeStack {
    fn push<T: Topping>(&mut self, goop: T) ->PancakeResult<()> {
        goop.pour(&self);
        self.absorb_topping(goop)
    }
}
```

- Les alias de type peuvent également être génériques :

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- Nous aborderons les traits génériques plus loin dans ce chapitre.

Toutes les fonctionnalités introduites dans cette section (limites, `where` clauses, paramètres de durée de vie, etc.) peuvent être utilisées sur tous les éléments génériques, pas seulement sur les fonctions..

Lequel utiliser

Le choix d'utiliser des objets de trait ou du code générique est subtil. Étant donné que les deux fonctionnalités sont basées sur des traits, elles ont beaucoup en commun.

Les objets de trait sont le bon choix chaque fois que vous avez besoin d'une collection de valeurs de types mixtes, toutes ensemble. Il est techniquement possible de faire de la salade générique :

```
trait Vegetable {  
    ...  
}  
  
struct Salad<V: Vegetable> {  
    veggies:Vec<V>  
}
```

Cependant, il s'agit d'une conception assez sévère. Chacune de ces salades se compose entièrement d'un seul type de légume. Tout le monde n'est pas fait pour ce genre de choses. Un de vos auteurs a déjà payé 14 \$ pour un `Salad<IcebergLettuce>` et ne s'est jamais vraiment remis de l'expérience.

Comment pouvons-nous construire une meilleure salade? Puisque `Vegetable` les valeurs peuvent être toutes de tailles différentes, nous ne pouvons pas demander à Rust un `Vec<dyn Vegetable>`:

```
struct Salad {  
    veggies:Vec<dyn Vegetable> // error: `dyn Vegetable` does  
                                // not have a constant size  
}
```

Les objets trait sont la solution :

```
struct Salad {  
    veggies:Vec<Box<dyn Vegetable>>  
}
```

Chacun `Box<dyn Vegetable>` peut posséder n'importe quel type de légume, mais la boîte elle-même a une taille constante - deux pointeurs - adaptée au stockage dans un vecteur. Mis à part la malheureuse métaphore mixte d'avoir des boîtes dans sa nourriture, c'est précisément ce qu'il faut, et cela fonctionnerait tout aussi bien pour les formes dans une application de dessin, les monstres dans un jeu, les algorithmes de routage enfichables dans un routeur réseau, etc. sur.

Une autre raison possible d'utiliser des objets trait est de réduire la quantité totale de code compilé. Rust peut devoir compiler une fonction générique plusieurs fois, une fois pour chaque type avec lequel elle est utili-

sée. Cela pourrait rendre le binaire volumineux, un phénomène appelé *gonflement du code* dans les cercles C++. De nos jours, la mémoire est abondante et la plupart d'entre nous ont le luxe d'ignorer la taille du code ; mais des environnements contraints existent.

En dehors des situations impliquant des salades ou des environnements à faibles ressources, les génériques ont trois avantages importants par rapport aux objets de trait, avec pour résultat que dans Rust, les génériques sont le choix le plus courant.

Le premier avantage est la rapidité. Notez l'absence du mot-`dyn` clé dans les signatures de fonctions génériques. Étant donné que vous spécifiez les types au moment de la compilation, soit explicitement, soit par inférence de type, le compilateur sait exactement quelle `write` méthode appeler. Le `dyn` mot-clé n'est pas utilisé car il n'y a pas d'objets de trait (et donc pas de répartition dynamique) impliqués.

La fonction générique `min()` présentée dans l'introduction est aussi rapide que si nous avions écrit des fonctions séparées `min_u8`, `min_i64`, `min_string`, etc. Le compilateur peut l'intégrer, comme n'importe quelle autre fonction, donc dans une version de version, un appel à `min::<i32>` n'est probablement que deux ou trois instructions. Un appel avec des arguments constants, comme `min(5, 3)`, sera encore plus rapide : Rust peut l'évaluer au moment de la compilation, de sorte qu'il n'y a aucun coût d'exécution.

Ou considérez cet appel de fonction générique :

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` renvoie un écrivain de type `Sink` qui supprime silencieusement tous les octets qui lui sont écrits.

Lorsque Rust génère du code machine pour cela, il peut émettre du code qui appelle `Sink::write_all`, vérifie les erreurs, puis appelle `Sink::flush`. C'est ce que le corps de la fonction générique dit de faire.

Ou, Rust pourrait regarder ces méthodes et réaliser ce qui suit :

- `Sink::write_all()` ne fait rien.
- `Sink::flush()` ne fait rien.
- Aucune des deux méthodes ne renvoie jamais d'erreur.

En bref, Rust dispose de toutes les informations dont il a besoin pour optimiser entièrement cet appel de fonction.

Comparez cela au comportement avec les objets trait. Rust ne sait jamais sur quel type de valeur un objet de trait pointe jusqu'au moment de l'exécution. Ainsi, même si vous transmettez à `Sink`, la surcharge liée à l'appel de méthodes virtuelles et à la recherche d'erreurs s'applique toujours.

Le deuxième avantage des génériques est que tous les traits ne peuvent pas prendre en charge les objets de trait. Les traits prennent en charge plusieurs fonctionnalités, telles que les fonctions associées, qui ne fonctionnent qu'avec les génériques : ils excluent entièrement les objets de trait. Nous soulignerons ces caractéristiques au fur et à mesure que nous les aborderons.

Le troisième avantage des génériques est qu'il est facile de lier un paramètre de type générique à plusieurs traits à la fois, comme notre `top_ten` fonction l'a fait lorsqu'elle a demandé à son `T` paramètre d'implémenter `Debug + Hash + Eq`. Les objets de trait ne peuvent pas faire cela : les types comme `&mut (dyn Debug + Hash + Eq)` ne sont pas pris en charge dans Rust. (Vous pouvez contourner ce problème avec des sous-[traits](#), définis plus loin dans ce chapitre, mais c'est un peu complexe.)

Définir et mettre en œuvre les traits

Définir un trait est simple. Donnez-lui un nom et répertoriez les signatures de type des méthodes de trait. Si nous écrivons un jeu, nous pourrions avoir un trait comme celui-ci :

```
/// A trait for characters, items, and scenery -
/// anything in the game world that's visible on screen.
trait Visible {
    /// Render this object on the given canvas.
    fn draw(&self, canvas:&mut Canvas);

    /// Return true if clicking at (x, y) should
    /// select this object.
    fn hit_test(&self, x: i32, y: i32) ->bool;
}
```

Pour implémenter un trait, utilisez la syntaxe : `impl TraitName for Type`

```

impl Visible for Broom {
    fn draw(&self, canvas:&mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) ->bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}

```

Notez que cela `impl` contient une implémentation pour chaque méthode du `Visible` trait, et rien d'autre. Tout ce qui est défini dans un trait `impl` doit en fait être une caractéristique du trait ; si nous voulions ajouter une méthode d'assistance à l'appui de `Broom::draw()`, nous devions la définir dans un `impl` bloc séparé :

```

impl Broom {
    /// Helper function used by Broom::draw() below.
    fn broomstick_range(&self) ->Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}

```

Ces fonctions d'assistance peuvent être utilisées dans les `impl` blocs de caractéristiques :

```

impl Visible for Broom {
    fn draw(&self, canvas:&mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}

```

Méthodes par défaut

Le `Sink` type d'écrivain dont nous avons parlé précédemment peut être implémenté en quelques lignes de code. Tout d'abord, nous définissons le type :

```
/// A Writer that ignores whatever data you write to it.  
pub struct Sink;
```

Sink est une structure vide, car nous n'avons pas besoin d'y stocker de données. Ensuite, nous fournissons une implémentation du `Write` trait pour `Sink`:

```
use std::io::{Write, Result};  
  
impl Write for Sink {  
    fn write(&mut self, buf: &[u8]) -> Result<usize> {  
        // Claim to have successfully written the whole buffer.  
        Ok(buf.len())  
    }  
  
    fn flush(&mut self) -> Result<()> {  
        Ok(())  
    }  
}
```

Jusqu'à présent, cela ressemble beaucoup au `Visible` trait. Mais nous avons aussi vu que le `Write` trait a une `write_all` méthode :

```
let mut out = Sink;  
out.write_all(b"hello world\n")?;
```

Pourquoi Rust nous laisse-t-il `impl Write for Sink` sans définir cette méthode ? La réponse est que la définition de la bibliothèque standard du `Write` trait contient une *implémentation par défaut* pour `write_all` :

```
trait Write {  
    fn write(&mut self, buf: &[u8]) -> Result<usize>;  
    fn flush(&mut self) -> Result<()>;  
  
    fn write_all(&mut self, buf: &[u8]) -> Result<()> {  
        let mut bytes_written = 0;  
        while bytes_written < buf.len() {  
            bytes_written += self.write(&buf[bytes_written..])?;  
        }  
        Ok(())  
    }  
    ...  
}
```

Les méthodes `write` et `flush` sont les méthodes de base que chaque écrivain doit mettre en œuvre. Un écrivain peut également implémenter `write_all`, mais si ce n'est pas le cas, l'implémentation par défaut présentée précédemment sera utilisée.

Vos propres traits peuvent inclure des implémentations par défaut utilisant la même syntaxe.

L'utilisation la plus spectaculaire des méthodes par défaut dans la bibliothèque standard est le `Iterator` trait, qui a une méthode requise (`.next()`) et des dizaines de méthodes par défaut. [Le chapitre 15](#) explique pourquoi.

Traits et autres types de personnes

Rouillervous permet d'implémenter n'importe quel trait sur n'importe quel type, tant que le trait ou le type est introduit dans la caisse actuelle.

Cela signifie que chaque fois que vous souhaitez ajouter une méthode à n'importe quel type, vous pouvez utiliser un trait pour le faire :

```
trait IsEmoji {
    fn is_emoji(&self) ->bool;
}

/// Implement IsEmoji for the built-in character type.
impl IsEmoji for char {
    fn is_emoji(&self) ->bool {
        ...
    }

    assert_eq!('$'.is_emoji(), false);
}
```

Comme toute autre méthode de trait, cette nouvelle `is_emoji` méthode n'est visible que lorsqu'elle `IsEmoji` est dans la portée.

Le seul but de ce trait particulier est d'ajouter une méthode à un type existant, `char`. C'est ce qu'on appelle un *trait d'extension*. Bien sûr, vous pouvez également ajouter ce trait aux types en écrivant `impl IsEmoji for str { ... }`, etc.

Vous pouvez même utiliser un bloc générique `impl` pour ajouter un trait d'extension à toute une famille de types à la fois. Ce trait pourrait être implementé sur n'importe quel type :

```

use std::io::{self, Write};

/// Trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}

```

L'implémentation du trait pour tous les écrivains en fait un trait d'extension, ajoutant une méthode à tous les écrivains Rust :

```

/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}

```

La ligne `impl<W: Write> WriteHtml for W` signifie "pour chaque type `W` qui implémente `Write`, voici une implémentation de `WriteHtml` pour `W`."

La `serde` bibliothèque offre un bel exemple de l'utilité d'implémenter des traits définis par l'utilisateur sur des types standard. `serde` est une bibliothèque de sérialisation. Autrement dit, vous pouvez l'utiliser pour écrire des structures de données Rust sur le disque et les recharger ultérieurement. La bibliothèque définit un trait, `Serialize`, qui est implémenté pour chaque type de données pris en charge par la bibliothèque. Ainsi, dans le `serde` code source, il existe un code implémentant `Serialize` pour les types `bool`, `i8`, `i16`, `i32`, `array` et `tuple`, etc., à travers toutes les structures de données standard telles que `Vec` et `HashMap`.

Le résultat de tout cela est qu'il `serde` ajoute une `.serialize()` méthode à tous ces types. Il peut être utilisé comme ceci :

```

use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()
{
    // Create a JSON serializer to write the data to a file.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);

    // The serde `serialize()` method does the rest.
}

```

```

    config.serialize(&mut serializer)?;

    Ok(())
}

```

Nous avons dit précédemment que lorsque vous implémentez un trait, le trait ou le type doit être nouveau dans la caisse actuelle. C'est ce qu'on appelle la *règle des orphelins*. Cela aide Rust à s'assurer que les implementations de traits sont uniques. Votre code ne peut pas `impl Write for u8`, car les deux `Write` et `u8` sont définis dans la bibliothèque standard. Si Rust laissait les caisses faire cela, il pourrait y avoir plusieurs implementations de `Write for u8`, dans différentes caisses, et Rust n'aurait aucun moyen raisonnable de décider quelle implementation utiliser pour un appel de méthode donné..

(C++ a une restriction d'unicité similaire : la règle de définition unique. En mode C++ typique, il n'est pas appliqué par le compilateur, sauf dans les cas les plus simples, et vous obtenez un comportement indéfini si vous le cassez.)

Soi dans les traits

Un trait peut utiliser le mot-clé `Self` comme type. Le `Clone` trait standard, par exemple, ressemble à ceci (légèrement simplifié) :

```

pub trait Clone {
    fn clone(&self) -> Self;
    ...
}

```

Utiliser `Self` comme type de retour ici signifie que le type de `x.clone()` est le même que le type de `x`, quel qu'il soit. Si `x` est un `String`, alors le type de `x.clone()` est `String` —not `dyn Clone` ou tout autre type clonable.

De même, si nous définissons ce trait :

```

pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}

```

avec deux implementations :

```

impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) ->Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) ->Self {
        ...
    }
}

```

puis à l'intérieur du premier `impl`, `Self` est simplement un alias pour `CherryTree`, et dans le second, c'est un alias pour `Mammoth`. Cela signifie que nous pouvons assembler deux cerisiers ou deux mammouths, pas que nous pouvons créer un hybride mammouth-cerisier. Le type de `self` et le type de `other` doivent correspondre.

Un trait qui utilise le `Self` type est incompatible avec les objets trait :

```

// error: the trait `Spliceable` cannot be made into an object
fn splice_anything(left: &dyn Spliceable, right:&dyn Spliceable) {
    let combo = left.splice(right);
    // ...
}

```

La raison en est quelque chose que nous verrons encore et encore à mesure que nous approfondirons les fonctionnalités avancées des traits. Rust rejette ce code car il n'a aucun moyen de vérifier le type de l'appel `left.splice(right)`. L'intérêt des objets trait est que le type n'est connu qu'au moment de l'exécution. Rust n'a aucun moyen de savoir au moment de la compilation si `left` et `right` seront du même type, comme requis.

Les objets de trait sont vraiment destinés aux types de traits les plus simples, ceux qui pourraient être implémentés à l'aide d'interfaces en Java ou de classes de base abstraites en C++. Les fonctionnalités les plus avancées des traits sont utiles, mais elles ne peuvent pas coexister avec les objets de trait car avec les objets de trait, vous perdez les informations de type dont Rust a besoin pour vérifier le type de votre programme.

Maintenant, si nous avions voulu un épissage génétiquement improbable, nous aurions pu concevoir un trait respectueux de l'objet :

```
pub trait MegaSpliceable {
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn MegaSpliceable>;
}
```

Ce trait est compatible avec les objets trait. Il n'y a aucun problème pour vérifier le type des appels à cette `.splice()` méthode car le type de l'argument `other` n'est pas obligé de correspondre au type de `self`, tant que les deux types sont `MegaSpliceable`.

Sous-trait

Nous pouvons déclarer qu'un trait est une extension d'un autre trait :

```
/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

L'expression `trait Creature: Visible` signifie que toutes les créatures sont visibles. Chaque type qui implémente `Creature` doit également implémenter le `Visible` trait :

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

Nous pouvons implémenter les deux traits dans n'importe quel ordre, mais c'est une erreur d'implémenter `Creature` pour un type sans également implémenter `Visible`. Ici, on dit que `Creature` c'est un sous-*trait* de `Visible`, et c'est `Visible` le `Creature` sur-*trait de*.

Les sous-trait ressemblent aux sous-interfaces en Java ou C#, en ce sens que les utilisateurs peuvent supposer que toute valeur qui implémente un sous-trait implémente également son super-trait. Mais dans Rust, un sous-trait n'hérite pas des éléments associés de son super-trait ; chaque trait doit toujours être dans la portée si vous souhaitez appeler ses méthodes.

En fait, les sous-trait de Rust ne sont en réalité qu'un raccourci pour un lien sur `Self`. Une définition de `Creature` like this est exactement équivalente à celle présentée précédemment :

```
trait Creature where Self:Visible {  
    ...  
}
```

Fonctions associées au type

Dans la plupart des langages orientés objet, les interfaces ne peuvent pas inclure de méthodes ou de constructeurs statiques, mais les traits peuvent inclure des fonctions associées au type, l'analogue de Rust aux méthodes statiques :

```
trait StringSet {  
    /// Return a new empty set.  
    fn new() ->Self;  
  
    /// Return a set that contains all the strings in `strings`.  
    fn from_slice(strings: &[&str]) ->Self;  
  
    /// Find out if this set contains a particular `value`.  
    fn contains(&self, string: &str) ->bool;  
  
    /// Add a string to this set.  
    fn add(&mut self, string:&str);  
}
```

Chaque type qui implémente le `StringSet` trait doit implémenter ces quatre fonctions associées. Les deux premiers, `new()` et `from_slice()`, ne prennent pas de `self` argument. Ils servent de constructeurs. Dans du code non générique, ces fonctions peuvent être appelées à l'aide de la `::` syntaxe, comme n'importe quelle autre fonction associée à un type :

```
// Create sets of two hypothetical types that impl StringSet:  
let set1 = SortedStringSet:: new();  
let set2 = HashedStringSet::new();
```

Dans le code générique, c'est la même chose, sauf que le type est souvent une variable de type, comme dans l'appel à `S::new()` montré ici :

```
/// Return the set of words in `document` that aren't in `wordlist`.  
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S) -> S {  
    let mut unknowns = S::new();
```

```

        for word in document {
            if !wordlist.contains(word) {
                unknowns.add(word);
            }
        }
        unknowns
    }
}

```

Comme les interfaces Java et C#, les objets trait ne prennent pas en charge les fonctions associées au type. Si vous souhaitez utiliser des `&dyn StringSet` objets trait, vous devez modifier le trait, en ajoutant le `where Self: Sized` lien à chaque fonction associée qui ne prend pas d'`self` argument par référence :

```

trait StringSet {
    fn new() -> Self
    where Self:Sized;

    fn from_slice(strings: &[&str]) -> Self
    where Self:Sized;

    fn contains(&self, string: &str) ->bool;
    fn add(&mut self, string:&str);
}

```

Cette limite indique à Rust que les objets de trait sont dispensés de prendre en charge cette fonction associée particulière. Avec ces ajouts, `StringSet` les objets trait sont autorisés ; ils ne prennent toujours pas en charge `new` ou `from_slice`, mais vous pouvez les créer et les utiliser pour appeler `.contains()` et `.add()`. La même astuce fonctionne pour toute autre méthode incompatible avec les objets trait. (Nous renoncerons à l'explication technique plutôt fastidieuse de la raison pour laquelle cela fonctionne, mais le `Sized` trait est couvert au [chapitre 13](#).)

Appels de méthode entièrement qualifiés

Tous les chemins pour appeler les méthodes de trait que nous avons vues jusqu'à présent, s'appuyer sur Rust pour remplir certaines pièces manquantes pour vous. Par exemple, supposons que vous écriviez ce qui suit :

```
"hello".to_string()
```

Il est entendu que `to_string` fait référence à la `to_string` méthode du `ToString` trait, dont nous appelons l' `str` implémentation du type. Il y a donc quatre joueurs dans ce jeu : le trait, la méthode de ce trait, la mise en œuvre de cette méthode et la valeur à laquelle cette mise en œuvre est appliquée. C'est bien que nous n'ayons pas à épeler tout cela à chaque fois que nous voulons appeler une méthode. Mais dans certains cas, vous avez besoin d'un moyen de dire exactement ce que vous voulez dire. Les appels de méthode entièrement qualifiés font l'affaire.

Tout d'abord, il est utile de savoir qu'une méthode n'est qu'un type particulier de fonction. Ces deux appels sont équivalents :

```
"hello".to_string()  
str::to_string("hello")
```

La deuxième forme ressemble exactement à un appel de fonction associée. Cela fonctionne même si la `to_string` méthode prend un `self` argument. Passez simplement `self` comme premier argument de la fonction.

Puisqu'il `to_string` s'agit d'une méthode du `Tostring` trait standard, vous pouvez utiliser deux autres formes :

```
ToString::to_string("hello")  
<str as ToString>::to_string("hello")
```

Ces quatre appels de méthode font exactement la même chose. Le plus souvent, vous n'écrivez que `value.method()`. Les autres formes sont des appels de méthode *qualifiés*. Ils spécifient le type ou le trait auquel une méthode est associée. La dernière forme, avec les chevrons, spécifie les deux : un appel de méthode *entièrement qualifié*.

Lorsque vous écrivez `"hello".to_string()`, en utilisant l' `.` opérateur, vous ne dites pas exactement quelle `to_string` méthode vous appelez. Rust a un algorithme de recherche de méthode qui calcule cela, en fonction des types, des coercitions déréférencées, etc. Avec des appels entièrement qualifiés, vous pouvez dire exactement de quelle méthode vous parlez, et cela peut aider dans quelques cas étranges :

- Lorsque deux méthodes portent le même nom. L'exemple classique du hokey est le `Outlaw` avec deux `.draw()` méthodes de deux traits dif-

férents, une pour le dessiner sur l'écran et une pour interagir avec la loi :

```
outlaw.draw(); // error: draw on screen or draw pistol?  
  
Visible:: draw(&outlaw); // ok: draw on screen  
HasPistol::draw(&outlaw); // ok: corral
```

Habituellement, vous feriez mieux de renommer l'une des méthodes, mais parfois vous ne pouvez pas.

- Lorsque le type de l' `self` argument ne peut pas être déduit :

```
let zero = 0; // type unspecified; could be `i8`, `u8`, ...  
  
zero.abs(); // error: can't call method `abs`  
// on ambiguous numeric type  
  
i64::abs(zero); // ok
```

- Lors de l'utilisation de la fonction elle-même comme valeur de fonction :

```
let words: Vec<String> =  
    line.split_whitespace() // iterator produces &str values  
    .map(ToString::to_string) // ok  
    .collect();
```

- Lors de l'appel de méthodes de trait dans des macros. Nous vous expliquerons au [chapitre 21](#).

La syntaxe entièrement qualifiée fonctionne également pour les fonctions associées. Dans la section précédente, nous avons écrit `S::new()` pour créer un nouvel ensemble dans une fonction générique. On aurait aussi pu écrire `StringSet::new()` ou `<S as StringSet>::new()`.

Caractéristiques qui définissent les relations entre les types

Jusqu'à présent, chaque trait nous avons examiné les supports isolés : un trait est un ensemble de méthodes que les types peuvent implémenter. Les traits peuvent également être utilisés dans des situations où plusieurs types doivent fonctionner ensemble. Ils peuvent décrire les relations entre les types.

- Le `std::iter::Iterator` trait lie chaque type d'itérateur au type de valeur qu'il produit.
- Le `std::ops::Mul` trait concerne des types qui peuvent être multipliés. Dans l'expression `a * b`, les valeurs `a` et `b` peuvent être du même type ou de types différents.
- La `rand` caisse comprend à la fois un trait pour les générateurs de nombres aléatoires (`rand::Rng`) et un trait pour les types qui peuvent être générés aléatoirement (`rand::Distribution`). Les traits eux-mêmes définissent exactement comment ces types fonctionnent ensemble.

Vous n'aurez pas besoin de créer des traits comme ceux-ci tous les jours, mais vous les rencontrerez dans la bibliothèque standard et dans des caisses tierces. Dans cette section, nous montrerons comment chacun de ces exemples est implémenté, en sélectionnant les fonctionnalités pertinentes du langage Rust au fur et à mesure que nous en avons besoin. La compétence clé ici est la capacité de lire les traits et les signatures de méthode et de comprendre ce qu'ils disent sur les types impliqués.

Types associés (ou fonctionnement des itérateurs)

Nous allons commencer par les itérateurs. À présent, chaque langage orienté objet a une sorte de support intégré pour les itérateurs, des objets qui représentent le parcours d'une séquence de valeurs.

La rouille a un `Iterator` trait standard, défini comme ceci :

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

La première caractéristique de ce trait, `type Item;`, est un *type associé*. Chaque type qui implémente `Iterator` doit spécifier le type d'élément qu'il produit.

La deuxième fonctionnalité, la `next()` méthode, utilise le type associé dans sa valeur de retour. `next()` Retourne un `Option<Self::Item>` : soit `Some(item)`, la valeur suivante dans la séquence, soit `None` lorsqu'il n'y a plus de valeurs à visiter. Le type s'écrit `Self::Item`, pas simplement plain `Item`, car il `Item` s'agit d'une fonctionnalité de chaque type d'itérateur, et non d'un type autonome. Comme toujours, `self` et le `Self` type

apparaît explicitement dans le code partout où leurs champs, méthodes, etc. sont utilisés.

Voici à quoi cela ressemble à implémenter `Iterator` pour un type :

```
// (code from the std::env standard library module)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` est le type d'itérateur renvoyé par la fonction de bibliothèque standard `std::env::args()` que nous avons utilisée au [chapitre 2](#) pour accéder aux arguments de la ligne de commande. Il produit des `String` valeurs, donc le `impl` déclare `type Item = String;`.

Généralement ce code peut utiliser des types associés :

```
/// Loop over an iterator, storing the values in a new vector.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

Dans le corps de cette fonction, Rust déduit le type de `value` pour nous, ce qui est bien ; mais nous devons préciser le type de retour de `collect_into_vector`, et le `Item` type associé est le seul moyen de le faire. (`Vec<I>` serait tout simplement faux : nous prétendrions renvoyer un vecteur d'itérateurs !)

L'exemple précédent n'est pas du code que vous écririez vous-même, car après avoir lu le [chapitre 15](#), vous saurez que les itérateurs ont déjà une méthode standard qui fait cela : `iter.collect()`. Prenons donc un autre exemple avant de poursuivre :

```
/// Print out all the values produced by an iterator
fn dump<I>(iter: I)
    where I:Iterator
{
```

```

    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // error
    }
}

```

Cela fonctionne presque. Il n'y a qu'un seul problème : `value` ce n'est peut-être pas un type imprimable.

```

error: `<I as Iterator>::Item` doesn't implement `Debug`
|
8 |     println!("{}: {:?}", index, value); // error
|           ^
|           `<I as Iterator>::Item` cannot be formatted
|           using `{:?}` because it doesn't implement `Deb
|
= help: the trait `Debug` is not implemented for `<I as Iterator>::Item`
= note: required by `std::fmt::Debug::fmt`
help: consider further restricting the associated type
|
5 |     where I: Iterator, <I as Iterator>::Item: Debug
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

Le message d'erreur est légèrement obscurci par l'utilisation par Rust de la syntaxe `<I as Iterator>::Item`, qui est une manière explicite mais verbeuse de dire `I::Item`. Il s'agit d'une syntaxe Rust valide, mais vous aurez rarement besoin d'écrire un type de cette façon.

L'essentiel du message d'erreur est que pour faire compiler cette fonction générique, nous devons nous assurer que `I::Item` implémente le `Debug` trait, le trait pour formater les valeurs avec `{:?}`. Comme le message d'erreur le suggère, nous pouvons le faire en plaçant une borne sur `I::Item` :

```

use std:: fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I:: Item:Debug
{
    ...
}

```

Ou, nous pourrions écrire, " `I` doit être un itérateur sur les `String` valeurs":

```

fn dump<I>(iter: I)
    where I:Iterator<Item=String>

```

```
{  
    ...  
}
```

`Iterator<Item=String>` est lui-même un trait. Si vous considérez `Iterator` comme l'ensemble de tous les types d'itérateurs, alors `Iterator<Item=String>` est un sous-ensemble de `Iterator`: l'ensemble des types d'itérateurs qui produisent `String`s. Cette syntaxe peut être utilisée partout où le nom d'un trait peut être utilisé, y compris les types d'objet trait :

```
fn dump(iter:&mut dyn Iterator<Item=String>) {  
    for (index, s) in iter.enumerate() {  
        println!("{}: {:?}", index, s);  
    }  
}
```

Les traits avec des types associés, comme `Iterator`, sont compatibles avec les méthodes de trait, mais seulement si tous les types associés sont épelés, comme illustré ici. Sinon, le type de `s` pourrait être n'importe quoi, et encore une fois, Rust n'aurait aucun moyen de vérifier le type de ce code.

Nous avons montré de nombreux exemples impliquant des itérateurs. C'est difficile de ne pas le faire; ils sont de loin l'utilisation la plus importante des types associés. Mais les types associés sont généralement utiles lorsqu'un trait doit couvrir plus que de simples méthodes :

- Dans une bibliothèque de pool de threads, un `Task` trait, représentant une unité de travail, peut avoir un `Output` type associé.
- Un `Pattern` trait, représentant une manière de rechercher une chaîne, peut avoir un `Match` type associé, représentant toutes les informations recueillies en faisant correspondre le modèle à la chaîne :

```
trait Pattern {  
    type Match;  
  
    fn search(&self, string: &str) -> Option<Self::Match>;  
}  
  
/// You can search a string for a particular character.  
impl Pattern for char {  
    /// A "match" is just the location where the  
    /// character was found.  
    type Match = usize;
```

```

fn search(&self, string: &str) ->Option<usize> {
    ...
}

```

Si vous êtes familier avec les expressions régulières, il est facile de voir comment `impl Pattern for RegExp` aurait un `Match` type plus élaboré, probablement une structure qui inclurait le début et la longueur de la correspondance, les emplacements où les groupes entre parenthèses correspondent, etc.

- Une bibliothèque pour travailler avec des bases de données relationnelles peut avoir un `DatabaseConnection` trait avec des types associés représentant des transactions, des curseurs, des instructions préparées, etc.

Les types associés sont parfaits pour les cas où chaque implémentation a *un* type lié spécifique : chaque type de `Task` produit un type particulier de `Output` ; chaque type de `Pattern` recherche un type particulier de `Match`. Cependant, comme nous le verrons, certaines relations entre les types ne sont pas comme ça.

Traits génériques (ou comment fonctionne la surcharge d'opérateur)

Multiplication dans Rust utilise ce trait :

```

/// std::ops::Mul, the trait for types that support `*`.
pub trait Mul<RHS> {
    /// The resulting type after applying the `*` operator
    type Output;

    /// The method for the `*` operator
    fn mul(self, rhs: RHS) -> Self::Output;
}

```

`Mul` est un générique caractéristique. Le paramètre de type, `RHS`, est l'abréviation de *righthand side*.

Le paramètre de type signifie ici la même chose que sur une structure ou une fonction : `Mul` est un trait générique, et ses instances `Mul<f64>`, `Mul<String>`, `Mul<Size>`, etc., sont tous des traits différents, tout comme `min::<i32>` et `min::<String>` sont des fonctions différentes et `Vec<i32>` et `Vec<String>` sont des types différents.

Un seul type, par exemple, `WindowSize` peut implémenter à la fois `Mul<f64>` et `Mul<i32>`, et bien d'autres. Vous seriez alors en mesure de multiplier a `windowSize` par de nombreux autres types. Chaque implémentation aurait son propre `Output` type associé.

Les traits génériques bénéficient d'une dispense spéciale en ce qui concerne la règle des orphelins : vous pouvez implémenter un trait étranger pour un type étranger, tant que l'un des paramètres de type du trait est un type défini dans la caisse actuelle. Donc, si vous avez défini `WindowSize`, vous pouvez implémenter `Mul<WindowSize> for f64`, même si vous n'avez défini ni `Mul` ni `f64`. Ces implémentations peuvent même être génériques, telles que `impl<T> Mul<WindowSize> for Vec<T>`. Cela fonctionne parce qu'il n'y a aucun moyen qu'une autre caisse puisse définir `Mul<WindowSize>` quoi que ce soit, et donc aucun conflit entre les implémentations ne pourrait survenir. (Nous avons introduit la règle des orphelins dans ["Traits et autres types de personnes"](#).) C'est ainsi que les caisses `nalgebra` définissent les opérations arithmétiques sur les vecteurs.

Le trait montré plus tôt manque un détail mineur. Le vrai `Mul` trait ressemble à ceci:

```
pub trait Mul<RHS=Self> {  
    ...  
}
```

La syntaxe `RHS=Self` signifie que `RHS` la valeur par défaut est `Self`. Si j'écris `impl Mul for Complex`, sans spécifier `Mul` le paramètre de type de, cela signifie `impl Mul<Complex> for Complex`. Dans une limite, si j'écris `where T: Mul`, cela signifie `where T: Mul<T>`.

Dans Rust, l'expression `lhs * rhs` est un raccourci pour `Mul::mul(lhs, rhs)`. Donc, surcharger l'`*` opérateur dans Rust est aussi simple que d'implémenter le `Mul` trait. Nous montrerons des exemples dans le chapitre suivant.

Trait de mise en œuvre

Comme vous pouvez l'imaginer, les combinaisons de nombreux types génériques peuvent devenir désordonnés. Par exemple, la combinaison de quelques itérateurs à l'aide de combinatoires de bibliothèque standard transforme rapidement votre type de retour en une horreur :

```

use std::iter;
use std::vec::IntoIter;
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
    v.into_iter().chain(u.into_iter()).cycle()
}

```

Nous pourrions facilement remplacer ce type de retour poilu par un objet trait :

```

fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}

```

Cependant, prendre la surcharge de la répartition dynamique et une allocation de tas inévitable à chaque fois que cette fonction est appelée juste pour éviter une signature de type laide ne semble pas être un bon échange, dans la plupart des cas.

Rust a une fonctionnalité appelée `impl Trait` conçue précisément pour cette situation. `impl Trait` permet d'"effacer" le type d'une valeur de retour, en spécifiant uniquement le ou les traits qu'elle implémente, sans dispatch dynamique ni allocation de tas :

```

fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}

```

Désormais, plutôt que de spécifier un type imbriqué particulier de structures combinatrices d'itérateurs, `cyclical_zip` la signature de indique simplement qu'elle renvoie une sorte d'itérateur sur `u8`. Le type de retour exprime l'intention de la fonction, plutôt que ses détails d'implémentation.

Cela a définitivement nettoyé le code et l'a rendu plus lisible, mais `impl Trait` c'est plus qu'un simple raccourci pratique. Utiliser `impl Trait` signifie que vous pouvez modifier le type réel renvoyé à l'avenir tant qu'il implémente toujours `Iterator<Item=u8>`, et tout code appelaient la fonction continuera à se compiler sans problème. Cela offre une grande flexibilité aux auteurs de bibliothèques, car seule la fonctionnalité pertinente est encodée dans la signature de type.

Par exemple, si la première version d'une bibliothèque utilise des combinatoires itérateurs comme dans le précédent, mais qu'un meilleur algo-

rithme pour le même processus est découvert, l'auteur de la bibliothèque peut utiliser différents combinatoires ou même créer un type personnalisé qui implémente `Iterator`, et les utilisateurs de la bibliothèque peuvent obtenir les améliorations de performances sans changer du tout leur code.

Il peut être tentant d'utiliser `impl Trait` pour approximer une version distribuée statiquement du modèle de fabrique couramment utilisé dans les langages orientés objet. Par exemple, vous pouvez définir un trait comme celui-ci :

```
trait Shape {  
    fn new() -> Self;  
    fn area(&self) -> f64;  
}
```

Après l'avoir implémenté pour quelques types, vous souhaiterez peut-être utiliser différents `Shape`s en fonction d'une valeur d'exécution, comme une chaîne saisie par un utilisateur. Cela ne fonctionne pas avec `impl Shape` comme type de retour :

```
fn make_shape(shape: &str) -> impl Shape {  
    match shape {  
        "circle" => Circle::new(),  
        "triangle" => Triangle::new(), // error: incompatible types  
        "shape" => Rectangle::new(),  
    }  
}
```

Du point de vue de l'appelant, une fonction comme celle-ci n'a pas beaucoup de sens. `impl Trait` est une forme de répartition statique, de sorte que le compilateur doit connaître le type renvoyé par la fonction au moment de la compilation afin d'allouer la bonne quantité d'espace sur la pile et d'accéder correctement aux champs et aux méthodes de ce type. Ici, il pourrait s'agir de `Circle`, `Triangle`, ou `Rectangle`, qui pourraient tous occuper des quantités d'espace différentes et tous avoir des implémentations différentes de `area()`.

Il est important de noter que Rust n'autorise pas les méthodes de trait à utiliser des `impl Trait` valeurs de retour. La prise en charge de cela nécessitera quelques améliorations dans le système de type des langues. Tant que ce travail n'est pas terminé, seules les fonctions libres et les fonctions associées à des types spécifiques peuvent utiliser des `impl Trait` retours.

`impl` Trait peut également être utilisé dans des fonctions qui acceptent des arguments génériques. Par exemple, considérons cette simple fonction générique :

```
fn print<T: Display>(val:T) {
    println!("{}", val);
}
```

Il est identique à cette version utilisant `impl Trait`:

```
fn print(val:impl Display) {
    println!("{}", val);
}
```

Il existe une exception importante. L'utilisation de génériques permet aux appelants de la fonction de spécifier le type des arguments génériques, comme `print::<i32>(42)`, tandis que l'utilisation `impl Trait` ne le permet pas.

Chaque `impl Trait` argument se voit attribuer son propre paramètre de type anonyme, donc `impl Trait` pour les arguments est limité aux seules fonctions génériques les plus simples, sans relations entre les types d'arguments.

Const associés

Comme les structures et les énumérations, les traits peuvent avoir des constantes associées. Vous pouvez déclarer un trait avec une constante associée en utilisant la même syntaxe que pour une structure ou une énumération :

```
trait Greet {
    const GREETING: &'static str = "Hello";
    fn greet(&self) ->String;
}
```

Les constantes associées aux traits ont cependant un pouvoir spécial. Comme les types et fonctions associés, vous pouvez les déclarer mais pas leur donner de valeur :

```
trait Float {
    const ZERO: Self;
    const ONE:Self;
}
```

Ensuite, les implémenteurs du trait peuvent définir ces valeurs :

```
impl Float for f32 {
    const ZERO: f32 = 0.0;
    const ONE:f32 = 1.0;
}

impl Float for f64 {
    const ZERO: f64 = 0.0;
    const ONE:f64 = 1.0;
}
```

Cela vous permet d'écrire du code générique qui utilise ces valeurs :

```
fn add_one<T: Float + Add<Output=T>>(value: T) -> T {
    value + T::ONE
}
```

Notez que les constantes associées ne peuvent pas être utilisées avec des objets trait, car le compilateur s'appuie sur les informations de type concernant l'implémentation afin de choisir la bonne valeur au moment de la compilation.

Même un simple trait sans aucun comportement, comme `Float`, peut donner suffisamment d'informations sur un type, en combinaison avec quelques opérateurs, pour implémenter des fonctions mathématiques courantes comme Fibonacci :

```
fn fib<T: Float + Add<Output=T>>(n: usize) -> T {
    match n {
        0 => T::ZERO,
        1 => T::ONE,
        n => fib::<T>(n - 1) + fib::<T>(n - 2)
    }
}
```

Dans les deux dernières sections, nous avons montré différentes manières dont les traits peuvent décrire les relations entre les types. Tous ces éléments peuvent également être considérés comme des moyens d'éviter les surcharges et les downcasts des méthodes virtuelles, car ils permettent à Rust de connaître des types plus concrets au moment de la compilation..

Limites de la rétro-ingénierie

Ecrire du code générique peut être une véritable corvée lorsqu'il n'y a pas de trait unique qui fait tout ce dont vous avez besoin. Supposons que nous ayons écrit cette fonction non générique pour effectuer des calculs :

```
fn dot(v1: &[i64], v2: &[i64]) ->i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Maintenant, nous voulons utiliser le même code avec des valeurs à virgule flottante. Nous pourrions essayer quelque chose comme ceci :

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total:N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Pas de chance : Rust se plaint de l'utilisation `*` et du type de `0`. Nous pouvons exiger `N` d'être un type qui supporte `+` et `*` utilise les traits `Add` et `Mul`. Notre utilisation de `0` doit changer, cependant, car `0` est toujours un entier dans Rust ; la valeur à virgule flottante correspondante est `0.0`. Heureusement, il existe un `Default` trait standard pour les types qui ont des valeurs par défaut. Pour les types numériques, la valeur par défaut est toujours 0 :

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Ceci est plus proche, mais ne fonctionne toujours pas tout à fait:

```

error: mismatched types
|
5 | fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
|     - this type parameter
...
8 |         total = total + v1[i] * v2[i];
|                         ^^^^^^^^^^^^^ expected type parameter `N`,
|                                         found associated type
|
= note: expected type parameter `N`
        found associated type `<N as Mul>::Output`
help: consider further restricting this bound
|
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2: &[N]) ->
|                         ^^^^^^^^^^^^^^^^^^

```

Notre nouveau code suppose que la multiplication de deux valeurs de type `N` produit une autre valeur de type `N`. Ce n'est pas nécessairement le cas. Vous pouvez surcharger l'opérateur de multiplication pour renvoyer le type de votre choix. Nous devons en quelque sorte dire à Rust que cette fonction générique ne fonctionne qu'avec des types qui ont la saveur normale de la multiplication, où la multiplication `N * N` renvoie un `N`. La suggestion dans le message d'erreur est *presque* juste : nous pouvons le faire en remplaçant `Mul` par `Mul<Output=N>`, et de même pour `Add` :

```

fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}

```

À ce stade, les limites commencent à s'accumuler, ce qui rend le code difficile à lire. Déplaçons les bornes dans une `where` clause :

```

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N:Add<Output=N> + Mul<Output=N> + Default
{
    ...
}

```

Super. Mais Rust se plaint toujours de cette ligne de code :

```

error: cannot move out of type `[N]`, a non-copy slice
|
8 |     total = total + v1[i] * v2[i];
|           ^^^^
|
|           |

```

```
|                                     cannot move out of here
|                                     move occurs because `v1[_]` has type `N`,
|                                     which does not implement the `Copy` trait
```

Puisque nous n'avons pas besoin `N` d'être un type copiable, Rust interprète `v1[i]` comme une tentative de déplacer une valeur hors de la tranche, ce qui est interdit. Mais nous ne voulons pas du tout modifier la tranche ; nous voulons juste copier les valeurs pour les opérer. Heureusement, tous les types numériques intégrés de Rust implémentent `Copy`, nous pouvons donc simplement ajouter cela à nos contraintes sur `N` :

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

Avec cela, le code se compile et s'exécute. Le code final ressemble à ceci :

```
use std:: ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

Cela arrive parfois dans Rust : il y a une période d'intenses discussions avec le compilateur, à la fin de laquelle le code a l'air plutôt sympa, comme s'il avait été un jeu d'enfant à écrire, et s'exécute à merveille.

Ce que nous avons fait ici, c'est rétro-concevoir les limites sur `N`, en utilisant le compilateur pour guider et vérifier notre travail. La raison pour laquelle c'était un peu pénible est qu'il n'y avait pas un seul `Number` trait dans la bibliothèque standard qui incluait tous les opérateurs et méthodes que nous voulions utiliser. Il se trouve qu'il existe une caisse open source populaire appelée `num` qui définit un tel trait! Si nous l'avions su, nous aurions pu ajouter `num` à notre `Cargo.toml` et écrire :

```

use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

```

Tout comme dans la programmation orientée objet, la bonne interface rend tout agréable, dans la programmation générique, le bon trait rend tout agréable.

Pourtant, pourquoi se donner tout ce mal ? Pourquoi les concepteurs de Rust n'ont-ils pas fait en sorte que les génériques ressemblent davantage à des modèles C++, où les contraintes sont laissées implicites dans le code, à la "duck typing" ?

L'un des avantages de l'approche de Rust est la compatibilité ascendante du code générique. Vous pouvez modifier l'implémentation d'une fonction ou d'une méthode générique publique, et si vous n'avez pas modifié la signature, vous n'avez cassé aucun de ses utilisateurs.

Un autre avantage des bornes est que lorsque vous obtenez une erreur du compilateur, au moins le compilateur peut vous dire où se trouve le problème. Les messages d'erreur du compilateur C++ impliquant des modèles peuvent être beaucoup plus longs que ceux de Rust, pointant vers de nombreuses lignes de code différentes, car le compilateur n'a aucun moyen de dire qui est à blâmer pour un problème : le modèle, ou son appelant, qui peut également être un modèle, ou l'appelant de *ce* modèle...

Peut-être que l'avantage le plus important d'écrire explicitement les limites est simplement qu'elles sont là, dans le code et dans la documentation. Vous pouvez regarder la signature d'une fonction générique dans Rust et voir exactement quel type d'arguments elle accepte. On ne peut pas en dire autant des modèles. Le travail de documentation complète des types d'arguments dans les bibliothèques C++ comme Boost est encore *plus* ardu que ce que nous avons vécu ici. Les développeurs Boost n'ont pas de compilateur qui vérifie leur travail.

Traits en tant que fondation

Les traits sont l'une des principales caractéristiques d'organisation de Rust, et pour cause. Il n'y a rien de mieux pour concevoir un programme ou une bibliothèque qu'une bonne interface.

Ce chapitre était un blizzard de syntaxe, de règles et d'explications. Maintenant que nous avons jeté les bases, nous pouvons commencer à parler des nombreuses façons dont les traits et les génériques sont utilisés dans le code Rust. Le fait est que nous avons seulement commencé à gratter la surface. Les deux chapitres suivants couvrent les traits communs fournis par la bibliothèque standard. Les prochains chapitres traitent des fermetures, des itérateurs, des entrées/sorties et de la concurrence. Les traits et les génériques jouent un rôle central dans tous ces sujets.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 12. Surcharge de l'opérateur

Dans le Mandelbrottraceur d'ensembles que nous avons montré au [chapitre 2](#), nous avons utilisé le type `num` de caisse `Complex` pour représenter un nombre sur le plan complexe :

```
#[derive(Clone, Copy, Debug)]  
struct Complex<T> {  
    /// Real portion of the complex number  
    re:T,  
  
    /// Imaginary portion of the complex number  
    im:T,  
}
```

Nous avons pu additionner et multiplier `Complex` des nombres comme n'importe quel type numérique intégré, en utilisant les opérateurs Rust `+` et `*`:

```
z = z * z + c;
```

Vous pouvez créer vos propres types prennent également en charge les opérateurs arithmétiques et autres, simplement en implémentant quelques traits intégrés. C'est ce qu'on appelle la *surcharge d'opérateur* et l'effet ressemble beaucoup à la surcharge d'opérateur en C++, C#, Python et Ruby.

Les traits pour la surcharge des opérateurs appartiennent à quelques catégories en fonction de la partie du langage qu'ils prennent en charge, comme indiqué dans le [Tableau 12-1](#). Dans ce chapitre, nous couvrirons chaque catégorie. Notre objectif n'est pas seulement de vous aider à bien intégrer vos propres types dans le langage, mais aussi de vous donner une meilleure idée de la façon d'écrire des fonctions génériques comme la fonction de produit scalaire décrite dans ["Reverse-Engineering Bounds"](#) qui fonctionnent sur les types le plus naturellement utilisé via ces opérateurs. Le chapitre devrait également donner un aperçu de la façon dont certaines fonctionnalités du langage lui-même sont implémentées.

Tableau 12-1. Résumé des caractéristiques de surcharge d'opérateur

Catégorie	Caractéristique	Opérateur
Opérateurs unaires	<code>std::ops::Ne</code> <code>g</code>	<code>-x</code>
	<code>std::ops::No</code> <code>t</code>	<code>!x</code>
Opérateurs arithmétiques	<code>std::ops::Ad</code> <code>d</code>	<code>x + y</code>
	<code>std::ops::Su</code> <code>b</code>	<code>x - y</code>
	<code>std::ops::Mu</code> <code>l</code>	<code>x * y</code>
	<code>std::ops::Di</code> <code>v</code>	<code>x / y</code>
	<code>std::ops::Re</code> <code>m</code>	<code>x % y</code>
Opérateurs au niveau du bit	<code>std::ops::Bi</code> <code>tAnd</code>	<code>x & y</code>
	<code>std::ops::Bi</code> <code>tOr</code>	<code>x y</code>
	<code>std::ops::Bi</code> <code>tXor</code>	<code>x ^ y</code>
	<code>std::ops::Sh</code> <code>l</code>	<code>x << y</code>
	<code>std::ops::Sh</code> <code>r</code>	<code>x >> y</code>
Opérateurs arithmétiques d'affectation composée	<code>std::ops::Ad</code> <code>dAssign</code>	<code>x += y</code>

Catégorie	Caractéristique	Opérateur
	std::ops::Su bAssign	x -= y
	std::ops::Mu lAssign	x *= y
	std::ops::Di vAssign	x /= y
	std::ops::Re mAssign	x %= y
Opérateurs bit à bit d' affectation composée	std::ops::Bi tAndAssign	x &= y
	std::ops::Bi tOrAssign	x = y
	std::ops::Bi tXorAssign	x ^= y
	std::ops::Sh lAssign	x <= y
	std::ops::Sh rAssign	x >= y
Comparaison	std::cmp::Pa rtialEq	x == y, x != y
	std::cmp::Pa rtialOrd	x < y, x <= y, x > y, x >= y
Indexage	std::ops::In dex	x[y], &x[y]
	std::ops::In dexMut	x[y] = z, &mut x[y]

Opérateurs arithmétiques et binaires

En rouille, l'expression `a + b` est en fait un raccourci pour `a.add(b)`, un appel à la méthode du trait `add` de la bibliothèque standard .

`std::ops::Add` Les types numériques standard de Rust implémentent tous `std::ops::Add`. Pour que l'expression `a + b` fonctionne pour `Complex` les valeurs, la `num` caisse implémente `Complex` également ce trait pour. Des traits similaires couvrent les autres opérateurs : `a * b` est un raccourci pour `a.mul(b)`, une méthode du `std::ops::Mul` trait, `std::ops::Neg` couvre l'opérateur de négation du préfixe `-`, etc.

Si vous voulez essayer d'écrire `z.add(c)`, vous devrez mettre le `Add` trait dans la portée afin que sa méthode soit visible. Cela fait, vous pouvez traiter toutes les opérations arithmétiques comme des appels de fonction :¹

```
use std:: ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

Voici la définition de `std::ops::Add`:

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

En d'autres termes, le trait `Add<T>` est la capacité d'ajouter une `T` valeur à vous-même. Par exemple, si vous souhaitez pouvoir ajouter des valeurs `i32` et à votre type, votre type doit implémenter à la fois et . Le paramètre de type du trait est par défaut , donc si vous implémentez l'addition entre deux valeurs du même type, vous pouvez simplement écrire pour ce cas. Le type associé décrit le résultat de l'addition. `u32 Add<i32> Add<u32> Rhs Self Add Output`

Par exemple, pour pouvoir additionner `Complex<i32>` des valeurs, `Complex<i32>` il faut implémenter `Add<Complex<i32>>`. Puisque nous ajoutons un type à lui-même, nous écrivons simplement `Add` :

```
use std:: ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
```

```

        im: self.im + rhs.im,
    }
}

```

Bien sûr, nous ne devrions pas avoir à implémenter `Add` séparément pour `Complex<i32>`, `Complex<f32>`, `Complex<f64>`, etc. Toutes les définitions auraient exactement la même apparence, à l'exception des types impliqués, nous devrions donc être en mesure d'écrire une seule implémentation générique qui les couvre toutes, tant que le type des composants complexes eux-mêmes prend en charge l'addition :

```

use std:: ops::Add;

impl<T> Add for Complex<T>
where
    T: Add<Output = T>,
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}

```

En écrivant `where T: Add<Output=T>`, nous restreignons `T` aux types qui peuvent être ajoutés à eux-mêmes, donnant une autre `T` valeur. C'est une restriction raisonnable, mais nous pourrions encore assouplir les choses : le `Add` trait n'exige pas que les deux opérandes de `+` aient le même type, et il ne contraint pas non plus le type de résultat. Ainsi, une implémentation au maximum générique laisserait les opérandes gauche et droit varier indépendamment et produirait une `Complex` valeur de n'importe quel type de composant produit par l'addition :

```

use std:: ops::Add;

impl<L, R> Add<Complex<R>> for Complex<L>
where
    L: Add<R>,
{
    type Output = Complex<L:: Output>;
    fn add(self, rhs: Complex<R>) -> Self:: Output {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}

```

```

    }
}

```

En pratique, cependant, Rust a tendance à éviter de prendre en charge les opérations de type mixte. Étant donné que notre paramètre de type `L` doit implémenter `Add<R>`, il suit généralement cela `L` et `R` sera du même type : il n'y a tout simplement pas beaucoup de types disponibles pour `L` cet implémentation. Donc, en fin de compte, cette version générique maximale peut ne pas être beaucoup plus utile que la définition générique précédente, plus simple.

Les traits intégrés de Rust pour les opérateurs arithmétiques et binaires se répartissent en trois groupes : les opérateurs unaires, les opérateurs binaires et les opérateurs d'affectation composés. Au sein de chaque groupe, les traits et leurs méthodes ont tous la même forme, nous allons donc couvrir un exemple de chaque.

Opérateurs unaires

De côté à partir de l'opérateur de déréférencement `*`, que nous aborderons séparément dans « [Deref et DerefMut](#) », Rust a deux opérateurs unaires que vous pouvez personnaliser, illustrés dans [le tableau 12-2](#).

Tableau 12-2. Traits intégrés pour les opérateurs unaires

Nom du trait	Expression	Expression équivalente
<code>std::ops::Neg</code>	<code>-x</code>	<code>x.neg()</code>
<code>std::ops::Not</code>	<code>!x</code>	<code>x.not()</code>

Tous les types numériques signés de Rust implémentent `std::ops::Neg`, pour la négation unaireopérateur `-` ; les types entiers et `bool` implement `std::ops::Not`, pour le complément unaireopérateur `!`. Il existe également des implementations pour les références à ces types.

Notez que `!` complète les `bool` valeurs et effectue un complément au niveau du bit (c'est-à-dire inverse les bits) lorsqu'il est appliqué à des entiers ; il joue le rôle à la fois des opérateurs `!` et `~` de C et C++.

Les définitions de ces traits sont simples :

```

trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}

```

Nier un nombre complexe nie simplement chacun de ses composants.
Voici comment nous pourrions écrire une implémentation générique de négation pour les `Complex` valeurs :

```

use std::ops::Neg;

impl<T> Neg for Complex<T>
where
    T: Neg<Output = T>,
{
    type Output = Complex<T>;
    fn neg(self) -> Complex<T> {
        Complex {
            re: -self.re,
            im: -self.im,
        }
    }
}

```

Opérateurs binaires

Le binaire de Rust les opérateurs arithmétiques et au niveau du bit et leurs traits intégrés correspondants apparaissent dans le [tableau 12-3](#).

Tableau 12-3. Traits intégrés pour les opérateurs binaires

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	<code>std::ops::Add</code>	<code>x + y</code>	<code>x.add(y)</code>
	<code>std::ops::Sub</code>	<code>x - y</code>	<code>x.sub(y)</code>
	<code>std::ops::Mul</code>	<code>x * y</code>	<code>x.mul(y)</code>
	<code>std::ops::Div</code>	<code>x / y</code>	<code>x.div(y)</code>
	<code>std::ops::Rem</code>	<code>x % y</code>	<code>x.rem(y)</code>
	<code>std::ops::B itAnd</code>	<code>x & y</code>	<code>x.bitand(y)</code>
	<code>std::ops::B itOr</code>	<code>x y</code>	<code>x.bitor(y)</code>
	<code>std::ops::B itXor</code>	<code>x ^ y</code>	<code>x.bitxor(y)</code>
	<code>std::ops::Shl</code>	<code>x << y</code>	<code>x.shl(y)</code>
	<code>std::ops::Shr</code>	<code>x >> y</code>	<code>x.shr(y)</code>

Tous les types numériques de Rust implémentent les opérateurs arithmétiques. Les types entiers de Rust et `bool` implémentent les opérateurs au niveau du bit. Il existe également des implementations qui acceptent les références à ces types comme opérandes ou les deux.

Tous les traits ont ici la même forme générale. La définition de `std::ops::BitXor`, pour l' `^` opérateur, ressemble à ceci :

```

trait BitXor<Rhs = Self> {
    type Output;
    fn bitxor(self, rhs: Rhs) -> Self::Output;
}

```

Au début de ce chapitre, nous avons également montré `std::ops::Add`, un autre trait de cette catégorie, ainsi que plusieurs exemples d'implémentations.

Vous pouvez utiliser l'opérateur `+` pour concaténer une `String` avec une `&str` ou une autre `String`. Cependant, Rust ne permet pas à l'opérande gauche de `+` d'être un `&str`, pour décourager la construction de longues chaînes en concaténant à plusieurs reprises de petits morceaux sur la gauche. (Cela fonctionne mal, nécessitant un temps quadratique dans la longueur finale de la chaîne.) Généralement, la `write!` macro est meilleur pour construire des cordes morceau par morceau; nous montrons comment procéder dans [« Ajout et insertion de texte »](#).

Opérateurs d'affectation composés

Un composéL'expression d'affectation est semblable à `x += y` ou `x &= y`: elle prend deux opérandes, effectue une opération sur eux comme une addition ou un ET au niveau du bit, et stocke le résultat dans l'opérande de gauche. Dans Rust, la valeur d'une expression d'affectation composée est toujours `()`, jamais la valeur stockée.

De nombreux langages ont des opérateurs comme ceux-ci et les définissent généralement comme des raccourcis pour des expressions telles que `x = x + y` ou `x = x & y`. Cependant, Rust n'adopte pas cette approche. Au lieu de cela, `x += y` est un raccourci pour l'appel de méthode `x.add_assign(y)`, où `add_assign` est la seule méthode du `std::ops::AddAssign` trait :

```

trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}

```

[Le tableau 12-4](#) montre tous les opérateurs d'affectation composés de Rust et les traits intégrés qui les implémentent.

Tableau 12-4. Caractéristiques intégrées pour les opérateurs d'affectation composés

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	std::ops::AddAssign	x += y	x.add_assign(y)
	std::ops::SubAssign	x -= y	x.sub_assign(y)
	std::ops::MulAssign	x *= y	x.mul_assign(y)
	std::ops::DivAssign	x /= y	x.div_assign(y)
	std::ops::RemAssign	x %= y	x.rem_assign(y)
	std::ops::BitAndAssign	x &= y	x.bitand_assign(y)
	std::ops::BitOrAssign	x = y	x.bitor_assign(y)
	std::ops::BitXorAssign	x ^= y	x.bitxor_assign(y)
	std::ops::ShlAssign	x <= y	x.shl_assign(y)
	std::ops::ShrAssign	x >= y	x.shr_assign(y)

Tous les types numériques de Rust implémentent les opérateurs d'affectation composés arithmétiques. Les types entiers de Rust et `bool` implémentent les opérateurs d'affectation composés au niveau du bit.

Une implémentation générique de `AddAssign` pour notre `Complex` type est simple :

```
use std:: ops::AddAssign;
```

```

impl<T> AddAssign for Complex<T>
where
    T: AddAssign<T>,
{
    fn add_assign(&mut self, rhs:Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}

```

Le trait intégré pour un opérateur d'affectation composé est complètement indépendant du trait intégré pour l'opérateur binaire correspondant. La mise en œuvre `std::ops::Add` n'implémente pas automatiquement `std::ops::AddAssign`; si vous voulez que Rust autorise votre type comme opérande gauche d'un `+=` opérateur, vous devez vous `AddAssign` implémenter.

Comparaisons d'équivalence

L'égalité de Rust les opérateurs, `==` et `!=`, sont des raccourcis pour les appels aux `std::cmp::PartialEq` traits `eq` et aux `ne` méthodes :

```

assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));

```

Voici la définition de `std::cmp::PartialEq`:

```

trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}

```

Étant donné que la `ne` méthode a une définition par défaut, il vous suffit de définir `eq` pour implémenter le `PartialEq` trait, voici donc une implémentation complète pour `Complex` :

```

impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}

```

```
    }
}
```

En d'autres termes, pour tout type de composant `T` qui peut lui-même être comparé pour l'égalité, cela implémente la comparaison pour `Complex<T>`. En supposant que nous ayons également implémenté `std::ops::Mul` quelque `Complex` part le long de la ligne, nous pouvons maintenant écrire :

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });
```

Les implémentations de `PartialEq` sont presque toujours de la forme montrée ici : elles comparent chaque champ de l'opérande de gauche au champ correspondant de droite. Ceux-ci deviennent fastidieux à écrire, et l'égalité est une opération courante à prendre en charge, donc si vous le demandez, Rust générera `PartialEq` automatiquement une implémentation de pour vous. Ajoutez simplement à l' attribut `PartialEq` de la définition de type comme ceci : derive

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

L'implémentation générée automatiquement de Rust est essentiellement identique à notre code écrit à la main, comparant tour à tour chaque champ ou élément du type. Rust peut également dériver `PartialEq` des implémentations pour `enum` les types. Naturellement, chacune des valeurs que le type contient (ou pourrait contenir, dans le cas d'un `enum`) doit elle-même implémenter `PartialEq`.

Contrairement aux traits arithmétiques et au niveau du bit, qui prennent leurs opérandes par valeur, `PartialEq` prend ses opérandes par référence. Cela signifie que la comparaison de non `Copy`-valeurs telles que `String`s, `Vec`s ou `HashMap`s ne les déplace pas, ce qui serait gênant :

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x691".to_string();
assert!(s == t); // s and t are only borrowed...

// ... so they still have their values here.
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

Cela nous amène à la limite du trait sur le `Rhs` paramètre de type, qui est d'un genre que nous n'avons jamais vu auparavant :

```
where
Rhs : ?Sized,
```

Cela assouplit l'exigence habituelle de Rust selon laquelle les paramètres de type doivent être des types dimensionnés, nous permettant d'écrire des traits comme `PartialEq<str>` ou `PartialEq<[T]>`. Les méthodes `eq` et `ne` prennent des paramètres de type `&Rhs`, et comparer quelque chose avec `a &str` ou `a &[T]` est tout à fait raisonnable. Depuis `str` implemente `PartialEq<str>`, les assertions suivantes sont équivalentes :

```
assert!("ungula" != "ungulate");
assert!("ungula".ne("ungulate"));
```

Ici, les deux `self` et `Rhs` seraient le type non dimensionné `str`, faisant `ne` de `self` et `rhs` les paramètres les deux `&str` valeurs. Nous discuterons des types dimensionnés, des types non dimensionnés et du `Sized` trait en détail dans "[Sized](#)".

Pourquoi appelle-t-on ce trait `PartialEq`? La définition mathématique traditionnelle d'une *relation d'équivalence*, dont l'égalité est une instance, impose trois exigences. Pour toutes valeurs `x` et `y`:

- Si `x == y` est vrai, alors `y == x` doit être vrai aussi. En d'autres termes, l'échange des deux côtés d'une comparaison d'égalité n'affecte pas le résultat.
- Si `x == y` et `y == z`, alors ce doit être le cas que `x == z`. Étant donné n'importe quelle chaîne de valeurs, chacune égale à la suivante, chaque valeur de la chaîne est directement égale à toutes les autres. L'égalité est contagieuse.
- Il doit toujours être vrai que `x == x`.

Cette dernière exigence peut sembler trop évidente pour être énoncée, mais c'est exactement là que les choses tournent mal. Rust `f32` et `f64` sont des valeurs à virgule flottante standard IEEE. Selon cette norme, les expressions comme `0.0/0.0` et autres sans valeur appropriée doivent produire un *non-nombre spécial* valeurs, généralement appelées valeurs `Nan`. La norme exige en outre qu'une valeur `Nan` soit traitée comme inégale à toute autre valeur, y compris elle-même. Par exemple, la norme exige tous les comportements suivants :

```
assert!(f64::is_nan(0.0 / 0.0));
assert_eq!(0.0 / 0.0 == 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 != 0.0 / 0.0, true);
```

De plus, toute comparaison ordonnée avec une valeur NaN doit retourner false :

```
assert_eq!(0.0 / 0.0 < 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 > 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 <= 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 >= 0.0 / 0.0, false);
```

Ainsi, alors que l'opérateur de Rust `==` répond aux deux premières exigences pour les relations d'équivalence, il ne répond clairement pas à la troisième lorsqu'il est utilisé sur des valeurs à virgule flottante IEEE. C'est ce qu'on appelle une *relation d'équivalence partielle*, donc Rust utilise le nom du trait intégré de `PartialEq` l'`==` opérateur. Si vous écrivez du code générique avec des paramètres de type connus uniquement pour être `PartialEq`, vous pouvez supposer que les deux premières conditions sont remplies, mais vous ne devez pas supposer que les valeurs sont toujours égales à elles-mêmes.

Cela peut être un peu contre-intuitif et peut entraîner des bogues si vous n'êtes pas vigilant. Si vous préférez que votre code générique exige une relation d'équivalence complète, vous pouvez à la place utiliser le `std::cmp::Eq` trait comme une limite, qui représente une relation d'équivalence complète : si un type implémente `Eq`, alors `x == x` doit être `true` pour chaque valeur `x` de ce type. En pratique, presque tous les types qui implémentent `PartialEq` devraient `Eq` également implémenter ; `f32` et `f64` sont les seuls types de la bibliothèque standard qui sont, `PartialEq` mais pas `Eq`.

La bibliothèque standard définit `Eq` comme une extension de `PartialEq`, n'ajoutant aucune nouvelle méthode :

```
trait Eq:PartialEq<Self> { }
```

Si votre type est `PartialEq` et que vous souhaitez qu'il le soit `Eq` également, vous devez implémenter explicitement `Eq`, même si vous n'avez pas besoin de définir de nouvelles fonctions ou de nouveaux types pour le faire. La mise en œuvre `Eq` pour notre `Complex` type est donc rapide :

```
impl<T:Eq> Eq for Complex<T> { }
```

Nous pourrions l'implémenter encore plus succinctement en incluant simplement `Eq` dans l' `derive` attribut sur la `Complex` définition de type :

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

Les implémentations dérivées sur un type générique peuvent dépendre des paramètres de type. Avec l' `derive` attribut, `Complex<i32>` implémenterait `Eq`, parce que `i32` le fait, mais `Complex<f32>` implémenterait uniquement `PartialEq`, puisque `f32` n'implémente pas `Eq`.

Lorsque vous implémentez `std::cmp::PartialEq` vous-même, Rust ne peut pas vérifier que vos définitions pour les méthodes `eq` et `ne` se comportent réellement comme requis pour une équivalence partielle ou totale. Ils pourraient faire tout ce que vous voulez. Rust vous croit simplement sur parole que vous avez implémenté l'égalité d'une manière qui répond aux attentes des utilisateurs du trait.

Bien que la définition de `PartialEq` fournit une définition par défaut pour `ne`, vous pouvez fournir votre propre implémentation si vous le souhaitez. Cependant, vous devez vous assurer que `ne` et `eq` sont des compléments exacts l'un de l'autre. Les utilisateurs du `PartialEq` trait supposeront qu'il en est ainsi.

Comparaisons ordonnées

Rust spécifie le comportement de la commande opérateurs de comparaison `<`, `>`, `<=`, et `>=` tous en termes d'un seul trait,

`std::cmp::PartialOrd`:

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Notez que `PartialOrd<Rhs>` s'étend `PartialEq<Rhs>` : vous ne pouvez effectuer des comparaisons ordonnées que sur des types que vous pouvez également comparer pour l'égalité.

La seule méthode que `PartialOrd` vous devez implémenter vous-même est `partial_cmp`. Lorsque `partial_cmp` renvoie `Some(o)`, alors `o` indique `self` la relation de `other`:

```
enum Ordering {
    Less,           // self < other
    Equal,          // self == other
    Greater,        // self > other
}
```

Mais si `partial_cmp` renvoie `None`, cela signifie que `self` et `other` ne sont pas ordonnés l'un par rapport à l'autre : aucun n'est plus grand que l'autre, ni égal. Parmi tous les types primitifs de Rust, seules les comparaisons entre des valeurs à virgule flottante sont renvoyées `None` : en particulier, la comparaison d'une valeur `Nan` (pas un nombre) avec n'importe quoi d'autre renvoie `None`. Nous donnons plus d'informations sur les valeurs de `Nan` dans "[Comparaisons d'équivalence](#)".

Comme les autres opérateurs binaires, pour comparer des valeurs de deux types `Left` et `Right`, `Left` il faut implémenter `PartialOrd<Right>`. Des expressions telles que `x < y` ou `x >= y` sont des raccourcis pour les appels de `PartialOrd` méthodes, comme indiqué dans le [Tableau 12-5](#).

Tableau 12-5. Opérateurs et `PartialOrd` méthodes de comparaison ordonnée

Expression	Appel de méthode équivalent	Définition par défaut
<code>x < y</code>	<code>x.lt(y)</code>	<code>x.partial_cmp(&y) == Some(Less)</code>
<code>x > y</code>	<code>x.gt(y)</code>	<code>x.partial_cmp(&y) == Some(Greater)</code>
<code>x <= y</code>	<code>x.le(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Less Equal))</code>
<code>x >= y</code>	<code>x.ge(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Greater Equal))</code>

Comme dans les exemples précédents, le code d'appel de méthode équivalent indiqué suppose que `std::cmp::PartialOrd` et `std::cmp::Ordering` sont dans la portée.

Si vous savez que les valeurs de deux types sont toujours ordonnées l'une par rapport à l'autre, vous pouvez implémenter le `std::cmp::Ord` trait plus strict :

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

La `cmp` méthode ici renvoie simplement un `Ordering`, au lieu d'un `Option<Ordering>` like `partial_cmp`: `cmp` déclare toujours ses arguments égaux ou indique leur ordre relatif. Presque tous les types qui implémentent `PartialOrd` devraient également implémenter `Ord`. Dans la bibliothèque standard, `f32` et `f64` sont les seules exceptions à cette règle.

Puisqu'il n'y a pas d'ordre naturel sur les nombres complexes, nous ne pouvons pas utiliser notre `Complex` type des sections précédentes pour montrer un exemple d'implémentation de `PartialOrd`. Au lieu de cela, supposons que vous travaillez avec le type suivant, représentant l'ensemble des nombres compris dans un intervalle semi-ouvert donné :

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
```

```

        lower: T, // inclusive
        upper:T, // exclusive
    }
}

```

Vous aimeriez que les valeurs de ce type soient partiellement ordonnées : un intervalle est inférieur à un autre s'il tombe entièrement avant l'autre, sans chevauchement. Si deux intervalles inégaux se chevauchent, ils ne sont pas ordonnés : un élément de chaque côté est inférieur à un élément de l'autre. Et deux intervalles égaux sont simplement égaux. L'implémentation suivante de `PartialOrd` implémente ces règles :

```

use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other {
            Some(Ordering::Equal)
        } else if self.lower >= other.upper {
            Some(Ordering::Greater)
        } else if self.upper <= other.lower {
            Some(Ordering::Less)
        } else {
            None
        }
    }
}

```

Avec cette implémentation en place, vous pouvez écrire ce qui suit :

```

assert!(Interval { lower: 10, upper: 20 } < Interval { lower: 20, upper: 40 });
assert!(Interval { lower: 7, upper: 8 } >= Interval { lower: 0, upper: 1 });
assert!(Interval { lower: 7, upper: 8 } <= Interval { lower: 7, upper: 8 });

// Overlapping intervals aren't ordered with respect to each other.
let left = Interval { lower: 10, upper: 30 };
let right = Interval { lower: 20, upper: 40 };
assert!(!(left < right));
assert!(!(left >= right));

```

Alors que `PartialOrd` c'est ce que vous verrez habituellement, les classements totaux définis avec `Ord` sont nécessaires dans certains cas, comme les méthodes de tri implémentées dans la bibliothèque standard. Par exemple, le tri des intervalles n'est pas possible avec seulement une `PartialOrd` implémentation. Si vous voulez les trier, vous devrez combler les vides des caisses non ordonnées. Vous voudrez peut-être trier par limite supérieure, par exemple, et c'est facile de le faire avec `sort_by_key`:

```
intervals.sort_by_key(| i| i.upper);
```

Le `Reverse` type wrapper en profite en implémentant `Ord` une méthode qui inverse simplement tout ordre. Pour tout type `T` qui implémente `Ord`, `std::cmp::Reverse<T>` implémente `Ord` aussi, mais avec un ordre inversé. Par exemple, trier nos intervalles de haut en bas par borne inférieure est simple:

```
use std::cmp::Reverse;
intervals.sort_by_key(| i| Reverse(i.lower));
```

Index et IndexMut

Vous pouvez spécifier comment une indexation expression like `a[i]` fonctionne sur votre type en implémentant les traits `std::ops::Index` et `std::ops::IndexMut`. Les tableaux prennent `[]` directement en charge l'opérateur, mais sur tout autre type, l'expression `a[i]` est normalement un raccourci pour `*a.index(i)`, où `index` est une méthode du `std::ops::Index` trait. Cependant, si l'expression est affectée ou empruntée de manière mutable, il s'agit plutôt d'un raccourci pour `*a.index_mut(i)`, un appel à la méthode du `std::ops::IndexMut` trait.

Voici les définitions des traits :

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Notez que ces traits prennent le type de l'expression d'index comme paramètre. Vous pouvez indexer une tranche avec un seul `usize`, faisant référence à un seul élément, car les tranches implémentent `Index<usize>`. Mais vous pouvez faire référence à une sous-tranche avec une expression comme `a[i..j]` parce qu'ils implémentent également `Index<Range<usize>>`. Cette expression est un raccourci pour :

```
*a.index(std::ops::Range { start: i, end:j })
```

Les collections Rust `HashMap` et `BTreeMap` vous permettent d'utiliser n'importe quel type hachable ou ordonné comme index. Le code suivant fonctionne car `HashMap<&str, i32>` implémente `Index<&str>`:

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("億", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

Ces expressions d'indexation sont équivalentes à :

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

Le `Index` type associé au trait `Output` spécifie le type produit par une expression d'indexation : pour notre `HashMap`, le type `Index` de l'implémentation `Output` est `i32`.

Le `IndexMut` trait s'étend `Index` avec une `index_mut` méthode qui prend une référence mutable à `self` et renvoie une référence mutable à une `Output` valeur. Rust sélectionne automatiquement `index_mut` lorsque l'expression d'indexation se produit dans un contexte où cela est nécessaire. Par exemple, supposons que nous écrivions ce qui suit :

```
let mut desserts =
    vec![ "Howalon".to_string(), "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

Étant donné que la `push_str` méthode fonctionne sur `&mut self`, ces deux dernières lignes sont équivalentes à :

```
use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");
```

Une limitation de `IndexMut` est que, de par sa conception, il doit renvoyer une référence mutable à une certaine valeur. C'est pourquoi vous ne pouvez pas utiliser une expression comme `m["+"] = 10;` pour insérer une valeur dans le `HashMap m`: la table devrait d' "+" abord créer une entrée, avec une valeur par défaut, et renvoyer une référence mutable à celle-ci. Mais tous les types n'ont pas de valeurs par défaut bon marché, et certains peuvent être coûteux à supprimer ; ce serait un gaspillage de créer une telle valeur pour être immédiatement abandonnée par l'affectation. (Il est prévu d'améliorer cela dans les versions ultérieures du langage.)

L'utilisation la plus courante de l'indexation concerne les collections. Par exemple, supposons que nous travaillions avec des images bitmap, comme celles que nous avons créées dans le traceur d'ensemble de Mandelbrot au [chapitre 2](#). Rappelez-vous que notre programme contenait un code comme celui-ci :

```
pixels[row * bounds.0 + column] = ...;
```

Il serait plus agréable d'avoir un `Image<u8>` type qui agit comme un tableau à deux dimensions, nous permettant d'accéder aux pixels sans avoir à écrire toute l'arithmétique :

```
image[row][column] = ...;
```

Pour ce faire, nous devrons déclarer une structure :

```
struct Image<P> {
    width: usize,
    pixels: Vec<P>,
}

impl<P: Default + Copy> Image<P> {
    /// Create a new image of the given size.
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height],
        }
    }
}
```

Et voici des implémentations de `Index` et `IndexMut` qui feraient l'affaire:

```

impl<P> std::ops:: Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) ->&[P] {
        let start = row * self.width;
        &self.pixels[start..start + self.width]
    }
}

impl<P> std::ops:: IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) ->&mut [P] {
        let start = row * self.width;
        &mut self.pixels[start..start + self.width]
    }
}

```

Lorsque vous indexez dans un `Image`, vous récupérez une tranche de pixels ; l'indexation de la tranche vous donne un pixel individuel.

Notez que lorsque nous écrivons `image[row][column]`, si `row` est hors limites, notre `.index()` méthode essaiera d'indexer `self.pixels` hors limites, déclenchant une panique. C'est ainsi `Index` que `IndexMut` les implémentations sont censées se comporter : un accès hors limites est détecté et provoque une panique, comme lorsque vous indexez un tableau, une tranche ou un vecteur hors limites.

Autres opérateurs

Pas toutes les opérateurs peuvent être surchargés dans Rust. Depuis Rust 1.56, l' `?` opérateur de vérification des erreurs ne fonctionne qu'avec `Result` et quelques autres types de bibliothèques standard, mais des travaux sont en cours pour l'étendre également aux types définis par l'utilisateur. De même, les opérateurs logiques `&&` et `||` sont limités aux valeurs booléennes uniquement. Les opérateurs `..` et `...=` créent toujours une structure représentant les limites de la plage, l' `&` opérateur empêche toujours des références et l' `=` opérateur déplace ou copie toujours des valeurs. Aucun d'entre eux ne peut être surchargé.

L'opérateur de déréférencement `*val`, et l'opérateur point pour accéder aux champs et aux méthodes d'appel, comme dans `val.field` et `val.method()`, peuvent être surchargés à l'aide [des traits](#) `Deref` et `DerefMut`, qui sont traités dans le chapitre suivant. (Nous ne les avons pas inclus ici car ces traits font plus que simplement surcharger quelques opérateurs.)

Rust ne prend pas en charge la surcharge de l'opérateur d'appel de fonction, `f(x)`. Au lieu de cela, lorsque vous avez besoin d'une valeur appelleable, vous écrirez généralement simplement une fermeture. Nous expliquerons comment cela fonctionne et aborderons les traits spéciaux `Fn`, `FnMut` et au [chapitre 14](#). `FnOnce`

- 1** Les programmeurs Lisp se réjouissent ! L'expression `<i32 as Add>::add` est l'
+ opérateur sur `i32`, capturé en tant que valeur de fonction.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 13. Caractéristiques utilitaires

La science n'est rien d'autre que la recherche de l'unité dans la variété sauvage de la nature - ou, plus exactement, dans la variété de notre expérience. La poésie, la peinture, les arts sont la même recherche, selon l'expression de Coleridge, de l'unité dans la variété.

—Jacob Bronowski

Ce chapitre décrit ce que nous appelons les traits « utilitaires » de Rust, un ensemble de divers traits de la bibliothèque standard qui ont suffisamment d'impact sur la façon dont Rust est écrit pour que vous deviez vous familiariser avec eux afin d'écrire du code et de la conception idiomatiques des interfaces publiques pour vos caisses que les utilisateurs jugeront comme étant proprement « rustiques ». Ils se répartissent en trois grandes catégories :

Traits d'extension de langue

Tout comme l'opérateur les traits de surcharge que nous avons abordés dans le chapitre précédent vous permettent d'utiliser les opérateurs d'expression de Rust sur vos propres types, il existe plusieurs autres traits de bibliothèque standard qui servent de points d'extension Rust, vous permettant d'intégrer plus étroitement vos propres types avec le langage. Ceux-ci incluent `Drop`, `Deref` et `DerefMut`, et les traits de conversion `From` et `Into`. Nous allons les décrire dans ce chapitre.

Traits marqueurs

Ce sont des traits principalement utilisé pour lier des variables de type générique pour exprimer des contraintes que vous ne pouvez pas capturer autrement. Ceux-ci incluent `Sized` et `Copy`.

Traits de vocabulaire public

Ceux-ci n'ont rien de magique intégration du compilateur ; vous pouvez définir des traits équivalents dans votre propre code. Mais ils servent l'objectif important d'établir des solutions conventionnelles pour des problèmes communs. Celles-ci sont particulièrement utiles dans les interfaces publiques entre les caisses et les modules : en réduisant les variations inutiles, elles facilitent la compréhension des interfaces, mais elles augmentent également la probabilité que les fonctionnalités de différentes caisses puissent simplement être connectées directement ensemble, sans passe-partout ou code de colle personnalisé. Ceux-ci incluent `Default` les traits d'emprunt de référence `AsRef`, et `AsMut` ; les traits de conversion faillibles et ; et le trait, une

généralisation de

. Borrow BorrowMut TryFrom TryInto ToOwned Clone

Ceux-ci sont résumés dans [le Tableau 13-1](#).

Tableau 13-1. Résumé des traits d'utilité

Caractéristique	La description
<u>Drop</u>	Destructeurs. Code de nettoyage que Rust exécute automatiquement chaque fois qu'une valeur est supprimée.
<u>Sized</u>	Trait de marqueur pour les types avec une taille fixe connue au moment de la compilation, par opposition aux types (tels que les tranches) qui sont dimensionnés dynamiquement.
<u>Clone</u>	Types prenant en charge les valeurs de clonage.
<u>Copy</u>	Trait de marqueur pour les types qui peuvent être clonés simplement en faisant une copie octet par octet de la mémoire contenant la valeur.
<u>Deref et DerefMut</u>	Caractéristiques pour les types de pointeurs intelligents.
<u>Default</u>	Les types qui ont une "valeur par défaut" sensible.
<u>AsRef et AsMut</u>	Traits de conversion pour emprunter un type de référence à un autre.
<u>Borrow et BorrowMut</u>	Traits de conversion, comme <code>AsRef / AsMut</code> , mais garantissant en outre un hachage, un ordre et une égalité cohérents.
<u>From et Into</u>	Traits de conversion pour transformer un type de valeur en un autre.
<u>TryFrom et TryInto</u>	Traits de conversion pour transformer un type de valeur en un autre, pour les transformations qui pourraient échouer.
<u>ToOwned</u>	Trait de conversion pour convertir une référence en une valeur possédée.

Il existe également d'autres caractéristiques de bibliothèque standard importantes. Nous couvrirons `Iterator` et `IntoIterator` au [chapitre 15](#). Le `Hash` trait, pour le calcul des codes de hachage, est traité au [chapitre 16](#). Et une paire de traits qui marquent les types thread-safe, `Send` et `Sync`, sont traités dans le [chapitre 19](#).

Goutte

Lorsque le propriétaire d'une valeur s'en va, on dit que Rust *tombela* valeur. La suppression d'une valeur implique la libération de toutes les autres valeurs, du stockage de tas et des ressources système que la valeur possède. Les baisses se produisent dans diverses circonstances : lorsqu'une variable sort de la portée ; à la fin d'une instruction d'expression ; lorsque vous tronquez un vecteur, supprimez des éléments de sa fin ; etc.

Pour la plupart, Rust gère automatiquement la suppression des valeurs pour vous. Par exemple, supposons que vous définissiez le type suivant :

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

Un `Appellation` possède un tas de stockage pour le contenu des chaînes et le tampon d'éléments du vecteur. Rust s'occupe de nettoyer tout cela chaque fois qu'un `Appellation` est tombé, sans aucun autre codage nécessaire de votre part. Cependant, si vous le souhaitez, vous pouvez personnaliser la manière dont Rust supprime les valeurs de votre type en implémentant le `std::ops::Drop` trait :

```
trait Drop {
    fn drop(&mut self);
}
```

Une implémentation de `Drop` est analogue à un destructeur en C++ ou à un finaliseur dans d'autres langages. Lorsqu'une valeur est supprimée, si elle implémente `std::ops::Drop`, Rust appelle sa `drop` méthode, avant de procéder à la suppression des valeurs propres à ses champs ou éléments, comme il le ferait normalement. Cette invocation implicite de `drop` est le seul moyen d'appeler cette méthode ; si vous essayez de l'invoquer explicitement vous-même, Rust le signale comme une erreur.

Étant donné que Rust appelle `Drop::drop` une valeur avant de supprimer ses champs ou éléments, la valeur que la méthode reçoit est toujours entièrement initialisée. Une implémentation de `Drop` pour notre `Appellation` type peut utiliser pleinement ses champs :

```
impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})".format(self.nicknames.join(", ")));
        }
        println!("");
    }
}
```

Compte tenu de cette implémentation, nous pouvons écrire ce qui suit :

```
{
    let mut a = Appellation {
        name: "Zeus".to_string(),
        nicknames: vec!["cloud collector".to_string(),
                        "king of the gods".to_string()]
    };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames: vec![] };
    println!("at end of block");
}
```

Lorsque nous affectons le second `Appellation` à `a`, le premier est supprimé, et lorsque nous quittons la portée de `a`, le second est supprimé. Ce code imprime ce qui suit :

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

Puisque notre `std::ops::Drop` implémentation de `Appellation` ne fait rien d'autre qu'afficher un message, comment, exactement, sa mémoire est-elle nettoyée ? Le `Vec` type implémente `Drop`, supprimant chacun de ses éléments, puis libérant le tampon alloué par tas qu'ils occupaient. A `String` utilise un `Vec<u8>` interne pour contenir son texte, il n'a donc `String` pas besoin de s'implémenter `Drop` lui-même ; il lui

laisse `vec` le soin de libérer les personnages. Le même principe s'étend aux `Appellation` valeurs : lorsqu'une est supprimée, c'est finalement l'`vec` implémentation de `Drop` qui s'occupe de libérer le contenu de chacune des chaînes, et enfin de libérer le tampon contenant les éléments du vecteur. Quant à la mémoire qui contient le `Appellation` valeur elle-même, elle a aussi un propriétaire, peut-être une variable locale ou une structure de données, qui est responsable de sa libération.

Si la valeur d'une variable est déplacée ailleurs, de sorte que la variable n'est pas initialisée lorsqu'elle sort de la portée, alors Rust n'essaiera pas de supprimer cette variable : il n'y a aucune valeur à supprimer.

Ce principe est valable même lorsqu'une variable peut ou non avoir vu sa valeur s'éloigner, selon le flux de contrôle. Dans de tels cas, Rust garde une trace de l'état de la variable avec un indicateur invisible indiquant si la valeur de la variable doit être supprimée ou non :

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames: vec![ "shotweed".to_string(),
                                            "bittercress".to_string() ] };
    if complicated_condition() {
        p = q;
    }
}
println!("Sproing! What was that?");
```

Selon qu'il `complicated_condition` renvoie `true` ou `false`, soit `p` ou `q` finira par posséder le `Appellation`, avec l'autre non initialisé. L'endroit où il atterrit détermine s'il est déposé avant ou après le `println!`, car il `q` est hors de portée avant le `println!`, et `p` après. Bien qu'une valeur puisse être déplacée d'un endroit à l'autre, Rust ne la supprime qu'une seule fois.

Vous n'aurez généralement pas besoin d'implémenter à `std::ops::Drop` moins que vous ne définissiez un type qui possède des ressources que Rust ne connaît pas déjà. Par exemple, sur les systèmes Unix, la bibliothèque standard de Rust utilise le type suivant en interne pour représenter un descripteur de fichier du système d'exploitation :

```
struct FileDesc {
    fd:c_int,
```

```
}
```

Le `fd` champ de `a FileDesc` est simplement le numéro du descripteur de fichier qui doit être fermé lorsque le programme en a fini avec lui ; `c_int` est un alias pour `i32`. La bibliothèque standard implémente `Drop` pour `FileDesc` comme suit :

```
impl Drop for FileDesc {  
    fn drop(&mut self) {  
        let _ = unsafe { libc::close(self.fd) };  
    }  
}
```

Voici le nom Rust de la fonction `libc::close` de la bibliothèque C. `close` Le code Rust peut appeler des fonctions C uniquement dans `unsafe` des blocs, donc la bibliothèque en utilise une ici.

Si un type implémente `Drop`, il ne peut pas implémenter le `Copy` trait. Si un type est `Copy`, cela signifie qu'une simple duplication octet par octet est suffisante pour produire une copie indépendante de la valeur. Mais c'est généralement une erreur d'appeler la même `drop` méthode plus d'une fois sur les mêmes données.

Le prélude standard inclut une fonction pour supprimer une valeur `drop`, mais sa définition est tout sauf magique :

```
fn drop<T>(_x:T) { }
```

En d'autres termes, il reçoit son argument par valeur, prenant possession de l'appelant, puis ne fait rien avec. Rust supprime la valeur `_x` lorsqu'il sort de la portée, comme il le ferait pour toute autre variable.

Taille

Une *tailletype* est celui dont les valeurs ont toutes la même taille en mémoire. Presque tous les types de Rust sont dimensionnés : chacun `u64` prend huit octets, chaque `(f32, f32, f32)` tuple douze. Même les énumérations sont dimensionnées : quelle que soit la variante réellement présente, une énumération occupe toujours suffisamment d'espace pour contenir sa plus grande variante. Et bien que `a Vec<T>` possède un tampon alloué par tas dont la taille peut varier, la `Vec` valeur elle-même est

un pointeur vers le tampon, sa capacité et sa longueur, tout `Vec<T>` comme un type dimensionné.

Tous les types dimensionnés implémentent le `std::marker::Sized` trait, qui n'a pas de méthodes ou de types associés. Rust l'implémente automatiquement pour tous les types auxquels il s'applique ; vous ne pouvez pas l'implémenter vous-même. La seule utilisation de `for sized` est en tant que borne pour les variables de type : une borne like `T: Sized` nécessite `T` d'être un type dont la taille est connue au moment de la compilation. Les traits de ce type sont appelés *traits marqueurs*, car le langage Rust lui-même les utilise pour marquer certains types comme ayant des caractéristiques intéressantes.

Cependant, Rust a également *quelques types* dont les valeurs ne sont pas toutes de la même taille. Par exemple, le type tranche de chaîne `str` (note, sans `&`) n'est pas dimensionné. Les littéraux de chaîne "diminutive" et "big" sont des références à des `str` tranches qui occupent dix et trois octets. Les deux sont illustrés à la [Figure 13-1](#). Les types de tranches de tableau comme `[T]` (encore une fois, sans `&`) ne sont pas non plus dimensionnés : une référence partagée comme `&[u8]` peut pointer vers une `[u8]` tranche de n'importe quelle taille. Étant donné que les types `str` et `[T]` désignent des ensembles de valeurs de tailles variables, ce sont des types non dimensionnés.

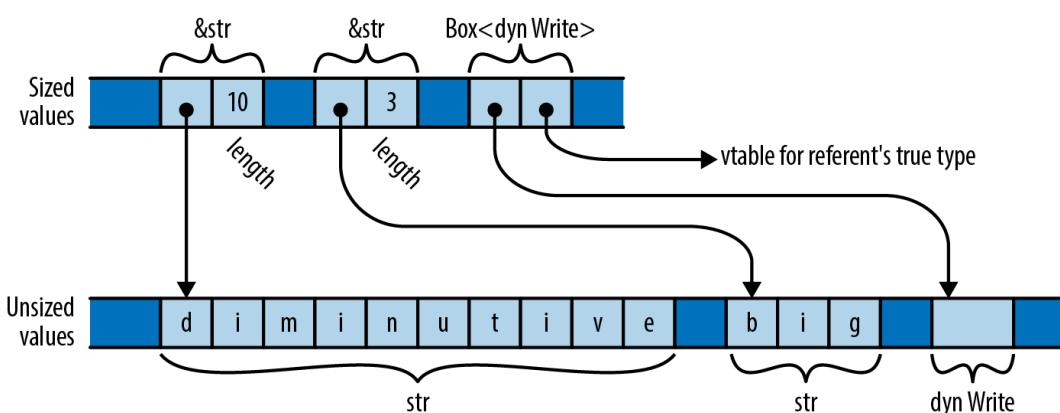


Illustration 13-1. Références à des valeurs non dimensionnées

L'autre type courant de type non dimensionné dans Rust est un `dyn` type, le référent d'un objet trait. Comme nous l'avons expliqué dans "[Trait Objects](#)", un objet trait est un pointeur vers une valeur qui implémente un trait donné. Par exemple, les types `&dyn std::io::Write` et `Box<dyn std::io::Write>` sont des pointeurs vers une valeur qui implémente le `Write` trait. Le référent peut être un fichier ou un socket réseau ou un type de votre choix pour lequel vous avez implémenté `Write`. Puisque l'ensemble des types qui implémentent `Write` est ouvert, `dyn`

`write` considéré comme un type n'est pas dimensionné : ses valeurs ont des tailles différentes.

Rust ne peut pas stocker de valeurs non dimensionnées dans des variables ni les transmettre en tant qu'arguments. Vous ne pouvez les traiter qu'à l'aide de pointeurs tels que `&str` ou `Box<dyn Write>`, eux-mêmes dimensionnés. Comme le montre la [figure 13-1](#), un pointeur vers une valeur non dimensionnée est toujours un *pointeur gras*, large de deux mots : un pointeur vers une trame porte également la longueur de la trame, et un objet trait porte également un pointeur vers une vtable d'implémentations de méthodes.

Les objets de trait et les pointeurs vers les tranches sont bien symétriques. Dans les deux cas, le type manque des informations nécessaires pour l'utiliser : vous ne pouvez pas indexer à `[u8]` sans connaître sa longueur, ni invoquer une méthode sur `a Box<dyn Write>` sans connaître l'implémentation de `write` appropriée à la valeur spécifique à laquelle il se réfère. Et dans les deux cas, le pointeur gras complète les informations manquantes du type, portant une longueur ou un pointeur vtable. Les informations statiques omises sont remplacées par des informations dynamiques .

Étant donné que les types non dimensionnés sont si limités, la plupart des variables de type génériques doivent être limitées aux `Sized` types. En fait, cela est si souvent nécessaire que c'est le défaut implicite de Rust : si vous écrivez `struct S<T> { ... }`, Rust comprend que vous voulez dire `struct S<T: Sized> { ... }`. Si vous ne souhaitez pas restreindre `T` cette méthode, vous devez explicitement vous désinscrire en écrivant `struct S<T: ?Sized> { ... }`. La `?Sized` syntaxe est spécifique à ce cas et signifie "pas nécessairement `Sized`". Par exemple, si vous écrivez `struct S<T: ?Sized> { b: Box<T> }`, alors Rust vous permettra d'écrire `S<str>` et `S<dyn Write>`, où la boîte devient un gros pointeur, ainsi que `S<i32>` et `S<String>`, où la boîte est un pointeur ordinaire.

Malgré leurs restrictions, les types non dimensionnés rendent le système de type de Rust plus fluide. En lisant la documentation de la bibliothèque standard, vous rencontrerez occasionnellement une `?Sized` borne sur une variable de type ; cela signifie presque toujours que le type donné est uniquement pointé et permet au code associé de fonctionner avec des tranches et des objets de trait ainsi qu'avec des valeurs ordinaires. Lors-

qu'une variable de type a la `?Sized` limite, les gens disent souvent qu'elle est *de taille douteuse* : elle peut être `Sized`, ou non.

Outre les tranches et les objets de trait, il existe un autre type de type non dimensionné. Le dernier champ d'un type de struct (mais seulement son dernier) peut être non dimensionné, et un tel struct est lui-même non dimensionné. Par exemple, un `Rc<T>` pointeur de comptage de références est implémenté en interne en tant que pointeur vers le type privé `RcBox<T>`, qui stocke le nombre de références à côté de `T`. Voici une définition simplifiée de `RcBox` :

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value:T,
}
```

Le `value` champ est le `T` vers lequel `Rc<T>` compte les références ; `Rc<T>` déréférence à un pointeur vers ce champ. Le `ref_count` champ contient le nombre de références.

Le real `RcBox` n'est qu'un détail d'implémentation de la bibliothèque standard et n'est pas disponible pour un usage public. Mais supposons que nous travaillions avec la définition précédente. Vous pouvez l'utiliser `RcBox` avec des types dimensionnés, comme `RcBox<String>` ; le résultat est un type de structure dimensionné. Ou vous pouvez l'utiliser avec des types non dimensionnés, comme `RcBox<dyn std::fmt::Display>` (où `Display` est le trait pour les types qui peuvent être formatés par `println!` et des macros similaires) ; `RcBox<dyn Display>` est un type de structure non dimensionné.

Vous ne pouvez pas créer une `RcBox<dyn Display>` valeur directement. Au lieu de cela, vous devez d'abord créer un format ordinaire `RcBox` dont le `value` type implémente `Display`, comme `RcBox<String>`. Rust permet alors de convertir une référence `&RcBox<String>` en référence fat `&RcBox<dyn Display>` :

```
let boxed_lunch: RcBox<String> = RcBox {
    ref_count: 1,
    value:"lunch".to_string()
};

use std::fmt::Display;
let boxed_displayable:&RcBox<dyn Display> = &boxed_lunch;
```

Cette conversion se produit implicitement lors du passage de valeurs à des fonctions, vous pouvez donc passer un `&RcBox<String>` à une fonction qui attend un `&RcBox<dyn Display>`:

```
fn display(boxed:&RcBox<dyn Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

Cela produirait la sortie suivante:

```
For your enjoyment: lunch
```

Cloner

Le `std::clone::Clone` traite pour les types qui peuvent faire des copies d'eux-mêmes. `Clone` est défini comme suit :

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source:&Self) {
        *self = source.clone()
    }
}
```

La `clone` méthode doit construire une copie indépendante de `self` et la renvoyer. Étant donné que le type de retour de cette méthode est `Self` et que les fonctions ne peuvent pas renvoyer de valeurs non dimensionnées, le `Clone` trait lui-même étend le `Sized` trait : cela a pour effet de limiter les `Self` types d'implémentations à `sized`.

Le clonage d'une valeur implique généralement d'allouer des copies de tout ce qu'elle possède, de sorte qu'une `clone` peut être coûteuse, en temps et en mémoire. Par exemple, le clonage de un `Vec<String>` non seulement copie le vecteur, mais copie également chacun de ses `String` éléments. C'est pourquoi Rust ne se contente pas de cloner automatiquement les valeurs, mais vous oblige à faire un appel de méthode explicite. Les types de pointeurs à comptage de références comme `Rc<T>` et `Arc<T>` sont des exceptions : le clonage de l'un d'entre eux in-

crémentement le nombre de références et vous donne un nouveau pointeur.

La `clone_from` méthode `self` se transforme en une copie de `source`. La définition par défaut de `clone_from` simplement cloner `source`, puis la déplace dans `*self`. Cela fonctionne toujours, mais pour certains types, il existe un moyen plus rapide d'obtenir le même effet. Par exemple, supposons que `s` et `t` sont `String`s. L'instruction `s = t.clone();` doit cloner `t`, supprimer l'ancienne valeur de `s`, puis déplacer la valeur clonée dans `s`; c'est une allocation de tas et une désallocation de tas. Mais si le tampon de tas appartenant à l'original `s` a une capacité suffisante pour contenir `t` le contenu de `t`, aucune allocation ou désallocation n'est nécessaire : vous pouvez simplement copier `t` le texte `s` de dans le tampon de `t` et ajuster la longueur. En code générique, vous devez utiliser `clone_from` dans la mesure du possible pour tirer parti des implémentations optimisées lorsqu'elles sont présentes.

Si votre `Clone` implémentation s'applique simplement `clone` à chaque champ ou élément de votre type, puis construit une nouvelle valeur à partir de ces clones, et que la définition par défaut de `clone_from` est suffisamment bonne, alors Rust l'implémentera pour vous : placez-le simplement `#[derive(Clone)]` au-dessus de votre définition de type.

À peu près tous les types de la bibliothèque standard qui ont du sens pour copier les outils `Clone`. Les types primitifs aiment `bool` et `i32` font. Les types de conteneurs tels que `String`, `Vec<T>` et `HashMap` do également. Certains types n'ont pas de sens à copier, comme `std::sync::Mutex`; ceux-ci ne sont pas implémentés `Clone`. Certains types comme `std::fs::File` peuvent être copiés, mais la copie peut échouer si le système d'exploitation ne dispose pas des ressources nécessaires ; ces types n'implémentent pas `Clone`, car ils `clone` doivent être infaillibles. Au lieu de cela, `std::fs::File` fournit une `try_clone` méthode qui renvoie un `std::io::Result<File>`, qui peut signaler un échec.

Copie

Au [chapitre 4](#), nous avons expliqué que, pour la plupart des types, l'affection déplace les valeurs au lieu de les copier. Le déplacement des valeurs facilite grandement le suivi des ressources qu'ils possèdent. Mais dans ["Copy Types : The Exception to Moves"](#), nous avons souligné l'exception : les types simples qui ne possèdent aucune ressource peuvent être

des `Copy` types, où l'affectation fait une copie de la source, plutôt que de déplacer la valeur et de laisser la source non initialisée .

À ce moment-là, nous avons laissé dans le vague exactement ce qui `Copy` était, mais maintenant nous pouvons vous dire : un type est `Copy` s'il implémente le `std::marker::Copy` trait marqueur, qui est défini comme suit :

```
trait Copy:Clone { }
```

Ceci est certainement facile à mettre en œuvre pour vos propres types :

```
impl Copy for MyType { }
```

Mais comme il `Copy` s'agit d'un trait de marqueur ayant une signification particulière pour le langage, Rust permet à un type de ne s'implémenter `Copy` que si une copie superficielle octet par octet est tout ce dont il a besoin. Les types qui possèdent d'autres ressources, comme les tampons de tas ou les handles du système d'exploitation, ne peuvent pas implémenter `Copy` .

Tout type qui implémente le `Drop` trait ne peut pas être `Copy` . Rust suppose que si un type nécessite un code de nettoyage spécial, il doit également nécessiter un code de copie spécial et ne peut donc pas être `Copy` .

Comme pour `Clone` , vous pouvez demander à Rust de dériver `Copy` pour vous, en utilisant `#[derive(Copy)]` . Vous verrez souvent les deux dérivés à la fois, avec `#[derive(Copy, Clone)]` .

Réfléchissez bien avant de faire un type `Copy` . Bien que cela facilite l'utilisation du type, cela impose de lourdes restrictions à sa mise en œuvre. Les copies implicites peuvent également être coûteuses. Nous expliquons ces facteurs en détail dans "[Types de copie : l'exception aux déplacements](#)".

Deref et DerefMut

Vous pouvez spécifier comment le déréférencement les opérateurs aiment `*` et `.` se comportent sur vos types en implémentant les traits `std::ops::Deref` et `. std::ops::DerefMut` Les types de pointeurs aiment `Box<T>` et `Rc<T>` implémentent ces traits afin qu'ils puissent se

comporter comme le font les types de pointeurs intégrés de Rust. Par exemple, si vous avez une `Box<Complex>` valeur `b`, alors `*b` fait référence à la `Complex` valeur qui `b` pointe vers et `b.re` fait référence à son composant réel. Si le contexte attribue ou emprunte une référence mutable au référent, Rust utilise le `DerefMut` trait (« déréférencer de manière mutable ») ; sinon, l'accès en lecture seule est suffisant et il utilise `Deref`.

Les traits sont définis comme ceci :

```
trait Deref {  
    type Target: ?Sized;  
    fn deref(&self) -> &Self::Target;  
}  
  
trait DerefMut: Deref {  
    fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

Les méthodes `deref` et prennent une référence et renvoient une référence. devrait être quelque chose qui contient, possède ou se réfère à : car le type est . Notez que cela s'étend : si vous pouvez déréférencer quelque chose et le modifier, vous devriez certainement pouvoir également lui emprunter une référence partagée. Étant donné que les méthodes renvoient une référence avec la même durée de vie que , reste emprunté aussi longtemps que la référence renvoyée

```
vit. deref_mut &Self &Self::Target Target Self Box<Complex> T  
arget Complex DerefMut Deref &self self
```

Les traits `Deref` et `DerefMut` jouent également un autre rôle. Puisque `deref` prend une `&Self` référence et renvoie une `&Self::Target` référence, Rust l'utilise pour convertir automatiquement les références du premier type dans le second. En d'autres termes, si l'insertion d'un `deref` appel permet d'éviter une incompatibilité de type, Rust en insère un pour vous. L'implémentation `DerefMut` permet la conversion correspondante pour les références mutables. Celles-ci sont appelées les *coercions de deref*: un type est « constraint » à se comporter comme un autre.

Bien que les contraintes de `deref` ne soient pas quelque chose que vous ne pourriez pas écrire explicitement vous-même, elles sont pratiques :

- Si vous avez une certaine `Rc<String>` valeur `r` et que vous souhaitez l'appliquer `String::find`, vous pouvez simplement écrire

`r.find('?')`, au lieu de `(*r).find('?')` : l'appel de méthode emprunte implicitement `r` et `&Rc<String>` contraint à `&String`, car `Rc<T>` implémente `Deref<Target=T>`.

- Vous pouvez utiliser des méthodes comme `split_at` on `String` values, même s'il `split_at`s'agit d'une méthode de `str` type slice, car `String` implements `Deref<Target=str>`. Il n'est pas nécessaire `String` de réimplémenter toutes `str` les méthodes de , puisque vous pouvez contraindre a à `&str` partir de `&String`.
- Si vous avez un vecteur d'octets `v` et que vous voulez le passer à une fonction qui attend une tranche d'octet `&[u8]`, vous pouvez simplement le passer `&v` comme argument, puisque `Vec<T>` implements `Deref<Target=[T]>`.

Rust appliquera successivement plusieurs coercitions de deref si nécessaire. Par exemple, en utilisant les coercions mentionnées précédemment, vous pouvez appliquer `split_at` directement à un `Rc<String>`, puisque `&Rc<String>` déréférence à `&String`, qui déréférence à `&str`, qui a la `split_at` méthode.

Par exemple, supposons que vous ayez le type suivant :

```
struct Selector<T> {
    /// Elements available in this `Selector`.
    elements: Vec<T>,
    /// The index of the "current" element in `elements`. A `Selector`
    /// behaves like a pointer to the current element.
    current: usize
}
```

Pour faire en sorte que le `Selector` comportement soit conforme au commentaire de la documentation, vous devez implémenter `Deref` et `DerefMut` pour le type :

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) ->&T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
```

```

fn deref_mut(&mut self) ->&mut T {
    &mut self.elements[self.current]
}

```

Compte tenu de ces implémentations, vous pouvez utiliser un Selector comme ceci:

```

let mut s = Selector { elements: vec!['x', 'y', 'z'],
                      current:2 };

// Because `Selector` implements `Deref`, we can use the `*` operator to
// refer to its current element.
assert_eq!(*s, 'z');

// Assert that 'z' is alphabetic, using a method of `char` directly on a
// `Selector`, via deref coercion.
assert!(s.is_alphabetic());

// Change the 'z' to a 'w', by assigning to the `Selector`'s referent.
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);

```

Les traits `Deref` et `DerefMut` sont conçus pour implémenter des types de pointeurs intelligents, tels que `Box`, `Rc`, et `Arc`, et des types qui servent de versions propriétaires de quelque chose que vous utiliseriez aussi fréquemment par référence, la manière `Vec<T>` et `String` servant de versions propriétaires de `[T]` et `str`. Vous ne devez pas implémenter `Deref` et `DerefMut` pour un type juste pour faire `Target` apparaître automatiquement les méthodes du type, de la même manière que les méthodes d'une classe de base C++ sont visibles sur une sous-classe. Cela ne fonctionnera pas toujours comme prévu et peut être source de confusion lorsque cela tourne mal.

Les coercitions `deref` s'accompagnent d'une mise en garde qui peut prêter à confusion : Rust les applique pour résoudre les conflits de type, mais pas pour satisfaire les limites sur les variables de type. Par exemple, le code suivant fonctionne correctement :

```

let s = Selector { elements: vec!["good", "bad", "ugly"],
                   current:2 };

```

```
fn show_it(thing:&str) { println!("{}", thing); }
show_it(&s);
```

Dans l'appel `show_it(&s)`, Rust voit un argument de type `&Selector<&str>` et un paramètre de type `&str`, trouve l'`Deref<Target=str>` implémentation et réécrit l'appel en tant que `show_it(s.deref())`, juste au besoin.

Cependant, si vous changez `show_it` en une fonction générique, Rust n'est soudainement plus coopératif :

```
use std::fmt::Display;
fn show_it_generic<T: Display>(thing:T) { println!("{}", thing); }
show_it_generic(&s);
```

La rouille se plaint :

```
error: `Selector<&str>` doesn't implement `std::fmt::Display`
|
31 |     show_it_generic(&s);
      ^ ^
      |
      |
      |         `Selector<&str>` cannot be formatted with
      |             the default formatter
      |             help: consider adding dereference here: `&*s`
      |
      |
      note: required by a bound in `show_it_generic`
|
30 | fn show_it_generic<T: Display>(thing: T) { println!("{}", thing);
      |             ^^^^^^ required by this bound
      |                 in `show_it_generic`
```

Cela peut être déconcertant : comment le fait de rendre une fonction générique pourrait-il introduire une erreur ? Certes, `Selector<&str>` ne s'implémente pas `Display`, mais il déréférence à `&str`, ce qui est certainement le cas.

Puisque vous passez un argument de type `&Selector<&str>` et que le type de paramètre de la fonction est `&T`, la variable de type `T` doit être `Selector<&str>`. Ensuite, Rust vérifie si la borne `T: Display` est satisfait : puisqu'il n'applique pas de coercitions de deref pour satisfaire les bornes sur les variables de type, cette vérification échoue.

Pour contourner ce problème, vous pouvez épeler la coercition à l'aide de l' `as` opérateur:

```
show_it_generic(&s as &str);
```

Ou, comme le suggère le compilateur, vous pouvez forcer la coercition avec `&*`:

```
show_it_generic(&*s);
```

Défaut

Certains types ont un défaut raisonnablement évident : le vecteur ou la chaîne par défaut est vide, le nombre par défaut est zéro, la valeur par défaut `Option` est `None`, etc. Des types comme celui-ci peuvent implémenter le `std::default::Default` trait :

```
trait Default {
    fn default() -> Self;
}
```

La `default` méthode renvoie simplement une nouvelle valeur de type `Self`. L'implémentation de `Default` est simple :

```
impl Default for String {
    fn default() -> String {
        String::new()
    }
}
```

Tous les types de collection de Rust — `Vec`, `HashMap`, `BinaryHeap`, etc. — implémentent `Default`, avec des `default` méthodes qui renvoient une collection vide. Ceci est utile lorsque vous devez créer une collection de valeurs, mais que vous souhaitez laisser votre appelant décider exactement du type de collection à créer. Par exemple, la méthode `Iterator` du trait `partition` divise les valeurs produites par l'itérateur en deux collections, en utilisant une fermeture pour décider où va chaque valeur :

```

use std::collections:: HashSet;
let squares = [4, 9, 16, 25, 36, 49, 64];
let (powers_of_two, impure):(HashSet<i32>, HashSet<i32>)
    = squares.iter().partition(|&n| n & (n-1) == 0);

assert_eq!(powers_of_two.len(), 3);
assert_eq!(impure.len(), 4);

```

La fermeture `|&n| n & (n-1) == 0` utilise un peu de bricolage pour reconnaître les nombres qui sont des puissances de deux, et `partition` l'utilise pour produire deux `HashSet`s. Mais bien sûr, `partition` n'est pas spécifique à `HashSet`s; vous pouvez l'utiliser pour produire n'importe quel type de collection, tant que le type de collection implémente `Default`, pour produire une collection vide pour commencer, et `Extend<T>`, pour ajouter un `T` à la collection. `String` implémente `Default` et `Extend<char>`, vous pouvez donc écrire :

```

let (upper, lower):(String, String)
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase());
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eacher nizuka");

```

Une autre utilisation courante de `Default` consiste à produire des valeurs par défaut pour les structures qui représentent une grande collection de paramètres, dont la plupart n'auront généralement pas besoin d'être modifiés. Par exemple, la `glum` caisse fournit des liaisons Rust pour la puissante et complexe bibliothèque graphique OpenGL. La `glum::DrawParameters` structure comprend 24 champs, chacun contrôlant un détail différent de la façon dont OpenGL doit rendre certains éléments graphiques. La `glum draw` fonction attend une `DrawParameters` structure comme argument. Depuis `DrawParameters` implements `Default`, vous pouvez en créer un à transmettre à `draw`, en mentionnant uniquement les champs que vous souhaitez modifier :

```

let params = glum:: DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();

```

Cela appelle `Default::default()` à créer une `DrawParameters` valeur initialisée avec les valeurs par défaut pour tous ses champs, puis utilise la `..` syntaxe des structures pour en créer une nouvelle avec les champs `line_width` et `point_size` modifiés, prêt à être transmis à `target.draw`.

Si un type `T` implémente `Default`, alors la bibliothèque standard implémente `Default` automatiquement pour `Rc<T>`, `Arc<T>`, `Box<T>`, `Cell<T>`, `RefCell<T>`, `Cow<T>`, `Mutex<T>`, et `RwLock<T>`. La valeur par défaut du type `Rc<T>`, par exemple, est un `Rc` pointant vers la valeur par défaut du type `T`.

Si tous les types d'éléments d'un type de tuple implémentent `Default`, alors le type de tuple le fait aussi, par défaut un tuple contenant la valeur par défaut de chaque élément.

Rust n'implémente pas implicitement `Default` pour les types de structure, mais si tous les champs d'une structure implémentent `Default`, vous pouvez implémenter `Default` pour la structure automatiquement en utilisant `#[derive(Default)]`.

AsRef et AsMut

Lorsqu'un genre met en œuvre `AsRef<T>`, cela signifie que vous pouvez lui emprunter un `&T` efficacement. `AsMut` est l'analogue pour les références mutables. Leurs définitions sont les suivantes :

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) ->&T;
}

trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) ->&mut T;
}
```

Ainsi, par exemple, `Vec<T>` implements `AsRef<[T]>`, et `String` implements `AsRef<str>`. Vous pouvez également emprunter `String` le contenu d'un sous forme de tableau d'octets, donc `String` implemente `AsRef<[u8]>` également.

`AsRef` est généralement utilisé pour rendre les fonctions plus flexibles dans les types d'arguments qu'elles acceptent. Par exemple, la

`std::fs::File::open` fonction est déclarée comme ceci :

```
fn open<P: AsRef<Path>>(path: P) ->Result<File>
```

Ce qu'il faut `open` vraiment, c'est un `&Path`, le type représentant un chemin de système de fichiers. Mais avec cette signature, `open` accepte tout ce à quoi il peut emprunter `&Path`, c'est-à-dire tout ce qui implémente `AsRef<Path>`. Ces types incluent `String` et `str`, les types de chaîne d'interface du système d'exploitation `OsString` et `OsStr`, et bien sûr `PathBuf` et `Path`; voir la documentation de la bibliothèque pour la liste complète. C'est ce qui vous permet de passer des littéraux de chaîne à `open`:

```
let dot_emacs = std:: fs:: File::open("/home/jimb/.emacs")?;
```

Toutes les fonctions d'accès au système de fichiers de la bibliothèque standard acceptent les arguments de chemin de cette façon. Pour les appels, l'effet ressemble à celui d'une fonction surchargée en C++, bien que Rust adopte une approche différente pour déterminer quels types d'arguments sont acceptables.

Mais cela ne peut pas être toute l'histoire. Un littéral de chaîne est un `&str`, mais le type qui implémente `AsRef<Path>` est `str`, sans un `&`. Et comme nous l'avons expliqué dans "[Deref et DerefMut](#)", Rust n'essaie pas de coercitions de deref pour satisfaire les limites de variables de type, donc elles ne seront pas utiles ici non plus.

Heureusement, la bibliothèque standard inclut l'implémentation de couverture :

```
impl<'a, T, U> AsRef<U> for &'a T
    where T: AsRef<U>,
          T: ?Sized, U: ?Sized
{
    fn as_ref(&self) ->&U {
        (*self).as_ref()
    }
}
```

En d'autres termes, pour tous les types `T` et `U`, si `T: AsRef<U>`, alors `&T: AsRef<U>` également : suivez simplement la référence et procédez comme avant. En particulier, depuis `str: AsRef<Path>`, alors `&str:`

`AsRef<Path>` aussi. Dans un sens, c'est un moyen d'obtenir une forme limitée de coercition de deref lors de la vérification des `AsRef` bornes sur les variables de type.

Vous pouvez supposer que si un type implémente `AsRef<T>`, il doit également implémenter `AsMut<T>`. Cependant, il y a des cas où cela n'est pas approprié. Par exemple, nous avons mentionné que `String` implemente `AsRef<[u8]>`; cela a du sens, car chacun `String` a certainement un tampon d'octets qui peut être utile pour accéder en tant que données binaires. Cependant, `String` garantit en outre que ces octets sont un codage UTF-8 bien formé du texte Unicode ; s'il était `String` implémenté `AsMut<[u8]>`, cela permettrait aux appelants de changer les `String` octets de pour tout ce qu'ils voulaient, et vous ne pourriez plus faire confiance à un `String` pour être UTF-8 bien formé. L'implémentation d'un type n'a de sens que `AsMut<T>` si la modification du donné `T` ne peut pas violer les invariants du type.

Bien que `AsRef` et `AsMut` soient assez simples, fournir des traits génériques standard pour la conversion de référence évite la prolifération de traits de conversion plus spécifiques. Vous devriez éviter de définir vos propres `AsFoo` traits alors que vous pourriez simplement implémenter `AsRef<Foo>`.

Emprunter et emprunterMut

Le `std::borrow::Borrow` traite est similaire à `AsRef` : si un type implémente `Borrow<T>`, alors sa `borrow` méthode lui emprunte efficacement `&T` à. Mais `Borrow` impose plus de restrictions : un type ne doit être implémenté `Borrow<T>` que lorsqu'un `&T` hachage et se compare de la même manière que la valeur à laquelle il est emprunté. (Rust ne l'applique pas ; c'est juste l'intention documentée du trait.) Cela rend `Borrow` utile le traitement des clés dans les tables de hachage et les arbres ou lors du traitement des valeurs qui seront hachées ou comparées pour une autre raison.

Cette distinction est importante lorsque vous empruntez à `String`s, par exemple : `String` implements `AsRef<str>`, `AsRef<[u8]>` et `AsRef<Path>`, mais ces trois types de cibles auront généralement des valeurs de hachage différentes. Seule la `&str` tranche est garantie de hacher comme l'équivalent `String`, donc `String` implémente uniquement `Borrow<str>`.

Borrow la définition de est identique à celle de AsRef ; seuls les noms ont été modifiés :

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) ->&Borrowed;
}
```

Borrow est conçu pour traiter une situation spécifique avec des tables de hachage génériques et d'autres types de collections associatives. Par exemple, supposons que vous ayez un std::collections ::HashMap<String, i32> mappage de chaînes à des nombres. Les clés de cette table sont String s ; chaque entrée en possède un. Quelle devrait être la signature de la méthode qui recherche une entrée dans cette table ? Voici une première tentative :

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) ->Option<&V> { ... }
}
```

Cela a du sens : pour rechercher une entrée, vous devez fournir une clé du type approprié pour la table. Mais dans ce cas, K est String ; cette signature vous obligerait à passer a String par valeur à chaque appel à get , ce qui est clairement un gaspillage. Vous avez vraiment juste besoin d'une référence à la clé:

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) ->Option<&V> { ... }
}
```

C'est un peu mieux, mais maintenant vous devez passer la clé en tant que &String , donc si vous voulez rechercher une chaîne constante, vous devriez écrire :

```
hashtable.get(&"twenty-two".to_string())
```

C'est ridicule : il alloue un String tampon sur le tas et y copie le texte, juste pour pouvoir l'emprunter en tant que &String , le passer à get , puis le déposer.

Il devrait être suffisant pour transmettre tout ce qui peut être haché et comparé avec notre type de clé ; `a &str` devrait être parfaitement adéquat, par exemple. Voici donc la dernière itération, qui correspond à ce que vous trouverez dans la bibliothèque standard :

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

En d'autres termes, si vous pouvez emprunter la clé d'une entrée en tant que `&Q` et que la référence résultante hache et compare exactement comme le ferait la clé elle-même, alors il `&Q` devrait clairement s'agir d'un type de clé acceptable. Depuis `String` implémente `Borrow<str>` et `Borrow<String>`, cette version finale de `get` vous permet de passer soit `&String` ou `&str` comme clé, selon vos besoins.

`Vec<T>` et mettre en `[T: N]` œuvre `Borrow<[T]>`. Chaque type de type chaîne permet d'emprunter son type de tranche correspondant :

`String` implements `Borrow<str>`, `PathBuf` implements `Borrow<Path>`, etc. Et tous les types de collection associatifs de la bibliothèque standard utilisent `Borrow` pour décider quels types peuvent être passés à leurs fonctions de recherche.

La bibliothèque standard inclut une implémentation globale afin que chaque type `T` puisse être emprunté à lui-même : `T: Borrow<T>`. Cela garantit qu'il `&K` s'agit toujours d'un type acceptable pour rechercher des entrées dans un fichier `HashMap<K, V>`.

Par commodité, chaque `&mut T` type implémente également `Borrow<T>`, renvoyant une référence partagée `&T` comme d'habitude. Cela vous permet de passer des références mutables aux fonctions de recherche de collection sans avoir à réemprunter une référence partagée, émulant la coercition implicite habituelle de Rust des références mutables aux références partagées.

Le `BorrowMut` trait est l'analogue de `Borrow` pour les références mutables :

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) ->&mut Borrowed;
}
```

Les mêmes attentes décrites pour `Borrow` s'appliquent `BorrowMut` également à.

De et vers

Les `std::convert::From` et `std::convert::Into` traits représentent les conversions qui consomment une valeur d'un type et renvoient une valeur d'un autre. Alors que les traits `AsRef` et empruntent une référence d'un type à un autre et s'approprient leur argument, le transforment, puis rendent la propriété du résultat à l'appelant. `AsMut` `From` `Into`

Leurs définitions sont bien symétriques :

```
trait Into<T>: Sized {
    fn into(self) ->T;
}

trait From<T>: Sized {
    fn from(other: T) ->Self;
}
```

La bibliothèque standard implémente automatiquement la conversion triviale de chaque type vers lui-même : chaque type `T` implémente `From<T>` and `Into<T>`.

Bien que les traits fournissent simplement deux façons de faire la même chose, ils se prêtent à des utilisations différentes.

Vous utilisez généralement `Into` pour rendre vos fonctions plus flexibles dans les arguments qu'elles acceptent. Par exemple, si vous écrivez :

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

then `ping` peut accepter non seulement `Ipv4Addr` comme argument, mais aussi un `u32` ou un `[u8; 4]` tableau, puisque ces types implémentent commodément `Into<Ipv4Addr>`. (Il est parfois utile de traiter une adresse IPv4 comme une seule valeur 32 bits ou un tableau de 4 octets.) Comme la seule chose `ping` connaît `address` est qu'elle implémente `Into<Ipv4Addr>`, il n'est pas nécessaire de spécifier le type que vous voulez lorsque vous appelez `into`; il n'y en a qu'un qui pourrait fonctionner, alors l'inférence de type le remplit pour vous.

Comme `AsRef` dans la section précédente, l'effet ressemble beaucoup à celui de la surcharge d'une fonction en C++. Avec la définition de `ping from before`, nous pouvons faire n'importe lequel de ces appels :

```
println!("{}:", ping(Ipv4Addr::new(23, 21, 68, 141))); // pass an Ipv4Addr
println!("{}:", ping([66, 146, 219, 98]));           // pass a [u8; 4]
println!("{}:", ping(0xd076eb94_u32));              // pass a u32
```

Le `From` trait, cependant, joue un rôle différent. La `from` méthode sert de constructeur générique pour produire une instance d'un type à partir d'une autre valeur unique. Par exemple, plutôt que d'`Ipv4Addr` avoir deux méthodes nommées `from_array` et `from_u32`, il implémente simplement `From<[u8; 4]>` et `From<u32>`, ce qui nous permet d'écrire :

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

Nous pouvons laisser l'inférence de type déterminer quelle implémentation s'applique.

Etant donné une `From` implémentation appropriée, la bibliothèque standard implémente automatiquement le `Into` trait correspondant. Lorsque vous définissez votre propre type, s'il a des constructeurs à argument unique, vous devez les écrire en tant qu'implémentations de `From<T>` pour les types appropriés ; vous obtiendrez `Into` gratuitement les implémentations correspondantes.

Étant donné que les méthodes de conversion `from` et `into` s'approprient leurs arguments, une conversion peut réutiliser les ressources de la valeur d'origine pour construire la valeur convertie. Par exemple, supposons que vous écriviez :

```
let text = "Beautiful Soup".to_string();
let bytes:Vec<u8> = text.into();
```

L'implémentation de `Into<Vec<u8>>` for `String` prend simplement le `String` tampon de tas de et le réutilise, tel quel, comme tampon d'élément du vecteur renvoyé. La conversion n'a pas besoin d'allouer ou de copier le texte. C'est un autre cas où les déplacements permettent des implémentations efficaces.

Ces conversions fournissent également un bon moyen d'assouplir une valeur d'un type contraint en quelque chose de plus flexible, sans affaiblir les garanties du type constraint. Par exemple, a `String` garantit que son contenu est toujours valide en UTF-8 ; ses méthodes de mutation sont soigneusement restreintes pour s'assurer que rien de ce que vous pouvez faire n'introduira jamais un mauvais UTF-8. Mais cet exemple "rétrograde" efficacement a `String` en un bloc d'octets simples avec lequel vous pouvez faire tout ce que vous voulez : vous allez peut-être le compresser ou le combiner avec d'autres données binaires qui ne sont pas UTF-8. Parce que `into` prend son argument par valeur, `text` n'est plus initialisé après la conversion, ce qui signifie que nous pouvons accéder librement au `String` tampon du premier sans pouvoir corrompre aucun fichier existant `String`.

Cependant, les conversions bon marché ne font pas partie du contrat de `Into` and `From`. Alors `AsRef` que les conversions et `AsMut` sont censées être bon marché, `From` et `Into` les conversions peuvent allouer, copier ou autrement traiter le contenu de la valeur. Par exemple, `String` implemente `From<&str>`, qui copie la tranche de chaîne dans un nouveau tampon alloué par tas pour le `String`. Et
`std::collections::BinaryHeap<T>` implemente `From<Vec<T>>`, qui compare et réordonne les éléments en fonction des exigences de son algorithme.

L' `? opérateur utilise From et Into pour aider à nettoyer le code dans les fonctions qui pourraient échouer de plusieurs façons en convertissant automatiquement des types d'erreurs spécifiques en types généraux si nécessaire.`

Par exemple, imaginez un système qui doit lire des données binaires et en convertir une partie à partir de nombres en base 10 écrits sous forme de texte UTF-8. Cela signifie utiliser `std::str::from_utf8` et l'`FromStr` implémentation de `i32`, qui peuvent chacune renvoyer des er-

reurs de types différents. En supposant que nous utilisions les types `GenericError` et que nous avons définis au [chapitre 7](#) lors de la discussion sur la gestion des erreurs, l' opérateur effectuera la conversion pour nous : `GenericResult` ?

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

fn parse_i32_bytes(b: &[u8]) -> GenericResult<i32> {
    Ok(std::str::from_utf8(b)?.parse::<i32>()?)
}
```

Comme la plupart des types d'erreurs, `Utf8Error` et `ParseIntError` implémentent le `Error` trait, et la bibliothèque standard nous donne une couverture `From` `impl` pour convertir tout ce qui implémente `Error` en un `Box<dyn Error>`, qui ? utilise automatiquement :

```
impl<'a, E: Error + Send + Sync + 'a> From<E>
for Box<dyn Error + Send + Sync + 'a> {
    fn from(err: E) -> Box<dyn Error + Send + Sync + 'a> {
        Box::new(err)
    }
}
```

Cela transforme ce qui aurait été une fonction assez volumineuse avec deux `match` instructions en une seule ligne.

Avant `From` et `Into` ont été ajoutés à la bibliothèque standard, le code Rust était plein de traits de conversion et de méthodes de construction ad hoc, chacun spécifique à un seul type. `From` et `Into` codifient les conventions que vous pouvez suivre pour rendre vos types plus faciles à utiliser, puisque vos utilisateurs les connaissent déjà. D'autres bibliothèques et le langage lui-même peuvent également s'appuyer sur ces caractéristiques comme moyen canonique et standardisé d'encoder les conversions.

`From` et `Into` sont des traits infaillibles - leur API exige que les conversions n'échouent pas. Malheureusement, de nombreuses conversions sont plus complexes que cela. Par exemple, de grands nombres entiers comme `i64` peuvent stocker des nombres bien plus grands que `i32`, et convertir un nombre comme `2_000_000_000i64` en `i32` n'a pas beaucoup de sens sans quelques informations supplémentaires. Faire une simple conversion au niveau du bit, dans laquelle les 32 premiers bits sont supprimés, ne donne pas souvent le résultat que nous espérions :

```
let huge = 2_000_000_000_000i64;
let smaller = huge as i32;
println!("{}", smaller); // -1454759936
```

Il existe de nombreuses options pour gérer cette situation. Selon le contexte, une telle conversion « enveloppante » peut être appropriée. D'autre part, des applications telles que le traitement numérique du signal et les systèmes de contrôle peuvent souvent se contenter d'une conversion "saturante", dans laquelle les nombres supérieurs à la valeur maximale possible sont limités à cette valeur maximale..

TryFrom et TryInto

Puisqu'il n'est pas clair comment une telle conversion devrait se comporter, Rust n'implémente pas `From<i64> for i32`, ou toute autre conversion entre types numériques qui perdrait des informations. Au lieu de cela, `i32` implémente `TryFrom<i64>`. `TryFrom` et `TryInto` sont les cousins faillibles de `From` et `Into` et sont pareillement réciproques ; moyens de mise en œuvre `TryFrom` qui `TryInto` sont également mis en œuvre.

Leurs définitions sont seulement un peu plus complexes que `From` et `Into`.

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

La `try_into()` méthode nous donne un `Result`, afin que nous puissions choisir ce qu'il faut faire dans le cas exceptionnel, comme un nombre trop grand pour tenir dans le type résultant :

```
// Saturate on overflow, rather than wrapping
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

Si nous voulons également traiter le cas négatif, nous pouvons utiliser la `unwrap_or_else()` méthode de `Result`:

```
let smaller: i32 = huge.try_into().unwrap_or_else(|_| {
    if huge >= 0 {
        i32::MAX
    } else {
        i32::MIN
    }
});
```

La mise en œuvre de conversions faillibles pour vos propres types est également facile. Le `Error` type peut être aussi simple ou aussi complexe qu'une application particulière l'exige. La bibliothèque standard utilise une structure vide, ne fournissant aucune information autre que le fait qu'une erreur s'est produite, puisque la seule erreur possible est un débordement. D'un autre côté, les conversions entre des types plus complexes peuvent vouloir renvoyer plus d'informations :

```
impl TryInto<LinearShift> for Transform {
    type Error = TransformError;

    fn try_into(self) -> Result<LinearShift, Self:: Error> {
        if !self.normalized() {
            return Err(TransformError::NotNormalized);
        }
        ...
    }
}
```

Où `From` et `Into` relient les types avec des conversions simples, `TryFrom` et `TryInto` étendent la simplicité de `From` et `Into` les conversions avec la gestion expressive des erreurs offerte par `Result`. Ces quatre traits peuvent être utilisés ensemble pour relier plusieurs types dans une seule caisse.

À Owned

Compte tenu d'une référence, la manière habituelle de produire une propriété copie de son référent est d'appeler `clone`, en supposant que le type implémente `std::clone::Clone`. Mais que se passe-t-il si vous voulez cloner a `&str` ou a `&[i32]`? Ce que vous voulez probablement,

c'est à `String` ou à `Vec<i32>`, mais `Clone` la définition de ne le permet pas : par définition, le clonage de `a &T` doit toujours renvoyer une valeur de type `T`, et `str` et `[u8]` ne sont pas dimensionnés ; ce ne sont même pas des types qu'une fonction pourrait renvoyer.

Le `std::borrow::ToOwned` trait fournit un moyen légèrement plus lâche de convertir une référence en une valeur possédée :

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

Contrairement à `clone`, qui doit retourner exactement `Self`, `to_owned` peut retourner tout ce à quoi vous pourriez emprunter `&Self` : le `Owned` type doit implémenter `Borrow<Self>`. Vous pouvez emprunter à `&[T]` à `a Vec<T>`, donc `[T]` vous pouvez mettre en œuvre `ToOwned<Owned=Vec<T>>`, tant que met en `T` œuvre `Clone`, afin que nous puissions copier les éléments de la tranche dans le vecteur. De même, `str` implements `ToOwned<Owned=String>`, `Path` implements `ToOwned<Owned=PathBuf>`, etc.

Emprunter et posséder au travail : la vache humble

Faire bon usage de Rust implique de réfléchir à des questions de propriété, comme si une fonction doit recevoir un paramètre par référence ou par valeur. Habituellement, vous pouvez choisir une approche ou l'autre, et le type de paramètre reflète votre décision. Mais dans certains cas, vous ne pouvez pas décider d'emprunter ou de devenir propriétaire tant que le programme n'est pas en cours d'exécution ; le `std::borrow::Cow` genre (pour "cloner en écriture") fournit un moyen de le faire.

Sa définition est présentée ici :

```
enum Cow<'a, B: ?Sized>
    where B: ToOwned
{
    Borrowed(&'a B),
```

```
    Owned(<B as ToOwned>::Owned),  
}
```

A `Cow` soit emprunte une référence partagée à `a` `B` ou possède une valeur à laquelle on pourrait emprunter une telle référence. Depuis `Cow` implements `Deref`, vous pouvez appeler des méthodes dessus comme s'il s'agissait d'une référence partagée à `a` `B` : si c'est `Owned`, il emprunte une référence partagée à la valeur possédée ; et si c'est `Borrowed`, il distribue simplement la référence qu'il détient.

Vous pouvez également obtenir une référence mutable à `Cow` la valeur de `a` en appelant sa `to_mut` méthode, qui renvoie `a &mut B`. Si le `Cow` se trouve être `Cow::Borrowed`, `to_mut` appelle simplement la `to_owned` méthode de la référence pour obtenir sa propre copie du référent, change le `Cow` en un `Cow::Owned`, et emprunte une référence mutable à la valeur nouvellement possédée. Il s'agit du comportement de « clonage en écriture » auquel le nom du type fait référence.

De même, `Cow` a une `into_owned` méthode qui promeut la référence à une valeur possédée, si nécessaire, puis la renvoie, transférant la propriété à l'appelant et consommant le `Cow` dans le processus.

Une utilisation courante de `Cow` consiste à renvoyer soit une constante de chaîne allouée statiquement, soit une chaîne calculée. Par exemple, supposons que vous deviez convertir une énumération d'erreur en message. La plupart des variantes peuvent être gérées avec des chaînes fixes, mais certaines d'entre elles ont des données supplémentaires qui doivent être incluses dans le message. Vous pouvez retourner un `Cow<'static, str>`:

```
use std::path::PathBuf;  
use std::borrow::Cow;  
fn describe(error: &Error) -> Cow<'static, str> {  
    match *error {  
        Error::OutOfMemory => "out of memory".into(),  
        Error::StackOverflow => "stack overflow".into(),  
        Error::MachineOnFire => "machine on fire".into(),  
        Error::Unfathomable => "machine bewildered".into(),  
        Error::NotFound(ref path) => {  
            format!("file not found: {}", path.display()).into()  
        }  
    }  
}
```

Ce code utilise `Cow` l'implémentation de `Into` pour construire les valeurs. La plupart des bras de cette `match` instruction renvoient une `Cow::Borrowed` référence à une chaîne allouée statiquement. Mais lorsque nous obtenons une `FileNotFoundException` variante, nous l'utilisons `format!` pour construire un message incorporant le nom de fichier donné. Ce bras de l'`match` instruction produit une `Cow::Owned` valeur.

Les appelants de `describe` qui n'ont pas besoin de changer la valeur peuvent simplement traiter le `Cow` comme un `&str`:

```
println!("Disaster has struck: {}", describe(&error));
```

Les appelants qui ont besoin d'une valeur possédée peuvent facilement en produire une :

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

L'utilisation des `Cow` aides `describe` et de ses appelants reporte l'attribution jusqu'au moment où cela devient nécessaire.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 14. Fermetures

Sauver l'environnement! Créez une fermeture aujourd'hui !

—Cormac Flanagan

Trier un vecteur d'entiers est facile :

```
integers.sort();
```

Il est donc triste de constater que lorsque nous voulons trier des données, il ne s'agit presque jamais d'un vecteur d'entiers. Nous avons généralement des enregistrements d'un certain type, et la `sort` méthode intégrée ne fonctionne généralement pas :

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities:&mut Vec<City>) {
    cities.sort(); // error: how do you want them sorted?
}
```

Rust se plaint de `City` ne pas implémenter `std::cmp::Ord`. Nous devons spécifier l'ordre de tri, comme ceci :

```
/// Helper function for sorting cities by population.
fn city_population_descending(city: &City) ->i64 {
    -city.population
}

fn sort_cities(cities:&mut Vec<City>) {
    cities.sort_by_key(city_population_descending); // ok
}
```

La fonction d'assistance, `city_population_descending`, prend un `City` enregistrement et extrait la *clé*, le champ par lequel nous voulons trier nos données. (Il renvoie un nombre négatif car `sort` organise les nombres dans l'ordre croissant, et nous voulons l'ordre décroissant : la

ville la plus peuplée en premier.) La `sort_by_key` méthode prend cette fonction clé comme paramètre.

Cela fonctionne bien, mais il est plus concis d'écrire la fonction d'assistance sous la forme d'une *fermeture*, une expression de fonction anonyme :

```
fn sort_cities(cities:&mut Vec<City>) {  
    cities.sort_by_key(|city| -city.population);  
}
```

La fermeture ici est `|city| -city.population`. Il prend un argument `city` et renvoie `-city.population`. Rust déduit le type d'argument et le type de retour de la façon dont la fermeture est utilisée.

D'autres exemples de fonctionnalités de bibliothèque standard qui acceptent les fermetures incluent :

- Iterator des méthodes telles que `map` et `filter`, pour travailler avec des données séquentielles. Nous aborderons ces méthodes au [chapitre 15](#).
- Des API de thread comme `thread::spawn`, qui démarrent un nouveau thread système. La simultanéité consiste à déplacer le travail vers d'autres threads, et les fermetures représentent commodément des unités de travail. Nous aborderons ces fonctionnalités au [chapitre 19](#).
- Certaines méthodes qui nécessitent conditionnellement de calculer une valeur par défaut, comme la `or_insert_with` méthode des `HashMap` entrées. Cette méthode obtient ou crée une entrée dans un `HashMap`, et elle est utilisée lorsque la valeur par défaut est coûteuse à calculer. La valeur par défaut est transmise en tant que fermeture qui n'est appelée que si une nouvelle entrée doit être créée.

Bien sûr, les fonctions anonymes sont partout de nos jours, même dans des langages comme Java, C#, Python et C++ qui n'en avaient pas à l'origine. À partir de maintenant, nous supposerons que vous avez déjà vu des fonctions anonymes et nous nous concentrerons sur ce qui rend les fermetures de Rust un peu différentes. Dans ce chapitre, vous apprendrez les trois types de fermetures, comment utiliser des fermetures avec des méthodes de bibliothèque standard, comment une fermeture peut « capturer » des variables dans sa portée, comment écrire vos propres fonctions et méthodes qui prennent des fermetures comme arguments, et comment stocker les fermetures pour une utilisation ultérieure comme rappels. Nous expliquerons également comment les fermetures Rust sont implémentées et pourquoi elles sont plus rapides que prévu.

Capture de variables

Une fermeture peut utiliser des données appartenant à une fonction englobante. Par exemple:

```
// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

La fermeture utilise ici `stat`, qui appartient à la fonction englobante, `sort_by_statistic`. On dit que la fermeture « capte » `stat`. C'est l'une des caractéristiques classiques des fermetures, donc naturellement, Rust la prend en charge ; mais dans Rust, cette fonctionnalité est accompagnée d'une chaîne.

Dans la plupart des langues avec fermetures, ramasse-miettes joue un rôle important. Par exemple, considérez ce code JavaScript :

```
// Start an animation that rearranges the rows in a table of cities.
function startSortingAnimation(cities, stat) {
    // Helper function that we'll use to sort the table.
    // Note that this function refers to stat.
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // Now kick off an animation, passing keyfn to it.
    // The sorting algorithm will call keyfn later.
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

La fermeture `keyfn` est stockée dans le nouvel `SortingAnimation` objet. Il est censé être appelé après les `startSortingAnimation` retours. Désormais, normalement, lorsqu'une fonction revient, toutes ses variables et tous ses arguments sortent de la portée et sont supprimés. Mais ici, le moteur JavaScript doit rester `stat` en place d'une manière ou d'une autre, puisque la fermeture l'utilise. La plupart des moteurs JavaScript le font en allouant `stat` dans le tas et en laissant le ramasse-miettes le récupérer plus tard.

Rust n'a pas de ramasse-miettes. Comment cela fonctionnera-t-il ? Pour répondre à cette question, nous allons examiner deux exemples.

Des fermetures qui empruntent

D'abord, répétons l'exemple d'ouverture de cette section :

```
// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

Dans ce cas, lorsque Rust crée la fermeture, il emprunte automatiquement une référence à `stat`. Cela va de soi : la fermeture fait référence à `stat`, elle doit donc y faire référence.

Le reste est simple. La fermeture est soumise aux règles d'emprunt et de durée de vie que nous avons décrites au [chapitre 5](#). En particulier, puisque la fermeture contient une référence à `stat`, Rust ne la laissera pas survivre à `stat`. Étant donné que la fermeture n'est utilisée que lors du tri, cet exemple est correct.

En bref, Rust assure la sécurité en utilisant des durées de vie au lieu de la collecte des ordures. La méthode de Rust est plus rapide : même une allocation GC rapide sera plus lente que le stockage `stat` sur la pile, comme le fait Rust dans ce cas.

Des fermetures qui volent

Le deuxième exemple est plus délicat :

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = |city: &City| ->i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

Cela ressemble un peu plus à ce que faisait notre exemple JavaScript :

`thread::spawn` prend une fermeture et l'appelle dans un nouveau

thread système. Notez qu'il || s'agit de la liste d'arguments vide de la fermeture.

Le nouveau thread s'exécute en parallèle avec l'appelant. Lorsque la fermeture revient, le nouveau thread se ferme. (La valeur de retour de la fermeture est renvoyée au thread appelant en tant que `JoinHandle` valeur. Nous aborderons cela au [chapitre 19](#).)

Encore une fois, la fermeture `key_fn` contient une référence à `stat`. Mais cette fois, Rust ne peut garantir que la référence est utilisée en toute sécurité. Rust rejette donc ce programme :

```
error: closure may outlive the current function, but it borrows `stat`,
      which is owned by the current function
|
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
|          ^^^^^^
|          |
|          |           `stat` is borrowed
|          |           may outlive borrowed value `stat`
```

En fait, il y a deux problèmes ici, car il `cities` est également partagé de manière non sécurisée. Tout simplement, `thread::spawn` on ne peut pas s'attendre à ce que le nouveau thread créé par finisse son travail avant `cities` et `stat` soit détruit à la fin de la fonction.

La solution aux deux problèmes est le même : dites à Rust de *déplacer* `cities` et `stat` dans les fermetures qui les utilisent au lieu de leur emprunter des références.

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| ->i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

La seule chose que nous avons changée est d'ajouter le mot- `move` clé avant chacune des deux fermetures. Le mot- `move` clé indique à Rust qu'une fermeture n'emprunte pas les variables qu'elle utilise : elle les vole.

La première fermeture, `key_fn`, s'approprie `stat`. Ensuite, la deuxième fermeture s'approprie à la fois `cities` et `key_fn`.

Rust offre donc deux façons pour les fermetures d'obtenir des données à partir de portées englobantes : les déplacements et l'emprunt. Vraiment, il n'y a rien de plus à dire que cela; les fermetures suivent les mêmes règles concernant les déménagements et les emprunts que nous avons déjà abordées dans les chapitres [4](#) et [5](#). Quelques cas concrets :

- Comme partout ailleurs dans le langage, si une fermeture était `move` une valeur d'un type copiable, comme `i32`, elle copie la valeur à la place. Donc, s'il `Statistic` s'agissait d'un type copiable, nous pourrions continuer à l'utiliser `stat` même après avoir créé une `move` fermeture qui l'utilise.
- Les valeurs de types non copiables, comme `Vec<City>`, sont vraiment déplacées : le code précédent est transféré `cities` dans le nouveau thread, par le biais de la `move` fermeture. Rust ne nous laisserait pas accéder `cities` par nom après avoir créé la fermeture.
- En l'occurrence, ce code n'a pas besoin d'être utilisé `cities` après le point où la fermeture le déplace. Si nous le faisions, cependant, la solution de contournement serait simple : nous pourrions dire à Rust de cloner `cities` et de stocker la copie dans une variable différente. La fermeture ne volerait qu'une des copies, quelle que soit celle à laquelle elle se réfère.

Nous obtenons quelque chose d'important en acceptant les règles strictes de Rust : la sécurité des threads. C'est précisément parce que le vecteur est déplacé, plutôt que d'être partagé entre les threads, que nous savons que l'ancien thread ne libérera pas le vecteur pendant que le nouveau thread le modifie.

Types de fonction et de fermeture

Tout au long de ce chapitre, nous avons vu les fonctions et les fermetures utilisées comme valeurs. Naturellement, cela signifie qu'ils ont des types. Par exemple:

```
fn city_population_descending(city: &City) ->i64 {  
    -city.population  
}
```

Cette fonction prend un argument (`a &City`) et renvoie un `i64`. Il a le type `fn(&City) -> i64`.

Vous pouvez faire la même chose avec des fonctions qu'avec d'autres valeurs. Vous pouvez les stocker dans des variables. Vous pouvez utiliser toute la syntaxe habituelle de Rust pour calculer les valeurs des fonctions :

```
let my_key_fn: fn(&City) ->i64 =
    if user.prefs.by_population {
        city_population_descending
    } else {
        city_monster_attack_risk_descending
    };

cities.sort_by_key(my_key_fn);
```

Les structures peuvent avoir des champs de type fonction. Les types génériques comme `vec` peuvent stocker des tas de fonctions, tant qu'ils partagent tous le même `fn` type. Et les valeurs des fonctions sont minimales : une `fn` valeur est l'adresse mémoire du code machine de la fonction, tout comme un pointeur de fonction en C++.

Une fonction peut prendre une autre fonction comme argument. Par exemple:

```
/// Given a list of cities and a test function,
/// return how many cities pass the test.
fn count_selected_cities(cities: &Vec<City>,
                         test_fn: fn(&City) -> bool) -> usize
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}

/// An example of a test function. Note that the type of
/// this function is `fn(&City) -> bool`, the same as
/// the `test_fn` argument to `count_selected_cities`.
fn has_monster_attacks(city: &City) ->bool {
    city.monster_attack_risk > 0.0
}

// How many cities are at risk for monster attack?
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

Si vous êtes familier avec les pointeurs de fonction en C/C++, vous verrez que les valeurs de fonction de Rust sont exactement la même chose.

Après tout cela, il peut être surprenant que les fermetures n'aient *pas* le même type que les fonctions :

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // error: type mismatch
```

Le deuxième argument provoque une erreur de type. Pour prendre en charge les fermetures, nous devons modifier la signature de type de cette fonction. Il doit ressembler à ceci :

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) ->bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

Nous n'avons changé que la signature de type de `count_selected_cities`, pas le corps. La nouvelle version est générique. Il faut un `test_fn` de n'importe quel type `F` tant qu'il `F` implémente le trait spécial `Fn(&City) -> bool`. Ce trait est automatiquement implémenté par toutes les fonctions et la plupart des fermetures qui prennent un single `&City` comme argument et renvoient une valeur booléenne :

```
fn(&City) -> bool      // fn type (functions only)
Fn(&City) ->bool      // Fn trait (both functions and closures)
```

Cette syntaxe spéciale est intégrée au langage. Le `->` et le type de retour sont facultatifs ; s'il est omis, le type de retour est `()`.

La nouvelle version de `count_selected_cities` accepte soit une fonction, soit une fermeture :

```

count_selected_cities(
    &my_cities,
    has_monster_attacks); // ok

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // also ok

```

Pourquoi notre première tentative n'a-t-elle pas fonctionné ? Eh bien, une fermeture est appellable, mais ce n'est pas un `fn`. La fermeture `|city| city.monster_attack_risk > limit` a son propre type qui n'est pas un `fn` type.

En fait, chaque fermeture que vous écrivez a son propre type, car une fermeture peut contenir des données : des valeurs empruntées ou volées dans les portées englobantes. Il peut s'agir de n'importe quel nombre de variables, dans n'importe quelle combinaison de types. Ainsi, chaque fermeture a un type ad hoc créé par le compilateur, suffisamment grand pour contenir ces données. Il n'y a pas deux fermetures exactement du même type. Mais chaque fermeture met en œuvre un `Fn` trait ; la fermeture dans notre exemple implémente `Fn(&City) -> i64`.

Étant donné que chaque fermeture a son propre type, le code qui fonctionne avec les fermetures doit généralement être générique, comme `count_selected_cities`. C'est un peu maladroit d'épeler les types génériques à chaque fois, mais pour voir les avantages de cette conception, il suffit de lire la suite.

Performances de fermeture

Les fermetures de Rust sont conçus pour être rapides : plus rapides que les pointeurs de fonction, suffisamment rapides pour que vous puissiez les utiliser même dans du code brûlant et sensible aux performances. Si vous êtes familier avec les lambdas C++, vous constaterez que les fermetures Rust sont tout aussi rapides et compactes, mais plus sûres.

Dans la plupart des langages, les fermetures sont allouées dans le tas, réparties dynamiquement et récupérées. Ainsi, créer, appeler et collecter chacun d'eux coûte un tout petit peu de temps CPU supplémentaire. Pire encore, les fermetures ont tendance à exclure l'*inlining*, une technique clé utilisée par les compilateurs pour éliminer la surcharge des appels de fonction et permettre une multitude d'autres optimisations. Tout compte fait, les fermetures sont suffisamment lentes dans ces langages pour qu'il

puisse valoir la peine de les supprimer manuellement des boucles internes étroites.

Les fermetures antirouille ne présentent aucun de ces inconvénients de performance. Ce ne sont pas des ordures ménagères. Comme tout le reste dans Rust, ils ne sont pas alloués sur le tas à moins que vous ne les placiez dans un conteneur `Box`, `Vec` ou autre. Et puisque chaque fermeture a un type distinct, chaque fois que le compilateur Rust connaît le type de fermeture que vous appelez, il peut incorporer le code pour cette fermeture particulière. Cela permet d'utiliser des fermetures dans des boucles serrées, et les programmes Rust le font souvent, avec enthousiasme, comme vous le verrez au [chapitre 15](#).

[La figure 14-1](#) montre comment les fermetures Rust sont disposées en mémoire. En haut de la figure, nous montrons quelques variables locales auxquelles nos fermetures feront référence : une chaîne `food` et un simple enum `weather`, dont la valeur numérique se trouve être 27.

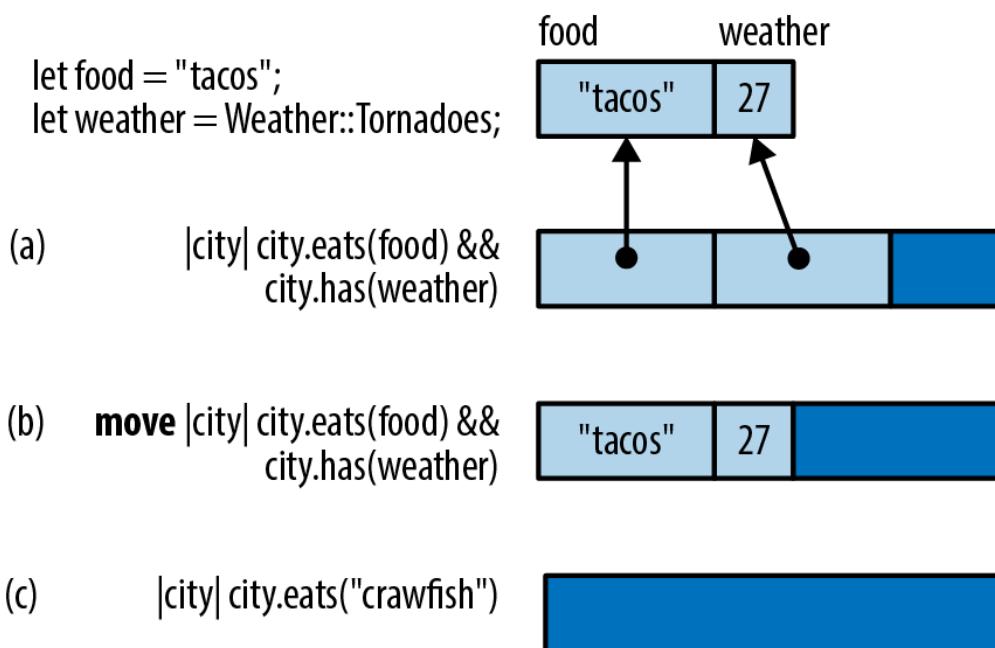


Illustration 14-1. Disposition des fermetures en mémoire

La clôture (a) utilise les deux variables. Apparemment, nous recherchons des villes qui ont à la fois des tacos et des tornades. En mémoire, cette fermeture ressemble à une petite structure contenant des références aux variables qu'elle utilise.

Notez qu'il ne contient pas de pointeur vers son code ! Ce n'est pas nécessaire : tant que Rust connaît le type de fermeture, il sait quel code exécuter lorsque vous l'appelez.

La fermeture (b) est exactement la même, sauf que c'est une `move` fermeture, donc elle contient des valeurs au lieu de références.

Closure (c) n'utilise aucune variable de son environnement. La structure est vide, donc cette fermeture ne prend aucune mémoire.

Comme le montre la figure, ces fermetures ne prennent pas beaucoup de place. Mais même ces quelques octets ne sont pas toujours nécessaires dans la pratique. Souvent, le compilateur peut intégrer tous les appels à une fermeture, puis même les petites structures présentées dans cette figure sont optimisées.

Dans ["Callbacks"](#), nous montrerons comment allouer des fermetures dans le tas et les appeler dynamiquement, à l'aide d'objets trait. C'est un peu plus lent, mais c'est toujours aussi rapide que n'importe quelle autre méthode d'objet trait.

Fermetures et sécurité

À traversJusqu'à présent, dans le chapitre, nous avons expliqué comment Rust s'assure que les fermetures respectent les règles de sécurité du langage lorsqu'elles empruntent ou déplacent des variables du code environnant. Mais il y a d'autres conséquences qui ne sont pas exactement évidentes. Dans cette section, nous expliquerons un peu plus ce qui se passe lorsqu'une fermeture supprime ou modifie une valeur capturée.

Des fermetures qui tuent

Nous avons vu des fermetures qui empruntent des valeurs et des fermetures qui les volent ; ce n'était qu'une question de temps avant qu'ils ne tournent mal.

Bien sûr, *tuer* n'est pas vraiment la bonne terminologie. A Rust, on laisse tomber valeurs. La façon la plus simple de le faire est d'appeler `drop()` :

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

Quand `f` est appelé, `my_str` est abandonné.

Alors que se passe-t-il si nous l'appelons deux fois ?

```
f();
f();
```

Réfléchissons-y. La première fois que nous appelons `f`, il abandonne `my_str`, ce qui signifie que la mémoire où la chaîne est stockée est libérée, renvoyée au système. La deuxième fois que nous appelons `f`, la même chose se produit. C'est un *double free*, une erreur classique de la programmation C++ qui déclenche un comportement indéfini.

Laisser tomber `String` deux fois serait une tout aussi mauvaise idée dans Rust. Heureusement, Rust ne se laisse pas tromper aussi facilement :

```
f(); // ok
f(); // error: use of moved value
```

Rust sait que cette fermeture ne peut pas être appelée deux fois.

Une fermeture qui ne peut être appelée qu'une seule fois peut sembler quelque chose d'assez extraordinaire, mais nous avons parlé tout au long de ce livre de propriété et de durée de vie. L'idée que les valeurs sont épuisées (c'est-à-dire déplacées) est l'un des concepts fondamentaux de Rust. Cela fonctionne de la même manière avec les fermetures qu'avec tout le reste.

FnOnce

Essayons une fois plus pour inciter Rust à en laisser tomber `String` deux fois. Cette fois, nous allons utiliser cette fonction générique :

```
fn call_twice<F>(closure: F) where F:Fn() {
    closure();
    closure();
}
```

Cette fonction générique peut recevoir n'importe quelle fermeture qui implémente le trait `Fn()` : c'est-à-dire des fermetures qui ne prennent aucun argument et retournent `()`. (Comme pour les fonctions, le type de retour peut être omis s'il s'agit de `()` ; `Fn()` est un raccourci pour `Fn() -> ()`.)

Maintenant, que se passe-t-il si nous passons notre fermeture non sécurisée à cette fonction générique ?

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

Encore une fois, la fermeture tombera `my_str` quand elle sera appelée.

L'appeler deux fois serait un double gratuit. Mais encore une fois, Rust n'est pas dupe :

```
error: expected a closure that implements the `Fn` trait, but
      this closure only implements `FnOnce`  
|  
8 | let f = || drop(my_str);  
|     ^^^^^^-----^  
|     |  
|     |     closure is `FnOnce` because it moves the variable `my_  
|     |     out of its environment  
|     |     this closure implements `FnOnce`, not `Fn`  
9 | call_twice(f);  
| ----- the requirement to implement `Fn` derives from here
```

Ce message d'erreur nous en dit plus sur la façon dont Rust gère les « fermetures qui tuent ». Ils auraient pu être entièrement bannis de la langue, mais les fermetures de nettoyage sont parfois utiles. Donc, à la place, Rust restreint leur utilisation. Les fermetures qui suppriment des valeurs, comme `f`, ne sont pas autorisées à avoir `Fn`. Ils sont, littéralement, non `Fn` du tout. Ils implémentent un trait moins puissant, `FnOnce`, le trait des fermetures qui peuvent être appelées une fois.

La première fois que vous appelez une `FnOnce` fermeture, *la fermeture elle-même est épuisée*. C'est comme si les deux traits, `Fn` et `FnOnce`, étaient définis comme ceci :

```
// Pseudocode pour les traits `Fn` et `FnOnce` sans arguments.
trait Fn() -> R {
    fn call( &self ) -> R;
}

trait FnOnce() -> R {
    fn call_once( self ) -> R;
}
```

Tout comme une expression arithmétique comme `a + b` est un raccourci pour un appel de méthode `Add::add(a, b)`, Rust traite `closure()` comme un raccourci pour l'une des deux méthodes de trait présentées dans l'exemple précédent. Pour une `Fn` fermeture, `closure()` se développe en `closure.call()`. Cette méthode prend `self` par référence, donc la fermeture n'est pas déplacée. Mais si la fermeture ne peut être appelée qu'une seule fois, alors elle `closure()` se développe en `closure.call_once()`. Cette méthode prend `self` par valeur, donc la fermeture est épuisée.

Bien sûr, nous avons délibérément semé le trouble ici en utilisant `drop()`. En pratique, vous vous retrouverez la plupart du temps dans cette situation par accident. Cela n'arrive pas souvent, mais de temps en temps, vous écrivez un code de fermeture qui utilise involontairement une valeur :

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict { // oops!
        println!("{} - {}", key, value);
    }
};
```

Ensuite, lorsque vous appelez `debug_dump_dict()` plus d'une fois, vous obtenez un message d'erreur comme celui-ci :

```
error: use of moved value: `debug_dump_dict`
|
19 |     debug_dump_dict();
|----- `debug_dump_dict` moved due to this call
20 |     debug_dump_dict();
|     ^^^^^^^^^^^^^^^^ value used here after move
|
note: closure cannot be invoked more than once because it moves the variable
`dict` out of its environment
|
13 |         for (key, value) in dict {
|             ^^^^
```

Pour déboguer cela, nous devons comprendre pourquoi la fermeture est un `FnOnce`. Quelle valeur est utilisée ici ? Le compilateur souligne utilement qu'il s'agit de `dict`, qui dans ce cas est le seul auquel nous faisons référence. Ah, voilà le bogue : nous l'utilisons `dict` en itérant directement dessus. Nous devrions boucler sur `&dict`, plutôt que plain `dict`, pour accéder aux valeurs par référence :

```
let debug_dump_dict = || {
    for (key, value) in &dict { // does not use up dict
        println!("{} - {}", key, value);
    }
};
```

Cela corrige l'erreur ; la fonction est maintenant un `Fn` et peut être appelée n'importe quel nombre de fois.

Mute Fn

Il y a un autre type de fermeture, le type qui contient des données ou des `mut` références modifiables.

Rust considère que les non `mut`-valeurs peuvent être partagées en toute sécurité entre les threads. Mais il ne serait pas sûr de partager des non `mut`-clôtures contenant des `mut` données : appeler une telle fermeture à partir de plusieurs threads pourrait entraîner toutes sortes de conditions de concurrence, car plusieurs threads tentent de lire et d'écrire les mêmes données en même temps.

Par conséquent, Rust a une autre catégorie de fermeture, `FnMut`, la catégorie des fermetures qui écrivent. `FnMut` les fermetures sont appelées par `mut` référence, comme si elles étaient définies comme ceci :

```
// Pseudocode pour les traits `Fn`, `FnMut` et `FnOnce`.  
trait Fn() -> R {  
    fn appel(&self) -> R ;  
}  
  
trait FnMut() -> R {  
    fn call_mut( &mut self ) -> R;  
}  
  
trait FnOnce() -> R {  
    fn call_once(soi) -> R ;  
}
```

Toute fermeture qui nécessite `mut` l'accès à une valeur, mais qui ne supprime aucune valeur, est une `FnMut` fermeture. Par exemple:

```
let mut i = 0;  
let incr = || {  
    i += 1; // incr borrows a mut reference to i  
    println!("Ding! i is now: {}", i);  
};  
call_twice(incr);
```

La façon dont nous avons écrit `call_twice`, nécessite un fichier `Fn`. Puisque `incr` est un `FnMut` et non un `Fn`, ce code ne se compile pas. Il y a une solution facile, cependant. Pour comprendre le correctif, prenons un peu de recul et résumons ce que vous avez appris sur les trois catégories de fermetures Rust.

- Fn est la famille de fermetures et de fonctions que vous pouvez appeler plusieurs fois sans restriction. Cette catégorie la plus élevée comprend également toutes les fn fonctions.
- FnMut est la famille de fermetures qui peut être appelée plusieurs fois si la fermeture elle-même est déclarée mut .
- FnOnce est la famille de fermetures qui peut être appelée une fois, si l'appelant possède la fermeture.

Chaque Fn répond aux exigences de FnMut , et chaque FnMut répond aux exigences de FnOnce . Comme le montre la [Figure 14-2](#) , il ne s'agit pas de trois catégories distinctes.

Au lieu de cela, Fn() est un sous-trait de FnMut() , qui est un sous-trait de FnOnce() . Cela en fait Fn la catégorie la plus exclusive et la plus puissante. FnMut et FnOnce sont des catégories plus larges qui incluent des fermetures avec des restrictions d'utilisation.

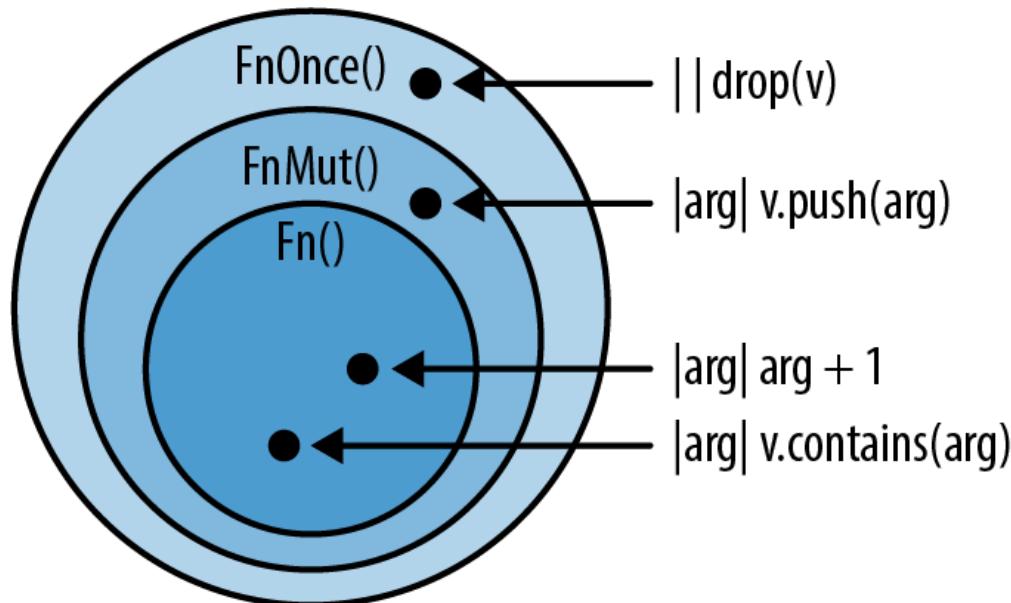


Illustration 14-2. Diagramme de Venn des trois catégories de fermeture

Maintenant que nous avons organisé ce que nous savons, il est clair que pour accepter le plus large éventail possible de fermetures, notre call_twice fonction doit vraiment accepter toutes les FnMut fermetures, comme ceci :

```
fn call_twice<F>(mut closure: F) where F:FnMut() {
    closure();
    closure();
}
```

La limite sur la première ligne était F: Fn() , et maintenant c'est F: FnMut() . Avec ce changement, nous acceptons toujours toutes les Fn fer-

metures, et nous pouvons également utiliser `call_twice` sur les fermetures qui modifient les données:

```
let mut i = 0;
call_twice(|| i += 1); // ok!
assert_eq!(i, 2);
```

Copier et cloner pour les fermetures

Tout comme Rust détermine automatiquement quelles fermetures ne peuvent être appelées qu'une seule fois, il peut déterminer quelles fermetures peuvent implémenter `Copy` et `Clone`, et qui ne le peut pas.

Comme nous l'avons expliqué précédemment, les fermetures sont représentées sous forme de structures contenant soit les valeurs (pour les `move` fermetures), soit des références aux valeurs (pour les non-`move` fermetures) des variables qu'elles capturent. Les règles pour `Copy` et `Clone` sur les fermetures sont comme les règles `Copy` et `Clone` pour les structures régulières. Une non `move`-fermeture qui ne mute pas les variables ne contient que des références partagées, qui sont à la fois `Clone` et `Copy`, de sorte que la fermeture est à la fois `Clone` et `Copy` aussi :

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y; // This closure is `Copy`, so...
assert_eq!(add_y(copy_of_add_y(22)), 42); // ... we can call both.
```

D'un autre côté, une non `move`-fermeture qui *fait* muter des valeurs a des références mutables dans sa représentation interne. Les références mutables ne sont ni `clone` ni ni `Copy`, donc une fermeture qui les utilise ne l'est pas non plus :

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x; // this moves, rather than copies
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // error: use of moved value
```

Pour une `move` fermeture, les règles sont encore plus simples. Si tout ce qu'une `move` fermeture capture est `Copy`, c'est `Copy`. Si tout ce qu'il capture est `Clone`, c'est `Clone`. Par exemple:

```

let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{} ", greeting);
};
greet.clone()( "Alfred");
greet.clone()( "Bruce");

```

Cette `.clone()(...)` syntaxe est un peu bizarre, mais cela signifie simplement que nous clonons la fermeture, puis appelons le clone. Ce programme produit :

```
Hello, Alfred
Hello, Bruce
```

Lorsqu'il `greeting` est utilisé dans `greet`, il est déplacé dans la structure qui représente `greet` en interne, car il s'agit d'une `move` fermeture. Ainsi, lorsque nous clonons `greet`, tout ce qu'il contient est également cloné. Il existe deux copies de `greeting`, qui sont chacune modifiées séparément lorsque les clones de `greet` sont appelés. Ce n'est pas si utile en soi, mais lorsque vous devez passer la même fermeture dans plus d'une fonction, cela peut être très utile.

Rappels

De nombreuses bibliothèques utilisent *des rappels* dans le cadre de leur API : fonctions fournies par l'utilisateur, que la bibliothèque pourra appeler ultérieurement. En fait, vous avez déjà vu des API de ce type dans ce livre. Au [chapitre 2](#), nous avons utilisé le `actix-web` cadre pour écrire un simple serveur web. Une partie importante de ce programme était le routeur, qui ressemblait à ceci :

```

App:: new()
    .route( "/", web:: get().to(get_index))
    .route( "/gcd", web::post().to(post_gcd))

```

Le but du routeur est d'acheminer les requêtes entrantes d'Internet vers le morceau de code Rust qui gère ce type particulier de requête. Dans cet exemple, `get_index` et `post_gcd` étaient les noms des fonctions que nous avons déclarées ailleurs dans le programme, en utilisant le mot-clé `fn`. Mais nous aurions pu adopter des fermetures à la place, comme ceci :

```

App::: new()
    .route("/", web:: get()).to(|| {
        HttpResponse:: Ok()
            .content_type("text/html")
            .body("<title>GCD Calculator</title>...")
    })
    .route("/gcd", web:: post()).to(|form: web:: Form<GcdParameters>| {
        HttpResponse:: Ok()
            .content_type("text/html")
            .body(format!("The GCD of {} and {} is {}.", form.n, form.m, gcd(form.n, form.m)))
    })
})

```

C'est parce `actix-web` qu'il a été écrit pour accepter n'importe quel thread-safe Fn comme argument.

Comment pouvons-nous faire cela dans nos propres programmes? Essayons d'écrire notre propre routeur très simple à partir de zéro, sans utiliser de code de `actix-web`. Nous pouvons commencer par déclarer quelques types pour représenter les requêtes et les réponses HTTP :

```

struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

```

Désormais, le travail d'un routeur consiste simplement à stocker une table qui mappe les URL aux rappels afin que le bon rappel puisse être appelé à la demande. (Par souci de simplicité, nous n'autoriserons les utilisateurs qu'à créer des routes qui correspondent à une seule URL exacte.)

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// Create an empty router.
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }
}

```

```

/// Add a route to the router.
fn add_route(&mut self, url: &str, callback:C) {
    self.routes.insert(url.to_string(), callback);
}

```

Malheureusement, nous avons fait une erreur. L'avez-vous remarqué ?

Ce routeur fonctionne bien tant que nous n'y ajoutons qu'une seule route :

```

let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());

```

Cela se compile et s'exécute. Malheureusement, si nous ajoutons un autre itinéraire :

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

alors nous obtenons des erreurs:

```

error: mismatched types
|
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
|                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
|                                         expected closure, found a different closure
|
= note: expected type `#[closure@closures_bad_router.rs:40:27: 40:50]`
         found type `#[closure@closures_bad_router.rs:41:30: 41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait object

```

Notre erreur était dans la façon dont nous avons défini le `BasicRouter` type :

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes:HashMap<String, C>
}

```

Nous avons involontairement déclaré que chacun `BasicRouter` a un seul type de rappel `C` et que tous les rappels du `HashMap` sont de ce type. De retour dans "[Which to Use](#)", nous avons montré un `Salad` type qui avait le même problème :

```

    struct Salad<V: Vegetable> {
        veggies:Vec<V>
    }

```

La solution ici est la même que pour la salade : puisque nous voulons prendre en charge une variété de types, nous devons utiliser des boîtes et des objets de trait :

```

type BoxedCallback = Box<dyn Fn(&Request) ->Response>;

struct BasicRouter {
    routes:HashMap<String, BoxedCallback>
}

```

Chaque boîte peut contenir un type de fermeture différent, donc une seule `HashMap` peut contenir toutes sortes de rappels. Notez que le paramètre de type `C` a disparu.

Cela nécessite quelques ajustements dans les méthodes :

```

impl BasicRouter {
    // Create an empty router.
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }

    // Add a route to the router.
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}

```

NOTER

Notez les deux bornes `C` dans la signature de type pour `add_route`: un `Fn` trait particulier et la `'static` durée de vie. Rust nous fait ajouter cette `'static` borne. Sans cela, l'appel à `Box::new(callback)` serait une erreur, car il n'est pas sûr de stocker une fermeture si elle contient des références empruntées à des variables qui sont sur le point de sortir de la portée.

Enfin, notre routeur simple est prêt à gérer les requêtes entrantes :

```

impl BasicRouter {
    fn handle_request(&self, request: &Request) ->Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}

```

Au prix d'une certaine flexibilité, nous pourrions également écrire une version plus économique en espace de ce routeur qui, plutôt que de stocker des objets de trait, utilise *des pointeurs de fonction*, ou `fn` types. Ces types, tels que `fn(u32) -> u32`, agissent un peu comme des fermetures :

```

fn add_ten(x: u32) ->u32 {
    x + 10
}

let fn_ptr: fn(u32) ->u32 = add_ten;
let eleven = fn_ptr(1); //11

```

En fait, les fermetures qui ne capturent rien de leur environnement sont identiques aux pointeurs de fonction, car elles n'ont pas besoin de contenir d'informations supplémentaires sur les variables capturées. Si vous spécifiez le `fn` type approprié, soit dans une liaison, soit dans une signature de fonction, le compilateur se fera un plaisir de vous permettre de les utiliser de cette manière :

```

let closure_ptr: fn(u32) ->u32 = |x| x + 1;
let two = closure_ptr(1); // 2

```

Contrairement aux fermetures de capture, ces pointeurs de fonction n'occupent qu'un seul fichier `usize`.

Une table de routage contenant des pointeurs de fonction ressemblerait à ceci :

```

struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) ->Response>
}

```

Ici, le `HashMap` stocke un seul `usize` par `String`, et surtout, il n'y a pas de `Box`. Mis à part lui- `HashMap` même, il n'y a pas d'allocation dynamique du tout. Bien sûr, les méthodes doivent également être ajustées :

```

impl FnPointerRouter {
    // Create an empty router.
    fn new() -> FnPointerRouter {
        FnPointerRouter { routes: HashMap::new() }
    }

    // Add a route to the router.
    fn add_route(&mut self, url: &str, callback: fn(&Request) ->Response)
    {
        self.routes.insert(url.to_string(), callback);
    }
}

```

Comme indiqué dans la [Figure 14-1](#), les fermetures ont des types uniques parce que chacune capture différentes variables, donc entre autres choses, elles ont chacune une taille différente. S'ils ne capturent rien, cependant, il n'y a rien à stocker. En utilisant des `fn` pointeurs dans les fonctions qui prennent des rappels, vous pouvez restreindre un appelant à utiliser uniquement ces fermetures non capturantes, gagnant une certaine performance et flexibilité dans le code en utilisant des rappels au détriment de la flexibilité pour les utilisateurs de votre API.

Utilisation efficace des fermetures

Comme nous l'avons vu, les fermetures de Rust sont différents des fermetures dans la plupart des autres langues. La plus grande différence est que dans les langages avec GC, vous pouvez utiliser des variables locales dans une fermeture sans avoir à penser aux durées de vie ou à la propriété. Sans GC, les choses sont différentes. Certains modèles de conception courants en Java, C # et JavaScript ne fonctionneront pas dans Rust sans modifications.

Par exemple, prenez la conception Modèle-Vue-Contrôleur modèle (MVC en abrégé), illustré à la [Figure 14-3](#). Pour chaque élément d'une interface utilisateur, un framework MVC crée trois objets : un *modèle* représentant l'état de cet élément d'interface utilisateur, une *vue* responsable de son apparence et un *contrôleur* qui gère l'interaction de l'utilisateur. D'innombrables variantes de MVC ont été implémentées au fil des ans, mais l'idée générale est que trois objets répartissent d'une manière ou d'une autre les responsabilités de l'interface utilisateur.

Voici le problème. En règle générale, chaque objet a une référence à l'un ou aux deux autres, directement ou via un rappel, comme illustré à la [figure 14-3](#). Chaque fois que quelque chose arrive à l'un des objets, il en in-

forme les autres, donc tout est mis à jour rapidement. La question de savoir quel objet « possède » les autres ne se pose jamais.

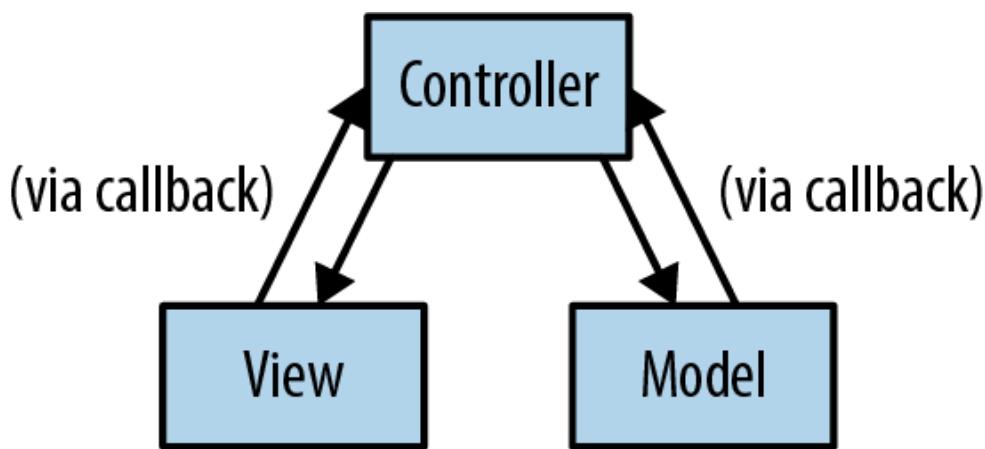


Illustration 14-3. Le modèle de conception Modèle-Vue-Contrôleur

Vous ne pouvez pas implémenter ce modèle dans Rust sans apporter quelques modifications. La propriété doit être rendue explicite et les cycles de référence doivent être éliminés. Le modèle et le contrôleur ne peuvent pas avoir de références directes l'un à l'autre.

Le pari radical de Rust est qu'il existe de bonnes conceptions alternatives. Parfois, vous pouvez résoudre un problème de propriété et de durée de vie des fermetures en faisant en sorte que chaque fermeture reçoive les références dont elle a besoin comme arguments. Parfois, vous pouvez attribuer un numéro à chaque élément du système et faire circuler les numéros au lieu des références. Ou vous pouvez implémenter l'une des nombreuses variantes de MVC où les objets n'ont pas tous des références les uns aux autres. Ou modélisez votre boîte à outils d'après un système non MVC avec un flux de données unidirectionnel, comme Flux de Facebookarchitecture, illustrée à la [Figure 14-4](#).

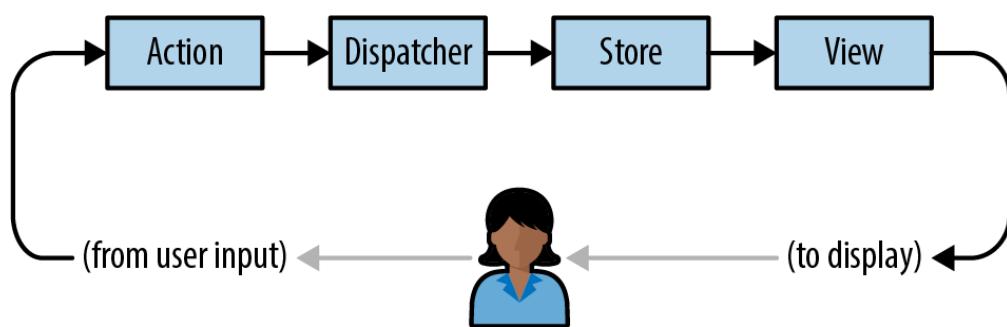


Illustration 14-4. L'architecture Flux, une alternative à MVC

En bref, si vous essayez d'utiliser les fermetures Rust pour créer une «mer d'objets», vous allez avoir du mal. Mais il existe des alternatives. Dans ce cas, il semble que le génie logiciel en tant que discipline gravite déjà de toute façon vers les alternatives, car elles sont plus simples.

Dans le chapitre suivant, nous passons à un sujet où les fermetures brillent vraiment. Nous allons écrire une sorte de code qui tire pleinement parti de la concision, de la rapidité et de l'efficacité des fermetures Rust et qui est amusant à écrire, facile à lire et éminemment pratique. À suivre : les itérateurs Rust.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 15. Itérateurs

C'était la fin d'une très longue journée.

—Phil

Un *itérateur* est une valeur qui produit une séquence de valeurs, généralement sur laquelle une boucle doit fonctionner. La bibliothèque standard de Rust fournit des itérateurs qui traversent des vecteurs, des chaînes, des tables de hachage et d'autres collections, mais aussi des itérateurs pour produire des lignes de texte à partir d'un flux d'entrée, des connexions arrivant sur un serveur réseau, des valeurs reçues d'autres threads sur un canal de communication, etc. sur. Et bien sûr, vous pouvez implémenter des itérateurs à vos propres fins. La boucle de Rust `for` fournit une syntaxe naturelle pour l'utilisation des itérateurs, mais les itérateurs eux-mêmes fournissent également un riche ensemble de méthodes pour mapper, filtrer, joindre, collecter, etc.

Les itérateurs de Rust sont flexibles, expressifs et efficaces. Considérez la fonction suivante, qui renvoie la somme des premiers `n` entiers positifs (souvent appelé le *n*ème nombre de triangle):

```
fn triangle(n: i32) ->i32 {
    let mut sum = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

L'expression `1..=n` est une `RangeInclusive<i32>` valeur. A `RangeInclusive<i32>` est un itérateur qui produit les entiers de sa valeur de départ à sa valeur de fin (toutes deux incluses), vous pouvez donc l'utiliser comme opérande de la `for` boucle pour additionner les valeurs de `1` à `n`.

Mais les itérateurs ont aussi une `fold` méthode, que vous pouvez utiliser dans la définition équivalente :

```
fn triangle(n: i32) ->i32 {
    (1..=n).fold(0, |sum, item| sum + item)
}
```

Starting with 0 as the running total, `fold` takes each value that `1..=n` produces and applies the closure `|sum, item| sum + item` to the running total and the value. The closure's return value is taken as the new running total. The last value it returns is what `fold` itself returns—in this case, the total of the entire sequence. This may look strange if you're used to `for` and `while` loops, but once you've gotten used to it, `fold` is a legible and concise alternative.

C'est un tarif assez standard pour les langages de programmation fonctionnels, qui privilégient l'expressivité. Mais les itérateurs de Rust ont été soigneusement conçus pour garantir que le compilateur puisse également les traduire en un excellent code machine. Dans une version de version de la deuxième définition présentée précédemment, Rust connaît la définition de `fold` et l'intègre dans `triangle`. Ensuite, la fermeture `|sum, item| sum + item` est intégrée à cela. Enfin, Rust examine le code combiné et reconnaît qu'il existe un moyen plus simple d'additionner les nombres de `un` à `n`: la somme est toujours égale à `n * (n+1) / 2`. Rust traduit le corps entier de `triangle`, boucle, fermeture, et tout, en une seule instruction de multiplication et quelques autres bits d'arithmétique.

Il se trouve que cet exemple implique une arithmétique simple, mais les itérateurs fonctionnent également bien lorsqu'ils sont utilisés de manière intensive. Ils sont un autre exemple de Rust fournissant des abstractions flexibles qui imposent peu ou pas de surcharge lors d'une utilisation typique.

Dans ce chapitre, nous expliquerons :

- Les traits `Iterator` et `IntoIterator`, qui sont à la base des itérateurs de Rust
- Les trois étapes d'un pipeline d'itérateur typique: création d'un itérateur à partir d'une sorte de source de valeur ; adapter une sorte d'itérateur à une autre en sélectionnant ou en traitant des valeurs au fur et à mesure de leur passage ; puis en consommant les valeurs produites par l'itérateur
- Comment implémenter des itérateurs pour vos propres types

Il existe de nombreuses méthodes, vous pouvez donc parcourir une section une fois que vous avez compris l'idée générale. Mais les itérateurs sont très courants dans Rust idiomatique, et il est essentiel de se familiariser avec les outils qui les accompagnent pour maîtriser le langage.

Les traits `Iterator` et `IntoIterator`

Un itérateur est une valeur qui implémente le

`std::iter::Iterator` trait :

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ... // many default methods
}
```

`Item` est le type de valeur produit par l'itérateur. La `next` méthode renvoie soit `Some(v)`, où `v` est la valeur suivante de l'itérateur, soit renvoie `None` pour indiquer la fin de la séquence. Ici, nous avons omis `Iterator` les nombreuses méthodes par défaut de; nous les aborderons individuellement dans le reste de ce chapitre.

S'il existe un moyen naturel d'itérer sur un type, ce type peut implémenter `std::iter::IntoIterator`, dont la `into_iter` méthode prend une valeur et renvoie un itérateur dessus :

```
trait IntoIterator where Self:: IntoIter: Iterator<Item=Self:: Item> {
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

`IntoIter` est le genre de la valeur de l'itérateur lui-même, et `Item` est le type de valeur qu'il produit. Nous appelons tout type qui implémente `IntoIterator` un *itérable*, car c'est quelque chose que vous pouvez répéter si vous le demandez.

La boucle de Rust `for` rassemble bien toutes ces parties. Pour itérer sur les éléments d'un vecteur, vous pouvez écrire :

```
println!("There's:");
let v = vec![ "antimony", "arsenic", "aluminum", "selenium" ];

for element in &v {
    println!("{}", element);
}
```

Sous le capot, chaque `for` boucle n'est qu'un raccourci pour les appels `IntoIterator` et `Iterator` les méthodes :

```

let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}

```

La `for` boucle utilise `IntoIterator::into_iter` pour convertir son opérande `&v` en un itérateur, puis appelle `Iterator::next` à plusieurs reprises. Chaque fois que renvoie `Some(element)`, la `for` boucle exécute son corps ; et s'il retourne `None`, la boucle se termine.

En gardant cet exemple à l'esprit, voici une terminologie pour les itérateurs :

- Comme nous l'avons dit, un *itérateur* est tout type qui implémente `Iterator`.
- Un *itérable* est tout type qui implémente `IntoIterator` : vous pouvez obtenir un itérateur dessus en appelant sa `into_iter` méthode. La référence vectorielle `&v` est l'itérable dans ce cas.
- Un itérateur *produit des valeurs*.
- Les valeurs produites par un itérateur sont des *items*. Ici, les éléments sont "antimony", "arsenic", etc.
- Le code qui reçoit les éléments produits par un itérateur est le *consommateur*. Dans cet exemple, la `for` boucle est le consommateur.

Bien qu'une `for` boucle appelle toujours `into_iter` son opérande, vous pouvez également passer `for` directement les itérateurs aux boucles ; cela se produit lorsque vous bouchez sur un `Range`, par exemple. Tous les itérateurs implémentent automatiquement `IntoIterator`, avec une `into_iter` méthode qui renvoie simplement l'itérateur.

Si vous appelez à nouveau la méthode d'un itérateur `next` après son retour `None`, le `Iterator` trait ne spécifie pas ce qu'il doit faire. La plupart des itérateurs reviendront à `None` nouveau, mais pas tous. (Si cela cause des problèmes, l' `fuse` adaptateur couvert de ["fusible"](#) peut aider.)

Création d'itérateurs

La rouilleLa documentation de la bibliothèque standard explique en détail le type d'itérateurs que chaque type fournit, mais la bibliothèque suit certaines conventions générales pour vous aider à vous orienter et à trouver ce dont vous avez besoin.

Méthodes `iter` et `iter_mut`

La plupart des collectestypes fournissent `iter` et `iter_mut` méthodes qui renvoient les itérateurs naturels sur le type, produisant une référence partagée ou modifiable à chaque élément. Les tranches de tableau aiment `&[T]` et `&mut [T]` ont `iter` et `iter_mut` les méthodes aussi. Ces méthodes sont le moyen le plus courant d'obtenir un itérateur, si vous ne laissez pas une `for` boucle s'en occuper pour vous :

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

Le type d'élément de cet itérateur est `&i32` : chaque appel à `next` produit une référence à l'élément suivant, jusqu'à ce que nous atteignions la fin du vecteur.

Chaque type est libre de mise en œuvre `iter` et `iter_mut` de la manière la plus logique pour son objectif. La `iter` méthode sur les `std::path::Path` retourne un itérateur qui produit un composant de chemin à la fois :

```
use std::ffi::OsStr;
use std::path::Path;

let path = Path::new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr::new("C:")));
assert_eq!(iterator.next(), Some(OsStr::new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...

```

Le type d'élément de cet itérateur est `&std::ffi::OsStr`, une tranche empruntée d'une chaîne du type accepté par les appels du système d'exploitation.

S'il existe plusieurs façons courantes d'itérer sur un type, le type fournit généralement des méthodes spécifiques pour chaque type de parcours, car une `iter` méthode simple serait ambiguë. Par exemple, il n'y a pas `iter` de méthode sur le `&str` type tranche de chaîne. Au lieu de cela, si `s` est un `&str`, puis `s.bytes()` renvoie un itérateur qui produit chaque

octet de `s`, alors `s.chars()` qu'il interprète le contenu comme UTF-8 et produit chaque caractère Unicode.

Implémentations IntoIterator

Lorsqu'un genre implemente `IntoIterator`, vous pouvez appeler sa `into_iter` méthode vous-même, comme le `for` ferait une boucle :

```
// You should usually use HashSet, but its iteration order is
// nondeterministic, so BTreeSet works better in examples.
use std::collections::BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());

let mut it = favorites.into_iter();
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With Diamonds".to_string()));
assert_eq!(it.next(), None);
```

La plupart des collections fournissent en fait plusieurs implémentations de `IntoIterator`, pour les références partagées(`&T`), les références mutables(`&mut T`) et les déplacements(`T`) :

- Étant donné une *référence partagée* à la collection, `into_iter` renvoie un itérateur qui produit des références partagées à ses éléments. Par exemple, dans le code précédent, `(&favorites).into_iter()` renverrait un itérateur dont le `Item` type est `&String`.
- Étant donné une *référence mutable* à la collection, `into_iter` renvoie un itérateur qui produit des références mutables aux éléments. Par exemple, si `vector` est certains `Vec<String>`, l'appel `(&mut vector).into_iter()` renvoie un itérateur dont le `Item` type est `&mut String`.
- Lors du passage de la collection *par valeur*, `into_iter` renvoie un itérateur qui s'approprie la collection et renvoie les éléments par valeur ; la propriété des articles passe de la collection au consommateur, et la collection d'origine est consommée dans le processus. Par exemple, l'appel `favorites.into_iter()` dans le code précédent renvoie un itérateur qui produit chaque chaîne par valeur ; le consommateur reçoit la propriété de chaque chaîne. Lorsque l'itérateur est supprimé, tous les éléments restants dans le `BTreeSet` sont également supprimés et l'enveloppe désormais vide de l'ensemble est supprimée.

Depuis une `for` boucles'applique `IntoIterator::into_iter` à son opérande, ces trois implémentations créent les idiomes suivants pour itérer sur des références partagées ou modifiables à une collection, ou consommer la collection et s'approprier ses éléments :

```
for element in &collection { ... }
for element in &mut collection { ... }
for element in collection { ... }
```

Chacun de ces résultats entraîne simplement un appel à l'une des `IntoIterator` implémentations répertoriées ici.

Tous les types ne fournissent pas les trois implémentations. Par exemple, `HashSet`, `BTreeSet` et `BinaryHeap` ne s'implémentent pas `IntoIterator` sur des références mutables, car la modification de leurs éléments violerait probablement les invariants du type : la valeur modifiée pourrait avoir une valeur de hachage différente, ou être ordonnée différemment par rapport à ses voisins, donc la modifier laisserait il est mal placé. D'autres types prennent en charge la mutation, mais seulement partiellement. Par exemple, `HashMap` et `BTreeMap` produisent une référence mutable aux valeurs de leurs entrées, mais uniquement des références partagées à leurs clés, pour des raisons similaires à celles données précédemment.

Le principe général est que l'itération doit être efficace et prévisible, donc plutôt que de fournir des implémentations qui sont coûteuses ou qui pourraient présenter un comportement surprenant (par exemple, resasser les `HashSet` entrées modifiées et éventuellement les rencontrer à nouveau plus tard dans l'itération), Rust les omet entièrement.

Tranchesmettre en œuvre deux des trois `IntoIterator` variantes ; puisqu'ils ne sont pas propriétaires de leurs éléments, il n'y a pas de cas « par valeur ». Au lieu de cela, `into_iter` for `&[T]` et `&mut [T]` renvoie un itérateur qui produit des références partagées et modifiables aux éléments. Si vous imaginez le type de tranche sous-jacent `[T]` comme une collection quelconque, cela s'intègre parfaitement dans le modèle global.

Vous avez peut-être remarqué que les deux premières `IntoIterator` variantes, pour les références partagées et mutables, sont équivalentes à appeler `iter` ou `iter_mut` sur le référent. Pourquoi Rust fournit-il les deux ?

`IntoIterator` c'est ce qui fait les `for` bouclestravail, donc c'est évidemment nécessaire. Mais lorsque vous n'utilisez pas de `for` boucle, il est

plus clair d'écrire `favorites.iter()` que
`(&favorites).into_iter()`. L'itération par référence partagée est quelque chose dont vous aurez fréquemment besoin, donc `iter` et `iter_mut` sont toujours précieux pour leur ergonomie.

`IntoIterator` peut également être utile en générique code : vous pouvez utiliser une limite comme `T: IntoIterator` pour restreindre la variable de type `T` aux types qui peuvent être itérés. Ou, vous pouvez écrire `T: IntoIterator<Item=U>` pour exiger davantage que l'itération produise un type particulier `U`. Par exemple, cette fonction vide les valeurs de tout itérable dont les éléments sont imprimables au "`{:?}`" format :

```
use std::fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U: Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

Vous ne pouvez pas écrire cette fonction générique en utilisant `iter` and `iter_mut`, car ce ne sont pas des méthodes d'aucun trait : la plupart des types itérables ont simplement des méthodes portant ces noms.

from_fn et successeurs

Un simple et la manière générale de produire une séquence de valeurs est de fournir une fermeture qui les renvoie.

Étant donné une fonction renvoyant `Option<T>`,
`std::iter::from_fn` renvoie un itérateur qui appelle simplement la fonction pour produire ses éléments. Par exemple:

```
use rand::random; // In Cargo.toml dependencies: rand = "0.7"
use std::iter::from_fn;

// Generate the lengths of 1000 random line segments whose endpoints
// are uniformly distributed across the interval [0, 1]. (This isn't a
// distribution you're going to find in the `rand_distr` crate, but
// it's easy to make yourself.)
let lengths: Vec<f64> =
    from_fn(|| Some((random::f64() - random::f64()).abs()))
```

```
.take(1000)
.collect();
```

Cela appelle `from_fn` à faire un itérateur produisant des nombres aléatoires. Puisque l'itérateur retourne toujours `Some`, la séquence ne se termine jamais, mais nous l'appelons `take(1000)` pour la limiter aux 1 000 premiers éléments. Construit ensuite `collect` le vecteur à partir de l'itération résultante. C'est un moyen efficace de construire des vecteurs initialisés ; nous expliquons pourquoi dans « [Construire des collections : collect et FromIterator](#) », plus loin dans ce chapitre.

Si chaque élément dépend du précédent, la

`std::iter::successors` fonction fonctionne bien. Vous fournissez un élément initial et une fonction qui prend un élément et renvoie un élément `Option` suivant. S'il renvoie `None`, l'itération se termine. Par exemple, voici une autre façon d'écrire la `escape_time` fonction de notre set plotter de Mandelbrot au [chapitre 2](#) :

```
use num:: Complex;
use std:: iter::successors;

fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let zero = Complex { re: 0.0, im: 0.0 };
    successors(Some(zero), |&z| { Some(z * z + c) })
        .take(limit)
        .enumerate()
        .find(|(_i, z)| z.norm_sqr() > 4.0)
        .map(|(i, _z)| i)
}
```

En commençant par zéro, l'`successors` appel produit une séquence de points sur le plan complexe en élevant à plusieurs reprises le dernier point au carré et en ajoutant le paramètre `c`. Lors du traçage de l'ensemble de Mandelbrot, nous voulons voir si cette séquence orbite près de l'origine pour toujours ou s'envole vers l'infini. L'appel `take(limit)` établit une limite sur la durée pendant laquelle nous poursuivrons la séquence et `enumerate` numérote chaque point, transformant chaque point `z` en un tuple `(i, z)`. Nous `find` cherchons le premier point qui s'éloigne suffisamment de l'origine pour s'échapper. La `find` méthode renvoie un `Option: Some((i, z))` s'il existe, ou `None` sinon. L'appel à `Option::map` se transforme `Some((i, z))` en `Some(i)`, mais retourne `None` inchangé : c'est exactement la valeur de retour que nous voulons.

Les deux `from_fn` et `successors` acceptent les `FnMut` fermetures, afin que vos fermetures puissent capturer et modifier les variables des éten-

dues environnantes. Par exemple, cette `fibonacci` fonction utilise une move fermeture pour capturer une variable et l'utiliser comme état d'exécution :

```
fn fibonacci() -> impl Iterator<Item=usize> {
    let mut state = (0, 1);
    std::iter::from_fn(move || {
        state = (state.1, state.0 + state.1);
        Some(state.0)
    })
}

assert_eq!(fibonacci().take(8).collect::<Vec<_>>(),
           vec![1, 1, 2, 3, 5, 8, 13, 21]);
```

Une note de prudence : les méthodes `from_fn` et `successors` sont suffisamment flexibles pour que vous puissiez transformer à peu près n'importe quelle utilisation d'itérateurs en un seul appel à l'un ou à l'autre, en passant des fermetures complexes pour obtenir le comportement dont vous avez besoin. Mais cela néglige l'opportunité qu'offrent les itérateurs de clarifier la façon dont les données circulent dans le calcul et d'utiliser des noms standard pour les modèles communs. Assurez-vous de vous être familiarisé avec les autres méthodes d'itération de ce chapitre avant de vous appuyer sur ces deux ; il y a souvent des façons plus agréables de faire le travail.

Méthodes de vidange

De nombreux types de collections fournissent une `drain` méthode qui prend une référence mutable à la collection et renvoie un itérateur qui transmet la propriété de chaque élément au consommateur. Cependant, contrairement à la `into_iter()` méthode, qui prend la collection par valeur et la consomme, `drain` emprunte simplement une référence mutable à la collection, et lorsque l'itérateur est supprimé, il supprime tous les éléments restants de la collection et la laisse vide.

Sur les types qui peuvent être indexés par une plage, comme `String`s, `vectors` et `VecDeque`s, la `drain` méthode prend une plage d'éléments à supprimer, plutôt que de vider toute la séquence :

```
let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

Si vous avez besoin de vider toute la séquence, utilisez la plage complète, `...`, comme argument.

Autres sources d'itérateurs

Les sections précédentes concernent principalement les types de collection comme les vecteurs et `HashMap`, mais il existe de nombreux autres types dans la norme bibliothèque prenant en charge l'itération. [Le tableau 15-1](#) résume les plus intéressantes, mais il y en a bien d'autres. Nous couvrons certaines de ces méthodes plus en détail dans les chapitres consacrés aux types spécifiques (à savoir, les chapitres [16](#), [17](#) et [18](#)).

Tableau 15-1. Autres itérateurs de la bibliothèque standard

Type ou caractère	Expression	Remarques
<code>std::op::Range</code>	<code>1 .. 10</code>	Les points de terminaison doivent être de type entier pour être itérables. La plage inclut la valeur de début et exclut la valeur de fin.
	<code>(1 .. 10).step_by(2)</code>	Produit 1, 3, 5, 7, 9.
<code>std::op::RangeFrom</code>	<code>1 ..</code>	Itération illimitée. Début doit être un entier. Peut paniquer ou déborder si la valeur atteint la limite du type.
<code>std::op::RangeInclusive</code>	<code>1 ..=10</code>	Comme <code>Range</code> , mais inclut la valeur finale.
<code>Option<T></code>	<code>Some(10).iter()</code>	Se comporte comme un vecteur dont la longueur est soit 0 (<code>None</code>) soit 1 (<code>Some(v)</code>).
<code>Result<T, E></code>	<code>Ok("blah").iter()</code>	Similaire à <code>Option</code> , produisant des <code>Ok</code> valeurs.
<code>Vec<T>, &[T]</code>	<code>v.window(16)</code>	Produit chaque tranche contiguë de la longueur donnée, de gauche à droite. Les fenêtres se chevauchent.
	<code>v.chunks(16)</code>	Produit des tranches contiguës sans chevauchement de la longueur donnée, de gauche à droite.
	<code>v.chunks_mut(1024)</code>	Comme <code>chunks</code> , mais les tranches sont modifiables.

Type ou caractère	Expression	Remarques
	v.split(byte byt e & 1 != 0)	Produit des tranches séparées par des éléments qui correspondent au prédicat donné.
	v.split_ mut(...)	Comme ci-dessus, mais produit des tranches modifiables.
	v.rsplit (...)	Comme <code>split</code> , mais produit des tranches de droite à gauche.
	v.splitn (n, ...)	Comme <code>split</code> , mais produit au plus des <code>n</code> tranches.
String , &str	s.bytes ()	Produit les octets du formulaire UTF-8.
	s.chars ()	Produit le <code>char s</code> représenté par UTF-8.
	s.split_w hitespace ()	Divise la chaîne par des espaces blancs et produit des tranches de caractères sans espace.
	s.lines ()	Produit des tranches des lignes de la chaîne.
	s.split (' / ')	Divise la chaîne sur un modèle donné, produisant les tranches entre les correspondances. Les modèles peuvent être de plusieurs choses : caractères, chaînes, fermetures.
	s.matches (char::is _numeric)	Produit des tranches correspondant au motif donné.

Type ou caractère	Expression	Remarques
std::co llection s::Hash Map , std::co llection s::BTre eMap	map.keys (), map.valu es() map.value s_mut()	Produit des références partagées aux clés ou aux valeurs de la carte. Produit des références mutables aux valeurs des entrées.
std::co llection s::Hash Set , std::co llection s::BTre eSet	set1.unio n(set2) set1.inte rsection (set2)	Produit des références partagées aux éléments d'union de set1 et set2 . Produit des références partagées aux éléments d'intersection de set1 et set2 .
std::sy nc::mps c::Rece iver	recv.ite r() stream.by tes()	Produit des valeurs envoyées depuis un autre thread sur le correspondant Send er . Produit des octets à partir d'un flux d'E/S.
	stream.ch ars()	Analyse le flux au format UTF-8 et produit char s.
std::i o::Read ead	bufstrea m.lines()	Analyse le flux en UTF-8, produit des lignes en String s.
	bufstrea m.split (0)	Divise le flux sur un octet donné, produit des vec<u8> tampons inter-octets.

Type ou caractère	Expression	Remarques
std::f s::Read Dir	std::fs:: read_dir (path)	Produit des entrées de répertoire.
std::ne t::TcpLi stener	listener. incoming ()	Produit des connexions réseau entrantes.
Fonctions gratuites	std::ite r::empty ()	Retourne None immédiatement.
	std::ite r::once (5)	Produit la valeur donnée puis se termine.
	std::ite r::repeat ("#9")	Produit la valeur donnée pour toujours.

Adaptateurs d'itérateur

Une fois que vous avez un itérateur en main, le `Iterator` trait fournit une large sélection de *méthodes d'adaptation*, ou simplement des *adaptateurs*, qui consomment un itérateur et en construisent un nouveau avec des comportements utiles. Pour voir comment fonctionnent les adaptateurs, nous commencerons par deux des adaptateurs les plus populaires, `map` et `filter`. Ensuite, nous couvrirons le reste de la boîte à outils de l'adaptateur, couvrant presque toutes les façons imaginables de créer des séquences de valeurs à partir d'autres séquences : troncature, saut, combinaison, inversion, concaténation, répétition, etc.

carte et filtre

L'adaptateur `Iterator` du trait `map` vous permet de transformer un itérateur en appliquant une fermeture à ses éléments. L'`filter` adaptateur vous permet de filtrer les éléments d'un itérateur, en utilisant une fermeture pour décider lesquels conserver et lesquels supprimer.

Par exemple, supposons que vous itérez sur des lignes de texte et que vous souhaitez omettre les espaces de début et de fin de chaque ligne. La `str::trim` méthode de la bibliothèque standard supprime les espaces blancs de début et de fin d'un seul `&str`, renvoyant un nouveau, coupé, `&str` qui emprunte à l'original. Vous pouvez utiliser l' `map` adaptateur pour appliquer `str::trim` à chaque ligne de l'itérateur :

```
let text = "  ponies  \n  giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

L' `text.lines()` appel renvoie un itérateur qui produit les lignes de la chaîne. L'appel `map` à cet itérateur renvoie un deuxième itérateur qui s'applique `str::trim` à chaque ligne et produit les résultats sous forme d'éléments. Enfin, `collect` rassemble ces éléments dans un vecteur.

L'itérateur qui `map` revient est, bien sûr, lui-même un candidat pour une adaptation ultérieure. Si vous souhaitez exclure les iguanes du résultat, vous pouvez écrire ce qui suit :

```
let text = "  ponies  \n  giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

Ici, `filter` renvoie un troisième itérateur qui produit uniquement les éléments de l' `map` itérateur pour lesquels la fermeture `|s| *s != "iguanas"` renvoie `true`. Une chaîne d'adaptateurs d'itérateurs est comme un pipeline dans le shell Unix : chaque adaptateur a un seul objectif, et il est clair comment la séquence est transformée lorsque l'on lit de gauche à droite.

Les signatures de ces adaptateurs sont les suivantes :

```
fn map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self:: Item) ->B;

fn filter<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Dans la bibliothèque standard, `map` et `filter` renvoient en fait des types opaques spécifiques `struct` nommés `std::iter::Map` et `std::iter::Filter`. Cependant, le simple fait de voir leurs noms n'est pas très informatif, donc dans ce livre, nous allons simplement écrire à la `-> impl Iterator<Item=...>` place, car cela nous dit ce que nous voulons vraiment savoir : la méthode renvoie un `Iterator` qui produit des éléments du type donné.

Étant donné que la plupart des adaptateurs prennent `self` par valeur, ils `Self` doivent être `Sized` (ce que sont tous les itérateurs courants).

Un `map` itérateur transmet chaque élément à sa fermeture par valeur et, à son tour, transmet la propriété du résultat de la fermeture à son consommateur. Un `filter` itérateur passe chaque élément à sa fermeture par référence partagée, conservant la propriété au cas où l'élément serait sélectionné pour être transmis à son consommateur. C'est pourquoi l'exemple doit déréférencer `s` pour le comparer avec "iguanas" : le `filter` type de l'élément de l'itérateur est `&str`, donc le type de l'argument de la fermeture `s` est `&&str`.

Il y a deux points importants à noter à propos des adaptateurs d'itérateur.

Tout d'abord, le simple fait d'appeler un adaptateur sur un itérateur ne consomme aucun élément ; il renvoie simplement un nouvel itérateur, prêt à produire ses propres éléments en puisant dans le premier itérateur si nécessaire. Dans une chaîne d'adaptateurs, la seule façon d'effectuer un travail est de faire appel `next` à l'itérateur final.

Ainsi, dans notre exemple précédent, l'appel de méthode `text.lines()` lui-même n'analyse aucune ligne de la chaîne ; il renvoie simplement un itérateur qui *analyserait* les lignes si demandé. De même, `map` et `filter` renvoient simplement de nouveaux itérateurs qui mapperaient ou filtreraient si demandé. Aucun travail n'a lieu tant que l'itérateur n'a pas `collect` commencé à être appelé. `.next filter`

Ce point est particulièrement important si vous utilisez des adaptateurs qui ont des effets secondaires. Par exemple, ce code n'imprime rien du tout :

```
[ "earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

L'`iter` appel renvoie un itérateur sur les éléments du tableau et l'`map` appel renvoie un deuxième itérateur qui applique la fermeture à

chaque valeur produite par le premier. Mais il n'y a rien ici qui exige réellement une valeur de toute la chaîne, donc aucune `next` méthode ne fonctionne jamais. En fait, Rust vous avertira à ce sujet :

```
warning: unused `std::iter::Map` that must be used
|
7 | /      ["earth", "water", "air", "fire"]
8 | |       .iter().map(|elt| println!("{}", elt));
| |
| = note: iterators are lazy and do nothing unless consumed
```

Le terme « paresseux » dans le message d'erreur n'est pas un terme désoobligeant ; c'est juste du jargon pour tout mécanisme qui retarde un calcul jusqu'à ce que sa valeur soit nécessaire. C'est la convention de Rust que les itérateurs doivent faire le travail minimum nécessaire pour satisfaire chaque appel à `next` ; dans l'exemple, il n'y a aucun appel de ce type, donc aucun travail n'a lieu.

Le deuxième point important est que les adaptateurs d'itérateur sont une abstraction sans surcharge. Étant donné que `map`, `filter` et leurs compagnons sont génériques, leur application à un itérateur spécialise leur code pour le type d'itérateur spécifique impliqué. Cela signifie que Rust dispose de suffisamment d'informations pour intégrer la méthode de chaque itérateur `next` dans son consommateur, puis traduire l'intégralité de l'arrangement en code machine en tant qu'unité. Ainsi, la chaîne `lines // map` d'`filter` itérateurs que nous avons montrée précédemment est aussi efficace que le code que vous écririez probablement à la main :

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

Le reste de cette section couvre les différents adaptateurs disponibles sur le `Iterator` trait.

filter_map et flat_map

L'`map` adaptateur convient dans les situations où chaque élément entrant produit un élément sortant. Mais que se passe-t-il si vous souhaitez sup-

primer certains éléments de l'itération au lieu de les traiter ou remplacer des éléments uniques par zéro ou plusieurs éléments ? Les adaptateurs `filter_map` et `flat_map` vous accorder cette flexibilité.

L' `filter_map` adaptateur est similaire à `map` sauf qu'il permet à sa fermeture de transformer l'élément en un nouvel élément (comme le `map` fait) ou de supprimer l'élément de l'itération. Ainsi, c'est un peu comme une combinaison de `filter` et `map`. Sa signature est la suivante :

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self:: Item) -> Option<B>;
```

C'est la même chose que `map` la signature de , sauf qu'ici la fermeture renvoie `Option` , pas simplement `B` . Lorsque la fermeture renvoie `None` , l'élément est supprimé de l'itération ; lorsqu'il retourne `Some(b)` , alors `b` est l'élément suivant produit par l' `filter_map` itérateur.

Par exemple, supposons que vous souhaitez rechercher dans une chaîne des mots séparés par des espaces pouvant être analysés comme des nombres, et traiter les nombres en supprimant les autres mots. Tu peux écrire:

```
use std:: str::FromStr;

let text = "1\nfrond .25 289\n3.1415 estuary\n";
for number in text
    .split_whitespace()
    .filter_map(|w| f64::from_str(w).ok())
{
    println!("{:4.2}", number.sqrt());
}
```

Cela imprime ce qui suit :

```
1.00
0.50
17.00
1.77
```

La fermeture donnée à `filter_map` essaie d'analyser chaque tranche séparée par des espaces en utilisant `f64::from_str` . Cela renvoie un `Result<f64, ParseFloatError>` , qui `.ok()` se transforme en un `Option<f64>` : une erreur d'analyse devient `None` , tandis qu'un résultat

d'analyse réussi devient `Some(v)`. L' `filter_map` itérateur supprime toutes les `None` valeurs et produit la valeur `v` pour chacune `Some(v)`.

Mais quel est l'intérêt de fusionner `map` en `filter` une seule opération comme celle-ci, au lieu d'utiliser directement ces adaptateurs ? L'`filter_map` adaptateur montre sa valeur dans des situations comme celle qui vient d'être montrée, lorsque la meilleure façon de décider d'inclure ou non l'élément dans l'itération est d'essayer réellement de le traiter. Vous pouvez faire la même chose avec seulement `filter` et `map`, mais c'est un peu disgracieux :

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

Vous pouvez considérer l'`flat_map` adaptateur comme continuant dans la même veine que `map` et `filter_map`, sauf que maintenant la fermeture peut renvoyer non seulement un élément (comme avec `map`) ou zéro ou un élément (comme avec `filter_map`), mais une séquence de n'importe quel nombre d'éléments. L'`flat_map` itérateur produit la concaténation des séquences renvoyées par la fermeture.

La signature de `flat_map` est montrée ici :

```
fn flat_map<U, F>(self, f: F) -> impl Iterator<Item=U:: Item>
    where F: FnMut(Self:: Item) -> U, U: IntoIterator;
```

La fermeture passée à `flat_map` doit renvoyer un itérable, mais n'importe quel type d'itérable fera l'affaire.¹

Par exemple, supposons que nous ayons une table mappant les pays à leurs principales villes. Étant donné une liste de pays, comment pouvons-nous itérer sur leurs principales villes ?

```
use std::collections::HashMap;

let mut major_cities = HashMap::new();
major_cities.insert("Japan", vec!["Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec!["Portland", "Nashville"]);
major_cities.insert("Brazil", vec!["São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec!["Amsterdam", "Utrecht"]);

let countries = ["Japan", "Brazil", "Kenya"];
```

```

for &city in countries.iter().flat_map(|country| &major_cities[country]) {
    println!("{}", city);
}

```

Cela imprime ce qui suit :

```

Tokyo
Kyoto
São Paulo
Brasília
Nairobi
Mombasa

```

Une façon de voir cela serait de dire que, pour chaque pays, nous récupérons le vecteur de ses villes, concaténons tous les vecteurs ensemble en une seule séquence et imprimons cela.

Mais rappelez-vous que les itérateurs sont paresseux : seuls les `for` appels de la boucle à la méthode de l' `flat_map` itérateur `next` provoquent l'exécution du travail. La séquence concaténée complète n'est jamais construite en mémoire. Au lieu de cela, ce que nous avons ici est une petite machine d'état qui tire de l'itérateur de ville, un élément à la fois, jusqu'à ce qu'il soit épuisé, et produit alors seulement un nouvel itérateur de ville pour le pays suivant. L'effet est celui d'une boucle imbriquée, mais emballée pour être utilisée comme itérateur.

aplatisir

L' `flatten` adaptateur concatène les éléments d'un itérateur, en supposant que chaque élément est lui-même un itérable :

```

use std::collections::BTreeMap;

// A table mapping cities to their parks: each value is a vector.
let mut parks = BTreeMap::new();
parks.insert("Portland", vec!["Mt. Tabor Park", "Forest Park"]);
parks.insert("Kyoto",     vec![ "Tadasu-no-Mori Forest", "Maruyama Koen"]);
parks.insert("Nashville", vec![ "Percy Warner Park", "Dragon Park"]);

// Build a vector of all parks. `values` gives us an iterator producing
// vectors, and then `flatten` produces each vector's elements in turn.
let all_parks:Vec<_> = parks.values().flatten().cloned().collect();

assert_eq!(all_parks,

```

```
vec![ "Tadasu-no-Mori Forest", "Maruyama Koen", "Percy Warner Pa  
"Dragon Park", "Mt. Tabor Park", "Forest Park"]);
```

Le nom « aplatisir » vient de l'image de l'aplatissement d'une structure à deux niveaux en une structure à un niveau : le `BTreeMap` et ses `Vec`s de noms sont aplatis en un itérateur produisant tous les noms.

La signature de `flatten` est la suivante :

```
fn flatten(self) -> impl Iterator<Item=Self:: Item:: Item>  
    where Self:: Item:IntoIterator;
```

En d'autres termes, les éléments de l'itérateur sous-jacent doivent eux-mêmes s'implémenter `IntoIterator` pour qu'il s'agisse effectivement d'une séquence de séquences. La `flatten` méthode renvoie ensuite un itérateur sur la concaténation de ces séquences. Bien sûr, cela se fait par-dessus, en dessinant un nouvel élément `self` uniquement lorsque nous avons fini d'itérer sur le dernier.

La `flatten` méthode est utilisée de quelques manières surprenantes. Si vous avez un `Vec<Option<...>>` et que vous souhaitez itérer uniquement sur les `Some` valeurs, `flatten` fonctionne à merveille:

```
assert_eq!(vec![None, Some("day"), None, Some("one")]  
    .into_iter()  
    .flatten()  
    .collect::<Vec<_>>(),  
    vec![ "day", "one"]);
```

Cela fonctionne parce que lui- `Option` même implémente `IntoIterator`, représentant une séquence de zéro ou un élément. Les `None` éléments n'apportent rien à l'itération, alors que chaque `Some` élément apporte une seule valeur. De même, vous pouvez utiliser `flatten` pour itérer sur les `Option<Vec<...>>` valeurs : `None` se comporte comme un vecteur vide.

`Result` implémente également `IntoIterator`, avec `Err` représentant une séquence vide, donc l'application `flatten` à un itérateur de `Result` valeurs extrait efficacement tous les `Err`s et les jette, ce qui entraîne un flux de valeurs de réussite non emballées. Nous ne recommandons pas d'ignorer les erreurs dans votre code, mais c'est une astuce que les gens utilisent lorsqu'ils pensent savoir ce qui se passe.

Vous pouvez vous retrouver à rechercher `flatten` ce dont vous avez réellement besoin `flat_map`. Par exemple, la méthode de la bibliothèque standard `str::to_uppercase`, qui convertit une chaîne en majuscule, fonctionne comme ceci :

```
fn to_uppercase(&self) -> String {
    self.chars()
        .map(char::to_uppercase)
        .flatten() // there's a better way
        .collect()
}
```

La raison pour laquelle `flatten` est nécessaire est que `ch.to_uppercase()` ne renvoie pas un seul caractère, mais un itérateur produisant un ou plusieurs caractères. Le mappage de chaque caractère à son équivalent majuscule donne un itérateur d'itérateurs de caractères, et le `flatten` s'occupe de les assembler tous en quelque chose que nous pouvons finalement `collect` transformer en un `String`.

Mais cette combinaison de `map` et `flatten` est si courante qu'elle `Iterator` fournit l'`flat_map` adaptateur pour ce cas précis. (En fait, `flat_map` a été ajouté à la bibliothèque standard avant `flatten`.) Ainsi, le code précédent pourrait à la place être écrit:

```
fn to_uppercase(&self) -> String {
    self.chars()
        .flat_map(char::to_uppercase)
        .collect()
}
```

prendre et prendre_pendant

Les `Iterator` traits `take` et `take_while` adaptateurs vous permet de terminer une itération après un certain nombre d'éléments ou lorsqu'une fermeture décide de couper les choses. Leurs signatures sont les suivantes :

```
fn take(self, n: usize) -> impl Iterator<Item=Self:: Item>
    where Self:Sized;

fn take_while<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Les deux s'approprient un itérateur et renvoient un nouvel itérateur qui transmet les éléments du premier, mettant éventuellement fin à la séquence plus tôt. L' `take` itérateur revient `None` après avoir produit au maximum `n` les éléments. L' `take_while` itérateur s'applique `predicate` à chaque élément et revient `None` à la place du premier élément pour lequel `predicate` renvoie `false` et à chaque appel ultérieur à `next`.

Par exemple, étant donné un message électronique avec une ligne vide séparant les en-têtes du corps du message, vous pouvez utiliser `take_while` pour parcourir uniquement les en-têtes :

```
let message = "To: jimb\r\n"
              From: superego <editor@oreilly.com>\r\n\
\r\n
Did you get any writing done today?\r\n\
When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

Rappelez-vous de "[String Literals](#)" que lorsqu'une ligne dans une chaîne se termine par une barre oblique inverse, Rust n'inclut pas l'indentation de la ligne suivante dans la chaîne, donc aucune des lignes de la chaîne n'a d'espace blanc au début. Cela signifie que la troisième ligne de `message` est vide. L' `take_while` adaptateur termine l'itération dès qu'il voit cette ligne vide, donc ce code n'imprime que les deux lignes :

```
To: jimb
From: superego <editor@oreilly.com>
```

sauter et sauter_pendant

Les `Iterator` traits `skip` et les `skip_while` méthodessont le complément de `take` et `take_while` : ils suppriment un certain nombre d'éléments depuis le début d'une itération, ou suppriment des éléments jusqu'à ce qu'une fermeture en trouve un acceptable, puis transmettent les éléments restants tels quels. Leurs signatures sont les suivantes :

```
fn skip(self, n: usize) -> impl Iterator<Item=Self:: Item>
where Self:Sized;

fn skip_while<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Une utilisation courante de l' `skip` adaptateur consiste à ignorer le nom de la commande lors de l'itération sur les arguments de ligne de commande d'un programme. Au [chapitre 2](#), notre calculateur de plus grand dénominateur commun a utilisé le code suivant pour parcourir ses arguments de ligne de commande :

```
for arg in std::env::args().skip(1) {  
    ...  
}
```

La `std::env::args` fonction renvoie un itérateur qui produit les arguments du programme sous la forme `string`s, le premier élément étant le nom du programme lui-même. Ce n'est pas une chaîne que nous voulons traiter dans cette boucle. L'appel `skip(1)` à cet itérateur renvoie un nouvel itérateur qui supprime le nom du programme la première fois qu'il est appelé, puis produit tous les arguments suivants.

L' `skip_while` adaptateur utilise une fermeture pour décider du nombre d'éléments à supprimer depuis le début de la séquence. Vous pouvez parcourir les lignes du corps du message de la section précédente comme ceci :

```
for body in message.lines()  
    .skip_while(|l| !l.is_empty())  
    .skip(1) {  
        println!("{}" , body);  
    }
```

Cela `skip_while` permet d'ignorer les lignes non vides, mais cet itérateur produit la ligne vide elle-même - après tout, la fermeture renvoyée `false` pour cette ligne. Nous utilisons donc `skip` également la méthode pour supprimer cela, nous donnant un itérateur dont le premier élément sera la première ligne du corps du message. Combiné avec la déclaration de `message` de la section précédente, ce code imprime :

```
Did you get any writing done today?  
When will you stop wasting time plotting fractals?
```

visible

Un aperçuiteator vous permet de jeter un coup d'œil au prochain élément qui sera produit sans le consommer réellement. Vous pouvez trans-

former n'importe quel itérateur en un itérateur visible en appelant la méthode `Iterator` du trait `peekable` :

```
fn peekable(self) -> std::iter::Peekable<Self>
    where Self:Sized;
```

Ici, `Peekable<Self>` est a `struct` qui implémente `Iterator<Item=Self::Item>`, et `Self` est le type de l'itérateur sous-jacent.

Un `Peekable` itérateur a une méthode supplémentaire `peek` qui renvoie un `Option<&Item>`: `None` si l'itérateur sous-jacent est terminé et sinon `Some(r)`, où `r` est une référence partagée à l'élément suivant. (Notez que si le type d'élément de l'itérateur est déjà une référence à quelque chose, cela finit par être une référence à une référence.)

L'appel `peek` essaie de dessiner l'élément suivant à partir de l'itérateur sous-jacent, et s'il y en a un, le met en cache jusqu'au prochain appel à `next`. Toutes les autres `Iterator` méthodes sur `Peekable` connaissent ce cache : par exemple, `iter.last()` sur un itérateur `peekable` `iter` sait vérifier le cache après avoir épuisé l'itérateur sous-jacent.

Les itérateurs `Peekable` sont essentiels lorsque vous ne pouvez pas décider du nombre d'éléments à consommer à partir d'un itérateur jusqu'à ce que vous soyez allé trop loin. Par exemple, si vous analysez des nombres à partir d'un flux de caractères, vous ne pouvez pas décider où le nombre se termine tant que vous n'avez pas vu le premier caractère non numérique qui le suit :

```
use std::iter::Peekable;

fn parse_number<I>(tokens: &mut Peekable<I>) -> u32
    where I:Iterator<Item=char>
{
    let mut n = 0;
    loop {
        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}
```

```

let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);
// Look, `parse_number` didn't consume the comma! So we will.
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);

```

La `parse_number` fonction utilise `peek` pour vérifier le caractère suivant et ne le consomme que s'il s'agit d'un chiffre. S'il ne s'agit pas d'un chiffre ou si l'itérateur est épuisé (c'est-à-dire si `peek` return `None`), nous renvoyons le nombre que nous avons analysé et laissons le caractère suivant dans l'itérateur, prêt à être utilisé.

fusible

Une fois un `Iterator` renvoyé `None`, le trait ne spécifie pas comment il doit se comporter si vous appelez `next` à nouveau sa méthode. La plupart des itérateurs reviennent simplement `None`, mais pas tous. Si votre code compte sur ce comportement, vous pourriez être surpris.

L'`Iterator` adaptateur `fuse` prend n'importe quel itérateur et en produit un qui continuera à revenir `None` une fois qu'il l'aura fait la première fois :

```

struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("totally the last item")
        } else {
            self.0 = true; // D'oh!
            None
        }
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("totally the last item"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("totally the last item"));

let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("totally the last item"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);

```

L' `fuse` adaptateur est probablement plus utile dans le code générique qui doit fonctionner avec des itérateurs d'origine incertaine. Plutôt que d'espérer que chaque itérateur auquel vous devrez faire face se comportera bien, vous pouvez utiliser `fuse` pour vous en assurer.

Itérateurs réversibles et rev

Quelques itérateurssont capables de tirer des éléments des deux extrémités de la séquence. Vous pouvez inverser ces itérateurs à l'aide de l'`rev` adaptateur. Par exemple, un itérateur sur un vecteur pourrait tout aussi bien dessiner des éléments depuis la fin du vecteur que depuis le début. De tels itérateurs peuvent implémenter le

`std::iter::DoubleEndedIterator` trait, qui s'étend `Iterator`:

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

Vous pouvez considérer un itérateur à deux extrémités comme ayant deux doigts marquant l'avant et l'arrière actuels de la séquence. Dessiner des éléments de chaque extrémité fait avancer ce doigt vers l'autre; quand les deux se rencontrent, l'itération est faite :

```
let bee_parts = ["head", "thorax", "abdomen"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(), Some(&"head"));
assert_eq!(iter.next_back(), Some(&"abdomen"));
assert_eq!(iter.next(), Some(&"thorax"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(), None);
```

La structure d'un itérateur sur une tranche rend ce comportement facile à implémenter : c'est littéralement une paire de pointeurs vers le début et la fin de la plage d'éléments que nous n'avons pas encore produits ; `next` et `next_back` tirez simplement un élément de l'un ou de l'autre. Les itérateurs pour les collections ordonnées comme `BTreeSet` et `BTreeMap` sont également à double extrémité : leur `next_back` méthode dessine les plus grands éléments ou entrées en premier. En général, la bibliothèque standard fournit une itération double chaque fois que cela est pratique.

Mais tous les itérateurs ne peuvent pas le faire aussi facilement : un itérateur produisant des valeurs à partir d'autres threads arrivant sur un canal `Receiver` n'a aucun moyen d'anticiper quelle pourrait être la dernière valeur reçue. En général, vous devrez consulter la documentation de la bibliothèque standard pour voir quels itérateurs implémentent `DoubleEndedIterator` et lesquels ne le font pas.

Si un itérateur est à deux extrémités, vous pouvez l'inverser avec l'`rev` adaptateur :

```
fn rev(self) -> impl Iterator<Item=Self>
    where Self:Sized + DoubleEndedIterator;
```

L'itérateur renvoyé est également à double extrémité : ses méthodes `next` et `prev` sont simplement échangées : `next_back`

```
let meals = ["breakfast", "lunch", "dinner"];

let mut iter = meals.iter().rev();
assert_eq!(iter.next(), Some(&"dinner"));
assert_eq!(iter.next(), Some(&"lunch"));
assert_eq!(iter.next(), Some(&"breakfast"));
assert_eq!(iter.next(), None);
```

La plupart des adaptateurs d'itérateur, s'ils sont appliqués à un itérateur réversible, renvoient un autre itérateur réversible. Par exemple, `map` et `filter` préserver la réversibilité.

inspecter

L' `inspect` adaptateur est pratique pour déboguer les pipelines des adaptateurs d'itérateur, mais il n'est pas beaucoup utilisé dans le code de production. Il applique simplement une fermeture à une référence partagée à chaque élément, puis transmet l'élément. La fermeture ne peut pas affecter les articles, mais elle peut faire des choses comme les imprimer ou faire des affirmations à leur sujet.

Cet exemple montre un cas dans lequel la conversion d'une chaîne en majuscule modifie sa longueur :

```
let upper_case:String = "große".chars()
    .inspect(|c| println!("before: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!(" after:      {:?}", c))
```

```
.collect();
assert_eq!(upper_case, "GROSSE");
```

L'équivalent majuscule de la lettre allemande minuscule "ß" est "SS", c'est pourquoi `char::to_uppercase` renvoie un itérateur sur les caractères, pas un seul caractère de remplacement. Le code précédent utilise `flat_map` pour concaténer toutes les séquences qui `to_uppercase` reviennent dans un seul `String`, en imprimant ce qui suit :

```
before: 'g'
after:    'G'
before: 'r'
after:    'R'
before: 'o'
after:    'O'
before: 'ß'
after:    'S'
after:    'S'
before: 'e'
after:    'E'
```

chaîne

L' `chain` adaptateur ajoute un itérateur à un autre. Plus précisément, `i1.chain(i2)` renvoie un itérateur qui tire les éléments de `i1` jusqu'à ce qu'il soit épuisé, puis tire les éléments de `i2`.

La `chain` signature de l'adaptateur est la suivante :

```
fn chain<U>(self, other: U) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

En d'autres termes, vous pouvez enchaîner un itérateur avec n'importe quel itérable qui produit le même type d'élément.

Par exemple:

```
let v:Vec<i32> = (1..4).chain([20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

Un `chain` itérateur est réversible si ses deux itérateurs sous-jacents sont :

```
let v:Vec<i32> = (1..4).chain([20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

Un `chain` itérateur garde une trace du retour de chacun des deux itérateurs sous-jacents `None` et dirige `next` et `next_back` appelle l'un ou l'autre selon le cas.

énumérer

L'adaptateur `Iterator` du trait `enumerate` attache un index courant à la séquence, prenant un itérateur qui produit des éléments `A`, `B`, `C`, ... et renvoyant un itérateur qui produit des paires `(0, A)`, `(1, B)`, `(2, C)`, Cela semble trivial à première vue, mais il est utilisé étonnamment souvent.

Les consommateurs peuvent utiliser cet index pour distinguer un élément d'un autre et établir le contexte dans lequel traiter chacun. Par exemple, le traceur d'ensemble Mandelbrot du [chapitre 2](#) divise l'image en huit bandes horizontales et affecte chacune à un fil différent. Ce code utilise `enumerate` pour indiquer à chaque thread à quelle partie de l'image correspond sa bande.

Cela commence par un tampon rectangulaire de pixels :

```
let mut pixels = vec![0; columns * rows];
```

Ensuite, il utilise `chunks_mut` pour diviser l'image en bandes horizontales, une par thread :

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands:Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).collect()
```

Et puis il itère sur les bandes, en commençant un fil pour chacune :

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // start a thread to render rows `top..top + band_rows`
    ...
}
```

Chaque itération obtient une paire `(i, band)`, où `band` est la `&mut [u8]` tranche de la mémoire tampon de pixels dans laquelle le thread doit puiser, et `i` est l'index de cette bande dans l'image globale, grâce à l'

enumerate adaptateur. Compte tenu des limites du tracé et de la taille des bandes, il s'agit d'informations suffisantes pour que le thread détermine quelle partie de l'image lui a été attribuée et donc dans quoi dessiner band .

Vous pouvez considérer les `(index, item)` paires qui enumèrent produisent comme analogues aux `(key, value)` paires que vous obtenez lors de l'itération sur une `HashMap` ou une autre collection associative. Si vous parcourrez une tranche ou un vecteur, la `index` est la « clé » sous laquelle `item` apparaît.

Zip *: français

L' `zip` adaptateur combine deux itérateurs en un seul itérateur qui produit des paires contenant une valeur de chaque itérateur, comme une fermeture éclair joignant ses deux côtés en une seule couture. L'itérateur compressé se termine lorsque l'un des deux itérateurs sous-jacents se termine.

Par exemple, vous pouvez obtenir le même effet que l'`enumerate` adaptateur en compressant la plage de fin illimitée `0..` avec l'autre itérateur :

```
let v:Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

En ce sens, vous pouvez considérer `zip` comme une généralisation de `enumerate` : alors que `enumerate` attache des indices à la séquence, `zip` attache n'importe quel élément d'itérateur arbitraire. Nous avons suggéré précédemment que cela `enumerate` peut aider à fournir un contexte pour le traitement des éléments ; `zip` est une manière plus flexible de faire la même chose.

L'argument `to` `zip` n'a pas besoin d'être lui-même un itérateur ; il peut s'agir de n'importe quel itérable :

by_ref

Tout au long de cette section, nous avons attaché des adaptateurs aux itérateurs. Une fois que vous l'avez fait, pouvez-vous retirer l'adaptateur ? Généralement, non : les adaptateurs s'approprient l'itérateur sous-jacent et ne fournissent aucune méthode pour le rendre.

`by_ref` La méthode d'un itérateur emprunte une référence mutable à l'itérateur afin que vous puissiez appliquer des adaptateurs à la référence. Lorsque vous avez fini de consommer des éléments de ces adaptateurs, vous les supprimez, l'emprunt prend fin et vous retrouvez l'accès à votre itérateur d'origine.

Par exemple, plus tôt dans le chapitre, nous avons montré comment pour utiliser `take_while` et `skip_while` traiter les lignes d'en-tête et le corps d'un message électronique. Mais que se passe-t-il si vous voulez faire les deux, en utilisant le même itérateur sous-jacent ? En utilisant `by_ref`, nous pouvons utiliser `take_while` pour gérer les en-têtes, et lorsque cela est fait, récupérer l'itérateur sous-jacent, qui `take_while` est resté exactement en position pour gérer le corps du message :

```
let message = "To: jimb\r\n\
                From: id\r\n\
                \r\n\
                Oooooh, donuts!!\r\n";\n\nlet mut lines = message.lines();\n\nprintln!("Headers:");\nfor header in lines.by_ref().take_while(|l| !l.is_empty()) {\n    println!("{}" , header);\n}\n\nprintln!("\nBody:");\nfor body in lines {\n    println!("{}" , body);\n}
```

L'appel `lines.by_ref()` emprunte une référence mutable à l'itérateur, et c'est cette référence dont l'`take_while` itérateur s'approprie. Cet itérateur sort de la portée à la fin de la première `for` boucle, ce qui signifie que l'emprunt est terminé, vous pouvez donc l'utiliser `lines` à nouveau dans la deuxième `for` boucle. Cela imprime ce qui suit :

Headers:

To: jimb

From: id

Body:

Oooooh, donuts!!

La `by_ref` définition de l'adaptateur est triviale : elle renvoie une référence mutable à l'itérateur. Ensuite, la bibliothèque standard inclut cette étrange petite implémentation :

```
impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I:: Item;
    fn next(&mut self) -> Option<I:: Item> {
        (**self).next()
    }
    fn size_hint(&self) ->(usize, Option<usize>) {
        (**self).size_hint()
    }
}
```

En d'autres termes, si `I` est un type d'itérateur, alors `&mut I` est aussi un itérateur, dont les méthodes `next` et `size_hint` s'en remettre à son référent. Lorsque vous appelez un adaptateur sur une référence mutable à un itérateur, l'adaptateur s'approprie la *référence*, et non l'itérateur lui-même. C'est juste un emprunt qui se termine lorsque l'adaptateur sort de la portée.

cloné, copié

L' `cloned` adaptateur prend un itérateur qui produit des références et renvoie un itérateur qui produit des valeurs clonées à partir de ces références, un peu comme `iter.map(|item| item.clone())`. Naturellement, le type référent doit implémenter `Clone`. Par exemple:

```
let a = ['1', '2', '3', '\u221e'];
assert_eq!(a.iter().next(), Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

L' `copied` adaptateur est la même idée, mais plus restrictive : le type référent doit implémenter `Copy`. Un appel comme `iter.copied()` est à peu près le même que `iter.map(|r| *r)`. Étant donné que chaque type qui implémente `Copy` également implémente `Clone`, `cloned` est strictement plus étendu que `copied`.

ment plus général, mais selon le type d'élément, un `clone` appel peut effectuer des quantités arbitraires d'allocation et de copie. Si vous supposez que cela ne se produira jamais parce que votre type d'élément est quelque chose de simple, il est préférable de l'utiliser `copied` pour que le vérificateur de type vérifie vos hypothèses.

cycle

L' `cycle` adaptateur renvoie un itérateur qui répète indéfiniment la séquence produite par l'itérateur sous-jacent. L'itérateur sous-jacent doit être implémenté `std::clone::Clone` pour `cycle` pouvoir sauvegarder son état initial et le réutiliser à chaque redémarrage du cycle.

Par exemple:

```
let dirs = ["North", "East", "South", "West"];
let mut spin = dirs.iter().cycle();
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
assert_eq!(spin.next(), Some(&"South"));
assert_eq!(spin.next(), Some(&"West"));
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
```

Ou, pour une utilisation vraiment gratuite des itérateurs :

```
use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
let fizzes_buzzes = fizzes.zip(buzzes);

let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
        match tuple {
            (i, ("", "")) => i.to_string(),
            (_, (fizz, buzz)) => format!("{}{}", fizz, buzz)
        });
    }

for line in fizz_buzz {
    println!("{}", line);
}
```

Cela joue un jeu de mots pour enfants, maintenant parfois utilisé comme question d'entretien d'embauche pour les codeurs, dans lequel les joueurs comptent à tour de rôle, remplaçant tout nombre divisible par trois par le

mot `fizz`, et tout nombre divisible par cinq par `buzz`. Nombres divisible par les deux devenir `fizzbuzz`.

Consommer des itérateurs

Jusqu'à présent, nous avons couvert la création d'itérateurs et leur adaptation dans de nouveaux itérateurs ; ici, nous terminons le processus en montrant des façons de consommer leur.

Bien sûr, vous pouvez utiliser un itérateur avec une `for` boucle ou l'appeler `next` explicitement, mais il existe de nombreuses tâches courantes que vous ne devriez pas avoir à écrire encore et encore. Le `Iterator` trait fournit une large sélection de méthodes pour couvrir bon nombre d'entre elles.

Accumulation simple : compte, somme, produit

La `count` méthode tire les éléments d'un itérateur jusqu'à ce qu'il revienne `None` et vous indique combien il en a. Voici un programme court qui compte le nombre de lignes sur son entrée standard :

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

fn main() {
    let stdin = std::io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

Les méthodes `sum` et `product` calcule la somme ou le produit des éléments de l'itérateur, qui doivent être des entiers ou des nombres à virgule flottante :

```
fn triangle(n: u64) ->u64 {
    (1..=n).sum()
}

assert_eq!(triangle(20), 210);

fn factorial(n: u64) ->u64 {
    (1..=n).product()
}

assert_eq!(factorial(20), 2432902008176640000);
```

(Vous pouvez étendre `sum` et `product` travailler avec d'autres types en implémentant les traits `std::iter::Sum` et `std::iter::Product`, que

nous ne décrirons pas dans ce livre.)

maximum minimum

Les méthodes `min` et `max` en `Iterator` retourne le plus petit ou le plus grand élément produit par l'itérateur. Le type d'élément de l'itérateur doit être implémenté `std::cmp::Ord` afin que les éléments puissent être comparés les uns aux autres. Par exemple:

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

Ces méthodes renvoient un `Option<Self::Item>` afin qu'elles puissent revenir `None` si l'itérateur ne produit aucun élément.

Comme expliqué dans "[Comparaisons d'équivalence](#)", la virgule flottante de Rusttypes `f32` et `f64` implémentez uniquement

`std::cmp::PartialOrd`, pas `std::cmp::Ord`, vous ne pouvez donc pas utiliser les méthodes `min` et `max` pour calculer le plus petit ou le plus grand d'une séquence de nombres à virgule flottante. Ce n'est pas un aspect populaire de la conception de Rust, mais c'est délibéré : il n'est pas clair ce que ces fonctions devraient faire avec les valeurs IEEE NaN. Les ignorer simplement risquerait de masquer des problèmes plus graves dans le code.

Si vous savez comment vous souhaitez gérer les valeurs NaN, vous pouvez utiliser les méthodes d'itération `max_by` et à la `min_by` place, qui vous permettent de fournir votre propre fonction de comparaison.

max_by, min_by

Les méthodes `max_by` et `min_by` renvoie l'élément maximum ou minimum produit par l'itérateur, tel que déterminé par une fonction de comparaison que vous fournissez :

```
use std::cmp::Ordering;

// Compare two f64 values. Panic if given a NaN.
fn cmp(lhs: &f64, rhs: &f64) -> Ordering {
    lhs.partial_cmp(rhs).unwrap()
}

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0));
```

```

assert_eq!(numbers.iter().copied().min_by(cmp), Some(1.0));

let numbers = [1.0, 4.0, std::f64::NAN, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0)); // panics

```

Les méthodes `max_by` et `min_by` transmettent les éléments à la fonction de comparaison par référence afin qu'ils puissent fonctionner efficacement avec n'importe quel type d'itérateur, donc `cmp` s'attend à prendre ses arguments par référence, même si nous avions l'habitude `copied` d'obtenir un itérateur qui produit des `f64` éléments.

max_by_key, min_by_key

Les méthodes `max_by_key` et `min_by_key` on `Iterator` vous permet de sélectionner l'élément maximum ou minimum déterminé par une fermeture appliquée à chaque élément. La fermeture peut sélectionner un champ de l'élément ou effectuer un calcul sur les éléments. Étant donné que vous êtes souvent intéressé par les données associées à un minimum ou à un maximum, pas seulement à l'extremum lui-même, ces fonctions sont souvent plus utiles que `min` et `max`. Leurs signatures sont les suivantes :

```

fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self:: Item>
    where Self: Sized, F: FnMut(&Self:: Item) ->B;

```

```

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self:: Item>
    where Self: Sized, F: FnMut(&Self:: Item) ->B;

```

Autrement dit, étant donné une fermeture qui prend un élément et renvoie n'importe quel type ordonné `B`, renvoie l'élément pour lequel la fermeture a renvoyé le maximum ou le minimum `B`, ou `None` si aucun élément n'a été produit.

Par exemple, si vous devez parcourir une table de hachage de villes pour trouver les villes les plus peuplées et les plus peuplées, vous pouvez écrire :

```

use std::collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland", 583_776);
populations.insert("Fossil", 449);
populations.insert("Greenhorn", 2);
populations.insert("Boring", 7_762);
populations.insert("The Dalles", 15_340);

```

```

assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
           Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
           Some((&"Greenhorn", &2)));

```

La fermeture `|&(_name, pop)| pop` est appliquée à chaque élément produit par l'itérateur et renvoie la valeur à utiliser pour la comparaison, dans ce cas, la population de la ville. La valeur renvoyée est l'élément entier, pas seulement la valeur renvoyée par la fermeture. (Naturellement, si vous faisiez souvent des requêtes comme celle-ci, vous voudriez probablement trouver un moyen plus efficace de trouver les entrées que de faire une recherche linéaire dans la table.)

Comparer des séquences d'articles

Vous pouvez utiliser les opérateurs `<` et `==` pour comparer chaînes, vecteurs et tranches, en supposant que leurs éléments individuels peuvent être comparés. Bien que les itérateurs ne prennent pas en charge les opérateurs de comparaison de Rust, ils fournissent des méthodes comme `eq` et `lt` qui font le même travail, tirant des paires d'éléments des itérateurs et les comparant jusqu'à ce qu'une décision puisse être prise. Par exemple:

```

let packed = "Helen of Troy";
let spaced = "Helen    of    Troy";
let obscure = "Helen of Sandusky"; // nice person, just not famous

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// This is true because ' ' < 'o'.
assert!(spaced < obscure);

// This is true because 'Troy' > 'Sandusky'.
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));

```

Les appels pour `split_whitespace` renvoient des itérateurs sur les mots séparés par des espaces de la chaîne. L'utilisation des méthodes `eq` et `gt` sur ces itérateurs effectue une comparaison mot par mot, au lieu d'une comparaison caractère par caractère. Tout cela est possible car `&str` implémente `PartialOrd` et `PartialEq`.

Les itérateurs fournissent les méthodes `eq` et `ne` pour les comparaisons d'égalité, et les méthodes `lt`, `le`, `gt` et `ge` pour les comparaisons or-

données. Les méthodes `cmp` et `partial_cmp` se comportent comme les méthodes correspondantes des traits `Ord` et `.PartialOrd`

tout et tout

Les méthodes `any` et `all` applique une fermeture à chaque élément produit par l'itérateur et retourne `true` si la fermeture revient `true` pour n'importe quel élément, ou pour tous les éléments :

```
let id = "Iterator";  
  
assert!(id.chars().any(char::is_uppercase));  
assert!(!id.chars().all(char::is_uppercase));
```

Ces méthodes ne consomment que le nombre d'éléments nécessaires pour déterminer la réponse. Par exemple, si la fermeture revient `true` pour un élément donné, elle `any` revient `true` immédiatement, sans tirer d'autres éléments de l'itérateur.

position, rposition et ExactSizeIterator

La `position` méthode applique une fermeture à chaque élément de l'itérateur et renvoie l'index du premier élément pour lequel la fermeture renvoie `true`. Plus précisément, il renvoie un `Option` de l'index : si la fermeture ne renvoie `true` aucun élément, `position` renvoie `None`. Il arrête de dessiner des éléments dès que la fermeture revient `true`. Par exemple:

```
let text = "Xerxes";  
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));  
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

La `rposition` méthode est le même, sauf qu'il recherche à partir de la droite. Par exemple :

```
let bytes = b"Xerxes";  
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));  
assert_eq!(bytes.iter().rposition(|&c| c == b'x'), Some(0));
```

La `rposition` méthode nécessite un itérateur réversible afin de pouvoir dessiner des éléments à partir de l'extrême droite de la séquence. Il nécessite également un itérateur de taille exacte afin qu'il puisse affecter des index de la même manière `position`, en commençant par 0 à

gauche. Un itérateur de taille exacte est celui qui implémente le `std::iter::ExactSizeIterator` trait:

```
trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

La `len` méthode renvoie le nombre d'éléments restants et la `is_empty` méthode renvoie `true` si l'itération est terminée.

Naturellement, tous les itérateurs ne savent pas à l'avance combien d'éléments ils produiront. Par exemple, l'`str::chars` itérateur utilisé précédemment ne le fait pas (puisque UTF-8 est un encodage à largeur variable), vous ne pouvez donc pas l'utiliser `rposition` sur des chaînes. Mais un itérateur sur un tableau d'octets connaît certainement la longueur du tableau, il peut donc implémenter `ExactSizeIterator`.

plier et replier

La `fold` méthode est un outil très général pour accumuler une sorte de résultat sur toute la séquence d'éléments produits par un itérateur. Étant donné une valeur initiale, que nous appellerons l'*accumulateur*, et une fermeture, `fold` applique à plusieurs reprises la fermeture à l'accumulateur actuel et à l'élément suivant de l'itérateur. La valeur renvoyée par la fermeture est prise comme nouvel accumulateur, à transmettre à la fermeture avec l'élément suivant. La valeur finale de l'accumulateur est ce qui `fold` lui-même renvoie. Si la séquence est vide, `fold` renvoie simplement l'accumulateur initial.

De nombreuses autres méthodes de consommation des valeurs d'un itérateur peuvent être écrites comme des utilisations de `fold` :

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);           // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);          // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);      // product

// max
assert_eq!(a.iter().cloned().fold(i32::min_value(), std::cmp::max),
          10);
```

La `fold` signature de la méthode est la suivante :

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self:: Item) ->A;
```

Ici, `A` c'est le type d'accumulateur. L' `init` argument est un `A`, tout comme le premier argument et la valeur de retour de la fermeture, et la valeur de retour d'elle- `fold` même.

Notez que les valeurs d'accumulateur sont déplacées vers l'intérieur et l'extérieur de la fermeture, vous pouvez donc les utiliser `fold` avec Copy des types sans accumulateur :

```
let a = ["Pack", "my", "box", "with",
         "five", "dozen", "liquor", "jugs"];

// See also: the `join` method on slices, which won't
// give you that extra space at the end.
let pangram = a.iter()
    .fold(String::new(), |s, w| s + w + " ");
assert_eq!(pangram, "Pack my box with five dozen liquor jugs");
```

La `rfold` méthode est identique à `fold`, sauf qu'il nécessite un itérateur à deux extrémités et traite ses éléments du dernier au premier :

```
let weird_pangram = a.iter()
    .rfold(String::new(), |s, w| s + w + " ");
assert_eq!(weird_pangram, "jugs liquor dozen five with box my Pack");
```

try_fold et try_rfold

La `try_fold` méthode est identique à `fold`, sauf que l'itération peut se terminer plus tôt, sans consommer toutes les valeurs de l'itérateur. La valeur renvoyée par la fermeture à laquelle vous passez `try_fold` indique si elle doit revenir immédiatement ou continuer à replier les éléments de l'itérateur.

Votre fermeture peut renvoyer n'importe lequel de plusieurs types, indiquant comment le pliage doit se dérouler :

- Si votre fermeture renvoie `Result<T, E>`, peut-être parce qu'elle effectue des E/S ou effectue une autre opération faillible, alors le retour `Ok(v)` indique `try_fold` de continuer le pliage, avec `v` comme nouvelle valeur d'accumulateur. Le retour `Err(e)` provoque l'arrêt im-

médiat du pliage. La valeur finale du repli est un `Result` portant la valeur finale de l'accumulateur, ou l'erreur renvoyée par la fermeture.

- Si votre fermeture renvoie `Option<T>`, `Some(v)` indique alors que le pliage doit continuer avec `v` comme nouvelle valeur d'accumulateur et `None` indique que l'itération doit s'arrêter immédiatement. La valeur finale du pli est également un `Option`.
- Enfin, la fermeture peut renvoyer une `std::ops::ControlFlow` valeur. Ce type est une énumération avec deux variantes, `Continue(c)` et `Break(b)`, signifiant continuer avec une nouvelle valeur d'accumulateur `c` ou s'arrêter plus tôt. Le résultat du pli est une `ControlFlow` valeur : `Continue(v)` si le pli a consommé tout l'itérateur, donnant la valeur finale de l'accumulateur `v`; ou `Break(b)`, si la fermeture a renvoyé cette valeur.
`Continue(c)` et `Break(b)` se comportent exactement comme `Ok(c)` et `Err(b)`. L'avantage d'utiliser à la `ControlFlow` place de `Result` est que cela rend votre code un peu plus lisible lorsqu'une sortie anticipée n'indique pas une erreur, mais simplement que la réponse est prête plus tôt. Nous en montrons un exemple ci-dessous.

Voici un programme qui additionne les nombres lus à partir de son entrée standard :

```
use std::error::Error;
use std::io::prelude::*;
use std::str::FromStr;

fn main() -> Result<(), Box<dyn Error>> {
    let stdin = std::io::stdin();
    let sum = stdin.lock()
        .lines()
        .try_fold(0, |sum, line| -> Result<u64, Box<dyn Error>> {
            Ok(sum + u64::from_str(&line?.trim())?)
        })?;
    println!("{}", sum);
    Ok(())
}
```

L'`lines` itérateur sur les flux d'entrée mis en mémoire tampon produit des éléments de type `Result<String, std::io::Error>`, et l'analyse `String` en tant qu'entier peut également échouer. L'utilisation `try_fold` ici permet à la fermeture de revenir `Result<u64, ...>`, nous pouvons donc utiliser l'`? opérateur` pour propager les échecs de la fermeture à la `main` fonction.

Parce `try_fold` qu'il est si flexible, il est utilisé pour implémenter de nombreuses `Iterator` autres méthodes grand public de . Par exemple, voici une implémentation de `all` :

```
fn all<P>(&mut self, mut predicate: P) -> bool
    where P: FnMut(Self:: Item) -> bool,
          Self: Sized
{
    use std::ops::ControlFlow::*;
    self.try_fold(() , |_| {
        if predicate(item) { Continue() } else { Break() }
    }) == Continue()
}
```

Notez que cela ne peut pas être écrit avec ordinaire `fold`: `all` promet d'arrêter de consommer les éléments de l'itérateur sous-jacent dès qu'il `predicate` renvoie `false`, mais `fold` consomme toujours l'itérateur entier.

Si vous implémentez votre propre type d'itérateur, il est utile de déterminer si votre itérateur pourrait être implémenté `try_fold` plus efficacement que la définition par défaut du `Iterator` trait. Si vous pouvez accélérer `try_fold`, toutes les autres méthodes construites dessus en bénéficieront également.

La `try_rfold` méthode, comme son nom l'indique, est la même que `try_fold`, sauf qu'elle tire les valeurs de l'arrière, au lieu de l'avant, et nécessite un itérateur à double extrémité.

nième, nième_retour

La `nth` méthode prend un index `n`, ignore autant d'éléments de l'itérateur et renvoie l'élément suivant, ou `None` si la séquence se termine avant ce point. L'appel `.nth(0)` est équivalent à `.next()`.

Il ne s'approprie pas l'itérateur comme le ferait un adaptateur, vous pouvez donc l'appeler plusieurs fois :

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

Sa signature est montrée ici :

```
fn nth(&mut self, n: usize) -> Option<Self:: Item>
    where Self: Sized;
```

La `nth_back` méthode est sensiblement la même, sauf qu'elle tire de l'arrière d'un itérateur à double extrémité. L'appel `.nth_back(0)` est équivalent à `.next_back()` : il renvoie le dernier élément, ou `None` si l'itérateur est vide.

dernière

La `last` méthode renvoie le dernier élément produit par l'itérateur, ou `None` s'il est vide. Sa signature est la suivante :

```
fn last(self) -> Option<Self:: Item>;
```

Par exemple:

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

Cela consomme tous les éléments de l'itérateur en commençant par le début, même si l'itérateur est réversible. Si vous avez un itérateur réversible et que vous n'avez pas besoin de consommer tous ses éléments, vous devriez plutôt simplement écrire `iter.next_back()`.

find, rfind et find_map

La `find` méthode tire des éléments d'un itérateur, renvoyant le premier élément pour lequel la fermeture donnée renvoie `true`, ou `None` si la séquence se termine avant qu'un élément approprié ne soit trouvé. Sa signature est :

```
fn find<P>(&mut self, predicate: P) -> Option<Self:: Item>
    where Self: Sized,
          P: FnMut(&Self:: Item) -> bool;
```

La `rfind` méthode est similaire, mais il nécessite un itérateur à deux extrémités et recherche les valeurs de l'arrière vers l'avant, renvoyant le dernier élément pour lequel la fermeture renvoie `true`.

Par exemple, en utilisant le tableau des villes et des populations de « [max by key](#), [min by key](#) », vous pouvez écrire :

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 1_000_000),
           None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 500_000),
           Some((&"Portland", &583_776)));
```

Aucune des villes du tableau n'a une population supérieure à un million, mais il y a une ville avec un demi-million d'habitants.

Parfois, votre clôture n'est pas qu'un simple prédicat jetant un jugement booléen sur chaque élément et passant à autre chose : il peut s'agir de quelque chose de plus complexe qui produit une valeur intéressante en soi. Dans ce cas, `find_map` est exactement ce que vous voulez. Sa signature est :

```
fn find_map<B, F>(&mut self, f: F) -> Option<B> where
    F: FnMut(Self:: Item) -> Option<B>;
```

C'est comme `find`, sauf qu'au lieu de retourner `bool`, la fermeture devrait retourner un `Option` d'une certaine valeur. `find_map` renvoie le premier `Option` qui est `Some`.

Par exemple, si nous avons une base de données des parcs de chaque ville, nous pourrions vouloir voir si certains d'entre eux sont des volcans et fournir le nom du parc si c'est le cas :

```
let big_city_with_volcano_park = populations.iter()
    .find_map(|(&city, _)|
        if let Some(park) = find_volcano_park(city, &parks) {
            // find_map returns this value, so our caller knows
            // *which* park we found.
            return Some((city, park.name));
        }
        // Reject this item, and continue the search.
        None
    );
assert_eq!(big_city_with_volcano_park,
           Some((&"Portland", &"Mt. Tabor Park")));
```

Création de collections : `collect` et `FromIterator`

Tout au long du livre, nous avons utilisé la `collect` méthode pour construire des vecteurs contenant les éléments d'un itérateur. Par exemple, au [chapitre 2](#), nous avons appelé `std::env::args()` pour obtenir un itérateur sur les arguments de la ligne de commande du programme, puis avons appelé la `collect` méthode de cet itérateur pour les rassembler dans un vecteur :

```
let args: Vec<String> = std:: env::args().collect();
```

Mais `collect` n'est pas spécifique aux vecteurs : en fait, il peut créer n'importe quel type de collection à partir de la bibliothèque standard de Rust, tant que l'itérateur produit un type d'élément approprié :

```
use std:: collections::{HashSet, BTreeSet, LinkedList, HashMap, BTreeMap};

let args: HashSet<String> = std:: env:: args().collect();
let args: BTreeSet<String> = std:: env:: args().collect();
let args: LinkedList<String> = std:: env::args().collect();

// Collecting a map requires (key, value) pairs, so for this example,
// zip the sequence of strings with a sequence of integers.
let args: HashMap<String, usize> = std:: env:: args().zip(0..).collect();
let args: BTreeMap<String, usize> = std:: env::args().zip(0..).collect();

// and so on
```

Naturellement, `collect` lui-même ne sait pas construire tous ces types. Au lieu de cela, lorsqu'un type de collection aime `Vec` ou `HashMap` sait se construire à partir d'un itérateur, il implémente le `std::iter::FromIterator` trait, pour qui `collect` n'est qu'un placage commode :

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) ->Self;
}
```

Si un type de collection implémente `FromIterator<A>`, alors sa fonction associée au type `from_iter` construit une valeur de ce type à partir d'un itérable produisant des éléments de type `A`.

Dans le cas le plus simple, l'implémentation pourrait simplement construire une collection vide, puis ajouter les éléments de l'itérateur un par un. Par exemple, `std::collections::LinkedList` la mise en œuvre de `FromIterator` fonctionne de cette façon.

Cependant, certains types peuvent faire mieux que cela. Par exemple, construire un vecteur à partir d'un itérateur `iter` pourrait être aussi simple que :

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item)
}
vec
```

Mais ce n'est pas idéal : à mesure que le vecteur grandit, il peut avoir besoin d'étendre son tampon, ce qui nécessite un appel à l'allocateur de tas et une copie des éléments existants. Les vecteurs prennent des mesures algorithmiques pour maintenir cette surcharge faible, mais s'il y avait un moyen d'allouer simplement un tampon de la bonne taille pour commencer, il n'y aurait aucun besoin de redimensionner.

C'est là qu'intervient la méthode `Iterator` du trait `:size_hint`

```
trait Iterator {
    ...
    fn size_hint(&self) ->(usize, Option<usize>) {
        (0, None)
    }
}
```

Cette méthode renvoie une limite inférieure et une limite supérieure facultative sur le nombre d'éléments que l'itérateur produira. La définition par défaut renvoie zéro comme limite inférieure et refuse de nommer une limite supérieure, en disant, en fait, "je n'en ai aucune idée", mais de nombreux itérateurs peuvent faire mieux que cela. Un itérateur sur `a Range`, par exemple, sait exactement combien d'éléments il produira, tout comme un itérateur sur `a Vec` ou `HashMap`. Ces itérateurs fournissent leurs propres définitions spécialisées pour `size_hint`.

Ces limites sont exactement les informations dont `vec` l'implémentation a `FromIterator` besoin pour dimensionner correctement le tampon du nouveau vecteur dès le départ. Les insertions vérifient toujours que le tampon est suffisamment grand, donc même si l'indice est incorrect, seules les performances sont affectées, pas la sécurité. D'autres types peuvent suivre des étapes similaires : par exemple, `HashSet` et `HashMap` également utiliser `Iterator::size_hint` pour choisir une taille initiale appropriée pour leur table de hachage.

Une remarque à propos de l'inférence de type : en haut de cette section, il est un peu étrange de voir le même appel,

`std::env::args().collect()`, produire quatre types de collections différents en fonction de son contexte. Le type de retour de `collect` est son paramètre de type, donc les deux premiers appels sont équivalents à ce qui suit :

```
let args = std::env::args().collect::<Vec<String>>();
let args = std::env::args().collect::<HashSet<String>>();
```

Mais tant qu'il n'y a qu'un seul type qui pourrait éventuellement fonctionner comme `collect` argument de , l'inférence de type de Rust le fournira pour vous. Lorsque vous épelez le type de `args`, vous vous assurez que c'est le cas.

Le trait d'extension

Si un type implémente le `std::iter::Extend` trait, alors sa `extend` méthode ajoute les éléments d'un itérable à la collection :

```
let mut v:Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend([31, 57, 99, 163]);
assert_eq!(v, [1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

Toutes les collections standard implémentent `Extend`, donc elles ont toutes cette méthode ; tout comme `String`. Les tableaux et les tranches, qui ont une longueur fixe, n'en ont pas.

La définition du trait est la suivante :

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T:IntoIterator<Item=A>;
}
```

Évidemment, cela ressemble beaucoup à `std::iter::FromIterator` qui crée une nouvelle collection, alors qu'il `Extend` étend une collection existante. En effet, plusieurs implementations de `FromIterator` dans la bibliothèque standard, créez simplement une nouvelle collection vide, puis appelez- `extend` la pour la remplir. Par exemple, l'implémentation de `FromIterator` for `std::collections::LinkedList` fonctionne de la manière suivante :

```

impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}

```

cloison

La partition méthode divise les éléments d'un itérateur entre deux collections, en utilisant une fermeture pour décider où chaque élément appartient :

```

let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"];

// Amazing fact: the name of a living thing always starts with an
// odd-numbered letter.
let (living, nonliving):(Vec<&str>, Vec<&str>)
    = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living, vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving, vec!["doorknob", "noodle"]);

```

Comme collect , partition peut faire toutes sortes de collections que vous aimez, bien que les deux doivent être du même type. Et comme collect , vous devrez spécifier le type de retour : l'exemple précédent écrit le type de living et nonliving et laisse l'inférence de type choisir les bons paramètres de type pour l'appel à partition .

La signature de partition est la suivante :

```

fn partition<B, F>(self, f: F) -> (B, B)
    where Self: Sized,
          B: Default + Extend<Self:: Item>,
          F: FnMut(&Self:: Item) ->bool;

```

Alors que collect requiert son type de résultat pour implémenter FromIterator , partition requiert à la place std::default::Default , ce qui toutes les collections Rust implémentent en retournant une collection vide, et std::default::Extend .

D'autres langages proposent des partition opérations qui divisent simplement l'itérateur en deux itérateurs, au lieu de créer deux collections.

Mais ce n'est pas un bon choix pour Rust : les éléments tirés de l'itérateur sous-jacent mais pas encore tirés de l'itérateur partitionné approprié devraient être mis en mémoire tampon quelque part ; vous finiriez par constituer une collection quelconque en interne, de toute façon.

for_each et try_for_each

La `for_each` méthode applique simplement une fermeture à chaque élément :

```
[ "doves", "hens", "birds" ].iter()
    .zip([ "turtle", "french", "calling" ])
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
    .for_each(|gift| {
        println!("You have received: {}", gift);
    });
}
```

Cela imprime :

```
You have received: 4 calling birds
You have received: 3 french hens
You have received: 2 turtle doves
```

`for` Ceci est très similaire à une boucle simple , dans laquelle vous pouvez également utiliser des structures de contrôle telles que `break` et `continue`. Mais les longues chaînes d'appels d'adaptateur comme celui-ci sont un peu gênantes dans les `for` boucles :

```
for gift in [ "doves", "hens", "birds" ].iter()
    .zip([ "turtle", "french", "calling" ])
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
{
    println!("You have received: {}", gift);
}
```

Le motif étant lié, `gift` , peut se retrouver assez loin du corps de boucle dans lequel il est utilisé.

Si votre fermeture doit être faillible ou sortir plus tôt, vous pouvez utiliser `try_for_each`:

```
...
    .try_for_each(|gift| {
        writeln!(&mut output_file, "You have received: {}", gift)
    })?;
```

Implémentation de vos propres itérateurs

Vous pouvez mettre en œuvre les `IntoIterator` et `Iterator` traits pour vos propres types, ce qui rend tous les adaptateurs et consommateurs présentés dans ce chapitre disponibles, ainsi que de nombreux autres codes de bibliothèque et de crate écrits pour fonctionner avec l'interface d'itérateur standard. Dans cette section, nous allons montrer un itérateur simple sur un type de plage, puis un itérateur plus complexe sur un type d'arbre binaire.

Supposons que nous ayons le type de plage suivant (simplifié à partir du type de bibliothèque standard `std::ops::Range<T>`) :

```
struct I32Range {
    start: i32,
    end: i32
}
```

L'itération sur un `I32Range` nécessite deux éléments d'état : la valeur actuelle et la limite à laquelle l'itération doit se terminer. Cela se trouve être un bon ajustement pour le `I32Range` type lui-même, en utilisant `start` comme valeur suivante et `end` comme limite. Vous pouvez donc implémenter `Iterator` comme ceci:

```
impl Iterator for I32Range {
    type Item = i32;
    fn next(&mut self) -> Option<i32> {
        if self.start >= self.end {
            return None;
        }
        let result = Some(self.start);
        self.start += 1;
        result
    }
}
```

```
    }
}
```

Cet itérateur produit des `i32` éléments, c'est donc le `Item` type. Si l'itération est terminée, `next` renvoie `None` ; sinon, il produit la valeur suivante et met à jour son état pour se préparer au prochain appel.

Bien sûr, une `for` boucle utilise `IntoIterator::into_iter` pour convertir son opérande en itérateur. Mais la bibliothèque standard fournit une implémentation globale de `IntoIterator` pour chaque type qui implémente `Iterator`, elle `I32Range` est donc prête à l'emploi :

```
let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
    pi += numerator / (2*k + 1) as f64;
    numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754 specifies this result exactly.
assert_eq!(pi as f32, std::f32::consts::PI);
```

Mais `I32Range` c'est un cas particulier, en ce sens que l'itérable et l'itérateur sont du même type. De nombreux cas ne sont pas si simples. Par exemple, voici le type d'arbre binaire du [chapitre 10](#) :

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

La manière classique de parcourir un arbre binaire consiste à effectuer une récurrence, en utilisant la pile d'appels de fonction pour garder une trace de votre place dans l'arbre et des nœuds qui n'ont pas encore été visités. Mais lors de la mise en œuvre `Iterator` pour `BinaryTree<T>`, chaque appel à `next` doit produire exactement une valeur et un retour. Pour garder une trace des nœuds d'arbre qu'il n'a pas encore produits,

l'itérateur doit maintenir sa propre pile. Voici un type d'itérateur possible pour `BinaryTree` :

```
use self:: BinaryTree::*;

// The state of an in-order traversal of a `BinaryTree`.
struct TreeIter<'a, T> {
    // A stack of references to tree nodes. Since we use `Vec`'s
    // `push` and `pop` methods, the top of the stack is the end of the
    // vector.
    //
    // The node the iterator will visit next is at the top of the stack,
    // with those ancestors still unvisited below it. If the stack is empty
    // the iteration is over.
    unvisited: Vec<&'a TreeNode<T>>
}
```

Lorsque nous créons un nouveau `TreeIter`, son état initial devrait être sur le point de produire le nœud feuille le plus à gauche de l'arbre. Selon les règles de la `unvisited` pile, elle devrait donc avoir cette feuille en haut, suivie de ses ancêtres non visités : les nœuds le long du bord gauche de l'arbre. Nous pouvons initialiser `unvisited` en parcourant le bord gauche de l'arbre de la racine à la feuille et en poussant chaque nœud que nous rencontrons, nous allons donc définir une méthode `TreeIter` pour le faire :

```
impl<'a, T: 'a> TreeIter<'a, T> {
    fn push_left_edge(&mut self, mut tree:&'a BinaryTree<T>) {
        while let NonEmpty(ref node) = *tree {
            self.unvisited.push(node);
            tree = &node.left;
        }
    }
}
```

L'écriture `mut tree` permet à la boucle de changer le nœud `tree` vers lequel pointe le long du bord gauche, mais comme il `tree` s'agit d'une référence partagée, elle ne peut pas muter les nœuds eux-mêmes.

Avec cette méthode d'assistance en place, nous pouvons donner `BinaryTree` une `iter` méthode qui renvoie un itérateur sur l'arbre :

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
    }
}
```

```

    iter
}
}

```

La `iter` méthode construit un `TreeIter` avec une pile vide `unvisited`, puis appelle `push_left_edge` pour l'initialiser. Le nœud le plus à gauche se retrouve en haut, comme l'exigent les `unvisited` règles de la pile.

Suivant les pratiques de la bibliothèque standard, nous pouvons alors implémenter `IntoIterator` sur une référence partagée à un arbre avec un appel à `BinaryTree::iter`:

```

impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
    type Item = &'a T;
    type IntoIter = TreeIter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}

```

La `IntoIter` définition établit `TreeIter` comme type d'itérateur pour un `&BinaryTree`.

Enfin, dans l'`Iterator` implémentation, nous parcourons réellement l'arbre. Comme `BinaryTree` la `iter` méthode de , la méthode de l'itérateur `next` est guidée par les règles de la pile :

```

impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<&'a T> {
        // Find the node this iteration must produce,
        // or finish the iteration. (Use the `?` operator
        // to return immediately if it's `None`.)
        let node = self.unvisited.pop()?;
        // After `node`, the next thing we produce must be the leftmost
        // child in `node`'s right subtree, so push the path from here
        // down. Our helper method turns out to be just what we need.
        self.push_left_edge(&node.right);

        // Produce a reference to this node's value.
        Some(&node.element)
    }
}

```

Si la pile est vide, l'itération est terminée. Sinon, `node` est une référence au nœud à visiter maintenant ; cet appel renverra une référence à son élément champ. Mais d'abord, nous devons avancer l'état de l'itérateur au nœud suivant. Si ce nœud a un sous-arbre droit, le prochain nœud à visiter est le nœud le plus à gauche du sous-arbre, et nous pouvons l'utiliser `push_left_edge` pour le pousser, ainsi que ses ancêtres non visités, sur la pile. Mais si ce nœud n'a pas de sous-arbre droit, `push_left_edge` n'a aucun effet, ce qui est exactement ce que nous voulons : nous pouvons compter sur le nouveau sommet de la pile pour être `node` le premier ancêtre non visité de , le cas échéant.

Avec `IntoIterator` et `Iterator` les implémentations en place, nous pouvons enfin utiliser une `for` boucle pour itérer sur une `BinaryTree` par référence. En utilisant la `add` méthode `BinaryTree` de [« Remplir un arbre binaire »](#) :

```
// Build a small tree.  
let mut tree = BinaryTree::Empty;  
tree.add("jaeger");  
tree.add("robot");  
tree.add("droid");  
tree.add("mecha");  
  
// Iterate over it.  
let mut v = Vec::new();  
for kind in &tree {  
    v.push(*kind);  
}  
assert_eq!(v, ["droid", "jaeger", "mecha", "robot"]);
```

[La figure 15-1](#) montre comment la `unvisited` pile se comporte lorsque nous parcourons un exemple d'arborescence. À chaque étape, le prochain nœud à visiter est en haut de la pile, avec tous ses ancêtres non visités en dessous.

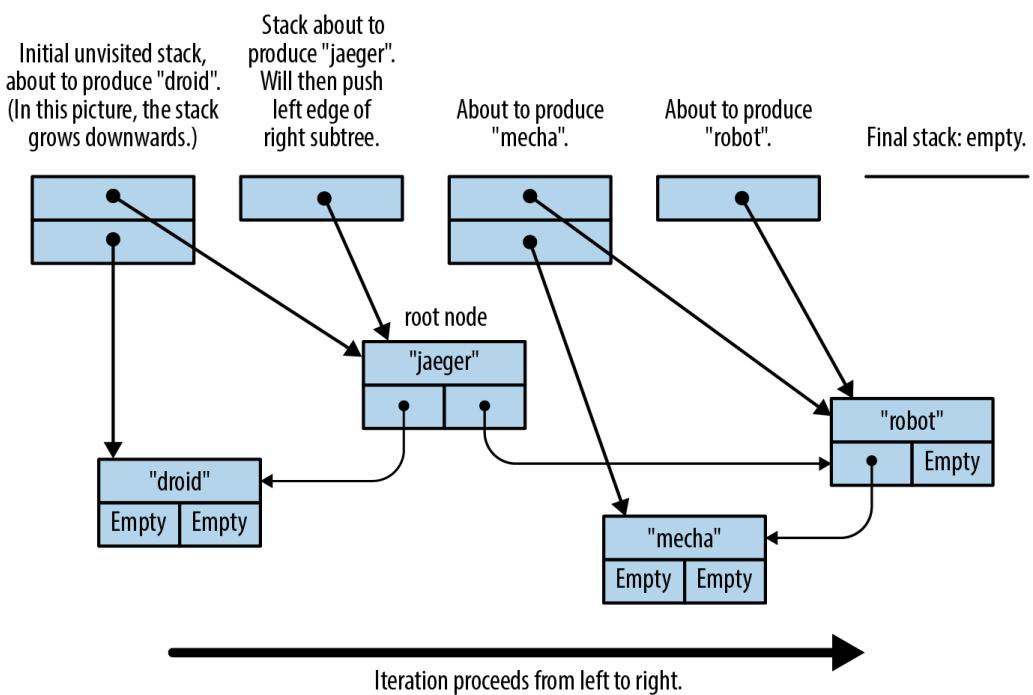


Illustration 15-1. Itérer sur un arbre binaire

Tous les adaptateurs et consommateurs d'itérateurs habituels sont prêts à être utilisés sur nos arbres:

```
assert_eq!(tree.iter()
    .map(|name| format!("mega-{}", name))
    .collect::<Vec<_>>(),
    vec![ "mega-droid", "mega-jaeger",
        "mega-mecha", "mega-robot"]);
```

Les itérateurs sont l'incarnation de la philosophie de Rust consistant à fournir des abstractions puissantes et sans coût qui améliorent l'expressivité et la lisibilité du code. Les itérateurs ne remplacent pas entièrement les boucles, mais ils fournissent une primitive capable avec une évaluation paresseuse intégrée et d'excellentes performances.

¹ En fait, puisque `Option` est un itérable se comportant comme une séquence de zéro ou un élément, `iterator.filter_map(closure)` est équivalent à `iterator.flat_map(closure)`, en supposant qu'il `closure` renvoie un `Option<T>`.

Chapitre 16. Collectes

Nous nous comportons tous comme le démon de Maxwell. Les organismes s'organisent. C'est dans l'expérience quotidienne que réside la raison pour laquelle des physiciens sobres au cours de deux siècles ont maintenu en vie ce fantasme de bande dessinée. Nous trions le courrier, construisons des châteaux de sable, résolvons des puzzles, séparons le blé de l'ivraie, réarrangeons les pièces d'échecs, collectionnons les timbres, classons les livres par ordre alphabétique, créons des symétries, composons des sonnets et des sonates et mettons de l'ordre dans nos pièces, et tout cela ne nécessite aucune grande énergie, tant que nous pouvons appliquer l'intelligence.

—James Gleick, *L'information : une histoire, une théorie, un déluge*

La bibliothèque standard Rust contient plusieurs *collections*, types génériques pour stocker des données en mémoire. Nous avons déjà utilisé des collections, telles que `Vec` et `HashMap`, tout au long de ce livre. Dans ce chapitre, nous couvrirons en détail les méthodes de ces deux types, ainsi que les autres collections standard d'une demi-douzaine. Avant de commencer, abordons quelques différences systématiques entre les collections de Rust et celles d'autres langues.

Premièrement, les déménagements et les emprunts sont partout. Rust utilise des déplacements pour éviter la copie profonde des valeurs. C'est pourquoi la méthode `Vec<T>::push(item)` prend son argument par valeur, pas par référence. La valeur est déplacée dans le vecteur. Les diagrammes du [chapitre 4](#) montrent comment cela fonctionne en pratique : pousser un Rust `String` vers a `Vec<String>` est rapide, car Rust n'a pas à copier les données de caractères de la chaîne, et la propriété de la chaîne est toujours claire.

Deuxièmement, Rust n'a pas d'invalidation erreurs - le type de bogue de pointeur pendant où une collection est redimensionnée ou modifiée d'une autre manière, alors que le programme contient un pointeur vers des données à l'intérieur. Les erreurs d'invalidation sont une autre source de comportement indéfini en C++, et elles provoquent occasionnellement `ConcurrentModificationException` même dans les langages sécurisés en mémoire. Le vérificateur d'emprunt de Rust les exclut au moment de la compilation.

Enfin, Rust n'a pas `null`, nous verrons donc `Option`s à des endroits où d'autres langages utiliseraient `null`.

En dehors de ces différences, les collections de Rust correspondent à ce que vous attendez. Si vous êtes un programmeur expérimenté pressé, vous pouvez survoler ici, mais ne manquez pas "[Entries](#)".

Aperçu

[Le tableau 16-1](#) montre les huit collections standard de Rust. Tous sont des types génériques.

Tableau 16-1. Résumé des collections standards

Le recueil	La description	Type de collection similaire dans...		
		C++	Java	Python
Vec<T>	Tableau évolutif	vector	ArrayList	list
VecDeque<T>	File d'attente double (tampon circulaire extensible)	deque	ArrayDeque	collections.deque
LinkedList<T>	Liste doublement liée	list	LinkedList	—
BinaryHeap<T> where T: Ord	Tas maximum	priori ty_ que ue	PriorityQueue	heapq
HashMap<K, V> where K: Eq + Hash	Table de hachage clé-valeur	unordered_map	HashMap	dict
BTreeMap<K, V> where K: Ord	Tableau des valeurs-clés triées	map	TreeMap	—
HashSet<T> where T: Eq + Hash	Ensemble non ordonné basé sur le hachage	unordered_set	HashSet	set

Le recueil	La description	Type de collection similaire dans...		
		C++	Java	Python
BTrees et<T> where T: Ord	Ensemble trié	set	TreeSet t	—

`Vec<T>`, `HashMap<K, V>`, et `HashSet<T>` sont les types de collection les plus généralement utiles. Les autres ont des utilisations de niche. Ce chapitre traite tour à tour de chaque type de collection :

`Vec<T>`

Un cultivable, tableau de valeurs alloué par tas de type `T`. Environ la moitié de ce chapitre est consacrée à `Vec` ses nombreuses méthodes utiles.

`VecDeque<T>`

Comme `Vec<T>`, mais en mieux à utiliser comme file d'attente premier entré, premier sorti. Il prend en charge efficacement l'ajout et la suppression de valeurs au début de la liste ainsi qu'à l'arrière. Cela se fait au prix de rendre toutes les autres opérations légèrement plus lentes.

`BinaryHeap<T>`

Une priorité file d'attente. Les valeurs de `a BinaryHeap` sont organisées de manière à ce qu'il soit toujours efficace de rechercher et de supprimer la valeur maximale.

`HashMap<K, V>`

Un tableau de paires clé-valeur. La recherche d'une valeur par sa clé est rapide. Les entrées sont stockées dans un ordre arbitraire.

`BTreesMap<K, V>`

Comme `HashMap<K, V>`, mais il conserve les entrées triées par clé. A `BTreesMap<String, i32>` stocke ses entrées dans l'`String` ordre de comparaison. À moins que vous n'ayez besoin que les entrées restent triées, `a HashMap` est plus rapide.

`HashSet<T>`

Un ensemble de valeurs de type `T`. L'ajout et la suppression de valeurs sont rapides, et il est rapide de demander si une valeur donnée est dans l'ensemble ou non.

`BTreesSet<T>`

Comme `HashSet<T>`, mais il conserve les éléments triés par valeur. Encore une fois, à moins que vous n'ayez besoin de trier les données, `a HashSet` est plus rapide.

Parce `LinkedList` qu'il est rarement utilisé (et qu'il existe de meilleures alternatives, à la fois en termes de performances et d'interface, pour la plupart des cas d'utilisation), nous ne le décrivons pas ici.

Vec<T>

Nous supposerons une certaine familiarité avec `vec`, puisque nous l'avons utilisé tout au long du livre. Pour une introduction, voir "["Vec-teurs"](#)". Ici, nous décrirons enfin ses méthodes et son fonctionnement interne en profondeur.

La façon la plus simple de créer un vecteur est d'utiliser la `vec!` macro:

```
// Create an empty vector
let mut numbers:Vec<i32> = vec![];

// Create a vector with given contents
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024 zeroed-out bytes
```

Comme décrit au [chapitre 4](#), un vecteur a trois champs : la longueur, la capacité et un pointeur vers une allocation de tas où les éléments sont stockés. [La figure 16-1](#) montre comment les vecteurs précédents apparaissent en mémoire. Le vecteur vide, `numbers`, a initialement une capacité de 0. Aucune mémoire de tas ne lui est allouée jusqu'à ce que le premier élément soit ajouté.

Comme toutes les collections, `vec` implémente `std::iter::FromIterator`, vous pouvez donc créer un vecteur à partir de n'importe quel itérateur en utilisant la `.collect()` méthode de l'itérateur, comme décrit dans « [Construire des collections : collect et FromIterator](#) » :

```
// Convert another collection to a vector.
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```

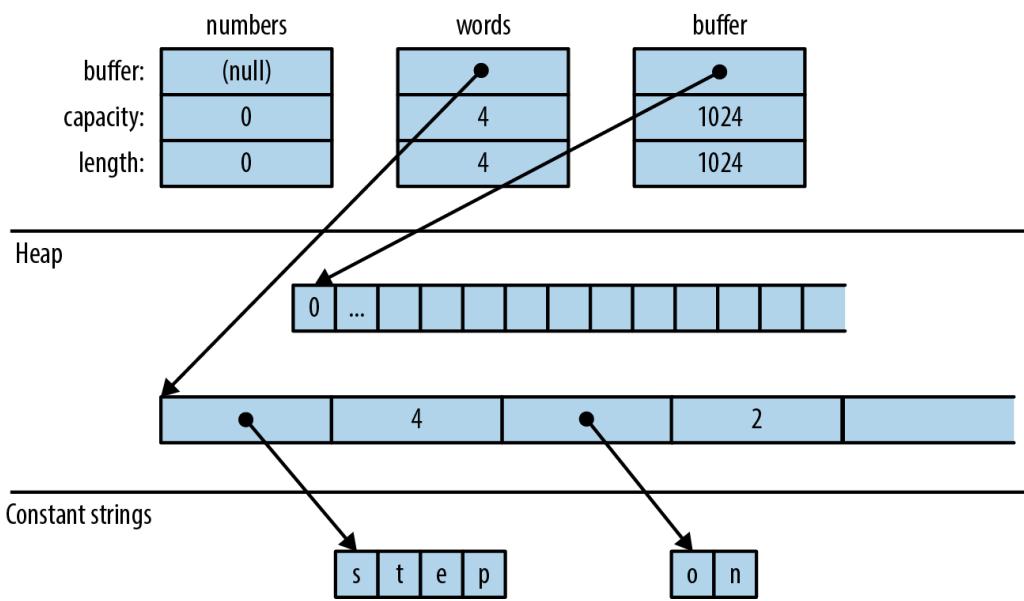


Illustration 16-1. Disposition vectorielle en mémoire : chaque élément de mots est une `&str` valeur composée d'un pointeur et d'une longueur

Accéder aux éléments

Obtenir des éléments d'un tableau, d'une tranche ou d'un vecteur par index est simple :

```
// Get a reference to an element
let first_line = &lines[0];

// Get a copy of an element
let fifth_number = numbers[4];           // requires Copy
let second_line = lines[1].clone();        // requires Clone

// Get a reference to a slice
let my_ref = &buffer[4..12];

// Get a copy of a slice
let my_copy = buffer[4..12].to_vec();    // requires Clone
```

Toutes ces formes paniquent si un index est hors limites.

Rust est pointilleux sur les types numériques et ne fait aucune exception pour les vecteurs. Longueurs et indices des vecteurs sont de type `usize`. Essayer d'utiliser un `u32`, `u64` ou `isize` comme index vectoriel est une erreur. Vous pouvez utiliser un `n as usize` casting pour convertir au besoin; voir "[Type Casts](#)".

Plusieurs méthodes permettent d'accéder facilement à des éléments particuliers d'un vecteur ou d'une tranche (notez que toutes les méthodes de tranche sont également disponibles sur les tableaux et les vecteurs) :

`slice.first()`

Retourne une référence au premier élément de `slice`, le cas échéant.

Le type de retour est `Option<&T>`, donc la valeur de retour est `None` si `slice` est vide et `Some(&slice[0])` si ce n'est pas vide :

```
if let Some(item) = v.first() {  
    println!("We got one! {}", item);  
}  
  
slice.last()
```

Similaire mais renvoie une référence au dernier élément.

```
slice.get(index)
```

Retourne une référence à `slice[index]`, si elle existe. Si `slice` a moins de `index+1` éléments, cela retourne `None` :

```
let slice = [0, 1, 2, 3];  
assert_eq!(slice.get(2), Some(&2));  
assert_eq!(slice.get(4), None);  
  
slice.first_mut(),  
slice.last_mut(), slice.get_mut(index)
```

Variantes des précédents qui empruntent `mut` les références :

```
let mut slice = [0, 1, 2, 3];  
{  
    let last = slice.last_mut().unwrap(); // type of last: &mut i32  
    assert_eq!(*last, 3);  
    *last = 100;  
}  
assert_eq!(slice, [0, 1, 2, 100]);
```

Étant donné que renvoyer une `T` valeur par signifierait la déplacer, les méthodes qui accèdent aux éléments en place renvoient généralement ces éléments par référence.

Une exception est la `.to_vec()` méthode, qui fait des copies :

```
slice.to_vec()
```

Cloner une tranche entière, renvoyant un nouveau vecteur :

```
let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
assert_eq!(v.to_vec(),
```

```

    vec![1, 2, 3, 4, 5, 6, 7, 8, 9];
assert_eq!(v[0..6].to_vec(),
    vec![1, 2, 3, 4, 5, 6]);

```

Cette méthode n'est disponible que si les éléments sont clonables, ceest, where `T: Clone`.

Itération

Vecteurs, les tableaux et les tranches sont itérables, soit par valeur, soit par référence, en suivant le modèle décrit dans [« IntoIterator Implementations »](#) :

- L'itération sur un `Vec<T>` tableau ou `[T; N]` produit des éléments de type `T`. Les éléments sont déplacés hors du vecteur ou du tableau un par un, le consommant.
- L'itération sur une valeur de type `&[T; N]`, `&[T]` ou `&Vec<T>` - c'est-à-dire une référence à un tableau, une tranche ou un vecteur - produit des éléments de type `&T`, des références aux éléments individuels, qui ne sont pas déplacés.
- L'itération sur une valeur de type `&mut [T; N]`, `&mut [T]` ou `&mut Vec<T>` produit des éléments de type `&mut T`.

Les tableaux, les tranches et les vecteurs ont également des méthodes `.iter()` et `.iter_mut()` (décrites dans ["Méthodes iter et iter_mut"](#)) pour créer des itérateurs qui produisent des références à leurs éléments.

Nous couvrirons quelques façons plus sophistiquées d'itérer sur une tranche dans ["Splitting"](#).

Vecteurs croissants et rétrécissants

La *longueur*d'un tableau, d'une tranche ou d'un vecteur est le nombre d'éléments qu'il contient :

```
slice.len()
```

Retourne `slice` longueur de , comme un `usize`.

```
slice.is_empty()
```

Est vrai si `slice` ne contient aucun élément (c'est-à-dire `slice.len() == 0`).

Les autres méthodes de cette section concernent les vecteurs croissants et rétrécissants. Ils ne sont pas présents sur les tableaux et les tranches, qui

ne peuvent pas être redimensionnés une fois créés.

Tous les éléments d'un vecteur sont stockés dans un morceau de mémoire contigu, alloué par tas. La *capacité* d'un vecteur est le nombre maximum d'éléments qui tiendraient dans ce morceau. `vec` gère normalement la capacité pour vous, allouant automatiquement un tampon plus grand et y déplaçant les éléments lorsque plus d'espace est nécessaire. Il existe également quelques méthodes de gestion explicite de la capacité :

`Vec::with_capacity(n)`

Crée un nouveau vecteur vide de capacité `n`.

`vec.capacity()`

`vec` Capacité de retour, en tant que `usize`. C'est toujours vrai que
`vec.capacity() >= vec.len()`.

`vec.reserve(n)`

Fait du assurez-vous que le vecteur a au moins une capacité de réserve suffisante pour `n` plus d'éléments : c'est- à-dire qu'il `vec.capacity()` est au moins `vec.len() + n`. S'il y a déjà assez de place, cela ne fait rien. Sinon, cela alloue un tampon plus grand et y déplace le contenu du vecteur.

`vec.reserve_exact(n)`

Comme `vec.reserve(n)`, mais dit `vec` de ne pas allouer de capacité supplémentaire pour la croissance future, au-delà de `n`. Après,
`vec.capacity()` c'est exactement `vec.len() + n`.

`vec.shrink_to_fit()`

Essaie pour libérer de la mémoire supplémentaire si `vec.capacity()` est supérieur à `vec.len()`.

`Vec<T>` a de nombreuses méthodes qui ajoutent ou suppriment des éléments, modifiant la longueur du vecteur. Chacun d'eux prend son `self` argument par `mut` référence.

Ces deux méthodes ajoutent ou suppriment une seule valeur à la fin d'un vecteur :

`vec.push(value)`

Ajoute le donné `value` à la fin de `vec`.

`vec.pop()`

Supprime et renvoie le dernier élément. Le type de retour est `Option<T>`. Ceci retourne `Some(x)` si l'élément poppé est `x` et `None` si le vecteur était déjà vide.

Notez que `.push()` prend son argument par valeur, pas par référence. De même, `.pop()` renvoie la valeur sautée, pas une référence. Il en va de même pour la plupart des autres méthodes de cette section. Ils déplacent des valeurs dans et hors des vecteurs.

Ces deux méthodes ajoutent ou suppriment une valeur n'importe où dans un vecteur :

`vec.insert(index, value)`

Encartsle donné `value` à `vec[index]`, en faisant glisser toutes les valeurs existantes à `vec[index..]` un endroit vers la droite pour faire de la place.

Panique si `index > vec.len()`.

`vec.remove(index)`

Supprimeet renvoie `vec[index]`, en faisant glisser toutes les valeurs existantes à `vec[index+1..]` un endroit vers la gauche pour combler l'écart.

Panique si `index >= vec.len()`, puisque dans ce cas il n'y a aucun élément `vec[index]` à supprimer.

Plus le vecteur est long, plus l'opération est lente. Si vous vous retrouvez à en faire `vec.remove(0)` beaucoup, pensez à utiliser a `VecDeque` (expliqué dans [« VecDeque<T> »](#)) au lieu de a `vec`.

Les deux `.insert()` et `.remove()` sont d'autant plus lents que les éléments doivent être déplacés.

Quatre méthodes changent la longueur d'un vecteur en une valeur spécifique :

`vec.resize(new_len, value)`

Définit `vec` la longueur deà `new_len`. Si cela augmente `vec` la longueur de , des copies de `value` sont ajoutées pour remplir le nouvel espace. Le type d'élément doit implémenter le `Clone` trait.

`vec.resize_with(new_len, closure)`

Justecomme `vec.resize`, mais appelle la fermeture pour construire chaque nouvel élément. Il peut être utilisé avec des vecteurs d'éléments qui ne sont pas `Clone`.

`vec.truncate(new_len)`

Réduit la longueur de `vec` à `new_len`, supprimant tous les éléments qui se trouvaient dans la plage `vec[new_len..]`.

Si `vec.len()` est déjà inférieur ou égal à `new_len`, rien ne se passe.

`vec.clear()`

Supprime tous les éléments de `vec`. C'est la même chose que `vec.truncate(0)`.

Quatre méthodes ajoutent ou suppriment plusieurs valeurs à la fois :

`vec.extend(iterable)`

Ajoute tous les éléments à partir de la `iterable` valeur donnée à la fin de `vec`, dans l'ordre. C'est comme une version multivaleur de `.push()`. L'`iterable` argument peut être tout ce qui implémente `IntoIterator<Item=T>`.

Cette méthode est si utile qu'il existe un trait standard pour cela, le `Extend` trait, que toutes les collections standard implémentent.

Malheureusement, cela provoque `rustdoc` un regroupement `.extend()` avec d'autres méthodes de trait dans une grosse pile au bas du code HTML généré, il est donc difficile de trouver quand vous en avez besoin. Vous n'avez qu'à vous rappeler qu'il est là ! Voir "[Le trait d'extension](#)" pour plus d'informations.

`vec.split_off(index)`

Comme `vec.truncate(index)`, sauf qu'il renvoie un `Vec<T>` contenant les valeurs supprimées à la fin de `vec`. C'est comme une version multivaleur de `.pop()`.

`vec.append(&mut vec2)`

Cette déplace tous les éléments de `vec2` vers `vec`, où `vec2` est un autre vecteur de type `Vec<T>`. Après, `vec2` c'est vide.

C'est comme `vec.extend(vec2)` sauf qu'il `vec2` existe toujours après, avec sa capacité non affectée.

`vec.drain(range)`

Cette supprime `range` `vec[range]` de `vec` et renvoie un itérateur sur les éléments supprimés, où `range` est une valeur de plage, comme `..` ou `0..4`.

Il existe également quelques méthodes bizarres pour supprimer sélectivement certains éléments d'un vecteur :

```
vec.retain(test)
```

Supprimetous les éléments qui ne passent pas le test donné. L' `test` argument est une fonction ou une fermeture qui implémente `FnMut(&T) -> bool`. Pour chaque élément de `vec`, cela appelle `test(&element)`, et s'il retourne `false`, l'élément est supprimé du vecteur et supprimé.

En dehors de la performance, c'est comme écrire:

```
vec = vec.into_iter().filter(test).collect();
```

```
vec.dedup()
```

Goutteséléments répétés. `uniq` C'est comme l' utilitaire shell Unix . Il recherche `vec` les endroits où les éléments adjacents sont égaux et supprime les valeurs supplémentaires égales afin qu'il n'en reste qu'une :

```
let mut byte_vec = b"Missssssssissippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Missisipi");
```

Notez qu'il y a encore deux 's' caractères dans la sortie. Cette méthode supprime uniquement les doublons *adjacents*. Pour éliminer tous les doublons, vous avez trois options : trier le vecteur avant d'appeler `.dedup()`, déplacer les données dans un ensemble ou (pour conserver les éléments dans leur ordre d'origine) utiliser cette `.retain()` astuce :

```
let mut byte_vec = b"Missssssssissippi".to_vec();

let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));

assert_eq!(&byte_vec, b"Misp");
```

Cela fonctionne car `.insert()` revient `false` lorsque l'ensemble contient déjà l'élément que nous insérons.

```
vec.dedup_by(same)
```

Le mêmeas `vec.dedup()`, mais il utilise la fonction ou la fermeture `same(&mut elem1, &mut elem2)`, au lieu de l' `==` opérateur, pour vérifier si deux éléments doivent être considérés comme égaux.

```
vec.dedup_by_key(key)
```

Le même comme `vec.dedup()`, mais il traite deux éléments comme égaux si `key(&mut elem1) == key(&mut elem2)`.

Par exemple, si `errors` est un `Vec<Box<dyn Error>>`, vous pouvez écrire :

```
// Remove errors with redundant messages.
errors.dedup_by_key(|err| err.to_string());
```

De toutes les méthodes couvertes dans cette section, seules `.resize()` les valeurs sont clonées. Les autres fonctionnent en déplaçant des valeurs d'un endroit à un autre.

Joindre

Deux méthodes travailler sur *des tableaux de tableaux*, par lequel nous entendons tout tableau, tranche ou vecteur dont les éléments sont eux-mêmes des tableaux, des tranches ou des vecteurs :

```
slices.concat()
```

Retourne un nouveau vecteur créé en concaténant toutes les tranches :

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
           vec![1, 2, 3, 4, 5, 6]);
```

```
slices.join(&separator)
```

Le même, sauf qu'une copie de la valeur `separator` est insérée entre les tranches :

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
           vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

Scission

C'est facile d'en avoir plusieurs des non `mut`-références dans un tableau, une tranche ou un vecteur à la fois :

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
```

```
let b = &v[j];  
  
let mid = v.len() / 2;  
let front_half = &v[..mid];  
let back_half = &v[mid..];
```

Obtenir plusieurs `mut` références n'est pas si simple :

```
let mut v = vec![0, 1, 2, 3];  
let a = &mut v[i];  
let b = &mut v[j]; // error: cannot borrow `v` as mutable  
                  // more than once at a time  
  
*a = 6;           // references `a` and `b` get used here,  
*b = 7;           // so their lifetimes must overlap
```

Rust l'interdit car si `i == j`, alors `a` et `b` seraient deux `mut` références au même entier, en violation des règles de sécurité de Rust. (Voir "[Partage contre mutation](#)".)

Rust a plusieurs méthodes qui peuvent emprunter `mut` des références à deux ou plusieurs parties d'un tableau, d'une tranche ou d'un vecteur à la fois. Contrairement au code précédent, ces méthodes sont sûres, car de par leur conception, elles divisent toujours les données en régions sans *chevauchement*. Beaucoup de ces méthodes sont également pratiques pour travailler avec des non `mut`-slices, il existe donc des `mut` non-`mut` versions de chacune.

[La figure 16-2](#) illustre ces méthodes.

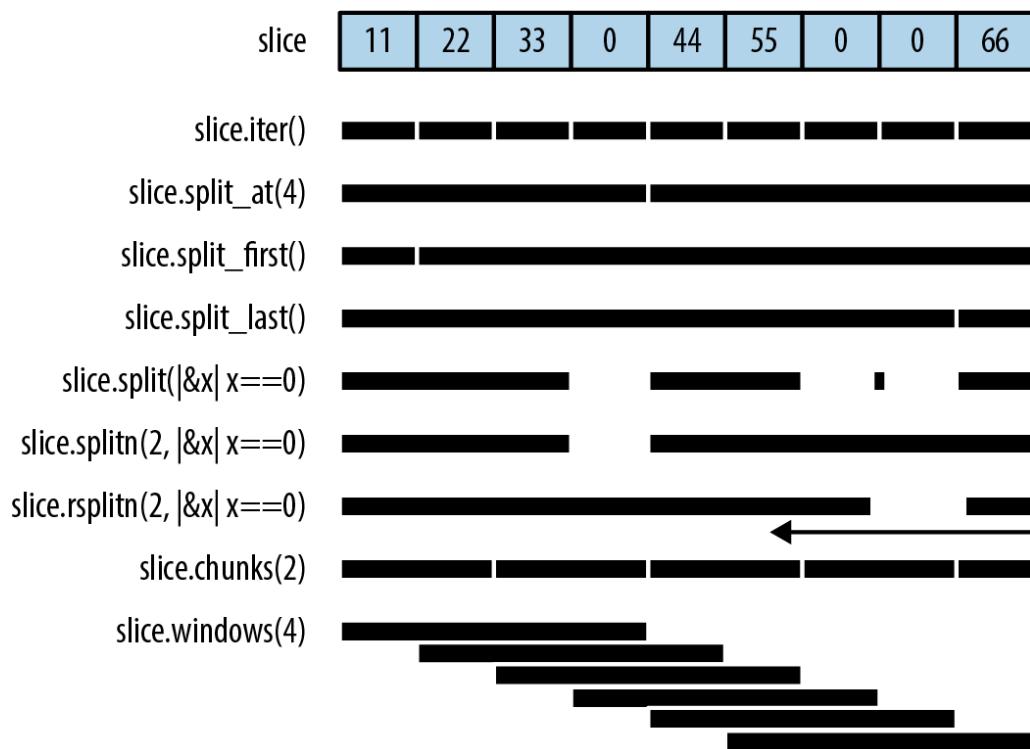


Illustration 16-2. Méthodes de fractionnement illustrées (remarque : le petit rectangle dans la sortie de `slice.split()` est une tranche vide causée par les deux séparateurs adjacents, et `rspltn` produit sa sortie dans l'ordre de bout en bout, contrairement aux autres)

Aucune de ces méthodes ne modifie directement un tableau, une tranche ou un vecteur ; ils renvoient simplement de nouvelles références à des parties des données à l'intérieur :

`slice.iter()`, `slice.iter_mut()`

Produire une référence à chaque élément de `slice`. Nous les avons couverts dans "[Itération](#)".

`slice.split_at(index)`, `slice.split_at_mut(index)`

Casser une tranche en deux, retournant une paire.

`slice.split_at(index)` est équivalent à `(&slice[..index], &slice[index..])`. Ces méthodes paniquent si elles `index` sont hors limites.

`slice.split_first()`, `slice.split_first_mut()`

Aussi renvoie une paire : une référence au premier élément (`slice[0]`) et une référence de tranche à tous les autres (`slice[1..]`).

Le type de retour de `.split_first()` est `Option<(&T, &[T])>` ; le résultat est `None` si `slice` est vide.

`slice.split_last()`, `slice.split_last_mut()`

Ces sont analogues mais séparent le dernier élément plutôt que le premier.

Le type de retour de `.split_last()` est `Option<(&T, &[T])>`.

`slice.split(is_sep), slice.split_mut(is_sep)`

Diviser `slice` en une ou plusieurs sous-tranches, en utilisant la fonction ou la fermeture `is_sep` pour déterminer où diviser. Ils renvoient un itérateur sur les sous-tranches.

Lorsque vous consommez l'itérateur, il appelle `is_sep(&element)` chaque élément de la tranche. Si `is_sep(&element)` est `true`, l'élément est un séparateur. Les séparateurs ne sont inclus dans aucune sous-tranche de sortie.

La sortie contient toujours au moins une sous-tranche, plus une par séparateur. Les sous-tranches vides sont incluses chaque fois que des séparateurs apparaissent adjacents les uns aux autres ou aux extrémités de `slice`.

`slice.split_inclusive(is_sep), slice.split_inclusive_mut(is_sep)`

Ceux-ci fonctionnent comme `split` et `split_mut`, mais incluent le séparateur à la fin de la sous-tranche précédente plutôt que de l'exclure.

`slice.rsplit(is_sep), slice.rsplit_mut(is_sep)`

Juste comme `slice` et `slice_mut`, mais commencez à la fin de la tranche.

`slice.splitn(n, is_sep), slice.splitn_mut(n, is_sep)`

Le même mais ils produisent au plus des `n` sous-tranches. Une fois les premières `n-1` tranches trouvées, `is_sep` n'est plus appelée. La dernière sous-tranche contient tous les éléments restants.

`slice.rsplitn(n, is_sep), slice.rsplitn_mut(n, is_sep)`

Juste comme `.splitn()` et `.splitn_mut()` sauf que la tranche est scannée dans l'ordre inverse. Autrement dit, ces méthodes se divisent sur les derniers `n-1` séparateurs de la tranche, plutôt que sur le premier, et les sous-tranches sont produites à partir de la fin.

`slice.chunks(n), slice.chunks_mut(n)`

Revenir un itérateur sur des sous-tranches non superposées de longueur `n`. Si `n` ne se divise pas `slice.len()` exactement, le dernier morceau contiendra moins de `n` éléments.

`slice.rchunks(n), slice.rchunks_mut(n)`

Juste comme `slice.chunks` et `slice.chunks_mut`, mais commencez à la fin de la tranche.

`slice.chunks_exact(n), slice.chunks_exact_mut(n)`

Revenir un itérateur sur des sous-tranches non superposées de longueur `n`. Si `n` ne divise pas `slice.len()`, le dernier morceau (avec moins de `n` éléments) est disponible dans la `remainder()` méthode du résultat.

`slice.rchunks_exact(n), slice.rchunks_exact_mut(n)`

Juste comme `slice.chunks_exact` et `slice.chunks_exact_mut`, mais commencez à la fin de la tranche.

Il existe une autre méthode pour itérer sur les sous-tranches :

`slice.windows(n)`

Retourne un itérateur qui se comporte comme une "fenêtre coulissante" sur les données dans `slice`. Il produit des sous-tranches qui couvrent `n` des éléments consécutifs de `slice`. La première valeur produite est `&slice[0..n]`, la seconde est `&slice[1..n+1]`, et ainsi de suite.

Si `n` est supérieur à la longueur de `slice`, aucune tranche n'est produite. Si `n` vaut 0, la méthode panique.

Par exemple, si `days.len() == 31`, alors nous pouvons produire toutes les périodes de sept jours en `days` appelant `days.windows(7)`.

Une fenêtre glissante de taille 2 est pratique pour explorer comment une série de données change d'un point de données à l'autre :

```
let changes = daily_high_temperatures
    .windows(2)                      // get adjacent days' temps
    .map(|w| w[1] - w[0])           // how much did it change?
    .collect::<Vec<_>>();
```

Parce que les sous-tranches se chevauchent, il n'y a pas de variation de cette méthode qui renvoie `mut` des références.

Échange

Il y a des méthodes pratiques pour échanger le contenu des tranches :

`slice.swap(i, j)`

Échange les deux éléments `slice[i]` et `slice[j]`.

`slice_a.swap(&mut slice_b)`

Permute tout le contenu de `slice_a` et `slice_b`. `slice_a` et `slice_b` doit être de la même longueur.

Les vecteurs ont une méthode connexe pour supprimer efficacement n'importe quel élément :

```
vec.swap_remove(i)
```

Supprimeet revient `vec[i]`. C'est comme `vec.remove(i)` sauf qu'au lieu de faire glisser le reste des éléments du vecteur pour combler l'espace, il déplace simplement `vec` le dernier élément de dans l'espace. C'est utile lorsque vous ne vous souciez pas de l'ordre des éléments laissés dans le vecteur.

Remplissage

Il existe deux méthodes pratiques pour remplacer le contenu des tranches modifiables :

```
slice.fill(value)
```

Remplit la tranche avec des clones de `value`.

```
slice.fill_with(function)
```

Remplit la tranche avec les valeurs créées en appelant la fonction donnée. Ceci est particulièrement utile pour les types qui implémentent `Default`, mais ne le sont pas `Clone`, comme `Option<T>` ou `Vec<T>` quand ne l' `T` est pas `Clone`.

Tri et recherche

Tranchesproposent trois méthodes de tri :

```
slice.sort()
```

Trie les éléments dans un ordre croissant. Cette méthode est présente uniquement lorsque le type d'élément implémente `Ord`.

```
slice.sort_by(cmp)
```

Trie les éléments d' `slice` utilisation d'une fonction ou d'une fermeture `cmp` pour spécifier l'ordre de tri. `cmp` doit mettre en œuvre `Fn(&T, &T) -> std::cmp::Ordering`.

La mise en œuvre manuelle `cmp` est pénible, à moins que vous ne délégiez à une `.cmp()` méthode :

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

Pour trier par un champ, en utilisant un deuxième champ comme condition de départage, comparez les tuples :

```
students.sort_by(|a, b| {
    let a_key = (&a.last_name, &a.first_name);
    let b_key = (&b.last_name, &b.first_name);
    a_key.cmp(&b_key)
});

slice.sort_by_key(key)
```

Trie les éléments de `slice` dans un ordre croissant par une clé de tri, donnée par la fonction ou la fermeture `key`. Le type de `key` doit implémenter `Fn(&T) -> K` où `K: Ord`.

Ceci est utile lorsqu'il `T` contient un ou plusieurs champs ordonnés, de sorte qu'il puisse être trié de plusieurs manières :

```
// Sort by grade point average, lowest first.
students.sort_by_key(|s| s.grade_point_average());
```

Notez que ces valeurs de clé de tri ne sont pas mises en cache lors du tri, de sorte que la `key` fonction peut être appelée plus de n fois.

Pour des raisons techniques, `key(element)` ne peut renvoyer aucune référence empruntée à l'élément. Cela ne fonctionnera pas :

```
students.sort_by_key(|s| &s.last_name); // error: can't infer lifetime
```

Rust ne peut pas comprendre les durées de vie. Mais dans ces cas, il est assez facile de se rabattre sur `.sort_by()`.

Les trois méthodes effectuent un tri stable.

Pour trier dans l'ordre inverse, vous pouvez utiliser `sort_by` avec une `cmp` fermeture qui échange les deux arguments. Prendre des arguments `|b, a|` plutôt que `|a, b|` de produire effectivement l'ordre inverse. Ou, vous pouvez simplement appeler la `.reverse()` méthode après le tri :

```
slice.reverse()
```

Reverser une tranche en place.

Une fois qu'une tranche est triée, elle peut être recherchée efficacement:

```
slice.binary_search(&value),  
slice.binary_search_by(&value,  
cmp), slice.binary_search_by_key(&value, key)
```

Toutes les recherches pour `value` dans le trié donné `slice`. Remarque qui `value` est passé par référence.

Le type de retour de ces méthodes est `Result<usize, usize>`. Ils renvoient `Ok(index)` si `slice[index]` égal `value` dans l'ordre de tri spécifié. S'il n'y a pas un tel index, alors ils retournent de `Err(insertion_point)` telle sorte que l'insertion `value` à `insertion_point` préservera l'ordre.

Bien sûr, une recherche binaire ne fonctionne que si la tranche est en fait triée dans l'ordre spécifié. Sinon, les résultats sont arbitraires : ordures entrantes, ordures sortantes.

Puisque `f32` et `f64` ont des valeurs `Nan`, ils ne s'implémentent pas `Ord` et ne peuvent pas être utilisés directement comme clés avec les méthodes de tri et de recherche binaire. Pour obtenir des méthodes similaires qui fonctionnent sur des données à virgule flottante, utilisez la `ord_subset` crate.

Il existe une méthode pour rechercher un vecteur qui n'est pas trié :

```
slice.contains(&value)
```

Renvoie `true` si un élément de `slice` est égal à `value`. Cela vérifie simplement chaque élément de la tranche jusqu'à ce qu'une correspondance soit trouvée. Encore une fois, `value` est passé par référence.

Pour trouver l'emplacement d'une valeur dans une tranche, comme `array.indexOf(value)` en JavaScript, utilisez un itérateur :

```
slice.iter().position(|x| *x == value)
```

Cela renvoie un `Option<usize>`.

Comparer des tranches

Si un type prend en charge les opérateurs `==` et `(=)` (le trait, décrit dans [« Comparaisons d'équivalence »](#)), puis les tableaux, les tranches et les vecteurs les prennent également en charge. Deux tranches sont égales si elles ont la même longueur et leurs éléments correspondants sont égaux.

Il en va de même pour les tableaux et les vecteurs. != PartialEq [T;

N] [T] Vec<T>

Si T prend en charge les opérateurs <, <= , > , et >= (le PartialOrd trait, décrit dans « [Comparaisons ordonnées](#) »), alors les tableaux, les tranches et les vecteurs de T do aussi. Les comparaisons de tranches sont lexicographiques.

Deux méthodes pratiques effectuent des comparaisons de tranches courantes :

`slice.starts_with(other)`

Retour true if slice commence par une séquence de valeurs égales aux éléments de la slice other :

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

`slice.ends_with(other)`

Similaire mais vérifie la fin de slice :

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

Éléments aléatoires

Aléatoires les nombres ne sont pas intégrés dans la bibliothèque standard de Rust. La rand caisse, qui les fournit, propose ces deux méthodes pour obtenir une sortie aléatoire à partir d'un tableau, d'une tranche ou d'un vecteur :

`slice.choose(&mut rng)`

Retourne une référence à un élément aléatoire d'une tranche. Comme `slice.first()` et `slice.last()`, cela renvoie un Option<&T> qui est None uniquement si la tranche est vide.

`slice.shuffle(&mut rng)`

Au hasard réordonne les éléments d'une tranche en place. La tranche doit être passée par mut référence.

Ce sont des méthodes du `Rng` trait, vous avez donc besoin d'un Rng générateur de nombres aléatoires pour les appeler. Heureusement, il est facile d'en obtenir un en appelant `rand::thread_rng()`. Pour mélanger le vecteur `my_vec`, on peut écrire :

```
use rand:: seq:: SliceRandom;
use rand::thread_rng;

my_vec.shuffle(&mut thread_rng());
```

Rust élimine les erreurs d'invalidation

Le plus grand publics langages de programmation ont des collections et des itérateurs, et ils ont tous une certaine variation sur cette règle : ne modifiez pas une collection pendant que vous itérez dessus. Par exemple, l'équivalent Python d'un vecteur est une liste :

```
my_list = [1, 3, 5, 7, 9]
```

Supposons que nous essayons de supprimer toutes les valeurs supérieures à 4 de `my_list` :

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index] # bug: modifying list while iterating

print(my_list)
```

(La `enumerate` fonction est l'équivalent Python de la `.enumerate()` méthode de Rust, décrite dans "[enumerate](#)".)

Ce programme, étonnamment, imprime `[1, 3, 7]`. Mais sept est plus grand que quatre. Comment cela s'est-il passé ? Il s'agit d'une erreur d'invalidation : le programme modifie les données tout en itérant dessus, *invalidant* l'itérateur. En Java, le résultat serait une exception ; en C++, c'est un comportement indéfini. En Python, bien que le comportement soit bien défini, il n'est pas intuitif : l'itérateur ignore un élément. `val` n'est jamais 7.

Essayons de reproduire ce bogue dans Rust :

```
fn main() {
    let mut my_vec = vec![1, 3, 5, 7, 9];

    for (index, &val) in my_vec.iter().enumerate() {
        if val > 4 {
            my_vec.remove(index); // error: can't borrow `my_vec` as muta
        }
    }
}
```

```
    println!("{:?}", my_vec);  
}
```

Naturellement, Rust rejette ce programme au moment de la compilation. Lorsque nous appelons `my_vec.iter()`, il emprunte une référence partagée (non-`mut`) au vecteur. La référence vit aussi longtemps que l'itérateur, jusqu'à la fin de la `for` boucle. Nous ne pouvons pas modifier le vecteur en appelant `my_vec.remove(index)` alors qu'une non-`mut` référence existe.

Se faire signaler une erreur, c'est bien, mais bien sûr, encore faut-il trouver un moyen d'obtenir le comportement souhaité ! La solution la plus simple ici est d'écrire :

```
my_vec.retain(|&val| val <= 4);
```

Ou, vous pouvez faire ce que vous feriez en Python ou dans tout autre langage : créer un nouveau vecteur en utilisant un `filter`.

VecDeque<T>

`Vec` prend en charge efficacement l'ajout et la suppression d'éléments uniquement à la fin. Lorsqu'un programme a besoin d'un endroit pour stocker des valeurs qui « attendent en ligne », `Vec` cela peut être lent.

Rust's `std::collections::VecDeque<T>` est un *deque* (prononcé "deck"), une file d'attente à double extrémité. Il prend en charge les opérations d'ajout et de suppression efficaces à l'avant et à l'arrière :

```
deque.push_front(value)
```

Ajoute une valeur au début de la file d'attente.

```
deque.push_back(value)
```

Ajoute une valeur à la fin. (Cette méthode est beaucoup plus utilisée que `.push_front()`, car la convention habituelle pour les files d'attente est que les valeurs sont ajoutées à l'arrière et supprimées à l'avant, comme les personnes qui attendent dans une file.)

```
deque.pop_front()
```

Supprime et renvoie la valeur avant de la file d'attente, renvoyant un `Option<T>` c'est-à-dire `None` si la file d'attente est vide, comme `vec.pop()`.

```
deque.pop_back()
```

Supprimeet renvoie la valeur à l'arrière, renvoyant à nouveau un `Option<T>`.

`deque.front()`, `deque.back()`

Travailler comme `vec.first()` et `vec.last()`. Ils renvoient une référence à l'élément avant ou arrière de la file d'attente. La valeur de retour est un `Option<&T>` si `None` la file d'attente est vide.

`deque.front_mut()`, `deque.back_mut()`

Travailler comme `vec.first_mut()` et `vec.last_mut()`, revenant `Option<&mut T>`.

L'implémentation de `VecDeque` est une mémoire tampon en anneau, comme illustré à la [Figure 16-3](#).

Comme un `vec`, il a une seule allocation de tas où les éléments sont stockés. Contrairement à `vec`, les données ne commencent pas toujours au début de cette région et peuvent « s'enrouler autour » de la fin, comme illustré. Les éléments de cette deque, dans l'ordre, sont `['A', 'B', 'C', 'D', 'E']`. `VecDeque` a des champs privés, étiquetés `start` et `stop` dans la figure, qu'il utilise pour se rappeler où dans le tampon les données commencent et se terminent.

Ajouter une valeur à la file d'attente, à chaque extrémité, signifie revendiquer l'un des emplacements inutilisés, illustré par les blocs les plus sombres, boucler ou allouer une plus grande quantité de mémoire si nécessaire.

`VecDeque` gère l'emballage, vous n'avez donc pas à y penser. [La figure 16-3](#) est une vue des coulisses de la `.pop_front()` rapidité de Rust.

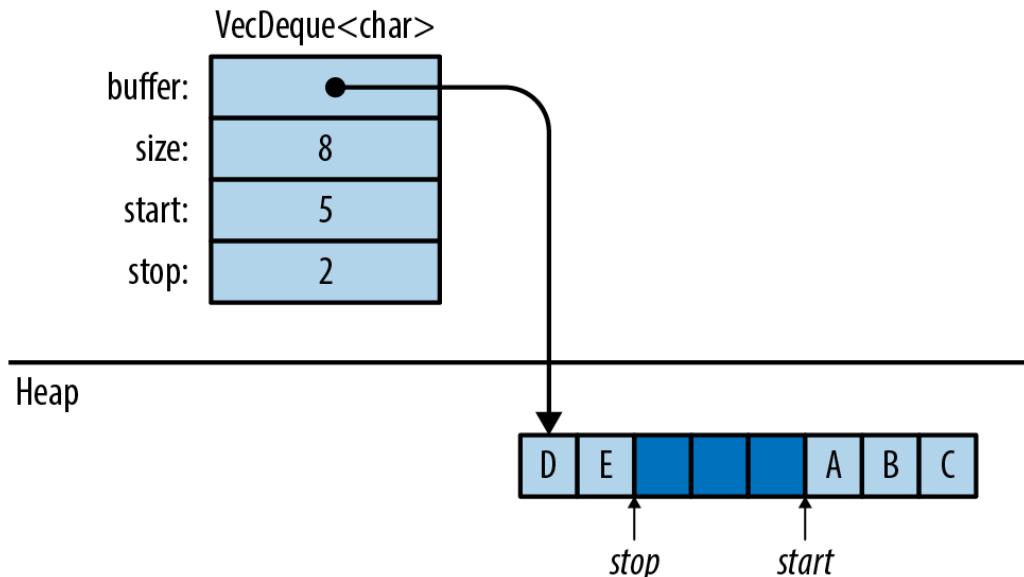


Illustration 16-3. Comment a `VecDeque` est stocké en mémoire

Souvent, lorsque vous avez besoin d'une deque, `.push_back()` et `.pop_front()` sont les deux seules méthodes dont vous aurez besoin. Les fonctions associées au type `VecDeque::new()` et `VecDeque::with_capacity(n)`, pour la création de files d'attente, sont identiques à leurs homologues dans `vec`. De nombreuses `vec` méthodes sont également implémentées pour `VecDeque`: `.len()` et `.is_empty()`, `.insert(index, value)`, `.remove(index)`, `.extend(iterable)`, etc.

Les deque, comme les vecteurs, peuvent être itérés par valeur, par référence partagée ou par `mut` référence. Ils ont les trois méthodes itératives `.into_iter()`, `.iter()` et `.iter_mut()`. Ils peuvent être indexés de la manière habituelle : `deque[index]`.

Comme les deque ne stockent pas leurs éléments de manière contiguë en mémoire, ils ne peuvent pas hériter de toutes les méthodes de tranches. Mais si vous êtes prêt à payer le coût du déplacement du contenu, `VecDeque` fournit une méthode qui résoudra cela :

`deque.make_contiguous()`

Prend `&mut self` et réorganise le `VecDeque` dans la mémoire contiguë, retournant `&mut [T]`.

`Vec`s et `VecDeque`s sont étroitement liés, et la bibliothèque standard fournit deux implémentations de trait pour une conversion facile entre les deux :

`Vec::from(deque)`

`Vec<T>` met en œuvre `From<VecDeque<T>>`, donc cela tourne une deque dans un vecteur. Cela coûte $O(n)$ en temps, car cela peut nécessiter un réarrangement des éléments.

`VecDeque::from(vec)`

`VecDeque<T>` met en œuvre `From<Vec<T>>`, donc cela transforme un vecteur dans une deque. C'est aussi $O(n)$, mais c'est généralement rapide, même si le vecteur est grand, car l'allocation de tas du vecteur peut simplement être déplacée vers le nouveau deque.

Cette méthode facilite la création d'un deque avec des éléments spécifiés, même s'il n'y a pas de `vec_deque![]` macro standard:

```
use std::collections::VecDeque;
```

```
let v = VecDeque::from(vec![1, 2, 3, 4]);
```

BinaryHeap<T>

A `BinaryHeap` est une collection dont les éléments sont organisés de manière lâche afin que la plus grande valeur bouillonne toujours au début de la file d'attente. Voici les trois `BinaryHeap` méthodes les plus couramment utilisées :

`heap.push(value)`

Ajoute une valeur au tas.

`heap.pop()`

Supprime et renvoie la plus grande valeur du tas. Il renvoie un `Option<T>` c'est-à-dire `None` si le tas était vide.

`heap.peek()`

Retourne une référence à la plus grande valeur du tas. Le type de retour est `Option<&T>`.

`heap.peek_mut()`

Retourne `PeekMut<T>`, qui agit comme une référence mutable à la plus grande valeur du tas et fournit la fonction associée au type `pop()` pour extraire cette valeur du tas. En utilisant cette méthode, nous pouvons choisir de sortir ou non du tas en fonction de la valeur maximale :

```
use std::collections::binary_heap::PeekMut;

if let Some(top) = heap.peek_mut() {
    if *top > 10 {
        PeekMut::pop(top);
    }
}
```

`BinaryHeap` prend également en charge un sous-ensemble des méthodes sur `Vec`, y compris `BinaryHeap::new()`, `.len()`, `.is_empty()`, `.capacity()`, `.clear()` et `.append(&mut heap2)`.

Par exemple, supposons que nous remplissions un `BinaryHeap` avec un groupe de nombres :

```
use std::collections::BinaryHeap;
```

```
let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

La valeur 9 est en haut du tas :

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

La suppression de la valeur 9 réorganise également légèrement les autres éléments pour qu'ils 8 soient maintenant au premier plan, et ainsi de suite :

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

Bien sûr, `BinaryHeap` ne se limite pas aux nombres. Il peut contenir n'importe quel type de valeur qui implémente le `Ord` trait intégré.

Cela rend `BinaryHeap` utile comme file d'attente de travail. Vous pouvez définir une structure de tâche qui s'implémente `Ord` sur la base de la priorité afin que les tâches de priorité supérieure soient les tâches `Greater` de priorité inférieure. Ensuite, créez un `BinaryHeap` pour contenir toutes les tâches en attente. Sa `.pop()` méthode renverra toujours l'élément le plus important, la tâche sur laquelle votre programme devrait travailler ensuite.

Remarque : `BinaryHeap` est itérable et possède une `.iter()` méthode, mais les itérateurs produisent les éléments du tas dans un ordre arbitraire, pas du plus grand au moins. Pour consommer les valeurs de a `BinaryHeap` par ordre de priorité, utilisez une `while` boucle:

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

HashMap<K, V> et BTreeMap<K, V>

Une *carte* est une collection de paires clé-valeur (appelées *entrées*). Deux entrées n'ont pas la même clé et les entrées sont organisées de sorte que si vous avez une clé, vous pouvez rechercher efficacement la valeur cor-

respondante dans une carte. En bref, une carte est une table de recherche.

Rust propose deux types de cartes : `HashMap<K, V>` et `BTreeMap<K, V>`. Les deux partagent bon nombre des mêmes méthodes; la différence réside dans la manière dont les deux conservent les entrées organisées pour une recherche rapide.

A `HashMap` stocke les clés et les valeurs dans une table de hachage, il nécessite donc un type de clé `K` qui implémente `Hash` et `Eq`, les traits standard pour le hachage et l'égalité.

[La figure 16-4](#) montre comment a `HashMap` est organisé en mémoire. Les régions plus sombres ne sont pas utilisées. Toutes les clés, valeurs et codes de hachage mis en cache sont stockés dans une seule table allouée par tas. L'ajout d'entrées force éventuellement le `HashMap` à allouer une table plus grande et à y déplacer toutes les données.

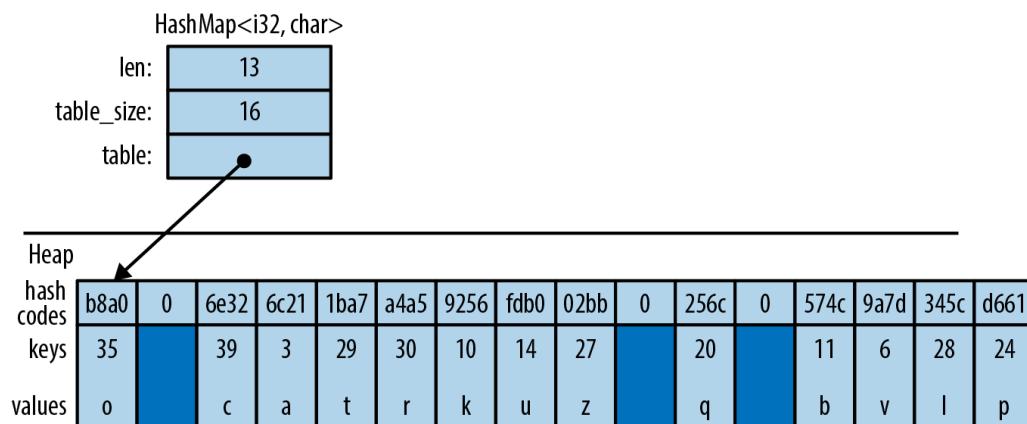


Illustration 16-4. A `HashMap` en mémoire

A `BTreeMap` stocke les entrées dans l'ordre par clé, dans une structure arborescente, il nécessite donc un type de clé `K` qui implémente `Ord`. [La figure 16-5](#) montre un `BTreeMap`. Encore une fois, les régions les plus sombres sont des capacités de réserve inutilisées.

A `BTreeMap` stocke ses entrées dans des *nœuds*. La plupart des nœuds d'un `BTreeMap` ne contiennent que des paires clé-valeur. Les nœuds non feuilles, comme le nœud racine illustré dans cette figure, ont également de la place pour les pointeurs vers les nœuds enfants. Le pointeur entre `(20, 'q')` et `(30, 'r')` pointe vers un nœud enfant contenant des clés entre `20` et `30`. L'ajout d'entrées nécessite souvent de faire glisser certaines des entrées existantes d'un nœud vers la droite, pour les garder triées, et implique parfois l'allocation de nouveaux nœuds.

Cette image est un peu simplifiée pour tenir sur la page. Par exemple, les vrais BTreeMap nœuds ont de la place pour 11 entrées, pas 4.

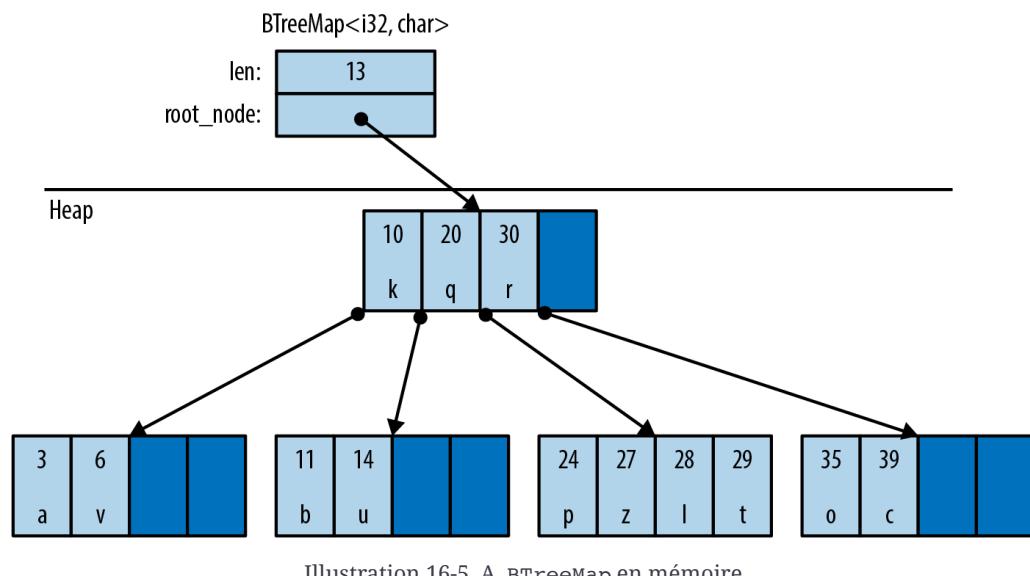


Illustration 16-5. A BTreeMap en mémoire

La bibliothèque standard Rust utilise des arbres B plutôt que des arbres binaires équilibrés car les arbres B sont plus rapides sur le matériel moderne. Un arbre binaire peut utiliser moins de comparaisons par recherche qu'un arbre B, mais la recherche d'un arbre B a une meilleure *localité* - c'est-à-dire que les accès mémoire sont regroupés plutôt que dispersés sur l'ensemble du tas. Cela rend les échecs du cache CPU plus rares. C'est un gain de vitesse significatif.

Il existe plusieurs façons de créer une carte :

`HashMap::new()`, `BTreeMap::new()`

Créez de nouvelles cartes vides.

`iter.collect()`

Boîte à outils utilisé pour créer et remplir une nouvelle `HashMap` ou `BTreeMap` à partir de paires clé-valeur. `iter` doit être un `Iterator<Item=(K, V)>`.

`HashMap::with_capacity(n)`

Crée une nouvelle carte de hachage vide avec de la place pour au moins `n` entrées. `HashMap`s, comme les vecteurs, stockent leurs données dans une seule allocation de tas, ils ont donc une capacité et les méthodes associées `hash_map.capacity()`, `hash_map.reserve(additional)` et `hash_map.shrink_to_fit()`. `BTreeMap` pas.

`HashMap`s et `BTreeMap`s ont les mêmes méthodes de base pour travailler avec des clés et des valeurs :

`map.len()`

Retourne le nombre d'entrées.

`map.is_empty()`

Retour `true` si `map` n'a pas d'entrées.

`map.contains_key(&key)`

Retour `true` si la carte a une entrée pour le donné `key`.

`map.get(&key)`

Recherche `map` une entrée avec le donné `key`. Si une entrée correspondante est trouvée, cela renvoie `Some(r)`, où `r` est une référence à la valeur correspondante. Sinon, cela renvoie `None`.

`map.get_mut(&key)`

Similaire, mais il renvoie une `mut` référence à la valeur.

En général, les cartes vous permettent d'`mut` accéder aux valeurs stockées à l'intérieur, mais pas aux clés. Les valeurs sont à vous de modifier comme bon vous semble. Les clés appartiennent à la carte elle-même ; il doit s'assurer qu'ils ne changent pas, car les entrées sont organisées par leurs clés. Modifier une clé sur place serait un bogue.

`map.insert(key, value)`

Encartsl'entrée `(key, value)` dans `map` et renvoie l'ancienne valeur, le cas échéant. Le type de retour est `Option<V>`. S'il existe déjà une entrée pour `key` dans la carte, la nouvelle entrée `value` écrase l'ancienne.

`map.extend(iterable)`

Itèresur les `(K, V)` éléments de `iterable` et insère chacune de ces paires clé-valeur dans `map`.

`map.append(&mut map2)`

Se déplace toutes les entrées de `map2` dans `map`. Après, `map2` c'est vide.

`map.remove(&key)`

Trouve et supprime toute entrée avec le donné `key` de `map`, renvoyant la valeur supprimée, le cas échéant. Le type de retour est `Option<V>`.

`map.remove_entry(&key)`

Trouve et supprime toute entrée avec le donné `key` de `map`, renvoyant la clé et la valeur supprimées, le cas échéant. Le type de retour est `Option<(K, V)>`.

`map.retain(test)`

Supprime tous les éléments qui ne passent pas le test donné. L'`test` argument est une fonction ou une fermeture qui implémente `FnMut(&K, &mut V) -> bool`. Pour chaque élément de `map`,

cela appelle `test(&key, &mut value)`, et s'il renvoie `false`, l'élément est supprimé de la carte et supprimé.

En dehors de la performance, c'est comme écrire:

```
map = map.into_iter().filter(test).collect();  
  
map.clear()
```

Supprime toutes les entrées.

Une carte peut également être interrogée à l'aide de crochets :

`map[&key]`. Autrement dit, les cartes implémentent le `Index` trait intégré. Cependant, cela panique si l'il n'y a pas déjà une entrée pour le donné `key`, comme un accès au tableau hors limites, donc n'utilisez cette syntaxe que si l'entrée que vous recherchez est sûre d'être remplie.

L' `key` argumentation à `.contains_key()`, `.get()`, `.get_mut()`, et `.remove()` n'a pas besoin d'avoir le type exact `&K`. Ces méthodes sont génériques par rapport aux types qui peuvent être empruntés à `K`. Il est acceptable d'appeler `fish_map.contains_key("conger")` un `HashMap<String, Fish>`, même s'il "conger" ne s'agit pas exactement d'un `String`, car `String` implements `Borrow<&str>`. Pour plus de détails, voir "[Emprunter et EmprunterMut](#)".

Étant donné que a `BTreeMap<K, V>` conserve ses entrées triées par clé, il prend en charge une opération supplémentaire :

```
btree_map.split_off(&key)
```

Se divise `btree_map` en deux. Les entrées avec des clés inférieures à `key` sont laissées dans `btree_map`. Retourne un new `BTreeMap<K, V>` contenant les autres entrées.

Entrées

Les deux `HashMap` et `BTreeMap` ont un `Entry` type correspondant. Le but des entrées est d'éliminer les recherches de carte redondantes. Par exemple, voici un code pour obtenir ou créer un dossier étudiant :

```
// Do we already have a record for this student?  
if !student_map.contains_key(name) {  
    // No: create one.  
    student_map.insert(name.to_string(), Student::new());  
}  
// Now a record definitely exists.
```

```
let record = student_map.get_mut(name).unwrap();  
...
```

Cela fonctionne bien, mais il accède `student_map` deux ou trois fois, en faisant la même recherche à chaque fois.

L'idée avec les entrées est que nous effectuons la recherche une seule fois, produisant une `Entry` valeur qui est ensuite utilisée pour toutes les opérations suivantes. Ce one-liner est équivalent à tout le code précédent, sauf qu'il n'effectue la recherche qu'une seule fois :

```
let record = student_map.entry(name.to_string()).or_insert_with(Student::n
```

La `Entry` valeur renvoyée par

`student_map.entry(name.to_string())` agit comme une référence mutable à un emplacement de la carte qui est soit *occupé par une* paire clé-valeur , soit *vacant* , ce qui signifie qu'il n'y a pas encore d'entrée à cet endroit. S'il est vacant, la `.or_insert_with()` méthode de l'entrée insère un nouveau `Student` . La plupart des utilisations des entrées sont comme ceci : courtes et douces.

Toutes les `Entry` valeurs sont créées par la même méthode :

```
map.entry(key)
```

Retourne un `Entry` pour le donné `key` . S'il n'y a pas une telle clé dans la carte, cela renvoie un vacant `Entry` .

Cette méthode prend son `self` argument par `mut` référence et renvoie un `Entry` avec une durée de vie correspondante :

```
pub fn entry<'a>(&'a mut self, key: K) ->Entry<'a, K, V>
```

Le `Entry` type a un paramètre de durée de vie ' `a` car il s'agit en fait d'une sorte de `mut` référence empruntée à la carte. Tant qu'il `Entry` existe, il a un accès exclusif à la carte.

De retour dans "[Structs Containing References](#)" , nous avons vu comment stocker des références dans un type et comment cela affecte les durées de vie. Nous voyons maintenant à quoi cela ressemble du point de vue de l'utilisateur. C'est ce qui se passe avec `Entry` .

Malheureusement, il n'est pas possible de passer une référence de type `&str` à cette méthode si la carte a des `String` clés. La `.entry()` méthode, dans ce cas, nécessite un réel `String`.

Entry fournissent trois méthodes pour traiter les entrées vacantes :

```
map.entry(key).or_insert(value)
```

Assure que `map` contient une entrée avec le donné `key`, en insérant une nouvelle entrée avec le donné `value` si nécessaire. Il renvoie une `mut` référence à la valeur nouvelle ou existante.

Supposons que nous ayons besoin de compter les votes. Nous pouvons écrire:

```
let mut vote_counts: HashMap<String, usize> = HashMap::new();
for name in ballots {
    let count = vote_counts.entry(name).or_insert(0);
    *count += 1;
}
```

`.or_insert()` renvoie une `mut` référence, donc le type de `count` est `&mut usize`.

```
map.entry(key).or_default()
```

Assure que `map` contient une entrée avec la clé donnée, en insérant une nouvelle entrée avec la valeur renvoyée par `Default::default()` si nécessaire. Cela ne fonctionne que pour les types qui implémentent `Default`. Comme `or_insert`, cette méthode renvoie une `mut` référence à la valeur nouvelle ou existante.

```
map.entry(key).or_insert_with(default_fn)
```

Cette est identique, sauf que s'il doit créer une nouvelle entrée, il appelle `default_fn()` pour produire la valeur par défaut. S'il existe déjà une entrée pour `key` dans le `map`, then `default_fn` n'est pas utilisé.

Supposons que nous voulions savoir quels mots apparaissent dans quels fichiers. Nous pouvons écrire:

```
// This map contains, for each word, the set of files it appears in.
let mut word_occurrence: HashMap<String, HashSet<String>> =
    HashMap::new();
for file in files {
    for word in read_words(file)? {
        let set = word_occurrence
```

```

        .entry(word)
        .or_insert_with(HashSet::new);
    set.insert(file.clone());
}
}

```

Entry fournit également un moyen pratique de modifier uniquement les champs existants.

```
map.entry(key).and_modify(closure)
```

Appelle `closure` si une entrée avec la clé `key` existe, en passant une référence mutable à la valeur. Il renvoie le `Entry`, il peut donc être enchaîné avec d'autres méthodes.

Par exemple, nous pourrions l'utiliser pour compter le nombre d'occurrences de mots dans une chaîne :

```

// This map contains all the words in a given string,
// along with the number of times they occur.
let mut word_frequency: HashMap<&str, u32> = HashMap::new();
for c in text.split_whitespace() {
    word_frequency.entry(c)
        .and_modify(|count| *count += 1)
        .or_insert(1);
}

```

Le `Entry` type est une énumération, définie comme ceci pour `HashMap` (et de la même manière pour `BTreeMap`):

```

// (in std::collections::hash_map)
pub enum Entry<'a, K, V> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>)
}

```

Les types `OccupiedEntry` et `VacantEntry` ont des méthodes pour insérer, supprimer et accéder aux entrées sans répéter la recherche initiale. Vous pouvez les trouver dans la documentation en ligne. Les méthodes supplémentaires peuvent parfois être utilisées pour éliminer une recherche redondante ou deux, mais `.or_insert()` et `.or_insert_with()` couvrir les cas courants.

Itération de carte

Il existe plusieurs façons d'itérer sur une carte :

- L'itération par valeur (`for (k, v) in map`) produit des (`K, V`) paires. Cela consomme la carte.
- L'itération sur une référence partagée (`for (k, v) in &map`) produit des (`&K, &V`) paires.
- L'itération sur une `mut` référence (`for (k, v) in &mut map`) produit des (`&K, &mut V`) paires. (Encore une fois, il n'y a aucun moyen d'`mut` accéder aux clés stockées dans une carte, car les entrées sont organisées par leurs clés.)

Comme les vecteurs, les cartes ont `.iter()` et `.iter_mut()` les méthodes qui renvoient des itérateurs par référence, tout comme l'itération sur `&map` ou `&mut map`. En outre:

`map.keys()`

Retourne un itérateur sur les clés seulement, par référence.

`map.values()`

Retourne un itérateur sur les valeurs, par référence.

`map.values_mut()`

Retourne un itérateur sur les valeurs, par `mut` référence.

`map.into_iter(), map.into_keys(), map.into_values()`

Consommez la carte, renvoyant un itérateur sur des tuples (`K, V`) de clés et de valeurs, de clés ou de valeurs, respectivement.

Tous les `HashMap` itérateurs visitent les entrées de la carte dans un ordre arbitraire. `BTreeMap` les itérateurs les visitent dans l'ordre par clé.

HashSet<T> et BTreeSet<T>

Ensembles sont des collections de valeurs organisées pour un test d'adhésion rapide :

```
let b1 = large_vector.contains(&"needle");      // slow, checks every element
let b2 = large_hash_set.contains(&"needle");    // fast, hash lookup
```

Un ensemble ne contient jamais plusieurs copies de la même valeur.

Les cartes et les ensembles ont des méthodes différentes, mais dans les coulisses, un ensemble est comme une carte avec uniquement des clés, plutôt que des paires clé-valeur. En fait, les deux types d'ensembles de

Rust, `HashSet<T>` et `BTreeSet<T>`, sont implémentés comme des enveloppes minces autour de `HashMap<T, ()>` et `BTreeMap<T, ()>`.

`HashSet::new()`, `BTreeSet::new()`

Créer nouveaux ensembles.

`iter.collect()`

Boîte à utilisation pour créer un nouvel ensemble à partir de n'importe quel itérateur. Si `iter` produit des valeurs plus d'une fois, les doublons sont supprimés.

`HashSet::with_capacity(n)`

Créer un vide `HashSet` avec de la place pour au moins `n` des valeurs.

`HashSet<T>` et `BTreeSet<T>` ont en commun toutes les méthodes de base :

`set.len()`

Retourne le nombre de valeurs dans `set`.

`set.is_empty()`

Retourne `true` si l'ensemble ne contient aucun élément.

`set.contains(&value)`

Retourne `true` si l'ensemble contient le donné `value`.

`set.insert(value)`

Ajoute une `value` à l'ensemble. Renvoie `true` si une valeur a été ajoutée, `false` si elle faisait déjà partie de l'ensemble.

`set.remove(&value)`

Supprime une `value` de l'ensemble. Renvoie `true` si une valeur a été supprimée, `false` si elle n'était déjà pas membre de l'ensemble.

`set.retain(test)`

Supprime tous les éléments qui ne passent pas le test donné. L'argument `test` est une fonction ou une fermeture qui implémente `FnMut(&T) -> bool`. Pour chaque élément de `set`, cela appelle `test(&value)`, et s'il renvoie `false`, l'élément est supprimé de l'ensemble et supprimé.

En dehors de la performance, c'est comme écrire :

```
set = set.into_iter().filter(test).collect();
```

Comme pour les cartes, les méthodes qui recherchent une valeur par référence sont génériques par rapport aux types qui peuvent être empruntés.

tés à `T`. Pour plus de détails, voir "[Emprunter et EmprunterMut](#)" .

Définir l'itération

Il y a deux façons d'itérer sur ensembles :

- L'itération par valeur (" `for v in set`") produit les membres de l'ensemble (et consomme l'ensemble).
- L'itération par référence partagée (" `for v in &set`") produit des références partagées aux membres de l'ensemble.

L'itération sur un ensemble par `mut` référence n'est pas prise en charge.

Il n'y a aucun moyen d'obtenir une `mut` référence à une valeur stockée dans un ensemble.

```
set.iterator()
```

Retourne un itérateur sur les membres de `set` par référence.

`HashSet` les itérateurs, comme les `HashMap` itérateurs, produisent leurs valeurs dans un ordre arbitraire. `BTreeSet` les itérateurs produisent des valeurs dans l'ordre, comme un vecteur trié.

Lorsque des valeurs égales sont différentes

Les ensembles ont quelques méthodes étranges que vous devez utiliser uniquement si vous vous souciez des différences entre les valeurs "égales".

De telles différences existent souvent. Deux `String` valeurs identiques, par exemple, stockent leurs caractères à des emplacements différents en mémoire :

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str); // 0x7f8b32060008
println!("{:p}", &s2 as &str); // 0x7f8b32060010
```

D'habitude, on s'en fout.

Mais au cas où vous le feriez, vous pouvez accéder aux valeurs réelles stockées dans un ensemble en utilisant les méthodes suivantes. Chacun renvoie une valeur `Option` si `None` elle `set` ne contient pas de valeur correspondante :

```
set.get(&value)
```

Retourne une référence partagée au membre de `set` qui est égale à `value`, le cas échéant. Renvoie un `Option<&T>`.

```
set.take(&value)
```

Comme `set.remove(&value)`, mais il renvoie la valeur supprimée, le cas échéant. Renvoie un `Option<T>`.

```
set.replace(value)
```

Comme `set.insert(value)`, mais si `set` contient déjà une valeur égale à `value`, cela remplace et renvoie l'ancienne valeur. Renvoie un `Option<T>`.

Opérations sur tout l'ensemble

Jusqu'à présent, la plupart des méthodes d'ensemble que nous avons vues se concentraient sur une seule valeur dans un seul ensemble. Les ensembles ont également des méthodes qui fonctionnent sur l'ensemble d'ensembles :

```
set1.intersection(&set2)
```

Retourne un itérateur sur toutes les valeurs qui sont à la fois dans `set1` et `set2`.

Par exemple, si nous voulons imprimer les noms de tous les étudiants qui suivent à la fois des cours de chirurgie cérébrale et de science des fusées, nous pourrions écrire :

```
for student in &brain_class {
    if rocket_class.contains(student) {
        println!("{}" , student);
    }
}
```

Ou, plus court :

```
for student in brain_class.intersection(&rocket_class) {
    println!("{}" , student);
}
```

Étonnamment, il y a un opérateur pour cela.

`&set1 & &set2` renvoie un nouvel ensemble qui est l'intersection de `set1` et `set2`. Il s'agit de l'opérateur AND binaire au niveau du

bit, appliqu    deux r  f  rence  s. Cela trouve les valeurs qui sont   la fois dans `set1` et `set2` :

```
let overachievers = &brain_class & &rocket_class;

set1.union(&set2)
```

Retourne un it  rateur sur les valeurs qui sont dans `set1` ou `set2`, ou les deux.

`&set1 | &set2` renvoie un nouvel ensemble contenant toutes ces valeurs. Il trouve les valeurs qui sont dans `set1` ou `set2`.

```
set1.difference(&set2)
```

Retourne un it  rateur sur les valeurs qui sont dans `set1` mais pas dans `set2`.

`&set1 - &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

```
set1.symmetric_difference(&set2)
```

Retourne un it  rateur sur les valeurs qui sont dans `set1` ou `set2`, mais pas les deux.

`&set1 ^ &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

Et il existe trois m  th  mes pour tester les relations entre les ensembles :

```
set1.is_disjoint(set2)
```

Vraie si `set1` et `set2` n'ont pas de valeurs en commun — l'intersection entre elles est vide.

```
set1.is_subset(set2)
```

Vraie si `set1` est un sous-ensemble de `set2` — c'est-  dire que toutes les valeurs de `set1` sont   galement dans `set2`.

```
set1.is_superset(set2)
```

Cette est l'inverse : c'est vrai si `set1` est un sur-ensemble de `set2`.

Les ensembles prennent   galement en charge les tests d'  galit  avec `==` et `!=` ; deux ensembles sont   gaux s'ils contiennent les m  mes valeurs.

Hachage

`std::hash::Hash` est le trait de bibliothèque standard pour hashables types. `HashMap` les clés et `HashSet` les éléments doivent implémenter à la fois `Hash` et `Eq`.

La plupart des types intégrés qui mettent en œuvre `Eq` également mettent en œuvre `Hash`. Les types entiers, `char`, et `String` sont tous hachables ; il en va de même pour les tuples, les tableaux, les tranches et les vecteurs, tant que leurs éléments sont hachables.

Un principe de la bibliothèque standard est qu'une valeur doit avoir le même code de hachage, quel que soit l'endroit où vous la stockez ou la manière dont vous la pointez. Par conséquent, une référence a le même code de hachage que la valeur à laquelle elle se réfère, et a `Box` a le même code de hachage que la valeur encadrée. Un vecteur `Vec` a le même code de hachage que la tranche contenant toutes ses données, `&vec[..]`. A `String` a le même code de hachage que a `&str` avec les mêmes caractères.

Structures et énumérations ne pas implémenter `Hash` par défaut, mais une implémentation peut être dérivée :

```
// The ID number for an object in the British Museum's collection.
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

Cela fonctionne tant que les champs du type sont tous hachables.

Si vous implémentez `PartialEq` à la main pour un type, vous devez également implémenter `Hash` à la main. Par exemple, supposons que nous ayons un type qui représente des trésors historiques inestimables :

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
    ...
}
```

Deux `Artifact`s sont considérés comme égaux s'ils ont le même ID :

```

impl PartialEq for Artifact {
    fn eq(&self, other: &Artifact) ->bool {
        self.id == other.id
    }
}

impl Eq for Artifact {}

```

Comme nous comparons les artefacts uniquement sur la base de leur ID, nous devons les hacher de la même manière :

```

use std::hash::{Hash, Hasher};

impl Hash for Artifact {
    fn hash<H: Hasher>(&self, hasher:&mut H) {
        // Delegate hashing to the MuseumNumber.
        self.id.hash(hasher);
    }
}

```

(Sinon, `HashSet<Artifact>` ne fonctionnerait pas correctement ; comme toutes les tables de hachage, il nécessite que `hash(a) == hash(b)` if `a == b`.)

Cela nous permet de créer un `HashSet` de `Artifacts` :

```
let mut collection = HashSet::<Artifact>::new();
```

Comme le montre ce code, même lorsque vous implémentez `Hash` à la main, vous n'avez pas besoin de savoir quoi que ce soit sur les algorithmes de hachage. `.hash()` reçoit une référence à un `Hasher`, qui représente l'algorithme de hachage. Vous n'avez `Hasher` qu'à lui fournir toutes les données pertinentes pour l'`==` opérateur. Le `Hasher` calcule un code de hachage à partir de tout ce que vous lui donnez.

Utilisation d'un algorithme de hachage personnalisé

La `hash` méthode est générique, de sorte que les `Hash` implémentations présentées précédemment peuvent fournir des données à tout type qui implémente `Hasher`. C'est ainsi que Rust prend en charge les algorithmes de hachage enfichables.

Un troisième trait, `std::hash::BuildHasher`, est le trait pour les types qui représentent l'état initial d'un algorithme de hachage. Chacun `Hasher` est à usage unique, comme un itérateur : vous l'utilisez une fois et le jetez. A `BuildHasher` est réutilisable.

Chaque `HashMap` contient un `BuildHasher` qu'il utilise chaque fois qu'il a besoin de calculer un code de hachage. La `BuildHasher` valeur contient la clé, l'état initial ou d'autres paramètres dont l'algorithme de hachage a besoin à chaque exécution.

Le protocole complet pour calculer un code de hachage ressemble à ceci :

```
use std:: hash::{Hash, Hasher, BuildHasher};\n\nfn compute_hash<B, T>(builder: &B, value: &T) -> u64\n    where B: BuildHasher, T:Hash\n{\n    let mut hasher = builder.build_hasher(); // 1. start the algorithm\n    value.hash(&mut hasher); // 2. feed it data\n    hasher.finish() // 3. finish, producing a u64\n}
```

`HashMap` appelle ces trois méthodes à chaque fois qu'il a besoin de calculer un code de hachage. Toutes les méthodes sont inlineables, donc c'est très rapide.

L'algorithme de hachage par défaut de Rust est un algorithme bien connu appelé SipHash-1-3. SipHash est rapide et très efficace pour minimiser les collisions de hachage. En fait, il s'agit d'un algorithme cryptographique : il n'existe aucun moyen efficace connu de générer des collisions SipHash-1-3. Tant qu'une clé différente et imprévisible est utilisée pour chaque table de hachage, Rust est protégé contre une sorte d'attaque par déni de service appelée HashDoS, où les attaquants utilisent délibérément des collisions de hachage pour déclencher les pires performances sur un serveur.

Mais peut-être que vous n'en avez pas besoin pour votre application. Si vous stockez de nombreuses petites clés, telles que des entiers ou des chaînes très courtes, il est possible d'implémenter une fonction de hachage plus rapide, au détriment de la sécurité HashDoS. La `fnv` caisse implémente un tel algorithme, le hachage Fowler – Noll – Vo (FNV). Pour l'essayer, ajoutez cette ligne à votre `Cargo.toml` :

```
[dépendances]\nfnv = "1.0"
```

Importez ensuite la carte et définissez les types à partir de `fnv` :

```
use fnv:::{FnvHashMap, FnvHashSet};
```

Vous pouvez utiliser ces deux types en remplacement de `HashMap` et `HashSet`. Un coup d'œil dans le `fnv` code source révèle comment ils sont définis :

```
/// A `HashMap` using a default FNV hasher.  
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;  
  
/// A `HashSet` using a default FNV hasher.  
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

La norme `HashMap` et `HashSet` les collections acceptent un paramètre de type supplémentaire facultatif spécifiant l'algorithme de hachage ; `FnvHashMap` et `FnvHashSet` sont des alias de type génériques pour `HashMap` et `HashSet`, spécifiant un hachage FNV pour ce paramètre.

Au-delà des collections standard

Créer un nouveau, personnaliséLe type de collection dans Rust est à peu près le même que dans n'importe quel autre langage. Vous organisez les données en combinant les parties fournies par le langage : structures et énumérations, collections standard, `Option`s, `Box`es, etc. Pour un exemple, voir le `BinaryTree<T>` type défini dans "[Generic Enums](#)" .

Si vous avez l'habitude d'implémenter des structures de données en C++, en utilisant des pointeurs bruts, une gestion manuelle de la mémoire, un placement `new` et des appels de destructeur explicites pour obtenir les meilleures performances possibles, vous trouverez sans aucun doute Safe Rust plutôt limitant. Tous ces outils sont intrinsèquement dangereux. Ils sont disponibles dans Rust, mais uniquement si vous optez pour un code non sécurisé. [Le chapitre 22](#) montre comment ; il inclut un exemple qui utilise du code non sécurisé pour implémenter une collection personnalisée sécurisée.

Pour l'instant, nous nous contenterons de profiter de la lueur chaleureuse des collections standard et de leurs API sûres et efficaces. Comme une grande partie de la bibliothèque standard Rust, ils sont conçus pour garantir que le besoin d'écrire `unsafe` est aussi rare que possible.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 17. Chaînes et texte

La chaîne est une structure de données rigide et partout où elle est transmise, il y a beaucoup de duplication de processus. C'est un véhicule parfait pour cacher des informations.

—Alan Perlis, épigramme #34

Nous avons utilisé le texte principal de Rusttypes, `String`, `str`, et `char`, tout au long du livre. Dans "[Types de chaînes](#)", nous avons décrit la syntaxe des littéraux de caractères et de chaînes et montré comment les chaînes sont représentées en mémoire. Dans ce chapitre, nous couvrons plus en détail la gestion de texte.

Dans ce chapitre:

- Nous vous donnons quelques informations sur Unicode qui devraient vous aider à comprendre la conception de la bibliothèque standard.
- Nous décrivons le `char` type, représentant un seul point de code Unicode.
- Nous décrivons les types `String` et `str`, représentant des séquences de caractères Unicode possédées et empruntées. `str` Ceux-ci ont une grande variété de méthodes pour construire, rechercher, modifier et itérer sur leur contenu.
- Nous couvrons les fonctionnalités de formatage de chaîne de Rust, comme les macros `println!` et `.format!` Vous pouvez écrire vos propres macros qui fonctionnent avec des chaînes de formatage et les étendre pour prendre en charge vos propres types.
- Nous donnons un aperçu du support des expressions régulières de Rust.
- Enfin, nous expliquons pourquoi la normalisation Unicode est importante et montrons comment le faire dans Rust.

Quelques arrière-plans Unicode

Ce livre concerne Rust, pas Unicode, qui a déjà des livres entiers qui lui sont consacrés. Mais les types de caractères et de chaînes de Rust sont conçus autour d'Unicode. Voici quelques extraits d'Unicode qui aident à expliquer Rust.

ASCII, Latin-1 et Unicode

Unicode et ASCII match pour tous les points de code ASCII, de 0 à 0x7f : par exemple, les deux attribuent au caractère * le point de code 42 . De même, Unicode attribue aux mêmes caractères que le jeu 0 de 0xff caractères ISO / CEI 8859-1, un sur-ensemble de huit bits d'ASCII à utiliser avec les langues d'Europe occidentale. Unicode appelle cette plage de points de code le *bloc de code Latin-1*, nous désignerons donc ISO/IEC 8859-1 par le nom plus évocateur *Latin-1* .

Comme Unicode est un sur-ensemble de Latin-1, la conversion de Latin-1 en Unicode ne nécessite même pas de table :

```
fn latin1_to_char(latin1: u8) ->char {  
    latin1 as char  
}
```

La conversion inverse est également triviale, en supposant que les points de code se situent dans la plage Latin-1 :

```
fn char_to_latin1(c: char) ->Option<u8> {  
    if c as u32 <= 0xff {  
        Some(c as u8)  
    } else {  
        None  
    }  
}
```

UTF-8

La rouille `String` et `str` les types représentent du texte en utilisant l'encodage UTF-8 formulaire. UTF-8 encode un caractère sous la forme d'une séquence de un à quatre octets ([Figure 17-1](#)).

UTF-8 encoding (one to four bytes long)	Code Point Represented	Range
0 x x x x x x	0xxxxxxx	0 to 0x7f
1 1 0 x x x x 1 0 y y y y y y	0bxxxxxxxxyyyyy	0x80 to 0x7ff
1 1 1 0 x x x 1 0 y y y y y 1 0 z z z z z z	0bxxxxxxxxyyzzzzzz	0x800 to 0xffff
1 1 1 1 0 x x x 1 0 y y y y y 1 0 z z z z z z 1 0 w w w w w w w w	0bxxxxxxxxyyzzzzzzwwwwww	0x10000 to 0x10ffff

Illustration 17-1. L'encodage UTF-8

Il existe deux restrictions sur les séquences UTF-8 bien formées. Premièrement, seul le codage le plus court pour un point de code donné est considéré comme bien formé ; vous ne pouvez pas passer quatre octets à encoder un point de code qui tiendrait dans trois. Cette règle garantit qu'il

existe exactement un encodage UTF-8 pour un point de code donné. Deuxièmement, UTF-8 bien formé ne doit pas encoder les nombres de bout en `0xd800` bout `0xffff` ou au-delà `0x10ffff` : ceux-ci sont soit réservés à des fins autres que les caractères, soit entièrement en dehors de la plage d'Unicode.

[La figure 17-2](#) montre quelques exemples.

UTF-8 encoding (one to four bytes long)	Code Point Represented	Range
<code>0 0 1 0 1 0 1 0</code>	<code>0b0101010 == 0x2a</code>	<code>'*</code>
<code>1 1 0 0 1 1 1 0 1 0 1 1 1 0 0</code>	<code>0b01110_111100 == 0x3bc</code>	<code>'μ'</code>
<code>1 1 1 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0</code>	<code>0b1001_001100_000110 == 0x9306</code>	<code>'鏽' (sabi: rust)</code>
<code>1 1 1 1 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0</code>	<code>0b000_011111_100110_000000 == 0x1f980</code>	<code>'🦀' (crab emoji)</code>

Illustration 17-2. Exemples UTF-8

Notez que, même si l'emoji crabe a un encodage dont l'octet de tête ne contribue que par des zéros au point de code, il a toujours besoin d'un encodage à quatre octets : les encodages UTF-8 à trois octets ne peuvent transmettre que des points de code de 16 bits, et `0x1f980` est de 17 peu de temps.

Voici un exemple rapide d'une chaîne contenant des caractères avec des encodages de longueurs variables :

```
assert_eq!( "うどん: udon".as_bytes(),
    &[ 0xe3, 0x81, 0x86, // う
        0xe3, 0x81, 0xa9, // ど
        0xe3, 0x82, 0x93, // ん
        0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
    ]);
```

[La figure 17-2](#) montre également quelques propriétés très utiles d'UTF-8 :

- Étant donné que UTF-8 encode les points de code 0 à travers `0x7f` comme rien de plus que les octets 0 à `0x7f`, une plage d'octets contenant du texte ASCII est UTF-8 valide. Et si une chaîne UTF-8 ne comprend que des caractères ASCII, l'inverse est également vrai : l'encodage UTF-8 est un ASCII valide.
Il n'en va pas de même pour Latin-1 : par exemple, Latin-1 encode é comme l'octet `0xe9`, ce que UTF-8 interpréterait comme le premier octet d'un encodage à trois octets.
- En regardant les bits supérieurs de n'importe quel octet, vous pouvez immédiatement dire s'il s'agit du début de l'encodage UTF-8 d'un ca-

ractère ou d'un octet au milieu d'un.

- Seul le premier octet d'un encodage vous indique la longueur totale de l'encodage, via ses bits de tête.
- Étant donné qu'aucun encodage ne dépasse quatre octets, le traitement UTF-8 ne nécessite jamais de boucles illimitées, ce qui est agréable lorsque vous travaillez avec des données non fiables.
- En UTF-8 bien formé, vous pouvez toujours dire sans ambiguïté où commence et se termine l'encodage des caractères, même si vous partez d'un point arbitraire au milieu des octets. Les premiers octets UTF-8 et les octets suivants sont toujours distincts, de sorte qu'un encodage ne peut pas commencer au milieu d'un autre. Le premier octet détermine la longueur totale de l'encodage, donc aucun encodage ne peut être le préfixe d'un autre. Cela a beaucoup de belles conséquences. Par exemple, la recherche d'un caractère délimiteur ASCII dans une chaîne UTF-8 ne nécessite qu'un simple balayage de l'octet du délimiteur. Il ne peut jamais apparaître comme une partie d'un encodage multi-octets, il n'est donc pas du tout nécessaire de suivre la structure UTF-8. De même, les algorithmes qui recherchent une chaîne d'octets dans une autre fonctionneront sans modification sur les chaînes UTF-8, même si certains n'examinent même pas chaque octet du texte recherché.

Bien que les encodages à largeur variable soient plus compliqués que les encodages à largeur fixe, ces caractéristiques rendent UTF-8 plus confortable à utiliser que vous ne le pensez. La bibliothèque standard gère la plupart des aspects pour vous.

Directionnalité du texte

Alors que les scripts comme le latin, le cyrillique et le thaï sont écrits de gauche à droite, d'autres scripts comme l'hébreu et l'arabe sont écrits de droite à gauche. Unicode stocke les caractères dans l'ordre dans lequel ils seraient normalement écrits ou lus, de sorte que les octets initiaux d'une chaîne contenant, par exemple, du texte hébreu encodent le caractère qui serait écrit à droite:

```
assert_eq!( "בָּרָא בְּרָא" .chars().next() , Some( ' ב' ));
```

Caractères (char)

Un Rust `char` est une valeur 32 bit tenant un point de code Unicode. A `char` est assuré de se situer dans la plage de 0 à 0xd7ff ou dans la plage

`0xe000` de `0x10ffff` ; toutes les méthodes de création et de manipulation de `char` valeurs garantissent que cela est vrai. Le `char` type implémente `Copy` and `Clone` , ainsi que tous les traits habituels de comparaison, de hachage et de formatage.

Une tranche de chaîne peut produire un itérateur sur ses caractères avec `slice.chars()` :

```
assert_eq!("力二".chars().next(), Some('力'));
```

Dans les descriptions qui suivent, la variable `ch` est toujours de type `char` .

Classer les caractères

Le `char` type a des méthodes pour classer caractères dans quelques catégories courantes, comme indiqué dans le [Tableau 17-1](#) . Ceux-ci tirent tous leurs définitions d'Unicode.

Tableau 17-1. Méthodes de classification pour le `char` type

Méthode	La description	Exemples
<code>ch.is_numberic()</code>	Un caractère numérique. Cela inclut les catégories générales Unicode « Nombre ; chiffre » et « Nombre ; lettre » mais pas « Chiffre ; autre».	'4'.is_numberic() '\u{A0}'.is_numberic() '\u{8}'.is_numberic() '\u{0}'.is_numberic()
<code>ch.is_alphatic()</code>	Un caractère alphabétique : la propriété dérivée "Alphabétique" d'Unicode.	'q'.is_alphatic() '七'.is_alphatic() '\u{F0}'.is_alphatic()
<code>ch.is_alphanumeric()</code>	Soit numérique, soit alphabétique, comme défini précédemment.	'9'.is_alphanumeric() '\u{F0}'.is_alphanumeric() '*'.is_alphanumeric() '\u{F0}'.is_alphanumeric()
<code>ch.is_whitespace()</code>	Un caractère d'espacement : propriété de caractère Unicode "WSpace=Y".	' '.is_whitespace() \n'.is_whitespace() \u{A0}'.is_whitespace() \u{0}'.is_whitespace()

Méthode	La description	Exemples
<code>ch.is_control()</code>	Un caractère de contrôle : la catégorie générale "Autre, contrôle" d'Unicode.	<code>'\n'.is_control()</code>
		<code>'\u{85}'.is_control()</code>

Un ensemble parallèle de méthodes se limite à ASCII uniquement, retournant `false` pour tout non-ASCII `char` ([Tableau 17-2](#)).

Tableau 17-2. Méthodes de classification ASCII pour char

Méthode	La description	Exemples
ch.is_ascii()	Un caractère ASCII : un dont le point de code se situe entre 0 et 127 inclus.	'n'.is_ascii() i() ! 'ñ'.is_ascii()
ch.is_alphatic()	Une lettre ASCII majuscule ou minuscule, dans la plage 'A'..='z' ou 'a'..='z' .	'n'.is_alpha() i_alpha() c() ! 'l'.is_alpha() i_alpha() ic() ! 'ñ'.is_alpha() i_alpha() ic()
ch.is_digital()	Un chiffre ASCII, dans la plage '0'.. ='9' .	'8'.is_alpha() i_alpha() !'- .is_alpha() ! '⑧'.is_alpha() c_alpha()
ch.is_hexdigit()	N'importe quel caractère dans les plages '0'..='9' , 'A'..='F' ou 'a'..='f' .	
ch.is_alphanumeric()	Un chiffre ASCII ou une lettre majuscule ou minuscule.	'q'.is_alpha() i_alpha() ric() '0'.is_alpha() i_alpha() ric()

Méthode	La description	Exemples
<code>ch.is_ascii_c ontrol()</code>	Un caractère de contrôle ASCII, y compris 'DEL'.	<code>'\n'.is_ascii_control()</code> <code>'\x7f'.is_ascii_control()</code>
<code>ch.is_ascii_g raphic()</code>	Tout caractère ASCII qui laisse de l'encre sur la page : ni un espace ni un caractère de contrôle.	<code>'Q'.is_ascii_graphic()</code> <code>'~'.is_ascii_graphic()</code> <code>!'.is_ascii_graphic()</code>
<code>ch.is_ascii_u ppercase(se(), ch.is_ascii_l owercase(se())</code>	Lettres majuscules et minuscules ASCII.	<code>'z'.is_ascii_lowercase()</code> <code>'Z'.is_ascii_uppercase()</code>
<code>ch.is_ascii_p unctuation()</code>	Tout caractère graphique ASCII qui n'est ni alphabétique ni un chiffre.	
<code>ch.is_ascii_w hitespace()</code>	Un caractère d'espacement ASCII : un espace, une tabulation horizontale, un saut de ligne, un saut de page ou un retour chariot.	<code>' '.is_ascii_whitespace()</code> <code>'\n'.is_ascii_whitespace()</code> <code>!'\u{A0}'.is_ascii_whitespace()</code>

Toutes les `is_ascii_...` méthodes sont également disponibles sur le `u8` type d'octet :

```
assert!(32u8.is_ascii_whitespace());  
assert!(b'9'.is_ascii_digit());
```

Faites attention lorsque vous utilisez ces fonctions pour implémenter une spécification existante comme une norme de langage de programmation ou un format de fichier, car les classifications peuvent différer de manière surprenante. Par exemple, notez que `is whitespace` et `is_ascii_whitespace` diffèrent dans leur traitement de certains caractères :

```
let line_tab = '\u{000b}'; // 'line tab', AKA 'vertical tab'  
assert_eq!(line_tab.is_whitespace(), true);  
assert_eq!(line_tab.is_ascii_whitespace(), false);
```

La `char::is_ascii_whitespace` fonction implémente une définition d'espace blanc commune à de nombreuses normes Web, alors que `char::is whitespace` suit la norme Unicode.

Manipulation des chiffres

Pour manipuler les chiffres, vous pouvez utiliser les méthodes suivantes :

`ch.to_digit(radix)`

Décide si `ch` est un chiffre en base `radix`. Si c'est le cas, il renvoie `Some(num)`, où `num` est un `u32`. Sinon, ça revient `None`. Ceci ne reconnaît que l'ASCII chiffres, et non la classe plus large de caractères couverte par `char::is_numeric`. Le `radix` paramètre peut aller de 2 à 36. Pour les bases supérieures à 10, les lettres ASCII des deux cas sont considérées comme des chiffres avec des valeurs comprises entre 10 et 35.

`std::char::from_digit(num, radix)`

Une fonction gratuite qui convertit la `u32` valeur numérique `num` en un `char` si possible. Si `num` peut être représenté par un seul chiffre dans `radix`, `from_digit` renvoie `Some(ch)`, où `ch` est le chiffre. Lorsque `radix` est supérieur à 10, `ch` il peut s'agir d'une lettre minuscule. Sinon, ça revient `None`.

C'est l'inverse de `to_digit`. Si `std::char::from_digit(num, radix)` est `Some(ch)`, alors `ch.to_digit(radix)` est

`Some(num)`. Si `ch` est un chiffre ASCII ou une lettre minuscule, l'inverse est également vrai.

`ch.is_digit(radix)`

Retour `true` si `ch` est un chiffre ASCII en base `radix`. Cela équivaut à `ch.to_digit(radix) != None`.

Ainsi, par exemple :

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

Conversion de casse pour les caractères

Pour gérer la casse des caractères:

`ch.is_lowercase()`, `ch.is_uppercase()`

Indiquent si `ch` s'agit d'un caractère alphabétique minuscule ou majuscule. Celles-ci suivent les propriétés dérivées des minuscules et des majuscules d'Unicode, elles couvrent donc les alphabets non latins comme le grec et le cyrillique et donnent également les résultats attendus pour l'ASCII.

`ch.to_lowercase()`, `ch.to_uppercase()`

Revenir à l'itérateur qui produisent les caractères des équivalents minuscules et majuscules de `ch`, selon les algorithmes de conversion de cas par défaut Unicode :

```
let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);
```

Ces méthodes renvoient un itérateur au lieu d'un seul caractère car la conversion de casse en Unicode n'est pas toujours un processus un à un :

```
// The uppercase form of the German letter "sharp S" is "SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Unicode says to lowercase Turkish dotted capital 'İ' to 'i'
// followed by ``\u{307}``, COMBINING DOT ABOVE, so that a
```

```
// subsequent conversion back to uppercase preserves the dot.
let ch = 'i'; // ``\u{130}``
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);
```

Pour plus de commodité, ces itérateurs implémentent le `std::fmt::Display` trait, vous pouvez donc les transmettre directement à une macro `println!` ou `.write!`

Conversions vers et depuis des entiers

Le `as` Opérateur de Rust convertira un `char` en n'importe quel entier que vous tapez, masquant silencieusement tous les bits supérieurs :

```
assert_eq!('B' as u32, 66);
assert_eq!('𩿱' as u8, 66);    // upper bits truncated
assert_eq!('二' as i8, -116); // same
```

Le `as` opérateur convertira n'importe quelle `u8` valeur en un `char`, et `char` implementera `From<u8>` également, mais des types entiers plus larges peuvent représenter des points de code invalides, donc pour ceux que vous devez utiliser `std::char::from_u32`, qui renvoie `Option<char>`:

```
assert_eq!(char:: from(66), 'B');
assert_eq!(std:: char:: from_u32(0x9942), Some('𩿱'));
assert_eq!(std:: char::from_u32(0xd800), None); // reserved for UTF-16
```

Chaîne et chaîne

Rouille `String` et `str` typessont garantis pour tenir seulement bien formé UTF-8. La bibliothèque garantit cela en limitant les façons dont vous pouvez créer des valeurs `String` et `str` les opérations que vous pouvez effectuer sur celles-ci, de sorte que les valeurs soient bien formées lors de leur introduction et le restent lorsque vous travaillez avec elles. Toutes leurs méthodes protègent cette garantie : aucune opération sûre sur elles ne peut introduire de l'UTF-8 mal formé. Cela simplifie le code qui fonctionne avec le texte.

Rust place les méthodes de gestion de texte sur `str` ou `String` selon que la méthode nécessite un tampon redimensionnable ou se contente d'utiliser le texte en place. Depuis `String` les déréférences à `&str`, chaque méthode définie sur `str` est également directement disponible sur `String`. Cette section présente les méthodes des deux types, regroupées par fonction approximative.

Ces méthodes indexent le texte par décalages d'octets et mesurent sa longueur en octets plutôt qu'en caractères. En pratique, compte tenu de la nature d'Unicode, l'indexation par caractère n'est pas aussi utile qu'il y paraît, et les décalages d'octets sont plus rapides et plus simples. Si vous essayez d'utiliser un décalage d'octet qui atterrit au milieu de l'encodage UTF-8 d'un caractère, la méthode panique, vous ne pouvez donc pas introduire un UTF-8 mal formé de cette façon.

A `String` est implémenté comme un wrapper autour de `a Vec<u8>` qui garantit que le contenu du vecteur est toujours bien formé en UTF-8. Rust ne changera jamais `String` pour utiliser une représentation plus compliquée, vous pouvez donc supposer que `String` partage `Vec` les caractéristiques de performance de .

Dans ces explications, les variables ont les types indiqués dans le [tableau 17-3](#).

Tableau 17-3. Types de variables utilisées dans les explications

Variable	Type présumé
<code>string</code>	<code>String</code>
<code>slice</code>	<code>&str</code> ou quelque chose qui déréférence à un, comme <code>String</code> ou <code>Rc<String></code>
<code>ch</code>	<code>char</code>
<code>n</code>	<code>usize</code> , une longueur
<code>i, j</code>	<code>usize</code> , un décalage d'octet
<code>range</code>	Une plage de <code>usize</code> décalages d'octets, entièrement délimitée comme <code>i..j</code> , ou partiellement délimitée comme <code>i..., ...j</code> ou <code>..</code>
<code>patter</code>	Tout type de motif: <code>char</code> , <code>String</code> , <code>&str</code> , <code>&[char]</code> , ou <code>FnMut(char) -> bool</code>

Nous décrivons les types de modèles dans [« Modèles de recherche de texte »](#).

Création de valeurs de chaîne

Il existe quelques façons courantes de créer des `String` valeurs:

`String::new()`

Retourne une nouvelle chaîne vide. Cela n'a pas de tampon alloué au tas, mais en allouera un selon les besoins.

`String::with_capacity(n)`

Retourne une nouvelle chaîne vide avec un tampon pré-alloué pour contenir au moins des `n` octets. Si vous connaissez à l'avance la longueur de la chaîne que vous construisez, ce constructeur vous permet de dimensionner correctement le tampon dès le début, au lieu de redimensionner le tampon au fur et à mesure que vous construisez la chaîne. La chaîne augmentera toujours son tampon selon les besoins si sa longueur dépasse les `n` octets. Comme les vecteurs, les chaînes ont des méthodes `capacity`, `reserve` et `shrink_to_fit`, mais généralement la logique d'allocation par défaut convient.

`str_slice.to_string()`

Attribue un fichier frais `String` dont le contenu est une copie de `str_slice`. Nous avons utilisé des expressions comme "literal text".`to_string()` tout au long du livre pour créer `String` des s à partir de littéraux de chaîne.

`iter.collect()`

Construit une chaîne en concaténant les éléments d'un itérateur, qui peuvent être `char`, `&str` ou `String` des valeurs. Par exemple, pour supprimer tous les espaces d'une chaîne, vous pouvez écrire :

```
let spacey = "man hat tan";
let spaceless:String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

L'utilisation `collect` de cette méthode tire parti de `String` l'implémentation du `std::iter::FromIterator` trait par.

`slice.to_owned()`

Retourne une copie de la trame en tant que fichier fraîchement alloué `String`. Le `str` type ne peut pas implémenter `Clone` : le trait nécessiterait `clone` sur un `&str` de retourner une `str` valeur, mais `str` n'est pas dimensionné. Cependant, `&str` implemente `ToOwned`, ce qui permet à l'implémenteur de spécifier son équivalent possédé.

Inspection simplifiée

Ces méthodes obtiennent des informations de base à partir de tranches de chaîne :

`slice.len()`

La durée de `slice`, en octets.

`slice.is_empty()`

Vrai si `slice.len() == 0`.

`slice[range]`

Retourne une trame empruntant la portion donnée de `slice`. Les plages partiellement délimitées et non délimitées sont acceptables ; par exemple :

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
```

```
assert_eq!(&full[2..4], "ok");
assert_eq!(full[...].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

Notez que vous ne pouvez pas indexer une tranche de chaîne avec une seule position, comme `slice[i]`. Récupérer un seul caractère à un décalage d'octet donné est un peu maladroit : vous devez produire un `chars` itérateur sur la tranche et lui demander d'analyser l'UTF-8 d'un caractère :

```
let parenthesized = "Rust (餓)";
assert_eq!(parenthesized[6..].chars().next(), Some('餓'));
```

Cependant, vous devriez rarement avoir besoin de le faire. Rust a des façons bien plus agréables d'itérer sur les tranches, que nous décrivons dans ["Itérer sur du texte"](#).

`slice.split_at(i)`

Retourne un tuple de deux tranches partagées empruntées à `slice` : la partie jusqu'à l'offset d'octet `i` et la partie qui suit. En d'autres termes, cela renvoie `(slice[..i], slice[i..])`.

`slice.is_char_boundary(i)`

Vraie si le décalage d'octet `i` tombe entre les limites de caractère et convient donc comme décalage dans `slice`.

Naturellement, les tranches peuvent être comparées pour l'égalité, ordonnées et hachées. La comparaison ordonnée traite simplement la chaîne comme une séquence de points de code Unicode et les compare dans l'ordre lexicographique.

Ajout et insertion de texte

Les méthodes suivantes ajoutent du texte à un `String`:

`string.push(ch)`

Ajoute le personnage `ch` jusqu'au bout `string`.

`string.push_str(slice)`

Ajoute le contenu complet de `slice`.

`string.extend(iter)`

Ajoute les éléments produits par l'itérateur `iter` à la chaîne. L'itérateur peut produire des valeurs `char`, `str` ou `String`. Voici

String les implémentations de std::iter::Extend :

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");

string.insert(i, ch)
```

Encarts le caractère unique `ch` au décalage d'octet `i` dans `string`. Cela implique de déplacer tous les caractères après `i` pour faire de la place pour `ch`, donc la construction d'une chaîne de cette manière peut nécessiter un temps quadratique dans la longueur de la chaîne.

```
string.insert_str(i, slice)
```

Cette fois de même pour `slice`, avec la même mise en garde en matière de performances.

String met en œuvre `std::fmt::Write`, c'est-à-dire que les macros `write!` et `writeln!` peut ajouter du texte formaté à `String`s :

```
use std:: fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n"
                  "His house is in the village though;\n");
```

Puisque `write!` et `writeln!` sont conçus pour écrire dans les flux de sortie, ils renvoient un `Result`, ce que Rust se plaint si vous l'ignorez. Ce code utilise l'`?` opérateur pour le gérer, mais écrire dans un `String` est en fait infaillible, donc dans ce cas, appeler `.unwrap()` serait également OK.

Depuis `String` implémente `Add<&str>` et `AddAssign<&str>`, vous pouvez écrire du code comme ceci :

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");

right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

Lorsqu'il est appliqué à des chaînes, l'`+` opérateur prend son opérande gauche par valeur, de sorte qu'il peut réellement le réutiliser `String` à la suite de l'addition. Par conséquent, si le tampon de l'opérande de gauche

est suffisamment grand pour contenir le résultat, aucune allocation n'est nécessaire.

Dans un malheureux manque de symétrie, l'opérande gauche de `+` ne peut pas être un `&str`, donc vous ne pouvez pas écrire :

```
let parenthetical = "(" + string + ")";
```

Vous devez plutôt écrire :

```
let parenthetical = ".to_string() + &string + ")";
```

Cependant, cette restriction décourage la création de chaînes à partir de la fin vers l'arrière. Cette approche fonctionne mal car le texte doit être déplacé à plusieurs reprises vers la fin du tampon.

Construire des chaînes du début à la fin en ajoutant de petits morceaux, cependant, est efficace. A `String` se comporte comme un vecteur, doublant toujours au moins la taille de son tampon lorsqu'il a besoin de plus de capacité. Cela maintient la surcharge de recopie proportionnelle à la taille finale. Même ainsi, utiliser `String::with_capacity` pour créer les chaînes avec la bonne taille de tampon pour commencer évitent tout redimensionnement et peuvent réduire le nombre d'appels à l'allocateur de tas.

Supprimer et remplacer du texte

`String` a quelques méthodes pour supprimer du texte (ceux-ci n'affectent pas la capacité de la chaîne ; utilisez -les `shrink_to_fit` si vous avez besoin de libérer de la mémoire) :

```
string.clear()
```

Réinitialise `string` à la chaîne vide.

```
string.truncate(n)
```

Rejetstous les caractères après le décalage d'octet `n`, laissant `string` une longueur d'au plus `n`. Si `string` est plus court que `n` octets, cela n'a aucun effet.

```
string.pop()
```

Supprimele dernier caractère de `string`, le cas échéant, et le renvoie sous la forme d'un `Option<char>`.

```
string.remove(i)
```

Supprime le caractère à l'octet de décalage `i` de `string` et le renvoie, décalant tous les caractères suivants vers l'avant. Cela prend un temps linéaire dans le nombre de caractères suivants.

```
string.drain(range)
```

Retourne un itérateur sur la plage donnée d'indices d'octets et supprime les caractères une fois l'itérateur supprimé. Les caractères après la plage sont décalés vers l'avant :

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

Si vous souhaitez simplement supprimer la plage, vous pouvez simplement supprimer l'itérateur immédiatement, sans en tirer aucun élément :

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

```
string.replace_range(range, replacement)
```

Remplace la plage donnée `string` avec la tranche de chaîne de remplacement donnée. La tranche n'a pas besoin d'être de la même longueur que la plage remplacée, mais à moins que la plage remplacée n'aille jusqu'à la fin de `string`, cela nécessitera de déplacer tous les octets après la fin de la plage :

```
let mut beverage = "a piña colada".to_string();
beverage.replace_range(2..7, "kahlua"); // 'ñ' is two bytes!
assert_eq!(beverage, "a kahlua colada");
```

Conventions de recherche et d'itération

Fonctions de la bibliothèque standard de Rust pour la recherche et itérations sur le texte, suivez certaines conventions de dénomination pour les rendre plus faciles à retenir :

`r`

La plupart des opérations traitent le texte du début à la fin, mais les opérations dont le nom commence par `r` fonctionnent de la fin au début. Par exemple, `rsplit` est la version de bout en bout de `split`. Dans certains cas, le changement de direction peut affecter

non seulement l'ordre dans lequel les valeurs sont produites, mais également les valeurs elles-mêmes. Voir le schéma de la [Figure 17-3](#) pour un exemple.

`n`

Les itérateurs dont le nom se termine par `n` se limitent à un nombre donné de correspondances.

`_indices`

Les itérateurs dont les noms se terminent par `_indices` produisent, avec leurs valeurs d'itération habituelles, les décalages d'octets dans la tranche à laquelle ils apparaissent.

La bibliothèque standard ne fournit pas toutes les combinaisons pour chaque opération. Par exemple, de nombreuses opérations n'ont pas besoin d'une `n` variante, car il est assez facile de simplement terminer l'itération plus tôt.

Modèles de recherche de texte

Lorsqu'une fonction de bibliothèque standard doit rechercher, faire correspondre, fractionner ou rogner du texte, il accepte plusieurs types différents pour représenter ce qu'il faut rechercher :

```
let haystack = "One fine day, in the middle of the night";  
  
assert_eq!(haystack.find(','), Some(12));  
assert_eq!(haystack.find("night"), Some(35));  
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

Ces types sont appelés *modèles* et la plupart des opérations les prennent en charge :

```
assert_eq!(## Elephants  
    .trim_start_matches(|ch:char| ch == '#' || ch.is_whitespace()),  
    "Elephants");
```

La bibliothèque standard prend en charge quatre principaux types de modèles :

- A `char` en tant que modèle correspond à ce caractère.
- Un `String` ou `&str` ou `&&str` en tant que modèle correspond à une sous-chaîne égale au modèle.

- Une `FnMut(char) -> bool` fermeture en tant que modèle correspond à un seul caractère pour lequel la fermeture renvoie vrai.
- A `&[char]` en tant que modèle (pas un `&str`, mais une tranche de `char` valeurs) correspond à n'importe quel caractère unique qui apparaît dans la liste. Notez que si vous écrivez la liste sous la forme d'un tableau littéral, vous devrez peut-être appeler `as_ref()` pour obtenir le bon type :

```
let code = "\t    function noodle() { ";
assert_eq!(code.trim_start_matches([' ', '\t']).as_ref(),
          "function noodle() { ");
// Shorter equivalent: &[' ', '\t'][..]
```

Sinon, Rust sera confus par le type de tableau de taille fixe `&[char; 2]`, qui n'est malheureusement pas un type de modèle.

Dans le propre code de la bibliothèque, un modèle est tout type qui implémente le `std::str::Pattern` trait. Les détails de `Pattern` ne sont pas encore stables, vous ne pouvez donc pas l'implémenter pour vos propres types dans Rust stable, mais la porte est ouverte pour autoriser les expressions régulières et d'autres modèles sophistiqués à l'avenir. Rust garantit que les types de modèles pris en charge actuellement continueront de fonctionner à l'avenir.

Recherche et remplacement

Rust a quelques méthodes pour rechercher des motifs dans les tranches et éventuellement en les remplaçant par un nouveau texte :

`slice.contains(pattern)`

Retourne true si `slice` contient une correspondance pour `pattern`.

`slice.starts_with(pattern), slice.ends_with(pattern)`

Retournent true si `slice` le texte initial ou final correspond à `pattern`:

```
assert!("2017".starts_with(char::is_numeric));
```

`slice.find(pattern), slice.rfind(pattern)`

Retournent `Some(i)` si `slice` contient une correspondance pour `pattern`, où `i` est le décalage d'octet auquel le motif apparaît. La `find` méthode renvoie la première correspondance, `rfind` la dernière :

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

slice.replace(pattern, replacement)

Retourne un nouveau `String` formé en remplaçant avec empressement tous les matches pour `pattern` par `replacement`:

```
assert_eq!("The only thing we have to fear is fear itself"
    .replace("fear", "spin"),
    "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
    .replace(|ch:char| !ch.is_alphanumeric(), ""),
    "BorrowandBorrowMut");
```

Étant donné que le remplacement est effectué avec empressement, `.replace()` le comportement de lors des correspondances qui se chevauchent peut être surprenant. Ici, il y a quatre instances du modèle, "aba", mais la deuxième et la quatrième ne correspondent plus après le remplacement de la première et de la troisième :

```
assert_eq!("cabababababbage"
    .replace("aba", "***"),
    "c***b***babbage")
```

slice.replace_n(pattern, replacement, n)

Cette fois fait la même chose, mais remplace au maximum les premiers `n` matchs.

Itérer sur du texte

La bibliothèque standard fournit plusieurs façons d'itérer sur le texte d'une tranche. [La figure 17-3](#) en montre des exemples.

Vous pouvez considérer les familles `split` et `match` comme étant complémentaires : les fractionnements sont les plages entre les correspondances.

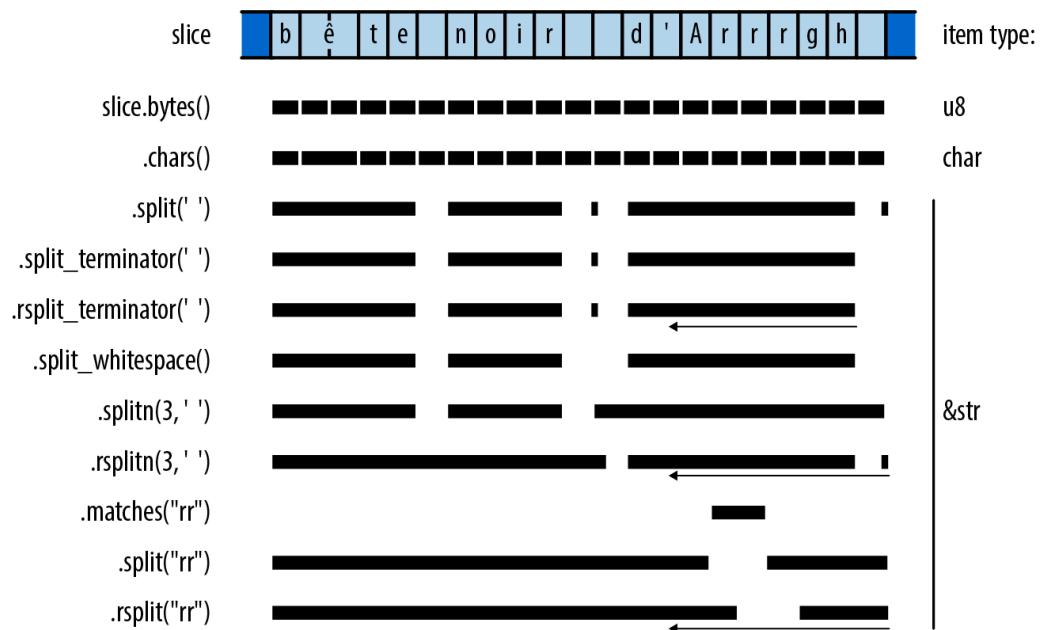


Illustration 17-3. Quelques façons d'itérer sur une tranche

La plupart de ces méthodes renvoient des itérateurs réversibles (c'est-à-dire qu'elles implémentent `DoubleEndedIterator`) : l'appel de leur `.rev()` méthode d'adaptation vous donne un itérateur qui produit les mêmes éléments, mais dans l'ordre inverse.

`slice.chars()`

Retourne un itérateur sur `slice` les caractères de .

`slice.char_indices()`

Retourne un itérateur sur `slice` les caractères de et leurs décalages d'octet :

```
assert_eq!( "élan" .char_indices() .collect::<Vec<_>>(),
            vec![ (0, 'é'), // has a two-byte UTF-8 encoding
                  (2, 'l'),
                  (3, 'a'),
                  (4, 'n') ]);
```

Notez que ce n'est pas équivalent à `.chars().enumerate()`, car il fournit le décalage d'octet de chaque caractère dans la tranche, au lieu de simplement numérotter les caractères.

`slice.bytes()`

Retourne un itérateur sur les octets individuels de `slice`, exposant l'encodage UTF-8 :

```
assert_eq!( "élan" .bytes() .collect::<Vec<_>>(),
            vec![ 195, 169, b'l', b'a', b'n']);
```

```
slice.lines()
```

Retourne un itérateur sur les lignes de `slice`. Les lignes se terminent par
"\n" ou "\r\n". Chaque pièce produite est un `&str` emprunt à `slice`.
Les éléments n'incluent pas les caractères de fin de ligne.

```
slice.split(pattern)
```

Retourne un itérateur sur les portions de `slice` séparées par des correspondances de `pattern`. Cela produit des chaînes vides entre les correspondances immédiatement adjacentes, ainsi que pour les correspondances au début et à la fin de `slice`.

L'itérateur retourné n'est pas réversible si `pattern` est un `&str`. De tels modèles peuvent produire différentes séquences de correspondances en fonction de la direction à partir de laquelle vous numérissez, ce que les itérateurs réversibles sont interdits de faire. Au lieu de cela, vous pourrez peut-être utiliser la `rsplit` méthode décrite ci-après.

```
slice.rsplit(pattern)
```

Cette est la même, mais analyse `slice` de bout en bout, produisant des correspondances dans cet ordre.

```
slice.split_terminator(pattern), slice.rsplit_terminator(pattern)
```

Ce sont similaires, sauf que le motif est traité comme un terminateur, pas comme un séparateur : si `pattern` les correspondances sont à la toute fin de `slice`, les itérateurs ne produisent pas de tranche vide représentant la chaîne vide entre cette correspondance et la fin de la tranche, comme `split` et `rsplit` faire. Par exemple :

```
// The ':' characters are separators here. Note the final "".
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
           vec!["jimb", "1000", "Jim Blandy", ""]);

// The '\n' characters are terminators here.
assert_eq!("127.0.0.1 localhost\n"
           "127.0.0.1 www.reddit.com\n"
           .split_terminator('\n').collect::<Vec<_>>(),
           vec!["127.0.0.1 localhost",
                 "127.0.0.1 www.reddit.com"]);
           // Note, no final ""!
```

```
slice.splitn(n, pattern), slice.rsplitn(n, pattern)
```

Cessent comme `split` et `rsplit`, sauf qu'ils divisent la chaîne en au plus `n` tranches, à la première ou à la dernière `n-1` correspondance pour `pattern`.

```
slice.split_whitespace(), slice.split_ascii_whitespace()
```

Revenir un itérateur sur les parties séparées par des espaces de `slice`. Une série de plusieurs caractères d'espacement est considérée comme un seul séparateur. Les espaces de fin sont ignorés.

La `split_whitespace` méthode utilise la définition Unicode des espaces blancs, telle qu'implémentée par la `is whitespace` méthode sur `char`. La `split_ascii_whitespace` méthode utilise à la `char::is_ascii whitespace` place, qui ne reconnaît que les caractères d'espacement ASCII.

```
let poem = "This is just to say\n\
           I have eaten\n\
           the plums\n\
           again\n";

assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
           vec![ "This", "is", "just", "to", "say",
                  "I", "have", "eaten", "the", "plums",
                  "again"]);
```

```
slice.matches(pattern)
```

Retourner un itérateur sur les correspondances pour `pattern` in `slice`.

`slice.rmatches(pattern)` est le même, mais itère de la fin au début.

```
slice.match_indices(pattern), slice.rmatch_indices(pattern)
```

Cessent similaires, sauf que les éléments produits sont des `(offset, match)` paires, où `offset` est le décalage d'octet auquel la correspondance commence, et `match` est la tranche correspondante.

Garniture

Pour *tailler* une chaîne consiste à supprimer du texte, généralement des espaces, au début ou à la fin de la chaîne. C'est souvent utile pour nettoyer l'entrée lue à partir d'un fichier où l'utilisateur peut avoir mis du texte en retrait pour la lisibilité ou laissé accidentellement un espace blanc à la fin d'une ligne.

```
slice.trim()
```

Retourne une sous-tranche de `slice` qui omet tout espace blanc de début et de fin. `slice.trim_start()` omet uniquement les espaces blancs de début, `slice.trim_end()` uniquement les espaces blancs de fin :

```
assert_eq!("t*.rs  ".trim(), "*.rs");
assert_eq!("t*.rs  ".trim_start(), "*.rs  ");
assert_eq!("t*.rs  ".trim_end(), "\t*.rs");
```

`slice.trim_matches(pattern)`

Retourne une sous-tranche de `slice` qui omet toutes les correspondances de `pattern` du début à la fin. Les méthodes `trim_start_matches` et `trim_end_matches` font de même pour les correspondances de début ou de fin uniquement :

```
assert_eq!("001990".trim_start_matches('0'), "1990");
```

`slice.strip_prefix(pattern), slice.strip_suffix(pattern)`

Si `slice` commence par `pattern`, `strip_prefix` retourne `Some` en tenant la tranche avec le texte correspondant supprimé. Sinon, ça revient `None`. La `strip_suffix` méthode est similaire, mais recherche une correspondance à la fin de la chaîne.

Ce sont comme `trim_start_matches` et `trim_end_matches`, sauf qu'ils renvoient un `Option`, et une seule copie de `pattern` est supprimée :

```
let slice = "banana";
assert_eq!(slice.strip_suffix("na"),
           Some("bana"))
```

Conversion de casse pour les chaînes

Les méthodes `slice.to_uppercase()` et le `slice.to_lowercase()` retournent une chaîne fraîchement allouée contenant le texte `slice` converti en majuscule ou en minuscule. Le résultat peut ne pas être de la même longueur que `slice`; voir "[Conversion de la casse pour les caractères](#)" pour plus de détails.

Analyse d'autres types à partir de chaînes

Rust fournit des traits standard pour l'analyse de valeurs à partir de chaînes et produire des représentations textuelles des valeurs.

Si un type implémente le `std::str::FromStr` trait, il fournit alors un moyen standard d'analyser une valeur à partir d'une tranche de chaîne :

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

Tous les types de machines usuelles mettent en œuvre `FromStr` :

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

Le `char` type implémente également `FromStr`, pour les chaînes avec un seul caractère :

```
assert_eq!(char::from_str("é"), Ok('é'));
assert!(char::from_str("abcdefg").is_err());
```

Le `std::net::IpAddr` type, `enum` qui détient une adresse Internet IPv4 ou IPv6, implémente `FromStr` également :

```
use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
           IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a50]))
```

Les tranches de chaîne ont une `parse` méthode qui analyse la tranche dans le type de votre choix, en supposant qu'elle implémente `FromStr`. Comme pour `Iterator::collect`, vous aurez parfois besoin d'épeler le type que vous voulez, ce `parse` n'est donc pas toujours beaucoup plus lisible que d'appeler `from_str` directement :

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

Conversion d'autres types en chaînes

Il existe trois façons principales de convertir des valeurs non textuelles en chaînes :

- Les types qui ont une forme imprimée naturelle lisible par l'homme peuvent implémenter le `std::fmt::Display` trait, qui vous permet d'utiliser le `{}` spécificateur de format dans la `format!` macro :

```
assert_eq!(format!("{}"), "wow", "doge", "doge, wow");
assert_eq!(format!("{}"), true, "true");
assert_eq!(format!("({:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
           "(0.500, 0.866)");

// Using `address` from above.
let formatted_addr:String = format!("{}", address);
assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
```

Tous les types numériques de machine de Rust implémentent `Display`, tout comme les caractères, les chaînes et les tranches. Le pointeur intelligent type `Box<T>`, `Rc<T>`, et `Arc<T>` implement `Display` si `T` lui-même le fait : leur forme affichée est simplement celle de leur référent. Les conteneurs aiment `Vec` et `HashMap` n'implémentent pas `Display`, car il n'y a pas de forme naturelle lisible par l'homme pour ces types.

- Si un type implémente `Display`, la bibliothèque standard implémente automatiquement le `std::str::ToString` trait correspondant, dont la seule méthode `to_string` peut être plus pratique lorsque vous n'avez pas besoin de la flexibilité de `format!` :

```
// Continued from above.
assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
```

Le `ToString` trait est antérieur à l'introduction de `Display` et est moins flexible. Pour vos propres types, vous devez généralement implémenter à la `Display` place de `ToString`.

- Toujours public dans la bibliothèque standard implements `std::fmt::Debug`, qui prend une valeur et la formate en tant que chaîne d'une manière utile aux programmeurs. Le moyen le plus simple `Debug` de produire une chaîne consiste à utiliser le spécificateur de format de la `format!` macro : `{:?}`

```
// Continued from above.

let addresses = vec![address,
                     IpAddr::from_str("192.168.0.1")?];
assert_eq!(format!("{}:{}:{:?}", addresses),
           "[fe80::3ea9:f4ff:fe34:7a50, 192.168.0.1]");
```

Cela tire parti d'une implémentation globale de `Debug` pour `Vec<T>`, pour tout `T` ce qui implémente lui-même `Debug`. Tous les types de collection de Rust ont de telles implémentations.

Vous devez également implémenter `Debug` pour vos propres types. Habituellement, il est préférable de laisser Rust dériver une implémentation, comme nous l'avons fait pour le `Complex` type au [chapitre 12](#) :

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im:f64 }
```

Les traits `Display` et de `Debug` formatage ne sont que deux parmi plusieurs que la `format!` macro et ses parents utilisent pour formater les valeurs sous forme de texte. Nous couvrirons les autres, et expliquerons comment les implémenter tous, dans "[Formatage des valeurs](#)".

Emprunter sous d'autres types de texte

Tu peux emprunter le contenu d'une trame de plusieurs manières :

- Tranches et `String`s implémentent `AsRef<str>`, `AsRef<[u8]>`, `AsRef<Path>` et `AsRef<OsStr>`. De nombreuses fonctions de bibliothèque standard utilisent ces traits comme limites sur leurs types de paramètres, vous pouvez donc leur transmettre directement des tranches et des chaînes, même lorsqu'elles veulent vraiment un autre type. Voir « [AsRef et AsMut](#) » pour une explication plus détaillée.
- Les tranches et les chaînes implémentent également le `std::borrow::Borrow<str>` trait. `HashMap` et `BTreeMap` utiliser `Borrow` pour que `String`s fonctionne bien comme clés dans une table. Voir "[Emprunter et EmprunterMut](#)" pour plus de détails.

Accéder au texte en UTF-8

Il y a deux principales façons d'accéder aux octets représentant le texte, selon que vous souhaitez vous appropier les octets ou simplement les emprunter :

```
slice.as_bytes()
```

Emprunte `slice` les octets de `&[u8]`. Comme il ne s'agit pas d'une référence mutable, `slice` on peut supposer que ses octets resteront bien formés en UTF-8.

```
string.into_bytes()
```

S'approprie `string` et renvoie un `Vec<u8>` des octets de la chaîne par valeur. Il s'agit d'une conversion bon marché, car elle transmet simplement le `Vec<u8>` que la chaîne utilisait comme tampon. Comme `string` il n'existe plus, il n'est pas nécessaire que les octets continuent d'être bien formés en UTF-8, et l'appelant est libre de modifier le `Vec<u8>` à sa guise.

Production de texte à partir de données UTF-8

Si vous avez un bloc d'octetsque vous pensez contenir des données UTF-8, vous avez quelques options pour les convertir en `String`s ou tranches, selon la façon dont vous souhaitez gérer les erreurs :

```
str::from_utf8(byte_slice)
```

Prend une `&[u8]` tranche d'octets et renvoie un `Result` : soit `Ok(&str)` s'il `byte_slice` contient de l'UTF-8 bien formé, soit une erreur dans le cas contraire.

```
String::from_utf8(vec)
```

Essaiepour construire une chaîne à partir d'une `Vec<u8>` valeur passée par. Si `vec` contient UTF-8 bien formé, `from_utf8` renvoie `Ok(string)`, où `string` a pris possession de `vec` pour l'utiliser comme tampon. Aucune allocation de tas ou copie du texte n'a lieu.

Si les octets ne sont pas UTF-8 valides, cela renvoie `Err(e)`, où `e` est une `FromUtf8Error` valeur d'erreur. L'appel `e.into_bytes()` vous renvoie le vector d'origine `vec`, de sorte qu'il n'est pas perdu lorsque la conversion échoue :

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(), Some("鍇".to_string()));

let bad_utf8: Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// Since String::from_utf8 failed, it didn't consume the original
// vector, and the error value hands it back to us unharmed.
assert_eq!(result.unwrap_err().into_bytes(),
          vec![0x9f, 0xf0, 0xa6, 0x80]);
```

```
String::from_utf8_lossy(byte_slice)
```

Essaie pour construire un `String` ou `&str` à partir d'une `&[u8]` tranche partagée d'octets. Cette conversion réussit toujours, remplaçant tout UTF-8 mal formé par des caractères de remplacement Unicode. La valeur de retour est un `Cow<str>` qui emprunte `&str` directement à `byte_slice` s'il contient de l'UTF-8 bien formé ou possède un nouvellement alloué `String` avec des caractères de remplacement substitués aux octets mal formés. Par conséquent, lorsque `byte_slice` est bien formé, aucune allocation de tas ou copie n'a lieu. Nous en discuterons `Cow<str>` plus en détail dans ["Reporter l'allocation"](#).

```
String::from_utf8_unchecked(vec)
```

Si vous savez pertinemment que votre `Vec<u8>` contient UTF-8 bien formé, vous pouvez appeler cette fonction non sécurisée. Cela se termine simplement `vec` par un `String` et le renvoie, sans examiner du tout les octets. Vous êtes responsable de vous assurer que vous n'avez pas introduit d'UTF-8 mal formé dans le système, c'est pourquoi cette fonction est marquée `unsafe`.

```
str::from_utf8_unchecked(byte_slice)
```

De la même manière, cela prend un `&[u8]` et le renvoie comme un `&str`, sans vérifier s'il contient de l'UTF-8 bien formé. Comme pour `String::from_utf8_unchecked`, vous êtes responsable de vous assurer que cela est sûr.

Différer l'attribution

Supposons que vous voulez que votre programme accueille l'utilisateur. Sous Unix, vous pourriez écrire :

```
fn get_name() -> String {
    std::env::var("USER") // Windows uses "USERNAME"
        .unwrap_or("whoever you are".to_string())
}

println!("Greetings, {}!", get_name());
```

Pour les utilisateurs Unix, cela les accueille par nom d'utilisateur. Pour les utilisateurs de Windows et les tragiquement anonymes, il fournit un texte de stock alternatif.

La `std::env::var` fonction renvoie un `String` —et a de bonnes raisons de le faire que nous n'aborderons pas ici. Mais cela signifie que le texte de stock alternatif doit également être renvoyé en tant que fichier `String`. C'est décevant : lors de la retour d'une chaîne statique, aucune allocation ne devrait être nécessaire du tout.

Le nœud du problème est que parfois la valeur de retour de `get_name` devrait être un `Owned String`, parfois ce devrait être un `&static str`, et nous ne pouvons pas savoir lequel ce sera tant que nous n'aurons pas exécuté le programme. Ce caractère dynamique est l'indice à envisager d'utiliser `std::borrow::Cow`, le clone-type en écriture pouvant contenir des données détenues ou empruntées.

Comme expliqué dans "[Borrow and ToOwned at Work: The Humble Cow](#)" , `Cow<'a, T>` est une énumération avec deux variantes : `Owned` et `Borrowed`. `Borrowed` contient une référence `&'a T` et `Owned` contient la version propriétaire de `&T: String` for `&str`, `Vec<i32>` for `&[i32]`, etc. Que ce soit `Owned` ou `Borrowed`, `a Cow<'a, T>` peut toujours produire un `&T` pour que vous puissiez l'utiliser. En fait, `Cow<'a, T>` les déréférences à `&T`, se comportent comme une sorte de pointeur intelligent.

Changer `get_name` pour renvoyer un `Cow` résultat dans ce qui suit :

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| Cow::Owned(v))
        .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

Si cela réussit à lire la "USER" variable d'environnement, le `map` renvoie le résultat `String` sous la forme d'un fichier `Cow::Owned`. En cas d'échec, le `unwrap_or` renvoie son statique `&str` sous la forme d'un fichier `Cow::Borrowed`. L'appelant peut rester inchangé :

```
println!("Greetings, {}!", get_name());
```

Tant que `T` le `std::fmt::Display` trait est implémenté, l'affichage d'un `Cow<'a, T>` produit les mêmes résultats que l'affichage d'un `T`.

`Cow` est également utile lorsque vous pouvez ou non avoir besoin de modifier un texte que vous avez emprunté. Lorsqu'aucune modification n'est nécessaire, vous pouvez continuer à l'emprunter. Mais le comportement de clonage sur écriture homonyme de `Cow` peut vous donner une copie propriétaire et modifiable de la valeur à la demande. La `to_mut` méthode de s'assure que le `Cow` est `Cow::Owned`, en appliquant l'`ToOwned` implémentation de la valeur si nécessaire, puis renvoie une référence mutable à la valeur.

Ainsi, si vous constatez que certains de vos utilisateurs, mais pas tous, ont des titres par lesquels ils préféreraient être adressés, vous pouvez dire :

```
fn get_title() ->Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}

println!("Greetings, {}!", name);
```

Cela peut produire une sortie comme celle-ci :

```
$course de fret
Greetings, jimb, Esq.!
$
```

Ce qui est bien ici, c'est que si `get_name()` renvoie une chaîne statique et `get_title` renvoie `None`, le `Cow` transporte simplement la chaîne statique jusqu'au `println!`. Vous avez réussi à reporter l'allocation à moins que ce ne soit vraiment nécessaire, tout en écrivant du code simple.

Étant donné `Cow` qu'il est fréquemment utilisé pour les chaînes, la bibliothèque standard a un support spécial pour `Cow<'a, str>`. Il fournit `From` et `Into` convertit à la fois `String` et `&str`, vous pouvez donc écrire `get_name` plus brièvement :

```
fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| v.into())
        .unwrap_or("whoever you are".into())
}
```

`Cow<'a, str>` implémente également `std::ops::Add` et `std::ops::AddAssign`, donc pour ajouter le titre au nom, vous pouvez écrire :

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

Ou, puisque `a String` peut être `write!` la destination d'une macro :

```
use std::fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}, title).unwrap();
}
```

Comme précédemment, aucune allocation ne se produit tant que vous n'essayez pas de modifier le fichier `Cow`.

Gardez à l'esprit que tous les `Cow<..., str>` doivent pas l'être. `'static` : vous pouvez utiliser `Cow` pour emprunter du texte précédemment calculé jusqu'au moment où une copie devient nécessaire.

Chaînes en tant que collections génériques

`String` met en œuvre les deux `std::default::Default` et `std::iter::Extend`. `default` renvoie une chaîne vide et `extend` peut ajouter des caractères, des tranches de chaîne, des `Cow<..., str>`s ou des chaînes à la fin d'une chaîne. Il s'agit de la même combinaison de traits implémentés par les autres types de collection de Rust comme `Vec` et `HashMap` pour les modèles de construction génériques tels que `collect` et `partition`.

Le `&str` type implémente également `Default`, renvoyant une tranche vide. C'est pratique dans certains cas d'angle ; par exemple, il vous permet de dériver `Default` des structures contenant des tranches de chaîne.

Valeurs de formatage

Tout au long du livre, nous avons utilisé la mise en forme du texte des macros comme `println!` :

```
println!("{:3}μs: relocated {} at {:#x} to {:#x}, {} bytes",
        0.84391, "object",
        140737488346304_usize, 6299664_usize, 64);
```

Cet appel produit la sortie suivante :

```
0.844μs: relocated object at 0x7fffffffcc0 to 0x602010, 64 bytes
```

Le littéral de chaîne sert de modèle pour la sortie : chacun `{...}` dans le modèle est remplacé par la forme formatée de l'un des arguments suivants. La chaîne de modèle doit être une constante afin que Rust puisse la comparer aux types des arguments au moment de la compilation. Chaque argument doit être utilisé ; Sinon, Rust signale une erreur de compilation.

Plusieurs fonctionnalités de la bibliothèque standard partagent ce petit langage de formatage des chaînes :

- La `format!` macrol'utilise pour construire `String`s.
- Les macros `println!` et `print!` écrire du texte formaté dans le flux de sortie standard.
- Les macros `writeln!` et `write!` l'écrire dans un flux de sortie désigné.
- La `panic!` macrol'utilise pour construire une expression (idéalement informative) de consternation terminale.

Les fonctions de formatage de Rust sont conçues pour être ouvertes. Vous pouvez étendre ces macros pour prendre en charge vos propres types en implémentant les `std::fmt` traits de formatage du module. Et vous pouvez utiliser la `format_args!` macro et le `std::fmt::Arguments` type pour que vos propres fonctions et macros prennent en charge le langage de formatage.

Les macros de formatage empruntent toujours des références partagées à leurs arguments ; ils ne s'en approprient jamais ni ne les transforment.

Les formulaires du modèle `{...}` sont appelés *paramètres de formatet* avoir le formulaire . Les deux parties sont facultatives ; est fréquemment utilisé. `{which:how} { }`

La `which` valeur sélectionne l'argument suivant le modèle qui doit prendre la place du paramètre. Vous pouvez sélectionner des arguments par index ou par nom. Les paramètres sans `which` valeur sont simplement associés à des arguments de gauche à droite.

La `how` valeur indique comment l'argument doit être formaté : combien de rembourrage, à quelle précision, dans quelle base numérique, etc. Si `how` est présent, le colon avant est requis. [Le tableau 17-4](#) présente quelques exemples.

Tableau 17-4. Exemples de chaînes formatées

Chaîne de modèle	Liste des arguments	Résultat
"number of {}: {}"	"elephants", 19	"number of elephants: 19"
"from {1} to {0}"	"the grave", "the cradle"	"from the cradle to the grave"
"v = {::?}"	vec![0,1,2,5,12, 29]	"v = [0, 1, 2, 5, 12, 29]"
"name = {::?}"	"Nemo"	"name = \"Nemo \""
"{:8.2} km/s"	11.186	" 11.19 km/s"
"{:20} {:02x} {:02x}"	"adc #42", 105, 42	"adc #42 69 2a"
"{:02x} {:02 x} {0}"	"adc #42", 105, 42	"69 2a adc #4 2"
"{lsb:02x} {ms b:02x} {insn}"	insn="adc #42", lsb=105, msb=42	"69 2a adc #4 2"
"{:02?}"	[110, 11, 9]	"[110, 11, 09]"
"{:02x?}"	[110, 11, 9]	"[6e, 0b, 09]"

Si vous souhaitez inclure des caractères { ou } dans votre sortie, doublez les caractères dans le modèle :

```
assert_eq!(format!("{{a, c}} < {{a, b, c}}"),
           "{a, c} < {a, b, c}");
```

Formatage des valeurs de texte

Lors de la mise en forme d'un texte de type comme `&str` ou `String` (`char` est traité comme une chaîne à un seul caractère), la `how` valeur d'un paramètre comporte plusieurs parties, toutes facultatives :

- Une *limite de longueur de texte*. Rust tronque votre argument s'il est plus long que cela. Si vous ne spécifiez aucune limite, Rust utilise le texte intégral.
- Une *largeur de champ minimale*. Après toute troncature, si votre argument est plus court que cela, Rust le remplit à droite (par défaut) avec des espaces (par défaut) pour créer un champ de cette largeur. S'il est omis, Rust ne complète pas votre argument.
- Un *alignement*. Si votre argument doit être rempli pour respecter la largeur de champ minimale, cela indique où votre texte doit être placé dans le champ. <, ^ et > placez votre texte au début, au milieu et à la fin, respectivement.
- Un caractère de *remplissage* à utiliser dans ce processus de remplissage. S'il est omis, Rust utilise des espaces. Si vous spécifiez le caractère de remplissage, vous devez également spécifier l'alignement.

[Le tableau 17-5](#) illustre quelques exemples montrant comment écrire les choses et leurs effets. Tous utilisent le même argument à huit caractères, "bookends".

Tableau 17-5. Formater les directives de chaîne pour le texte

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	"{}"	"bookends"
Largeur de champ minimale	"{:4}"	"bookends"
	"{:12}"	"bookends" "
Limite de longueur de texte	"{:.4}"	"book"
	"{:.12}"	"bookends"
Largeur de champ, limite de longueur	"{:12.20}"	"bookends" "
	"{:4.20}"	"bookends"
	"{:4.6}"	"booken"
	"{:6.4}"	"book "
Aligné à gauche, largeur	"{:<12}"	"bookends" "
Centré, largeur	"{:^12}"	" bookends "
Aligné à droite, largeur	"{:>12}"	" booken ds"
Pad avec '=' , centré, largeur	"{:=^12}"	"==bookends =="
Pad '*' , aligné à droite, largeur, limite	"{:*>12.4}"	"*****bo ok"

Le formateur de Rust a une compréhension naïve de la largeur : il suppose que chaque caractère occupe une colonne, sans tenir compte des combinaisons de caractères, des katakana demi-largeur, des espaces de

largeur nulle ou des autres réalités désordonnées d'Unicode. Par exemple:

```
assert_eq!(format!("{:4}", "th\u{e9}"),    "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

Bien qu'Unicode indique que ces chaînes sont toutes deux équivalentes à "thé" , le formateur de Rust ne sait pas que des caractères tels que '\u{301}' , COMBINING ACUTE ACCENT, nécessitent un traitement spécial. Il remplit correctement la première chaîne, mais suppose que la seconde a une largeur de quatre colonnes et n'ajoute aucun remplissage. Bien qu'il soit facile de voir comment Rust pourrait s'améliorer dans ce cas spécifique, le véritable formatage de texte multilingue pour tous les scripts Unicode est une tâche monumentale, mieux gérée en s'appuyant sur les boîtes à outils de l'interface utilisateur de votre plate-forme, ou peut-être en générant du HTML et du CSS et en créant un site Web. navigateur trier tout cela. Il existe une caisse populaire, `unicode-width` , qui gère certains aspects de cela.

Avec `&str` et `String` , vous pouvez également passer des types de pointeurs intelligents de macros de mise en forme avec des référents textuels, comme `Rc<String>` ou `Cow<'a, str>` , sans cérémonie.

Étant donné que les chemins de nom de fichier ne sont pas nécessairement UTF bien formés-8, `std::path::Path` n'est pas tout à fait un type textuel ; vous ne pouvez pas passer a `std::path::Path` directement à une macro de formatage. Cependant, une `Path` méthode `display` renvoie une valeur que vous pouvez formater et qui trie les choses d'une manière adaptée à la plate-forme:

```
println!("processing file: {}", path.display());
```

Formatage des nombres

Lorsque l'argument de formatage a un nombre tapez comme `usize` ou `f64` , la `how` valeur du paramètre comporte les parties suivantes, toutes facultatives :

- Un *rembourrage* et un *alignement* , qui fonctionnent comme ils le font avec les types textuels.
- Un `+` caractère, demandant que le signe du nombre soit toujours affiché, même lorsque l'argument est positif.

- Un `#` caractère, demandant un préfixe de base explicite comme `0x` ou `0b`. Voir la puce "notation" qui conclut cette liste.
- Un `0` caractère, demandant que la largeur de champ minimale soit satisfaite en incluant des zéros non significatifs dans le nombre, au lieu de l'approche de remplissage habituelle.
- Une *largeur de champ minimale*. Si le nombre formaté n'est pas au moins aussi large, Rust le remplit à gauche (par défaut) avec des espaces (par défaut) pour créer un champ de la largeur donnée.
- Une *précision* pour les arguments à virgule flottante, indiquant le nombre de chiffres que Rust doit inclure après la virgule décimale. Rust arrondit ou étend zéro si nécessaire pour produire exactement ce nombre de chiffres fractionnaires. Si la précision est omise, Rust essaie de représenter avec précision la valeur en utilisant le moins de chiffres possible. Pour les arguments de type entier, la précision est ignorée.
- Une *notation*. Pour entiertypes, cela peut être `b` pour les binaires, `o` pour les octaux `x` ou `X` pour les hexadécimaux avec des lettres minuscules ou majuscules. Si vous avez inclus le `#` caractère, ceux-ci incluent un préfixe de base explicite de style Rust `0b`, `0o`, `0x`, ou `0X`. Pour les types à virgule flottante, une base de `e` ou `E` demande une notation scientifique, avec un coefficient normalisé, en utilisant `e` ou `E` pour l'exposant. Si vous ne spécifiez aucune notation, Rust formate les nombres en décimal.

[Le tableau 17-6](#) montre quelques exemples de formatage de la `i32` valeur `1234`.

Tableau 17-6. Formater les directives de chaîne pour les entiers

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	"{ }"	"1234"
Signe forcé	"{:+}"	"+1234"
Largeur de champ minimale	"{:12}"	"1234"
	"{:2}"	"1234"
Signe, largeur	"{:+12}"	"+1234"
Zéros non significatifs, largeur	"{:012}"	"000000001234"
Signe, zéros, largeur	"{:+012}"	"+000000001234"
Aligné à gauche, largeur	"{:<12}"	"1234"
Centré, largeur	"{:^12}"	"1234"
Aligné à droite, largeur	"{:>12}"	"1234"
Aligné à gauche, signe, largeur	"{:<+12}"	+1234
Centré, signe, largeur	"{:^+12}"	+1234
Aligné à droite, signe, largeur	"{:>+12}"	+1234"
Rembourré avec '=' , centré, largeur	"{:=^12}"	====1234===="

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Notation binaire	"{ :b}"	"10011010010"
Largeur, notation octale	"{ :12o}"	"2322"
Signe, largeur, notation hexadécimale	"{ :+12x}"	"+4d2"
Signe, largeur, hexadécimal avec chiffres majuscules	"{ :+12X}"	"+4D2"
Signe, préfixe de base explicite, largeur, hexadécimal	"{ :+#12x}"	"+0x4d2"
Signe, base, zéros, largeur, hexadécimal	"{ :+#012x}"	"+0x000004d2"
	"{ :+#06x}"	"+0x4d2"

Comme le montrent les deux derniers exemples, la largeur de champ minimale s'applique au nombre entier, au signe, au préfixe de base et à tous.

Les nombres négatifs incluent toujours leur signe. Les résultats sont similaires à ceux présentés dans les exemples de « signe forcé ».

Lorsque vous demandez des zéros non significatifs, les caractères d'alignement et de remplissage sont simplement ignorés, car les zéros étendent le nombre pour remplir tout le champ.

En utilisant l'argument `1234.5678`, nous pouvons montrer des effets spécifiques à la virgule flottante les types([Tableau 17-7](#)).

Tableau 17-7. Formater les directives de chaîne pour les nombres à virgule flottante

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	"{ }"	"1234.567 8"
Précision	"{:.2}"	"1234.57"
	"{:.6}"	"1234.5678 00"
Largeur de champ minimale	"{:12}"	" 1234.5 678"
Minimum, précision	"{:12.2}"	" 123 4.57"
	"{:12.6}"	" 1234.567 800"
Zéros non significatifs, minimum, précision	"{:012.6}"	"01234.567 800"
Scientifique	"{:e}"	"1.2345678 e3"
Scientifique, précision	"{:.3e}"	"1.235e3"
Scientifique, minimum, précision	"{:12.3e}"	" 1.23 5e3"
	"{:12.3E}"	" 1.23 5E3"

Formatage d'autres types

Au-delà des chaînes et des nombres, vous pouvez formater plusieurs autres types de bibliothèques standard:

- Erreurs types peuvent tous être formatés directement, ce qui facilite leur inclusion dans les messages d'erreur. Chaque type d'erreur doit

implémenter le `std::error::Error` trait, qui étend le trait de formatage par défaut `std::fmt::Display`. Par conséquent, tout type qui implémente `Error` est prêt à être formaté.

- Vous pouvez formater le protocole Internet les types d'adresse comme `std::net::IpAddr` et `std::net::SocketAddr`.
- Les booléens `true` et les `false` valeurs peuvent être formatés, bien que ce ne soient généralement pas les meilleures chaînes à présenter directement aux utilisateurs finaux.

Vous devez utiliser les mêmes sortes de paramètres de format que vous utiliseriez pour les chaînes. Les contrôles de limite de longueur, de largeur de champ et d'alignement fonctionnent comme prévu.

Formatage des valeurs pour le débogage

Pour aider au débogage et journalisation, le `{ :? }` paramètre reformate tout type public dans la bibliothèque standard Rust d'une manière destinée à être utile aux programmeurs. Vous pouvez l'utiliser pour inspecter des vecteurs, des tranches, des tuples, des tables de hachage, des threads et des centaines d'autres types.

Par exemple, vous pouvez écrire ce qui suit :

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606, -122.6819273));
map.insert("Taipei", (25.0375167, 121.5637));
println!("{:?}", map);
```

Cela imprime :

```
{"Taipei": (25.0375167, 121.5637), "Portland": (45.5237606, -122.6819273)}
```

Les types `HashMap` et `(f64, f64)` savent déjà comment se formater, sans aucun effort de votre part.

Si vous incluez le `#` caractère dans le paramètre de format, Rust imprimeira joliment la valeur. Changer ce code pour dire `println!("{:#?}", map)` conduit à cette sortie :

```
{
    "Taipei": (
        25.0375167,
        121.5637
```

```
),
"Portland": (
    45.5237606,
    -122.6819273
)
}
```

Ces formes exactes ne sont pas garanties et changent parfois d'une version de Rust à l'autre.

Le formatage de débogage imprime généralement les nombres en décimal, mais vous pouvez mettre un `x` ou `X` avant le point d'interrogation pour demander une valeur hexadécimale à la place. La syntaxe du zéro non significatif et de la largeur de champ est également respectée. Par exemple, vous pouvez écrire :

```
println!("ordinary: {:02?}", [9, 15, 240]);
println!("hex:       {:02x?}", [9, 15, 240]);
```

Cela imprime :

```
ordinary: [09, 15, 240]
hex:       [09, 0f, f0]
```

Comme nous l'avons mentionné, vous pouvez utiliser la `#[derive(Debug)]` syntaxe pour faire fonctionner vos propres types avec `{:?:}` :

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im:f64 }
```

Avec cette définition en place, nous pouvons utiliser un `{:?:}` format pour imprimer les `Complex` valeurs :

```
let third = Complex { re: -0.5, im: f64::sqrt(0.75) };
println!("{:?}", third);
```

Cela imprime :

```
Complex { re: -0.5, im: 0.8660254037844386 }
```

C'est bien pour le débogage, mais ce serait bien si {} on pouvait les imprimer sous une forme plus traditionnelle, comme -0.5 + 0.8660254037844386i . Dans "[Formater vos propres types](#)" , nous montrerons comment faire exactement cela.

Formatage des pointeurs pour le débogage

Normalement, si vous passez n'importe quel type de pointeur à une macro de formatage — une référence, un Box , un Rc — la macro suit simplement le pointeur et formate son référent ; le pointeur lui-même n'a pas d'intérêt. Mais lorsque vous déboguez, il est parfois utile de voir le pointeur : une adresse peut servir de "nom" approximatif pour une valeur individuelle, ce qui peut être éclairant lors de l'examen de structures avec des cycles ou du partage.

La {:p} notation formate les références, les boîtes et autres types de type pointeur en tant qu'adresses :

```
use std:: rc:: Rc;
let original = Rc:: new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text: {}, {}, {}", original, cloned, impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned, impostor);
```

Ce code imprime :

```
text: mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

Bien sûr, les valeurs de pointeur spécifiques varient d'une exécution à l'autre, mais même ainsi, la comparaison des adresses montre clairement que les deux premières sont des références au même String , tandis que la troisième pointe vers une valeur distincte.

Les adresses ont tendance à ressembler à de la soupe hexadécimale, donc des visualisations plus raffinées peuvent être utiles, mais le {:p} style peut toujours être une solution efficace et rapide.

Faire référence aux arguments par index ou par nom

Un paramètre de format peut sélectionner explicitement quel argument il utilise. Par exemple:

```
assert_eq!(format!("{}{},{}{}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

Vous pouvez inclure des paramètres de format après deux-points :

```
assert_eq!(format!("{}:{}{:b},{}{:>10}", "first", 10, 100),
           "0x0064,1010,=====first");
```

Vous pouvez également sélectionner des arguments par leur nom. Cela rend les modèles complexes avec de nombreux paramètres beaucoup plus lisibles. Par exemple:

```
assert_eq!(format!("{}description:<25}{}quantity:2} @ {}price:5.2}",
           price=3.25,
           quantity=3,
           description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @ 3.25");
```

(Les arguments nommés ici ressemblent aux arguments de mots-clés en Python, mais il ne s'agit que d'une fonctionnalité spéciale des macros de formatage, qui ne fait pas partie de la syntaxe d'appel de fonction de Rust.)

Vous pouvez mélanger des paramètres indexés, nommés et positionnels (c'est-à-dire sans index ni nom) dans une seule macro de mise en forme. Les paramètres positionnels sont associés à des arguments de gauche à droite comme si les paramètres indexés et nommés n'étaient pas là :

```
assert_eq!(format!("{}mode} {} {} {}", "people", "eater", "purple", mode="flying"),
           "flying purple people eater");
```

Les arguments nommés doivent apparaître à la fin de la liste.

Largeurs et précisions dynamiques

Un paramètre `width` minimale la largeur, la limite de longueur du texte et la précision numérique ne doivent pas toujours être des valeurs fixes ; vous pouvez les choisir au moment de l'exécution.

Nous avons examiné des cas comme cette expression, qui vous donne la chaîne `content` justifiée à droite dans un champ de 20 caractères :

```
format!("{:>20}", content)
```

Mais si vous souhaitez choisir la largeur du champ au moment de l'exécution, vous pouvez écrire :

```
format!("{:>1$}", content, get_width())
```

L'écriture `1$` pour la largeur de champ minimale indique `format!` d'utiliser la valeur du deuxième argument comme largeur. L'argument cité doit être un `usize`. Vous pouvez également faire référence à l'argument par son nom :

```
format!("{:>width$}", content, width=get_width())
```

La même approche fonctionne également pour la limite de longueur du texte :

```
format!("{:>width$.limit$}", content,
       width=get_width(), limit=get_limit())
```

À la place de la limite de longueur du texte ou de la précision en virgule flottante, vous pouvez également écrire `*`, qui indique de prendre le prochain argument positionnel comme précision. Les clips suivants `content` à la plupart `get_limit()` des personnages :

```
format!("{:.*}", get_limit(), content)
```

L'argument pris comme précision doit être un `usize`. Il n'y a pas de syntaxe correspondante pour la largeur de champ.

Formater vos propres types

Les macros de formatage utilisent un ensemble de traits définis dans le `std::fmt` module pour convertir des valeurs en texte. Vous pouvez faire en sorte que les macros de formatage de Rust formatent vos propres types en implémentant un ou plusieurs de ces traits vous-même.

La notation d'un paramètre de format indique quel trait le type de son argument doit implémenter, comme illustré dans le [Tableau 17-8](#).

Tableau 17-8. Notation de directive de chaîne de format

Notation	Exemple	Caractéristique	Objectif
rien	{ }	std::fmt::Display	Texte, chiffres, erreurs : le trait fourre-tout
b	{bits: #b}	std::fmt::Binary	Nombres en binaire
o	{:#5o}	std::fmt::Octal	Nombres en octal
x	{:4x}	std::fmt::LowerHex	Nombres en hexadécimal, chiffres minuscules
X	{:016X}	std::fmt::UpperHex	Nombres en hexadécimal, chiffres majuscules
e	{:.3e}	std::fmt::LowerExp	Nombres à virgule flottante en notation scientifique
E	{:.3E}	std::fmt::UpperExp	Idem, majuscule E
?	{:#?}	std::fmt::Debug	Vue de débogage, pour les développeurs
p	{:p}	std::fmt::Pointer	Pointeur comme adresse, pour les développeurs

Lorsque vous placez l' `#[derive(Debug)]` attribut sur une définition de type afin de pouvoir utiliser le `{:#?}` paramètre de format, vous demandez simplement à Rust d'implémenter le `std::fmt::Debug` trait pour vous.

Les traits de formatage ont tous la même structure, ne différant que par leurs noms. Nous utiliserons `std::fmt::Display` comme représentant :

```
trait Display {  
    fn fmt(&self, dest: &mut std::fmt::Formatter)  
        -> std::fmt::Result;  
}
```

Le `fmt` travail de la méthode consiste à produire une représentation correctement formatée de `self` et à écrire ses caractères dans `dest`. En plus de servir de flux de sortie, l' `dest` argument contient également des détails analysés à partir du paramètre de format, comme l'alignement et la largeur de champ minimale.

Par exemple, plus tôt dans ce chapitre, nous avons suggéré qu'il serait bien que `Complex` les valeurs s'impriment elles-mêmes sous la `a + bi` forme habituelle. Voici une `Display` implémentation qui fait cela :

```
use std::fmt;  
  
impl fmt::Display for Complex {  
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {  
        let im_sign = if self.im < 0.0 { '-' } else { '+' };  
        write!(dest, "{} {} {}i", self.re, im_sign, f64::abs(self.im))  
    }  
}
```

Cela tire parti du fait qu'il `Formatter` s'agit lui-même d'un flux de sortie, de sorte que la `write!` macro peut faire la majeure partie du travail pour nous. Avec cette implémentation en place, nous pouvons écrire ce qui suit :

```
let one_twenty = Complex { re: -0.5, im: 0.866 };  
assert_eq!(format!("{}", one_twenty),  
          "-0.5 + 0.866i");  
  
let two_forty = Complex { re: -0.5, im: -0.866 };  
assert_eq!(format!("{}", two_forty),  
          "-0.5 - 0.866i");
```

Il est parfois utile d'afficher les nombres complexes sous forme polaire : si vous imaginez une ligne tracée sur le plan complexe de l'origine au nombre, la forme polaire donne la longueur de la ligne et son angle dans le sens des aiguilles d'une montre par rapport à l'axe des x positif. Le

caractère dans un paramètre de format sélectionne généralement une autre forme d'affichage ; l' `Display` implémentation pourrait le traiter comme une demande d'utilisation de la forme polaire :

```
impl fmt:: Display for Complex {
    fn fmt(&self, dest: &mut fmt:: Formatter) -> fmt:: Result {
        let (re, im) = (self.re, self.im);
        if dest.alternate() {
            let abs = f64:: sqrt(re * re + im * im);
            let angle = f64:: atan2(im, re) / std:: f64:: consts:: PI * 18
            write!(dest, "{} < {}°", abs, angle)
        } else {
            let im_sign = if im < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", re, im_sign, f64::abs(im))
        }
    }
}
```

En utilisant cette implémentation :

```
let ninety = Complex { re: 0.0, im:2.0 };
assert_eq!(format!("{}", ninety),
           "0 + 2i");
assert_eq!(format!("{:#}", ninety),
           "2 < 90°");
```

Bien que les méthodes des traits de formatage `fmt` renvoient une `fmt::Result` valeur (un type typique spécifique au module `Result`), vous ne devez propager les échecs qu'à partir des opérations sur le `Formatter`, comme le `fmt::Display` fait l'implémentation avec ses appels à `write!` ; vos fonctions de formatage ne doivent jamais générer elles-mêmes des erreurs. Cela permet aux macros `format!` de renvoyer simplement a `String` au lieu de a `Result<String, ...>`, car l'ajout du texte formaté à a `String` n'échoue jamais. Cela garantit également que toutes les erreurs que vous obtenez `write!` ou `writeln!` reflètent de vrais problèmes du flux d'E / S sous-jacent, et non des problèmes de formatage.

`Formatter` a beaucoup d'autres méthodes utiles, y compris certaines pour gérer des données structurées comme des cartes, des listes, etc., que nous n'aborderons pas ici ; consultez la documentation en ligne pour tous les détails.

Utilisation du langage de formatage dans votre propre code

Vous pouvez écrire vos propres fonctions et les macros qui acceptent les modèles de format et les arguments en utilisant la `format_args!` macro de Rust et le `std::fmt::Arguments` type. Par exemple, supposons que votre programme doive se connecter aux messages d'état pendant son exécution, et vous souhaitez utiliser le langage de formatage de texte de Rust pour les produire. Ce qui suit serait un début :

```
fn logging_enabled() ->bool { ... }

use std::fs::OpenOptions;
use std::io::Write;

fn write_log_entry(entry: std::fmt::Arguments) {
    if logging_enabled() {
        // Keep things simple for now, and just
        // open the file every time.
        let mut log_file = OpenOptions::new()
            .append(true)
            .create(true)
            .open("log-file-name")
            .expect("failed to open log file");

        log_file.write_fmt(entry)
            .expect("failed to write to log");
    }
}
```

Vous pouvez appeler `write_log_entry` ainsi :

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

Au moment de la compilation, la `format_args!` macro analyse la chaîne du modèle et la compare aux types des arguments, signalant une erreur en cas de problème. Au moment de l'exécution, il évalue les arguments et construit une `Arguments` valeur contenant toutes les informations nécessaires pour formater le texte : une forme pré-parsée du modèle, ainsi que des références partagées aux valeurs des arguments.

Construire une `Arguments` valeur n'est pas cher : c'est juste rassembler quelques pointeurs. Aucun travail de formatage n'a encore lieu, seulement la collecte des informations nécessaires pour le faire plus tard. Cela peut être important : si la journalisation n'est pas activée, tout le temps

passé à convertir des nombres en décimal, à remplir des valeurs, etc. serait perdu.

Le `File` type implémente le `std::io::Write` trait, dont la `write_fmt` méthode prend un `Argument` et effectue le formatage. Il écrit les résultats dans le flux sous-jacent.

Cet appel `write_log_entry` n'est pas joli. C'est là qu'une macro peut aider :

```
macro_rules! log { // no ! needed after name in macro definitions
    ($format: tt, $($arg:expr),*) => (
        write_log_entry(format_args!($format, $($arg),*))
    )
}
```

Nous couvrons les macros en détail au [chapitre 21](#). Pour l'instant, croyez que cela définit une nouvelle `log!` macro qui transmet ses arguments à `format_args!` puis appelle votre `write_log_entry` fonction sur la `Arguments` valeur résultante. Les macros de formatage comme `println!`, `writeln!` et `format!` sont toutes à peu près la même idée.

Vous pouvez utiliser `log!` comme ceci :

```
log!("O day and night, but this is wondrous strange! {:?}", mysterious_value);
```

Idéalement, cela semble un peu mieux.

Expressions régulières

`regex` La caisse externe est l'expression régulière officielle de Rust bibliothèque. Il fournit les fonctions habituelles de recherche et de correspondance. Il prend bien en charge Unicode, mais il peut également rechercher des chaînes d'octets. Bien qu'il ne prenne pas en charge certaines fonctionnalités que vous trouverez souvent dans d'autres packages d'expressions régulières, comme les références arrière et les modèles de recherche, ces simplifications permettent `regex` de garantir que les recherches prennent un temps linéaire dans la taille de l'expression et dans la longueur du texte étant recherché. Ces garanties, entre autres, garantissent `regex` une utilisation sûre même avec des expressions non fiables recherchant du texte non fiable.

Dans ce livre, nous ne fournirons qu'un aperçu de `regex` ; vous devriez consulter sa documentation en ligne pour plus de détails.

Bien que la `regex` caisse ne soit pas dans `std`, elle est entretenue par l'équipe de la bibliothèque Rust, le même groupe responsable de `std`. Pour utiliser `regex`, placez la ligne suivante dans la `[dependencies]` section du fichier `Cargo.toml` de votre caisse :

```
expression régulième = "1"
```

Dans les sections suivantes, nous supposerons que vous avez mis en place ce changement.

Utilisation de base des expressions régulières

Une `Regex` valeur représente une expression régulière analysée prête à l'emploi. Le `Regex::new` constructeur essaie d'analyser `a &str` comme une expression régulière et renvoie `a Result`:

```
use regex::Regex;

// A semver version number, like 0.2.1.
// May contain a pre-release version suffix, like 0.2.1-alpha.
// (No build metadata suffix, for brevity.)
//
// Note use of r"..." raw string syntax, to avoid backslash blizzard.
let semver = Regex::new(r"(\d+)\.( \d+)\.( \d+)([-.[:alnum:]]*)?")?;

// Simple search, with a Boolean result.
let haystack = r#"regex = "0.2.5" "#;
assert!(semver.is_match(haystack));
```

La `Regex::captures` méthode recherche une chaîne pour la première correspondance et renvoie une `regex::Captures` valeur contenant des informations de correspondance pour chaque groupe dans l'expression :

```
// You can retrieve capture groups:
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

L' indexation d'une `Captures` valeur panique si le groupe demandé ne correspond pas. Pour tester si un groupe particulier correspond, vous pouvez appeler `Captures::get`, qui renvoie un `Option<regex::Match>`. Une `Match` valeur enregistre la correspondance d'un seul groupe :

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

Vous pouvez parcourir toutes les correspondances d'une chaîne :

```
let haystack = "In the beginning, there was 1.0.0. \
    For a while, we used 1.0.1-beta, \
    but in the end, we settled on 1.2.4.";

let matches:Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

L' `find_iter` itérateur produit une `Match` valeur pour chaque correspondance sans chevauchement de l'expression, du début à la fin de la chaîne. La `Captures_iter` méthode est similaire, mais produit des `Captures` valeurs enregistrant tous les groupes de capture. La recherche est plus lente lorsque les groupes de capture doivent être signalés, donc si vous n'en avez pas besoin, il est préférable d'utiliser l'une des méthodes qui ne les renvoie pas.

Construire des valeurs Regex paresseusement

Le `Regex::new` constructeur peut être coûteux : la construction `Regex` d'une expression régulière de 1 200 caractères peut prendre près d'une milliseconde sur une machine de développement rapide, et même une expression triviale prend des microsecondes. Il est préférable de garder la `Regex` construction hors des boucles de calcul lourdes ; au lieu de cela, vous devez construire votre `Regex` une fois, puis réutiliser le même.

La `lazy_static` caisse fournit un bon moyen de construire des valeurs statiques paresseusement la première fois qu'ils sont utilisés. Pour commencer, notez la dépendance dans votre fichier `Cargo.toml` :

```
[dépendances]
paresseux_statique = "1"
```

Ce crate fournit une macro pour déclarer de telles variables :

```
use lazy_static::lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.( \d+)\.( \d+)([-.[:alnum:]]*)?")
            .expect("error parsing regex");
}
```

La macro se développe en une déclaration d'une variable statique nommée `SEMVER`, mais son type n'est pas exactement `Regex`. Au lieu de cela, il s'agit d'un type généré par macro qui implémente `Deref<Target=Regex>` et expose donc toutes les mêmes méthodes qu'un `Regex`. La première fois `SEMVER` est déréférencée, l'initialiseur est évalué et la valeur est enregistrée pour une utilisation ultérieure. Puisqu'il `SEMVER` s'agit d'une variable statique, et pas seulement d'une variable locale, l'initialiseur s'exécute au plus une fois par exécution du programme.

Avec cette déclaration en place, l'utilisation `SEMVER` est simple :

```
use std:: io::BufRead;

let stdin = std:: io::stdin();
for line_result in stdin.lock().lines() {
    let line = line_result?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}" , match_.as_str());
    }
}
```

Vous pouvez mettre la `lazy_static!` déclaration dans un module, ou même à l'intérieur de la fonction qui utilise le `Regex`, si c'est la portée la plus appropriée. L'expression régulière est toujours compilée une seule fois par exécution du programme.

Normalisation

La plupart des utilisateurs considéreraient que le mot français pour thé, *thé*, comporte trois caractères. Cependant, Unicode en fait deux façons de représenter ce texte :

- Dans la forme *composée*, « *thé* » comprend les trois caractères 't', 'h', et 'é', où 'é' est un seul caractère Unicode avec le point de code `0xe9`.
- Dans sa forme *décomposée*, "thé" comprend les quatre caractères 't', 'h', 'e', et '\u{301}', où le 'e' est le caractère ASCII simple, sans accent, et le point de code `0x301` est le caractère "COMBINING ACUTE ACCENT", qui ajoute un accent aigu à tout caractère qu'il suit.

Unicode ne considère ni la forme composée ni la forme décomposée de é comme étant la « correcte » ; il les considère plutôt comme des représentations équivalentes du même caractère abstrait. Unicode indique que les deux formulaires doivent être affichés de la même manière et que les méthodes de saisie de texte sont autorisées à produire l'un ou l'autre, de sorte que les utilisateurs ne sauront généralement pas quel formulaire ils visualisent ou tapent. (Rust vous permet d'utiliser des caractères Unicode directement dans les littéraux de chaîne, vous pouvez donc simplement écrire "thé" si vous ne vous souciez pas de l'encodage que vous obtenez. Ici, nous utiliserons les \u échappements pour plus de clarté.)

Cependant, considérés comme Rust `&str` ou `String` valeurs, "th\u{e9}" et "the\u{301}" sont complètement distincts. Ils ont des longueurs différentes, se comparent comme inégaux, ont des valeurs de hachage différentes et s'ordonnent différemment par rapport aux autres chaînes :

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" > "the\u{301}");

// A Hasher is designed to accumulate the hash of a series of values,
// so hashing just one is a bit clunky.
use std::hash:: {Hash, Hasher};
use std::collections:: hash_map:: DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}

// These values may change in future Rust releases.
assert_eq!(hash("th\u{e9}"), 0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

De toute évidence, si vous avez l'intention de comparer du texte fourni par l'utilisateur ou de l'utiliser comme clé dans une table de hachage ou un arbre B, vous devrez d'abord mettre chaque chaîne sous une forme canonique.

Heureusement, Unicode spécifie des formes *normalisées pour les chaînes*. Chaque fois que deux chaînes doivent être traitées comme équivalentes selon les règles d'Unicode, leurs formes normalisées sont identiques caractère pour caractère. Lorsqu'ils sont encodés avec UTF-8, ils sont identiques octet par octet. Cela signifie que vous pouvez comparer des chaînes normalisées avec `==`, les utiliser comme clés dans un `HashMap` ou `HashSet`, etc., et vous obtiendrez la notion d'égalité d'Unicode.

L'absence de normalisation peut même avoir des conséquences sur la sécurité. Par exemple, si votre site Web normalise les noms d'utilisateur dans certains cas mais pas dans d'autres, vous pourriez vous retrouver avec deux utilisateurs distincts nommés `bananasflambé`, que certaines parties de votre code traitent comme le même utilisateur, mais que d'autres distinguent, ce qui entraînerait l'extension incorrecte de ses priviléges au autre. Bien sûr, il existe de nombreuses façons d'éviter ce genre de problème, mais l'histoire montre qu'il existe également de nombreuses façons de ne pas le faire.

Formulaires de normalisation

Unicode définit quatre formes normalisées, dont chacune est appropriée pour différentes utilisations. Il y a deux questions auxquelles répondre :

- Premièrement, préférez-vous que les personnages soient aussi *composés* que possible ou aussi *décomposés* que possible ?

Par exemple, la représentation la plus composée du mot vietnamien *Phở* est la chaîne de trois caractères "`Ph\u00e1`", où la marque tonale ' et la marque de voyelle ' sont appliquées au caractère de base "o" dans un seul caractère Unicode, '`\u00e1`', que Unicode nomme consciencieusement LETTRE MINUSCULE LATINE O AVEC CORNE ET CROCHET AU-DESSUS.

La représentation la plus décomposée divise la lettre de base et ses deux marques en trois caractères Unicode distincts : 'o', '`\u031b`' (COMBINING HORN) et '`\u0309`' (COMBINING HOOK ABOVE), ce qui donne "`Pho\u031b\u0309`". (Chaque fois que des marques combinées apparaissent comme des caractères séparés, plutôt que comme faisant partie d'un caractère composé, toutes les formes normalisées spécifient un ordre fixe dans lequel elles doivent

apparaître, de sorte que la normalisation est bien spécifiée même lorsque les caractères ont plusieurs accents.)

La forme composée a généralement moins de problèmes de compatibilité, car elle correspond plus étroitement aux représentations que la plupart des langues utilisaient pour leur texte avant l'établissement d'Unicode. Cela peut également mieux fonctionner avec des fonctionnalités de formatage de chaîne naïves comme la `format!` macro de Rust. La forme décomposée, en revanche, peut être plus adaptée à l'affichage du texte ou à la recherche, car elle rend la structure détaillée du texte plus explicite.

- La deuxième question est la suivante : si deux séquences de caractères représentent le même texte fondamental mais diffèrent dans la manière dont le texte doit être formaté, voulez-vous les traiter comme équivalentes ou les garder distinctes ?

Unicode a des caractères séparés pour le chiffre ordinaire `5`, le chiffre en exposant ⁵ (ou '`\u{2075}`') et le chiffre encerclé `⑤` (ou '`\u{2464}`'), mais déclare que les trois sont *équivalents en termes de compatibilité*. De même, Unicode a un seul caractère pour la ligature `ffi` (`\u{fb03}`), mais déclare qu'il s'agit d'une compatibilité équivalente à la séquence de trois caractères `ffi`.

L'équivalence de compatibilité est logique pour les recherches : une recherche de "difficult", en utilisant uniquement des caractères ASCII, doit correspondre à la chaîne "di\u{fb03}cult", qui utilise la ligature `ffi`. L'application de la décomposition de compatibilité à cette dernière chaîne remplacerait la ligature par les trois lettres simples "ffi", ce qui faciliterait la recherche. Mais la normalisation du texte dans une forme équivalente de compatibilité peut perdre des informations essentielles, elle ne doit donc pas être appliquée avec négligence. Par exemple, il serait incorrect dans la plupart des contextes de stocker "2⁵" en tant que "25".

La forme de normalisation Unicode C et la forme de normalisation D (NFC et NFD) utilisent les formes composées au maximum et décomposées au maximum de chaque caractère, mais n'essayez pas d'unifier les séquences équivalentes de compatibilité. Les formes de normalisation NFKC et NFKD sont comme NFC et NFD, mais normalisent toutes les séquences équivalentes de compatibilité à un simple représentant de leur classe.

Le « Modèle de personnage pour le World Wide Web » du World Wide Web Consortium recommande d'utiliser NFC pour tout le contenu. L'annexe Unicode Identifier and Pattern Syntax suggère d'utiliser NFKC pour les identifiants dans les langages de programmation et propose des principes pour adapter la forme si nécessaire.

La caisse de normalisation unicode

unicode-normalization La caisse de Rust fournit un trait qui ajoute des méthodes pour `&str` mettre le texte dans l'une des quatre formes normalisées. Pour l'utiliser, ajoutez la ligne suivante à la `[dependencies]` section de votre fichier `Cargo.toml` :

```
normalisation unicode = "0.1.17"
```

Avec cette déclaration en place, `a &str` a quatre nouvelles méthodes qui renvoient des itérateurs sur une forme normalisée particulière de la chaîne :

```
use unicode_normalization::UnicodeNormalization;

// No matter what representation the left-hand string uses
// (you shouldn't be able to tell just by looking),
// these assertions will hold.
assert_eq!("Phő".nfd().collect::<String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phő".nfc().collect::<String>(), "Ph\u{1edf}");

// The left-hand side here uses the "ffi" ligature character.
assert_eq!(① Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficulty")
```

Prendre une chaîne normalisée et la normaliser à nouveau sous la même forme est garanti pour renvoyer un texte identique.

Bien que toute sous-chaîne d'une chaîne normalisée soit elle-même normalisée, la concaténation de deux chaînes normalisées n'est pas nécessairement normalisée : par exemple, la deuxième chaîne peut commencer par des caractères de combinaison qui doivent être placés avant les caractères de combinaison à la fin de la première chaîne.

Tant qu'un texte n'utilise aucun point de code non attribué lorsqu'il est normalisé, Unicode promet que sa forme normalisée ne changera pas dans les futures versions de la norme. Cela signifie que les formulaires normalisés peuvent généralement être utilisés en toute sécurité dans un stockage persistant, même si la norme Unicode évolue.

Chapitre 18. Entrée et sortie

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

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

Doolittle : C'est bien. C'est très bien. Mais comment savez-vous que quelque chose d'autre existe?

Bombe #20 : Mon appareil sensoriel me le révèle.

—Étoile Noire

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

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

[La figure 18-1](#) montre ces trois traits et quelques exemples de types de lecteurs et d'écrivains.

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

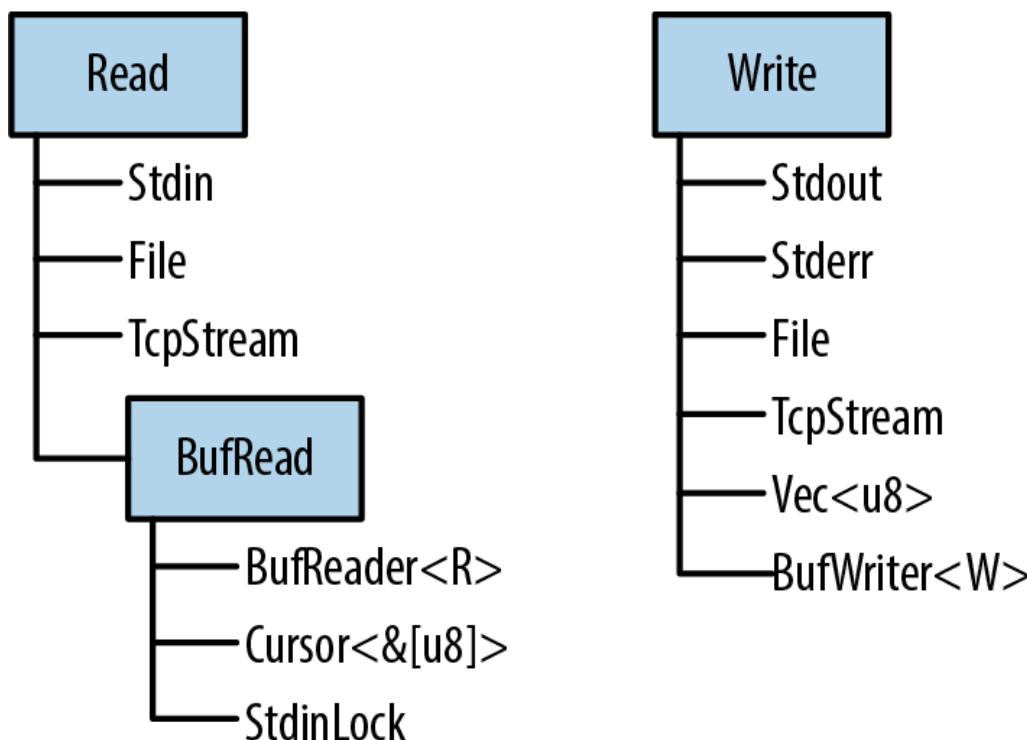


Image 18-1. Les trois principaux traits d'E/S de Rust et les types sélectionnés qui les implémentent

Lecteurs et écrivains

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

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

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

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

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

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

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

const DEFAULT_BUF_SIZE:usize = 8 * 1024;

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

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

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

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

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


```

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

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

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

Lecteurs

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

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

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

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

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

`io::ErrorKind::Interrupted` correspond au code d'erreur Unix `EINTR`, ce qui signifie que la lecture a été interrompue par un signal. À moins que le programme ne soit conçu pour faire quelque chose d'intelligent avec les signaux, il devrait simplement réessayer la lecture. Le code de `copy()`, dans la section précédente, en montre un exemple.

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

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

`reader.read_to_end(&mut byte_vec)`

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

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

`reader.read_to_string(&mut string)`

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

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

`reader.read_exact(&mut buf)`

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

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

`reader.bytes()`

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

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

`reader.chain(reader2)`

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

`reader.take(n)`

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

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

Lecteurs tamponnés

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

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

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

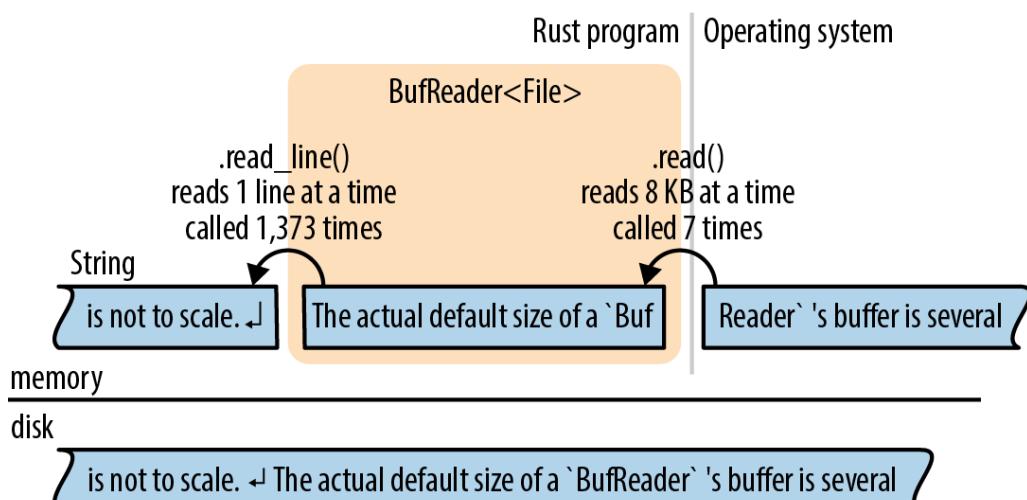


Image 18-2. Un lecteur de fichiers tamponné

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

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

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

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

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

```
reader.lines()
```

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

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

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

Ces deux méthodes fonctionnent comme `.read_line()` et `.lines()`, mais orientées octets, produisant `Vec<u8>`s au lieu de `String`s. Vous choisissez le délimiteur `stop_byte`.

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

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

Lignes de lecture

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

mande, pour une chaîne donnée :

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

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

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

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

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

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

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

Maintenant, nous pouvons lui passer un `StdinLock` ou un `buffered File`:

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

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

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

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

Le programme complet, y compris la gestion des erreurs et une analyse grossière des arguments, est présenté ici:

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

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

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

```

    }

    Ok(())
}

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

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

Ok(())
}

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

```

Lignes de collecte

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

```

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

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

```

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

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

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

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

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

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

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

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

Écrivains

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

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

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

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

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

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

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

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

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

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

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

Le `Write` trait a ces méthodes:

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

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

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

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

Écrit tous les octets de la trame `buf`. Retours `Result<()>`.

```
writer.flush()
```

bouffées de chaleur toutes les données mises en mémoire tampon au flux sous-jacent. Retours `Result<()>`.

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

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

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

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

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

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

Des dossiers

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

`File::open(filename)`

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

`File::create(filename)`

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

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

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

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

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

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

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

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

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

En cherchant

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

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

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

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

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

Autres types de lecteurs et d'enregistreurs

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

`io::stdin()`

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

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

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

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

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

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

`Vec<u8>`

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

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

`Cursor::new(buf)`

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

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

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

`std::net::TcpStream`

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

La fonction associée au type

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

`std::process::Command`

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

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

```
let mut child =
```

```

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

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

```

Le type de `child.stdin` est

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

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

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

`io::sink()`

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

`io::empty()`

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

`io::repeat(byte)`

Renvoie un lecteur qui répète indéfiniment l'octet donné.

Données binaires, compression et sérialisation

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

La `byteorder` caisse offre `ReadBytesExt` et `WriteBytesExt` caractéristiques qui ajoutent des méthodes à tous les lecteurs et écrivains pour le binaire entrée et sortie :

```

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

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

```

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

```

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

```

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

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

```

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

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

La sortie ressemble à ceci:

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

Fichiers et répertoires

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

OsStr et Chemin

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

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

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

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

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

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

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

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

Tableau 18-1. Types de noms de fichiers

	chaîne	OsStr	Chemin
Type non dimensionné, toujours passé par référence	Oui	Oui	Oui
Peut contenir n'importe quel texte Unicode	Oui	Oui	Oui
Ressemble à UTF-8, normalement	Oui	Oui	Oui
Peut contenir des données non Unicode	Non	Oui	Oui
Méthodes de traitement de texte	Oui	Non	Non
Méthodes liées aux noms de fichiers	Non	Non	Oui
Équivalent détenu, extensible et alloué en tas	<code>String</code>	<code>OsString</code>	<code>PathBuf</code>
Convertir en type possédé	<code>.to_string()</code>	<code>.to_os_string()</code>	<code>.to_path_buf()</code>

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

```

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

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

```

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

Méthodes Path et PathBuf

`Path` propose les méthodes suivantes, entre autres :

Path::new(str)

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

```

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

```

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

path.parent()

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

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

```

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

```

path.file_name()

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

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

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

path.is_absolute(), path.is_relative()
```

Ce s'indique si le fichier est absolu, comme le chemin Unix `/usr/bin/advent` ou le chemin Windows `C:\Program Files`, ou relatif, comme `src/main.rs`.

```
path1.join(path2)
```

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

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

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

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

path.components()
```

Retourne un itérateur sur les composants du chemin donné, de gauche à droite. Le type d'élément de cet itérateur est

`std::path::Component`, une énumération qui peut représenter tous les différents éléments pouvant apparaître dans les noms de fichiers :

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

Par exemple, le chemin Windows `||venice|Music|A Love Supreme|04-Psalm.mp3` se compose d'un `Prefix` représentant `||ve-`

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

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

`path.ancestors()`

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

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

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

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

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

`path.to_str()`

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

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

`path.to_string_lossy()`

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

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

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

`path.display()`

Cette est pour les chemins d'impression :

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

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

Fonctions d'accès au système de fichiers

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

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

Fonction rouille	Unix	les fenêtres
<code>create_dir(path)</code>	<code>mkdir()</code>	<code>CreateDirectory()</code>
<code>create_dir_all(path)</code>	<code>mkdir -p</code>	<code>CreateDirectory()</code>
<code>remove_dir(path)</code>	<code>rmdir()</code>	<code>RemoveDirectory()</code>
<code>remove_dir_all(path)</code>	<code>rmdir -r</code>	<code>RemoveDirectory()</code>
<code>remove_file(path)</code>	<code>unlink()</code>	<code>DeleteFile()</code>
Création et suppression		
<code>copy(src_path, dest_path) -> Result<u64></code>	<code>copy -p</code>	<code>CopyFileEx()</code>
Copier, déplacer et lier Inspecter	<code>rename(src_path, dest_path)</code>	<code>MoveFileEx()</code>
	<code>hard_link(src_path, dest_path)</code>	<code>CreateHardLink()</code>
	<code>canonicalize(path) -> Result<PathBuf></code>	<code>GetFinalPathNameByHandle()</code>
	<code>metadata(path) -> Result<Metadata></code>	<code>GetFileInformationByHandle()</code>
<code>symlink_metadata(path) -> Result<Metadata></code>	<code>lstat()</code>	<code>GetFileInformationByHandle()</code>
<code>read_dir(path) -> Result<ReadDir></code>	<code>opendir()</code>	<code>FindFirstFile()</code>

Fonction rouille	Unix	les fenêtres
read_link(path) -> Result<PathBuf>	readlink ()	FSCTL_GET_R EPARSE_POINTER
set_permissions(path, perm)	chmod ()	SetFileAttributes ()
Autorisations		

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

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

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

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

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

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

Répertoires de lecture

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

```

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

```

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

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

`entry.file_name()`

La nom du fichier ou du répertoire, sous la forme d'un `OSSString`.

`entry.path()`

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

`entry.file_type()`

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

`entry.metadata()`

Obtient le reste des métadonnées sur cette entrée.

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

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

```

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

/// Copy the existing directory `src` to the target path `dst`.
fn copy_dir_to(src: &Path, dst: &Path) -> io:: Result<()> {
    if !dst.is_dir() {
        fs::create_dir(dst)?;
    }
}

```

```

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

        Ok(())
    }
}

```

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

```

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

```

Fonctionnalités spécifiques à la plate-forme

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

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

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

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

```

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

```

```
let target = src.read_link()?;
symlink(target, dst)?;
...
```

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

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

```
//! OS-specific functionality.

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

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

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

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

/// Stub implementation of `symlink` for platforms that don't provide it.
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(src: P, _dst: Q)
    -> std:: io:: Result<()
{
    Err(io:: Error:: new(io:: ErrorKind::Other,
        format!("can't copy symbolic link: {}", src.as_ref().display())))
}
```

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

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


```

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

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

La mise en réseau

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

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

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

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

/// Accept connections forever, spawning a thread for each one.
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
```

```

    println!("listening on {}", addr);
    loop {
        // Wait for a client to connect.
        let (mut stream, addr) = listener.accept()?;
        println!("connection received from {}", addr);

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

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

```

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

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

Les protocoles de niveau supérieur sont pris en charge par des caisses tierces. Par exemple, la `reqwest` offre une belle API pour les clients HTTP. Voici un programme complet en ligne de commande qui récupère tout document avec une URL `http:` ou le vide sur votre terminal.

`https:` Ce code a été écrit en utilisant `reqwest = "0.11"`, avec sa "blocking" fonctionnalité activée. `reqwest` fournit également une interface asynchrone.

```

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

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

```

```

    }

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

    Ok(())
}

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

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

```

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

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 19. Concurrence

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

—Par Brinch Hansen (1977)

Les modèles de communication sont des modèles de parallélisme.

—Whit Morris

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

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

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

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

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

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

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

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

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

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

- Parallélisme fourche-jointure
- Canaux
- État mutable partagé

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

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

Parallélisme fork-join

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

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

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

Le programme s'exécute comme illustré à la [Figure 19-1](#).

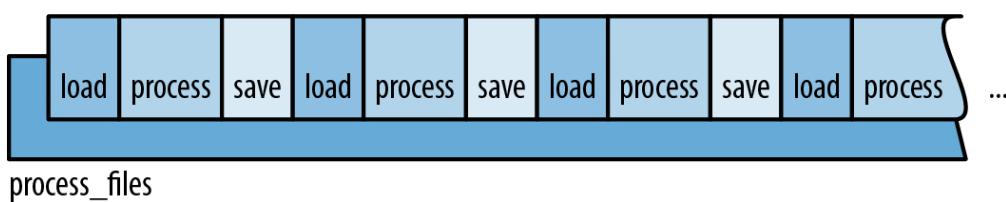


Image 19-1. Exécution monothread de `process_files()`

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

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

Le parallélisme fork-join est attrayant pour plusieurs raisons :

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

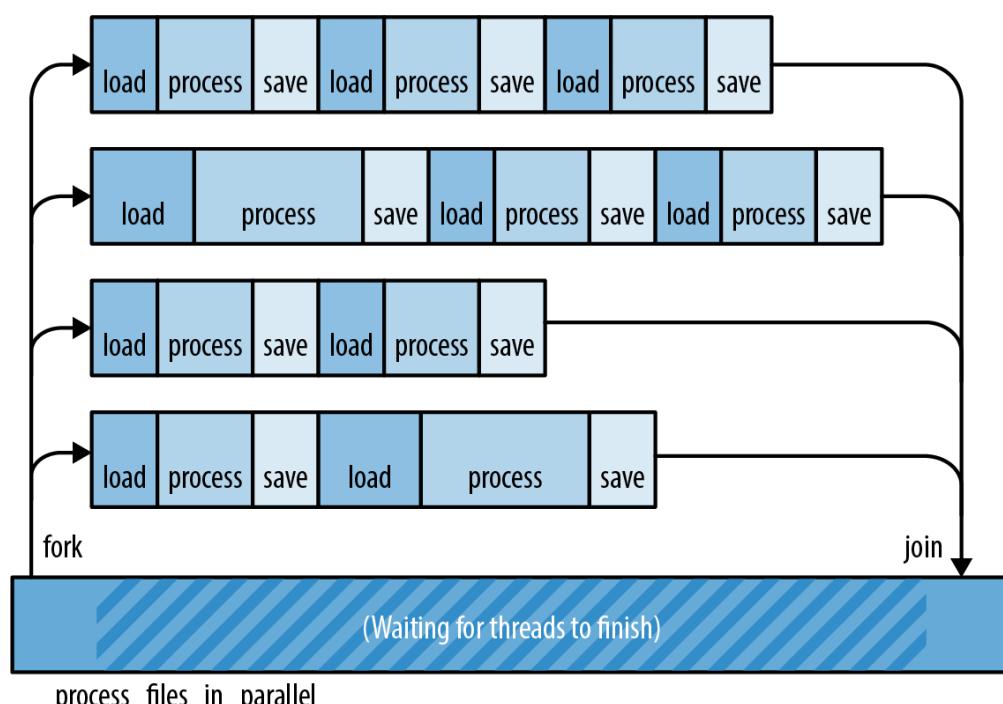


Image 19-2. Traitement de fichiers multithread à l'aide d'une approche fork-join

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

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

frayer et rejoindre

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

```
use std::thread;

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

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

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

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

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

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

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

    Ok(())
}
```

Prenons cette fonction ligne par ligne.

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

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

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

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

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

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

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

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

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

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

Passons à autre chose :

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

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

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

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

Gestion des erreurs dans les threads

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

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

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

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

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

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

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

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

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

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

Partage de données immuables entre les threads

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

```

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

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

```

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

Le changement évident ne fonctionne pas :

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

menter un compteur de références.

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

Rayonne

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

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

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

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

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

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

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

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

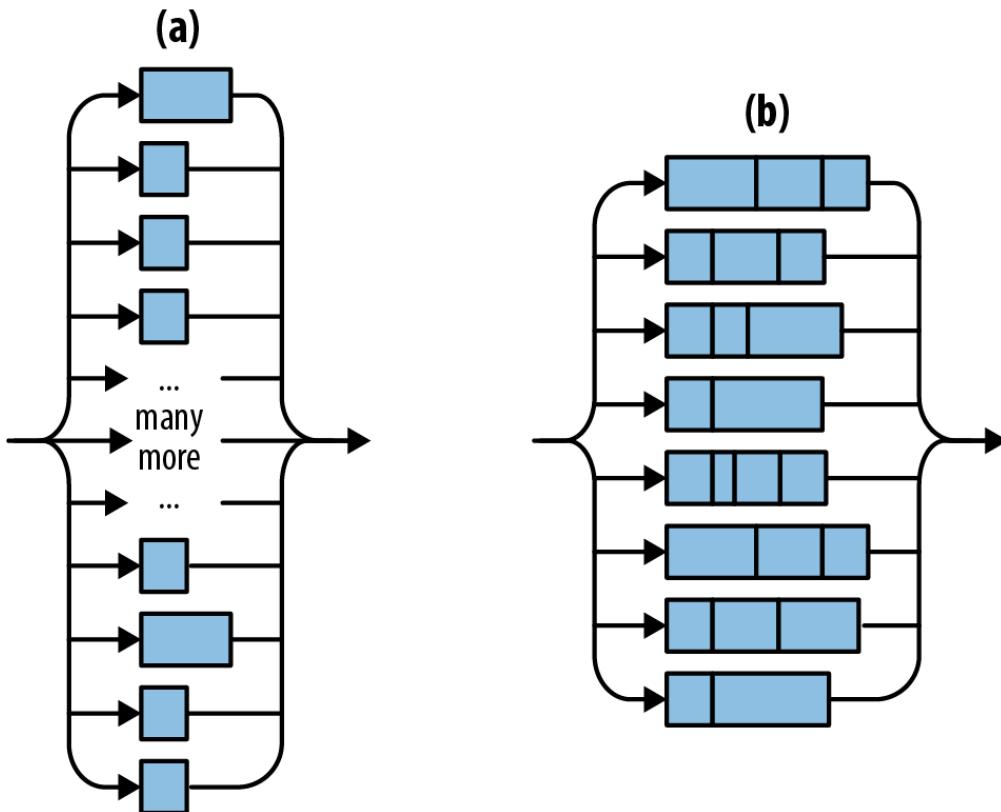


Image 19-3. Rayon en théorie et en pratique

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

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

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

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

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

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

- Nous utilisons `.reduce_with()` pour combiner les résultats. Ici, nous gardons la première erreur, le cas échéant, et supprimons le reste. Si nous voulions accumuler toutes les erreurs, ou les imprimer, nous pourrions le faire ici.

La `.reduce_with()` méthode est également pratique lorsque vous passez une `.map()` fermeture qui renvoie une valeur utile en cas de succès. Ensuite, vous pouvez passer `.reduce_with()` une fermeture qui sait combiner deux résultats réussis.

- `reduce_with` renvoie un `Option` qui est `None` seulement si `filenames` était vide. Nous utilisons la `Option` méthode `.unwrap_or()` de pour faire le résultat `Ok(())` dans ce cas.

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

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

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

Revisiter l'ensemble de Mandelbrot

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

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

izontales de taille égale, nous créions des charges de travail inégales, comme le montre la [figure 19-4](#).

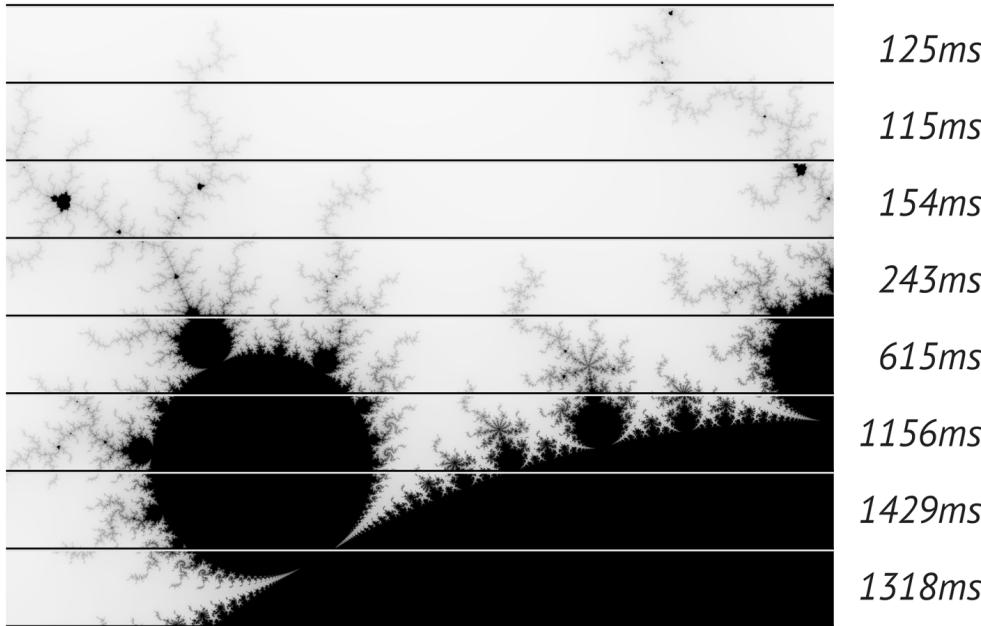


Image 19-4. Répartition inégale du travail dans le programme Mandelbrot

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

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

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

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

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

```

    } );
}

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

```

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

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

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

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


```

et ceci à `Cargo.toml` :

```
[dépendances]
rayonne = "1"
```

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

Canaux

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

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

voyer des octets, les canaux servent à envoyer des valeurs Rust.

`sender.send(item)` place une seule valeur dans le canal ;

`receiver.recv()` en supprime un. La propriété est transférée du thread d'envoi au thread de réception. Si le canal est vide,

`receiver.recv()` bloque jusqu'à ce qu'une valeur soit envoyée.

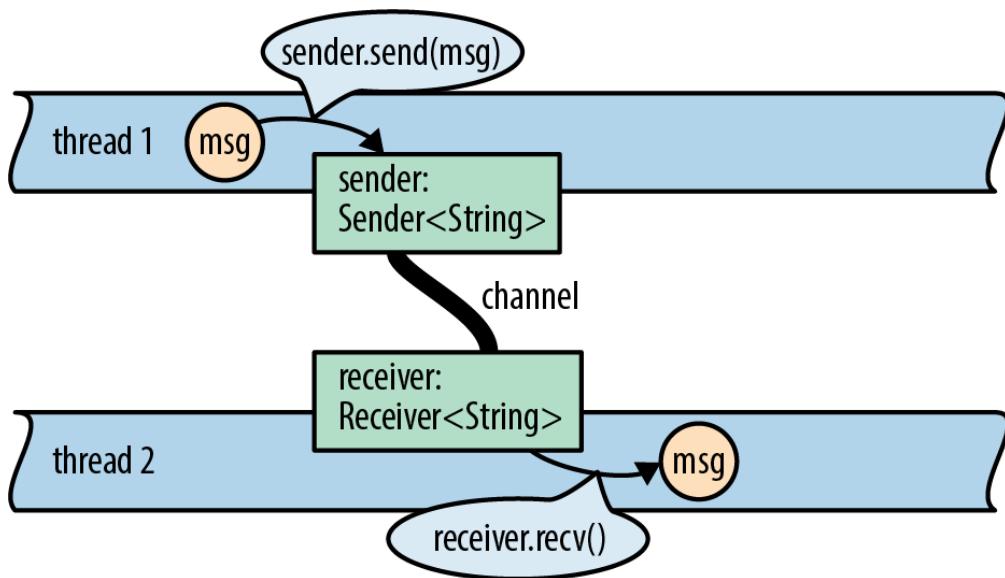


Image 19-5. Un canal pour `String`s : la propriété de la chaîne `msg` est transférée du thread 1 au thread 2.

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

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

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

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

sh (the unix shell)

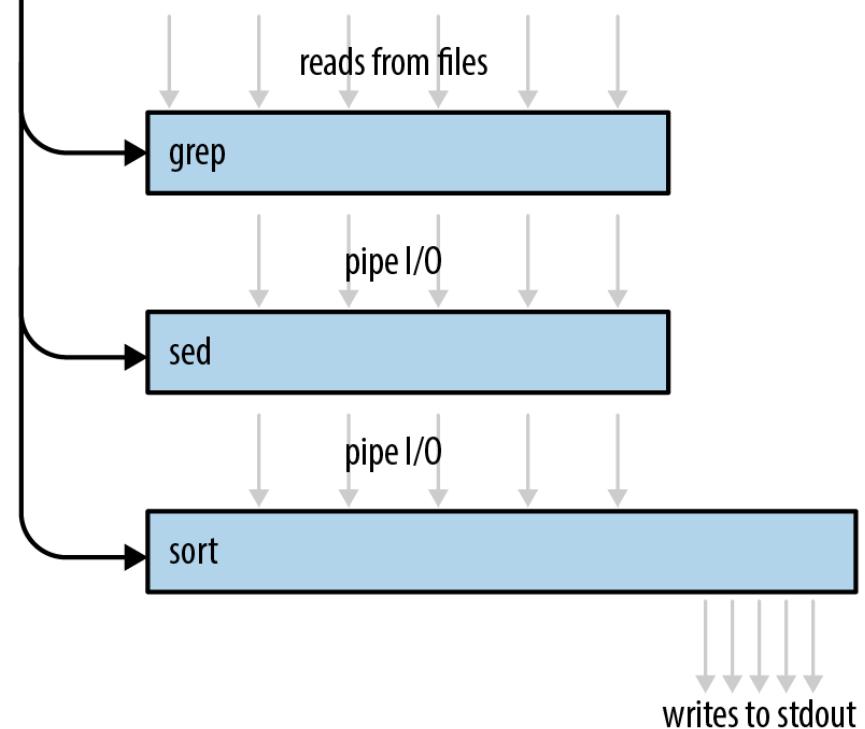


Illustration 19-6. Exécution d'un pipeline Unix

Envoi de valeurs

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

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

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

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

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

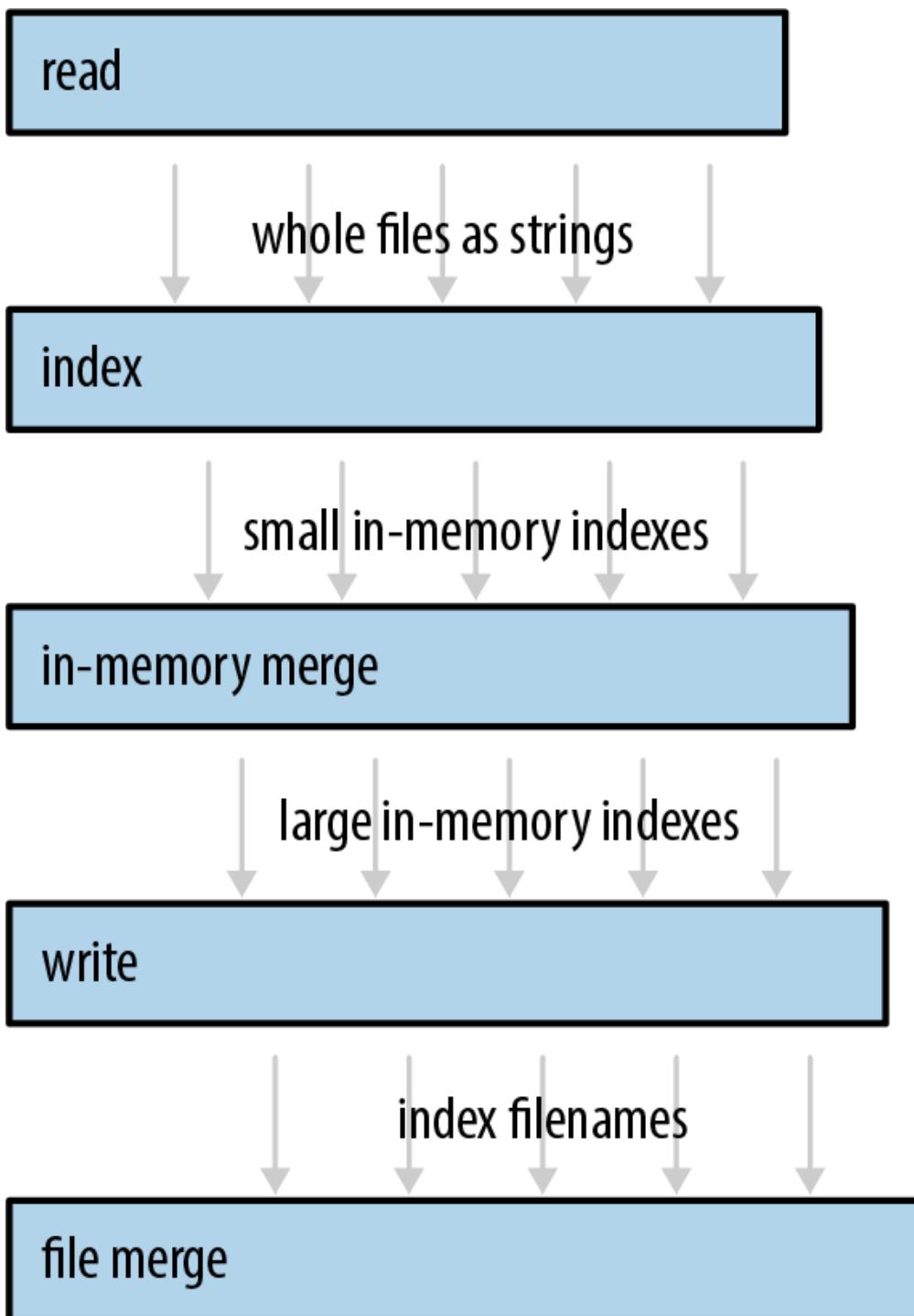


Image 19-7. Le pipeline de création d'index, où les flèches représentent les valeurs envoyées via un canal d'un thread à un autre (les E/S de disque ne sont pas affichées)

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

```

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

```

```

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

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

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

```

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

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

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

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

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

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

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

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

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

```

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

```

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

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

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

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

```

Ok(())
});

```

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

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

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

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

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

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

    (receiver, handle)
}
```

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

Recevoir des valeurs

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

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

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

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

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

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

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

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

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

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

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

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

Exécution du pipeline

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

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

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

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

Fonctionnalités et performances de la chaîne

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

naux de Rust fournissent.

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

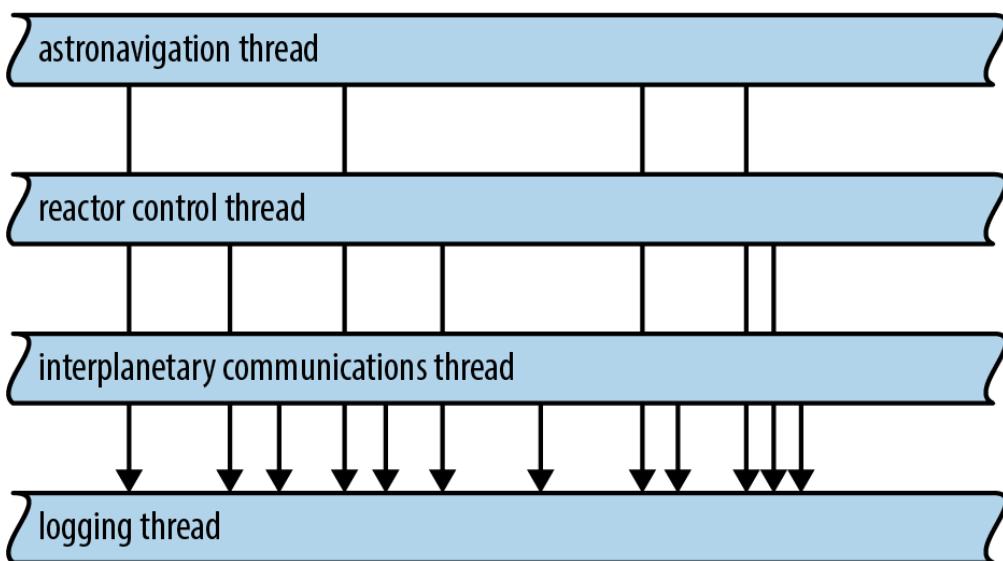


Image 19-8. Un canal unique recevant les requêtes de nombreux expéditeurs

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

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

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

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

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

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

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

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

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

Sécurité des threads : envoyer et synchroniser

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

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

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

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

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

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

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

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

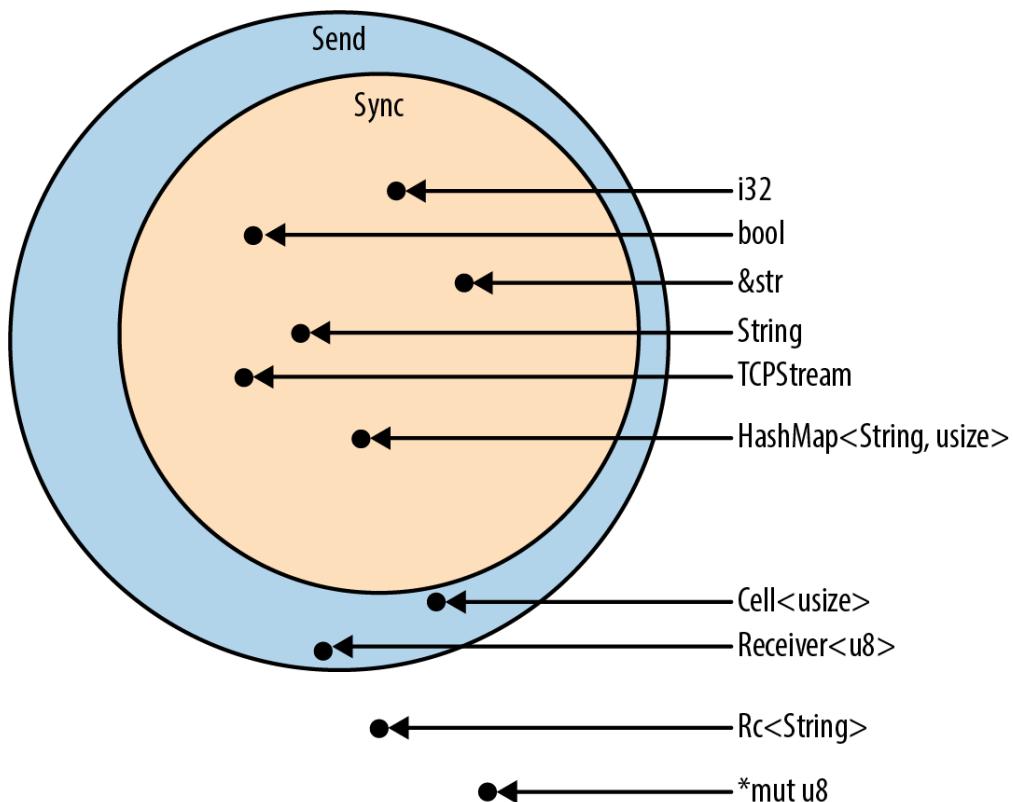


Illustration 19-9. Send et Sync types

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

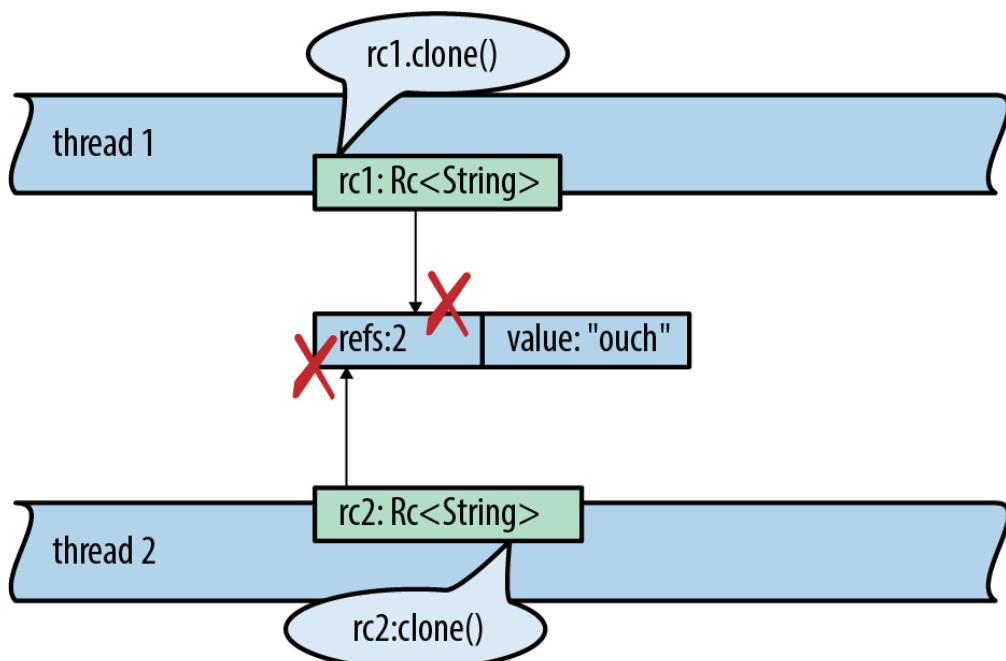


Figure 19-10. Pourquoi `Rc<String>` ni `Sync` ni `Send`

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

```

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

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

```

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

```

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

```

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

Canaliser presque n'importe quel itérateur vers un canal

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

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

```

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

```

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

```

use std::sync::mpsc;

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

```

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

```

use std::thread;

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

        // Move this iterator to a new worker thread and run it there.
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });
        // Return an iterator that pulls values from the channel.
        receiver.into_iter()
    }
}

```

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

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

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

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

Au-delà des pipelines

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

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

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

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

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

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

État mutable partagé

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

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

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

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

Qu'est-ce qu'un mutex ?

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

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

    waitingList.push_back(player);

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

    mutex.Release();
}
```

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

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

Les mutex sont utiles pour plusieurs raisons :

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

qui fonctionne même si d'autres threads modifient ses données pendant son exécution. Imaginez essayer de le déboguer. Ce serait comme si votre programme était hanté.

- Les mutex prennent en charge la programmation avec *des invariants*, règles sur les données protégées qui sont vraies par construction lorsque vous les configurez et maintenues par chaque section critique.

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

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

```
class FernEmpireApp {  
    ...  
  
private:  
    // List of players waiting to join a game. Protected by `mutex`.  
    vector<PlayerId> waitingList;  
  
    // Lock to acquire before reading or writing `waitingList`.  
    Mutex mutex;  
    ...  
};
```

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

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

Mutex<T>

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

```
type PlayerId = u32;
```

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

```
const GAME_SIZE:usize = 8;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

mut et mutex

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

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

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

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

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

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

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

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

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

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

Pourquoi les mutex ne sont pas toujours une bonne idée

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

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

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

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

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

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

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

Impasse

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

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

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

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

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

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

Mutex empoisonnés

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

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

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

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

Canaux multiconsommateurs utilisant des mutex

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

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

```

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

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

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

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

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

```

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

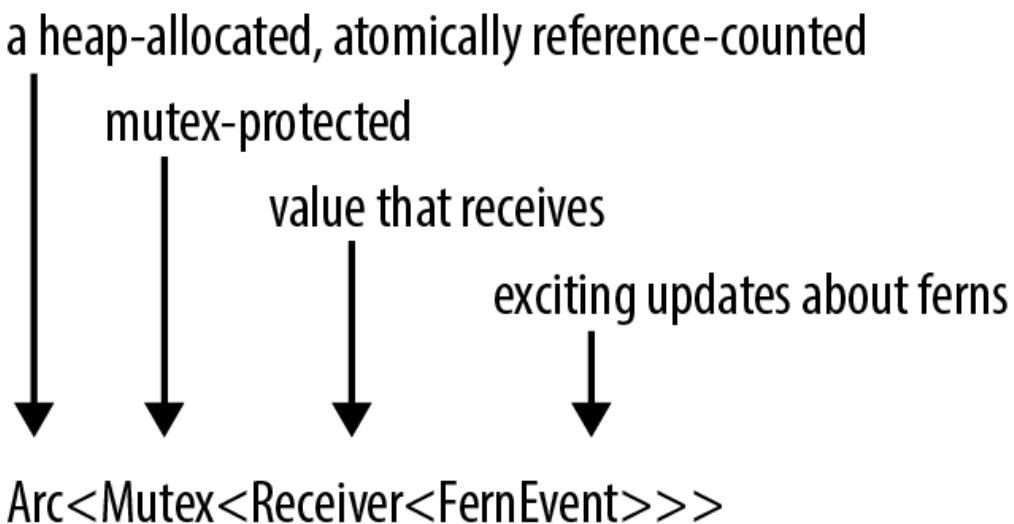


Image 19-11. Comment lire un type complexe

Verrouillages en lecture/écriture (RwLock<T>)

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

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

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

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

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

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

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

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

La méthode pour recharger la configuration utiliserait

RwLock::write():

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

Rust, bien sûr, est particulièrement bien adapté pour faire respecter les règles de sécurité sur RwLock les données. Le concept d'écrivain unique ou de lecteur multiple est au cœur du système d'emprunt de Rust.

`self.config.read()` renvoie une garde qui fournit `mut` un accès non (partagé) au `AppConfig`; `self.config.write()` renvoie un autre type de garde qui fournit `mut` un accès (exclusif).

Variables de condition (Condvar)

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

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

Parfois, il existe une API de blocage pratique pour la condition exacte que nous voulons attendre, comme `JoinHandle::join` pour l'exemple d'arrêt du serveur. Dans d'autres cas, il n'y a pas d'API de blocage intégrée.

Les programmes peuvent utiliser *des variables de condition* pour créer les leurs. Dans Rust, le `std::sync::Condvar` type implémente des variables de condition. A `Condvar` a des méthodes `.wait()` et `.notify_all()`; `.wait()` bloque jusqu'à ce qu'un autre thread appelle `.notify_all()`.

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

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

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

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

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

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

Atomique

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

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

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

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

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

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

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

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

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

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

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

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

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

Voici le code du travailleur :

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

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

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

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

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

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

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

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

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

Variables globales

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

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

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

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

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

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

```
use std::sync::atomic::AtomicUsize;  
  
static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

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

```
use std::sync::atomic::Ordering;  
  
PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

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

vient à résoudre deux problèmes.

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

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

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

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

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

```

    }

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

```

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

```

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

```

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

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

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

```

use lazy_static::lazy_static;

use std:: sync::Mutex;

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

```

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

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

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

À quoi ressemble le piratage de code concurrent dans Rust

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

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

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

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 20. Programmation asynchrone

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

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

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

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

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

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

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

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

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

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

```
use async_std::net, task;

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

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

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

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

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

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

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

De synchrone à asynchrone

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

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

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

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

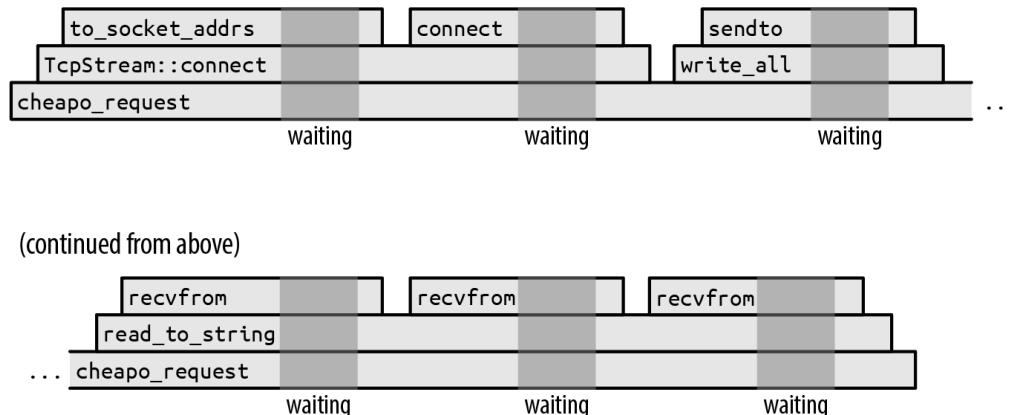
    let mut response = String::new();
    socket.read_to_string(&mut response)?;
}

Ok(response)
```

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

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

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



(continued from above)

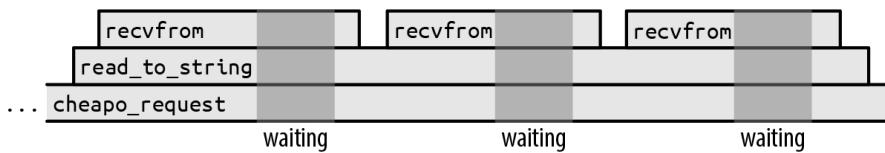


Illustration 20-1. Avancement d'une requête HTTP synchronie (des zones grises plus foncées attendent le système d'exploitation)

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

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

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

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

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

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

Contrats à terme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bonnes nouvelles! Ce n'est pas.

Fonctions asynchrones et expressions d'attente

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

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

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

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

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

    Ok(response)
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appel de fonctions asynchrones à partir de code synchrone : `block_on`

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

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

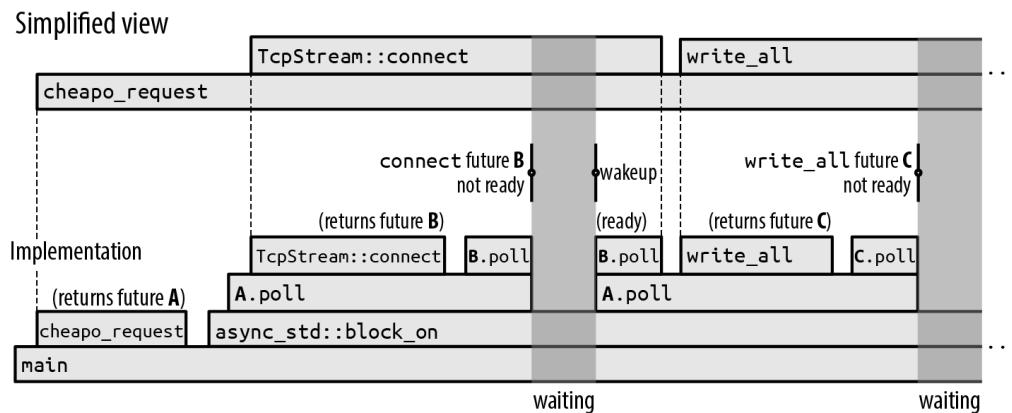
```
fn main() -> std::io:: Result<()> {
    use async_std::task;

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

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

[La figure 20-2](#) montre une exécution possible de `main`.

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



(Continued from above)

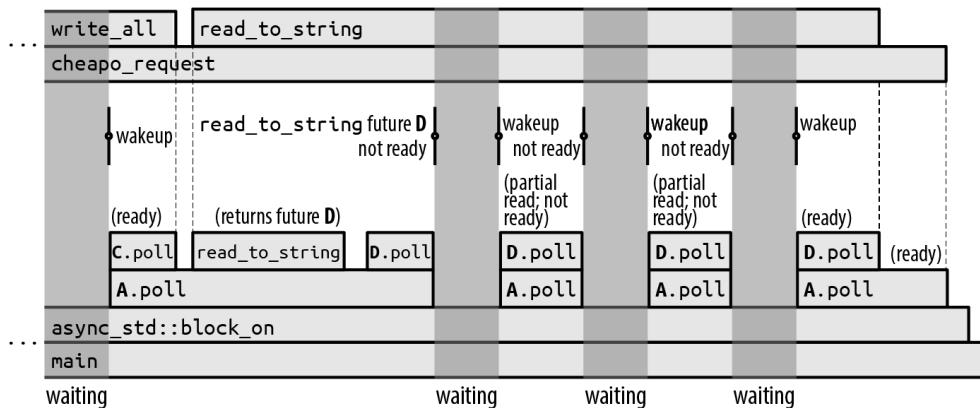


Illustration 20-2. Blocage sur une fonction asynchrone

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

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

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

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

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

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

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

Génération de tâches asynchrones

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

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

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

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

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

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

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

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

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

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

    results
}
```

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

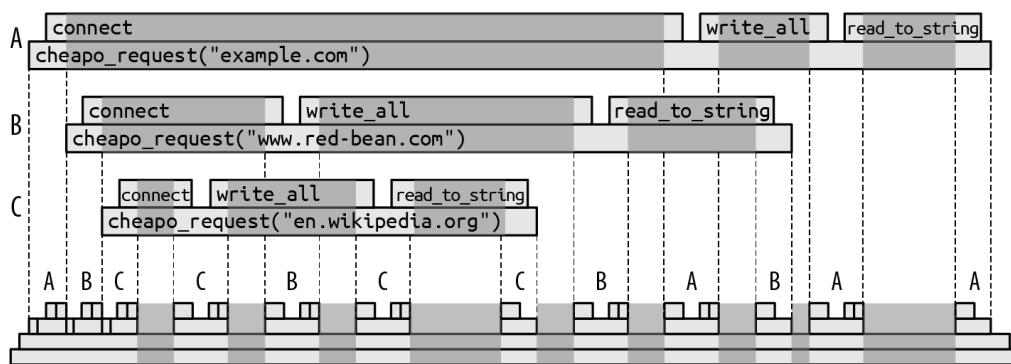
let requests = vec![
    ("example.com".to_string(), 80, "/".to_string()),
    ("www.red-bean.com".to_string(), 80, "/".to_string()),
    ("en.wikipedia.org".to_string(), 80, "/".to_string()),
];
let results = async_std::task::block_on(many_requests(requests));
for result in results {
    match result {
        Ok(response) => println!("{}: {}", response),
        Err(err) => eprintln!("error: {}", err),
    }
}

```

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

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

Asynchronous tasks



Synchronous calls

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

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

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

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

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

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

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

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

Blocs asynchrones

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

Ok(())
};

```

Cela échoue avec l'erreur suivante :

```

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

```

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

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

```

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

```

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

Création de fonctions asynchrones à partir de blocs asynchrones

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

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

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

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

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

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

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

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

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

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

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

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

... other work ...

let response = join_handle.await?;
```

Génération de tâches asynchrones sur un pool de threads

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

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

use async_std::task;

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

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

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

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

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

Mais votre futur outil envoie-t-il ?

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

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

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

```

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

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

    some_asynchronous_thing().await;

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

task::spawn(reluctant());

```

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

```

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

```

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

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

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

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

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

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

    return_value
}
```

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

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

```
// Not recommended!
type GenericError = Box;
type GenericResult<T> = Result<T, GenericError>;
```

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

```

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

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

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

```

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

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

```

type GenericError = Box;
type GenericResult<T> = Result<T, GenericError>;

```

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

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

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

Calculs de longue durée : `yield_now` et `spawn_blocking`

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

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

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

La première fois que l' `yield_now` avenir est interrogé, il revient `Poll::Pending`, mais indique qu'il vaut la peine d'être interrogé à nouveau bientôt. L'effet est que votre appel asynchrone abandonne le thread et que d'autres tâches ont une chance de s'exécuter, mais votre appel aura bientôt un autre tour. La seconde fois `yield_now` que le futur est interrogé, il renvoie `Poll::Ready(())` et votre fonction asynchrone peut reprendre son exécution.

Cette approche n'est cependant pas toujours réalisable. Si vous utilisez une caisse externe pour effectuer le calcul de longue durée ou pour appeler C ou C++, il peut ne pas être pratique de modifier ce code pour qu'il soit plus convivial pour l'asynchronisme. Ou il peut être difficile de s'assurer que chaque chemin à travers le calcul est sûr d'atteindre le `await` de temps en temps.

Pour des cas comme celui-ci, vous pouvez utiliser `async_std::task::spawn_blocking`. Cette fonction prend une fermeture, la lance sur son propre thread et renvoie un futur de sa valeur de retour. Le code asynchrone peut attendre ce futur, cédant son fil à d'autres tâches jusqu'à ce que le calcul soit prêt. En mettant le travail acharné sur un thread séparé, vous pouvez laisser le système d'exploitation s'occuper de lui faire partager le processeur de manière agréable.

Par exemple, supposons que nous devions vérifier les mots de passe fournis par les utilisateurs par rapport aux versions hachées que nous avons stockées dans notre base de données d'authentification. Pour des raisons de sécurité, la vérification d'un mot de passe doit nécessiter beaucoup de calculs, de sorte que même si les attaquants obtiennent une copie de notre base de données, ils ne peuvent pas simplement essayer des milliards de mots de passe possibles pour voir s'ils correspondent. La `argonautica` offre une fonction de hachage conçue spécifiquement pour stocker les mots de passe : un `argonautica` hachage correctement généré prend une fraction de seconde significative à vérifier. Nous pouvons utiliser `argonautica` (version 0.2) dans notre application asynchrone comme ceci :

```
async fn verify_password(password: &str, hash: &str, key: &str)
    -> Result<bool, argonautica::Error>
{
    // Make copies of the arguments, so the closure can be 'static.
    let password = password.to_string();
    let hash = hash.to_string();
    let key = key.to_string();

    async_std::task::spawn_blocking(move || {
        argonautica::Verifier::default()
            .with_hash(hash)
            .with_password(password)
            .with_secret_key(key)
            .verify()
    }).await
}
```

Cela renvoie `Ok(true)` si `password` correspond à `hash`, étant donné `key`, une clé pour la base de données dans son ensemble. En effectuant la vérification dans la fermeture transmise à `spawn_blocking`, nous poussons le calcul coûteux sur son propre thread, en veillant à ce qu'il n'affecte pas notre réactivité aux demandes des autres utilisateurs.

Comparaison de conceptions asynchrones

Dans de nombreux langages de programmation asynchrone ressemble à celle adoptée par d'autres langages. Par exemple, JavaScript, C# et Rust ont tous des fonctions asynchrones avec des `await` expressions. Et tous ces langages ont des valeurs qui représentent des calculs incomplets : Rust les appelle « futurs », JavaScript les appelle « promesses » et C# appelle les « tâches », mais elles représentent toutes une valeur que vous devrez peut-être attendre.

L'utilisation des sondages par Rust, cependant, est inhabituel. En JavaScript et C#, une fonction asynchrone commence à s'exécuter dès qu'elle est appelée, et il existe une boucle d'événements globale intégrée à la bibliothèque système qui reprend les appels de fonction asynchrone suspendus lorsque les valeurs qu'ils attendaient deviennent disponibles. Dans Rust, cependant, un appel asynchrone ne fait rien jusqu'à ce que vous le passiez à une fonction comme `block_on`, `spawn` ou `spawn_local` qui l'interrogera et conduira le travail à son terme. Ces fonctions, appelées *exécuteurs*, jouent le rôle que d'autres langages couvrent avec une boucle d'événements globale.

Parce que Rust vous oblige, le programmeur, à choisir un exécuteur pour interroger votre avenir, Rust n'a pas besoin d'une boucle d'événements globale intégré au système. La `async-std` caisse offre les fonctions d'exécuteur que nous avons utilisées dans ce chapitre jusqu'ici, mais la `tokio` caisse, que nous utiliserons plus tard dans ce chapitre, définit son propre ensemble de fonctions d'exécution similaires. Et vers la fin de ce chapitre, nous implémenterons notre propre exécuteur. Vous pouvez utiliser les trois dans le même programme.

Un vrai client HTTP asynchrone

Nous serions négligents si nous ne montrions pas un exemple d'utilisation d'une caisse de client HTTP asynchrone appropriée, car c'est si facile, et il y a plusieurs bonnes caisses parmi lesquelles choisir, y compris `reqwest` et `surf`.

Voici une réécriture de `many_requests`, encore plus simple que celle basée sur `cheapo_request`, qui `surf` permet d'exécuter simultanément une série de requêtes. Vous aurez besoin de ces dépendances dans votre fichier `Cargo.toml` :

```
[dépendances]
async-std = "1.7"
surf = "1.0"
```

Ensuite, nous pouvons définir `many_requests` comme suit :

```
pub async fn many_requests(urls: &[String])
    -> Vec<Result<String, surf::Exception>>
{
    let client = surf::Client::new();

    let mut handles = vec![];
```

```

        for url in urls {
            let request = client.get(&url).recv_string();
            handles.push(async_std:: task::spawn(request));
        }

        let mut results = vec![];
        for handle in handles {
            results.push(handle.await);
        }

        results
    }

fn main() {
    let requests = &[ "http://example.com".to_string(),
                      "https://www.red-bean.com".to_string(),
                      "https://en.wikipedia.org/wiki/Main_Page".to_string()]

    let results = async_std:: task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("*** {}\n", response),
            Err(err) => eprintln!("error: {}\n", err),
        }
    }
}

```

Utiliser un seul `surf::Client` pour faire toutes nos requêtes nous permet de réutiliser les connexions HTTP si plusieurs d'entre elles sont dirigées vers le même serveur. Et aucun bloc `async` n'est nécessaire : puisque `recv_string` c'est une méthode asynchrone qui renvoie un `Send + 'static` futur, nous pouvons passer son futur directement à `spawn`.

Un client et un serveur asynchrones

C'est l'heure de prendre les idées clés dont nous avons discuté jusqu'à présent et de les assembler dans un programme de travail. Dans une large mesure, les applications asynchrones ressemblent aux applications multi-thread ordinaires, mais il existe de nouvelles opportunités de code compact et expressif que vous pouvez rechercher.

L'exemple de cette section est un serveur et un client de chat. Découvrez le [code complet](#). Les vrais systèmes de chat sont compliqués, avec des préoccupations allant de la sécurité et de la reconnexion à la confidentialité et à la modération, mais nous avons réduit le nôtre à un ensemble

austère de fonctionnalités afin de nous concentrer sur quelques points d'intérêt.

En particulier, nous voulons bien gérer *la contre-pression*. Nous entendons par là que si un client a une connexion Internet lente ou abandonne complètement sa connexion, cela ne doit jamais affecter la capacité des autres clients à échanger des messages à leur propre rythme. Et puisqu'un client lent ne devrait pas obliger le serveur à dépenser une mémoire illimitée pour conserver son arriéré de messages sans cesse croissant, notre serveur devrait abandonner les messages pour les clients qui ne peuvent pas suivre, mais les avertir que leur flux est incomplet. (Un vrai serveur de chat enregistrerait les messages sur le disque et permettrait aux clients de récupérer ceux qu'ils ont manqués, mais nous avons laissé cela de côté.)

On démarre le projet avec la commande `cargo new --lib async-chat` et on met le texte suivant dans `async-chat/Cargo.toml` :

```
[forfait]
nom = "chat asynchrone"
version = "0.1.0"
auteurs = ["Vous <vous@example.com>"]
édition = "2021"

[dépendances]
async-std = { version = "1.7", fonctionnalités = ["unstable"] }
tokio = { version = "1.0", fonctionnalités = ["sync"] }
serde = { version = "1.0", fonctionnalités = ["dériver", "rc"] }
serde_json = "1.0"
```

Nous dépendons de quatre caisses :

- La `async-std` caisse est la collection de primitives et d'utilitaires d'E/S asynchrones que nous avons utilisés tout au long de ce chapitre.
 - La `tokio` caisse est une autre collection de primitives asynchrones comme `async-std`, l'une des plus anciennes et des plus matures. Il est largement utilisé et sa conception et sa mise en œuvre respectent des normes élevées, mais son utilisation nécessite un peu plus de soin que `async-std`.
- `tokio` est une grande caisse, mais nous n'en avons besoin que d'un seul composant, de sorte que le `features = ["sync"]` champ de la ligne de dépendance `Cargo.toml` `tokio` se limite aux pièces dont nous avons besoin, ce qui en fait une dépendance légère.
- Lorsque l'écosystème de la bibliothèque asynchrone était moins mature, les gens évitaient d'utiliser les deux `tokio` et `async-std` dans le

même programme, mais les deux projets ont coopéré pour s'assurer que cela fonctionne, tant que les règles documentées de chaque caisse sont suivies.

- Les caisses `serde` et `serde_json` nous avons vu auparavant, au [chapitre 18](#). Ceux-ci nous donnent des outils pratiques et efficaces pour générer et analyser JSON, que notre protocole de chat utilise pour représenter les données sur le réseau. Nous voulons utiliser certaines fonctionnalités facultatives de `serde`, nous les sélectionnons donc lorsque nous donnons la dépendance.

La structure entière de notre application de chat, client et serveur, ressemble à ceci :

```
async-chat
├── Cargo.toml
└── src
    ├── lib.rs
    └── utils.rs
    └── bin
        ├── client.rs
        └── server
            ├── main.rs
            ├── connection.rs
            ├── group.rs
            └── group_table.rs
```

Cette disposition de paquet utilise une fonctionnalité Cargo dont nous avons parlé dans [« Le répertoire src/bin »](#) : en plus de la caisse principale de la bibliothèque, `src/lib.rs`, avec son sous-module `src/utils.rs`, il comprend également deux exécutables :

- `src/bin/client.rs` est un fichier exécutable unique pour le client de chat.
- `src/bin/server` est l'exécutable du serveur, réparti sur quatre fichiers : `main.rs` contient la `main` fonction, et il y a trois sous-modules, `connection.rs`, `group.rs` et `group_table.rs`.

Nous présenterons le contenu de chaque fichier source au cours du chapitre, mais une fois qu'ils sont tous en place, si vous tapez `cargo build` dans cet arbre, cela compile la bibliothèque crate puis construit les deux exécutables. Cargo inclut automatiquement la caisse de la bibliothèque en tant que dépendance, ce qui en fait un endroit pratique pour mettre les définitions partagées par le client et le serveur. De même, `cargo check` vérifie l'intégralité de l'arborescence des sources. Pour exécuter l'un ou l'autre des exécutables, vous pouvez utiliser des commandes comme celles-ci :

```
$cargo run --release --bin serveur -- localhost:8088
$cargo run --release --bin client -- localhost:8088
```

L' `--bin` option indique quel exécutable exécuter et tous les arguments suivant l'`--` option sont transmis à l'exécutable lui-même. Notre client et notre serveur veulent juste connaître l'adresse et le port TCP du serveur.

Types d'erreur et de résultat

`utils` Le module de la caisse de la bibliothèque définit le résultat et l'erreur types que nous utiliserons tout au long de l'application. Depuis `src/utils.rs` :

```
use std::error::Error;

pub type ChatError = Box;
pub type ChatResult<T> = Result<T, ChatError>;
```

Il s'agit des types d'erreurs à usage général que nous avons suggérés dans [« Utilisation de plusieurs types d'erreurs »](#). Les `async_std`, `serde_json` et `tokio` crates définissent chacun leurs propres types d'erreur, mais l'`? opérateur` peut les convertir automatiquement en un `ChatError`, en utilisant l'implémentation de la bibliothèque standard du `From` trait qui peut convertir n'importe quel type d'erreur approprié en `Box<dyn Error + Send + Sync + 'static>`. Les limites `Send` et `Sync` garantissent que si une tâche générée sur un autre thread échoue, elle peut signaler l'erreur en toute sécurité au thread principal.

Dans une application réelle, pensez à utiliser la `anyhow` caisse, qui fournit `Error` et `Result` types similaires à ceux-ci. La `anyhow` caisse est facile à utiliser et offre des fonctionnalités intéressantes au-delà de ce que notre `ChatError` et `ChatResult` peuvent offrir.

Le protocole

La caisse de la bibliothèque capture l'intégralité 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' `FromClient` énumération représente les paquets qu'un client peut envoyer au serveur : il peut demander à rejoindre un groupe et envoyer des messages à n'importe quel groupe qu'il a rejoint. `FromServer` représente ce que le serveur peut renvoyer : messages postés à un groupe et messages d'erreur. L'utilisation d'une référence comptée `Arc<String>` au lieu d'une plaine `String` aide le serveur à éviter de faire des copies de chaînes lorsqu'il gère des groupes et distribue des messages.

Les `#[derive]` attributs indiquent à la `serde` caisse de générer des implémentations de ses traits `Serialize` et pour et . Cela nous permet d'appeler pour les convertir en valeurs JSON, de les envoyer sur le réseau et enfin d'appeler pour les reconvertis dans leurs formes

```
Rust. Deserialize FromClient FromServer serde_json::to_string  
g serde_json::from_str
```

Le `test_fromclient_json` test unitaire illustre comment cela est utilisé. Compte tenu de l' `Serialize` implémentation dérivée de `serde`, nous pouvons appeler `serde_json::to_string` pour transformer la `FromClient` valeur donnée en ce JSON :

```
{"Post": {"group_name": "Dogs", "message": "Samoyeds rock!"}}
```

Ensuite, l' `Deserialize` implémentation dérivée analyse cela en une `FromClient` valeur équivalente. Notez que les `Arc` pointeurs dans `FromClient` n'ont aucun effet sur le formulaire sérialisé : les chaînes comptées par référence apparaissent directement en tant que valeurs de membre d'objet JSON.

Prendre l'entrée de l'utilisateur : flux asynchrones

Notre conversationLa première responsabilité du client est de lire les commandes de l'utilisateur et d'envoyer les paquets correspondants au serveur. La gestion d'une interface utilisateur appropriée dépasse le cadre de ce chapitre, nous allons donc faire la chose la plus simple qui fonctionne : lire des lignes directement à partir de l'entrée standard. Le code suivant va dans `src/bin/client.rs` :

```
use async_std:: prelude::*;
use async_chat:: utils:: {self, ChatResult};
use async_std:: io;
use async_std::net;

async fn send_commands(mut to_server: net:: TcpStream) -> ChatResult<()> {
    println!("Commands:\n\
              join GROUP\n\
              post GROUP MESSAGE...\n\
              Type Control-D (on Unix) or Control-Z (on Windows) \
              to close the connection.");
}

let mut command_lines = io:: BufReader:: new(io::stdin()).lines();
while let Some(command_result) = command_lines.next().await {
    let command = command_result?;
    // See the GitHub repo for the definition of `parse_command`.
    let request = match parse_command(&command) {
        Some(request) => request,
        None => continue,
    };
}
```

```

        utils::send_as_json(&mut to_server, &request).await?;
        to_server.flush().await?;
    }

    Ok(())
}

```

Cela appelle `async_std::io::stdin` pour obtenir un handle asynchrone sur l'entrée standard du client, l'enveloppe dans un `async_std::io::BufReader` pour le mettre en mémoire tampon, puis appelle `lines` pour traiter l'entrée de l'utilisateur ligne par ligne. Il essaie d'analyser chaque ligne comme une commande correspondant à une `FromClient` valeur et, s'il réussit, envoie cette valeur au serveur. Si l'utilisateur saisit une commande non reconnue, `parse_command` imprime un message d'erreur et renvoie `None`, il `send_commands` peut donc à nouveau faire le tour de la boucle. Si l'utilisateur tape une indication de fin de fichier, le `lines` flux renvoie `None`, et `send_commands` revient. Cela ressemble beaucoup au code que vous écririez dans un programme synchrone ordinaire, sauf qu'il utilise `async_std` les versions des fonctionnalités de la bibliothèque.

La méthode `BufReader` de l'asynchrone `lines` est intéressant. Elle ne peut pas renvoyer d'itérateur, comme le fait la bibliothèque standard : la `Iterator::next` méthode est une fonction synchrone ordinaire, donc l'appel `command_lines.next()` bloquerait le thread jusqu'à ce que la ligne suivante soit prête. Au lieu de cela, `lines` renvoie un *flux* de `Result<String>` valeurs. Un flux est l'analogie asynchrone d'un itérateur : il produit une séquence de valeurs à la demande, de manière asynchrone. Voici la définition du `Stream` trait, du `async_std::stream` module :

```

trait Stream {
    type Item;

    // For now, read `Pin<&mut Self>` as `&mut Self`.
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}

```

Vous pouvez considérer cela comme un hybride des traits `Iterator` et `Future`. Comme un itérateur, a `Stream` a un type associé `Item` et utilise `Option` pour indiquer quand la séquence est terminée. Mais comme un futur, un flux doit être interrogé : pour obtenir l'élément suivant (ou savoir que le flux est terminé), vous devez appeler `poll_next` jusqu'à ce

qu'il renvoie `Poll::Ready`. L'implémentation d'un flux `poll_next` doit toujours revenir rapidement, sans blocage. Et si un flux revient `Poll::Pending`, il doit avertir l'appelant quand cela vaut la peine d'interroger à nouveau via le `Context`.

La `poll_next` méthode est difficile à utiliser directement, mais vous n'aurez généralement pas besoin de le faire. Comme les itérateurs, les flux ont une large collection de méthodes utilitaires telles que `filter` et `map`. Parmi celles-ci se trouve une `next` méthode, qui renvoie un futur du flux suivant `Option<Self::Item>`. Plutôt que d'interroger explicitement le flux, vous pouvez appeler `next` et attendre le futur qu'il renvoie à la place.

L'assemblage de ces éléments `send_commands` consomme le flux de lignes d'entrée en bouclant les valeurs produites par un flux utilisant `next with while let`:

```
while let Some(item) = stream.next().await {  
    ... use item ...  
}
```

(Les futures versions de Rust introduiront probablement une variante asynchrone de la `for` syntaxe de boucle pour consommer des flux, tout comme une boucle ordinaire `for` consomme des `Iterator` valeurs.)

Interroger un flux après sa fin, c'est-à-dire après qu'il est revenu `Poll::Ready(None)` pour indiquer la fin du flux, revient à appeler `next` un itérateur après son retour `None` ou interroger un futur après son retour `Poll::Ready`: le `Stream` trait ne précise pas ce que le flux doit faire, et certains flux peuvent mal se comporter. Comme les futurs et les itérateurs, les flux ont une `fuse` méthode pour s'assurer que ces appels se comportent de manière prévisible, lorsque cela est nécessaire ; voir la documentation pour plus de détails.

Lorsque vous travaillez avec des flux, il est important de ne pas oublier d'utiliser le `async_std` prélude :

```
use async_std:: prelude::*;


```

C'est parce que les méthodes d'utilité pour le `Stream` trait, comme `next`, `map`, `filter`, etc., ne sont en fait pas définies sur `Stream` elles-mêmes. Au lieu de cela, ce sont des méthodes par défaut d'un trait distinct, `StreamExt`, qui est automatiquement implémenté pour tous les `Stream`s :

```

pub trait StreamExt:Stream {
    ... define utility methods as default methods ...
}

impl<T:Stream> StreamExt for T { }

```

Ceci est un exemple du *trait d'extension* modèle que nous avons décrit dans "[Traits et types d'autres personnes](#)". Le `async_std::prelude` module apporte les `StreamExt` méthodes dans la portée, donc l'utilisation du prélude garantit que ses méthodes sont visibles dans votre code.

Envoi de paquets

Pour transmettre des paquets sur une socket réseau, notre client et notre serveur utilisent la `send_as_json` fonction `utils` du module de notre caisse de bibliothèque :

```

use async_std:: prelude::*;
use serde:: Serialize;
use std:: marker::Unpin;

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) -> ChatResult
where
    S: async_std:: io:: Write + Unpin,
    P: Serialize,
{
    let mut json = serde_json::to_string(&packet)?;
    json.push('\n');
    outbound.write_all(json.as_bytes()).await?;
    Ok(())
}

```

Cette fonction construit la représentation JSON de `packet` en tant que `String`, ajoute une nouvelle ligne à la fin, puis écrit le tout dans `outbound`.

D'après sa `where` clause, vous pouvez voir que `send_as_json` c'est assez flexible. Le type de paquet à envoyer, `P`, peut être tout ce qui implémente `serde::Serialize`. Le flux de sortie `S` peut être tout ce qui implémente `async_std::io::Write`, la version asynchrone du `std::io::Write` trait pour les flux de sortie. Cela nous suffit pour envoyer `FromClient` et des `FromServer` valeurs sur un fichier `TcpStream`. Garder la définition de `send_as_json` générique garantit qu'il ne dépend pas des détails des types de flux ou de paquets de ma-

nière surprenante : `send_as_json` ne peut utiliser que des méthodes à partir de ces traits.

La `Unpin` contrainte sur `S` est nécessaire pour utiliser la `write_all` méthode. Nous aborderons l'épinglage et le désépinglage plus loin dans ce chapitre, mais pour le moment, il devrait suffire d'ajouter des `Unpin` contraintes aux variables de type là où c'est nécessaire ; le compilateur Rust signalera ces cas si vous oubliez.

Plutôt que de sérialiser le paquet directement dans le `outbound` flux, le `send_as_json` sérialise dans un fichier temporaire `String`, puis l'écrit dans `outbound`. Le `serde_json` crate fournit des fonctions pour sérialiser les valeurs directement dans les flux de sortie, mais ces fonctions ne prennent en charge que les flux synchrones. L'écriture dans des flux asynchrones nécessiterait des changements fondamentaux à la fois `serde_json` et dans le `serde` noyau indépendant du format de la caisse, car les traits autour desquels ils sont conçus ont des méthodes synchrones.

Comme pour les flux, de nombreuses méthodes des `async_std` traits d'E/S sont en fait définies sur des traits d'extension, il est donc important de s'en souvenir `use async_std::prelude::*` chaque fois que vous les utilisez.

Réception de paquets : davantage de flux asynchrones

Pour recevoir des paquets, notre serveur et notre client utiliseront cette fonction du `utils` module pour recevoir `FromClient` et les `FromServer` valeurs d'un socket TCP tamponné asynchrone, un `async_std::io::BufReader<TcpStream>` :

```
use serde:: de::DeserializeOwned;

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item = ChatResult<P>>
    where S: async_std:: io:: BufRead + Unpin,
          P: DeserializeOwned,
{
    inbound.lines()
        .map(|line_result| -> ChatResult<P> {
            let line = line_result?;
            let parsed = serde_json:: from_str::<P>(&line)?;
            Ok(parsed)
        })
}
```

Comme `send_as_json`, cette fonction est générique dans le flux d'entrée et les types de paquets :

- Le type de flux `s` doit implémenter `async_std::io::BufRead`, l'analogue asynchrone de `std::io::BufRead`, représentant un flux d'octets d'entrée mis en mémoire tampon.
- Le type de paquet `P` doit implémenter `DeserializeOwned`, une variante plus stricte du `serde` trait `Deserialize` de. Pour plus d'efficacité, `Deserialize` peut produire `&str` et des `&[u8]` valeurs qui empruntent leur contenu directement à la mémoire tampon à partir de laquelle elles ont été déserialisées, pour éviter de copier des données. Dans notre cas, cependant, cela ne sert à rien : nous devons renvoyer les valeurs déserialisées à notre appelant, afin qu'elles puissent survivre aux tampons à partir desquels nous les avons analysées. Un type qui implémente `DeserializeOwned` est toujours indépendant du tampon à partir duquel il a été déserialisé.

L'appel `inbound.lines()` donne nous un `Stream` de `std::io::Result<String>` valeurs. Nous utilisons ensuite l'adaptateur du flux `map` pour appliquer une fermeture à chaque élément, en gérant les erreurs et en analysant chaque ligne sous la forme JSON d'une valeur de type `P`. Cela nous donne un flux de `ChatResult<P>` valeurs, que nous renvoyons directement. Le type de retour de la fonction est :

```
impl Stream<Item = ChatResult<P>>
```

Cela indique que nous renvoyons *un* type qui produit une séquence de `ChatResult<P>` valeurs de manière asynchrone, mais notre appelant ne peut pas dire exactement de quel type il s'agit. Étant donné que la fermeture à laquelle nous passons `map` a de toute façon un type anonyme, c'est le type le plus spécifique qui `receive_as_json` pourrait éventuellement être renvoyé.

Notez que ce `receive_as_json` n'est pas, en soi, une fonction asynchrone. C'est une fonction ordinaire qui renvoie une valeur asynchrone, un flux. Comprendre les mécanismes du support asynchrone de Rust plus en profondeur que "juste ajouter `async` et `.await` partout" ouvre le potentiel pour des définitions claires, flexibles et efficaces comme celle-ci qui tirent pleinement parti du langage.

Pour voir comment `receive_as_json` est utilisé, voici la `handle_replies` fonction de notre client de chat de `src/bin/client.rs`, qui reçoit un flux de `FromServer` valeurs du réseau et les imprime pour que l'utilisateur puisse les voir :

```

use async_chat::FromServer;

async fn handle_replies(from_server: net:: TcpStream) -> ChatResult<()> {
    let buffered = io:: BufReader:: new(from_server);
    let mut reply_stream = utils::receive_as_json(buffered);

    while let Some(reply) = reply_stream.next().await {
        match reply? {
            FromServer:: Message { group_name, message } => {
                println!("message posted to {}: {}", group_name, message);
            }
            FromServer:: Error(message) => {
                println!("error from server: {}", message);
            }
        }
    }

    Ok(())
}

```

Cette fonction prend une socket recevant des données du serveur, l' `BufReader` entoure d'un (notez bien, la `async_std` version), puis la transmet à pour obtenir un flux de valeurs `receive_as_json` entrantes . `FromServer` Ensuite, il utilise une `while let` boucle pour gérer les réponses entrantes, en vérifiant les résultats d'erreur et en imprimant chaque réponse du serveur pour que l'utilisateur puisse la voir..

La fonction principale du client

Puisque nous avons présenté à la fois `send_commands` et `handle_replies`, nous pouvons montrer la fonction principale du client de chat, depuis *src/bin/client.rs* :

```

use async_std::task;

fn main() -> ChatResult<()> {
    let address = std:: env::args().nth(1)
        .expect("Usage: client ADDRESS:PORT");

    task:: block_on(async {
        let socket = net:: TcpStream:: connect(address).await?;
        socket.set_nodelay(true)?;

        let to_server = send_commands(socket.clone());
        let from_server = handle_replies(socket);

        from_server.race(to_server).await?;
    })
}

```

```
    ok(())

})
}
```

Après avoir obtenu l'adresse du serveur à partir de la ligne de commande, `main` a une série de fonctions asynchrones qu'il aimerait appeler, il encapsule donc le reste de la fonction dans un bloc asynchrone et passe le futur du bloc `async_std::task::block_on` à s'exécuter.

Une fois la connexion établie, nous voulons que les fonctions `send_commands` et `handle_replies` s'exécutent en tandem, afin que nous puissions voir les messages des autres arriver pendant que nous tapons. Si nous entrons dans l'indicateur de fin de fichier ou si la connexion au serveur tombe, le programme doit se fermer.

Compte tenu de ce que nous avons fait ailleurs dans le chapitre, vous pourriez vous attendre à un code comme celui-ci :

```
let to_server = task:: spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));

to_server.await?;
from_server.await?;
```

Mais puisque nous attendons les deux poignées de jointure, cela nous donne un programme qui se termine une fois *les deux* tâches terminées. Nous voulons sortir dès que l'un *ou* l'autre a terminé. La `race` méthode sur les contrats à terme accomplit cela. L'appel

`from_server.race(to_server)` renvoie un nouveau futur qui interroge à la fois `from_server` et `to_server` et revient `Poll::Ready(v)` dès que l'un d'eux est prêt. Les deux contrats à terme doivent avoir le même type de sortie : la valeur finale est celle du futur qui s'est terminé en premier. Le futur inachevé est abandonné.

La `race` méthode, ainsi que de nombreux autres utilitaires pratiques, est défini sur le `async_std::prelude::FutureExt` trait, ce qui `async_std::prelude` nous rend visible.

À ce stade, la seule partie du code du client que nous n'avons pas montrée est la `parse_command` fonction. C'est un code de traitement de texte assez simple, nous ne montrerons donc pas sa définition ici. Voir le code complet dans le référentiel Git pour plus de détails.

La fonction principale du serveur

Voici l'intégralité du contenu du fichier principal du serveur, *src/bin/server/main.rs* :

```
use async_std:: prelude::*;

use async_chat:: utils:: ChatResult;
use std:: sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;

fn main() -> ChatResult<()> {
    let address = std:: env::args().nth(1).expect("Usage: server ADDRESS");

    let chat_group_table = Arc:: new(group_table:: GroupTable::new());

    async_std:: task:: block_on(async {
        // This code was shown in the chapter introduction.
        use async_std::{net, task};

        let listener = net:: TcpListener::bind(address).await?;

        let mut new_connections = listener.incoming();
        while let Some(socket_result) = new_connections.next().await {
            let socket = socket_result?;
            let groups = chat_group_table.clone();
            task::spawn(async {
                log_error(serve(socket, groups).await);
            });
        }
    })
}

fn log_error(result:ChatResult<()>) {
    if let Err(error) = result {
        eprintln!("Error: {}", error);
    }
}
```

La `main` fonction du serveur ressemble à celle du client : il effectue un peu de configuration, puis appelle `block_on` pour exécuter un bloc asynchrone qui fait le vrai travail. Pour gérer les connexions entrantes des

clients, il crée un `TcpListener` socket, dont la `incoming` méthode renvoie un flux de `std::io::Result<TcpStream>` valeurs.

Pour chaque connexion entrante, nous générerons une tâche asynchrone exécutant la `connection::serve` fonction. Chaque tâche reçoit également une référence à une `GroupTable` valeur représentant la liste actuelle des groupes de discussion de notre serveur, partagée par toutes les connexions via un `Arc` pointeur à références comptées.

Si `connection::serve` renvoie une erreur, nous enregistrons un message dans la sortie d'erreur standard et laissons la tâche se terminer. Les autres connexions continuent de fonctionner normalement.

Gestion des connexions de chat : mutex asynchrones

Voici le cheval de bataille du serveur: la `serve` fonction du `connection` module dans `src/bin/server/connection.rs` :

```
use async_chat:: {FromClient, FromServer};
use async_chat:: utils:: {self, ChatResult};
use async_std:: prelude::*;
use async_std:: io:: BufReader;
use async_std:: net:: TcpStream;
use async_std:: sync::Arc;

use crate:: group_table::GroupTable;

pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>)
    -> ChatResult<()>
{
    let outbound = Arc:: new(Outbound::new(socket.clone()));

    let buffered = BufReader:: new(socket);
    let mut from_client = utils::receive_as_json(buffered);
    while let Some(request_result) = from_client.next().await {
        let request = request_result?;

        let result = match request {
            FromClient::Join { group_name } => {
                let group = groups.get_or_create(group_name);
                group.join(outbound.clone());
                Ok(())
            }
            FromClient::Post { group_name, message } => {
                match groups.get(&group_name) {
                    Some(group) => {
                        group.post(message);
                    }
                    None => Err("Group not found".into())
                }
            }
        };
        if let Err(error) = result {
            eprintln!("Error handling request: {}", error);
        }
    }
}
```

```

        group.post(message);
        Ok(())
    }
    None => {
        Err(format!("Group '{}' does not exist", group_name))
    }
}
};

if let Err(message) = result {
    let report = FromServer::Error(message);
    outbound.send(report).await?;
}
}

Ok(())
}

```

C'est presque une image miroir de la `handle_replies` fonction du client : la majeure partie du code est une boucle gérant un flux entrant de `FromClient` valeurs, construit à partir d'un flux TCP mis en mémoire tampon avec `receive_as_json`. Si une erreur se produit, nous générerons un `FromServer::Error` paquet pour transmettre la mauvaise nouvelle au client.

En plus des messages d'erreur, les clients aimeraient également recevoir des messages des groupes de discussion qu'ils ont rejoins, de sorte que la connexion au client doit être partagée avec chaque groupe. Nous pourrions simplement donner à chacun un clone du `TcpStream`, mais si deux de ces sources essayaient d'écrire un paquet sur le socket en même temps, leur sortie pourrait être entrelacée et le client finirait par recevoir du JSON brouillé. Nous devons organiser un accès simultané sécurisé à la connexion.

Ceci est géré avec le `Outbound` type, défini dans `src/bin/server/connection.rs` comme suit :

```

use async_std::sync::Mutex;

pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
    pub fn new(to_client: TcpStream) -> Outbound {
        Outbound(Mutex::new(to_client))
    }
}

```

```

pub async fn send(&self, packet: FromServer) -> ChatResult<()> {
    let mut guard = self.0.lock().await;
    utils::send_as_json(&mut *guard, &packet).await?;
    guard.flush().await?;
    Ok(())
}

```

Une fois créée, une `Outbound` valeur prend possession de a `TcpStream` et l'enveloppe dans a `Mutex` pour s'assurer qu'une seule tâche peut l'utiliser à la fois. La `serve` fonction enveloppe chacun d'eux `Outbound` dans un `Arc` pointeur à références comptées afin que tous les groupes auxquels le client se joint puissent pointer vers la même `Outbound` instance partagée.

Un appel à `Outbound::send` first verrouille le mutex, renvoyant une valeur de garde qui déréférence à l' `TcpStream` intérieur. Nous utilisons `send_as_json` pour transmettre `packet`, puis nous appelons finalement pour nous `guard.flush()` assurer qu'il ne languira pas à moitié transmis dans un tampon quelque part. (À notre connaissance, `TcpStream` ne met pas réellement les données en mémoire tampon, mais le `Write` trait permet à ses implémentations de le faire, nous ne devrions donc pas prendre de risques.)

L'expression `&mut *guard` nous permet de contourner le fait que Rust n'applique pas de contraintes de deref pour respecter les limites des traits. Au lieu de cela, nous déréférencons explicitement la garde du mutex, puis empruntons une référence mutable à celle `TcpStream` qu'elle protège, produisant la `&mut TcpStream` qui l' `send_as_json` exige.

Notez qu'il `Outbound` utilise le `async_std::sync::Mutex` type, pas celui de la bibliothèque standard `Mutex`. Il y a trois raisons à cela.

Tout d'abord, la bibliothèque standard `Mutex` peut mal se comporter si une tâche est suspendue alors qu'elle maintient une protection mutex. Si le thread qui exécutait cette tâche sélectionne une autre tâche qui essaie de verrouiller le même `Mutex`, des problèmes surviennent : du `Mutex` point de vue de , le thread qui le possède déjà essaie de le verrouiller à nouveau. La norme `Mutex` n'est pas conçue pour gérer ce cas, donc elle panique ou se bloque. (Il n'accordera jamais le verrou de manière inappropriée.) Des travaux sont en cours pour que Rust détecte ce problème au moment de la compilation et émette un avertissement chaque fois qu'un `std::sync::Mutex` garde est actif sur une `await` expression. Puisqu'il `Outbound::send` doit maintenir le verrou en atten-

dant le futur de `send_as_json` et `guard.flush`, il doit utiliser celui `async_std` de `Mutex`.

Deuxièmement, la méthode asynchrone `Mutex` renvoie `lock` un futur d'un garde, donc une tâche attendant de verrouiller un mutex cède son thread pour que d'autres tâches l'utilisent jusqu'à ce que le mutex soit prêt. (Si le mutex est déjà disponible, le `lock` futur est prêt immédiatement, et la tâche ne se suspend pas du tout.) La méthode standard `Mutex`, `lock` d'autre part, épingle l'intégralité du thread en attendant d'acquérir le verrou. Étant donné que le code précédent contient le mutex pendant qu'il transmet un paquet sur le réseau, cela peut prendre un certain temps.

Enfin, la norme `Mutex` ne doit être déverrouillée que par le même thread qui l'a verrouillée. Pour faire respecter cela, le type de garde du mutex standard n'est pas implémenté `Send` : il ne peut pas être transmis à d'autres threads. Cela signifie qu'un futur contenant une telle garde n'implémente pas lui-même `Send`, et ne peut pas être passé à `spawn` pour s'exécuter sur un pool de threads ; il ne peut être exécuté qu'avec `block_on` ou `spawn_local`. Le garde d'un `async_std Mutex` implémente `Send` donc il n'y a aucun problème à l'utiliser dans les tâches engendrées.

La table de groupe : mutex synchrones

Mais la morale de l'histoire n'est pas aussi simple que "Toujours utiliser `async_std::sync::Mutex` en code asynchrone". Souvent, il n'est pas nécessaire d'attendre quoi que ce soit en maintenant un mutex, et le verrou n'est pas maintenu longtemps. Dans de tels cas, la bibliothèque standard `Mutex` peut-être beaucoup plus efficace. Le type de notre serveur de chat `GroupTable` illustre ce cas. Voici le contenu complet de `src/bin/server/group_table.rs` :

```
use crate:: group:: Group;
use std:: collections:: HashMap;
use std:: sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
    pub fn new() -> GroupTable {
        GroupTable(Mutex:: new(HashMap:: new()))
    }

    pub fn get(&self, name: &String) -> Option<Arc<Group>> {
        self.0.lock().unwrap().get(name)
    }
}
```

```

        self.0.lock()
        .unwrap()
        .get(name)
        .cloned()
    }

    pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
        self.0.lock()
        .unwrap()
        .entry(name.clone())
        .or_insert_with(|| Arc::new(Group::new(name)))
        .clone()
    }
}

```

A `GroupTable` est simplement une table de hachage protégée par mutex, mappant les noms des groupes de discussion aux groupes réels, tous deux gérés à l'aide de pointeurs comptés par référence. Les méthodes `get` et `get_or_create` verrouillent le mutex, effectuent quelques opérations de table de hachage, peut-être quelques allocations, et retournent.

Dans `GroupTable`, nous utilisons un vieux simple `std::sync::Mutex`. Il n'y a pas du tout de code asynchrone dans ce module, il n'y a donc pas `await` de s à éviter. En effet, si nous voulions utiliser `async_std::sync::Mutex` ici, nous aurions besoin de faire `get` et `get_or_create` dans des fonctions asynchrones, ce qui introduit le sur-coût de futures créations, suspensions et reprises pour peu d'avantages : le mutex n'est verrouillé que pour certaines opérations de hachage et peut-être quelques allocations.

Si notre serveur de chat se retrouvait avec des millions d'utilisateurs et que le `GroupTable` mutex devenait un goulot d'étranglement, le rendre asynchrone ne résoudrait pas ce problème. Il serait probablement préférable d'utiliser une sorte de type de collection spécialisé pour l'accès simultané au lieu de `HashMap`. Par exemple, la `dashmap` caisse fournit un tel type.

Groupes de discussion : chaînes de diffusion de tokio

Dans notre serveur, le `group::Group` genrereprésente un groupe de discussion. Ce type doit uniquement prendre en charge les deux méthodes qui `connection::serve` appellent : `join`, pour ajouter un nouveau membre, et `post`, pour publier un message. Chaque message posté doit être distribué à tous les membres.

C'est là que nous abordons le défi mentionné précédemment de la contre-*pression*. Il y a plusieurs besoins en tension les uns avec les autres :

- Si un membre ne peut pas suivre les messages envoyés au groupe (s'il a une connexion réseau lente, par exemple), les autres membres du groupe ne devraient pas être affectés.
- Même si un membre prend du retard, il devrait y avoir un moyen pour lui de rejoindre la conversation et de continuer à participer d'une manière ou d'une autre.
- La mémoire utilisée pour la mise en mémoire tampon des messages ne doit pas croître sans limite.

Étant donné que ces défis sont courants lors de la mise en œuvre de modèles de communication plusieurs à plusieurs, la `tokio` caisse fournit un type de *canal de diffusion* qui met en œuvre un ensemble raisonnable de compromis. Un `tokio` canal de diffusion est une file d'attente de valeurs (dans notre cas, des messages de chat) qui permet à un nombre quelconque de threads ou de tâches différents d'envoyer et de recevoir des valeurs. C'est ce qu'on appelle un canal de « diffusion », car chaque consommateur reçoit sa propre copie de chaque valeur envoyée. (Le type de valeur doit implémenter `Clone`.)

Normalement, une chaîne de 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 ne pouvaient pas suivre reçoivent une erreur la prochaine fois qu'ils tentent de recevoir leur prochain message, et le canal les rattrape jusqu'au message le plus ancien encore disponible.

Par exemple, la [Figure 20-4](#) montre un canal de diffusion avec une capacité maximale de 16 valeurs.

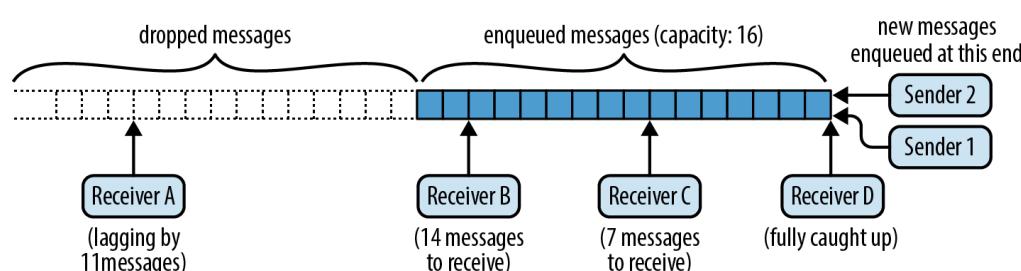


Illustration 20-4. Une chaîne de diffusion tokio

Deux expéditeurs mettent les messages en file d'attente et quatre destinataires les retirent de la file d'attente, ou plus précisément, copient les messages hors de la file d'attente. Le récepteur B a encore 14 messages à recevoir, le récepteur C en a 7 et le récepteur D est entièrement rattrapé. Le

récepteur A a pris du retard et 11 messages ont été abandonnés avant qu'il ne puisse les voir. Sa prochaine tentative de réception d'un message échouera, renvoyant une erreur indiquant la situation, et il sera rattrapé à la fin actuelle de la file d'attente.

Notre serveur de chat représente chaque groupe de chat comme un canal de diffusion porteur de `Arc<String>` valeurs : poster un message au groupe le diffuse à tous les membres actuels. Voici la définition du `group::Group` type, défini dans `src/bin/server/group.rs` :

```
use async_std:: task;
use crate:: connection:: Outbound;
use std:: sync:: Arc;
use tokio:: sync::broadcast;

pub struct Group {
    name: Arc<String>,
    sender: broadcast::Sender<Arc<String>>
}

impl Group {
    pub fn new(name: Arc<String>) -> Group {
        let (sender, _receiver) = broadcast::channel(1000);
        Group { name, sender }
    }

    pub fn join(&self, outbound: Arc<Outbound>) {
        let receiver = self.sender.subscribe();

        task::spawn(handle_subscriber(self.name.clone(),
                                      receiver,
                                      outbound));
    }

    pub fn post(&self, message: Arc<String>) {
        // This only returns an error when there are no subscribers. A
        // connection's outgoing side can exit, dropping its subscription,
        // slightly before its incoming side, which may end up trying to se
        let _ignored = self.sender.send(message);
    }
}
```

Une `Group` structure contient le nom du groupe de discussion, ainsi qu'un `broadcast::Sender` représentant l'extrémité d'envoi du canal de diffusion du groupe. La `Group::new` fonction appelle `broadcast::channel` à créer un canal de diffusion avec une capacité maximale de 1 000 messages. La `channel` fonction renvoie à la fois un

expéditeur et un destinataire, mais nous n'avons pas besoin du destinataire à ce stade, puisque le groupe n'a pas encore de membres.

Pour ajouter un nouveau membre au groupe, la `Group::join` méthode appelle la méthode de l'expéditeur `subscribe` pour créer un nouveau récepteur pour le canal. Ensuite, il génère une nouvelle tâche asynchrone pour surveiller ce récepteur pour les messages et les réécrire au client, dans la `handle_subscribe` fonction.

Avec ces détails en main, la `Group::post` méthode est simple : elle envoie simplement le message au canal de diffusion. Étant donné que les valeurs transportées par le canal sont des `Arc<String>` valeurs, donner à chaque récepteur sa propre copie d'un message augmente simplement le nombre de références du message, sans aucune copie ni allocation de tas. Une fois que tous les abonnés ont transmis le message, le compteur de références tombe à zéro et le message est libéré.

Voici la définition de `handle_subscriber`:

```
use async_chat:: FromServer;
use tokio:: sync:: broadcast:: error::RecvError;

async fn handle_subscriber(group_name: Arc<String>,
                           mut receiver: broadcast:: Receiver<Arc<String>>,
                           outbound: Arc<Outbound>)
{
    loop {
        let packet = match receiver.recv().await {
            Ok(message) => FromServer:: Message {
                group_name: group_name.clone(),
                message: message.clone(),
            },
            Err(RecvError:: Lagged(n)) => FromServer:: Error(
                format!("Dropped {} messages from {}.", n, group_name)
            ),
            Err(RecvError:: Closed) => break,
        };

        if outbound.send(packet).await.is_err() {
            break;
        }
    }
}
```

Bien que les détails soient différents, la forme de cette fonction est familière : c'est une boucle qui reçoit les messages du canal de diffusion et les retransmet au client via la `Outbound` valeur partagée. Si la boucle ne peut pas suivre le canal de diffusion, elle reçoit une `Lagged` erreur, qu'elle signale consciencieusement au client.

Si le renvoi d'un paquet au client échoue complètement, peut-être parce que la connexion s'est fermée, `handle_subscriber` quitte sa boucle et revient, provoquant la fermeture de la tâche asynchrone. Cela supprime le canal de diffusion `Receiver`, en le désabonnant du canal. De cette façon, lorsqu'une connexion est abandonnée, chacune de ses appartenances au groupe est nettoyée la prochaine fois que le groupe essaie de lui envoyer un message.

Nos groupes de discussion ne se ferment jamais, car nous ne supprimons jamais un groupe de la table des groupes, mais juste pour être complet, nous sommes `handle_subscriber` prêts à gérer une `Closed` erreur en quittant la tâche.

Notez que nous créons une nouvelle tâche asynchrone pour chaque appartenance à un groupe de chaque client. Cela est possible car les tâches asynchrones utilisent beaucoup moins de mémoire que les threads et parce que le passage d'une tâche asynchrone à une autre au sein d'un processus est assez efficace.

Ceci est donc le code complet du serveur de chat. C'est un peu spartiate, et il y a beaucoup plus de fonctionnalités intéressantes dans les `async_std`, `tokio` et `futures` crates que nous ne pouvons couvrir dans ce livre, mais idéalement cet exemple étendu parvient à illustrer comment certaines des fonctionnalités de l'écosystème asynchrone fonctionnent ensemble : tâches asynchrones, flux, les traits d'E/S asynchrones, les canaux et les mutex des deux versions.

Futurs primitifs et exécuteurs : quand un futur mérite-t-il à nouveau d'être interrogé ?

La `DiscussionServer` montre comment nous pouvons écrire du code en utilisant des primitives asynchrones comme `TcpListener` et le `broadcast` canal, et utiliser des exécuteurs comme `block_on` et `spawn` pour piloter leur exécution. Maintenant, nous pouvons jeter un œil à la façon dont ces choses sont mises en œuvre. La question clé est,

quand un futur revient `Poll::Pending`, comment se coordonne-t-il avec l'exécuteur testamentaire pour l'interroger à nouveau au bon moment ?

Pensez à ce qui se passe lorsque nous exécutons un code comme celui-ci, à partir de la fonction du client de chat `main` :

```
task:: block_on(async {
    let socket = net:: TcpStream::connect(address).await?;
    ...
})
```

Les premiers `block_on` sondent l'avenir du bloc asynchrone, la connexion réseau n'est presque certainement pas prête immédiatement, donc `block_on` se met en veille. Mais quand devrait-il se réveiller en haut? D'une manière ou d'une autre, une fois que la connexion réseau est prête, il `TcpStream` doit indiquer `block_on` qu'il doit réessayer d'interroger l'avenir du bloc asynchrone, car il sait que cette fois, le `await` sera terminé et que l'exécution du bloc asynchrone peut progresser.

Lorsqu'un exécuteur comme `block_on` interroge un futur, il doit passer un rappel appelé un *waker*. Si l'avenir n'est pas encore prêt, les règles du `Future` trait disent qu'il doit revenir `Poll::Pending` pour le moment et faire en sorte que l'éveilleur soit invoqué plus tard, si et quand l'avenir vaut la peine d'être interrogé à nouveau.

Ainsi, une implémentation manuscrite de `Future` ressemble souvent à ceci :

```
use std:: task::Waker;

struct MyPrimitiveFuture {
    ...
    waker: Option<Waker>,
}

impl Future for MyPrimitiveFuture {
    type Output = ...;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<...> {
        ...
        if ... future is ready ... {
            return Poll::Ready(final_value);
        }
    }
}
```

```

    // Save the waker for later.
    self.waker = Some(cx.waker().clone());
    Poll::Pending
}
}

```

En d'autres termes, si la valeur du futur est prête, retournez-la. Sinon, Context placez quelque part un clone de l'éveilleur de et retournez Poll::Pending .

Lorsque le futur vaut la peine d'être interrogé à nouveau, le futur doit notifier le dernier exécuteur qui l'a interrogé en appelant la wake méthode de son éveilleur :

```

// If we have a waker, invoke it, and clear `self.waker`.
if let Some(waker) = self.waker.take() {
    waker.wake();
}

```

Idéalement, l'exécuteur et le futur interrogent et se réveillent à tour de rôle: l'exécuteur interroge le futur et s'endort, puis le futur invoque le réveil, de sorte que l'exécuteur se réveille et interroge à nouveau le futur.

Les futurs des fonctions et des blocs asynchrones ne traitent pas des wakers eux-mêmes. Ils transmettent simplement le contexte qui leur est donné aux sous-futurs qu'ils attendent, leur déléguant l'obligation de sauver et d'invoquer des éveilleurs. Dans notre client de chat, le premier sondage sur l'avenir du bloc asynchrone transmet simplement le contexte lorsqu'il attend `TcpStream::connect` l'avenir de . Les sondages suivants transmettent de la même manière leur contexte à l'avenir que le bloc attend ensuite.

`TcpStream::connect` Le futur de gère l'interrogation comme indiqué dans l'exemple précédent : il transmet l'éveilleur à un thread d'assistance qui attend que la connexion soit prête, puis appelle l'éveilleur.

`Waker` implémente `Clone` et `Send`, ainsi un futur peut toujours faire sa propre copie du waker et l'envoyer à d'autres threads si nécessaire. La `Waker::wake` méthode consomme le waker. Il existe également une `wake_by_ref` méthode qui ne le fait pas, mais certains exécuteurs peuvent implémenter la version consommatrice un peu plus efficacement. (La différence est au plus de `clone`.)

Il est inoffensif pour un exécuteur de surinterroger un futur, simplement inefficace. Les contrats à terme, cependant, devraient veiller à n'invoquer

un réveil que lorsque l'interrogation ferait des progrès réels : un cycle de réveils et d'interrogations parasites peut empêcher un exécuteur de dormir, gaspillant de l'énergie et laissant le processeur moins réactif aux autres tâches.

Maintenant que nous avons montré comment les exécuteurs et les futurs primitifs communiquent, nous allons implémenter nous-mêmes un futur primitif, puis parcourir une implémentation de l' `block_on` exécuteur.

Invocation de Wakers : `spawn_blocking`

Plus tôt dans le chapitre, nous avons décrit la `spawn_blocking` fonction, qui démarre une fermeture donnée s'exécutant sur un autre thread et renvoie un futur de sa valeur de retour. Nous avons maintenant toutes les pièces dont nous avons besoin pour `spawn_blocking` nous mettre en œuvre. Pour plus de simplicité, notre version crée un nouveau thread pour chaque fermeture, plutôt que d'utiliser un pool de threads, comme `async_std` le fait la version de .

Bien que `spawn_blocking` renvoie un futur, nous n'allons pas l'écrire comme un `async fn`. Il s'agira plutôt d'une fonction ordinaire et synchrone qui renvoie une structure, `SpawnBlocking`, sur laquelle nous nous implémenterons `Future` nous-mêmes.

La signature de notre `spawn_blocking` est la suivante :

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T:Send + 'static,
```

Puisque nous devons envoyer la fermeture à un autre thread et ramener la valeur de retour, la fermeture `F` et sa valeur de retour `T` doivent implémenter `Send`. Et comme nous n'avons aucune idée de la durée de fonctionnement du thread, ils doivent l'être tous les deux `'static` également. Ce sont les mêmes limites que lui- `std::thread::spawn` même impose.

`SpawnBlocking<T>` est un futur de la valeur de retour de la fermeture. Voici sa définition :

```
use std::sync:: {Arc, Mutex};
use std::task::Waker;

pub struct SpawnBlocking<T>(Arc<Mutex<Shared<T>>>);
```

```

    struct Shared<T> {
        value: Option<T>,
        waker: Option<Waker>,
    }
}

```

La `Shared` structure doit servir de rendez-vous entre le futur et le thread exécutant la fermeture, il appartient donc à un `Arc` et est protégé par un `Mutex`. (Un mutex synchrone convient ici.) L'interrogation du futur vérifie si `value` est présent et enregistre le réveil dans le `waker` cas contraire. Le thread qui exécute la fermeture enregistre sa valeur de retour dans `value` puis appelle `waker`, s'il est présent.

Voici la définition complète de `spawn_blocking`:

```

pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T: Send + 'static,
{
    let inner = Arc::new(Mutex::new(Shared {
        value: None,
        waker: None,
    }));
    std::thread::spawn({
        let inner = inner.clone();
        move || {
            let value = closure();

            let maybe_waker = {
                let mut guard = inner.lock().unwrap();
                guard.value = Some(value);
                guard.waker.take()
            };

            if let Some(waker) = maybe_waker {
                waker.wake();
            }
        }
    });
    SpawnBlocking(inner)
}

```

Après avoir créé la `Shared` valeur, cela génère un thread pour exécuter la fermeture, stocker le résultat dans le champ `Shared`'s `value` et invoquer le `waker`, le cas échéant.

Nous pouvons implémenter Future pour SpawnBlocking comme suit:

```
use std::future:: Future;
use std::pin:: Pin;
use std::task::{Context, Poll};

impl<T:Send> Future for SpawnBlocking<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
        let mut guard = self.0.lock().unwrap();
        if let Some(value) = guard.value.take() {
            return Poll::Ready(value);
        }

        guard.waker = Some(cx.waker().clone());
        Poll::Pending
    }
}
```

L'interrogation d'un SpawnBlocking vérifie si la valeur de la fermeture est prête, prend possession et la renvoie si c'est le cas. Sinon, le futur est toujours en attente, il enregistre donc un clone de l'éveil du contexte dans le waker champ du futur.

Une fois que Future a été revenu Poll::Ready, vous n'êtes pas censé l'interroger à nouveau. Les façons habituelles de consommer des contrats à terme, comme await et block_on, respectent toutes cette règle. Si un SpawnBlocking avenir est dépassé, rien de particulièrement terrible ne se produit, mais il ne fait aucun effort pour gérer ce cas non plus. Ceci est typique des contrats à terme manuscrits.

Implémentation de block_on

en outre pour pouvoir implémenter des futurs primitifs, nous avons également toutes les pièces dont nous avons besoin pour construire un exécuteur simple. Dans cette section, nous allons écrire notre propre version de block_on. Ce sera un peu plus simple que `async_std` la version de ; par exemple, il ne prend pas en charge `spawn_local`, les variables locales de tâche ou les invocations imbriquées (appels `block_on` à partir de code asynchrone). Mais c'est suffisant pour exécuter notre client et serveur de chat.

Voici le code :

```

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 `crossbeam` de crate `Parker` est une simple primitive bloquante : l'appel `parker.park()` bloque le thread jusqu'à ce que quelqu'un d'autre appelle `.unpark()` le correspondant `Unparker`, que vous obtenez au préalable en appelant `parker.unparker()`. Si vous avez `unpark` un thread qui n'est pas encore parqué, son prochain appel à `park` revient immédiatement, sans blocage. Notre `block_on` utilisera le `Parker` pour attendre chaque fois que le futur n'est pas prêt, et le réveil que nous passons au futur le débloquera.

```

let waker = waker_fn(move || unparker.unpark());

```

La `waker_fn` fonction, à partir de la caisse du même nom, crée un `Waker` à partir d'une fermeture donnée. Ici, nous faisons un `Waker` qui, lorsqu'il est invoqué, appelle la fermeture `move || unparker.unpark()`. Vous pouvez également créer des wakers en implémentant le `std::task::Wake` trait, mais `waker_fn` c'est un peu plus pratique ici.

```
pin!(future);
```

Étant donné une variable contenant un futur de type `F`, la `pin!` macro s'approprie le futur et déclare une nouvelle variable de même nom dont le type est `Pin<&mut F>` et qui emprunte le futur. Cela nous donne le `Pin<&mut Self>` requis par la `poll` méthode. Pour des raisons que nous expliquerons dans la section suivante, les futurs des fonctions et des blocs asynchrones doivent être référencés via un `Pin` avant de pouvoir être interrogés.

```
loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

Enfin, la boucle de sondage est assez simple. Passant un contexte portant notre réveil, nous interrogeons le futur jusqu'à ce qu'il revienne `Poll::Ready`. S'il retourne `Poll::Pending`, nous garons le thread, qui se bloque jusqu'à ce qu'il `waker` soit appelé. Puis nous réessayons.

L'`as_mut` appel nous permet d'interroger `future` sans renoncer à la propriété ; nous expliquerons cela plus en détail dans la section suivante.

Épingler

Bien que les fonctions et les blocs asynchrones sont essentiels pour écrire du code asynchrone clair, la gestion de leur avenir nécessite un peu de prudence. Le `Pin` type aide Rust à s'assurer qu'ils sont utilisés en toute sécurité.

Dans cette section, nous montrerons pourquoi les contrats à terme d'appels de fonction et de blocs asynchrones ne peuvent pas être gérés aussi librement que les valeurs Rust ordinaires. Ensuite, nous montrerons comment `Pin` sert de «sceau d'approbation» sur des pointeurs sur lesquels on peut compter pour gérer ces contrats à terme en toute sécurité. Enfin, nous montrerons quelques façons de travailler avec des `Pin` valeurs.

Les deux étapes de la vie d'un avenir

Considérez cette fonction asynchrone simple :

```

use async_std:: io:: prelude::*;
use async_std:: {io, net};

async fn fetch_string(address: &str) -> io:: Result<String> {
    ①
    let mut socket = net:: TcpStream:: connect(address).await②?;
    let mut buf = String::new();
    socket.read_to_string(&mut buf).await③?;
    Ok(buf)
}

```

Cela ouvre une connexion TCP à l'adresse donnée et renvoie, sous forme de `String`, tout ce que le serveur veut envoyer. Les points étiquetés ①, ② et ③ sont les points de *reprise*, les points dans le code de la fonction asynchrone auxquels l'exécution peut être suspendue.

Supposons que vous l'appeliez, sans attendre, comme ceci :

```
let response = fetch_string("localhost:6502");
```

Now `response` est un futur prêt à commencer l'exécution au début de `fetch_string`, avec l'argument donné. En mémoire, le futur ressemble à [la figure 20-5](#).

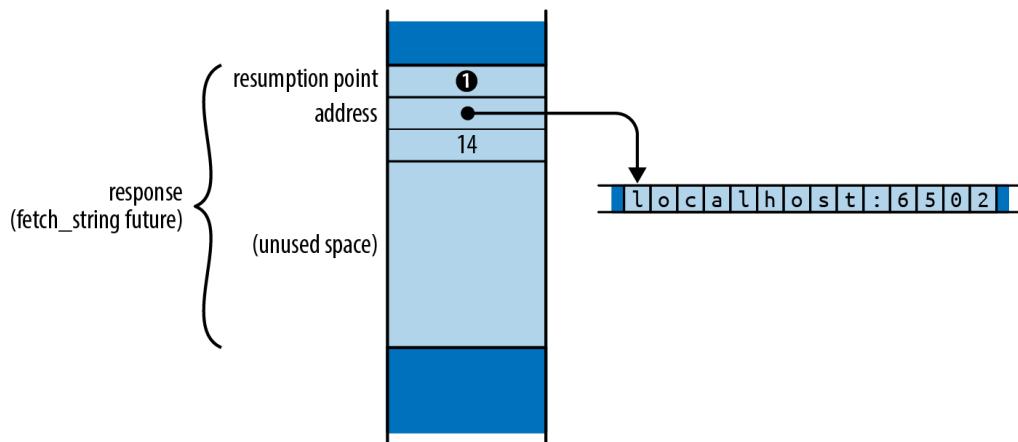


Illustration 20-5. L'avenir construit pour un appel à `fetch_string`

Puisque nous venons de créer ce futur, il est indiqué que l'exécution doit commencer au point de reprise ①, en haut du corps de la fonction. Dans cet état, les seules valeurs dont un futur a besoin pour continuer sont les arguments de la fonction.

Supposons maintenant que vous interrogez `response` plusieurs fois et qu'il atteigne ce point dans le corps de la fonction :

```
socket.read_to_string(&mut buf).await③?;
```

Supposons en outre que le résultat de `read_to_string` n'est pas prêt, donc le sondage renvoie `Poll::Pending`. À ce stade, le futur ressemble à la [Figure 20-6](#).

Un futur doit toujours contenir toutes les informations nécessaires pour reprendre l'exécution la prochaine fois qu'il est interrogé. Dans ce cas c'est :

- Point de reprise ❸, disant que l'exécution devrait reprendre dans le futur du `await scrutin.read_to_string`
- Les variables actives à ce point de reprise : `socket` et `buf`. La valeur de `address` n'est plus présente dans le futur, puisque la fonction n'en a plus besoin.
- Le `read_to_string` sous-futur, dont l'`await` expression est en pleine vogue.

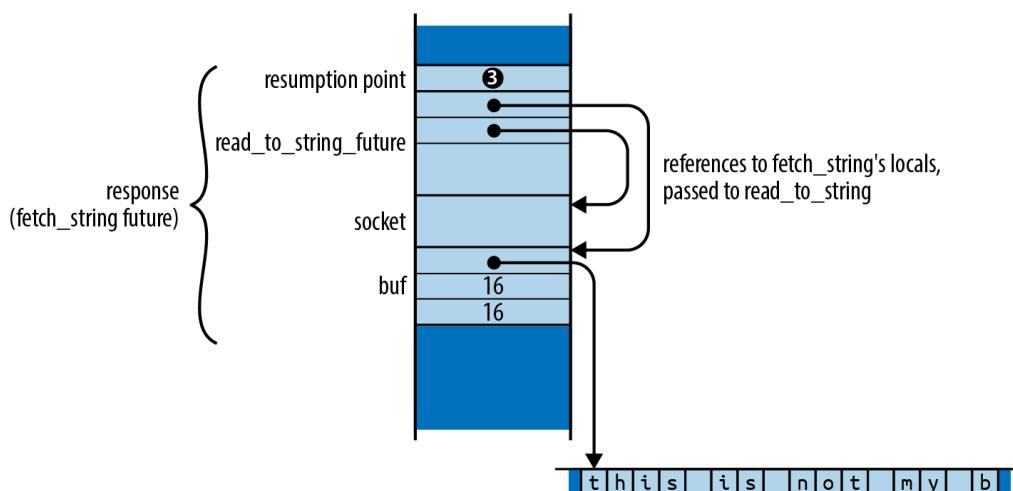


Illustration 20-6. Le même avenir, au milieu de l'attente `read_to_string`

Notez que l'appel à `read_to_string` a emprunté des références à `socket` et `buf`. Dans une fonction synchrone, toutes les variables locales vivent sur la pile, mais dans une fonction asynchrone, les variables locales qui sont actives sur un `await` doivent être localisées dans le futur, elles seront donc disponibles lorsqu'elle sera à nouveau interrogée. Emprunter une référence à une telle variable emprunte une partie du futur.

Cependant, Rust exige que les valeurs ne soient pas déplacées pendant qu'elles sont empruntées. Supposons que vous deviez déplacer ce futur vers un nouvel emplacement :

```
let new_variable = response;
```

Rust n'a aucun moyen de trouver toutes les références actives et de les ajuster en conséquence. Au lieu de pointer vers `socket` et `buf` vers leurs nouveaux emplacements, les références continuent de pointer vers leurs

anciens emplacements dans le fichier `response`. Ils sont devenus des pointeurs pendants, comme le montre la [Figure 20-7](#).

Empêcher les valeurs empruntées d'être déplacées est généralement la responsabilité du vérificateur d'emprunt. Le vérificateur d'emprunt traite les variables comme les racines des arbres de propriété, mais contrairement aux variables stockées sur la pile, les variables stockées dans les contrats à terme sont déplacées si le futur lui-même se déplace. Cela signifie que les emprunts de `socket` et `buf` affectent non seulement ce qui `fetch_string` peut être fait avec ses propres variables, mais aussi ce que son appelant peut faire en toute sécurité avec `response`, le futur qui les détient. L'avenir des fonctions asynchrones est un angle mort pour le vérificateur d'emprunt, que Rust doit couvrir d'une manière ou d'une autre s'il veut tenir ses promesses de sécurité de la mémoire.

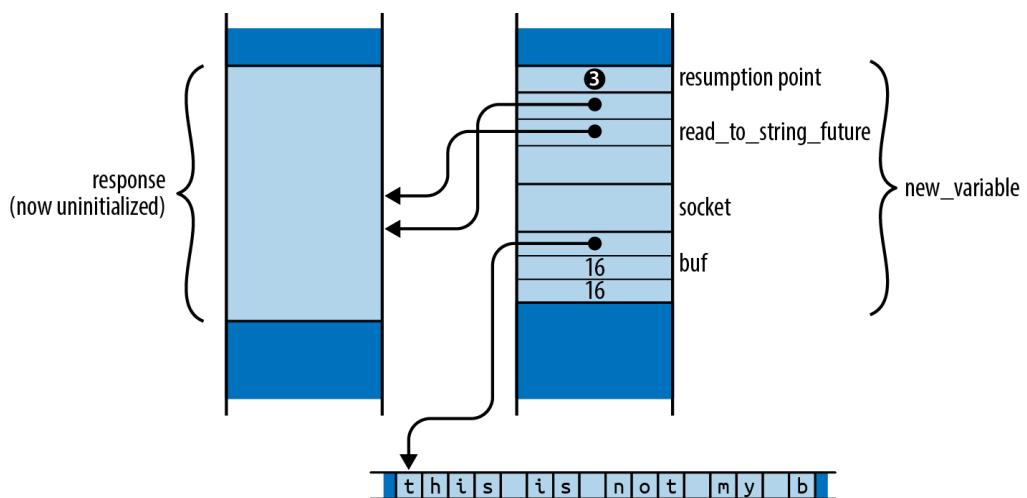


Illustration 20-7. `fetch_string`'s future, déplacé pendant l'emprunt (Rust empêche cela)

La solution de Rust à ce problème repose sur l'idée que les contrats à terme sont toujours sûrs à déplacer lorsqu'ils sont créés pour la première fois et ne deviennent dangereux à déplacer que lorsqu'ils sont interrogés. Un futur qui vient d'être créé en appelant une fonction asynchrone contient simplement un point de reprise et les valeurs des arguments. Celles-ci ne concernent que le corps de la fonction asynchrone, qui n'a pas encore commencé son exécution. Seul l'interrogation d'un futur peut emprunter son contenu.

À partir de là, nous pouvons voir que chaque futur a deux étapes de vie :

- La première étape commence lorsque l'avenir est créé. Étant donné que le corps de la fonction n'a pas commencé son exécution, aucune partie de celle-ci ne peut encore être empruntée. À ce stade, il est aussi sûr de se déplacer que n'importe quelle autre valeur de Rust.
- La deuxième étape commence la première fois que le futur est interrogé. Une fois que le corps de la fonction a commencé son exécution, il

peut emprunter des références à des variables stockées dans le futur, puis attendre, laissant cette partie du futur empruntée. À partir de son premier sondage, nous devons supposer que l'avenir n'est peut-être pas sûr.

La flexibilité de la première étape de la vie est ce qui nous permet de passer les contrats à terme à `block_on` et `spawn` et d'appeler des méthodes d'adaptation comme `race` et `fuse`, qui prennent toutes les contrats à terme par valeur. En fait, même l'appel de fonction asynchrone qui a créé le futur en premier lieu devait le renvoyer à l'appelant ; c'était un mouvement aussi.

Pour entrer dans sa deuxième étape de vie, l'avenir doit être interrogé. La `poll` méthode nécessite que le futur soit passé en tant que `Pin<&mut Self>` valeur. `Pin` est un wrapper pour les types de pointeurs (comme `&mut Self`) qui limite la façon dont les pointeurs peuvent être utilisés, garantissant que leurs référents (comme `Self`) ne peuvent plus jamais être déplacés. Vous devez donc produire un `Pin` pointeur encapsulé vers le futur avant de pouvoir l'interroger.

Voici donc la stratégie de Rust pour assurer la sécurité des futurs : un futur ne peut pas devenir dangereux à déplacer tant qu'il n'est pas interrogé ; vous ne pouvez pas interroger un futur tant que vous n'avez pas construit un `Pin` pointeur encapsulé vers celui-ci ; et une fois que vous avez fait cela, l'avenir ne peut pas être déplacé.

"Une valeur que vous ne pouvez pas déplacer" semble impossible : les déplacements sont partout dans Rust. Nous expliquerons exactement comment `Pin` protège les contrats à terme dans la section suivante.

Bien que cette section ait traité des fonctions asynchrones, tout ici s'applique également aux blocs asynchrones. Un futur fraîchement créé d'un bloc asynchrone capture simplement les variables qu'il utilisera à partir du code environnant, comme une fermeture. Seul l'interrogation du futur peut créer des références à son contenu, le rendant dangereux à déplacer.

Gardez à l'esprit que cette fragilité de mouvement est limitée aux futures des fonctions et des blocs asynchrones, avec leurs `Future` implémentations spéciales générées par le compilateur. Si vous implémentez `Future` à la main pour vos propres types, comme nous l'avons fait pour notre `SpawnBlocking` type dans "[Invoking Wakers: spawn_blocking](#)", ces futurs sont parfaitement sûrs à déplacer à la fois avant et après avoir été interrogés. Dans toute implémentation manuscrite `poll`, le vérificateur d'emprunt garantit que toutes les références que vous avez emprun-

tées à des parties de `self` ont disparu au moment du `poll` retour. Ce n'est que parce que les fonctions et les blocs asynchrones ont le pouvoir de suspendre l'exécution au milieu d'un appel de fonction, avec des emprunts en cours, que nous devons gérer leur avenir avec soin.

Pointeurs épingleés

Le `Pin` type est un emballage pour les pointeurs vers des contrats à terme qui limitent la façon dont les pointeurs peuvent être utilisés pour s'assurer que les contrats à terme ne peuvent pas être déplacés une fois qu'ils ont été interrogés. Ces restrictions peuvent être levées pour les futurs qui ne craignent pas d'être déplacés, mais elles sont essentielles pour interroger en toute sécurité les futurs des fonctions et des blocs asynchrones.

Par *pointeur*, nous entendons tout type qui implémente `Deref`, et éventuellement `DerefMut`. Un `Pin` pointeur enroulé autour d'un pointeur est appelé un *pointeur épingleé*. `Pin<&mut T>` et `Pin<Box<T>>` sont typiques.

La définition de `Pin` dans la bibliothèque standard est simple :

```
pub struct Pin<P> {
    pointer:P,
}
```

Notez que le `pointer` champ n'est *pas pub*. Cela signifie que la seule façon de construire ou d'utiliser un `Pin` passe par les méthodes soigneusement choisies fournies par le type.

Étant donné le futur d'une fonction ou d'un bloc asynchrone, il n'y a que quelques façons d'obtenir un pointeur épingleé vers celui-ci :

- La `pin!` macro, depuis le `futures-lite` crate, masque une variable de type `T` avec une nouvelle de type `Pin<&mut T>`. La nouvelle variable pointe vers la valeur d'origine, qui a été déplacée vers un emplacement temporaire anonyme sur la pile. Lorsque la variable sort de la portée, la valeur est supprimée. Nous avons utilisé `pin!` dans notre `block_on` implémentation pour épingleer le futur que nous voulions interroger.
- Le `Box::pin` constructeur de la bibliothèque standard s'approprie une valeur de n'importe quel type `T`, la déplace dans le tas et renvoie un `Pin<Box<T>>`.
- `Pin<Box<T>>` implements `From<Box<T>>`, `Pin::from(boxed)` s'approprie donc `boxed` et vous restitue une boîte épingleée pointant vers

la même chose T sur le tas.

Chaque façon d'obtenir un pointeur épinglé vers ces contrats à terme implique de renoncer à la propriété du futur, et il n'y a aucun moyen de le récupérer. Le pointeur épinglé lui-même peut être déplacé comme bon vous semble, bien sûr, mais le déplacement d'un pointeur ne déplace pas son référent. Ainsi, la possession d'un pointeur épinglé vers un futur sert de preuve que vous avez définitivement renoncé à la capacité de déplacer ce futur. C'est tout ce dont nous avons besoin pour savoir qu'il peut être interrogé en toute sécurité.

Une fois que vous avez épinglé un futur, si vous souhaitez l'interroger, tous les `Pin<pointer to T>` types ont une `as_mut` méthode qui déréférence le pointeur et renvoie le `Pin<&mut T>` qui l' `poll` exige.

La `as_mut` méthode peut également vous aider à interroger un avenir sans renoncer à la propriété. Notre `block_on` implémentation l'a utilisé dans ce rôle :

```
pin!(future);

loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

Ici, la `pin!` macro a été redéclarée `future` en tant que `Pin<&mut F>`, nous pouvons donc simplement la transmettre à `poll`. Mais les références mutables ne sont pas `Copy`, donc `Pin<&mut F>` ne peuvent pas l'être non `Copy` plus, ce qui signifie que l'appel `future.poll()` direct prendrait possession de `future`, laissant la prochaine itération de la boucle avec une variable non initialisée. Pour éviter cela, nous appelons `future.as_mut()` à réemprunter un nouveau `Pin<&mut F>` pour chaque itération de boucle.

Il n'y a aucun moyen d'obtenir une `&mut` référence à un futur épinglé : si vous le pouviez, vous pourriez utiliser `std::mem::replace` ou `std::mem::swap` pour le déplacer et mettre un futur différent à sa place.

La raison pour laquelle nous n'avons pas à nous soucier d'épingler des contrats à terme dans du code asynchrone ordinaire est que les moyens les plus courants d'obtenir la valeur d'un contrat à terme - l'attendre ou le transmettre à un exécuteur - s'approprient tous le futur et gèrent l'épin-

glage en interne. Par exemple, notre `block_on` implémentation s'approprie l'avenir et utilise la `pin!` macro pour produire le `Pin<&mut F>` nécessaire à interroger. Une `await` expression s'approprie également le futur et utilise une approche similaire à la `pin!` macro en interne.

Le trait de détachement

Cependant, tous les contrats à terme ne nécessitent pas ce type de manipulation prudente. Pour toute implémentation manuscrite d'`Future` un type ordinaire, comme notre `SpawnBlocking` type mentionné précédemment, les restrictions sur la construction et l'utilisation de pointeurs épinglés sont inutiles.

Ces types durables implémentent le `Unpin` marqueur caractéristique:

```
trait Unpin { }
```

Presque tous les types de Rust implémentent automatiquement `Unpin`, en utilisant un support spécial dans le compilateur. Les fonctions asynchrones et les contrats à terme de blocs sont les exceptions à cette règle.

Pour `Unpin` les types, `Pin` n'impose aucune restriction. Vous pouvez créer un pointeur épinglé à partir d'un pointeur ordinaire avec `Pin::new` et récupérer le pointeur avec `Pin::into_inner`. Le lui-même passe le long des implémentations et des implémentations `Pin` du pointeur `.Deref DerefMut`

Par exemple, `String` implements `Unpin`, nous pouvons donc écrire :

```
let mut string = "Pinned?".to_string();
let mut pinned: Pin<&mut String> = Pin::new(&mut string);

pinned.push_str(" Not");
Pin::into_inner(pinned).push_str(" so much.");

let new_home = string;
assert_eq!(new_home, "Pinned? Not so much.");
```

Même après avoir créé un `Pin<&mut String>`, nous avons un accès mutable complet à la chaîne et pouvons le déplacer vers une nouvelle variable une fois que le `Pin` a été consommé par `into_inner` et que la référence mutable a disparu. Donc, pour les types qui sont `Unpin` - ce qui est presque tous - il y a une enveloppe ennuyeuse autour des pointeurs vers ce type.

Cela signifie que lorsque vous implémentez `Future` pour vos propres `Unpin` types, votre `pool` implémentation peut traiter `self` comme s'il s'agissait de `&mut Self`, et non de `Pin<&mut Self>`. L'épinglage devient quelque chose que vous pouvez généralement ignorer.

Il peut être surprenant d'apprendre cela `Pin<&mut F>` et de l'`Pin<Box<F>>` implémenter `Unpin`, même si ce `F` n'est pas le cas. Cela ne se lit pas bien—comment est-ce `Pin` possible `Unpin`?—mais si vous réfléchissez bien à la signification de chaque terme, cela a du sens. Même s'il `F` n'est pas sûr de se déplacer une fois qu'il a été interrogé, un pointeur vers celui-ci peut toujours être déplacé en toute sécurité, interrogé ou non. Seul le pointeur se déplace ; son référent fragile reste en place.

Ceci est utile pour savoir quand vous souhaitez passer le futur d'une fonction ou d'un bloc asynchrone à une fonction qui n'accepte que `Unpin` les futurs. (De telles fonctions sont rares dans `async_std`, mais moins ailleurs dans l'écosystème asynchrone.) `Pin<Box<F>>` est `Unpin` même si ce `F` n'est pas le cas, donc l'application `Box::pin` à une fonction asynchrone ou à un futur de bloc vous donne un futur que vous pouvez utiliser n'importe où, au prix d'une allocation de tas.

Il existe diverses méthodes non sécurisées pour travailler avec `Pin` qui vous permettent de faire ce que vous voulez avec le pointeur et sa cible, même pour les types de cible qui ne sont pas `Unpin`. Mais comme expliqué au [chapitre 22](#), Rust ne peut pas vérifier que ces méthodes sont utilisées correctement ; vous devenez responsable d'assurer la sécurité du code qui les utilise.

Quand le code asynchrone est-il utile ?

Asynchrone le code est plus délicat à écrire que le code multithread. Vous devez utiliser les bonnes primitives d'E/S et de synchronisation, décomposer manuellement les calculs de longue durée ou les répartir sur d'autres threads, et gérer d'autres détails comme l'épinglage qui n'apparaissent pas dans le code threadé. Alors, quels avantages spécifiques le code asynchrone offre-t-il ?

Deux affirmations que vous entendrez souvent ne résistent pas à une inspection minutieuse :

- "Le code asynchrone est idéal pour les E/S." Ce n'est pas tout à fait exact. Si votre application passe son temps à attendre des E/S, la rendre asynchrone ne fera pas que ces E/S s'exécutent plus rapidement. Il n'y a rien dans les interfaces d'E/S asynchrones généralement

utilisées aujourd'hui qui les rende plus efficaces que leurs homologues synchrones. Le système d'exploitation a le même travail à faire dans les deux cas. (En fait, une opération d'E/S asynchrone qui n'est pas prête doit être réessayée plus tard, il faut donc deux appels système pour se terminer au lieu d'un.)

- "Le code asynchrone est plus facile à écrire que le code multithread." Dans des langages comme JavaScript et Python, cela pourrait bien être vrai. Dans ces langages, les programmeurs utilisent `async/await` comme une forme de concurrence bien comportée : il y a un seul thread d'exécution, et les interruptions ne se produisent qu'au niveau `await` des expressions, donc il n'y a souvent pas besoin d'un mutex pour garder les données cohérentes : n'attendez pas pendant que vous êtes en train de l'utiliser ! Il est beaucoup plus facile de comprendre votre code lorsque les changements de tâche ne se produisent qu'avec votre autorisation explicite.

Mais cet argument ne s'applique pas à Rust, où les threads ne sont pas aussi gênants. Une fois votre programme compilé, il est exempt de courses de données. Le comportement non déterministe se limite aux fonctionnalités de synchronisation telles que les mutex, les canaux, les atomes, etc., qui ont été conçues pour y faire face. Ainsi, le code asynchrone n'a pas d'avantage unique pour vous aider à voir quand d'autres threads pourraient vous affecter ; c'est clair dans *tout* code Rust sécurisé.

Et bien sûr, le support asynchrone de Rust brille vraiment lorsqu'il est utilisé en combinaison avec des threads. Ce serait dommage d'y renoncer.

Alors, quels sont les réels avantages du code asynchrone ?

- *Les tâches asynchrones peuvent utiliser moins de mémoire.* Sous Linux, l'utilisation de la mémoire d'un thread commence à 20 Kio, en comptant à la fois l'espace utilisateur et l'espace noyau.² Les contrats à terme peuvent être beaucoup plus petits : les contrats à terme de notre serveur de discussion ont une taille de quelques centaines d'octets et sont devenus plus petits à mesure que le compilateur Rust s'améliore.
- *Les tâches asynchrones sont plus rapides à créer.* Sous Linux, la création d'un thread prend environ 15 µs. Générer une tâche asynchrone prend environ 300 ns, soit environ un cinquantième du temps.
- *Les changements de contexte sont plus rapides entre les tâches asynchrones qu'entre les threads du système d'exploitation*, 0,2 µs contre 1,7 µs sous Linux.³ Cependant, il s'agit de chiffres dans le meilleur des cas pour chacun : si le commutateur est dû à la disponibilité des E/S, les deux coûts augmentent à 1,7 µs. Que le basculement se fasse entre des threads ou des tâches sur différents coeurs de processeur fait égale-

ment une grande différence : la communication entre les cœurs est très lente.

Cela nous donne un indice sur les types de problèmes que le code asynchrone peut résoudre. Par exemple, un serveur asynchrone peut utiliser moins de mémoire par tâche et être ainsi capable de gérer plus de connexions simultanées. (C'est probablement là que le code asynchrone tire sa réputation d'être "bon pour les E/S".) les changements de contexte rapides sont tous des avantages importants. C'est pourquoi les serveurs de chat sont l'exemple classique de la programmation asynchrone, mais les jeux multi-joueurs et les routeurs réseau seraient probablement aussi de bons usages.

Dans d'autres situations, les arguments en faveur de l'utilisation de l'asynchrone sont moins clairs. Si votre programme dispose d'un pool de threads effectuant des calculs lourds ou restant inactifs en attendant la fin des E/S, les avantages énumérés précédemment n'ont probablement pas une grande influence sur ses performances. Vous devrez optimiser votre calcul, trouver une connexion Internet plus rapide ou faire autre chose qui affecte réellement le facteur limitant.

En pratique, chaque récit de mise en œuvre de serveurs à volume élevé que nous avons pu trouver mettait l'accent sur l'importance de la mesure, du réglage et d'une campagne incessante pour identifier et supprimer les sources de conflit entre les tâches. Une architecture asynchrone ne vous laissera pas ignorer tout ce travail. En fait, bien qu'il existe de nombreux outils prêts à l'emploi pour évaluer le comportement des programmes multithreads, les tâches asynchrones Rust sont invisibles pour ces outils et nécessitent donc leurs propres outils.. (Comme un ancien sage a dit un jour : « Maintenant, vous avez *deux* problèmes. »)

Même si vous n'utilisez pas de code asynchrone maintenant, il est bon de savoir que l'option est là si jamais vous avez la chance d'être beaucoup plus occupé que vous ne l'êtes maintenant.

- ¹ Si vous avez réellement besoin d'un client HTTP, envisagez d'utiliser l'un des nombreux excellents crates comme `surf` ou `reqwest` qui fera le travail correctement et de manière asynchrone. Ce client parvient principalement à obtenir des redirections HTTPS.
- ² Cela inclut la mémoire du noyau et compte les pages physiques allouées pour le thread, et non les pages virtuelles qui n'ont pas encore été allouées. Les chiffres sont similaires sur macOS et Windows.

- 3** Les commutateurs de contexte Linux étaient également dans la plage de 0,2 µs, jusqu'à ce que le noyau soit contraint d'utiliser des techniques plus lentes en raison de failles de sécurité du processeur.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 21. Macros

Un cento (du latin pour "patchwork") est un poème composé entièrement de lignes citées d'un autre poète.

—Matt Madden

Rust prend en charge les *macros*, un moyen d'étendre le langage d'une manière qui va au-delà de ce que vous pouvez faire avec des fonctions seules. Par exemple, nous avons vu la `assert_eq!` macro, ce qui est pratique pour les tests :

```
assert_eq!(gcd(6, 10), 2);
```

Cela aurait pu être écrit comme une fonction générique, mais la `assert_eq!` macro fait plusieurs choses que les fonctions ne peuvent pas faire. La première est que lorsqu'une assertion échoue, `assert_eq!` génère un message d'erreur contenant le nom de fichier et le numéro de ligne de l'assertion. Les fonctions n'ont aucun moyen d'obtenir ces informations. Les macros le peuvent, car leur mode de fonctionnement est complètement différent.

Les macros sont une sorte de raccourci. Lors de la compilation, avant que les types ne soient vérifiés et bien avant qu'un code machine ne soit généré, chaque appel de macro est *développé*, c'est-à-dire qu'il est remplacé par du code Rust. L'appel de macro précédent se développe à peu près comme ceci :

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `{} == {}`",
                  left_val, right_val);
        }
    }
}
```

`panic!` est aussi une macro, qui lui-même s'étend à encore plus de code Rust (non montré ici). Ce code utilise deux autres macros, `file!()` et `line!()`. Une fois que chaque appel de macro dans la caisse est complètement développé, Rust passe à la phase suivante de compilation.

Au moment de l'exécution, un échec d'assertion ressemblerait à ceci (et indiquerait un bogue dans la `gcd()` fonction, puisque 2 c'est la bonne

réponse) :

```
thread 'main' panicked at 'assertion failed: `!(left == right)`', (left: `17`,  
right: `2`)', gcd.rs:7
```

Si tu viens à partir de C++, vous avez peut-être eu de mauvaises expériences avec les macros. Les macros Rust adoptent une approche différente, similaire à celle de Scheme `syntax-rules`. Par rapport aux macros C++, les macros Rust sont mieux intégrées au reste du langage et donc moins sujettes aux erreurs. Les appels de macro sont toujours marqués d'un point d'exclamation, de sorte qu'ils se démarquent lorsque vous lisez du code et qu'ils ne peuvent pas être appelés accidentellement lorsque vous voulez appeler une fonction. Les macros Rust n'insèrent jamais de crochets ou de parenthèses sans correspondance. Et les macros Rust sont livrées avec une correspondance de modèle, ce qui facilite l'écriture de macros à la fois maintenables et attrayantes à utiliser.

Dans ce chapitre, nous montrerons comment écrire des macros à l'aide de plusieurs exemples simples. Mais comme beaucoup de Rust, les macros récompensent une compréhension approfondie, nous allons donc parcourir la conception d'une macro plus compliquée qui nous permet d'intégrer des littéraux JSON directement dans nos programmes. Mais il y a plus de macros que ce que nous pouvons couvrir dans ce livre, nous terminerons donc avec quelques conseils pour une étude plus approfondie, à la fois des techniques avancées pour les outils que nous vous avons montrés ici, et pour une fonction encore plus puissante appelée *macros procédurales*.

Principes de base des macros

[La figure 21-1](#) montre une partie du code source de la `assert_eq!` macro.

`macro_rules!` est le principal moyen de définir des macros dans Rust. Notez qu'il n'y a pas `!` de après `assert_eq` dans cette définition de macro : le `!` n'est inclus que lors de l'appel d'une macro, pas lors de sa définition.

Toutes les macros ne sont pas définies de cette façon : quelques-unes, comme `file!`, `line!`, et elle- `macro_rules!` même, sont intégrées au compilateur, et nous parlerons d'une autre approche, appelée macros procédurales, à la fin de ce chapitre. Mais pour la plupart, nous nous concentrerons sur `macro_rules!`, qui est (jusqu'à présent) le moyen le plus simple d'écrire le vôtre.

Une macro définie avec `macro_rules!` fonctionne entièrement par correspondance de modèle. Le corps d'une macro n'est qu'une série de règles :

```
( modèle1 ) => ( modèle1 );
```

```
( modèle2 ) => ( modèle2 );
```

```
...
```

```
macro_rules! assert_eq {  
    ($left:expr , $right:expr) => ({  
        match (&$left, &$right) {  
            (left_val, right_val) => {  
                if !(*left_val == *right_val) {  
                    panic!("assertion failed: `({} == {})`\n(left: `{:?}` , right: `{:?}`)",  
                          left_val, right_val)  
                }  
            }  
        }  
    });  
}
```

pattern

template

Illustration 21-1. La `assert_eq!` macro

La version de `assert_eq!` de la [Figure 21-1](#) n'a qu'un motif et un modèle.

Incidentement, vous pouvez utiliser des crochets ou des accolades au lieu de parenthèses autour du motif ou du modèle ; cela ne fait aucune différence pour Rust. De même, lorsque vous appelez une macro, elles sont toutes équivalentes :

```
assert_eq!(gcd(6, 10), 2);  
assert_eq![gcd(6, 10), 2];  
assert_eq!{gcd(6, 10), 2}
```

La seule différence est que les points-virgules sont généralement facultatifs après les accolades. Par convention, nous utilisons des parenthèses pour appeler `assert_eq!`, des crochets pour `vec!` et des accolades pour `macro_rules!`.

Maintenant que nous avons montré un exemple simple d'expansion d'une macro et la définition qui l'a générée, nous pouvons entrer dans les détails nécessaires pour que cela fonctionne :

- Nous expliquerons exactement comment Rust s'y prend pour trouver et étendre les définitions de macros dans votre programme.

- Nous soulignerons certaines subtilités inhérentes au processus de génération de code à partir de modèles de macro.
- Enfin, nous montrerons comment les modèles gèrent la structure répétitive.

Principes de base de l'extension de macro

Rouillerdéveloppe les macros très tôt lors de la compilation. Le compilateur lit votre code source du début à la fin, définissant et développant les macros au fur et à mesure. Vous ne pouvez pas appeler une macro avant qu'elle ne soit définie, car Rust développe chaque appel de macro avant même de regarder le reste du programme. (En revanche, les fonctions et autres [éléments](#) n'ont pas besoin d'être dans un ordre particulier. Il est normal d'appeler une fonction qui ne sera définie que plus tard dans la caisse.)

Lorsque Rust développe un `assert_eq!` appel de macro, ce qui se passe ressemble beaucoup à l'évaluation d'une `match` expression. Rust compare d'abord les arguments avec le motif, comme le montre la [figure 21-2](#)

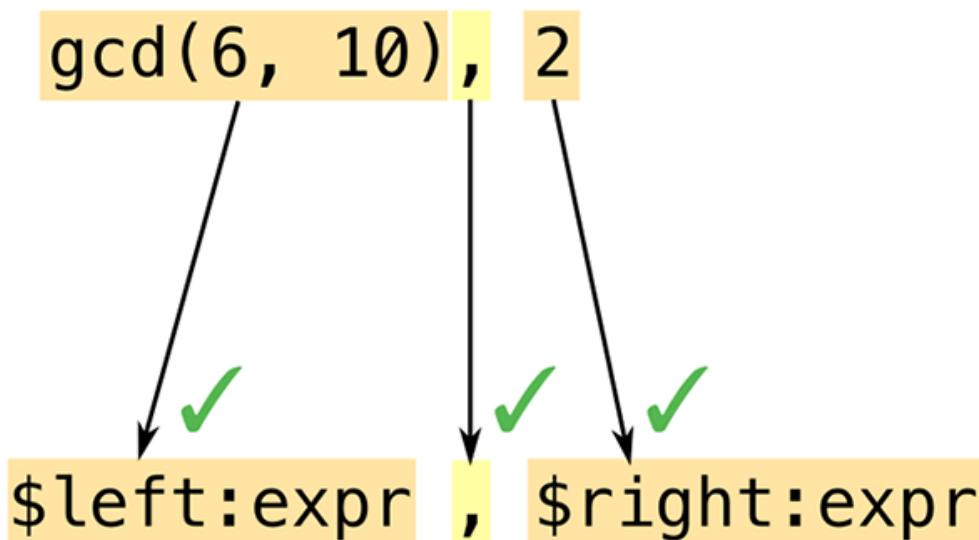


Illustration 21-2. Développer une macro, partie 1 : faire correspondre les arguments aux motifs

Les modèles de macro sont un mini-langage dans Rust. Ce sont essentiellement des expressions régulières pour le code correspondant. Mais là où les expressions régulières opèrent sur des caractères, les modèles opèrent sur des *jetons*— les chiffres, les noms, les signes de ponctuation, etc. qui sont les éléments constitutifs des programmes Rust. Cela signifie que vous pouvez utiliser librement les commentaires et les espaces blancs dans les modèles de macro pour les rendre aussi lisibles que possible. Les commentaires et les espaces ne sont pas des jetons, ils n'affectent donc pas la correspondance.

Une autre différence importante entre les expressions régulières et les modèles de macro est que les parenthèses, les crochets et les accolades apparaissent toujours par paires dans Rust. Ceci est vérifié avant que les macros ne soient développées, non seulement dans les modèles de macro mais dans tout le langage.

Dans cet exemple, notre modèle contient le fragment `$left:expr`, qui indique à Rust de faire correspondre une expression (dans ce cas, `gcd(6, 10)`) et de lui attribuer le nom `$left`. Rust fait alors correspondre la virgule du motif avec la virgule qui suit `gcd` les arguments de . Tout comme les expressions régulières, les modèles n'ont que quelques caractères spéciaux qui déclenchent un comportement de correspondance intéressant ; tout le reste, comme cette virgule, doit correspondre textuellement, sinon la correspondance échoue. Enfin, Rust correspond à l'expression `2` et lui donne le nom `$right`.

Les deux fragments de code de ce modèle sont de type `expr` : ils attendent des expressions. Nous verrons d'autres types de fragments de code dans ["Types de fragments"](#).

Étant donné que ce modèle correspond à tous les arguments, Rust développe le correspondant *modèle* ([Figure 21-3](#)).

```
{           replace with gcd(6, 10)
  ↓          ↓
  match (&$left, &$right) { replace with 2
    (left_val, right_val) => {
      if !(*left_val == *right_val) {
        panic!("assertion failed: `left == right` \
                (left: `{:?}` , right: `{:?}` ), \
                left_val, right_val)
      }
    }
}
```

Illustration 21-3. Développer une macro, partie 2 : remplir le modèle

Rust remplace `$left` et `$right` par les fragments de code trouvés lors de la correspondance.

C'est une erreur courante d'inclure le type de fragment dans le modèle de sortie : écrire `$left:expr` plutôt que simplement `$left`. Rust ne détecte pas immédiatement ce genre d'erreur. Il considère `$left` comme une substitution, puis il traite `:expr` comme tout le reste du modèle : les jetons à inclure dans la sortie de la macro. Ainsi, les erreurs ne se produiront pas tant que vous n'aurez pas *appelé* la macro, alors il générera une fausse sortie qui ne sera pas compilée. Si vous obtenez des messages d'erreur comme `cannot find type `expr` in this scope` et `help: maybe you meant to use a path separator here` lors de l'utilisation de la macro, vérifiez que vous n'avez pas oublié de supprimer le type de fragment.

tion d'une nouvelle macro, vérifiez-la pour cette erreur. ("[Debugging Macros](#)" offre des conseils plus généraux pour des situations comme celle-ci.)

Les modèles de macros ne sont pas très différents des douze langages de modèles couramment utilisés dans la programmation Web. La seule différence - et elle est significative - est que la sortie est du code Rust.

Conséquences inattendues

Bouchage des fragments de code dans des modèles est subtilement différent du code normal qui fonctionne avec des valeurs. Ces différences ne sont pas toujours évidentes au premier abord. La macro que nous avons examinée, `assert_eq!`, contient des morceaux de code un peu étranges pour des raisons qui en disent long sur la programmation de macros. Regardons deux morceaux amusants en particulier.

Tout d'abord, pourquoi cette macro crée-t-elle les variables `left_val` et `right_val`? Y a-t-il une raison pour laquelle nous ne pouvons pas simplifier le modèle pour qu'il ressemble à ceci ?

```
if !($left == $right) {  
    panic!("assertion failed: `($left == $right)`\n"  
          "left: `{:?}`, right: `{:?}`)", $left, $right)  
}
```

Pour répondre à cette question, essayez de développer mentalement l'appel de macro `assert_eq!(letters.pop(), Some('z'))`. Quelle serait la sortie ? Naturellement, Rust intégrerait les expressions correspondantes dans le modèle à plusieurs endroits. Cela semble être une mauvaise idée d'évaluer à nouveau les expressions lors de la construction du message d'erreur, et pas seulement parce que cela prendrait deux fois plus de temps : puisque `letters.pop()` supprime une valeur d'un vecteur, cela produira une valeur différente la deuxième fois nous l'appelons! C'est pourquoi la vraie macro calcule `$left` et `$right` une seule fois et stocke leurs valeurs.

Passons à la deuxième question : pourquoi cette macro emprunte-t-elle des références aux valeurs de `$left` et `$right`? Pourquoi ne pas simplement stocker les valeurs dans des variables, comme ceci ?

```
macro_rules! bad_assert_eq {  
    ($left: expr, $right:expr) => ({  
        match ($left, $right) {  
            ($left_val, $right_val) => {  
                if !(left_val == right_val) {  
                    panic!("assertion failed" /* ... */);  
                }  
            }  
        }  
    })  
}
```

```
    }  
  }  
});  
}
```

Pour le cas particulier que nous avons envisagé, où les arguments de la macro sont des entiers, cela fonctionnerait bien. Mais si l'appelant passait, disons, une `String` variable comme `$left` ou `$right`, ce code déplacerait la valeur hors de la variable !

```
fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s); // error: use of moved value "s"
}
```

Puisque nous ne voulons pas que les assertions déplacent les valeurs, la macro emprunte des références à la place.

(Vous vous êtes peut-être demandé pourquoi la macro utilise `match` plutôt que `let` de définir les variables. Nous nous sommes également posé la question. Il s'avère qu'il n'y a pas de raison particulière à cela. Cela `let` aurait été équivalent.)

En bref, les macros peuvent faire des choses surprenantes. Si des choses étranges se produisent autour d'une macro que vous avez écrite, il y a fort à parier que la macro est à blâmer.

Un bogue que vous ne verrez *pas* est ce bogue de macro C++ classique :

```
// buggy C++ macro to add 1 to a number  
#define ADD ONE(n)  n + 1
```

Pour des raisons connues pour la plupart des programmeurs C++, et cela ne vaut pas la peine d'être expliqué en détail ici, un code banal comme `ADD_ONE(1) * 10` ou `ADD_ONE(1 << 4)` produit des résultats très surprenants avec cette macro. Pour résoudre ce problème, vous ajouteriez plus de parenthèses à la définition de la macro. Ce n'est pas nécessaire dans Rust, car les macros Rust sont mieux intégrées au langage. Rust sait quand il gère les expressions, donc il ajoute efficacement des parenthèses chaque fois qu'il colle une expression dans une autre.

Répétition

La `vec!` macro standard se présente sous deux formes :

```

// Repeat a value N times
let buffer = vec![0_u8; 1000];

// A list of values, separated by commas
let numbers = vec!["udon", "ramen", "soba"];

```

Il peut être implémenté comme ceci :

```

macro_rules! vec {
    ($elem: expr ; $n: expr) => {
        :: std::vec:: from_elem($elem, $n)
    };
    ($($x: expr),*) => {
        <[_]>:: into_vec(Box:: new([$($x),*]))
    };
    ($($x:expr),+,) => {
        vec![ $($x),* ]
    };
}

```

Il y a trois règles ici. Nous expliquerons le fonctionnement de plusieurs règles, puis examinerons chaque règle à tour de rôle.

Lorsque Rust développe un appel de macro comme `vec![1, 2, 3]`, il commence par essayer de faire correspondre les arguments `1, 2, 3` avec le modèle de la première règle, dans ce cas `$elem:expr ; $n:expr`. Cela ne correspond pas : `1` est une expression, mais le modèle nécessite un point-virgule après cela, et nous n'en avons pas. Alors Rust passe ensuite à la deuxième règle, et ainsi de suite. Si aucune règle ne correspond, c'est une erreur.

La première règle gère les utilisations comme `vec![0u8; 1000]`. Il se trouve qu'il existe une fonction standard (mais non documentée) `std::vec::from_elem`, qui fait exactement ce qui est nécessaire ici, donc cette règle est simple.

La deuxième règle gère `vec!["udon", "ramen", "soba"]`. Le motif, `($x:expr),*`, utilise une fonctionnalité inédite : la répétition. Il correspond à 0 ou plusieurs expressions, séparées par des virgules. Plus généralement, la syntaxe `($PATTERN),*` est utilisée pour faire correspondre n'importe quelle liste séparée par des virgules, où chaque élément de la liste correspond à `PATTERN`.

Le `*` ici a la même signification que dans les expressions régulières ("0 ou plus") bien que les regexps n'aient pas de `,*` répéteur spécial. Vous pouvez également utiliser `+` pour exiger au moins une correspondance, ou `?`

pour zéro ou une correspondance. [Le tableau 21-1](#) donne la suite complète des modèles de répétition.

Tableau 21-1. Motifs de répétition

Motif	Sens
<code>\$(...)</code>	Correspondance 0 fois ou plus sans séparateur
<code>* ()</code>	
<code>\$(...),*</code>	Match 0 ou plusieurs fois, séparés par des virgules
<code>) ; *</code>	Correspondance 0 fois ou plus, séparées par des points-virgules
<code>\$(...)+</code>	Match 1 ou plusieurs fois sans séparateur
<code>+ ()</code>	
<code>\$(...),+</code>	Match 1 ou plusieurs fois, séparés par des virgules
<code>) ; +</code>	Match 1 ou plusieurs fois, séparés par des points-virgules
<code>\$(...)?</code>	Match 0 ou 1 fois sans séparateur
<code>? ()</code>	
<code>\$(...),?</code>	Match 0 ou 1 fois, séparés par des virgules
<code>) ; ?</code>	Match 0 ou 1 fois, séparés par des points-virgules

Le fragment de code `$x` n'est pas simplement une expression unique mais une liste d'expressions. Le modèle de cette règle utilise également la syntaxe de répétition :

```
<[_]>:: into_vec(Box::new([ $( $x ),* ]))
```

Encore une fois, il existe des méthodes standard qui font exactement ce dont nous avons besoin. Ce code crée un tableau encadré, puis utilise la `[T]::into_vec` méthode pour convertir le tableau encadré en vecteur.

Le premier bit, `<[_]>`, est une façon inhabituelle d'écrire le type "tranche de quelque chose", tout en s'attendant à ce que Rust en déduise le type d'élément. Les types dont les noms sont des identificateurs simples peuvent être utilisés dans des expressions sans problème, mais des types tels que `fn()`, `&str` ou `[_]` doivent être entourés de crochets.

La répétition arrive à la fin du modèle, où nous avons `$($x),*`. C'est `$(...),*` la même syntaxe que nous avons vue dans le modèle. Il parcourt la liste des expressions que nous avons mises en correspondance `$x` et les insère toutes dans le modèle, séparées par des virgules.

Dans ce cas, la sortie répétée ressemble à l'entrée. Mais cela ne doit pas être le cas. On aurait pu écrire la règle comme ceci :

```
( $( $x: expr ),* ) => {
    {
        let mut v = Vec::new();
        $($x.push($x)); *
        v
    }
};
```

Ici, la partie du modèle qui lit `$($x.push($x)); *` insère un appel à `v.push()` pour chaque expression dans `$x`. Un bras de macro peut se développer en une séquence d'expressions, mais ici nous n'avons besoin que d'une seule expression, nous enveloppons donc l'assemblage du vecteur dans un bloc.

Contrairement au reste de Rust, les modèles utilisant `$(...),*` ne prennent pas automatiquement en charge une virgule finale facultative. Cependant, il existe une astuce standard pour prendre en charge les virgules de fin en ajoutant une règle supplémentaire. C'est ce que fait la troisième règle de notre `vec!` macro :

```
( $( $x:expr ),+ ,) => { // if trailing comma is present,
    vec![ $($x ),* ] // retry without it
};
```

Nous utilisons `$(...),+ ,` pour faire correspondre une liste avec une virgule supplémentaire. Ensuite, dans le modèle, nous appelons `vec!` de manière récursive, en omettant la virgule supplémentaire. Cette fois, la deuxième règle correspondra.

Macros intégrées

La rouillecompiler fournit plusieurs macros utiles lorsque vous définissez vos propres macros. Aucun de ceux-ci ne pourrait être mis en œuvre en utilisant `macro_rules!` seul. Ils sont codés en dur dans `rustc` :

`file!(), line!(), column!()`

`file!()` se développe à un littéral de chaîne : le nom de fichier actuel. `line!()` et `column!()` développer en `u32` littéraux donnant la ligne et la colonne actuelles (en partant de 1).

Si une macro en appelle une autre, qui en appelle une autre, toutes dans des fichiers différents, et que la dernière macro appelle `file!()`, `line!()` ou `column!()`, elle se développera pour indiquer l'emplacement du *premier* appel de macro.

`stringify!(...tokens...)`

Se développe à un littéral de chaîne contenant les jetons donnés. La `assert!` macro l'utilise pour générer un message d'erreur qui inclut le code de l'assertion.

Les appels de macro dans l'argument ne sont *pas* développés :

`stringify!(line!())` se développe jusqu'à la chaîne "line!()".

Rust construit la chaîne à partir des jetons, il n'y a donc pas de saut de ligne ni de commentaire dans la chaîne.

`concat!(str0, str1, ...)`

Se développe à un littéral de chaîne unique créé en concaténant ses arguments.

Rust définit également ces macros pour interroger l'environnement de construction :

`cfg!(...)`

Se développe à une constante booléenne, `true` si la configuration de construction actuelle correspond à la condition entre parenthèses. Par exemple, `cfg!(debug_assertions)` est vrai si vous compilez avec les assertions de débogage activées.

Cette macro prend en charge exactement la même syntaxe que l'`# [cfg(...)]` attribut décrit dans "[Attributs](#)" mais au lieu d'une compilation conditionnelle, vous obtenez une réponse vraie ou fausse.

`env!("VAR_NAME")`

Se développe à une chaîne : la valeur de la variable d'environnement spécifiée au moment de la compilation. Si la variable n'existe pas, c'est une erreur de compilation.

Cela ne servirait à rien si ce n'est que Cargo définit plusieurs variables d'environnement intéressantes lorsqu'il compile une caisse. Par exemple, pour obtenir la chaîne de version actuelle de votre crate, vous pouvez écrire :

```
let version = env!("CARGO_PKG_VERSION");
```

Une liste complète de ces variables d'environnement est incluse dans la [documentation Cargo](#).

`option_env!("VAR_NAME")`

Cette est identique à `env!` sauf qu'il renvoie un `Option<&'static str>` c'est-à-dire `None` si la variable spécifiée n'est pas définie.

Trois autres macros intégrées vous permettent d'importer du code ou des données à partir d'un autre fichier :

`include!("file.rs")`

Se développe au contenu du fichier spécifié, qui doit être un code Rust valide, soit une expression, soit une séquence d'[éléments](#).

`include_str!("file.txt")`

Se développe à `&'static str` contenant le texte du fichier spécifié. Vous pouvez l'utiliser comme ceci :

```
const COMPOSITOR_SHADER:&str =
    include_str!("../resources/compositor.glsl");
```

Si le fichier n'existe pas ou n'est pas valide en UTF-8, vous obtiendrez une erreur de compilation.

`include_bytes!("file.dat")`

Cette est le même sauf que le fichier est traité comme des données binaires, et non comme du texte UTF-8. Le résultat est un `&'static [u8]`.

Comme toutes les macros, celles-ci sont traitées au moment de la compilation. Si le fichier n'existe pas ou ne peut pas être lu, la compilation échoue. Ils ne peuvent pas échouer au moment de l'exécution. Dans tous les cas, si le nom de fichier est un chemin relatif, il est résolu par rapport au répertoire qui contient le fichier courant.

Rust fournit également plusieurs macros pratiques que nous n'avons pas abordées précédemment :

```
todo!(), unimplemented!()
```

Cessent équivalents à `panic!()`, mais transmettent une intention différente. `unimplemented!()` va dans `if` les clauses, `match` les armes et autres cas qui ne sont pas encore traités. Ça panique toujours. `todo!()` est à peu près la même chose, mais transmet l'idée que ce code n'a tout simplement pas encore été écrit ; certains IDE le signalent pour avis.

```
matches!(value, pattern)
```

Compare une valeur à un modèle, et retourne `true` si elle correspond, ou `false` autrement. C'est équivalent à écrire :

```
match value {
    pattern => true,
    _ => false
}
```

Si vous cherchez un exercice d'écriture de macros de base, c'est une bonne macro à répliquer, d'autant plus que l'implémentation réelle, que vous pouvez voir dans la documentation de la bibliothèque standard, est assez simple.

Macros de débogage

Débogage une macro capricieuse peut être difficile. Le plus gros problème est le manque de visibilité sur le processus d'expansion macro. Rust développera souvent toutes les macros, trouvera une sorte d'erreur, puis imprimera un message d'erreur qui n'affichera pas le code entièrement développé contenant l'erreur !

Voici trois outils pour aider à dépanner les macros. (Ces fonctionnalités sont toutes instables, mais comme elles sont vraiment conçues pour être utilisées pendant le développement, pas dans le code que vous voudriez enregistrer, ce n'est pas un gros problème en pratique.)

Tout d'abord et le plus simple, vous pouvez demander `rustc` de montrer à quoi ressemble votre code après avoir développé toutes les macros. Utilisez `cargo build --verbose` pour voir comment Cargo appelle `rustc`. Copiez la `rustc` ligne de commande et ajoutez- `-Z unstable-options --pretty expanded` la en tant qu'options. Le code entièrement développé est vidé sur votre terminal. Malheureusement, cela ne fonctionne que si votre code est exempt d'erreurs de syntaxe.

Deuxièmement, Rust fournit une `log_syntax!()` macro qui affiche simplement ses arguments sur le terminal au moment de la compilation. Vous pouvez l'utiliser pour le `println!` débogage de style. Cette macro nécessite l'`#![feature(log_syntax)]` indicateur de fonctionnalité.

Troisièmement, vous pouvez demander au compilateur Rust de consigner tous les appels de macro au terminal. Insérer `trace_macros!(true);` quelque part dans votre code. À partir de ce moment, chaque fois que Rust développe une macro, il imprime le nom et les arguments de la macro. Par exemple, considérez ce programme :

```
#![feature(trace_macros)]  
  
fn main() {  
    trace_macros!(true);  
    let numbers = vec![1, 2, 3];  
    trace_macros!(false);  
    println!("total: {}", numbers.iter().sum::<u64>());  
}
```

Il produit cette sortie :

```
$rustup override set nightly  
...  
$rustc trace_example.rs  
note: trace_macro  
--> trace_example.rs:5:19  
|  
5 |     let numbers = vec![1, 2, 3];  
|          ^^^^^^  
|  
|= note: expanding `vec! { 1 , 2 , 3 }`  
= note: to `< [ _ ] > :: into_vec ( box [ 1 , 2 , 3 ] )`
```

Le compilateur affiche le code de chaque appel de macro, à la fois avant et après l'expansion. La ligne `trace_macros!(false);` désactive à nouveau le traçage, de sorte que l'appel à `println!()` n'est pas tracé.

Construire le json! Macro

Nous avons maintenant discutées principales fonctionnalités de `macro_rules!`. Dans cette section, nous allons développer progressivement une macro pour créer des données JSON. Nous allons utiliser cet exemple pour montrer à quoi ressemble le développement d'une macro, présenter les quelques éléments restants de `macro_rules!` et offrir

quelques conseils sur la façon de s'assurer que vos macros se comportent comme vous le souhaitez.

De retour au [chapitre 10](#), nous avons présenté cette énumération pour représenter les données JSON :

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

La syntaxe d'écriture des `Json` valeurs est malheureusement assez verbeuse :

```
let students = Json:: Array(vec![
    Json:: Object(Box:: new(vec![
        ("name".to_string(), Json:: String("Jim Blandy".to_string())),
        ("class_of".to_string(), Json:: Number(1926.0)),
        ("major".to_string(), Json:: String("Tibetan throat singing".to_string()))
    ].into_iter().collect())),
    Json:: Object(Box:: new(vec![
        ("name".to_string(), Json:: String("Jason Orendorff".to_string())),
        ("class_of".to_string(), Json:: Number(1702.0)),
        ("major".to_string(), Json:: String("Knots".to_string()))
    ].into_iter().collect()))
]);

```

Nous aimerions pouvoir écrire ceci en utilisant une syntaxe plus proche de JSON :

```
let students = json!([
    {
        "name": "Jim Blandy",
        "class_of": 1926,
        "major": "Tibetan throat singing"
    },
    {
        "name": "Jason Orendorff",
        "class_of": 1702,
        "major": "Knots"
    }
]);
```

Ce que nous voulons, c'est une `json!` macro qui prend une valeur JSON comme argument et se développe en une expression Rust comme celle de l'exemple précédent.

Types de fragments

La premièreLe travail d'écriture d'une macro complexe consiste à déterminer comment faire correspondre ou *analyser* l'entrée souhaitée.

Nous pouvons déjà voir que la macro aura plusieurs règles, car il existe différentes sortes de choses dans les données JSON : objets, tableaux, nombres, etc. En fait, nous pourrions supposer que nous aurons une règle pour chaque type JSON :

```
macro_règles ! json {
    (null) => { Json :: Null } ;
    ([ ... ]) => { Json::Array(...) } ;
    ({ ... }) => { Json::Object(...) } ;
    (????) => { Json::Booléen(...) } ;
    (????) => { Json::Number(...) } ;
    (????) => { Json::String(...) } ;
}
```

Ce n'est pas tout à fait correct, car les macro-modèles n'offrent aucun moyen de distinguer les trois derniers cas, mais nous verrons comment traiter cela plus tard. Les trois premiers cas, au moins, commencent clairement avec des jetons différents, alors commençons par ceux-là.

La première règle fonctionne déjà :

```
macro_rules! json {
    (null) => {
        Json::Null
    }

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null); // passes!
}
```

Pour ajouter la prise en charge des tableaux JSON, nous pourrions essayer de faire correspondre les éléments en tant que `expr`s :

```
macro_rules! json {
    (null) => {
        Json:: Null
```

```

};

([ $( $element: expr ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};

}

```

Malheureusement, cela ne correspond pas à tous les tableaux JSON. Voici un test qui illustre le problème :

```

#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // valid JSON that doesn't match `$( $element:expr )` 
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ].into_iter().collect()))
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}

```

Le motif `$($element:expr),*` signifie "une liste d'expressions Rust séparées par des virgules". Mais de nombreuses valeurs JSON, en particulier des objets, ne sont pas des expressions Rust valides. Ils ne correspondent pas.

Étant donné que tous les bits de code que vous souhaitez faire correspondre ne sont pas des expressions, Rust prend en charge plusieurs autres types de fragments., répertoriés dans [le Tableau 21-2](#) .

Tableau 21-2. Types de fragments pris en charge par `macro_rules!`

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
<code>expr</code>	Une expression: <code>2 + 2, "udon", x.len()</code>	<code>=> , ;</code>
<code>stmt</code>	Une expression ou une déclaration, n'incluant aucun point-virgule final (difficile à utiliser ; essayez <code>expr</code> ou à la <code>block</code> place)	<code>=> , ;</code>
<code>ty</code>	Un type: <code>String, Vec<u8>, (&str, bool), dyn Read + Send</code>	<code>=> , ; = { [: > as where</code>
<code>path</code>	Un chemin (discuté): <code>ferns, ::std::sync::mpsc</code>	<code>=> , ; = { [: > as where</code>
<code>pat</code>	Un modèle (discuté): <code>_ , Some(ref x)</code>	<code>=> , = if in</code>
<code>item</code>	Un élément (discuté): <code>struct Point { x: f64, y: f64 } , mod ferns;</code>	N'importe quoi
<code>block</code>	Un bloc (discuté): <code>{ s += "ok\n"; true }</code>	N'importe quoi
<code>meta</code>	Le corps d'un attribut (discuté): <code>inline, derive(Copy, Clone), doc="3D models."</code>	N'importe quoi
<code>literal</code>	Une valeur littérale : <code>1024, "Hello, world!", 1_000_000f64</code>	N'importe quoi
<code>lifetim e</code>	Une durée de vie: <code>'a, 'item, 'static</code>	N'importe quoi
<code>vis</code>	Un spécificateur de visibilité : <code>pub, pub(crate), pub(in module::submodule)</code>	N'importe quoi

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
ident	Un identifiant : std, Json, longish_variable_name	N'importe quoi
tt	Un arbre à jetons (voir texte) : ; , >=, {}, [0 1 (+ 0 1)]	N'importe quoi

La plupart des options de ce tableau appliquent strictement la syntaxe Rust. Le `expr` type correspond uniquement aux expressions Rust (et non aux valeurs JSON), `ty` correspond uniquement aux types Rust, etc. Ils ne sont pas extensibles : il n'y a aucun moyen de définir de nouveaux opérateurs arithmétiques ou de nouveaux mots-clés qui `expr` les reconnaîtraient. Nous ne pourrons pas faire en sorte qu'aucune de ces données corresponde à des données JSON arbitraires.

Les deux derniers, `ident` et `tt`, prennent en charge les arguments de macro correspondants qui ne ressemblent pas au code Rust. `ident` correspond à n'importe quel identifiant. `tt` correspond à un seul *arbre à jetons*: soit une paire de parenthèses correctement appariées, `(...)`, `[...]`, ou `{ ... }`, et tout ce qui se trouve entre les deux, y compris les arbres de jetons imbriqués, soit un seul jeton qui n'est pas une parenthèse, comme `1926` ou `"Knots"`.

Les arbres à jetons sont exactement ce dont nous avons besoin pour notre `json!` macro. Chaque valeur JSON est une arborescence de jetons unique : les nombres, les chaînes, les valeurs booléennes et `null` sont tous des jetons uniques ; les objets et les tableaux sont entre crochets. On peut donc écrire les motifs comme ceci :

```
macro_rules! json {
    (null) => {
        Json:: Null
    };
    ([ $( $element: tt ),* ]) => {
        Json:: Array(...)
    };
    ({ $($key: tt : $value: tt ),* }) => {
        Json:: Object(...)
    };
    ($other:tt) => {
        ... // TODO: Return Number, String, or Boolean
    };
}
```

Cette version de la `json!` macro peut correspondre à toutes les données JSON. Il ne nous reste plus qu'à produire le code Rust correct.

Pour s'assurer que Rust puisse acquérir de nouvelles fonctionnalités syntaxiques à l'avenir sans casser les macros que vous écrivez aujourd'hui, Rust restreint les jetons qui apparaissent dans les modèles juste après un fragment. La colonne "Peut être suivi de..." du [Tableau 21-2](#) indique les jetons autorisés. Par exemple, le modèle `$x:expr ~ $y:expr` est une erreur, car `~` n'est pas autorisé après un `expr`. Le modèle `$vars:pat => $handler:expr` est correct, car `$vars:pat` est suivi de la flèche `=>`, l'un des jetons autorisés pour un `pat`, et `$handler:expr` est suivi de rien, ce qui est toujours autorisé.

Récurivité dans les macros

Vous avez déjà vu un cas trivial d'une macro s'appelant elle-même : notre implémentation de `vec!` utilise la récursivité pour prendre en charge les virgules de fin. Ici, nous pouvons montrer un exemple plus significatif : les `json!` besoينss'appeler récursivement.

Nous pourrions essayer de prendre en charge les tableaux JSON sans utiliser la récursivité, comme ceci :

```
([ $( $element: tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};
```

Mais cela ne fonctionnerait pas. Nous collerions des données JSON (les `$element` arbres de jetons) directement dans une expression Rust. Ce sont deux langues différentes.

Nous devons convertir chaque élément du tableau du formulaire JSON en Rust. Heureusement, il existe une macro qui fait cela : celle que nous écrivons !

```
([ $( $element: tt ),* ]) => {
    Json::Array(vec![ $( json!($element) ),* ])
};
```

Les objets peuvent être pris en charge de la même manière :

```
({ $( $key: tt : $value: tt ),* }) => {
    Json::Object(Box::new(vec![
        $( ($key.to_string(), json!($value)) ),*
    ].into_iter().collect())))
};
```

Le compilateur impose une limite de récursivité aux macros : 64 appels, par défaut. C'est plus que suffisant pour les utilisations normales de `json!`, mais les macros récursives complexes atteignent parfois la limite. Vous pouvez l'ajuster en ajoutant cet attribut en haut de la caisse où la macro est utilisée :

```
#![recursion_limit = "256"]
```

Notre `json!` macro est presque terminée. Il ne reste plus qu'à prendre en charge les valeurs booléennes, numériques et de chaîne.

Utiliser des caractéristiques avec des macros

Complexes d'écriture les macros posent toujours des énigmes. Il est important de se rappeler que les macros elles-mêmes ne sont pas le seul outil de résolution d'éénigmes à votre disposition.

Ici, nous devons prendre en charge `json!(true)`, `json!(1.0)` et `json!("yes")`, en convertissant la valeur, quelle qu'elle soit, en le type de `Json` valeur approprié. Mais les macros ne sont pas bonnes pour distinguer les types. On peut imaginer écrire :

```
macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
    ...
}
```

Cette approche tombe en panne tout de suite. Il n'y a que deux valeurs booléennes, mais plutôt plus de nombres que cela, et encore plus de chaînes.

Heureusement, il existe un moyen standard de convertir des valeurs de différents types en un type spécifié : le `From` trait, couvert. Nous devons simplement implémenter ce trait pour quelques types :

```
impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}
```

```

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
...

```

En fait, les 12 types numériques devraient avoir des implémentations très similaires, il peut donc être judicieux d'écrire une macro, juste pour éviter le copier-coller :

```

macro_rules! impl_from_num_for_json {
    ( $( $t: ident )* ) => {
        $($(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
        )*)
    };
}

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
                       usize isize f32 f64);

```

Nous pouvons maintenant utiliser `Json::from(value)` pour convertir `value` n'importe quel type pris en charge en `Json`. Dans notre macro, cela ressemblera à ceci :

```

( $other: tt ) => {
    Json::from($other) // Handle Boolean/number/string
};

```

L'ajout de cette règle à notre `json!` macro lui permet de passer tous les tests que nous avons écrits jusqu'à présent. En rassemblant toutes les

pièces, cela ressemble actuellement à ceci :

```
macro_rules! json {
    (null) => {
        Json:: Null
    };
    ([ $( $element: tt ),* ]) => {
        Json:: Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key: tt : $value: tt ),* }) => {
        Json:: Object(Box:: new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect()))
    };
    ( $other: tt ) => {
        Json::from($other) // Handle Boolean/number/string
    };
}
```

Il s'avère que la macro prend en charge de manière inattendue l'utilisation de variables et même d'expressions Rust arbitraires dans les données JSON, une fonctionnalité supplémentaire pratique :

```
let width = 4.0;
let desc =
    json!({
        "width": width,
        "height":(width * 9.0 / 4.0)
   });
```

Parce `(width * 9.0 / 4.0)` qu'il est entre parenthèses, il s'agit d'un arbre à jetons unique, de sorte que la macro le correspond avec succès `$value:tt` lors de l'analyse de l'objet.

Cadrage et hygiène

Un étonnammentL'aspect délicat de l'écriture de macros est qu'elles impliquent de coller ensemble du code provenant de différentes étendues. Les pages suivantes couvrent donc les deux manières dont Rust gère la portée : une manière pour les variables et les arguments locaux, et une autre pour tout le reste.

Pour montrer pourquoi cela est important, réécrivons notre règle d'analyse des objets JSON (la troisième règle de la `json!` macro présentée précédemment) pour éliminer le vecteur temporaire. Nous pouvons l'écrire comme ceci :

```

({ $($key: tt : $value: tt),* }) => {
{
    let mut fields = Box:: new(HashMap:: new());
    $($ fields.insert($key.to_string(), json!($value)); )*
    Json::Object(fields)
}
};


```

Maintenant, nous remplissons le `HashMap` pas en utilisant `collect()` mais en appelant à plusieurs reprises la `.insert()` méthode. Cela signifie que nous devons stocker la carte dans une variable temporaire, que nous avons appelée `fields`.

Mais alors que se passe-t-il si le code qui appelle `json!` utilise une variable qui lui est propre, également nommée `fields` ?

```

let fields = "Fields, W.C.";
let role = json!({
    "name": "Larson E. Whipsnade",
    "actor": fields
});

```

L'expansion de la macro collerait ensemble deux morceaux de code, les deux utilisant le nom `fields` pour des choses différentes !

```

let fields = "Fields, W.C.";
let role = {
    let mut fields = Box:: new(HashMap:: new());
    fields.insert("name".to_string(), Json:: from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json:: from(fields));
    Json::Object(fields)
};

```

Cela peut sembler être un piège inévitable chaque fois que les macros utilisent des variables temporaires, et vous réfléchissez peut-être déjà aux correctifs possibles. Peut-être devrions-nous renommer la variable `json!` définie par la macro en quelque chose que ses appelants ne sont pas susceptibles de transmettre : au lieu de `fields`, nous pourrions l'appeler `__json$fields`.

La surprisevoici que *la macro fonctionne telle quelle*. Rust renomme la variable pour vous ! Cette fonctionnalité, implémentée pour la première fois dans les macros Scheme, est appelée *hygiène*, et on dit donc que Rust a *des macros hygiéniques*.

La façon la plus simple de comprendre l'hygiène des macros est d'imaginer qu'à chaque fois qu'une macro est agrandie, les parties de l'expansion qui proviennent de la macro elle-même sont peintes d'une couleur différente.

Les variables de couleurs différentes sont alors traitées comme si elles avaient des noms différents :

```
let fields = "Champs, WC" ;
laisser rôle = {
    let champs mut = Box::new(HashMap::new());
    fields.insert( "nom" .to_string(), Json::from( "Larson E. Whipsnade" ) );
    fields.insert( "actor" .to_string(), Json::from( fields ) );
    Json::Objet(champs)
} ;
```

Notez que les morceaux de code transmis par l'appelant de la macro et collés dans la sortie, tels que "name" et "actor", conservent leur couleur d'origine (noir). Seuls les jetons provenant du modèle de macro sont dessinés.

Maintenant, il y a une variable nommée `fields` (déclarée dans l'appelant) et une variable distincte nommée `fields` (introduite par la macro). Comme les noms sont de couleurs différentes, les deux variables ne se confondent pas.

Si une macro a vraiment besoin de faire référence à une variable dans la portée de l'appelant, l'appelant doit transmettre le nom de la variable à la macro.

(La métaphore de la peinture n'est pas censée être une description exacte du fonctionnement de l'hygiène. Le véritable mécanisme est même un peu plus intelligent que cela, reconnaissant deux identifiants comme identiques, indépendamment de la « peinture », s'ils se réfèrent à une variable commune qui est à la fois pour la macro et son appelant. Mais des cas comme celui-ci sont rares dans Rust. Si vous comprenez l'exemple précédent, vous en savez assez pour utiliser des macros hygiéniques.)

Vous avez peut-être remarqué que de nombreux autres identificateurs étaient peints d'une ou plusieurs couleurs au fur et à mesure que les macros étaient développées : `Box`, `HashMap` et `Json`, par exemple. Malgré la peinture, Rust n'a eu aucun mal à reconnaître ces noms de types. C'est parce que l'hygiène dans Rust est limitée aux variables et arguments locaux. En ce qui concerne les constantes, les types, les méthodes, les modules, les statiques et les noms de macros, Rust est "daltonien".

Cela signifie que si notre `json!` macro est utilisée dans un module où `Box`, `HashMap` ou `Json` n'est pas dans la portée, la macro ne fonctionnera pas. Nous montrerons comment éviter ce problème dans la section suivante.

Tout d'abord, nous allons considérer un cas où l'hygiène stricte de Rust gêne, et nous devons le contourner. Supposons que nous ayons plusieurs fonctions contenant cette ligne de code :

```
let req = ServerRequest::new(server_socket.session());
```

Copier et coller cette ligne est une douleur. Pouvons-nous utiliser une macro à la place ?

```
macro_rules! setup_req {
    () => {
        let req = ServerRequest::new(server_socket.session());
    }
}

fn handle_http_request(server_socket:&ServerSocket) {
    setup_req!(); // declares `req`, uses `server_socket`
    ... // code that uses `req`
}
```

Comme écrit, cela ne fonctionne pas. Il faudrait que le nom `server_socket` dans la macro fasse référence au local `server_socket` déclaré dans la fonction, et vice versa pour la variable `req`. Mais l'hygiène empêche les noms dans les macros de "entrer en collision" avec des noms dans d'autres portées, même dans des cas comme celui-ci, où c'est ce que vous voulez.

La solution consiste à transmettre à la macro tous les identifiants que vous prévoyez d'utiliser à l'intérieur et à l'extérieur du code de la macro :

```
macro_rules! setup_req {
    ($req: ident, $server_socket: ident) => {
        let $req = ServerRequest::new($server_socket.session());
    }
}

fn handle_http_request(server_socket:&ServerSocket) {
    setup_req!(req, server_socket);
    ... // code that uses `req`
}
```

Puisque `req` et `server_socket` sont maintenant fournis par la fonction, ils sont la bonne "couleur" pour cette portée.

L'hygiène rend cette macro un peu plus longue à utiliser, mais c'est une fonctionnalité, pas un bogue : il est plus facile de raisonner sur les macros hygiéniques en sachant qu'elles ne peuvent pas jouer avec les variables locales derrière votre dos. Si vous recherchez un identifiant comme `server_socket` dans une fonction, vous trouverez tous les endroits où il est utilisé, y compris les appels de macro.

Importation et exportation de macros

Depuis les macrossont développés au début de la compilation, avant que Rust ne connaisse la structure complète des modules de votre projet, le compilateur dispose de moyens spéciaux pour les exporter et les importer.

Les macros visibles dans un module sont automatiquement visibles dans ses modules enfants. Pour exporter des macros d'un module "vers le haut" vers son module parent, utilisez l'`#[macro_use]` attribut. Par exemple, supposons que notre `lib.rs` ressemble à ceci :

```
#[macro_use] mod macros;
mod client;
mod server;
```

Toutes les macros définies dans le `macros` module sont importées dans `lib.rs` et donc visibles dans le reste du crate, y compris dans `client` et `server`.

Les macros marquées d'un `#[macro_export]` sont automatiquement pub et peuvent être référencées par chemin, comme les autres éléments.

Par exemple, le `lazy_static` crate fournit une macro appelée `lazy_static`, qui est marquée par `#[macro_export]`. Pour utiliser cette macro dans votre propre caisse, vous écririez :

```
use lazy_static::lazy_static;
lazy_static!{ }
```

Une fois qu'une macro est importée, elle peut être utilisée comme n'importe quel autre élément :

```
use lazy_static::lazy_static;
```

```

mod m {
    crate::lazy_static!{ }
}

```

Bien sûr, faire l'une de ces choses signifie que votre macro peut être appelée dans d'autres modules. Une macro exportée ne devrait donc pas dépendre de quoi que ce soit dans la portée - on ne sait pas ce qui sera dans la portée où elle est utilisée. Même les caractéristiques du prélude standard peuvent être masquées.

Au lieu de cela, la macro doit utiliser des chemins absous vers tous les noms qu'elle utilise. `macro_rules!` fournit le fragment `$crate` pour aider à cela. Ce n'est pas la même chose que `crate`, qui est un mot-clé qui peut être utilisé n'importe où dans les chemins, pas seulement dans les macros. `$crate` agit comme un chemin absolu vers le module racine du crate où la macro a été définie. Au lieu de dire `Json`, nous pouvons écrire `$crate::Json`, ce qui fonctionne même s'il `Json` n'a pas été importé. `HashMap` peut être remplacé par

```
::std::collections::HashMap ou $crate::macros::HashMap.
```

Dans ce dernier cas, nous devrons réexporter `HashMap`, car `$crate` ne peuvent pas être utilisés pour accéder aux fonctionnalités privées d'une caisse. Cela s'étend vraiment à quelque chose comme `::jsonlib`, un chemin ordinaire. Les règles de visibilité ne sont pas affectées.

Après avoir déplacé la macro vers son propre module `macros` et l'avoir modifiée pour utiliser `$crate`, elle ressemble à ceci. C'est la version finale:

```

// macros.rs
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $( $element: tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $($key: tt : $value: tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new(
                $crate::macros::HashMap::new());
            $( fields.insert($crate::macros::ToString::to_string($key),
                json!($value)); )
        }
    }
}

```

```

    )*
    $crate:: : :: :: Json_Object(fields)
}
};

($other_tt) => {
    $crateJsonfrom($other)
};

}

```

Étant donné que la `.to_string()` méthode fait partie du `Tostring` trait standard, nous l'utilisons `$crate` également pour nous y référer, en utilisant la syntaxe que nous avons introduite dans [« Appels de méthode entièrement qualifiés »](#):

`$crate::macros::ToString::to_string($key)`. Dans notre cas, ce n'est pas strictement nécessaire pour faire fonctionner la macro, car `ToString` c'est dans le prélude standard. Mais si vous appelez des méthodes d'un trait qui n'est peut-être pas dans la portée au moment où la macro est appelée, un appel de méthode entièrement qualifié est la meilleure façon de le faire.

Éviter les erreurs de syntaxe lors de la correspondance

La macro suivante semble raisonnable, mais cela pose quelques problèmes à Rust :

```

macro_rules! complain {
    ($msg: expr) => {
        println!("Complaint filed: {}", $msg)
    };
    ($user : $userid: tt , $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg)
    };
}

```

Supposons que nous l'appelions ainsi :

```
complain!(user:"jimb", "the AI lab's chatbots keep picking on me");
```

Aux yeux de l'homme, cela correspond évidemment au deuxième modèle. Mais Rust essaie d'abord la première règle, en essayant de faire correspondre toutes les entrées avec `$msg:expr`. C'est là que les choses commencent à mal tourner pour nous. `user: "jimb"` n'est pas une expression, bien sûr, donc nous obtenons une erreur de syntaxe. Rust refuse de balayer une erreur de syntaxe sous le tapis - les macros sont déjà assez

difficiles à déboguer. Au lieu de cela, il est signalé immédiatement et la compilation s'arrête.

Si un autre jeton d'un modèle ne correspond pas, Rust passe à la règle suivante. Seules les erreurs de syntaxe sont fatales, et elles ne se produisent que lorsque vous essayez de faire correspondre des fragments.

Le problème ici n'est pas si difficile à comprendre : nous essayons de faire correspondre un fragment, `$msg:expr`, dans la mauvaise règle. Ça ne va pas correspondre parce que nous ne sommes même pas censés être ici. L'appelant voulait l'autre règle. Il existe deux façons simples d'éviter cela.

Tout d'abord, évitez les règles confuses. Nous pourrions, par exemple, modifier la macro afin que chaque motif commence par un identifiant différent :

```
macro_rules! complain {
    ($msg : $msg: expr) => {
        println!("Complaint filed: {}", $msg);
    };
    ($user : $userid: tt, $msg : $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

Lorsque les arguments de la macro commencent par `msg`, nous obtenons la règle 1. Lorsqu'ils commencent par `user`, nous obtenons la règle 2. Dans tous les cas, nous savons que nous avons la bonne règle avant d'essayer de faire correspondre un fragment.

L'autre façon d'éviter les fausses erreurs de syntaxe consiste à mettre en place des règles plus spécifiques en premier. Placer la `user:` règle en premier résout le problème avec `complain!`, car la règle qui provoque l'erreur de syntaxe n'est jamais atteinte.

Au-delà des macro_règles !

Les modèles de macro peuvent analyser des entrées encore plus complexes que JSON, mais nous avons constaté que la complexité devient rapidement incontrôlable.

[The Little Book of Rust Macros](#), de Daniel Keep et al., est un excellent manuel de `macro_rules!` programmation avancée. Le livre est clair et intelligent, et il décrit chaque aspect de l'expansion macro plus en détail que nous l'avons ici. Il présente également plusieurs techniques très astucieuses pour mettre `macro_rules!` des modèles en service comme une

sorte de langage de programmation ésotérique, pour analyser des entrées complexes. Nous sommes moins enthousiastes. Utiliser avec précaution.

Rust 1.15 a introduit un mécanisme distinct appelé *macros procédurales*. Les macros procédurales prennent en charge l'extension de l' `#[derive]` attribut pour gérer les dérivations personnalisées, comme illustré à la [Figure 21-4](#), ainsi que la création d'attributs personnalisés et de nouvelles macros qui sont appelées comme les `macro_rules!` macros décrites précédemment.

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```

custom derive

Image 21-4. invoquer une `IntoJson` macro procédurale hypothétique via un `#[derive]` attribut

Il n'y a pas de `IntoJson` trait, mais cela n'a pas d'importance : une macro procédurale peut utiliser ce crochet pour insérer le code qu'elle souhaite (dans ce cas, probablement `impl From<Money> for Json { ... }`).

Ce qui rend une macro procédurale "procédurale", c'est qu'elle est implémentée comme une fonction Rust, et non comme un ensemble de règles déclaratives. Cette fonction interagit avec le compilateur à travers une fine couche d'abstraction et peut être arbitrairement complexe. Par exemple, la `diesel` bibliothèque de base de données utilise des macros procédurales pour se connecter à une base de données et générer du code basé sur le schéma de cette base de données au moment de la compilation.

Étant donné que les macros procédurales interagissent avec les composants internes du compilateur, l'écriture de macros efficaces nécessite une compréhension du fonctionnement du compilateur qui n'entre pas dans le cadre de ce livre. Il est cependant largement couvert dans la [documentation en ligne](#).

Après avoir lu tout cela, vous avez peut-être décidé que vous détestez les macros. Quoi alors ? Une alternative consiste à générer du code Rust à l'aide d'un script de construction. La [documentation Cargo](#) montre comment procéder étape par étape. Cela implique d'écrire un programme qui génère le code Rust que vous voulez, d'ajouter une ligne à `Cargo.toml` pour exécuter ce programme dans le cadre du processus de construction et d'utiliser `include!` pour obtenir le code généré dans votre caisse.

Chapitre 22. Code dangereux

*Que personne ne pense de moi que je suis humble ou faible ou passif;
Qu'ils comprennent que je suis d'un genre différent :
dangereux pour mes ennemis, loyal envers mes amis.
A une telle vie appartient la gloire.*

—Euripide, *Médée*

Le secretLa joie de la programmation système est que, sous chaque langage sûr et chaque abstraction soigneusement conçue, se trouve un maelström tourbillonnant de langage machine extrêmement dangereux et de petits bricolages. Vous pouvez aussi écrire cela dans Rust.

Le langage que nous avons présenté jusqu'à présent dans le livre garantit que vos programmes sont exempts d'erreurs de mémoire et de courses de données de manière entièrement automatique, via des types, des durées de vie, des vérifications de limites, etc. Mais ce type de raisonnement automatisé a ses limites ; il existe de nombreuses techniques précieuses que Rust ne peut pas reconnaître comme sûres.

Le code non sécurisé vous permet de dire à Rust : "Je choisis d'utiliser des fonctionnalités dont vous ne pouvez pas garantir la sécurité". En marquant un bloc ou une fonction comme non sécurisé, vous acquérez la capacité d'appeler `unsafe` des fonctions dans la bibliothèque standard, de déréférencer des pointeurs non sécurisés et d'appeler des fonctions écrites dans d'autres langages comme C et C++, entre autres pouvoirs. Les autres vérifications de sécurité de Rust s'appliquent toujours : les vérifications de type, les vérifications de durée de vie et les vérifications de limites sur les index se produisent toutes normalement. Le code non sécurisé active simplement un petit ensemble de fonctionnalités supplémentaires.

Cette capacité à sortir des limites de Rust sûr est ce qui permet d'implémenter bon nombre des fonctionnalités les plus fondamentales de Rust dans Rust lui-même, tout comme C et C++ sont utilisés pour implémenter leurs propres bibliothèques standard. Le code non sécurisé est ce qui permet au `Vec` type de gérer efficacement son tampon ; le `std::io` module pour parler au système d'exploitation ; et les modules `std::thread` et pour fournir des primitives `std::sync` de concurrence .

Ce chapitre couvre les éléments essentiels de l'utilisation de fonctions non sécurisées :

- `unsafe` Blocs de rouilleétablir la frontière entre le code Rust ordinaire et sûr et le code qui utilise des fonctionnalités non sûres.
- Vous pouvez marquer des fonctions comme `unsafe`, alertant les appelleurs de la présence de contrats supplémentaires qu'ils doivent suivre pour éviter un comportement indéfini.
- Pointeurs brutset leurs méthodes permettent un accès illimité à la mémoire et vous permettent de créer des structures de données que le système de type de Rust interdirait autrement. Alors que les références de Rust sont sûres mais contraintes, les pointeurs bruts, comme tout programmeur C ou C++ le sait, sont un outil puissant et pointu.
- Comprendre la définition d'un comportement indéfini vous aidera à comprendre pourquoi il peut avoir des conséquences bien plus graves que le simple fait d'obtenir des résultats incorrects.
- Les traits non sûrs, analogues aux `unsafe` fonctions, imposent un contrat que chaque implémentation (plutôt que chaque appelant) doit suivre.

Pas à l'abri de quoi ?

Au début de ce livre, nous avons montré un programme C qui plante de manière surprenante parce qu'il ne respecte pas l'une des règles prescrites par le standard C. Vous pouvez faire la même chose dans Rust :

```
$cat crash.rs
fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}
$cargo build
Compiling unsafe-samples v0.1.0
Finished debug [unoptimized + debuginfo] target(s) in 0.44s
$../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

Ce programme emprunte une référence mutable à la variable locale `a`, la convertit en un pointeur brut de type `*mut usize`, puis utilise la `offset` méthode pour produire un pointeur trois mots plus loin dans la mémoire. Il se trouve que c'est là `main` que l'adresse de retour de est stockée. Le programme remplace l'adresse de retour par une constante, de sorte que le retour de `main` se comporte de manière surprenante. Ce qui rend ce plantage possible, c'est l'utilisation incorrecte par le programme de fonctionnalités non sécurisées, dans ce cas, la possibilité de déréférencer les pointeurs bruts.

Une fonctionnalité non sécurisée est une fonctionnalité qui impose un *contrat*: règles que Rust ne peut pas appliquer automatiquement, mais que vous devez néanmoins suivre pour éviter *un comportement indéfini*.

Un contrat va au-delà des contrôles de type habituels et des contrôles de durée de vie, imposant des règles supplémentaires spécifiques à cette fonctionnalité dangereuse. Typiquement, Rust lui-même n'est pas du tout au courant du contrat ; c'est juste expliqué dans la documentation de la fonctionnalité. Par exemple, le type pointeur brut a un contrat vous interdisant de déréférer un pointeur qui a été avancé au-delà de la fin de son référent d'origine. L'expression `*ptr.offset(3) = ...` dans cet exemple rompt ce contrat. Mais, comme le montre la transcription, Rust compile le programme sans se plaindre : ses contrôles de sécurité ne détectent pas cette violation. Lorsque vous utilisez des fonctionnalités non sécurisées, vous, en tant que programmeur, êtes responsable de vérifier que votre code respecte leurs contrats.

De nombreuses fonctionnalités ont des règles que vous devez suivre pour les utiliser correctement, mais ces règles ne sont pas des contrats au sens où nous l'entendons ici, à moins que les conséquences possibles incluent un comportement indéfini. Indéfinile comportement est le comportement Rust suppose fermement que votre code ne pourrait jamais s'afficher. Par exemple, Rust suppose que vous n'écraserez pas l'adresse de retour d'un appel de fonction avec quelque chose d'autre. Un code qui passe les contrôles de sécurité habituels de Rust et est conforme aux contrats des fonctionnalités dangereuses qu'il utilise ne peut pas faire une telle chose. Étant donné que le programme viole le contrat de pointeur brut, son comportement n'est pas défini et il déraille.

Si votre code présente un comportement indéfini, vous avez rompu votre moitié de votre marché avec Rust, et Rust refuse de prédire les conséquences. Extraire des messages d'erreur non pertinents des profondeurs

des bibliothèques système et planter est une conséquence possible ; donner le contrôle de votre ordinateur à un attaquant en est une autre. Les effets peuvent varier d'une version de Rust à l'autre, sans avertissement. Parfois, cependant, un comportement indéfini n'a pas de conséquences visibles. Par exemple, si la `main` fonction ne revient jamais (peut-être appelle-t-elle `std::process::exit` pour terminer le programme plus tôt), alors l'adresse de retour corrompue n'aura probablement pas d'importance.

Vous ne pouvez utiliser que des fonctionnalités non sécurisées dans un `unsafe` bloc ou une `unsafe` fonction ; nous expliquerons les deux dans les sections qui suivent. Cela rend plus difficile l'utilisation de fonctionnalités dangereuses sans le savoir : en vous forçant à écrire un `unsafe` bloc ou une fonction, Rust s'assure que vous avez reconnu que votre code peut avoir des règles supplémentaires à suivre..

Blocs dangereux

Un `unsafe` bloc ressemble à un bloc Rust ordinaire précédé du mot-`unsafe` clé, à la différence que vous pouvez utiliser des fonctionnalités non sécurisées dans le bloc :

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```

Sans le mot-`unsafe` clé devant le bloc, Rust s'opposerait à l'utilisation de `from_utf8_unchecked`, qui est une `unsafe` fonction. Avec le `unsafe` bloc qui l'entoure, vous pouvez utiliser ce code n'importe où.

Comme un bloc Rust ordinaire, la valeur d'un `unsafe` bloc est celle de son expression finale, ou `()` s'il n'en a pas. L'appel à `String::from_utf8_unchecked` indiqué précédemment fournit la valeur du bloc.

Un `unsafe` bloc déverrouille cinq options supplémentaires pour vous :

- Vous pouvez appeler `unsafe` des fonctions. Chaque `unsafe` fonction doit spécifier son propre contrat, en fonction de son objet.
- Vous pouvez déréférencer pointeurs bruts. Le code sécurisé peut transmettre des pointeurs bruts, les comparer et les créer par conversion à

partir de références (ou même d'entiers), mais seul le code non sécurisé peut réellement les utiliser pour accéder à la mémoire. Nous couvrirons les pointeurs bruts en détail et expliquerons comment les utiliser en toute sécurité dans "[Raw Pointers](#)" .

- Vous pouvez accéder aux champs de unions, dont le compilateur ne peut pas être sûr qu'ils contiennent des modèles de bits valides pour leurs types respectifs.
- Vous pouvez accéder static variables mutables . Comme expliqué dans "[Variables globales](#)" , Rust ne peut pas être sûr que les threads utilisent des static variables mutables, leur contrat vous oblige donc à vous assurer que tous les accès sont correctement synchronisés.
- Vous pouvez accéder aux fonctions et aux variables déclarées via l'étranger de Rust interface de fonction. Ceux-ci sont considérés unsafe même lorsqu'ils sont immuables, car ils sont visibles par du code écrit dans d'autres langages qui peuvent ne pas respecter les règles de sécurité de Rust.

Restreindre les fonctionnalités dangereuses aux unsafe blocs ne vous empêche pas vraiment de faire ce que vous voulez. Il est parfaitement possible de simplement coller un unsafe bloc dans votre code et de passer à autre chose. L'avantage de la règle réside principalement dans le fait d'attirer l'attention humaine sur du code dont Rust ne peut garantir la sécurité :

- Vous n'utiliserez pas accidentellement des fonctionnalités dangereuses et découvrirez ensuite que vous étiez responsable de contrats dont vous ignoriez même l'existence.
- Un unsafe bloc attire davantage l'attention des examinateurs. Certains projets ont même une automatisation pour garantir cela, en signalant les changements de code qui affectent les unsafe blocs pour une attention particulière.
- Lorsque vous envisagez d'écrire un unsafe bloc, vous pouvez prendre un moment pour vous demander si votre tâche nécessite vraiment de telles mesures. Si c'est pour les performances, avez-vous des mesures pour montrer qu'il s'agit en fait d'un goulot d'étranglement ? Peut-être existe-t-il un bon moyen d'accomplir la même chose dans Rust en toute sécurité.

Exemple : un type de chaîne ASCII efficace

Voici la définition de `Ascii`, un type de chaîne qui garantit que son contenu est toujours en ASCII valide. Ce type utilise une fonctionnalité non sécurisée pour fournir une conversion sans coût en `String` :

```
mod my_ascii {
    /// An ASCII-encoded string.
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // This must hold only well-formed ASCII text:
        // bytes from `0` to `0x7f`.
        Vec<u8>
    );

    impl Ascii {
        /// Create an `Ascii` from the ASCII text in `bytes`. Return a
        /// `NotAsciiError` error if `bytes` contains any non-ASCII
        /// characters.
        pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii, NotAsciiError>
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // When conversion fails, we give back the vector we couldn't convert
    // This should implement `std::error::Error`; omitted for brevity.
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // Safe, efficient conversion, implemented using unsafe code.
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // If this module has no bugs, this is safe, because
            // well-formed ASCII text is also well-formed UTF-8.
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }
    ...
}
```

La clé de ce module est la définition du `Ascii` type. Le type lui-même est marqué `pub`, pour le rendre visible en dehors du `my_ascii` module. Mais l' `Vec<u8>` élément du type n'est *pas* public, donc seul le `my_ascii` module peut construire une `Ascii` valeur ou faire référence à son élément. Cela laisse au code du module un contrôle total sur ce qui

peut ou non y apparaître. Tant que les constructeurs publics et les méthodes garantissent que les `Ascii` valeurs fraîchement créées sont bien formées et le restent tout au long de leur vie, alors le reste du programme ne peut pas violer cette règle. Et en effet, le constructeur public

`Ascii::from_bytes` vérifie soigneusement le vecteur qui lui est donné avant d'accepter de construire un `Ascii` à partir de cela. Par souci de brièveté, nous ne montrons aucune méthode, mais vous pouvez imaginer un ensemble de méthodes de gestion de texte qui garantissent que les `Ascii` valeurs contiennent toujours le texte ASCII approprié, tout comme `String` les méthodes de `a` garantissent que son contenu reste UTF-8 bien formé.

Cette disposition nous permet de mettre `From<Ascii>` en œuvre `String` très efficacement. La fonction `unsafe`

`String::from_utf8_unchecked` prend un vecteur d'octets et en construit un `String` sans vérifier si son contenu est du texte UTF-8 bien formé ; le contrat de la fonction tient son appelant responsable de cela. Heureusement, les règles imposées par le `Ascii` type sont exactement ce dont nous avons besoin pour satisfaire `from_utf8_unchecked` le contrat de `.Comme` nous l'avons expliqué dans "[UTF-8](#)", tout bloc de texte ASCII est également UTF-8 bien formé, de sorte que le sous-`Ascii` jacent d'`Vec<u8>` un est immédiatement prêt à servir de `String` tampon pour un

Avec ces définitions en place, vous pouvez écrire :

```
use my_ascii::Ascii;

let bytes:Vec<u8> = b"ASCII and ye shall receive".to_vec();

// This call entails no allocation or text copies, just a scan.
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // We know these chosen bytes are ok.

// This call is zero-cost: no allocation, copies, or scans.
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

Aucun `unsafe` bloc n'est requis pour utiliser `Ascii`. Nous avons implémenté une interface sécurisée utilisant des opérations non sécurisées et nous nous sommes arrangés pour respecter leurs contrats en fonction

uniquement du code propre au module, et non du comportement de ses utilisateurs.

Un `Ascii` n'est rien de plus qu'un wrapper autour d'un `Vec<u8>`, caché à l'intérieur d'un module qui applique des règles supplémentaires sur son contenu. Un type de ce type est appelé un *nouveau type*, un modèle courant dans Rust. Le propre type de Rust `String` est défini exactement de la même manière, sauf que son contenu est limité à UTF-8, et non ASCII. En fait, voici la définition de `String` de la bibliothèque standard :

```
pub struct String {  
    vec: Vec<u8>,  
}
```

Au niveau de la machine, avec les types de Rust hors de l'image, un nouveau type et son élément ont des représentations identiques en mémoire, donc la construction d'un nouveau type ne nécessite aucune instruction machine. Dans `Ascii::from_bytes`, l'expression `Ascii(bytes)` considère simplement que la `Vec<u8>` représentation de contient maintenant une `Ascii` valeur. De même, `String::from_utf8_unchecked` ne nécessite probablement aucune instruction machine lorsqu'il est en ligne : le `Vec<u8>` est maintenant considéré comme un fichier `String`.

Fonctions dangereuses

Une `unsafe` fonctionLa définition ressemble à une définition de fonction ordinaire précédée du mot- `unsafe` clé. Le corps d'une `unsafe` fonction est automatiquement considéré comme un `unsafe` bloc.

Vous ne pouvez appeler `unsafe` des fonctions qu'à l'intérieur de `unsafe` blocs. Cela signifie que le marquage d'une fonction `unsafe` avertit ses appelants que la fonction a un contrat qu'ils doivent satisfaire pour éviter un comportement indéfini.

Par exemple, voici un nouveau constructeur pour le `Ascii` type que nous avons introduit précédemment qui construit un `Ascii` à partir d'un vecteur d'octets sans vérifier si son contenu est en ASCII valide :

```
// This must be placed inside the `my_ascii` module.  
impl Ascii {  
    /// Construct an `Ascii` value from `bytes`, without checking
```

```

    /// whether `bytes` actually contains well-formed ASCII.
    ///
    /// This constructor is infallible, and returns an `Ascii` directly,
    /// rather than a `Result<Ascii, NotAsciiError>` as the `from_bytes`
    /// constructor does.
    ///
    /// # Safety
    ///
    /// The caller must ensure that `bytes` contains only ASCII
    /// characters: bytes no greater than 0x7f. Otherwise, the effect is
    /// undefined.
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}

```

Vraisemblablement, l'appel de code

`Ascii::from_bytes_unchecked` sait déjà d'une manière ou d'une autre que le vecteur en main ne contient que des caractères ASCII, de sorte que la vérification qui `Ascii::from_bytes` insiste sur l'exécution serait une perte de temps, et l'appelant devrait écrire du code pour gérer les `Err` résultats dont il sait qu'ils ne se produiront jamais.

`Ascii::from_bytes_unchecked` permet à un tel appelant de contourner les vérifications et la gestion des erreurs.

Mais plus haut, nous avons souligné l'importance des `Ascii` constructeurs publics de et des méthodes garantissant que `Ascii` les valeurs sont bien formées. Ne manque-t-il pas `from_bytes_unchecked` à cette responsabilité ?

Pas tout à fait : `from_bytes_unchecked` remplit ses obligations en les répercutant sur son appelant via son contrat. La présence de ce contrat est ce qui rend correct le marquage de cette fonction `unsafe` : malgré le fait que la fonction elle-même n'effectue aucune opération dangereuse, ses appelants doivent suivre des règles que Rust ne peut pas appliquer automatiquement pour éviter un comportement indéfini.

Pouvez-vous vraiment provoquer un comportement indéfini en rompant le contrat de `Ascii::from_bytes_unchecked` ? Oui. Vous pouvez construire un `String` holding UTF-8 mal formé comme suit :

```

// Imagine that this vector is the result of some complicated process
// that we expected to produce ASCII. Something went wrong!
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

```

```

let ascii = unsafe {
    // This unsafe function's contract is violated
    // when `bytes` holds non-ASCII bytes.
    Ascii::from_bytes_unchecked(bytes)
};

let bogus:String = ascii.into();

// `bogus` now holds ill-formed UTF-8. Parsing its first character produces
// a `char` that is not a valid Unicode code point. That's undefined
// behavior, so the language doesn't say how this assertion should behave.
assert_eq!(bogus.chars().next().unwrap() as u32, 0xffff);

```

Dans certaines versions de Rust, sur certaines plates-formes, cette affirmation a échoué avec le message d'erreur divertissant suivant :

```

thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `2097151`,
  right: `2097151`', src/main.rs:42:5

```

Ces deux nombres nous semblent égaux, mais ce n'est pas la faute de Rust ; c'est la faute du `unsafe` bloc précédent. Lorsque nous disons qu'un comportement indéfini conduit à des résultats imprévisibles, c'est le genre de chose que nous voulons dire.

Cela illustre deux faits critiques sur les bogues et code non sécurisé :

- *Les bugs qui se produisent avant le `unsafe` blocage peuvent rompre les contrats.* Le fait qu'un `unsafe` bloc provoque un comportement indéfini peut dépendre non seulement du code dans le bloc lui-même, mais également du code qui fournit les valeurs sur lesquelles il opère. Tout ce sur quoi votre `unsafe` code s'appuie pour satisfaire les contrats est critique pour la sécurité. La conversion de `Ascii` vers `String` basée sur `String::from_utf8_unchecked` n'est bien définie que si le reste du module maintient correctement `Ascii` les invariants de .
- *Les conséquences de la rupture d'un contrat peuvent apparaître après avoir quitté le `unsafe` bloc.* Le comportement indéfini courtisé par le non-respect du contrat d'une fonctionnalité dangereuse ne se produit souvent pas dans le `unsafe` bloc lui-même. Construire un faux `String` comme indiqué précédemment peut ne pas causer de problèmes jusqu'à bien plus tard dans l'exécution du programme.

Essentially, Rust's type checker, borrow checker, and other static checks are inspecting your program and trying to construct proof that it cannot exhibit undefined behavior. When Rust compiles your program successfully, that means it succeeded in proving your code sound. An `unsafe` block is a gap in this proof: “This code,” you are saying to Rust, “is fine, trust me.” Whether your claim is true could depend on any part of the program that influences what happens in the `unsafe` block, and the consequences of being wrong could appear anywhere influenced by the `unsafe` block. Writing the `unsafe` keyword amounts to a reminder that you are not getting the full benefit of the language’s safety checks.

Si vous avez le choix, vous devriez naturellement préférer créer des interfaces sécurisées, sans contrats. Ceux-ci sont beaucoup plus faciles à utiliser, car les utilisateurs peuvent compter sur les contrôles de sécurité de Rust pour s’assurer que leur code est exempt de comportement indéfini. Même si votre implémentation utilise des fonctionnalités non sécurisées, il est préférable d'utiliser les types, les durées de vie et le système de modules de Rust pour respecter leurs contrats tout en utilisant uniquement ce que vous pouvez vous garantir, plutôt que de confier des responsabilités à vos appelants.

Malheureusement, il n'est pas rare de rencontrer des fonctions dangereuses dans la nature dont la documentation ne prend pas la peine d'expliquer leurs contrats. Vous êtes censé déduire les règles vous-même, en fonction de votre expérience et de votre connaissance du comportement du code. Si vous vous êtes déjà demandé avec inquiétude si ce que vous faites avec une API C ou C++ est OK, alors vous savez ce que c'est.

Blocage non sécurisé ou fonction non sécurisée ?

Vous pouvez vous demander s'il faut utiliser un `unsafe` bloc ou marquez simplement toute la fonction comme non sécurisée. L'approche que nous recommandons est de prendre d'abord une décision concernant la fonction :

- S'il est possible d'abuser de la fonction d'une manière qui se compile correctement mais provoque toujours un comportement indéfini, vous devez la marquer comme non sécurisée. Les règles d'utilisation correcte de la fonction sont son contrat ; l'existence d'un contrat est ce qui rend la fonction dangereuse.

- Sinon, la fonction est sûre : aucun appel correctement typé ne peut provoquer un comportement indéfini. Il ne doit pas être marqué `unsafe`.

Que la fonction utilise des fonctionnalités dangereuses dans son corps n'est pas pertinent ; ce qui compte, c'est la présence d'un contrat. Auparavant, nous avons montré une fonction non sécurisée qui n'utilise aucune fonctionnalité non sécurisée et une fonction sécurisée qui utilise des fonctionnalités non sécurisées.

Ne marquez pas une fonction sûre `unsafe` simplement parce que vous utilisez des fonctionnalités dangereuses dans son corps. Cela rend la fonction plus difficile à utiliser et déroute les lecteurs qui s'attendront (correctement) à trouver un contrat expliqué quelque part. Utilisez plutôt un `unsafe` bloc, même s'il s'agit du corps entier de la fonction.

Comportement indéfini

Dans l'introduction, nous avons dit que le terme *comportement indéfini* signifie "comportement que Rust suppose fermement que votre code ne pourrait jamais présenter". C'est une tournure de phrase étrange, d'autant plus que nous savons par notre expérience avec d'autres langues que ces comportements *se* produisent par accident avec une certaine fréquence. Pourquoi ce concept est-il utile pour définir les obligations d'un code non sécurisé ?

Un compilateur est un traducteur d'un langage de programmation à un autre. Le compilateur Rust prend un programme Rust et le traduit en un programme équivalent en langage machine. Mais qu'est-ce que cela veut dire de dire que deux programmes dans des langages aussi complètement différents sont équivalents ?

Heureusement, cette question est plus facile pour les programmeurs que pour les linguistes. Nous disons généralement que deux programmes sont équivalents s'ils auront toujours le même comportement visible lors de leur exécution : ils effectuent les mêmes appels système, interagissent avec des bibliothèques étrangères de manière équivalente, etc. C'est un peu comme un test de Turing pour les programmes : si vous ne savez pas si vous interagissez avec l'original ou la traduction, alors ils sont équivalents.

Considérez maintenant le code suivant :

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

Même en ne sachant rien de la définition de `very_trustworthy`, nous pouvons voir qu'il ne reçoit qu'une référence partagée à `i`, donc l'appel ne peut pas changer `i` la valeur de . Puisque la valeur passée à `println!` sera toujours `1000`, Rust peut traduire ce code en langage machine comme si nous avions écrit :

```
very_trustworthy(&10);
println!("{}", 1000);
```

Cette version transformée a le même comportement visible que l'original, et elle est probablement un peu plus rapide. Mais il est logique de ne considérer les performances de cette version que si nous convenons qu'elle a la même signification que l'original. Et si `very_trustworthy` étaient définis comme suit ?

```
fn very_trustworthy(shared:&i32) {
    unsafe {
        // Turn the shared reference into a mutable pointer.
        // This is undefined behavior.
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

Ce code enfreint les règles des références partagées : il change la valeur de `i` en `20`, même s'il devrait être gelé car il `i` est emprunté pour le partage. Par conséquent, la transformation que nous avons faite à l'appelant a maintenant un effet très visible : si Rust transforme le code, le programme imprime `1000`; s'il laisse le code seul et utilise la nouvelle valeur de `i`, il imprime `2000`. Enfreindre les règles des références partagées dans `very_trustworthy` signifie que les références partagées ne se comporteront pas comme prévu dans ses appelants.

Ce genre de problème se pose avec presque tous les types de transformation que Rust pourrait tenter. Même l'intégration d'une fonction dans son site d'appel suppose, entre autres, que lorsque l'appelé a terminé, le flux de contrôle revient au site d'appel. Mais nous avons ouvert le chapitre avec un exemple de code mal élevé qui viole même cette hypothèse.

Il est fondamentalement impossible pour Rust (ou tout autre langage) d'évaluer si une transformation vers un programme conserve sa signification à moins qu'il ne puisse faire confiance aux caractéristiques fondamentales du langage pour qu'il se comporte comme prévu. Et qu'ils le fassent ou non peuvent dépendre non seulement du code à portée de main, mais aussi d'autres parties potentiellement éloignées du programme. Pour faire quoi que ce soit avec votre code, Rust doit supposer que le reste de votre programme se comporte bien.

Voici donc les règles de Rust pour les programmes bien élevés :

- Le programme ne doit pas lire la mémoire non initialisée.
- Le programme ne doit pas créer de valeurs primitives invalides :
 - Les références, cases ou `fn` pointeurs qui sont `null`
 - `bool` des valeurs qui ne sont ni `0` ni `1`
 - `enum` valeurs avec des valeurs discriminantes non valides
 - `char` valeurs non valides, points de code Unicode non substituts
 - `str` valeurs qui ne sont pas bien formées UTF-8
 - Pointeurs gras avec vtables/longueurs de tranches non valides
 - Toute valeur du type "jamais", écrite `!`, pour les fonctions qui ne retournent pas
- Les règles de références expliquées au [chapitre 5](#) doivent être respectées. Aucune référence ne peut survivre à son référent ; l'accès partagé est un accès en lecture seule ; et l'accès mutable est un accès exclusif.
- Le programme ne doit pas déréférencer les pointeurs nuls, incorrectement alignés ou pendants .
- Le programme ne doit pas utiliser un pointeur pour accéder à la mémoire en dehors de l'allocation à laquelle le pointeur est associé. Nous expliquerons cette règle en détail dans [« Déréférencer les pointeurs bruts en toute sécurité »](#).
- Le programme doit être exempt de courses aux données. Une course aux données se produit lorsque deux threads accèdent au même emplacement mémoire sans synchronisation et qu'au moins l'un des accès est une écriture.
- Le programme ne doit pas se dérouler sur un appel effectué depuis une autre langue, via l'interface de la fonction étrangère, comme expliqué dans ["Déroulement"](#).
- Le programme doit être conforme aux contrats des fonctions standard de la bibliothèque.

Comme nous ne disposons pas encore d'un modèle complet de la sémantique de Rust pour le `unsafe` code, cette liste évoluera probablement

avec le temps, mais ceux-ci resteront probablement interdits.

Toute violation de ces règles constitue un comportement indéfini et rend les efforts de Rust pour optimiser votre programme et le traduire en langage machine indignes de confiance. Si vous enfreignez la dernière règle et passez l'UTF-8 mal formé à `String::from_utf8_unchecked`, peut-être que 2097151 n'est pas si égal à 2097151 après tout.

Le code Rust qui n'utilise pas de fonctionnalités dangereuses est garanti de suivre toutes les règles précédentes, une fois qu'il est compilé (en supposant que le compilateur n'a pas de bogue ; nous y arrivons, mais la courbe ne croisera jamais l'asymptote). Ce n'est que lorsque vous utilisez des fonctionnalités non sécurisées que ces règles deviennent votre responsabilité.

En C et C++, le fait que votre programme se compile sans erreurs ni avertissements signifie beaucoup moins ; comme nous l'avons mentionné dans l'introduction de ce livre, même les meilleurs programmes C et C++ écrits par des projets très respectés qui maintiennent leur code à des normes élevées présentent un comportement indéfini dans la pratique.

Caractéristiques dangereuses

Un *trait dangereux* est un trait qui a un contrat Rust ne peut pas vérifier ou appliquer ce que les implémentateurs doivent faire pour éviter un comportement indéfini. Pour implémenter une caractéristique non sécurisée, vous devez marquer l'implémentation comme non sécurisée. C'est à vous de comprendre le contrat du trait et de vous assurer que votre type le satisfait.

Une fonction qui limite ses variables de type avec un trait non sécurisé est généralement une fonction qui utilise elle-même des fonctionnalités non sécurisées et ne satisfait leurs contrats qu'en fonction du contrat du trait non sécurisé. Une implémentation incorrecte du trait pourrait amener une telle fonction à présenter un comportement indéfini.

`std::marker::Send` et `std::marker::Sync` sont les exemples classiques de traits dangereux. Ces traits ne définissent aucune méthode, ils sont donc simples à implémenter pour n'importe quel type que vous aimez. Mais ils ont des contrats : `Send` exige que les implémentateurs soient en sécurité pour passer à un autre thread, et `Sync` exige qu'ils soient en sécurité pour partager entre les threads via des références partagées. La

mise en œuvre `Send` pour un type inapproprié, par exemple, ne ferait `std::sync::Mutex` plus à l'abri des courses de données.

À titre d'exemple simple, la bibliothèque standard Rust incluait un trait non sécurisé, `core::nonzero::Zeroable`, pour les types qui peuvent être initialisés en toute sécurité en mettant tous leurs octets à zéro. De toute évidence, la mise à zéro de `a.usize` est correcte, mais la mise à zéro de `a.T` vous donne une référence nulle, ce qui provoquera un plantage s'il est déréférencé. Pour les types qui étaient `Zeroable`, certaines optimisations étaient possibles : vous pouviez en initialiser un tableau rapidement avec `std::ptr::write_bytes` (l'équivalent de Rust de `memset`) ou utiliser des appels du système d'exploitation qui allouent des pages mises à zéro. (`Zeroable` était instable et déplacé vers une utilisation interne uniquement dans la `num` caisse de Rust 1.26, mais c'est un bon exemple simple et concret.)

`Zeroable` était un trait marqueur typique, dépourvu de méthodes ou de types associés :

```
pub unsafe trait Zeroable {}
```

Les implémentations pour les types appropriés étaient tout aussi simples :

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// and so on for all the integer types
```

Avec ces définitions, on pourrait écrire une fonction qui alloue rapidement un vecteur d'une longueur donnée contenant un `Zeroable` type :

```
use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{
    let mut vec = Vec::with_capacity(len);
    unsafe {
        std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

Cette fonction commence par créer un vide `vec` avec la capacité requise, puis appelle `write_bytes` pour remplir le tampon inoccupé avec des zéros. (La `write_byte` fonction traite `len` comme un nombre d'éléments, pas un nombre d'octets, donc cet appel remplit tout le tampon.) La `set_len` méthode d'un vecteur change sa longueur sans rien faire au tampon ; ceci n'est pas sûr, car vous devez vous assurer que l'espace tampon nouvellement inclus contient réellement des valeurs correctement initialisées de type `T`. Mais c'est exactement ce que la `T: Zeroable` limite établit : un bloc de zéro octet représente une `T` valeur valide. Notre utilisation de `set_len` était sûre.

Ici, nous l'utilisons :

```
let v:Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

De toute évidence, `Zeroable` doit être un trait non sûr, car une implémentation qui ne respecte pas son contrat peut conduire à un comportement indéfini :

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v:Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1; // crashes: dereferences null pointer
```

Rust n'a aucune idée de ce qui `Zeroable` est censé signifier, il ne peut donc pas dire quand il est implémenté pour un type inapproprié. Comme pour toute autre fonctionnalité dangereuse, c'est à vous de comprendre et de respecter le contrat d'un trait dangereux.

Notez que le code non sécurisé ne doit pas dépendre de la mise en œuvre correcte de traits ordinaires et sûrs. Par exemple, supposons qu'il y ait une implémentation du `std::hash::Hasher` trait qui renvoie simplement une valeur de hachage aléatoire, sans relation avec les valeurs hachées. Le trait exige que le hachage deux fois des mêmes bits produise la même valeur de hachage, mais cette implémentation ne répond pas à cette exigence ; c'est tout simplement incorrect. Mais comme `Hasher` il ne s'agit pas d'un trait non sécurisé, le code non sécurisé ne doit pas présenter de comportement indéfini lorsqu'il utilise ce hachage. Le `std::collections::HashMap` type est écrit avec soin pour respecter les

contrats des fonctionnalités non sécurisées qu'il utilise, quel que soit le comportement du hacheur. Certes, la table ne fonctionnera pas correctement : les recherches échoueront et les entrées apparaîtront et disparaîtront au hasard. Mais la table ne présentera pas de comportement indéfini.

Pointeurs bruts

Un *pointeur brut* dans Rust est un pointeur sans contrainte. Vous pouvez utiliser des pointeurs bruts pour former toutes sortes de structures que les types de pointeurs vérifiés de Rust ne peuvent pas, comme des listes doublement liées ou des graphiques arbitraires d'objets. Mais parce que les pointeurs bruts sont si flexibles, Rust ne peut pas dire si vous les utilisez en toute sécurité ou non, vous ne pouvez donc les déréférencer que dans un `unsafe` bloc.

Les pointeurs bruts sont essentiellement équivalents aux pointeurs C ou C++, ils sont donc également utiles pour interagir avec du code écrit dans ces langages.

Il existe deux types de pointeurs bruts:

- A `*mut T` est un brutpointeur vers `a T` qui permet de modifier son référent.
- A `*const T` est un brutpointeur vers `a T` qui ne permet de lire que son référent.

(Il n'y a pas de type simple `*T` ; vous devez toujours spécifier soit `const` ou `mut`.)

Vous pouvez créer un pointeur brut par conversion à partir d'une référence, et le déréférencer avec l' `*` opérateur:

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}

assert_eq!(x, 30);
```

Contrairement aux boîtes et aux références, les pointeurs brutspeut être nul, comme `NULL` en C ou `nullptr` en C++ :

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std::ptr::null(),
        Some(r) => r as *const T
    }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw:: <i32>(None), std::ptr::null());
```

Cet exemple n'a pas de `unsafe` blocs : créer des pointeurs bruts, les faire circuler et les comparer sont tous sûrs. Seul le déréférencement d'un pointeur brut n'est pas sûr.

Un pointeur brut vers un type non dimensionné est un pointeur gras, tout comme le serait la référence ou `Box` le type correspondant. Un `*const [u8]` pointeur inclut une longueur avec l'adresse, et un objet trait comme un `*mut dyn std::io::Write` pointeur porte une vtable.

Bien que Rust déréférence implicitement les types de pointeurs sûrs dans diverses situations, les déréférencements de pointeurs bruts doivent être explicites :

- L' `.` opérateur ne déréférera pas implicitement un pointeur brut ; vous devez écrire `(*raw).field` ou `(*raw).method(...)`.
- Les pointeurs bruts ne s'implémentent pas `Deref`, donc déréférez les coercionsne s'applique pas à eux.
- Les opérateurs aiment `==` et `<` comparantpointeurs bruts comme adresses : deux pointeurs bruts sont égaux s'ils pointent vers le même emplacement en mémoire. De même, le hachage d'un pointeur brut hache l'adresse vers laquelle il pointe, pas la valeur de son référent.
- Mise en pagetraits comme `std::fmt::Display` suivrereférences automatiquement, mais ne gèrent pas du tout les pointeurs bruts. Les exceptions sont `std::fmt::Debug` et `std::fmt::Pointer`, qui afficher les pointeurs brutssous forme d'adresses hexadécimales, sans les déréférer.

Contrairement à l' `+` opérateur en C et C++, Rust `+` ne gère pas les pointeurs bruts, mais vous pouvez effectuer une arithmétique de pointeur via

leurs méthodes `offset` et `wrapping_offset`, ou les méthodes `,`, `,` et `add` plus pratiques. Inversement, la méthode donne la distance entre deux pointeurs en octets, bien que nous soyons responsables de nous assurer que le début et la fin sont dans la même région mémoire (le même, par exemple) : `sub wrapping_add wrapping_sub offset_from Vec`

```
let trucks = vec![ "garbage truck", "dump truck", "moonstruck" ];
let first: *const &str = &trucks[0];
let last: *const &str = &trucks[2];
assert_eq!(unsafe { last.offset_from(first) }, 2);
assert_eq!(unsafe { first.offset_from(last) }, -2);
```

Aucune conversion explicite n'est nécessaire pour `first` et `last`; il suffit de spécifier le type. Rust constraint implicitement les références aux pointeurs bruts (mais pas l'inverse, bien sûr).

L' `as` opérateur permet presque toutes les conversions plausibles de références en pointeurs bruts ou entre deux types de pointeurs bruts. Cependant, vous devrez peut-être décomposer une conversion complexe en une série d'étapes plus simples. Par exemple:

```
&vec![42_u8] as *const String; // error: invalid conversion
&vec![42_u8] as *const Vec<u8> as *const String; // permitted
```

Notez que `as` cela ne convertira pas les pointeurs bruts en références. De telles conversions seraient dangereuses et `as` devraient rester une opération sûre. Au lieu de cela, vous devez déréférencer le pointeur brut (dans un `unsafe` bloc) puis emprunter la valeur résultante.

Soyez très prudent lorsque vous faites cela : une référence ainsi produite a une durée de vie illimitée: il n'y a pas de limite à sa durée de vie, car le pointeur brut ne donne à Rust rien sur quoi fonder une telle décision.

Dans [« Une interface sécurisée pour libgit2 »](#), plus loin dans ce chapitre, nous montrons plusieurs exemples de la façon de contraindre correctement les durées de vie.

De nombreux types ont `as_ptr` et `as_mut_ptr` méthodes qui renvoient un pointeur brut vers leur contenu. Par exemple, les tranches de tableau et les chaînes renvoient des pointeurs vers leurs premiers éléments, et certains itérateurs renvoient un pointeur vers l'élément suivant qu'ils produiront. Posséder des types de pointeurs tels que `Box`, `Rc` et `Arc` have `into_raw` et des `from_raw` fonctions qui convertissent vers et

à partir de pointeurs bruts. Certains des contrats de ces méthodes imposent des exigences surprenantes, alors vérifiez leur documentation avant de les utiliser.

Vous pouvez également construire des pointeurs bruts par conversion à partir d'entiers, bien que les seuls entiers auxquels vous pouvez faire confiance pour cela soient généralement ceux que vous avez obtenus à partir d'un pointeur en premier lieu. "[Exemple : RefWithFlag](#)" utilise des pointeurs bruts de cette façon.

Contrairement aux références, les pointeurs bruts ne sont ni `Send` ni `Sync`. Par conséquent, tout type qui inclut des pointeurs bruts n'implémente pas ces traits par défaut. Il n'y a rien d'intrinsèquement dangereux à envoyer ou à partager des pointeurs bruts entre les threads ; après tout, où qu'ils aillent, vous avez toujours besoin d'un `unsafe` bloc pour les déréférencer. Mais étant donné les rôles que jouent généralement les pointeurs bruts, les concepteurs de langage ont considéré que ce comportement était le comportement par défaut le plus utile. Nous avons déjà discuté de la façon de mettre en œuvre `Send` et `Sync` de vous-même dans "[Unsafe Traits](#)".

Déréférencer les pointeurs bruts en toute sécurité

IciVoici quelques règles de bon sens pour utiliser les pointeurs bruts en toute sécurité :

- Le déréférencement des pointeurs nuls ou des pointeurs pendants est un comportement indéfini, tout comme la référence à une mémoire non initialisée ou à des valeurs qui sont sorties de la portée.
- Le déréférencement des pointeurs qui ne sont pas correctement alignés pour leur type de référent est un comportement indéfini.
- Vous ne pouvez emprunter des valeurs à un pointeur brut déréférencé que si cela respecte les règles de sécurité des références expliquées au [chapitre 5](#) : aucune référence ne peut survivre à son référent, l'accès partagé est un accès en lecture seule et l'accès mutable est un accès exclusif. (Cette règle est facile à violer par accident, car les pointeurs bruts sont souvent utilisés pour créer des structures de données avec un partage ou une propriété non standard.)
- Vous ne pouvez utiliser le référent d'un pointeur brut que s'il s'agit d'une valeur bien formée de son type. Par exemple, vous devez vous assurer que le déréférencement de `a *const char` génère un point de code Unicode approprié et non de substitution.

- Vous pouvez utiliser les méthodes `offset` et `wrapping_offset` sur les pointeurs bruts uniquement pour pointer vers des octets dans la variable ou le bloc de mémoire alloué par tas auquel le pointeur d'origine faisait référence, ou vers le premier octet au-delà d'une telle région.

Si vous faites de l'arithmétique de pointeur en convertissant le pointeur en entier, en faisant de l'arithmétique sur l'entier, puis en le reconvertissant en pointeur, le résultat doit être un pointeur que les règles de la `offset` méthode vous auraient permis de produire.

- Si vous affectez au référent d'un pointeur brut, vous ne devez pas violer les invariants de tout type dont le référent fait partie. Par exemple, si vous avez `a *mut u8` pointant vers un octet de `a String`, vous ne pouvez stocker que des valeurs dans ce `u8` qui laisse le `String` maintien UTF-8 bien formé.

La règle d'emprunt mise à part, ce sont essentiellement les mêmes règles que vous devez suivre lorsque vous utilisez des pointeurs en C ou C++.

La raison pour ne pas violer les invariants des types doit être claire. De nombreux types standard de Rust utilisent du code non sécurisé dans leur implémentation, mais fournissent toujours des interfaces sûres en supposant que les contrôles de sécurité, le système de modules et les règles de visibilité de Rust seront respectés. L'utilisation de pointeurs bruts pour contourner ces mesures de protection peut entraîner un comportement indéfini.

Le contrat complet et exact pour les pointeurs bruts n'est pas facile à énoncer et peut changer à mesure que le langage évolue. Mais les principes décrits ici devraient vous garder en territoire sûr.

Exemple : RefWithFlag

Voici un exemple de la façon de prendre un hack classique au niveau `1 bit` rendu possible par des pointeurs bruts et de le transformer en un type Rust complètement sûr. Ce module définit un type, `RefWithFlag<'a, T>`, qui contient à la fois `a &'a T` et `a bool`, comme le tuple `(&'a T, bool)` et parvient toujours à n'occuper qu'un seul mot machine au lieu de deux. Ce type de technique est régulièrement utilisé dans les ramasse-miettes et les machines virtuelles, où certains types (par exemple, le type représentant un objet) sont si nombreux que l'ajout d'un seul mot à chaque valeur augmenterait considérablement l'utilisation de la mémoire :

```

mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// A `&T` and a `bool`, wrapped up in a single word.
    /// The type `T` must require at least two-byte alignment.
    ///
    /// If you're the kind of programmer who's never met a pointer whose
    /// 20-bit you didn't want to steal, well, now you can do it safely!
    /// ("But it's not nearly as exciting this way...")
    pub struct RefWithFlag<'a, T> {
        ptr_and_bit: usize,
        behaves_like: PhantomData<&'a T> // occupies no space
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
        pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
            assert!(align_of::<T>() % 2 == 0);
            RefWithFlag {
                ptr_and_bit: ptr as *const T as usize | flag as usize,
                behaves_like: PhantomData
            }
        }
    }

    pub fn get_ref(&self) ->&'a T {
        unsafe {
            let ptr = (self.ptr_and_bit & !1) as *const T;
            &*ptr
        }
    }

    pub fn get_flag(&self) -> bool {
        self.ptr_and_bit & 1 != 0
    }
}

```

Ce code tire parti du fait que de nombreux types doivent être placés à des adresses paires en mémoire : puisque le bit le moins significatif d'une adresse paire est toujours zéro, nous pouvons y stocker autre chose, puis reconstruire de manière fiable l'adresse d'origine simplement en masquant le bit inférieur. . Tous les types ne sont pas éligibles ; par exemple, les types `u8` et `(bool, [i8; 2])` peuvent être placés à n'importe quelle adresse. Mais nous pouvons vérifier l'alignement du type sur la construction et les types de déchets qui ne fonctionneront pas.

Vous pouvez utiliser `RefWithFlag` comme ceci :

```
use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);
```

Le constructeur `RefWithFlag::new` prend une référence et une `bool` valeur, affirme que le type de la référence est approprié, puis convertit la référence en un pointeur brut, puis en un `usize`. Le `usize` type est défini pour être suffisamment grand pour contenir un pointeur sur le processeur pour lequel nous compilons, donc la conversion d'un pointeur brut en un `usize` et inversement est bien définie. Une fois que nous avons un `usize`, nous savons qu'il doit être pair, nous pouvons donc utiliser l' `|` opérateur au niveau du bit ou pour le combiner avec le `bool`, que nous avons converti en un entier 0 ou 1.

La `get_flag` méthode extrait le `bool` composant de un `RefWithFlag`. C'est simple : il suffit de masquer le bit du bas et de vérifier s'il est différent de zéro.

La `get_ref` méthode extrait la référence d'un fichier `RefWithFlag`. Tout d'abord, il masque le `usize` bit inférieur de et le convertit en un pointeur brut. L' `as` opérateur ne convertira pas les pointeurs bruts en références, mais nous pouvons déréférencer le pointeur brut (dans un `unsafe` bloc, naturellement) et l'emprunter. Emprunter le référent d'un pointeur brut vous donne une référence avec une durée de vie illimitée : Rust accordera à la référence la durée de vie qui ferait vérifier le code qui l'entoure, s'il y en a une. Habituellement, cependant, il existe une durée de vie spécifique qui est plus précise et qui détecterait donc plus d'erreurs. Dans ce cas, puisque `get_ref` le type de retour de est `&'a T`, Rust voit que la durée de vie de la référence est la même que `RefWithFlag` le paramètre de durée de vie de `'a`, c'est exactement ce que nous voulons : c'est la durée de vie de la référence avec laquelle nous avons commencé.

En mémoire, un `RefWithFlag` ressemble à un `usize` : puisque `PhantomData` c'est un type de taille nulle, le `behaves_like` champ ne prend pas de place dans la structure. Mais il `PhantomData` est nécessaire que Rust sache comment traiter les durées de vie dans le code qui utilise `RefWithFlag`. Imaginez à quoi ressemblerait le type sans le `behaves_like` champ :

```
// This won't compile.
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

Au [chapitre 5](#), nous avons souligné que toute structure contenant des références ne doit pas survivre aux valeurs qu'elles empruntent, de peur que les références ne deviennent des pointeurs pendant. La structure doit respecter les restrictions qui s'appliquent à ses champs. Cela s'applique certainement à `RefWithFlag` : dans l'exemple de code que nous venons de voir, `flagged` must not outlive `vec`, puisque

`flagged.get_ref()` renvoie une référence à celui-ci. Mais notre type réduit `RefWithFlag` ne contient aucune référence et n'utilise jamais son paramètre de durée de vie `'a`. C'est juste un `usize`. Comment Rust doit-il savoir que des restrictions s'appliquent à `flagged` la durée de vie de ? L'inclusion d'un `PhantomData<& 'a T>` champ indique à Rust de traiter `RefWithFlag<'a, T>` comme s'il contenait un `& 'a T`, sans affecter réellement la représentation de la structure.

Bien que Rust ne sache pas vraiment ce qui se passe (c'est ce qui le rend `RefWithFlag` dangereux), il fera de son mieux pour vous aider. Si vous omettez le `behaves_like` champ, Rust se plaindra que les paramètres `'a` et `T` ne sont pas utilisés et suggérera d'utiliser un fichier `PhantomData`.

`RefWithFlag` utilise la même tactique que le `Ascii` type que nous avons présenté précédemment pour éviter un comportement indéfini dans son `unsafe` bloc. Le type lui-même est `pub`, mais ses champs ne le sont pas, ce qui signifie que seul le code du `ref_with_flag` module peut créer ou regarder à l'intérieur d'une `RefWithFlag` valeur. Vous n'avez pas besoin d'inspecter beaucoup de code pour être sûr que le `ptr_and_bit` champ est bien construit.

Pointeurs nullables

Un nulle pointeur brut dans Rust est une adresse zéro, tout comme en C et C++. Pour tout type `T`, la `std::ptr::null<T>` fonction renvoie un `*const T` pointeur null et `std::ptr::null_mut<T>` renvoie un `*mut T` pointeur null.

Il existe plusieurs façons de vérifier si un pointeur brut est nul. Le plus simple est la `is_null` méthode, mais la `as_ref` méthode peut être plus

pratique : elle prend un `*const T` pointeur et renvoie un `Option<&'a T>`, transformant un pointeur nul en un `None`. De même, la `as_mut` méthode convertit les `*mut T` pointeurs en `Option<&'a mut T>` valeurs.

Tailles et alignements des caractères

Une valeur de tout `sized` type occupe un nombre constant d'octets en mémoire et doit être placé à une adresse qui est un multiple d'un certain *alignement* valeur, déterminée par l'architecture de la machine. Par exemple, un `(i32, i32)` tuple occupe huit octets et la plupart des processeurs préfèrent qu'il soit placé à une adresse multiple de quatre.

L'appel `std::mem::size_of::<T>()` renvoie la taille d'une valeur de type `T`, en octets, et `std::mem::align_of::<T>()` renvoie son alignement requis. Par exemple :

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

L'alignement de n'importe quel type est toujours une puissance de deux.

La taille d'un type est toujours arrondie à un multiple de son alignement, même s'il pourrait techniquement tenir dans moins d'espace. Par exemple, même si un tuple comme `(f32, u8)` ne nécessite que cinq octets, `size_of::<(f32, u8)>()` est 8, car `align_of::<(f32, u8)>()` est 4. Cela garantit que si vous avez un tableau, la taille du type d'élément reflète toujours l'espacement entre un élément et le suivant.

Pour les types non dimensionnés, la taille et l'alignement dépendent de la valeur disponible. Étant donné une référence à une valeur non dimensionnée, les fonctions `std::mem::size_of_val` et `std::mem::align_of_val` renvoient la taille et l'alignement de la valeur. Ces fonctions peuvent opérer sur des références à des `sized` types à la fois et non dimensionnés :

```
// Fat pointers to slices carry their referent's length.
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std::mem::size_of_val(slice), 20);

let text: &str = "alligator";
assert_eq!(std::mem::size_of_val(text), 9);

use std::fmt::Display;
```

```

let unremarkable: &dyn Display = &193_u8;
let remarkable:&dyn Display = &0.0072973525664;

// These return the size/alignment of the value the
// trait object points to, not those of the trait object
// itself. This information comes from the vtable the
// trait object refers to.
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);

```

Arithmétique du pointeur

Rouiller présente les éléments d'un tableau, d'une tranche ou d'un vecteur sous la forme d'un seul bloc de mémoire contigu, comme illustré à la [Figure 22-1](#). Les éléments sont régulièrement espacés, de sorte que si chaque élément occupe `size` des octets, alors le `i` ème élément commence par le `i * size` ème octet.

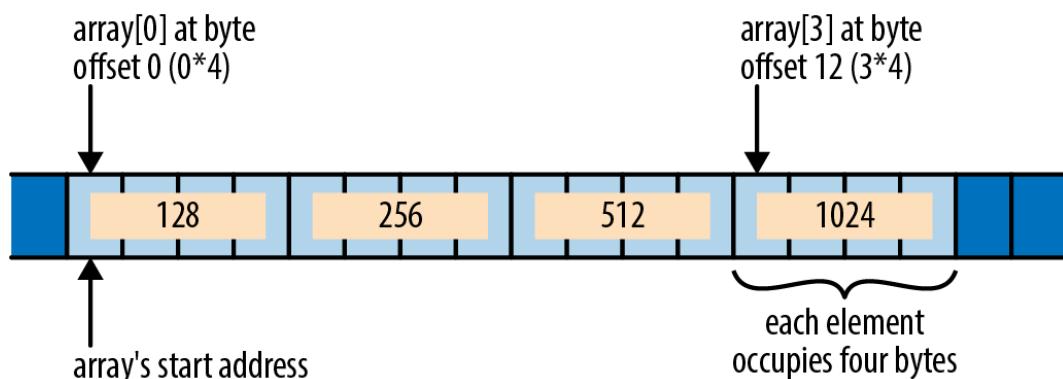


Image 22-1. Un tableau en mémoire

Une bonne conséquence de ceci est que si vous avez deux pointeurs bruts vers des éléments d'un tableau, la comparaison des pointeurs donne les mêmes résultats que la comparaison des indices des éléments : si `i < j`, alors un pointeur brut vers le `i` ème élément est inférieur à un pointeur brut vers le `j` ème élément. Cela rend les pointeurs bruts utiles comme limites sur les traversées de tableaux. En fait, l'itérateur simple de la bibliothèque standard sur une tranche était initialement défini comme ceci :

```

struct Iter<'a, T> {
    ptr: *const T,
    end: *const T,
    ...
}

```

Le `ptr` champ pointe vers l'élément suivant que l'itération doit produire, et le `end` champ sert de limite : quand `ptr == end`, l'itération est terminée.

Une autre conséquence intéressante de la disposition des tableaux : si `element_ptr` est un pointeur brut `*const T` ou vers le i ème élément d'un tableau, alors `offset(i)` est un pointeur brut vers le $(i + o)$ ème élément. Sa définition est équivalente à ceci : `*mut T i element_ptr.offset(o) (i + o)`

```
fn offset<T>(ptr: *const T, count: isize) -> *const T
    where T: Sized
{
    let bytes_per_element = std::mem::size_of::<T>() as isize;
    let byte_offset = count * bytes_per_element;
    (ptr as isize).checked_add(byte_offset).unwrap() as *const T
}
```

La `std::mem::size_of::<T>` fonction renvoie la taille du type `T` en octets. Comme `isize` est, par définition, assez grand pour contenir une adresse, vous pouvez convertir le pointeur de base en un `isize`, effectuer une arithmétique sur cette valeur, puis reconvertisr le résultat en un pointeur.

C'est bien de produire un pointeur sur le premier octet après la fin d'un tableau. Vous ne pouvez pas déréférencer un tel pointeur, mais il peut être utile pour représenter la limite d'une boucle ou pour des contrôles de limites.

Cependant, il s'agit d'un comportement indéfini à utiliser `offset` pour produire un pointeur au-delà de ce point ou avant le début du tableau, même si vous ne le déréférez jamais. Dans un souci d'optimisation, Rust aimeraient supposer que `ptr.offset(i) > ptr` quand `i` est positif et que `ptr.offset(i) < ptr` quand `i` est négatif. Cette hypothèse semble sûre, mais elle peut ne pas tenir si l'arithmétique `offset` dépasse une `isize` valeur. Si `i` est contraint de rester dans le même tableau que `ptr`, aucun débordement ne peut se produire : après tout, le tableau lui-même ne dépasse pas les limites de l'espace d'adressage. (Pour créer des pointeurs vers le premier octet après la fin du coffre-fort, Rust ne place jamais de valeurs à l'extrême supérieure de l'espace d'adressage.)

Si vous avez besoin de décaler des pointeurs au-delà des limites du tableau auquel ils sont associés, vous pouvez utiliser la `wrapping_offset` méthode. Ceci est équivalent à `offset`, mais Rust ne

fait aucune hypothèse sur l'ordre relatif de `ptr.wrapping_offset(i)` et lui- `ptr` même. Bien sûr, vous ne pouvez toujours pas déréférencer ces pointeurs à moins qu'ils ne fassent partie du tableau.

Entrer et sortir de la mémoire

Si vous implémentez un type qui gère sa propre mémoire, vous devrez suivre quelles parties de votre mémoire contiennent des valeurs actives et lesquelles ne sont pas initialisées, tout comme Rust le fait avec les variables locales. Considérez ce code :

```
let pot = "pasta".to_string();
let plate = pot;
```

Une fois ce code exécuté, la situation ressemble à la [Figure 22-2](#).

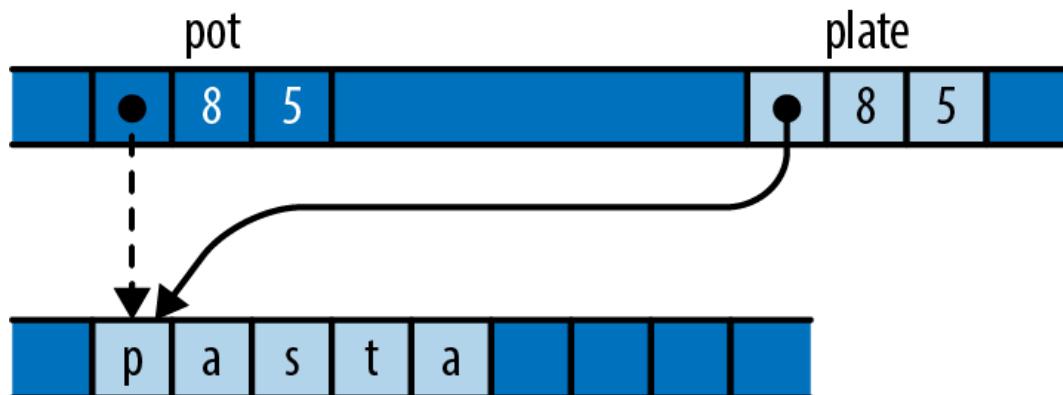


Illustration 22-2. Déplacer une chaîne d'une variable locale à une autre

Après l'affectation, `pot` n'est pas initialisé et `plate` est le propriétaire de la chaîne.

Au niveau de la machine, il n'est pas spécifié ce qu'un mouvement fait à la source, mais en pratique, il ne fait généralement rien du tout. L'affectation laisse probablement `pot` encore un pointeur, une capacité et une longueur pour la chaîne. Naturellement, il serait désastreux de traiter cela comme une valeur en direct, et Rust garantit que vous ne le ferez pas.

Les mêmes considérations s'appliquent aux structures de données qui gèrent leur propre mémoire. Supposons que vous exécutez ce code :

```
let mut noodles = vec![ "udon".to_string() ];
let soba = "soba".to_string();
let last;
```

En mémoire, l'état ressemble à la [Figure 22-3](#).

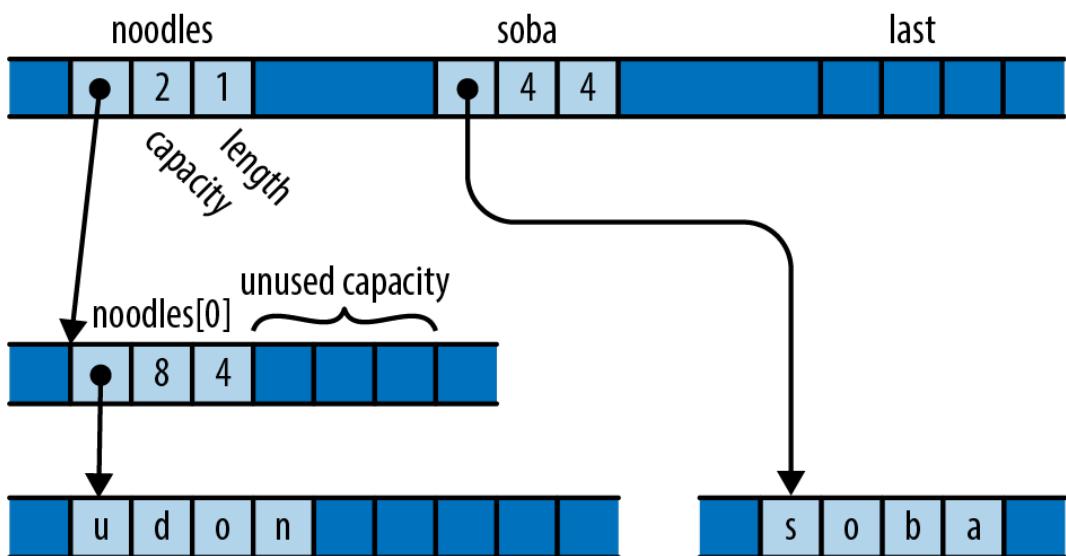


Illustration 22-3. Un vecteur avec une capacité de réserve non initialisée

Le vecteur a la capacité de réserve pour contenir un élément de plus, mais son contenu est indésirable, probablement quel que soit ce que la mémoire contenait auparavant. Supposons que vous exécutez ensuite ce code :

```
noodles.push(soba);
```

Pousser la chaîne sur le vecteur transforme cette mémoire non initialisée en un nouvel élément, comme illustré à la [Figure 22-4](#).

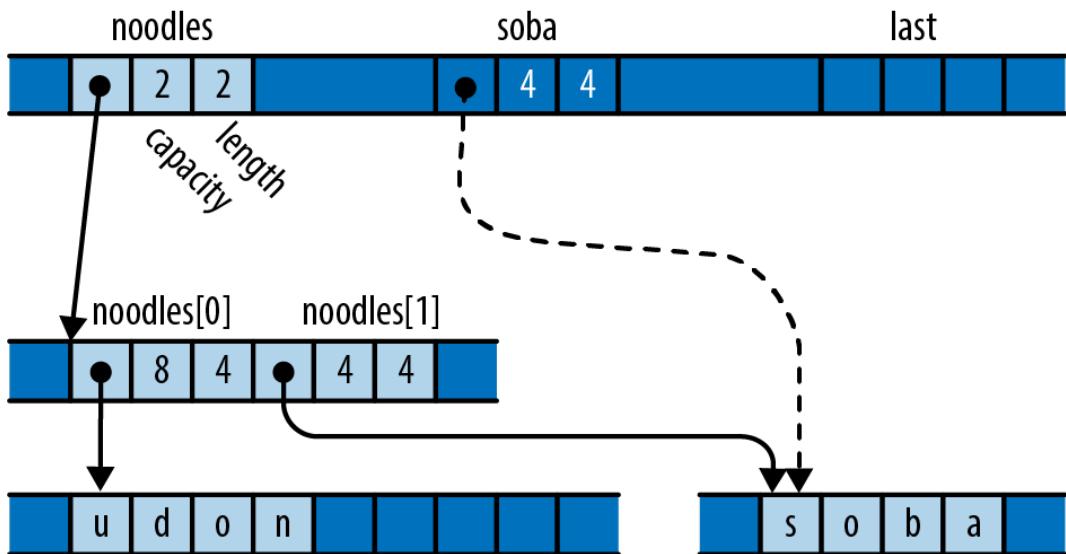


Image 22-4. Après avoir poussé `soba` la valeur de sur le vecteur

Le vecteur a initialisé son espace vide pour posséder la chaîne et incrémenté sa longueur pour le marquer comme un nouvel élément actif. Le vecteur est maintenant le propriétaire de la chaîne ; vous pouvez vous ré-

férer à son deuxième élément, et supprimer le vecteur libérera les deux chaînes. Et `soba` est maintenant non initialisé.

Enfin, considérez ce qui se passe lorsque nous extrayons une valeur du vecteur :

```
last = noodles.pop().unwrap();
```

En mémoire, les choses ressemblent maintenant à [la Figure 22-5](#).

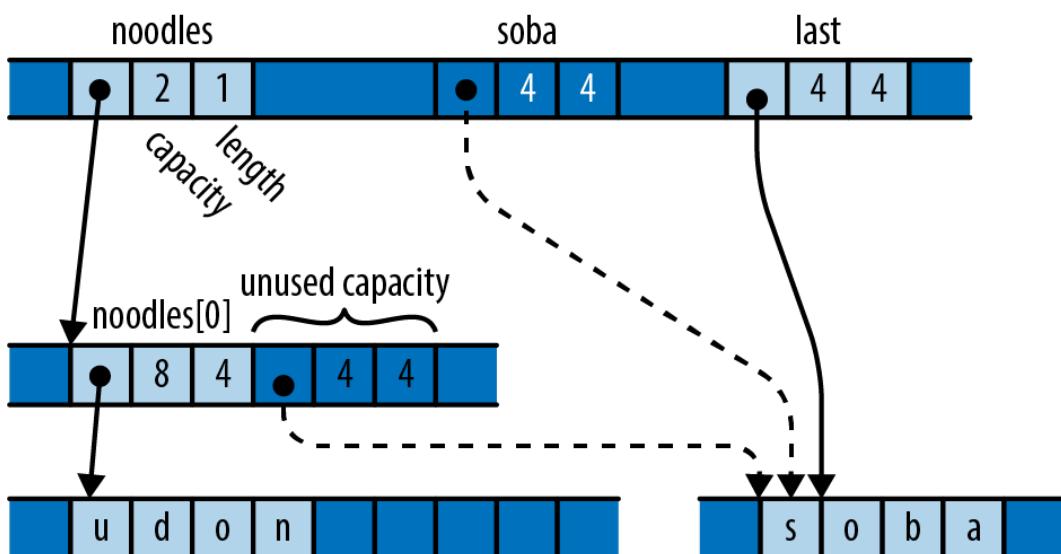


Illustration 22-5. Après avoir inséré un élément du vecteur dans `last`

La variable `last` a pris possession de la chaîne. Le vecteur a décrémenté sa longueur pour indiquer que l'espace qui contenait la chaîne n'est plus initialisé.

Tout comme avec `pot` et `pasta` précédemment, les trois de `soba`, `last`, et l'espace libre du vecteur contiennent probablement des modèles de bits identiques. Mais seul `last` est considéré comme propriétaire de la valeur. Traiter l'un ou l'autre des deux autres emplacements comme étant en direct serait une erreur.

La véritable définition d'une valeur initialisée est celle qui est *traitée comme live*. L'écriture dans les octets d'une valeur est généralement une partie nécessaire de l'initialisation, mais uniquement parce que cela prépare la valeur à être traitée comme active. Un déplacement et une copie ont tous deux le même effet sur la mémoire ; la différence entre les deux est qu'après un déplacement, la source n'est plus traitée comme en direct, alors qu'après une copie, la source et la destination sont en direct.

Rust suit les variables locales actives au moment de la compilation et vous empêche d'utiliser des variables dont les valeurs ont été déplacées ailleurs. Des types tels que `Vec`, `HashMap`, `Box`, etc. suivent leurs tampons de manière dynamique. Si vous implémentez un type qui gère sa propre mémoire, vous devrez faire de même.

Rust fournit deux opérations essentielles pour implémenter de tels types:

`std::ptr::read(src)`

Se déplace une valeur hors de l'emplacement `src` vers lequel pointe, transférant la propriété à l'appelant. L' `src` argument doit être un `*const T` pointeur brut, où `T` est un type dimensionné. Après avoir appelé cette fonction, le contenu de `*src` n'est pas affecté, mais à moins que ce ne `T` soit `Copy`, vous devez vous assurer que votre programme les traite comme de la mémoire non initialisée.

C'est l'opération derrière `Vec::pop`. Extraire une valeur appelle `read` à déplacer la valeur hors du tampon, puis décrémente la longueur pour marquer cet espace comme capacité non initialisée.

`std::ptr::write(dest, value)`

Se déplace `value` dans l'emplacement `dest` pointé vers, qui doit être une mémoire non initialisée avant l'appel. Le référent est désormais propriétaire de la valeur. Ici, `dest` doit être un `*mut T` pointeur brut et `value` une `T` valeur, où `T` est un type dimensionné.

C'est l'opération derrière `Vec::push`. Pousser une valeur appelle `write` à déplacer la valeur dans l'espace disponible suivant, puis incrémenter la longueur pour marquer cet espace comme un élément valide.

Les deux sont des fonctions libres, pas des méthodes sur les types de pointeurs bruts.

Notez que vous ne pouvez pas faire ces choses avec l'un des types de pointeurs sûrs de Rust. Ils exigent tous que leurs référents soient initialisés à tout moment, donc transformer une mémoire non initialisée en une valeur, ou vice versa, est hors de leur portée. Les pointeurs bruts font l'affaire.

La bibliothèque standard fournit également des fonctions pour déplacer des tableaux de valeurs d'un bloc de mémoire à un autre :

`std::ptr::copy(src, dst, count)`

Se déplace le tableau de `count` valeurs en mémoire commençant à `src` jusqu'à la mémoire à `dst`, comme si vous aviez écrit une boucle d'appels `read` et pour les déplacer un par un. `write` La mémoire destination doit être désinitialisée avant l'appel, et ensuite la mémoire source reste non initialisée. Les arguments `src` et `dst` doivent être des pointeurs bruts et doivent être un `.dest *const T *mut T count usize`

`ptr. copy_to(dst, compter)`

Une version pratique de `copy` qui déplace le tableau de `count` valeurs en mémoire de `ptr` à `dst`, plutôt que de prendre son point de départ comme argument.

`std::ptr::copy_nonoverlapping(src, dst, count)`

Comme l'appel correspondant à `copy`, sauf que son contrat exige en outre que les blocs de mémoire source et destination ne doivent pas se chevaucher. Cela peut être légèrement plus rapide que d'appeler `copy`.

`ptr.copy_to_nonoverlapping(dst, count)`

Une version pratique de `copy_nonoverlapping`, comme `copy_to`.

Il existe deux autres familles de fonctions `read` et `write`, également dans le `std::ptr` module :

`read_unaligned, write_unaligned`

Ces fonctions sont comme `read` et `write`, sauf que le pointeur n'a pas besoin d'être aligné comme normalement requis pour le type référent. Ces fonctions peuvent être plus lentes que les fonctions plain `read` et `write`.

`read_volatile, write_volatile`

Ces fonctions sont l'équivalent des lectures et écritures volatiles en C ou C++.

Exemple : GapBuffer

Voici un exemple qui utilise les fonctions de pointeur brutes décrites ci-dessus.

Supposons que vous écrivez un éditeur de texte et que vous recherchez un type pour représenter le texte. Vous pouvez choisir `String` et utiliser

les méthodes `insert` et `remove` pour insérer et supprimer des caractères au fur et à mesure que l'utilisateur tape. Mais s'ils éditent du texte au début d'un fichier volumineux, ces méthodes peuvent être coûteuses : l'insertion d'un nouveau caractère implique de décaler tout le reste de la chaîne vers la droite en mémoire, et la suppression la décale entièrement vers la gauche. Vous aimeriez que ces opérations courantes soient moins chères.

L'éditeur de texte Emacs utilise une structure de données simple appelée *tampon* d'espacement qui peut insérer et supprimer des caractères en temps constant. Tandis que `a String` garde toute sa capacité de réserve à la fin du texte, ce qui rend `push` et `pop` bon marché, un tampon d'écart garde sa capacité de réserve au milieu du texte, à l'endroit où l'édition a lieu. Cette capacité de réserve s'appelle l'`écart`. L'insertion ou la suppression d'éléments au niveau de l'espace est bon marché : il vous suffit de réduire ou d'agrandir l'espace selon vos besoins. Vous pouvez déplacer l'espace à n'importe quel endroit de votre choix en déplaçant le texte d'un côté de l'espace à l'autre. Lorsque l'espace est vide, vous migrez vers un tampon plus grand.

Alors que l'insertion et la suppression dans un tampon d'espace sont rapides, la modification de la position à laquelle elles ont lieu implique le déplacement de l'espace vers la nouvelle position. Le déplacement des éléments nécessite un temps proportionnel à la distance parcourue. Heureusement, une activité d'édition typique implique de faire un tas de changements dans un quartier du tampon avant de partir et de jouer avec du texte ailleurs.

Dans cette section, nous allons implémenter un tampon d'écart dans Rust. Pour éviter d'être distrait par UTF-8, nous allons faire en sorte que nos `char` valeurs de stockage tampon soient directement, mais les principes de fonctionnement seraient les mêmes si nous stockions le texte sous une autre forme.

Tout d'abord, nous allons montrer un tampon d'écart en action. Ce code crée un `GapBuffer`, y insère du texte, puis déplace le point d'insertion juste avant le dernier mot :

```
let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

Après avoir exécuté ce code, le tampon ressemble à celui illustré à la [Figure 22-6](#).

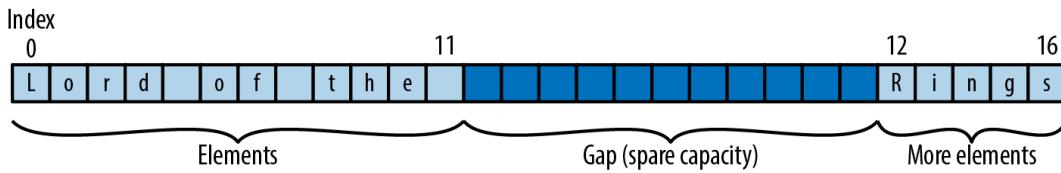


Image 22-6. Un tampon vide contenant du texte

L'insertion consiste à combler le vide avec un nouveau texte. Ce code ajoute un mot et ruine le film :

```
buf.insert_iter("Onion ".chars());
```

Il en résulte l'état illustré à la [Figure 22-7](#).

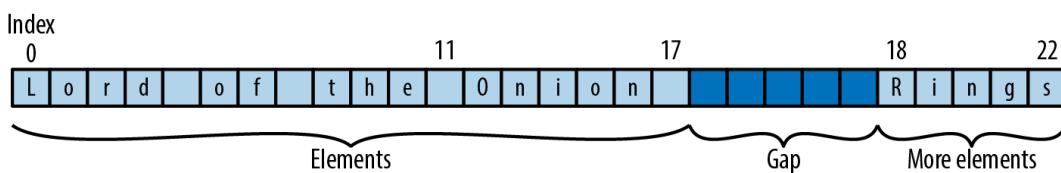


Image 22-7. Un tampon vide contenant du texte supplémentaire

Voici notre `GapBuffer` type :

```
use std;
use std::ops::Range;

pub struct GapBuffer<T> {
    // Storage for elements. This has the capacity we need, but its length
    // always remains zero. GapBuffer puts its elements and the gap in the
    // `Vec`'s "unused" capacity.
    storage:Vec<T>,

    // Range of uninitialized elements in the middle of `storage`.
    // Elements before and after this range are always initialized.
    gap:Range<usize>
}
```

`GapBuffer` utilise son `storage` champ d'une manière étrange.² Il ne stocke jamais réellement d'éléments dans le vecteur, ou pas tout à fait. Il appelle simplement `vec::with_capacity(n)` pour obtenir un bloc de mémoire suffisamment grand pour contenir des `n` valeurs, obtient des pointeurs bruts vers cette mémoire via les vecteurs `as_ptr` et les `as_mut_ptr` méthodes, puis utilise le tampon directement à ses propres

fins. La longueur du vecteur reste toujours nulle. Lorsque le `vec` est supprimé, le `vec` n'essaie pas de libérer ses éléments, car il ne sait pas qu'il en a, mais il libère le bloc de mémoire. C'est ce qui `GapBuffer` veut; il a sa propre `Drop` implémentation qui sait où se trouvent les éléments actifs et les supprime correctement.

`GapBuffer` Les méthodes les plus simples correspondent à ce que vous attendez :

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// Return the number of elements this GapBuffer could hold without
    /// reallocation.
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// Return the number of elements this GapBuffer currently holds.
    pub fn len(&self) -> usize {
        self.capacity() - self.gap.len()
    }

    /// Return the current insertion position.
    pub fn position(&self) -> usize {
        self.gap.start
    }

    ...
}
```

Il nettoie la plupart des fonctions suivantes pour avoir une méthode utilitaire qui renvoie un pointeur brut vers l'élément buffer à un index donné. Ceci étant Rust, nous finissons par avoir besoin d'une méthode pour les `mut` pointeurs et d'une autre pour `const`. Contrairement aux méthodes précédentes, celles-ci ne sont pas publiques. Continuation de ce `impl` bloc :

```
/// Return a pointer to the `index`th element of the underlying storage,
/// regardless of the gap.
///
/// Safety: `index` must be a valid index into `self.storage`.
unsafe fn space(&self, index: usize) -> *const T {
```

```

        self.storage.as_ptr().offset(index as isize)
    }

    /// Return a mutable pointer to the `index`th element of the underlying
    /// storage, regardless of the gap.
    ///
    /// Safety: `index` must be a valid index into `self.storage`.
    unsafe fn space_mut(&mut self, index: usize) -> *mut T {
        self.storage.as_mut_ptr().offset(index as isize)
    }
}

```

Pour trouver l'élément à un index donné, vous devez déterminer si l'index tombe avant ou après l'écart et ajuster en conséquence :

```

/// Return the offset in the buffer of the `index`th element, taking
/// the gap into account. This does not check whether index is in range,
/// but it never returns an index in the gap.
fn index_to_raw(&self, index: usize) -> usize {
    if index < self.gap.start {
        index
    } else {
        index + self.gap.len()
    }
}

/// Return a reference to the `index`th element,
/// or `None` if `index` is out of bounds.
pub fn get(&self, index: usize) -> Option<&T> {
    let raw = self.index_to_raw(index);
    if raw < self.capacity() {
        unsafe {
            // We just checked `raw` against self.capacity(),
            // and index_to_raw skips the gap, so this is safe.
            Some(&*self.space(raw))
        }
    } else {
        None
    }
}

```

Lorsque nous commençons à faire des insertions et des suppressions dans une autre partie du tampon, nous devons déplacer l'espace vers le nouvel emplacement. Déplacer l'espace vers la droite implique de déplacer des éléments vers la gauche, et vice versa, tout comme la bulle d'un niveau à bulle se déplace dans un sens lorsque le fluide s'écoule dans l'autre :

```

/// Set the current insertion position to `pos`.
/// If `pos` is out of bounds, panic.
pub fn set_position(&mut self, pos:usize) {
    if pos > self.len() {
        panic!("index {} out of range for GapBuffer", pos);
    }

    unsafe {
        let gap = self.gap.clone();
        if pos > gap.start {
            // `pos` falls after the gap. Move the gap right
            // by shifting elements after the gap to before it.
            let distance = pos - gap.start;
            std::ptr::copy(self.space(gap.end),
                          self.space_mut(gap.start),
                          distance);
        } else if pos < gap.start {
            // `pos` falls before the gap. Move the gap left
            // by shifting elements before the gap to after it.
            let distance = gap.start - pos;
            std::ptr::copy(self.space(pos),
                          self.space_mut(gap.end - distance),
                          distance);
        }
    }

    self.gap = pos .. pos + gap.len();
}
}

```

Cette fonction utilise la `std::ptr::copy` méthode pour décaler les éléments ; `copy` nécessite que la destination soit non initialisée et laisse la source non initialisée. Les plages source et destination peuvent se chevaucher, mais `copy` gère ce cas correctement. Étant donné que l'espace est une mémoire non initialisée avant l'appel et que la fonction ajuste la position de l'espace pour couvrir l'espace libéré par la copie, le `copy` contrat de la fonction est satisfait.

L'insertion et le retrait d'éléments sont relativement simples. L'insertion prend un espace à partir de l'espace pour le nouvel élément, tandis que la suppression déplace une valeur vers l'extérieur et agrandit l'espace pour couvrir l'espace qu'il occupait auparavant :

```

/// Insert `elt` at the current insertion position,
/// and leave the insertion position after it.
pub fn insert(&mut self, elt:T) {
    if self.gap.len() == 0 {

```

```

        self.enlarge_gap();
    }

    unsafe {
        let index = self.gap.start;
        std::ptr::write(self.space_mut(index), elt);
    }
    self.gap.start += 1;
}

/// Insert the elements produced by `iter` at the current insertion
/// position, and leave the insertion position after them.
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}

/// Remove the element just after the insertion position
/// and return it, or return `None` if the insertion position
/// is at the end of the GapBuffer.
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}

```

Semblable à la façon dont `Vec` utilise `std::ptr::write` pour pousser et `std::ptr::read` pour `pop`, `GapBuffer` utilise `write` pour `insert` et `read` pour `remove`. Et tout comme `Vec` doit ajuster sa longueur pour maintenir la frontière entre les éléments initialisés et la capacité de réserve, `GapBuffer` ajuste son écart.

Lorsque l'espace a été comblé, la `insert` méthode doit agrandir la mémoire tampon pour acquérir plus d'espace libre. La `enlarge_gap` méthode (la dernière du `impl` bloc) gère ceci :

```

/// Double the capacity of `self.storage`.
fn enlarge_gap(&mut self) {

```

```

let mut new_capacity = self.capacity() * 2;
if new_capacity == 0 {
    // The existing vector is empty.
    // Choose a reasonable starting capacity.
    new_capacity = 4;
}

// We have no idea what resizing a Vec does with its "unused"
// capacity. So just create a new vector and move over the elements.
let mut new = Vec::with_capacity(new_capacity);
let after_gap = self.capacity() - self.gap.end;
let new_gap = self.gap.start .. new.capacity() - after_gap;

unsafe {
    // Move the elements that fall before the gap.
    std::ptr::copy_nonoverlapping(self.space(0),
                                  new.as_mut_ptr(),
                                  self.gap.start);

    // Move the elements that fall after the gap.
    let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize);
    std::ptr::copy_nonoverlapping(self.space(self.gap.end),
                                  new_gap_end,
                                  after_gap);
}

// This frees the old Vec, but drops no elements,
// because the Vec's length is zero.
self.storage = new;
self.gap = new_gap;
}

```

Alors que `set_position` doit utiliser `copy` pour déplacer des éléments d'avant en arrière dans l'espace, `enlarge_gap` peut utiliser `copy_nonoverlapping`, car il déplace des éléments vers un tout nouveau tampon.

Déplacer le nouveau vecteur dans `self.storage` les gouttes de l'ancien vecteur. Comme sa longueur est nulle, l'ancien vecteur pense qu'il n'a aucun élément à supprimer et libère simplement son tampon. Soigneusement, `copy_nonoverlapping` laisse sa source non initialisée, donc l'ancien vecteur a raison dans cette croyance : tous les éléments appartiennent maintenant au nouveau vecteur.

Enfin, nous devons nous assurer que déposer a `GapBuffer` supprime tous ses éléments :

```

impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {
                std::ptr::drop_in_place(self.space_mut(i));
            }
            for i in self.gap.end .. self.capacity() {
                std::ptr::drop_in_place(self.space_mut(i));
            }
        }
    }
}

```

Les éléments se trouvent avant et après l'écart, nous parcourons donc chaque région et utilisons la `std::ptr::drop_in_place` fonction pour supprimer chacun d'eux. La `drop_in_place` fonction est un utilitaire qui se comporte comme `drop(std::ptr::read(ptr))`, mais ne prend pas la peine de déplacer la valeur vers son appelant (et fonctionne donc sur des types non dimensionnés). Et tout comme dans `enlarge_gap`, au moment où le vecteur `self.storage` est supprimé, son tampon n'est vraiment pas initialisé.

Comme les autres types que nous avons montrés dans ce chapitre, `GapBuffer` garantit que ses propres invariants sont suffisants pour s'assurer que le contrat de chaque fonctionnalité non sécurisée qu'il utilise est respecté, de sorte qu'aucune de ses méthodes publiques n'a besoin d'être marquée comme non sécurisée. `GapBuffer` implémente une interface sécurisée pour une fonctionnalité qui ne peut pas être écrite efficacement dans un code sécurisé.

Sécurité anti-panique dans un code dangereux

A Rust, une panique peut généralement pas provoquer un comportement indéfini ; la `panic!` macro n'est pas une fonctionnalité dangereuse. Mais lorsque vous décidez de travailler avec un code non sécurisé, la sécurité anti-panique fait partie de votre travail.

Considérez la `GapBuffer::remove` méthode de la section précédente :

```

pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }
}

```

```
let element = unsafe {
    std::ptr::read(self.space(self.gap.end))
};

self.gap.end += 1;
Some(element)
}
```

L'appel à `read` déplace l'élément suivant immédiatement l'espace hors du tampon, laissant derrière lui un espace non initialisé. À ce stade, le `GapBuffer` est dans un état incohérent : nous avons rompu l'invariant selon lequel tous les éléments en dehors de l'espace doivent être initialisés. Heureusement, la toute prochaine déclaration agrandit l'écart pour couvrir cet espace, donc au moment où nous revenons, l'invariant tient à nouveau.

Mais considérez ce qui se passerait si, après l'appel à `read` mais avant l'ajustement à `self.gap.end`, ce code tentait d'utiliser une fonctionnalité susceptible de paniquer, par exemple l'indexation d'une tranche. Quitter la méthode brusquement n'importe où entre ces deux actions laisserait le `GapBuffer` avec un élément non initialisé en dehors de l'espace. Le prochain appel à `remove` pourrait essayer à `read` nouveau ; même simplement laisser tomber le `GapBuffer` essaierait de le laisser tomber. Les deux ont un comportement indéfini, car ils accèdent à une mémoire non initialisée.

Il est presque inévitable que les méthodes d'un type relâchent momentanément les invariants du type pendant qu'elles font leur travail, puis remettent tout en ordre avant leur retour. Une méthode intermédiaire de panique pourrait raccourcir ce processus de nettoyage, laissant le type dans un état incohérent.

Si le type utilise uniquement du code sécurisé, cette incohérence peut entraîner un mauvais comportement du type, mais elle ne peut pas introduire de comportement indéfini. Mais le code utilisant des fonctionnalités non sécurisées compte généralement sur ses invariants pour respecter les contrats de ces fonctionnalités. Les invariants rompus conduisent à des contrats rompus, qui conduisent à un comportement indéfini.

Lorsque vous travaillez avec des fonctionnalités non sécurisées, vous devez faire particulièrement attention à identifier ces régions sensibles du code où les invariants sont temporairement relâchés, et vous assurer qu'ils ne font rien qui pourrait paniquer..

Réinterpréter la mémoire avec les syndicats

Rouiller fournit de nombreuses abstractions utiles, mais en fin de compte, le logiciel que vous écrivez ne fait que pousser des octets. Les unions sont l'une des fonctionnalités les plus puissantes de Rust pour manipuler ces octets et choisir comment ils sont interprétés. Par exemple, toute collection de 32 bits - 4 octets - peut être interprétée comme un entier ou comme un nombre à virgule flottante. L'une ou l'autre interprétation est valide, bien que l'interprétation des données destinées à l'une comme l'autre entraînera probablement un non-sens.

Une union représentant une collection d'octets pouvant être interprétés comme un entier ou un nombre à virgule flottante s'écrirait comme suit :

```
union FloatOrInt {
    f: f32,
    i:i32,
}
```

C'est une union avec deux champs, `f` et `i`. Ils peuvent être affectés à la même manière que les champs d'une structure, mais lors de la construction d'une union, contrairement à une structure, vous devez en choisir exactement un. Là où les champs d'une structure font référence à différentes positions en mémoire, les champs d'une union font référence à différentes interprétations de la même séquence de bits. Attribuer à un champ différent signifie simplement écraser certains ou tous ces bits, conformément à un type approprié. Ici, `one` fait référence à une seule étendue de mémoire de 32 bits, qui stocke d'abord 1 codée sous la forme d'un entier simple, puis 1.0 sous la forme d'un nombre à virgule flottante IEEE 754. Dès que `f` est écrit dans , la valeur précédemment écrite dans `FloatOrInt` est écrasée :

```
let mut one = FloatOrInt { i:1 };
assert_eq!(unsafe { one.i }, 0x00_00_00_01);
one.f = 1.0;
assert_eq!(unsafe { one.i }, 0x3F_80_00_00);
```

Pour la même raison, la taille d'une union est déterminée par son plus grand champ. Par exemple, cette union a une taille de 64 bits, même si ce `SmallOrLarge::s` n'est qu'un `bool`:

```

union SmallOrLarge {
    s: bool,
    l:u64
}

```

Bien que la construction d'une union ou l'affectation à ses champs soit totalement sûre, la lecture à partir de n'importe quel champ d'une union est toujours dangereuse :

```

let u = SmallOrLarge { l:1337 };
println!("{}", unsafe {u.l}); // prints 1337

```

En effet, contrairement aux énumérations, les unions n'ont pas de balise. Le compilateur n'ajoute aucun bit supplémentaire pour différencier les variantes. Il n'y a aucun moyen de savoir au moment de l'exécution si a `SmallOrLarge` doit être interprété comme a `u64` ou a `bool`, à moins que le programme n'ait un contexte supplémentaire.

Il n'y a pas non plus de garantie intégrée que la configuration binaire d'un champ donné est valide. Par exemple, écrire dans le champ d'une `SmallOrLarge` valeur `1` écrasera son `s` champ, créant un motif de bits qui ne signifie certainement rien d'utile et qui n'est probablement pas valide `bool`. Par conséquent, bien que l'écriture dans les champs union soit sûre, chaque lecture nécessite `unsafe`. La lecture de `u.s` n'est autorisée que lorsque les bits du `s` champ forment un `bool`; sinon, il s'agit d'un comportement indéfini.

Avec ces restrictions à l'esprit, les unions peuvent être un moyen utile de réinterpréter temporairement certaines données, en particulier lors de calculs sur la représentation des valeurs plutôt que sur les valeurs elles-mêmes. Par exemple, le type mentionné précédemment `FloatOrInt` peut facilement être utilisé pour imprimer les bits individuels d'un nombre à virgule flottante, même s'il `f32` n'implémente pas le `Binary` formateur :

```

let float = FloatOrInt { f:31337.0 };
// prints 1000110111101001101001000000000
println!("{:b}", unsafe { float.i });

```

Bien que ces exemples simples fonctionnent presque certainement comme prévu sur n'importe quelle version du compilateur, il n'y a aucune garantie qu'un champ commence à un endroit spécifique à moins

qu'un attribut ne soit ajouté à la `union` définition indiquant au compilateur comment disposer les données en mémoire. L'ajout de l'attribut `# [repr(C)]` garantit que tous les champs commencent à l'offset 0, plutôt qu'à l'endroit souhaité par le compilateur. Avec cette garantie en place, le comportement d'écrasement peut être utilisé pour extraire des bits individuels, comme le bit de signe d'un entier :

```
#[repr(C)]
union SignExtractor {
    value: i64,
    bytes:[u8; 8]
}

fn sign(int: i64) -> bool {
    let se = SignExtractor { value:int };
    println!( "{:b} ({:?}", se.value, unsafe { se.bytes });
    unsafe { se.bytes[7] >= 0b10000000 }

    assert_eq!(sign(-1), true);
    assert_eq!(sign(1), false);
    assert_eq!(sign(i64::MAX), false);
    assert_eq!(sign(i64::MIN), true);
}
```

Ici, le bit de signe est le bit le plus significatif de l'octet le plus significatif. Comme les processeurs x86 sont little-endian, l'ordre de ces octets est inversé ; l'octet le plus significatif n'est pas `bytes[0]`, mais `bytes[7]`. Normalement, ce n'est pas quelque chose que le code Rust doit gérer, mais comme ce code travaille directement avec la représentation en mémoire du `i64`, ces détails de bas niveau deviennent importants.

Étant donné que les syndicats ne peuvent pas dire comment supprimer leur contenu, tous leurs champs doivent être `Copy`. Cependant, si vous devez simplement stocker un `String` dans une union, il existe une solution de contournement ; consultez la documentation de la bibliothèque standard pour `std::mem::ManuallyDrop`.

Unions correspondantes

Correspondant à sur une union Rust, c'est comme faire correspondre une structure, sauf que chaque motif doit spécifier exactement un champ :

```

unsafe {
    match u {
        SmallOrLarge { s: true } => { println!("boolean true"); }
        SmallOrLarge { l:2 } => { println!("integer 2"); }
        _ => { println!("something else"); }
    }
}

```

Un `match` bras qui correspond à une variante d'union sans spécifier de valeur réussira toujours. Le code suivant provoquera un comportement indéfini si le dernier champ écrit de `u` était `u.i`:

```

// Undefined behavior!
unsafe {
    match u {
        FloatOrInt { f } => { println!("float {}", f) },
        // warning: unreachable pattern
        FloatOrInt { i } => { println!("int {}", i) }
    }
}

```

Syndicats d'emprunt

Empruntun champ d'une union emprunte toute l'union. Cela signifie que, conformément aux règles d'emprunt normales, emprunter un champ comme mutable exclut tout emprunt supplémentaire sur celui-ci ou sur d'autres champs, et emprunter un champ comme immuable signifie qu'il ne peut y avoir d'emprunt mutable sur aucun champ.

Comme nous le verrons dans le chapitre suivant, Rust vous aide à construire des interfaces sûres non seulement pour votre propre code non sûr, mais aussi pour du code écrit dans d'autres langages.. Unsafe est, comme son nom l'indique, lourd, mais utilisé avec précaution, il peut vous permettre de créer un code hautement performant qui conserve les garanties dont bénéficient les programmeurs Rust.

¹ Eh bien, c'est un classique d'où nous venons.

² Il existe de meilleures façons de gérer cela en utilisant le `RawVec` type du crate interne au compilateur `alloc`, mais ce crate est toujours instable.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 23. Fonctions étrangères

Cyberespace. Complexité impensable. Des lignes de lumière s'étendent dans le non-espace de l'esprit, des grappes et des constellations de données. Comme les lumières de la ville, qui s'éloignent... .

—William Gibson, *neuromancien*

Tragiquement, tous les programmes du monde ne sont pas écrits en Rust. Il existe de nombreuses bibliothèques et interfaces critiques implémentées dans d'autres langages que nous aimerais pouvoir utiliser dans nos programmes Rust. L'*interface de fonction étrangère* (FFI) de Rust permet au code Rust d'appeler des fonctions écrites en C et, dans certains cas, en C++. Étant donné que la plupart des systèmes d'exploitation offrent des interfaces C, l'interface de fonction étrangère de Rust permet un accès immédiat à toutes sortes d'installations de bas niveau.

Dans ce chapitre, nous allons écrire un programme qui relie avec `libgit2`, une bibliothèque C pour travailler avec le système de contrôle de version Git. Tout d'abord, nous allons montrer ce que c'est que d'utiliser des fonctions C directement depuis Rust, en utilisant les fonctionnalités non sécurisées présentées dans le chapitre précédent. Ensuite, nous montrerons comment construire une interface sécurisée pour , en nous inspirant du crate `libgit2` open source `git2-rs` , qui fait exactement cela.

Nous supposerons que vous connaissez le C et les mécanismes de compilation et de liaison des programmes C. Travailler avec C++ est similaire. Nous supposerons également que vous êtes quelque peu familiarisé avec le système de contrôle de version Git.

Il existe des caisses Rust pour communiquer avec de nombreux autres langages, notamment Python, JavaScript, Lua et Java. Nous n'avons pas la place de les couvrir ici, mais en fin de compte, toutes ces interfaces sont construites à l'aide de l'interface de fonction étrangère C, donc ce chapitre devrait vous donner une longueur d'avance, quel que soit le langage avec lequel vous devez travailler.

Trouver des représentations de données communes

Le dénominateur commun de Rust et C est le langage machine, donc pour anticiper à quoi ressemblent les valeurs de Rust pour le code C, ou vice versa, vous devez considérer leurs représentations au niveau de la machine. Tout au long du livre, nous nous sommes efforcés de montrer comment les valeurs sont réellement représentées en mémoire, vous avez donc probablement remarqué que les mondes de données de C et de Rust ont beaucoup en commun : un Rust `usize` et un C `size_t` sont identiques, par exemple , et les structures sont fondamentalement la même idée dans les deux langages. Pour établir une correspondance entre les types Rust et C, nous allons commencer par les primitives, puis progresser vers des types plus compliqués.

Donné son utilisation principale en tant que langage de programmation système, C a toujours été étonnamment lâche sur les représentations de ses types: un `int` est généralement long de 32 bits, mais peut être plus long ou aussi court que 16 bits ; un C `char` peut être signé ou non signé ; etc. Pour faire face à cette variabilité, le `std::os::raw` module de Rust définit un ensemble de types Rust qui sont garantis d'avoir la même représentation que certains types C ([Tableau 23-1](#)). Celles-ci couvrent les types entiers et caractères primitifs.

Tableau 23-1. std::os::raw types en rouille

type C	Correspondant std::os::raw type
<code>short</code>	<code>c_short</code>
<code>int</code>	<code>c_int</code>
<code>long</code>	<code>c_long</code>
<code>long long</code>	<code>c_longlong</code>
<code>unsigned short</code>	<code>c_ushort</code>
<code>unsigned, unsigned int</code>	<code>c_uint</code>
<code>unsigned long</code>	<code>c_ulong</code>
<code>unsigned long long</code>	<code>c_ulonglong</code>
<code>char</code>	<code>c_char</code>
<code>signed char</code>	<code>c_schar</code>
<code>unsigned char</code>	<code>c_uchar</code>
<code>float</code>	<code>c_float</code>
<code>double</code>	<code>c_double</code>
<code>void *, const void *</code>	<code>*mut c_void, *const c_void</code>

Quelques remarques sur [le Tableau 23-1](#) :

- À l'exception de `c_void`, tous les types Rust ici sont des alias pour certains types Rust primitifs : `c_char`, par exemple, est soit `i8` ou `u8`.
- Un Rust `bool` est équivalent à un C ou C++ `bool`.
- Le type 32 bits de Rust `char` n'est pas l'analogue de `wchar_t`, dont la largeur et l'encodage varient d'une implémentation à l'autre. Le type C `char32_t` est plus proche, mais son encodage n'est toujours pas garanti comme étant Unicode.
- Les types primitifs `usize` et `isize` de Rust ont les mêmes représentations que C et `.size_t ptrdiff_t`

- Les pointeurs C et C++ et les références C++ correspondent aux types de pointeurs bruts de Rust, `*mut T` et `*const T`.
- Techniquement, la norme C permet aux implémentations d'utiliser des représentations pour lesquelles Rust n'a pas de type correspondant : entiers 32 bits, représentations de signe et de grandeur pour les valeurs signées, etc. En pratique, sur chaque plate-forme sur laquelle Rust a été porté, chaque type d'entier C commun a une correspondance dans Rust.

Pour définir des types de structures Rust compatibles avec les structures C, vous pouvez utiliser l'`#[repr(C)]` attribut. Placer `#[repr(C)]` au-dessus d'une définition de structure demande à Rust de disposer les champs de la structure en mémoire de la même manière qu'un compilateur C disposerait le type de structure C analogue. Par exemple, `libgit2` le fichier d'en-tête `git2/errors.h` de définit la structure C suivante pour fournir des détails sur une erreur précédemment signalée :

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

Vous pouvez définir un type Rust avec une représentation identique comme suit :

```
use std::os::raw::{c_char, c_int};

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass:c_int
}
```

L'`#[repr(C)]` attribut n'affecte que la disposition de la structure elle-même, pas les représentations de ses champs individuels, donc pour correspondre à la structure C, chaque champ doit également utiliser le type C : `*const c_char` for `char *`, `c_int` for `int`, etc.

Dans ce cas particulier, l'`#[repr(C)]` attribut ne change probablement pas la disposition de `git_error`. Il n'y a vraiment pas trop de façons intéressantes de disposer un pointeur et un entier. Mais alors que C et C++ garantissent que les membres d'une structure apparaissent en mémoire dans l'ordre dans lequel ils sont déclarés, chacun à une adresse distincte, Rust réorganise les champs pour minimiser la taille globale de la structure et les types de taille nulle ne prennent pas de place. L'`#`

`[repr(C)]` attribut indique à Rust de suivre les règles de C pour le type donné.

Vous pouvez également utiliser `##[repr(C)]` pour contrôler la représentation des énumérations de style C :

```
##[repr(C)]
#[allow(non_camel_case_types)]
enum git_error_code {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
}
```

Normalement, Rust joue à toutes sortes de jeux lorsqu'il choisit comment représenter les énumérations. Par exemple, nous avons mentionné l'astuce que Rust utilise pour stocker `Option<&T>` en un seul mot (si `T` est dimensionné). Sans `##[repr(C)]`, Rust utiliserait un seul octet pour représenter l'`git_error_code` énumération ; avec `##[repr(C)]`, Rust utilise une valeur de la taille d'un C `int`, tout comme C le ferait.

Vous pouvez également demander à Rust de donner à une énumération la même représentation qu'un type entier. Commencer la définition précédente with `##[repr(i16)]` vous donnerait un type 16 bits avec la même représentation que l'énumération C++ suivante :

```
#include <stdint.h>

enum git_error_code: int16_t {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
};
```

Comme mentionné précédemment, `##[repr(C)]` s'applique également aux syndicats. Les champs des `##[repr(C)]` unions commencent toujours au premier bit de la mémoire de l'union, index 0.

Supposons que vous ayez une structure C qui utilise une union pour contenir certaines données et une valeur de balise pour indiquer quel champ de l'union doit être utilisé, similaire à une énumération Rust.

```

enum tag {
    FLOAT = 0,
    INT   = 1,
};

union number {
    float f;
    short i;
};

struct tagged_number {
    tag t;
    number n;
};

```

Le code Rust peut interagir avec cette structure en s'appliquant `#[repr(C)]` aux types enum, structure et union, et en utilisant une `match` instruction qui sélectionne un champ union dans une structure plus grande basée sur la balise :

```

#[repr(C)]
enum Tag {
    Float = 0,
    Int   = 1
}

#[repr(C)]
union FloatOrInt {
    f: f32,
    i:i32,
}

#[repr(C)]
struct Value {
    tag: Tag,
    union:FloatOrInt
}

fn is_zero(v: Value) -> bool {
    use self::Tag::*;

    unsafe {
        match v {
            Value { tag: Int, union: FloatOrInt { i: 0 } } => true,
            Value { tag: Float, union: FloatOrInt { f:num } } => (num == 0.0)
            _ => false
        }
    }
}

```

Même des structures complexes peuvent être facilement utilisées à travers la frontière FFI en utilisant ce type de technique.

Qui passe les cordes entre Rust et C sont un peu plus dures. C représente une chaîne sous la forme d'un pointeur vers un tableau de caractères, terminé par un caractère nul. Rust, d'autre part, stocke explicitement la longueur d'une chaîne, soit en tant que champ de `a`, `String` soit en tant que deuxième mot d'une référence fat `&str`. Les chaînes de rouille ne sont pas terminées par null ; en fait, ils peuvent inclure des caractères nuls dans leur contenu, comme tout autre caractère.

Cela signifie que vous ne pouvez pas emprunter une chaîne Rust en tant que chaîne C : si vous transmettez un pointeur de code C dans une chaîne Rust, il pourrait confondre un caractère nul intégré avec la fin de la chaîne ou courir à la fin à la recherche d'une terminaison null qui n'est pas là. Dans l'autre sens, vous pourrez peut-être emprunter une chaîne C en tant que Rust `&str`, tant que son contenu est bien formé en UTF-8.

Cette situation oblige effectivement Rust à traiter les chaînes C comme des types entièrement distincts de `String` et `&str`. Dans le `std::ffi` module, les types `CString` et représentent des tableaux d'octets à terminaison nulle possédés et empruntés. `cstr` Par rapport à `String` et `str`, les méthodes sur `CString` et `CStr` sont assez limitées, restreintes à la construction et à la conversion vers d'autres types. Nous montrerons ces types en action dans la section suivante.

Déclarer des fonctions étrangères et des variables

Un `extern` bloc déclare les fonctions ou des variables définies dans une autre bibliothèque avec laquelle l'exécutable Rust final sera lié. Par exemple, sur la plupart des plates-formes, chaque programme Rust est lié à la bibliothèque C standard, nous pouvons donc informer Rust de la `strlen` fonction de la bibliothèque C comme ceci :

```
use std::os::raw::c_char;

extern {
    fn strlen(s: *const c_char) -> usize;
}
```

Cela donne à Rust le nom et le type de la fonction, tout en laissant la définition à lier plus tard.

Rust suppose que les fonctions déclarées à l'intérieur `extern` des blocs utilisent les conventions C pour passer des arguments et accepter les valeurs de retour. Ils sont définis comme `unsafe` des fonctions. Ce sont les bons choix pour `strlen` : il s'agit bien d'une fonction C, et sa spécification en C nécessite que vous lui passiez un pointeur valide vers une chaîne correctement terminée, ce qui est un contrat que Rust ne peut pas appliquer. (Presque toute fonction qui prend un pointeur brut doit être `unsafe` : safe Rust peut construire des pointeurs bruts à partir d'entiers arbitraires, et déréférencer un tel pointeur serait un comportement indéfini.)

Avec ce `extern` bloc, on peut appeler `strlen` comme n'importe quelle autre fonction Rust, bien que son type le trahisse en touriste :

```
use std:: ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
    assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

La `CString::new` fonction construit une chaîne C terminée par null. Il vérifie d'abord dans son argument les caractères nuls intégrés, car ceux-ci ne peuvent pas être représentés dans une chaîne C, et renvoie une erreur s'il en trouve (d'où la nécessité `unwrap` du résultat). Sinon, il ajoute un octet nul à la fin et renvoie un `CString` propriétaire des caractères résultants.

Le coût `CString::new` dépend du type que vous lui passez. Il accepte tout ce qui implémente `Into<Vec<u8>>`. Passer `a &str` implique une allocation et une copie, car la conversion en `Vec<u8>` construit une copie allouée par tas de la chaîne que le vecteur doit posséder. Mais le passage d'une `String` valeur par consomme simplement la chaîne et prend le contrôle de son tampon, donc à moins que l'ajout du caractère nul ne force le redimensionnement du tampon, la conversion ne nécessite aucune copie de texte ni aucune allocation.

`CString` déréférence à `CStr`, dont la `as_ptr` méthode renvoie un `*const c_char` pointage au début de la chaîne. C'est le type qui `strlen` attend. Dans l'exemple, `strlen` parcourt la chaîne, trouve le caractère nul qui `CString::new` y est placé et renvoie la longueur, sous forme de nombre d'octets.

Vous pouvez également déclarer des variables globales dans des `extern` blocs. Les systèmes POSIX ont une variable globale nommée

environ qui contient les valeurs des variables d'environnement du processus. En C, il est déclaré :

```
extern char **environ;
```

En Rust, vous diriez :

```
use std:: ffi:: CStr;
use std:: os:: raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

Pour imprimer le premier élément de l'environnement, vous pouvez écrire :

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("first environment variable: {}",
                 var.to_string_lossy())
    }
}
```

Après s'être assuré environ d'avoir un premier élément, le code appelle `CStr::from_ptr` pour construire un `cstr` qui l'emprunte. La `to_string_lossy` méthode renvoie à `Cow<str>` : si la chaîne C contient de l'UTF-8 bien formé, le `Cow` emprunte son contenu en tant que à `&str`, sans compter l'octet nul de fin. Sinon, `to_string_lossy` fait une copie du texte dans le tas, remplace les séquences UTF-8 mal formées par le caractère de remplacement Unicode officiel `\u{fffd}`, et crée un propriétaire `Cow` à partir de cela. Dans tous les cas, le résultat implémente `Display`, vous pouvez donc l'imprimer avec le `{}` paramètre format.

Utiliser les fonctions des bibliothèques

Pour utiliser les fonctions fourni par une bibliothèque particulière, vous pouvez placer un `#[link]` attribut au-dessus du `extern` bloc qui nomme la bibliothèque avec laquelle Rust doit lier l'exécutable. Par exemple, voici un programme qui appelle `libgit2` l'initialisation de et les méthodes d'arrêt, mais ne fait rien d'autre :

```

use std:: os:: raw::c_int;

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() ->c_int;
}

fn main() {
    unsafe {
        git_libgit2_init();
        git_libgit2_shutdown();
    }
}

```

Le `extern` bloc déclare les fonctions externes comme précédemment. L'`#[link(name = "git2")]` attribut laisse une note dans la caisse à l'effet que, lorsque Rust crée l'exécutable final ou la bibliothèque partagée, il doit être lié à la `git2` bibliothèque. Rust utilise l'éditeur de liens système pour créer des exécutables ; sous Unix, cela passe l'argument `-lgit2` sur la ligne de commande de l'éditeur de liens ; sous Windows, ça passe `git2.LIB`.

`#[link]` les attributs fonctionnent également dans les caisses de bibliothèque. Lorsque vous construisez un programme qui dépend d'autres caisses, Cargo rassemble les notes de lien de l'ensemble du graphique de dépendance et les inclut toutes dans le lien final.

Dans cet exemple, si vous souhaitez suivre sur votre propre machine, vous devrez créer `libgit2` vous-même. Nous avons utilisé [libgit2](#) la version 0.25.1. Pour compiler `libgit2`, vous devrez installer l'outil de compilation CMake et le langage Python ; nous avons utilisé [CMake](#) version 3.8.0 et [Python](#) version 2.7.13.

Les instructions complètes pour la construction `libgit2` sont disponibles sur son site Web, mais elles sont suffisamment simples pour que nous montrons ici l'essentiel. Sous Linux, supposons que vous avez déjà décompressé le source de la bibliothèque dans le répertoire `/home/jimb/libgit2-0.25.1` :

```

$ cd/home/jimb/libgit2-0.25.1
$mkdir build
$ cd build
$cmake ..
$cmake --build .

```

Sous Linux, cela produit une bibliothèque partagée `/home/jimb/libgit2-0.25.1/build/libgit2.so`.0.25.1 avec le nid habituel de liens symboliques pointant vers elle, dont un nommé `libgit2.so`. Sur macOS, les résultats sont similaires, mais la bibliothèque est nommée `libgit2.dylib`.

Sous Windows, les choses sont également simples. Supposons que vous avez décompressé la source dans le répertoire `C:\Users\JimB\libgit2-0.25.1`. Dans une invite de commandes Visual Studio :

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

Ce sont les mêmes commandes que celles utilisées sous Linux, sauf que vous devez demander une version 64 bits lorsque vous exécutez CMake la première fois pour correspondre à votre compilateur Rust. (Si vous avez installé la chaîne d'outils Rust 32 bits, vous devez omettre l'`-A x64` indicateur de la première `cmake` commande.) Cela produit une bibliothèque d'importation `git2.LIB` et une bibliothèque de liens dynamiques `git2.DLL`, toutes deux dans le répertoire `C:\Users\JimB\libgit2-0.25.1\build\Debug`. (Les instructions restantes sont affichées pour Unix, sauf lorsque Windows est sensiblement différent.)

Créez le programme Rust dans un répertoire séparé :

```
$ cd /home/jimb
$cargo nouveau --bin git-toy
Created binary (application) `git-toy` package
```

Prenez le code montré précédemment et placez-le dans `src/main.rs`. Naturellement, si vous essayez de construire ceci, Rust n'a aucune idée d'où trouver ce `libgit2` que vous avez construit :

```
$ cd $course/de/fret
git-toy
Compiling git-toy v0.1.0 (/home/jimb/git-toy)
error: linking with `cc` failed: exit status: 1
|
= note: /usr/bin/ld: error: cannot find -lgit2
src/main.rs:11: error: undefined reference to 'git_libgit2_init'
src/main.rs:12: error: undefined reference to 'git_libgit2_shutdown'
collect2: error: ld returned 1 exit status
```

```
error: could not compile `git-toy` due to previous error
```

Vous pouvez indiquer à Rust où rechercher des bibliothèques en écrivant un *script de construction*, Code de rouille que Cargocompile et s'exécute au moment de la construction. Les scripts de construction peuvent faire toutes sortes de choses : générer du code dynamiquement, compiler du code C à inclure dans le crate, etc. Dans ce cas, tout ce dont vous avez besoin est d'ajouter un chemin de recherche de bibliothèque à la commande de lien de l'exécutable. Lorsque Cargo exécute le script de construction, il analyse la sortie du script de construction à la recherche d'informations de ce type, de sorte que le script de construction a simplement besoin d'imprimer la bonne magie sur sa sortie standard.

Pour créer votre script de construction, ajoutez un fichier nommé *build.rs* dans le même répertoire que le fichier *Cargo.toml*, avec le contenu suivant :

```
fn main() {
    println!(r"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/buil
}
```

C'est la bonne voie pour Linux ; sous Windows, vous modifieriez le chemin suivant le texte `native=` en `C:\Users\JimB\libgit2-0.25.1\build\Debug`. (Nous prenons quelques raccourcis pour que cet exemple reste simple ; dans une application réelle, vous devriez éviter d'utiliser des chemins absous dans votre script de construction. Nous citions la documentation qui montre comment le faire à la fin de cette section.)

Maintenant, vous pouvez presque exécuter le programme. Sur macOS, cela peut fonctionner immédiatement ; sur un système Linux, vous verrez probablement quelque chose comme ceci :

```
$ course de fret
Compiling git-toy v0.1.0 (/tmp/rustbook-transcript-tests/git-toy)
Finished dev [unoptimized + debuginfo] target(s)
    Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or directory
```

Cela signifie que, bien que Cargo ait réussi à lier l'exécutable à la bibliothèque, il ne sait pas où trouver la bibliothèque partagée au moment de l'exécution. Windows signale cet échec en faisant apparaître une boîte de

dialogue. Sous Linux, vous devez définir la `LD_LIBRARY_PATH` variable d'environnement :

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build : $LD_LIBRARY_PATH  
$course de cargaison  
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs  
    Running `target/debug/git-toy`
```

Sur macOS, vous devrez peut-être définir à la `DYLD_LIBRARY_PATH` place.

Sous Windows, vous devez définir la `PATH` variable d'environnement :

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug ; %PATH%  
>course de fret  
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs  
    Running `target/debug/git-toy`  
>
```

Naturellement, dans une application déployée, vous voudriez éviter d'avoir à définir des variables d'environnement juste pour trouver le code de votre bibliothèque. Une alternative consiste à lier statiquement la bibliothèque C dans votre crate. Cela copie les fichiers objets de la bibliothèque dans le fichier `.rlib` de la caisse , ainsi que les fichiers objets et les métadonnées du code Rust de la caisse. L'ensemble de la collection participe alors au lien final.

C'est une convention Cargo qu'une caisse qui donne accès à une bibliothèque C doit être nommée `LIB-sys` , où `LIB` est le nom de la bibliothèque C. Une `-sys` caisse ne doit contenir que la bibliothèque liée statiquement et les modules Rust contenant des `extern` blocs et des définitions de type. Les interfaces de niveau supérieur appartiennent alors à des caisses qui dépendent de la `-sys` caisse. Cela permet à plusieurs caisses en amont de dépendre de la même `-sys` caisse, en supposant qu'il existe une seule version de la `-sys` caisse qui répond aux besoins de chacun.

Pour plus de détails sur la prise en charge par Cargo des scripts de construction et de la liaison avec les bibliothèques système, consultez [la documentation en ligne de Cargo](#) . Il montre comment éviter les chemins absolus dans les scripts de construction, contrôler les drapeaux de compilation, utiliser des outils comme `pkg-config` , etc. La `git2-rs` caisse fournit également de bons exemples à imiter; son script de construction gère certaines situations complexes.

Une interface brute vers libgit2

Figurant comment bien l'utiliser libgit2 se décompose en deux questions :

- Que faut-il pour utiliser les libgit2 fonctions de Rust ?
- Comment pouvons-nous construire une interface Rust sûre autour d'eux ?

Nous allons répondre à ces questions une par une. Dans cette section, nous allons écrire un programme qui est essentiellement un seul `unsafe` bloc géant rempli de code Rust non idiomatique, reflétant le conflit des systèmes de types et des conventions inhérent au mélange de langages. Nous l'appellerons l' interface *brute* . Le code sera désordonné, mais il expliquera clairement toutes les étapes qui doivent se produire pour que le code Rust utilise libgit2 .

Ensuite, dans la section suivante, nous construirons une interface sécurisée libgit2 qui utilisera les types de Rust en appliquant les règles libgit2 imposées à ses utilisateurs. Heureusement, libgit2 est une bibliothèque C exceptionnellement bien conçue, de sorte que les questions que les exigences de sécurité de Rust nous obligent à poser ont toutes de bonnes réponses, et nous pouvons construire une interface Rust idiomatique sans `unsafe` fonctions.

Le programme que nous allons écrire est très simple : il prend un chemin comme argument de ligne de commande, y ouvre le référentiel Git et affiche le commit principal. Mais cela suffit pour illustrer les stratégies clés pour créer des interfaces Rust sûres et idiomatiques.

Pour l'interface brute, le programme finira par avoir besoin d'une collection de fonctions et de types un peu plus grande libgit2 que celle que nous utilisions auparavant, il est donc logique de déplacer le `extern` bloc dans son propre module. Nous allons créer un fichier nommé `raw.rs` dans `git-toy/src` dont le contenu est le suivant :

```
#![allow(non_camel_case_types)]  
  
use std::os::raw::{c_int, c_char, c_uchar};  
  
#[link(name = "git2")]  
extern {  
    pub fn git_libgit2_init() -> c_int;  
    pub fn git_libgit2_shutdown() -> c_int;  
    pub fn giterr_last() -> *const git_error;
```

```

pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) -> c_int;
pub fn git_repository_free(repo: *mut git_repository);

pub fn git_reference_name_to_id(out: *mut git_oid,
                                 repo: *mut git_repository,
                                 reference: *const c_char) ->c_int;

pub fn git_commit_lookup(out: *mut *mut git_commit,
                         repo: *mut git_repository,
                         id: *const git_oid) ->c_int;

pub fn git_commit_author(commit: *const git_commit) -> *const git_signature;
pub fn git_commit_message(commit: *const git_commit) -> *const c_char;
pub fn git_commit_free(commit: *mut git_commit);
}

#[repr(C)] pub struct git_repository { _private: [u8; 0] }
#[repr(C)] pub struct git_commit { _private:[u8; 0] }

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass:c_int
}

pub const GIT_OID_RAWSZ:usize = 20;

#[repr(C)]
pub struct git_oid {
    pub id:[c_uchar; GIT_OID_RAWSZ]
}

pub type git_time_t = i64;

#[repr(C)]
pub struct git_time {
    pub time: git_time_t,
    pub offset:c_int
}

#[repr(C)]
pub struct git_signature {
    pub name: *const c_char,
    pub email: *const c_char,
    pub when:git_time
}

```

Ici, chaque élément est modélisé sur une déclaration libgit2 des propres fichiers d'en-tête de . Par exemple, *libgit2-0.25.1/include/git2/repository.h* inclut cette déclaration :

```
extern int git_repository_open(git_repository **out, const char *path);
```

Cette fonction essaie d'ouvrir le référentiel Git sur `path`. Si tout se passe bien, il crée un `git_repository` objet et stocke un pointeur vers celui-ci à l'emplacement pointé par `out`. La déclaration Rust équivalente est la suivante :

```
pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) ->c_int;
```

Les `libgit2` fichiers d'en-tête publics définissent le `git_repository` type en tant que `typedef` pour un type de structure incomplet :

```
typedef struct git_repository git_repository;
```

Étant donné que les détails de ce type sont privés pour la bibliothèque, les en-têtes publics ne définissent jamais `struct git_repository`, garantissant que les utilisateurs de la bibliothèque ne peuvent jamais créer eux-mêmes une instance de ce type. Voici un analogue possible d'un type de structure incomplet dans Rust :

```
#[repr(C)] pub struct git_repository { _private:[u8; 0] }
```

Il s'agit d'un type `struct` contenant un tableau sans éléments. Puisque le `_private` champ n'est pas `pub`, les valeurs de ce type ne peuvent pas être construites en dehors de ce module, qui est parfait en tant que reflet d'un type C qui ne `libgit2` devrait jamais être construit, et qui est manipulé uniquement par des pointeurs bruts.

Écrire de gros `extern` blocs à la main peut être une corvée. Si vous créez une interface Rust vers une bibliothèque C complexe, vous pouvez essayer d'utiliser le `bindgen` crate, qui a des fonctions que vous pouvez utiliser à partir de votre script de génération pour analyser les fichiers d'en-tête C et générer automatiquement les déclarations Rust correspondantes. Nous n'avons pas d'espace pour montrer `bindgen` en action ici, mais [bindgen la page de crates.io](#) inclut des liens vers sa documentation.

Ensuite, nous *réécrirons* complètement `main.rs`. Tout d'abord, nous devons déclarer le `raw` module :

```
mod raw;
```

Selon `libgit2` les conventions de , les fonctions faillibles renvoient un code entier qui est positif ou nul en cas de succès et négatif en cas d'échec. Si une erreur se produit, la `giterr_last` fonction renverra un pointeur vers une `git_error` structure fournissant plus de détails sur ce qui s'est mal passé. `libgit2` possède cette structure, nous n'avons donc pas besoin de la libérer nous-mêmes, mais elle pourrait être écrasée par le prochain appel à la bibliothèque que nous ferons. Une interface Rust appropriée utiliserait `Result`, mais dans la version brute, nous voulons utiliser les `libgit2` fonctions telles qu'elles sont, nous devrons donc lancer notre propre fonction pour gérer les erreurs :

```
use std::ffi:: CStr;
use std::os:: raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
    if status < 0 {
        unsafe {
            let error = &*raw:: giterr_last();
            println!("error while {}: {} ({})", activity,
                    CStr:: from_ptr(error.message).to_string_lossy(),
                    error.klass);
            std:: process::exit(1);
        }
    }

    status
}
```

Nous allons utiliser cette fonction pour vérifier les résultats d'`libgit2` appels comme celui-ci :

```
check("initializing library", raw::git_libgit2_init());
```

Cela utilise les mêmes `cstr` méthodes que celles utilisées précédemment: `from_ptr` construire le `cstr` à partir d'une chaîne C et `to_string_lossy` le transformer en quelque chose que Rust peut imprimer.

Ensuite, nous avons besoin d'une fonction pour imprimer un commit :

```
unsafe fn show_commit(commit: *const raw:: git_commit) {
    let author = raw::git_commit_author(commit);

    let name = CStr:: from_ptr((*author).name).to_string_lossy();
    let email = CStr:: from_ptr((*author).email).to_string_lossy();
    println!("{} <{}>\n", name, email);
```

```

    let message = raw::: git_commit_message(commit);
    println!("{}", CStr:::from_ptr(message).to_string_lossy());
}

```

Étant donné un pointeur vers a `git_commit`, `show_commit` appelle `git_commit_author` et `git_commit_message` pour récupérer les informations dont il a besoin. Ces deux fonctions suivent une convention que la `libgit2` documentation explique comme suit :

Si une fonction renvoie un objet comme valeur de retour, cette fonction est un getter et la durée de vie de l'objet est liée à l'objet parent.

En termes Rust, `author` et `message` sont empruntés à `commit`:

`show_commit` n'a pas besoin de les libérer lui-même, mais il ne doit pas les conserver après avoir `commit` été libéré. Étant donné que cette API utilise des pointeurs bruts, Rust ne vérifiera pas leur durée de vie pour nous : si nous créons accidentellement des pointeurs pendents, nous ne le saurons probablement pas avant que le programme ne plante.

Le code précédent suppose que ces champs contiennent du texte UTF-8, ce qui n'est pas toujours correct. Git autorise également d'autres encodages. Interpréter correctement ces chaînes impliquerait probablement l'utilisation de la `encoding` caisse. Par souci de brièveté, nous passerons sous silence ces questions ici.

La fonction de notre programme se `main` lit comme suit :

```

use std:: ffi:: CString;
use std:: mem;
use std:: ptr;
use std:: os:: raw::c_char;

fn main() {
    let path = std:: env:: args().skip(1).next()
        .expect("usage: git-toy PATH");
    let path = CString:::new(path)
        .expect("path contains null characters");

    unsafe {
        check("initializing library", raw:::git_libgit2_init());

        let mut repo = ptr::: null_mut();
        check("opening repository",
              raw:::git_repository_open(&mut repo, path.as_ptr()));

        let c_name = b"HEAD\0".as_ptr() as *const c_char;
        let oid = {

```

```

        let mut oid = mem:: MaybeUninit:: uninit();
        check("looking up HEAD",
              raw::git_reference_name_to_id(oid.as_mut_ptr(), repo, c_n
        oid.assume_init()
    };

    let mut commit = ptr:: null_mut();
    check("looking up commit",
          raw::git_commit_lookup(&mut commit, repo, &oid));

    show_commit(commit);

    raw::git_commit_free(commit);

    raw::git_repository_free(repo);

    check("shutting down library", raw::git_libgit2_shutdown());
}
}

```

Cela commence par le code pour gérer l'argument path et initialiser la bibliothèque, ce que nous avons déjà vu. Le premier code roman est celui-ci :

```

let mut repo = ptr:: null_mut();
check("opening repository",
      raw::git_repository_open(&mut repo, path.as_ptr()));

```

L'appel à `git_repository_open` essaie d'ouvrir le dépôt Git au chemin donné. S'il réussit, il lui alloue un nouvel `git_repository` objet et `repo` pointe vers celui-ci. Rust constraint implicitement les références en pointeurs bruts, donc le passage `&mut repo` ici fournit `*mut *mut git_repository` l'appel attendu.

Cela montre une autre `libgit2` convention utilisée (à partir de la `libgit2` documentation) :

Les objets renvoyés via le premier argument en tant que pointeur à pointeur appartiennent à l'appelant et il est responsable de leur libération.

En termes de Rust, des fonctions telles que `git_repository_open` transmettre la propriété de la nouvelle valeur à l'appelant.

Ensuite, considérez le code qui recherche le hachage d'objet du commit principal actuel du référentiel :

```

let oid = {
    let mut oid = mem:: MaybeUninit:: uninit();
    check("looking up HEAD",
          raw::git_reference_name_to_id(oid.as_mut_ptr(), repo, c_name));
    oid.assume_init()
};

```

Le `git_oid` type stocke un identifiant d'objet, un code de hachage de 160 bits que Git utilise en interne (et dans son interface utilisateur conviviale) pour identifier les validations, les versions individuelles des fichiers, etc. Cet appel à `git_reference_name_to_id` recherche l'identifiant d'objet du "HEAD" commit en cours.

En C, il est parfaitement normal d'initialiser une variable en lui passant un pointeur vers une fonction qui remplit sa valeur ; c'est ainsi `git_reference_name_to_id` qu'il s'attend à traiter son premier argument. Mais Rust ne nous laissera pas emprunter une référence à une variable non initialisée. On pourrait initialiser `oid` avec des zéros, mais c'est du gâchis : toute valeur qui y est stockée sera simplement écrasée.

Il est possible de demander à Rust de nous donner de la mémoire non initialisée, mais comme la lecture de mémoire non initialisée à tout moment est un comportement instantané et indéfini, Rust fournit une abstraction, `MaybeUninit`, pour faciliter son utilisation. `MaybeUninit<T>` indique au compilateur de réservé suffisamment de mémoire pour votre type `T`, mais de ne pas y toucher jusqu'à ce que vous disiez que vous pouvez le faire en toute sécurité. Bien que cette mémoire appartienne à `MaybeUninit`, le compilateur évitera également certaines optimisations qui pourraient autrement provoquer un comportement indéfini même sans aucun accès explicite à la mémoire non initialisée dans votre code.

`MaybeUninit` fournit une méthode, `as_mut_ptr()`, qui produit un `*mut T` pointage vers la mémoire potentiellement non initialisée qu'elle encapsule. En transmettant ce pointeur à une fonction étrangère qui initialise la mémoire, puis en appelant la méthode unsafe `assume_init` sur le `MaybeUninit` pour produire un complètement initialisé `T`, vous pouvez éviter un comportement indéfini sans la surcharge supplémentaire résultant de l'initialisation et de la suppression immédiate d'une valeur. `assume_init` n'est pas sûr car l'appeler sur un `MaybeUninit` sans être certain que la mémoire est réellement initialisée provoquera immédiatement un comportement indéfini.

Dans ce cas, il est sûr car `git_reference_name_to_id` initialise la mémoire appartenant au `MaybeUninit`. Nous pourrions également utiliser `MaybeUninit` pour les variables `repo` et `c_name`, mais comme ce ne sont que

des mots simples, nous allons simplement de l'avant et les initialisons à null : commit

```
let mut commit = ptr::null_mut();
check("looking up commit",
      raw::git_commit_lookup(&mut commit, repo, &oid));
```

Cela prend l'identifiant d'objet du commit et recherche le commit réel, en stockant un `git_commit` pointeur en cas `commit` de succès.

Le reste de la `main` fonction devrait être explicite. Il appelle la `show_commit` fonction définie précédemment, libère les objets de validation et de référentiel et arrête la bibliothèque.

Maintenant, nous pouvons essayer le programme sur n'importe quel référentiel Git prêt à portée de main:

```
$ cargo run /home/jimb/rbattle
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/git-toy /home/jimb/rbattle`
Jim Blandy <jimb@red-bean.com>

Animate goop a bit.
```

Une interface sécurisée pour libgit2

L'interface brutale `libgit2` est un exemple parfait d'une fonctionnalité non sécurisée : elle peut certainement être utilisée correctement (comme nous le faisons ici, pour autant que nous le sachions), mais Rust ne peut pas appliquer les règles que vous devez suivre. Concevoir une API sûre pour une bibliothèque comme celle-ci consiste à identifier toutes ces règles, puis à trouver des moyens de transformer toute violation de celles-ci en une erreur de type ou de vérification d'emprunt.

Voici donc `libgit2` les règles de pour les fonctionnalités utilisées par le programme :

- Vous devez appeler `git_libgit2_init` avant d'utiliser toute autre fonction de la bibliothèque. Vous ne devez utiliser aucune fonction de bibliothèque après avoir appelé `git_libgit2_shutdown`.
- Toutes les valeurs transmises aux `libgit2` fonctions doivent être entièrement initialisées, à l'exception des paramètres de sortie.
- Lorsqu'un appel échoue, les paramètres de sortie transmis pour conserver les résultats de l'appel ne sont pas initialisés et vous ne de-

vez pas utiliser leurs valeurs.

- Un `git_commit` objet fait référence à l' `git_repository` objet dont il est dérivé, de sorte que le premier ne doit pas survivre au second.
(Ceci n'est pas précisé dans la `libgit2` documentation ; nous l'avons déduit de la présence de certaines fonctions dans l'interface, puis nous l'avons vérifié en lisant le code source.)
- De même, a `git_signature` est toujours emprunté à un donné `git_commit`, et le premier ne doit pas survivre au second. (La documentation couvre ce cas.)
- Le message associé à un commit ainsi que le nom et l'adresse e-mail de l'auteur sont tous empruntés au commit et ne doivent pas être utilisés après la libération du commit.
- Une fois qu'un `libgit2` objet a été libéré, il ne doit plus jamais être utilisé.

Il s'avère que vous pouvez créer une interface Rust `libgit2` qui applique toutes ces règles, soit via le système de type de Rust, soit en gérant les détails en interne.

Avant de commencer, restructurons un peu le projet. Nous aimerions avoir un `git` module qui exporte l'interface sécurisée, dont l'interface brute du programme précédent est un sous-module privé.

L'ensemble de l'arborescence des sources ressemblera à ceci :

```
git-toy/
├── Cargo.toml
├── build.rs
└── src/
    ├── main.rs
    └── git/
        ├── mod.rs
        └── raw.rs
```

En suivant les règles que nous avons expliquées dans "[Modules in Separate Files](#)" , la source du `git` module apparaît dans `git/mod.rs` et la source de son `git::raw` sous-module va dans `git/raw.rs` .

Encore une fois, nous allons réécrire entièrement `main.rs`. Il doit commencer par une déclaration du `git` module :

```
mod git;
```

Ensuite, nous devrons créer le sous-répertoire `git` et y déplacer `raw.rs` :

```
$ cd/home/jimb/git-toy  
$ mkdir src/git  
$ mv src/raw.rs src/git/raw.rs
```

Le `git` module doit déclarer son `raw` sous-module. Le fichier `src/git/mod.rs` doit indiquer :

```
mod raw;
```

Comme ce n'est pas `pub`, ce sous-module n'est pas visible pour le programme principal.

Dans quelques instants, nous aurons besoin d'utiliser certaines fonctions de la `libc` caisse, nous devons donc ajouter une dépendance dans `Cargo.toml`. Le fichier complet lit maintenant :

```
[forfait]  
nom = "git-jouet"  
version = "0.1.0"  
auteurs = ["Vous <vous@example.com>"]  
édition = "2021"  
  
[dépendances]  
libc = "0.2"
```

Maintenant que nous avons restructuré nos modules, considérons la gestion des erreurs. Même `libgit2` la fonction d'initialisation de peut renvoyer un code d'erreur, nous devrons donc régler ce problème avant de pouvoir commencer. Une interface Rust idiomatique a besoin de son propre `Error` type qui capture le `libgit2` code d'échec ainsi que le message d'erreur et la classe de `giterr_last`. Un type d'erreur approprié doit implémenter les traits habituels `Error`, `Debug` et `Display`. Ensuite, il a besoin de son propre `Result` type qui utilise ce `Error` type. Voici les définitions nécessaires dans `src/git/mod.rs` :

```
use std::error;  
use std::fmt;  
use std::result;  
  
#[derive(Debug)]  
pub struct Error {  
    code: i32,  
    message: String,  
    class:i32  
}
```

```

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(), fmt::Error> {
        // Displaying an `Error` simply displays the message from libgit2.
        self.message.fmt(f)
    }
}

impl error::Error for Error { }

pub type Result<T> = result::Result<T, Error>;

```

Pour vérifier le résultat des appels bruts à la bibliothèque, le module a besoin d'une fonction qui transforme un libgit2 code de retour en un Result :

```

use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) ->Result<c_int> {
    if code >= 0 {
        return Ok(code);
    }

    unsafe {
        let error = raw::giterr_last();

        // libgit2 ensures that (*error).message is always non-null and null
        // terminated, so this call is safe.
        let message = CStr::from_ptr((*error).message)
            .to_string_lossy()
            .into_owned();

        Err(Error {
            code: code as i32,
            message,
            class: (*error).klass as i32
        })
    }
}

```

La principale différence entre ceci et la `check` fonction de la version brute est que cela construit une `Error` valeur au lieu d'afficher un message d'erreur et de quitter immédiatement.

Nous sommes maintenant prêts à nous attaquer à l' libgit2 initialisation. L'interface sécurisée fournira un `Repository` type qui représente un référentiel Git ouvert, avec des méthodes pour résoudre les références, rechercher des commits, etc. En continuant dans `git/mod.rs`, voici la définition de `Repository` :

```

/// A Git repository.
pub struct Repository {
    // This must always be a pointer to a live `git_repository` structure.
    // No other `Repository` may point to it.
    raw: *mut raw::git_repository
}

```

Le champ `Repository` de `A` n'est `raw` pas public. Étant donné que seul le code de ce module peut accéder au `raw::git_repository` pointeur, obtenir ce module correctement devrait garantir que le pointeur est toujours utilisé correctement.

Si la seule façon de créer un `Repository` est d'ouvrir avec succès un nouveau référentiel Git, cela garantira que chacun `Repository` pointe vers un objet distinct `git_repository` :

```

use std::path::Path;
use std::ptr;

impl Repository {
    pub fn open<P: AsRef<Path>>(path: P) ->Result<Repository> {
        ensure_initialized();

        let path = path_to_cstring(path.as_ref())?;
        let mut repo = ptr::null_mut();
        unsafe {
            check(raw::git_repository_open(&mut repo, path.as_ptr()))?;
        }
        Ok(Repository { raw:repo })
    }
}

```

Étant donné que la seule façon de faire quoi que ce soit avec l'interface sécurisée est de commencer par une `Repository` valeur, et `Repository::open` commence par un appel à `ensure_initialized`, nous pouvons être sûrs que `ensure_initialized` sera appelé avant toute `libgit2` fonction. Sa définition est la suivante :

```

fn ensure_initialized() {
    static ONCE: std::sync::Once = std::sync::Once::new();
    ONCE.call_once(|| {
        unsafe {
            check(raw::git_libgit2_init())
                .expect("initializing libgit2 failed");
            assert_eq!(libc::atexit(shutdown), 0);
        }
    });
}

```

```

    extern fn shutdown() {
        unsafe {
            if let Err(e) = check(raw::: git_libgit2_shutdown()) {
                eprintln!("shutting down libgit2 failed: {}", e);
                std::process::abort();
            }
        }
    }
}

```

Le `std::sync::Once` type permet d'exécuter le code d'initialisation de manière thread-safe. Seul le premier thread à appeler `ONCE.call_once` exécute la fermeture donnée. Tous les appels suivants, par ce thread ou tout autre, bloquent jusqu'à ce que le premier soit terminé, puis reviennent immédiatement, sans exécuter à nouveau la fermeture. Une fois la fermeture terminée, l'appel `ONCE.call_once` est bon marché, ne nécessitant rien de plus qu'une charge atomique d'un indicateur stocké dans `ONCE`.

Dans le code précédent, la fermeture d'initialisation appelle `git_libgit2_init` et vérifie le résultat. Il lance un peu et utilise juste `.expect` pour s'assurer que l'initialisation a réussi, au lieu d'essayer de propager les erreurs à l'appelant.

Pour s'assurer que le programme appelle `git_libgit2_shutdown`, la fermeture d'initialisation utilise la `atexit` fonction de la bibliothèque C, qui prend un pointeur vers une fonction à invoquer avant que le processus ne se termine. Les fermetures Rust ne peuvent pas servir de pointeurs de fonction C : une fermeture est une valeur d'un type anonyme portant les valeurs de toutes les variables qu'elle capture ou y fait référence ; un pointeur de fonction C n'est qu'un pointeur. Cependant, les types Rust fonctionnent bien, tant que vous les déclarez `extern` afin que Rust sache utiliser les conventions d'appel C. La fonction locale fait `shutdown` l'affaire et garantit une `libgit2` fermeture correcte.

Dans "["Unwinding"](#)", nous avons mentionné qu'il s'agit d'un comportement indéfini pour une panique de franchir les frontières linguistiques. L'appel de `atexit` à `shutdown` est une telle limite, il est donc essentiel de `shutdown` ne pas paniquer. C'est pourquoi `shutdown` ne peut pas simplement utiliser `.expect` pour gérer les erreurs signalées à partir de `raw:::git_libgit2_shutdown`. Au lieu de cela, il doit signaler l'erreur et terminer le processus lui-même. POSIX interdit d'appeler `exit` dans un `atexit` gestionnaire, donc `shutdown` appelle `std::process::abort` pour terminer le programme brusquement.

Il pourrait être possible de s'arranger pour appeler `git_libgit2_shutdown` plus tôt, par exemple, lorsque la dernière `Repository` valeur est supprimée. Mais peu importe comment nous organisons les choses, l'appel `git_libgit2_shutdown` doit être la responsabilité de l'API sécurisée. Au moment où elle est appelée, tous les objets existants `libgit2` deviennent dangereux à utiliser, donc une API sûre ne doit pas exposer directement cette fonction.

Le `Repository` pointeur brut d'un doit toujours pointer vers un `git_repository` objet actif. Cela implique que la seule façon de fermer un dépôt est de supprimer la `Repository` valeur qui le possède :

```
impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {
            raw::git_repository_free(self.raw);
        }
    }
}
```

En n'appelant `git_repository_free` que lorsque le pointeur unique vers le `raw::git_repository` est sur le point de disparaître, le `Repository` type garantit également que le pointeur ne sera jamais utilisé après sa libération.

La `Repository::open` méthode utilise une fonction privée appelée `path_to_cstring`, qui a deux définitions, une pour les systèmes de type Unix et une pour Windows :

```
use std:: ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // The `as_bytes` method exists only on Unix-like systems.
    use std:: os:: unix:: ffi::OsStrExt;

    Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // Try to convert to UTF-8. If this fails, libgit2 can't handle the path
    // anyway.
    match path.to_str() {
        Some(s) => Ok(CString::new(s)?),
        None => {
            let message = format!("Couldn't convert path '{}' to UTF-8",
                                  path.display());
            Err(error::Error::new(message))
        }
    }
}
```

```

        Err(message.into())
    }
}

```

L' `libgit2` interface rend ce code un peu délicat. Sur toutes les plates-formes, `libgit2` accepte les chemins en tant que chaînes C terminées par un caractère nul. Sous Windows, `libgit2` suppose que ces chaînes C contiennent de l'UTF-8 bien formé et les convertit en interne en chemins 16 bits dont Windows a réellement besoin. Cela fonctionne généralement, mais ce n'est pas idéal. Windows autorise les noms de fichiers qui ne sont pas bien formés en Unicode et ne peuvent donc pas être représentés en UTF-8. Si vous avez un tel fichier, il est impossible de passer son nom à `libgit2`.

Dans Rust, la représentation correcte d'un chemin de système de fichiers est un `std::path::Path`, soigneusement conçu pour gérer tout chemin pouvant apparaître sous Windows ou POSIX. Cela signifie qu'il existe des `Path` valeurs sous Windows que l'on ne peut pas transmettre à `libgit2`, car elles ne sont pas bien formées en UTF-8. Ainsi, bien que `path_to_cstring` le comportement de soit loin d'être idéal, c'est en fait le mieux que nous puissions faire compte tenu `libgit2` de l'interface de .

Les deux `path_to_cstring` définitions qui viennent d'être présentées reposent sur des conversions vers notre `Error` type : l' `? opérateur` tente de telles conversions et la version Windows appelle explicitement `.into()`. Ces conversions sont banales :

```

impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class:0 }
    }
}

// NulError is what `CString::new` returns if a string
// has embedded zero bytes.
impl From<std::ffi::NulError> for Error {
    fn from(e: std::ffi::NulError) -> Error {
        Error { code: -1, message: e.to_string(), class:0 }
    }
}

```

Voyons ensuite comment résoudre une référence Git en un identifiant d'objet. Puisqu'un identifiant d'objet n'est qu'une valeur de hachage de 20 octets, il est parfaitement acceptable de l'exposer dans l'API sécurisée :

```

/// The identifier of some sort of object stored in the Git object
/// database: a commit, tree, blob, tag, etc. This is a wide hash of the
/// object's contents.
pub struct Oid {
    pub raw: raw::git_oid
}

```

Nous allons ajouter une méthode `Repository` pour effectuer la recherche :

```

use std::mem;
use std::os::raw::c_char;

impl Repository {
    pub fn reference_name_to_id(&self, name: &str) -> Result<Oid> {
        let name = CString::new(name)?;
        unsafe {
            let oid = {
                let mut oid = mem::MaybeUninit::uninit();
                check(raw::git_reference_name_to_id(
                    oid.as_mut_ptr(), self.raw,
                    name.as_ptr() as *const c_char))?;
                oid.assume_init()
            };
            Ok(Oid { raw:oid })
        }
    }
}

```

Bien qu'elle `oid` ne soit pas initialisée lorsque la recherche échoue, cette fonction garantit que son appelant ne pourra jamais voir la valeur non initialisée simplement en suivant l' `Result` idiome de Rust : soit l'appelant obtient un `Ok` portant une valeur correctement initialisée `Oid`, soit il obtient un `Err`.

Ensuite, le module a besoin d'un moyen de récupérer les commits du référentiel. Nous allons définir un `Commit` type comme suit :

```

use std::marker::PhantomData;

pub struct Commit<'repo> {
    // This must always be a pointer to a usable `git_commit` structure.
    raw: *mut raw::git_commit,
    _marker: PhantomData<&'repo Repository>
}

```

Comme nous l'avons mentionné précédemment, un `git_commit` objet ne doit jamais survivre à l' `git_repository` objet à partir duquel il a été récupéré. Les durées de vie de Rust permettent au code de capturer précisément cette règle.

L' `RefWithFlag` exemple précédent dans ce chapitre utilisait un `PhantomData` champ pour indiquer à Rust de traiter un type comme s'il contenait une référence avec une durée de vie donnée, même si le type ne contenait apparemment aucune référence de ce type. Le `Commit` type doit faire quelque chose de similaire. Dans ce cas, le `_marker` type du champ est `PhantomData<& 'repo Repository>`, indiquant que Rust doit traiter `Commit<'repo>` comme s'il contenait une référence avec une durée de vie '`'repo` à certains `Repository`.

La méthode pour rechercher un commit est la suivante :

```
impl Repository {
    pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
        let mut commit = ptr::null_mut();
        unsafe {
            check(raw::git_commit_lookup(&mut commit, self.raw, &oid.raw))?
        }
        Ok(Commit { raw: commit, _marker: PhantomData })
    }
}
```

Comment cela relie-t-il la `Commit` durée de vie de 's à celle de `Repository`'s? La signature de `find_commit` omet les durées de vie des références concernées conformément aux règles décrites dans [« Omettre les paramètres de durée de vie »](#). Si nous devions écrire les durées de vie, la signature complète se lirait :

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
->Result<Commit<'repo>>
```

C'est exactement ce que nous voulons : Rust traite le retour `Commit` comme s'il empruntait quelque chose à `self`, qui est le `Repository`.

Quand a `Commit` est lâché, il doit libérer son `raw::git_commit`:

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

```
    }
}
```

À partir d'un `Commit`, vous pouvez emprunter un `Signature` (un nom et une adresse e-mail) et le texte du message de validation :

```
impl<'repo> Commit<'repo> {
    pub fn author(&self) -> Signature {
        unsafe {
            Signature {
                raw: raw::git_commit_author(self.raw),
                _marker: PhantomData
            }
        }
    }

    pub fn message(&self) -> Option<&str> {
        unsafe {
            let message = raw::git_commit_message(self.raw);
            char_ptr_to_str(self, message)
        }
    }
}
```

Voici le `Signature` genre :

```
pub struct Signature<'text> {
    raw: *const raw::git_signature,
    _marker: PhantomData<&'text str>
}
```

Un `git_signature` objet emprunte toujours son texte ailleurs ; en particulier, les signatures renvoyées par `git_commit_author` empruntent leur texte au `git_commit`. Ainsi, notre type de sécurité `Signature` inclut a `PhantomData<&'text str>` pour dire à Rust de se comporter comme s'il contenait un `&str` avec une durée de vie de `'text`. Comme précédemment, `Commit::author` relie bien cette `'text` durée de vie du `Signature` il revient à celle du `Commit` sans qu'on ait besoin d'écrire quoi que ce soit. La `Commit::message` méthode fait de même avec le `Option<&str>` maintien du message de validation.

A `Signature` inclut des méthodes pour récupérer le nom et l'adresse e-mail de l'auteur :

```
impl<'text> Signature<'text> {
    /// Return the author's name as a `&str`,
    /// or `None` if it is not well-formed UTF-8.
```

```

pub fn name(&self) ->Option<&str> {
    unsafe {
        char_ptr_to_str(self, (*self.raw).name)
    }
}

/// Return the author's email as a `&str`,
/// or `None` if it is not well-formed UTF-8.
pub fn email(&self) ->Option<&str> {
    unsafe {
        char_ptr_to_str(self, (*self.raw).email)
    }
}
}

```

Les méthodes précédentes dépendent d'une fonction d'utilité privée `char_ptr_to_str`:

```

/// Try to borrow a `&str` from `ptr`, given that `ptr` may be null or
/// refer to ill-formed UTF-8. Give the result a lifetime as if it were
/// borrowed from `'_owner`.
///
/// Safety: if `ptr` is non-null, it must point to a null-terminated C
/// string that is safe to access for at least as long as the lifetime of
/// `'_owner`.
unsafe fn char_ptr_to_str<'T>(_owner: &T, ptr: *const c_char) -> Option<&str>
{
    if ptr.is_null() {
        return None;
    } else {
        CStr::from_ptr(ptr).to_str().ok()
    }
}

```

La `_owner` valeur du paramètre n'est jamais utilisée, mais sa durée de vie l'est. Rendre explicites les durées de vie dans la signature de cette fonction nous donne :

```

fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
->Option<&'o str>

```

La `CStr::from_ptr` fonction renvoie un `&cstr` dont la durée de vie est totalement illimitée, puisqu'il a été emprunté à un pointeur brut déréférencé. Les durées de vie illimitées sont presque toujours inexactes, il est donc bon de les contraindre dès que possible. L'inclusion du `_owner` paramètre oblige Rust à attribuer sa durée de vie au type de la valeur de retour, afin que les appelants puissent recevoir une référence délimitée plus précisément.

Il n'est pas clair d'après la `libgit2` documentation si un `git_signature`'s `email` et des `author` pointeurs peuvent être nuls, bien que la documentation `libgit2` soit assez bonne. Vos auteurs ont creusé dans le code source pendant un certain temps sans pouvoir se persuader d'une manière ou d'une autre et ont finalement décidé qu'il `char_ptr_to_str` valait mieux se préparer aux pointeurs nuls au cas où. Dans Rust, ce genre de question est répondu immédiatement par le type : si c'est `&str`, vous pouvez compter sur la chaîne pour être là ; si c'est le cas `Option<&str>`, c'est facultatif.

Enfin, nous avons fourni des interfaces sécurisées pour toutes les fonctionnalités dont nous avons besoin. La nouvelle `main` fonction dans `src/main.rs` est un peu allégée et ressemble à du vrai code Rust :

```
fn main() {
    let path = std::env::args_os().skip(1).next()
        .expect("usage: git-toy PATH");

    let repo = git::Repository::open(&path)
        .expect("opening repository");

    let commit_oid = repo.reference_name_to_id("HEAD")
        .expect("looking up 'HEAD' reference");

    let commit = repo.find_commit(&commit_oid)
        .expect("looking up commit");

    let author = commit.author();
    println!("{} <{}>\n",
            author.name().unwrap_or("(none)"),
            author.email().unwrap_or("none"));

    println!("{}", commit.message().unwrap_or("(none)"));
}
```

Dans ce chapitre, nous sommes passés d'interfaces simplistes qui n'offrent pas beaucoup de garanties de sécurité à une API sûre enveloppant une API intrinsèquement non sûre en faisant en sorte que toute violation du contrat de cette dernière soit une erreur de type Rust. Le résultat est une interface que Rust peut vous assurer d'utiliser correctement. Pour la plupart, les règles que nous avons imposées à Rust sont le genre de règles que les programmeurs C et C++ finissent par s'imposer de toute façon. Ce qui rend Rust tellement plus strict que C et C++, ce n'est pas que les règles soient si étrangères, mais que cette application soit mécanique et complète..

Conclusion

Rust n'est pas un langage simple. Son objectif est de traverser deux mondes très différents. C'est un langage de programmation moderne, sûr par sa conception, avec des commodités comme les fermetures et les itérateurs, mais il vise à vous donner le contrôle des capacités brutes de la machine sur laquelle il s'exécute, avec une surcharge d'exécution minimale.

Les contours de la langue sont déterminés par ces buts. Rust parvient à combler la majeure partie de l'écart avec un code sécurisé. Son vérificateur d'emprunt et ses abstractions à coût zéro vous rapprochent le plus possible du métal nu sans risquer un comportement indéfini. Lorsque cela ne suffit pas ou lorsque vous souhaitez tirer parti du code C existant, le code non sécurisé et l'interface de fonction étrangère sont prêts. Mais encore une fois, le langage ne se contente pas de vous offrir ces fonctionnalités dangereuses et vous souhaite bonne chance. L'objectif est toujours d'utiliser des fonctionnalités non sécurisées pour créer des API sécurisées. C'est ce qu'on a fait avec `libgit2`. C'est aussi ce que l'équipe Rust a fait avec `Box`, `Vec`, les autres collections, canaux, etc. : la bibliothèque standard regorge d'abstractions sûres, implémentées avec du code dangereux en coulisses.

Un langage avec les ambitions de Rust n'était peut-être pas destiné à être le plus simple des outils. Mais Rust est sûr, rapide, simultané et efficace. Utilisez-le pour construire de grands systèmes rapides, sécurisés et robustes qui tirent parti de toute la puissance du matériel sur lequel ils s'exécutent. Utilisez-le pour améliorer le logiciel.

[Soutien](#) [Se déconnecter](#)

© 2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

à propos des auteurs

Jim Blandy programme depuis 1981 et écrit des logiciels libres depuis 1990. Il a été mainteneur de GNU Emacs et GNU Guile, et mainteneur de GDB, le débogueur GNU. Il est l'un des concepteurs originaux du système de contrôle de version Subversion. Jim travaille maintenant sur les graphismes et le rendu de Firefox pour Mozilla.

Jason Orendorff travaille sur des projets Rust non divulgués chez GitHub. Il a précédemment travaillé sur le moteur JavaScript SpiderMonkey chez Mozilla. Il s'intéresse à la grammaire, à la pâtisserie, aux voyages dans le temps et à aider les gens à apprendre sur des sujets compliqués.

Leonora Tindall est une passionnée de type système et une ingénierie logicielle qui utilise Rust, Elixir et d'autres langages avancés pour créer des logiciels système robustes et résilients dans des domaines à fort impact tels que la santé et la propriété des données. Elle travaille sur une variété de projets open source, des algorithmes génétiques qui font évoluer des programmes dans des langages étranges aux bibliothèques de base de Rust et à l'écosystème de caisses, et apprécie l'expérience de contribuer à des projets communautaires de soutien et diversifiés. Pendant son temps libre, Leonora construit de l'électronique pour la synthèse audio et est une passionnée de radio. Son amour du matériel s'étend également à sa pratique du génie logiciel. Elle a construit des logiciels d'application pour les radios LoRa en Rust et Python et utilise des logiciels et du matériel de bricolage pour créer de la musique électronique expérimentale sur un synthétiseur Eurorack.

[Soutien](#) [Se déconnecter](#)

Indice

Symboles

! opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Opérateurs unaires](#)

!= opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons d'équivalence](#)

Attribut #![feature] , [Attributs](#)

#allow attribut , [Attributs](#)

#cfg attribut , [Attributs](#) , [Fonctionnalités spécifiques à la plate-forme](#)

Attribut #[derive] , [diffusion de pages sur le Web](#)

Attribut #[inline] , [Attributs](#)

Attribut #[link] , [Utilisation des fonctions des bibliothèques](#)

Attribut #[repr(C)] , [Recherche de représentations de données communes](#)

Attribut #[repr(i16)] , [Recherche de représentations de données communes](#)

Attribut #[should_panic] , [Tests et Documentation](#)

#test attribut , [Attributs](#)

\$ (invite de commande) , [rustup et Cargo](#)

Opérateur % , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

& opérateur , [Gestion des arguments de ligne de commande](#) , [Références aux valeurs](#) , [Opérateurs arithmétiques, binaires, de comparaison et logiques](#) , [Modèles de référence](#)

& modèle , [Modèles de référence](#)

Opérateur && , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

Opérateur &mut , [Références aux valeurs](#)

&mut type , [mut et mutex](#)

&mut [T] type , [Références](#)

&str (tranche de chaîne) , [Chaînes en mémoire](#)

Type &[T] , [Références](#)

* opérateur
accéder à la valeur référencée , [Opérateurs de référence](#)

déréférencement , [Gestion des arguments de ligne de commande](#) , [Références Rust versus références C++](#) , [Deref et DerefMut](#) , [Pointeurs bruts](#)

multiplication , [arithmétique, au niveau du bit, comparaison et opérateurs logiques](#)

surcharge de , [Deref et DerefMut](#)

correspondance de motifs et , [Motifs de référence](#)

* caractère générique, pour les versions de caisse , [Versions](#)

*const T , [pointeurs bruts](#)

*mut T , [pointeurs bruts](#)

+ , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Opérateurs binaires](#) , Ajout [et insertion de texte](#)

- opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Opérateurs unaires](#)

. opérateur , [Rust References Versus C++ References](#) , [Fields and Elements](#) , [Deref et DerefMut](#)

.. opérateur , [Champs et Eléments](#)

Opérateur ..= , [Champs et Eléments](#)

/ opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

/// (commentaires de la documentation) , [Ce qu'est réellement l'ensemble de Mandelbrot](#) , [Documentation](#)

:: opérateur , [chemins et importations](#)

::<...> (symbole turbofish) , [Appels de fonction et de méthode](#) , [Structures génériques](#)

< opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons ordonnées](#) , [Pointeurs bruts](#)

<< opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

Opérateur <= , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons ordonnées](#)

= opérateur , [Affectation](#)

Opérateur == , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons d'équivalence](#) , [Pointeurs bruts](#)

=> opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

> opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons ordonnées](#)

>= opérateur , [comparaisons ordonnées](#)

>> opérateur , opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

? opérateur , [propagation des erreurs](#)

@ Patterns , [Reliure avec @ Patterns](#)

^ opérateur , opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

Paramètre de format {?:} , [Formatage des valeurs pour le débogage](#)

Paramètre de format {:p} , [Formatage des pointeurs pour le débogage](#)

| (barre verticale) dans les motifs correspondants , [Possibilités multiples de correspondance](#)

| operator , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

|| operator , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

~ opérateur , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

UN

abandon , [Abandon](#)

chemin absolu , [chemins et importations](#)

"L'abstraction et le modèle de machine C++" (Stroustrup) , [et pourtant la rouille est toujours rapide](#)

méthodes d'accumulation pour les itérateurs , [Accumulation simple : count, sum, product , fold et rfold](#)

Framework actix-web , [Servir des pages sur le Web - Servir des pages sur le Web](#) , [Rappels](#) , Mise en [réseau](#)

méthodes d'adaptation

énumérer , [Un programme de Mandelbrot simultané](#)

fusible , [Futures](#)

méthodes d'adaptation pour les itérateurs , [Adaptateurs](#) d'itérateurs - [cycle](#)

by_ref , [by_ref - by_ref](#)

chaîne , [chaîne](#)

cloné , [cloné, copié](#)

copié , [cloné, copié](#)

cycler , [cycler](#)

énumérer , [énumérer](#) , [zip](#)

filter_map et flat_map , [filter_map et flat_map](#) - [filter_map et flat_map](#)

aplatir , [aplatir - aplatir](#)

fusible , [fusible](#)

inspecter , [inspecter](#)

mapper et filtrer , [mapper et filtrer - mapper et filtrer](#)

visible , [visible](#)

itérateurs réversibles et rev , [Itérateurs réversibles et rev - Itérateurs réversibles et rev](#)

sauter et sauter_pendant , [sauter et sauter_pendant](#)

prendre et prendre_pendant , [prendre et prendre_pendant](#) , by_ref

fermeture éclair , [fermeture éclair](#)

méthodes d'adaptation pour les lecteurs

 méthode des octets , [Lecteurs](#)

 méthode de la chaîne , [Lecteurs](#)

 prendre la méthode , [Lecteurs](#)

types de données algébriques , [Enums et Patterns](#)

valeur d'alignement, requise par les types , les [tailles de caractères et les alignements](#)

Fonction align_of , [Tailles et alignements des caractères](#)

Fonction align_of_val , [Tailles de type et alignements](#)

all, iterator method , [any and all](#)

[allow] attribut , [Attributs](#)

any, iterator method , [any et all](#)

de toute façon la caisse de gestion des erreurs , [Travailler avec plusieurs types d'erreurs](#), [Types d'erreurs](#) et de résultats

Type de pointeur d'arc , [Rc et Arc : Propriété partagée](#) - [Rc et Arc : Propriété partagée](#) , [Passage de soi en tant que boîte](#), [Rc ou Arc](#) - [Passage de soi en tant que boîte](#), [Rc ou Arc](#) , [Partage de données immuables entre les threads](#) , [Mutex<T>](#)

fonction args , [Gestion des arguments de ligne de commande](#) , skip et skip_while

Type d'arguments, pour le formatage de chaîne , [Utilisation du langage de formatage dans votre propre code](#)

opérateurs arithmétiques , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , Opérateurs [arithmétiques et binaires](#) - Opérateurs [arithmétiques et binaires](#)

 opérateurs d'affectation composés , [Opérateurs d'affectation](#) composés - Opérateurs d' [affectation composés](#)

 surcharge , [opérateurs binaires](#)

arithmétique, pointeur , [Arithmétique du pointeur](#) - [Arithmétique du pointeur](#)

modèles de tableau , modèles de [tableau et de tranche](#)
tableaux , [tableaux](#)

concaténation de tableaux de , [Rejoindre](#)
pointeurs bruts vers , [Pointer Arithmetic](#) - [Pointer Arithmetic](#)
tranches et , [tranches](#)
tuples contre , [Tuples](#)

tableaux, joindre des tableaux de , [Joindre](#)
en tant qu'opérateur , [Conversions vers et à partir d'entiers](#)
Caractères ASCII , [Types d'entiers](#) , [ASCII, Latin-1 et Unicode](#) , [Caractères de classification](#) , [Traitement des chiffres](#)

Type de chaîne Ascii, code non sécurisé pour la conversion en chaîne ,
[Exemple : Un type de chaîne ASCII efficace](#) - [Fonctions non sécurisées](#)

Trait AsMut , [AsRef et AsMut](#)

Caractéristique AsRef , [AsRef et AsMut](#) , [OsStr et Path](#)
affirmer! macro , [Fonctions Rust](#) , [Tests et Documentation](#)
assert_eq ! macro , [Tests et documentation](#) , [Macros](#) , [Principes de base des macros](#) - Principes de base des macros

mission

C++ versus Rust , [Moves - Moves](#)

opérateurs d'affectation composés , [Affectation](#) , [Opérateurs](#) d'affectation composés - Opérateurs d' [affectation composés](#)
expressions , [Affectation](#)

coups et (voir coups)

Python contre Rust , [Moves - Moves](#)

références , [Attribuer des références](#)

dans Rust , [plus d'opérations qui bougent](#) - [Déplacements et contenu indexé](#)

à une variable , [Plus d'opérations qui bougent](#)

opérateurs d'affectation , [Affectation](#)

Const associés , Const associés , [Const associés](#)

fonctions associées , [Définir des méthodes avec impl](#)

types associés , [Types associés \(ou Fonctionnement des itérateurs\)](#) -
[Types associés \(ou Fonctionnement des itérateurs\)](#)

associativité , [priorité et associativité](#)

fonctions asynchrones , fonctions asynchrones [et expressions d'attente](#)
- [Appel de fonctions asynchrones à partir de code synchrone : blo-](#)

[ck_on](#) , [création de fonctions asynchrones à partir de blocs asynchrones](#)

Blocs de déplacement asynchrones , [Blocs asynchrones](#)
flux asynchrones , [Réception de paquets : plus de flux asynchrones](#) -
[Réception de paquets : plus de flux asynchrones](#)
async-std crate , [Programmation asynchrone](#) , [Futures](#) , [Fonctions asynchrones et expressions d'attente](#) , Création de [tâches asynchrones](#)
programmation asynchrone , [Programmation asynchrone](#) - [Quand le code asynchrone est-il utile ?](#)

blocs asynchrones , [Programmation asynchrone](#) , [Blocs asynchrones](#) - [Création de fonctions asynchrones à partir de blocs asynchrones](#)

fonctions asynchrones , fonctions asynchrones [et expressions d'attente](#) - [Appel de fonctions asynchrones à partir de code synchrone : block_on](#) , [création de fonctions asynchrones à partir de blocs asynchrones](#)

client et serveur , [Un client et un serveur asynchrones](#) - [Groupes de discussion : canaux de diffusion de tokio](#)

par rapport à la programmation synchrone , [De synchrone à asynchrone](#) - [Un vrai client HTTP asynchrone](#)

futurs et exécuteurs, coordination , [Primitive Futures and Executors: Quand un futur mérite-t-il d'être interrogé à nouveau?](#) - [Implémentation de block_on](#)

Caisse de client HTTP , [un vrai client HTTP asynchrone](#)
épingler des contrats à terme , [Épingler](#) - [Le trait de détachement](#)
tâches versus threads traditionnels , [Programmation asynchrone](#)
situations utiles pour , [Quand le code asynchrone est-il utile ?](#) -
[Quand le code asynchrone est-il utile ?](#)

méthode as_mut_ptr , [pointeurs bruts](#)

méthode as_ptr , [pointeurs bruts](#)

opérations sur les entiers atomiques , [variables globales](#)

nombre de références atomiques (voir type de pointeur d'arc)

Types et opérations atomiques , Atomiques , Variables [globales](#)

attributs , [Ecrire et exécuter des tests unitaires](#) , [Attributs](#) - [Attributs](#)

await expressions , [Async Functions et Await Expressions](#) - [Appel de fonctions asynchrones à partir de code synchrone : block_on](#) , Génération de [tâches](#) asynchrones - Génération de tâches [asynchrones](#)

fil d'arrière -plan , [Concurrence](#)

contre-pression

client asynchrone et serveur de discussion , [Groupes de discussion :](#)

[canaux de diffusion de tokio](#) - [Groupes de discussion : canaux de diffusion de tokio](#)

approche pipeline , [fonctionnalités et performances du canal](#)

commande bat , [systèmes de fichiers et outils de ligne de commande](#)

entrée/sortie binaire , [données binaires, compression et sérialisation](#)

littéral numérique binaire , [types entiers](#)

opérateurs binaires , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , Opérateurs [binaires](#)

binaire, formatage des nombres dans , [Formatage des nombres](#)

BinaryHeap , [Vue](#) d'ensemble , [BinaryHeap<T>](#) - [BinaryHeap<T>](#)

Type BinaryTree , [Énumérations génériques](#) , Remplir [un arbre binaire](#) , [Implémentation de vos propres itérateurs](#) - [Implémentation de vos propres itérateurs](#)

bindgen crate , [une interface brute pour libgit2](#)

opérateurs au niveau du bit , Opérateurs [arithmétiques, au niveau du bit, de comparaison et logiques](#) , Opérateurs [d'affectation](#) composés -

Opérateurs [d'affectation composés](#)

blocs , [blocs et points-virgules](#) - [Déclarations](#)

asynchrone , [Programmation asynchrone](#) , [Blocs](#) asynchrones -

[Création de fonctions asynchrones à partir de blocs asynchrones](#)

déclarations dans , [Déclarations](#)

extern block , [Déclarer des fonctions étrangères et des variables](#) -

[Déclarer des fonctions étrangères et des variables](#)

bloc impl , [Définir des méthodes avec impl](#) - [Fonctions associées au type](#)

unsafe , [Raw Pointers](#) , [Unsafe Code](#) , [Unsafe Blocks](#) - [Exemple : Un type de chaîne ASCII efficace](#) , [Unsafe Block ou Unsafe Function ?](#)

block_on , [Appel de fonctions asynchrones à partir de code](#)

[synchrone : block_on](#) - [Appel de fonctions asynchrones à partir de code synchrone : block_on](#) , [Futurs primitifs et exécuteurs : quand un futur mérite-t-il d'être interrogé à nouveau ?](#) , [Implémenter block_on](#) - [Implémenter block_on](#)

Type booléen (bool) , [Le type bool](#) , [Mise en forme d'autres types](#)

Emprunter le trait , [Emprunter et EmprunterMut](#) - [Emprunter et EmprunterMut](#)

Emprunter<str> trait , [Emprunter sous d'autres types de texte](#)

emprunt , [Références](#)

futurs et , Génération de [tâches asynchrones](#) , [Les deux étapes de la vie d'un futur](#) - [Les deux étapes de la vie d'un futur](#)

itération et , [Gestion des arguments de la ligne de commande](#) , [by_ref](#)

variables locales et , [Emprunter une variable locale](#) - [Emprunter une variable locale](#)

syndicats , syndicats [emprunteurs](#)

valeurs d'expressions arbitraires , [Emprunter des références à des expressions arbitraires](#)

Trait BorrowMut , [Emprunter et EmprunterMut](#) - [Emprunter et EmprunterMut](#)

limites, rétro-ingénierie , [Reverse-Engineering Bounds](#) - [Reverse-Engineering Bounds](#)

Type de boîte , [Boîtes](#) , [Propriété](#) , [Se faire passer pour une boîte](#), [Rc ou Arc](#) - [Se passer pour une boîte](#), [Rc ou Arc](#)

break expressions , [flux de contrôle dans les boucles](#)

canal de diffusion , [Groupes de chat : canaux de diffusion de tokio](#) - [Groupes de chat : canaux de diffusion de tokio](#)

Type de collection BTreeMap<K, V> , [HashMap<K, V>](#) et [BTreeMap<K, V>](#) - [Itération de carte](#)

Type de collection BTreeSet , [HashSet<T>](#) et [BTreeSet<T>](#) - [Opérations sur l'ensemble](#)

BTreeSet::new , [HashSet<T>](#) et [BTreeSet<T>](#)

lecteurs tamponnés , [Lecteurs tamponnés](#) - [Lignes de collecte](#)

Caractéristique BufRead , [lecteurs tamponnés](#)

méthode de consommation , [lecteurs tamponnés](#)

méthode fill_buf , [lecteurs tamponnés](#)

méthode read_line , [lecteurs tamponnés](#)

Méthode read_until , [Lecteurs mis en mémoire tampon](#)

méthode fractionnée , [lecteurs tamponnés](#)

Type BufReader , [Lecteurs tamponnés](#)

BufReader<TcpStream> , [Réception de paquets : Plus de flux asynchrones](#)

BufWriter::with_capacity , [Écrivains](#)

bogues, code non sécurisé et fonctions [non sécurisées](#)

créer des profils , [Créer des profils](#)

build script , [Utilisation des fonctions des bibliothèques](#)

Trait BuildHasher , [Utilisation d'un algorithme de hachage personnalisé](#)

par valeur/par référence

passage d'une collection , [Implémentations IntoIterator](#) ,
[Collections](#)

passage des arguments de la fonction , [Références aux valeurs - Références aux valeurs](#)

Littéraux d'octets , [Types entiers](#)

chaînes d'octets , chaînes d' [octets](#)

byteorder crate , [données binaires, compression et sérialisation](#)

adaptateur itérateur by_ref , [by_ref](#) - [by_ref](#)

C

C , les [programmeurs système peuvent avoir de belles choses](#)

(voir aussi fonctions étrangères)

énumérations , [énumérations](#) - [énumérations](#)

passage de chaînes entre Rust et , [Recherche de représentations de données communes](#)

pointeurs dans , [Pointeurs bruts](#) , Pointeurs [bruts](#)

règles pour éviter les comportements indéfinis , les [programmeurs système peuvent avoir de belles choses](#)

représentations de type , [Recherche de représentations de données communes](#) - [Recherche de représentations de données communes](#)

C#

fonctions asynchrones , [Comparaison de conceptions asynchrones](#)

Enums , [Enums et Patterns](#) - [Enums](#)

traits versus méthodes virtuelles , [Utilisation des traits](#)

C++ , [les programmeurs système peuvent avoir de belles choses](#)

(voir aussi fonctions étrangères)

affectation dans , [Moves](#) - [Moves](#)

constexpr , [Variables globales](#)

Enums , [Enums et Patterns](#) - [Enums](#)

erreurs d'invalidation et , [Partage contre mutation](#) , [Collections](#) ,
[Rust exclut les erreurs d'invalidation](#)

macros , [Macros](#) , [Conséquences inattendues](#)

mutex dans , [Qu'est-ce qu'un mutex ?](#) - [Qu'est-ce qu'un mutex ?](#)

propriété dans , [Propriété](#) - [Propriété](#)

pointeurs dans , [Pointeurs bruts](#) , Pointeurs [bruts](#)

pointeurs vers const versus références partagées , [partage versus mutation](#)

création de références dans , [Références Rust Versus Références C++](#)

règles pour éviter les comportements indéfinis , les [programmeurs système peuvent avoir de belles choses](#)

traits versus méthodes virtuelles , [Utilisation des traits](#)

méthode calculate_tides , [Gestion des erreurs dans main\(\)](#)

rappels, fermetures et , [Rappels - Rappels](#)

annulation, atomique et , [Atomique](#)

capacité d'un vecteur , [Vecteurs](#) , Vecteurs [croissants et rétrécissants](#)

méthode captures_iter sur le type Regex , [utilisation de base de Regex](#)

Cargo , [Rust facilite la collaboration](#)

build script , [Utilisation des fonctions des bibliothèques](#)

documentation , [Documentation - Doc-Tests](#)

rustup et , [rustup et Cargo - rustup et Cargo](#)

Répertoire src/bin , [Le répertoire src/bin - Le répertoire src/bin](#)

gestion des versions , [spécification des dépendances](#)

construction de cargaison , [Caisse - Caisse](#)

Commande de fret , [Rustup et Cargo](#)

Commande cargo doc , [Documentation](#)

commande de package de fret , [publication de caisses sur crates.io](#)

commande de test cargo , [Tests et Documentation - Tests et Documentation](#)

Cargo.lock , [Cargo.lock](#)

conversion de casse

pour les caractères , [Conversion de casse pour les caractères](#)

pour les chaînes , [Conversion de casse pour les chaînes](#)

moulages , moulages de [type](#)

Fonction catch_unwind , [Déroulement](#)

Type cellulaire , [Mutabilité intérieure](#)

#cfg attribut , [Attributs , Fonctionnalités spécifiques à la plate-forme](#)

cfg ! macro , [Macros intégrées](#)

Méthode ch.to_digit , [Gestion des chiffres](#)

adaptateur de chaîne , [chaîne](#)

canaux , [Canaux](#) - Au- [delà des pipelines](#)

canal de diffusion , [Groupes de chat : canaux de diffusion de tokio](#) -

[Groupes de chat : canaux de diffusion de tokio](#)

impasse avec , [impasse](#)

Fonctionnalités et performances , [Fonctionnalités et performances](#)

[de la chaîne](#) - Fonctionnalités et performances de la chaîne

canaux multiconsommateurs utilisant mutex , [Canaux multicon-](#)

[sommateurs utilisant mutex](#)

utilisations hors pipeline , Au- [delà des pipelines](#)

rediriger l'itérateur vers , [Diriger presque n'importe quel itérateur](#)

[vers un canal](#) - [Diriger presque n'importe quel itérateur vers un](#)

[canal](#)

recevoir des valeurs , [recevoir des valeurs](#)

Envoyer et synchroniser pour la sécurité des threads , [Sécurité des](#)

[threads : envoyer et synchroniser](#) - [Sécurité des threads : envoyer](#)

[et synchroniser](#)

envoyer des valeurs , [Canaux](#) - Au- [delà des pipelines](#)

caractères littéraux , [Caractères](#)

caractères (char) , [Caractères](#) - [Caractères](#) , [Caractères \(char\)](#) - [Conver-](#)

[sions vers et depuis des entiers](#)

conversion de casse , [Conversion de casse pour les caractères](#)

classer , [Classer les caractères](#) - [Classer les caractères](#)

chiffres , [Manipulation des chiffres](#)

conversion d'entiers , [Conversions vers et depuis des entiers](#)

Méthode is_digit , [Gestion des chiffres](#)

Méthode is_lowercase , [Conversion de casse pour les caractères](#)

Méthode is_uppercase , [Conversion de casse pour les caractères](#)

types numériques versus , [Caractères](#)

Points communs de type Rust et C , [Trouver des représentations de](#)

[données communes](#)

Méthode to_digit , [Gestion des chiffres](#)

Méthode to_lowercase , [Conversion de casse pour les caractères](#)

Méthode to_uppercase , [Conversion de casse pour les caractères](#)

groupes de discussion, chaînes de diffusion de tokio , [Groupes de dis-](#)

[cussion : chaînes de diffusion de tokio](#) - [Groupes de discussion :](#)

[chaînes de diffusion de tokio](#)

opérations vérifiées , [Arithmétique vérifiée, enveloppante, saturée et](#)

[débordante](#)

processus enfant , [Autres types de lecteurs et d'enregistreurs](#)

Type ChildStdin , [Autres types de lecteurs et d'enregistreurs](#)

client et serveur, chat asynchrone , [Un client et un serveur asyn-](#)

[chrones](#) - [Groupes de discussion : canaux de diffusion de tokio](#)

connexions de chat avec des mutex asynchrones , [Gestion des connexions de chat](#) : mutex asynchrones - [Gestion des connexions de chat : mutex asynchrones](#)

groupes de discussion, chaînes de diffusion de tokio , [Groupes de discussion : chaînes de diffusion de tokio](#) - [Groupes de discussion : chaînes de diffusion de tokio](#)

la fonction principale du client , La fonction principale du [client](#) - [La fonction principale du client](#)

types d' erreur et de résultat , [Types d'erreur et de résultat](#)

protocole , [Le protocole](#) - [Le protocole](#)

réception de paquets , [Réception de paquets : plus de flux asynchrones](#) - [Réception de paquets : plus de flux asynchrones](#)

envoi de paquets , [envoi de paquets](#)

fonction principale du serveur , La fonction principale du [serveur](#) flux pour prendre l'entrée de l'utilisateur , [Prise de l'entrée de l'utilisateur : flux asynchrones](#) - [Prise de l'entrée de l'utilisateur : flux asynchrones](#)

méthode de clonage , [Mouvements](#) , [Utilisation de caractéristiques](#)

Clone trait , [Clone](#) , [Copy et Clone for Closures](#) , [Accessing Elements](#)

méthode de l'adaptateur cloné pour les itérateurs , [cloné, copié](#)

plages fermées (inclusives) , [champs et éléments](#)

Fermetures , [Un programme Mandelbrot simultané](#) , [Fermetures](#) , [Fermetures](#) - [Utilisation efficace des fermetures](#)

références d'emprunt , [Fermetures qui empruntent](#)

rappels , [Rappels](#) - [Rappels](#)

capture de variables , [Capture de variables](#)

Cloner pour , [copier et cloner pour les fermetures](#)

Copier pour , [copier et cloner pour les fermetures](#)

laisser tomber des valeurs , des [fermetures qui tuent](#) - [FnOnce](#)

utilisation efficace de , [Utilisation efficace des fermetures](#)

FnMut , [FnMut](#) - [FnMut](#)

FnOnce , [FnOnce](#) - [FnOnce](#)

inspecter l'adaptateur et , [inspecter](#)

mise en page en mémoire , [performance de fermeture](#)

Mot- clé de déplacement , [Des fermetures qui volent](#)

performances , performances de [fermeture](#)

sécurité , [fermetures et sécurité](#) - [Copier et cloner pour les fermetures](#)

"qui tue" , [Des fermetures qui tuent](#)

types , types de [fonction et de fermeture](#) - Types [de fonction et de fermeture](#)

dans l'exemple de serveur Web , [Servir des pages sur le Web](#)

fragments de code, macros , [Principes de base de l'expansion des macros](#) - [Conséquences inattendues](#)

collaboration, Rust et , [Rust facilite la collaboration](#)

méthode collect , [Un programme Mandelbrot concurrent](#) , [L'interface de ligne de commande](#) , [Construire des collections : collect et FromIterator](#) - [Construire des collections : collect et FromIterator](#) , [partition](#) , [Collecter des lignes](#)

collections , [Collections](#) - Au- [delà des collections standard](#)

BinaryHeap<T>type de collection , [BinaryHeap<T>](#) -
[BinaryHeap<T>](#)

BTreeMap<K, V> , [HashMap<K, V>](#) et [BTreeMap<K, V>](#) - [Itération de carte](#)

BTreeSet<T> , [HashSet<T>](#) et [BTreeSet<T>](#) - [Opérations sur l'ensemble](#)

sur mesure , [Au-delà des collections standard](#)

hachage , [Hachage](#) - [Utilisation d'un algorithme de hachage personnalisé](#)

HashMap<K, V> , [HashMap<K, V>](#) et [BTreeMap<K, V>](#) - [Itération de carte](#)

HashSet<T> , [HashSet<T>](#) et [BTreeSet<T>](#) - [Opérations sur l'ensemble](#)

itération par valeur , [Implémentations IntoIterator](#) , [Collections itérateurs](#) et , [iter et iter mut Méthodes](#)

chaînes en tant que génériques , [chaînes en tant que collections génériques](#)

Vec<T> , [Vec<T>](#) - [Rust élimine les erreurs d'invalidation](#)

VecDeque<T> , [VecDeque<T>](#) - [VecDeque<T>](#)

colonne! macro , [Macros intégrées](#)

invite de commande (\$), [rustup](#) et [Cargo](#)

Type de commande , [Autres types de lecteurs et d'enregistreurs](#)

arguments de ligne de commande , [Gestion des arguments de ligne de commande](#) arguments de ligne de commande - [Gestion des arguments de ligne de commande](#)

interface de ligne de commande , [Systèmes de fichiers et outils de ligne de commande](#) - [L'interface de ligne de commande](#)

communauté, Rust , [Plus de belles choses](#)

opérateurs de comparaison , [Type](#) booléen , [Opérateurs arithmétiques, binaires, de comparaison et logiques](#)

avec des itérateurs , [Comparer des séquences d'éléments](#)

surcharge , [comparaisons ordonnées](#)

références et , [Comparaison des références](#)

avec des chaînes , [Utilisation de chaînes](#)

équivalence de compatibilité pour les caractères Unicode , [Formes de normalisation](#)

nombres complexes , [Analyse des arguments de la ligne de commande d'une paire](#)

caractères Unicode composés ou décomposés , [Normalisation](#)

opérateurs d'affectation composés , [Affectation](#) , [Opérateurs](#) d'affectation composés - Opérateurs d' [affectation composés](#)

compression , [données binaires, compression et sérialisation](#)

méthode concat , [chaîne](#)

concat ! macro , [Macros intégrées](#)

simultanéité , [La programmation parallèle est apprivoisée](#) , [Concurrency](#) - [À quoi ressemble le piratage de code simultané dans Rust](#)

canaux , [Canaux](#) - Au- [delà des pipelines](#)

parallélisme fork-join , [Parallélisme fork-join](#) - [Revisiter l'ensemble de Mandelbrot](#)

Ensemble de Mandelbrot , [Un programme de Mandelbrot concurrent](#) - [Un programme de Mandelbrot concurrent](#)

Prise en charge de Rust pour , [Concurrency - Safety Is Invisible](#) , [Ownership and Moves](#)

état mutable partagé , État [mutable partagé](#) - [Variables globales](#)

condition (avec instruction if) , [if et match](#)

Variables de condition (Condvar) , [Variables de condition \(Condvar\)](#)

fonction const , [variables globales](#)

const generics , [structures génériques avec paramètres constants](#) , [fonctions génériques et paramètres de type](#)

Constantes , [Modules](#) , [Statique et constantes](#) , Variables [globales](#)
consts , [statiques et constantes](#)

*const T , [pointeurs bruts](#)

associés , [Const](#) associés , Const [associés](#)

références partagées versus pointeurs vers , [partage versus mutation](#)

méthode de consommation , [lecteurs tamponnés](#)

consommer des itérateurs , [Consommer des itérateurs - for each et try for each](#)

méthodes d'accumulation , [Accumulation simple : comptage, somme, produit](#)

toutes les méthodes , [toutes et toutes](#)

méthode collect , [Création de collections : collect et FromIterator - Création de collections : collect et FromIterator](#)

comparaison de séquences d'articles , [Comparaison de séquences d'articles](#)

méthode de comptage , [Accumulation simple : comptage, somme, produit](#)

ExactSizeIterator , [position, rposition et ExactSizeIterator](#)

Trait d'extension , [Le trait d'extension](#)

Méthodes find, rfind et find_map , [find, rfind et find_map](#)

méthode de pliage , [plier et replier](#)

méthode for_each , [for each et try for each](#)

Caractéristique FromIterator , Création [de collections : collect et FromIterator](#)

dernière méthode , [dernière](#)

méthodes max_by et min_by , [max by, min by](#)

Méthodes max_by_key et min_by_key , [max by key, min by key](#)

méthodes min et max , [max, min](#)

Méthodes nth et nth_back , [nth, nth back](#)

méthode de partition , [partition](#)

position method , [position, rposition et ExactSizeIterator](#)

méthode du produit , [Accumulation simple : compte, somme, produit](#)

méthode rfind , [find, rfind et find map](#)

méthode rfold , [fold et rfold](#)

méthode rposition , [position, rposition et ExactSizeIterator](#)

méthode somme , [Accumulation simple : compte, somme, produit](#)

méthodes try_fold et try_rfold , [try fold et try rfold - try fold et try rfold](#)

méthode try_for_each , [for each et try for each](#)

contrats

fonctionnalité non sécurisée et , [Dangereux de quoi ?](#)

fonctions non sécurisées et , [Fonctions non sécurisées](#)

traits dangereux et , [traits dangereux](#)

adaptateur copié , [cloné, copié](#)

méthode de copie , [lecteurs et écrivains](#)

Type de copie , [types de copie : l'exception aux mouvements](#) - [Types de copie : l'exception aux mouvements](#) , [copie](#) , [copie et clonage pour les fermetures](#)

méthode de comptage , [Accumulation simple : comptage, somme, produit](#)

Type de vache (clone en écriture) , [emprunter et posséder au travail : la vache humble](#) , reporter l' [allocation](#) - [reporter l'allocation caisses](#) , [Caisse](#)s - [Construire des profils](#)

Attribut #[inline] , [Attributs](#)

\$crate fragment versus crate keyword , [Importation et exportation de macros](#)

doc-tests , [Doc-Tests](#) - [Doc-Tests](#)

publication sur crates.io , [Publication de caisses sur crates.io](#)

spécification des dépendances , [spécification des dépendances](#) - [Cargo.lock](#)

Répertoire src/bin et , [Le répertoire src/bin](#) - [Le répertoire src/bin espaces de travail](#) , Espaces de [travail](#)

crates.io , [Publier des caisses sur crates.io](#)

section critique du code , [Qu'est-ce qu'un mutex ?](#)

caisse à traverses , [un programme Mandelbrot simultané](#)

Cursor::new , [Autres types de lecteurs et d'enregistreurs](#)

adaptateur vélo , [vélo](#)

ré

pointeur suspendu , [Propriété et mouvements](#) , [Partage contre mutation](#)

parallélisme des données , [Concurrence](#)

courses aux données , [Concurrence](#) , La [sécurité est invisible](#) , [Qu'est-ce qu'un mutex ?](#) , [Pourquoi les mutex ne sont pas toujours une bonne idée](#)

impasse , [impasse](#)

Trait de formatage de débogage , [Conversion d'autres types en chaînes](#) , [Pointeurs bruts](#)

débogage

valeurs de formatage pour , [Valeurs de formatage pour le débogage](#) - [Pointeurs de formatage pour le débogage](#)

macros , [Macros](#) de débogage - [Macros de débogage](#)

debug_assert ! macro , [Fonctions Rust](#) , [Tests et Documentation](#)

debug_assert_eq ! macro , [Tests et Documentation](#)
déclarations , [Déclarations](#) , [Déclarer des fonctions étrangères et des variables](#) - [Utilisation des fonctions des bibliothèques](#)
caractères Unicode décomposés et composés , [Normalisation](#)
Trait par défaut , [Par défaut](#) - [Par défaut](#) , partition
implémentation des traits par défaut , [méthodes par défaut](#)
dépendances

Cargo.lock , [Cargo.lock](#)
dans le contexte de la caisse , [Caisses](#)
spécifiant , [Spécification des dépendances](#) - [Cargo.lock](#)
versions et , [Spécification des dépendances](#)

graph de dépendance , [Crates](#)
Deref coercions , [Type](#) Casts , [Deref et DerefMut](#) , [Raw Pointers](#)
Deref trait , [Deref et DerefMut](#) - [Deref et DerefMut](#)
déréférencement

* opérateur , [Gestion des arguments de ligne de commande](#) , [Références Rust versus références C++](#) , [Pointeurs bruts](#)
pointeurs bruts , pointeurs [bruts](#) , [dangereux de quoi ?](#) , [Blocs non sécurisés](#) , [Déréférencement des pointeurs bruts en toute sécurité](#) - [Exemple : RefWithFlag](#)

DerefMut trait , [Deref et DerefMut](#) - [Deref et DerefMut](#)
Attribut #[derive] , [diffusion de pages sur le Web](#)
Trait de déserialisation , [Réception de paquets : plus de flux asynchrones](#)
chiffres, manipulation , [Manipulation des chiffres](#)
directionnalité du texte , [Directionnalité du texte](#)
répertoires

modules et , [Modules dans des fichiers séparés](#)
lecture , [Répertoires](#) de lecture - [Répertoires de lecture](#)
src/bin , [Le répertoire src/bin](#) - [Le répertoire src/bin](#)

Structure DirEntry , [Lecture des répertoires](#)
méthode nom_fichier , [Lecture des répertoires](#)
méthode file_type , [Lecture des répertoires](#)
méthode des métadonnées , [lecture des répertoires](#)
méthode path , [Lecture des répertoires](#)

unions discriminées , [énumérations et modèles](#)
Trait de formatage d'affichage , [Conversion d'autres types en chaînes](#) , [Pointeurs bruts](#)
fonction divergente , [pourquoi Rust a une boucle](#)

commentaires doc , [Documentation](#)
doc-tests , [Doc-Tests - Doc-Tests](#)
documentation , [Documentation - Doc-Tests](#)
commentaires de documentation (///) , [Ce qu'est réellement l'ensemble de Mandelbrot](#) , [Documentation](#)
guillemets doubles , [servir des pages sur le Web](#)
Trait DoubleEndedIterator , [itérateurs réversibles et rev](#)
méthode de vidange , [Méthodes de vidange](#)
Trait de chute , [Drop - Drop](#) , [Lecteurs](#)
suppression de valeurs
dans les fermetures , [Les fermetures qui tuent](#) - [FnOnce](#)
FnOnce , [FnOnce - FnOnce](#)
propriété et , [Propriété](#)
dans Rust , [Propriété](#)
typage canard , [Types fondamentaux](#)
largeurs et précisions dynamiques , largeurs et précisions [dynamiques](#)

E

éditions , [éditions](#)
méthode écoulée , [Traiter les erreurs qui "ne peuvent pas se produire"](#)
éléments
valeurs de structure de type tuple , structures de type [tuple](#)
Type de collection Vec<T> , [Accès aux éléments - Accès](#) aux éléments , [Éléments aléatoires](#)
algorithme parallèle embarrassant , [Concurrence](#)
plages exclusives (semi-ouvertes) , [Champs et éléments](#) , [Possibilités multiples correspondantes](#)
plages inclusives (fermées) , [champs et éléments](#)
entrées, mappez les paires clé-valeur comme , [HashMap<K, V>](#) et [BTreeMap<K, V>](#)
Type d'entrée, HashMap et BTreeMap , [Entrées - Entrées](#)
adaptateur d'énumération , [un programme Mandelbrot concurrent](#) , [énumération](#) , [zip](#)
type énuméré (enum) , [Qu'est-ce que l'ensemble de Mandelbrot est réellement](#) , [Énumérations et modèles - Vue d'ensemble](#)
Style C , [Enums - Enums](#)
avec données , [Énumérations avec données](#)
générique , [Énumérations](#) génériques - [Enumérations génériques](#)
implémentation de hachage , [Hachage](#)

en mémoire , [Énumérations en mémoire](#)

structures de données riches avec , [Structures de données riches utilisant des énumérations](#) - [Structures de données riches utilisant des énumérations](#)

module env , [gestion des arguments de la ligne de commande](#)

env! macro , [Macros intégrées](#)

eprintln ! macro , [Gestion des arguments de ligne de commande](#)

Trait d'équation , [Hachage](#)

opérateurs d'égalité , Opérateurs [arithmétiques, binaires, de comparaison et logiques](#) , [Comparaisons](#) d'équivalence - [Comparaisons d'équivalence](#)

gestion des erreurs , [Gestion des erreurs](#) - [Pourquoi des résultats ?](#)

sur les threads , [Gestion des erreurs sur les threads](#)

anyhow crate , [Travailler avec plusieurs types d'erreurs](#)

chat asynchrone , [types d'erreur et de résultat](#)

éviter les erreurs de syntaxe dans la correspondance des macros ,
[Éviter les erreurs de syntaxe lors de la correspondance](#)

attraper les erreurs , [Attraper les erreurs](#) - [Attraper les erreurs canaux et](#) , [Valeurs d'envoi](#)

déclaration d'un type d'erreur personnalisé , [Déclaration d'un type d'erreur personnalisé](#)

erreurs qui "ne peuvent pas arriver" , [Traiter les erreurs qui "ne peuvent pas arriver"](#)

types d'erreurs de formatage , [Autres types de formatage](#)

ignorer les erreurs , [Ignorer les erreurs](#)

erreurs d'invalidation , [Collections](#) , [Rust exclut les erreurs d'invalidation](#)

dans la fonction main , [Gestion des erreurs dans main\(\)](#)

avec plusieurs types d'erreurs , [Utilisation de plusieurs types d'erreurs](#) - [Utilisation de plusieurs types d'erreurs](#)

panique , [Panique](#) - [Abandon](#)

PoisonError::into_inner , [mutex empoisonnés](#)

erreurs d'impression , [Erreurs](#) d'impression - [Erreurs d'impression](#)

propagation des erreurs , [propagation des erreurs](#)

Type de résultat , [Résultat](#) - [Pourquoi des résultats ?](#) , [Types d'erreur et de résultat](#)

code non sécurisé et , [Une interface sécurisée pour libgit2](#)

Caractéristique d'erreur

méthode source , [Erreurs d'impression](#)

méthode to_string , [Erreurs d'impression](#)
fonction escape_time , [from fn et successeurs](#)
ExactSizeIterator trait , [position, rposition et ExactSizeIterator](#)
exceptions, Résultat versus , [Pourquoi des résultats ?](#)
plages exclusives (semi-ouvertes) , [champs et éléments](#)
exécuteurs (asynchrones)

block_on , [Appel de fonctions asynchrones à partir de code synchrone : block on](#) - [Appel de fonctions asynchrones à partir de code synchrone : block on](#) , [Futurs primitifs et exécuteurs : quand un futur mérite-t-il d'être interrogé à nouveau ?](#) , [Implémenter block on](#) - [Implémenter block on](#)

fonction spawn , [spawn and join](#) - [spawn and join](#) , [Error Handling Across Threads](#) , [Spawning Async Tasks](#) , [Spawning Async Tasks on a Thread Pool](#)

spawn_local , Génération de [tâches](#) asynchrones - Génération de tâches asynchrones , Génération de [tâches asynchrones sur un pool de threads](#)

méthode expect , [Gestion des arguments de ligne de commande](#) , [Écriture de fichiers image](#) , [Gestion des erreurs dans main\(\)](#)

expressions , [Expressions](#) - [En avant](#)

affectation , [affectation](#)

blocs et points-virgules , [Blocs et points-virgules](#) - [Déclarations](#)

Fermetures , [Fermetures](#)

déclarations , [Déclarations](#)

champs et éléments , [Champs et éléments](#)

appels de fonction/méthode , Appels de [fonction et de méthode](#)

si et match , [si et match](#) - [si let](#)

si laissé , [si laissé](#)

loops , [Flux de contrôle dans les boucles](#) - [Flux de contrôle dans les boucles](#)

Priorité et associativité , [Priorité et associativité](#)

opérateurs de référence , [Opérateurs de référence](#)

expressions régulières , [Expressions régulières](#) - [Création de valeurs régulières paresseusement](#)

retour , [retour Expressions](#)

Rust comme langage d'expression , [Un langage d'expression](#)

déclarations versus , [un langage d'expression](#)

struct , [Structures de champ nommé](#)

moulages de type , moulages de [type](#)

fonction d'extension , [partage contre mutation](#)

Trait d'extension , [Le trait d'extension](#)

méthode extend_from_slice , [partage contre mutation](#)

Traits d' extension , [Traits et autres types de personnes](#) , [Prise d'entrée utilisateur : flux asynchrones](#)

extern block , [Déclarer des fonctions étrangères et des variables](#) - [Déclarer des fonctions étrangères et des variables](#)

F

pointeur gras , [tranches](#) , [références aux tranches et aux objets caractéristiques](#) , [pointeurs bruts](#)

Attribut #![feature] , [Attributs](#)

FFI (voir fonctions étrangères)

champs, expressions et , [champs et éléments](#)

Type de fichier , [Utilisation du langage de formatage dans votre propre code](#) , [Recherche](#)

dossier! macro , [Macros intégrées](#)

Fichier :: créer , [Fichiers](#)

Fichier :: ouvrir , [Fichiers](#)

types de noms de fichiers , [OsStr et Path](#)

fichiers , [Fichiers et Répertoires](#) - [Fonctionnalités spécifiques à la plate-forme](#)

fonctions d'accès au système de fichiers , Fonctions d'accès au système de [fichiers](#) - [Fonctions d'accès au système de fichiers](#)

OsStr et Chemin , [OsStr et Chemin](#) - [OsStr et Chemin](#)

Types Path et PathBuf , [Méthodes](#) Path et PathBuf - Méthodes [Path et PathBuf](#)

fonctionnalités spécifiques à la plate-forme , fonctionnalités spécifiques à la plate-forme - [fonctionnalités spécifiques à la plate-forme](#)

lire et écrire , [Lire et écrire des fichiers](#) , [Fichiers](#)

répertoires de lecture , [Répertoires](#) de lecture - [Répertoires de lecture](#)

systèmes de fichiers , [Systèmes de fichiers et outils de ligne de commande](#) - [Rechercher et remplacer](#) , [Fonctions d'accès au système de fichiers](#)

méthode fill_buf , [lecteurs tamponnés](#)

adaptateur de filtre , [carte et filtre](#) - [carte et filtre](#)

adaptateur filter_map , [filter map et flat map](#) - [filter map et flat map](#)

rechercher et remplacer , [rechercher et remplacer](#)

find method , [Parsing Pair Command-Line Arguments](#) , [find, rfind et find_map](#)

find_iter iterator , [Utilisation de base de Regex](#)

méthode find_map , [find, rfind et find_map](#)

types numériques à largeur [fixe](#) , Types numériques à largeur fixe - Types à virgule [flottante](#)

flate2 crate , [données binaires, compression et sérialisation](#)

aplatir l'adaptateur , [aplatir - aplatir](#)

adaptateur flat_map , [filter_map et flat_map](#) - [filter_map et flat_map](#)

littéraux à virgule flottante , [Types à virgule flottante](#)

types à virgule flottante , Types à virgule [flottante](#) - [Types](#) à virgule flottante , [max, min](#) , [Formatage des nombres](#)

analyses sensibles au flux , [boucle Why Rust Has](#)

méthode flush , [Méthodes par défaut](#)

Architecture de flux , [Utilisation efficace des fermetures](#)

module fmt , [Formatage de vos propres types](#)

fn mot-clé , [Fonctions Rust](#) , [Déclarations](#)

Trait Fn , [FnMut](#)

type fn , [Rappels](#)

Trait FnMut , [FnMut](#) - [FnMut](#) , [from_fn et successeurs](#)

Trait FnOnce , [FnOnce](#) - [FnOnce](#)

fnv crate , [Utilisation d'un algorithme de hachage personnalisé](#)

méthode fold , [itérateurs](#) , [fold et rfold](#)

boucle for , [gestion des arguments de la ligne de commande](#)

contrôler le flux dans , [contrôler le flux dans les boucles](#)

IntoIterator , [L'itérateur et les traits](#) IntoIterator , [Implémentations](#)

IntoIterator , [Implémentations](#) [IntoIterator](#)

fonctions étrangères , [Fonctions étrangères](#) - [Une interface sécurisée](#)

[pour libgit2](#)

déclaration de fonctions et de variables étrangères , [Déclaration de fonctions et de variables étrangères](#) - [Déclaration de fonctions et de variables étrangères](#)

recherche de représentations de données communes , [Recherche de représentations de données communes](#) - [Recherche de représentations de données communes](#)

à partir des bibliothèques , [Utilisation des fonctions des bibliothèques](#) - [Utilisation des fonctions des bibliothèques](#)

interface brute vers libgit2 , [Une interface brute vers libgit2](#) - [Une interface brute vers libgit2](#)

interface sécurisée vers libgit2 , [Une interface sécurisée vers libgit2](#)

- [Une interface sécurisée vers libgit2](#)

code non sécurisé et , [Blocs non sécurisés](#)

parallélisme fork-join , [Parallélisme fork-join](#) - [Revisiter l'ensemble de Mandelbrot](#)

gestion des erreurs sur les threads , [Gestion des erreurs sur les threads](#)

Rendu de l'ensemble de Mandelbrot , [Revisiter l'ensemble de Mandelbrot](#) - [Revisiter l'ensemble de Mandelbrot](#)

Bibliothèque Rayon , [Rayon](#) - [Rayon](#)

partagé des données immuables entre les threads , [Partage de données immuables entre les threads](#) - [Partage de données immuables entre les threads](#)

engendrer et rejoindre , [engendrer et rejoindre](#) - [engendrer et rejoindre](#)

paramètres de format , [Valeurs de formatage](#)

format! macro , [chaîne](#) , [valeurs de formatage](#)

formatage des arguments, par index ou nom , [référence aux arguments par index ou nom](#)

formatage des nombres , [Formatage des nombres](#) - [Formatage des nombres](#)

valeurs de formatage , [Valeurs](#) de formatage - [Utilisation du langage de formatage dans votre propre code](#)

Valeurs booléennes , [Mise en forme d'autres types](#)

pour le débogage , [Formatage des valeurs pour le débogage](#) - [Formatage des pointeurs pour le débogage](#)

Trait d'affichage , [Conversion d'autres types en chaînes](#) , [Pointeurs bruts](#)

largeurs et précisions dynamiques , largeurs et précisions [dynamiques](#)

types d'erreurs , [Formatage d'autres types](#)

notation de directive de format de chaîne , [formatage de vos propres types](#)

langage de formatage dans votre propre code , [Utilisation du langage de formatage dans votre propre code](#) - [Utilisation du langage de formatage dans votre propre code](#)

implémenter des traits pour vos propres types , [Formatting Your Own Types](#) - [Formatting Your Own Types](#)

types d'adresses de protocole Internet , [Formatage d'autres types](#)

Trait de pointeur , [Pointeurs bruts](#)

se référant aux arguments par index ou nom , [se référant aux arguments par index ou nom](#)

exemples de chaînes , [Formatage des valeurs](#)

valeurs de texte , [Formatage des valeurs de texte](#) - [Formatage des valeurs de texte](#)

format_args ! macro , [Formatage des valeurs](#) , [Utilisation du langage de formatage dans votre propre code](#)

méthode for_each , [for each et try for each](#)

fonctions libres , [Définir des méthodes avec impl](#)

From trait , [From and Into](#) - [From and Into](#) , [Utilisation de traits avec des macros](#)

Trait FromIterator , Création [de collections : collect et FromIterator](#) , [Le trait Extend](#) , [Création de valeurs de chaîne](#)

FromStr trait , [Analyse d'autres types à partir de chaînes](#)

Méthode from_digit , [Gestion des chiffres](#)

méthode from_fn , [from fn et successeurs](#) - [from fn et successeurs](#)

fonction from_slice , [Fonctions associées au type](#)

Méthode from_str , [Gestion des arguments de la ligne de commande](#)

module fs , [Fonctions d'accès au système de fichiers](#)

Appels de méthode entièrement qualifiés , Appels de méthode entièrement qualifiés - [Appels de méthode entièrement qualifiés](#)

arguments de fonction, réception de références en tant que , [Réception de références en tant qu'arguments de fonction](#) - [Réception de références en tant qu'arguments de fonction](#)

pointeurs de fonction (type fn) , [Callbacks](#)

langage fonctionnel , [Vecteurs](#)

les fonctions

associés , [Définir des méthodes avec impl](#)

async , [Fonctions asynchrones et expressions Await](#) - [Appel de fonctions asynchrones à partir de code synchrone : block on](#) , Création

de fonctions asynchrones [à partir de blocs asynchrones](#)

appelant , [Appels de fonction et de méthode](#)

const , [variables globales](#)

accès au système de fichiers , Fonctions d'accès au système de [fichiers](#) - [Fonctions d'accès au système de fichiers](#)

étranger (voir fonctions étrangères)

gratuit , [Définir des méthodes avec impl](#)

générique , [Arguments de ligne de commande de paire d'analyse](#) ,
[Types fondamentaux](#) , [Fonctions générées et paramètres de type](#) -
[Fonctions générées et paramètres de type](#)

passage de références à , [passage de références à des fonctions](#)
syntaxe pour , [Fonctions Rust](#) - [Fonctions Rust](#)
associées au type , Fonctions associées au type , [Fonctions associées](#)
[au type](#)

types , types de [fonction et de fermeture](#) - Types [de fonction et de](#)
[fermeture](#)

non sécurisé , [Code](#) non sécurisé , [Fonctions](#) non sécurisées - Fonc-
tions [non sécurisées](#)

adaptateur de fusible , [fusible](#) , [Futures](#)

Trait futur , [Futures](#) - [Futures](#)

Trait FutureExt , [la fonction principale du client](#)

futurs , [De synchrone à asynchrone](#) - [Un vrai client HTTP asynchrone](#) ,
[Futurs primitifs et exécuteurs : quand un futur mérite-t-il d'être à nou-
veau interrogé ?](#) - [Implémentation de block_on](#)

blocs asynchrones , [Blocs asynchrones](#) - [Création de fonctions asyn-
chrones à partir de blocs asynchrones](#)

fonctions asynchrones , fonctions asynchrones [et expressions d'at-
tente](#) - [Appel de fonctions asynchrones à partir de code synchrone :](#)
[block_on](#) , [création de fonctions asynchrones à partir de blocs](#)
[asynchrones](#)

caisse de client HTTP asynchrone , [un vrai client HTTP asynchrone](#)
attendre l'expression , [les fonctions asynchrones et les expressions](#)
[d'attente](#)

[block_on](#) , [Appel de fonctions asynchrones à partir de code syn-
chrone : block_on](#) - [Appel de fonctions asynchrones à partir de code](#)
[synchrone : block_on](#) , [Implémentation de block_on](#) - [Implémen-
tation de block_on](#)

emprunt et , génération de [tâches asynchrones](#)

comparaison de conceptions asynchrones , [Comparaison de](#)
[conceptions asynchrones](#)

implémente Send , [mais votre futur implémente-t-il Send ?](#) - [Mais](#)
[votre futur outil envoie-t-il ?](#)

calculs de longue durée , calculs de [longue durée : yield now et](#)
[spawn blocking](#) - calculs de [longue durée : yield now et](#)
[spawn blocking](#)

épinglage , [Épinglage](#) - [Le trait de désépinglage](#)

générant des tâches asynchrones , [Générant](#) des tâches asynchrones - [Générant](#) des tâches asynchrones , Générant des tâches asynchrones [sur un pool de threads](#)
[spawn_blocking](#) , [Appel des Wakers : spawn_blocking](#) - [Appel des Wakers : spawn_blocking](#)

g

GapBuffer , [Exemple : GapBuffer](#) - [Sécurité anti-panique dans le code non sécurisé](#)
garbage collection , [Types de pointeurs](#) , [Propriété et déplacements](#) , [Capture de variables](#)
fonction gcd , [Fonctions Rust](#) , [Servir des pages sur le Web](#)
code générique , [Traits et génériques](#) , [Traits et génériques](#)
types associés et , [Types associés \(ou Fonctionnement des itérateurs\)](#) - [Types associés \(ou Fonctionnement des itérateurs\)](#)
Const , [Const associé](#)
fonctions génériques , [Analyse d'arguments de ligne de commande de paires](#) , [Types fondamentaux](#) , [Fonctions génériques et paramètres de type](#) - [Fonctions génériques et paramètres de type](#)
traits génériques , traits [génériques \(ou comment fonctionne la surcharge d'opérateur\)](#) - [impl Trait](#)
Implémentations IntoIterator et , [IntoIterator](#)
limites de la rétro-ingénierie , [Limites de la](#) rétro-ingénierie - Li-
mites [de la rétro-ingénierie](#)
objets de trait versus , [lequel utiliser](#) - [lequel utiliser](#)
collections génériques, chaînes en tant que , [chaînes en tant que collections génériques](#)
énumérations génériques , [Énumérations](#) génériques - [Énumérations génériques](#)
fonctions génériques
 avec des paramètres constants , [des fonctions génériques et des paramètres de type](#)
paramètres génériques
 constantes , [structures génériques avec paramètres constants](#) ,
 [fonctions génériques et paramètres de type](#)
structures génériques , [Ce qu'est réellement l'ensemble de Mandelbrot](#) , [Structures](#) génériques - [Structures génériques](#)
swaps génériques , [Tuples](#)
types génériques

à paramètres constants , [structures génériques à paramètres constants](#)

méthode get , [mutabilité intérieure](#)

Fonction get_form , [Concurrence](#)

Fonction get_index , [Servir des pages sur le Web](#)

git2-rs crate , [Fonctions étrangères](#)

boucle d'événements globale par rapport aux exécuteurs Rust , [Comparaison des conceptions asynchrones](#)

variables globales , [Variables](#) globales - [Variables globales](#)

utilitaire grep , [lecture des lignes](#)

gardes , gardes de [match](#)

H

plages semi-ouvertes (exclusives à la fin) , [champs et éléments](#)

gammes semi-ouvertes , [possibilités multiples assorties](#)

méthode handle.join , [Gestion des erreurs sur les threads](#)

méthode de hachage , [Utilisation d'un algorithme de hachage personnalisé](#)

Trait de hachage , [Hachage - Utilisation d'un algorithme de hachage personnalisé](#)

Hachage , [hachage](#)

Trait HashMap , [Structures de données riches à l'aide d'énumérations](#)

HashMap::with_capacity , [HashMap<K, V> et BTreeMap<K, V>](#)

Type de collection HashMap<K, V> , [HashMap<K, V> et BTreeMap<K, V>](#) - [Itération de carte](#)

HashSet::new , [HashSet<T> et BTreeSet<T>](#)

HashSet::with_capacity , [HashSet<T> et BTreeSet<T>](#)

Type de collection HashSet<T> , [HashSet<T> et BTreeSet<T>](#) - [Opérations sur l'ensemble](#)

Table de hachage , [partage contre mutation](#)

méthode heap.peek , [BinaryHeap<T>](#)

méthode heap.peek_mut , [BinaryHeap<T>](#)

Méthode heap.pop , [BinaryHeap<T>](#)

méthode heap.push , [BinaryHeap<T>](#)

littéral numérique hexadécimal , [types entiers](#) , [caractères](#)

hexadécimal, formatage des nombres dans , [Formatage des nombres](#)

Caisse de client HTTP , [un vrai client HTTP asynchrone](#)

macros hygiéniques , [cadrage et hygiène](#)

je

if expression , [Rust Functions](#) , [if et match](#) - [if let](#)
si let expressions , [si let](#)
fichiers image, pour l'ensemble de Mandelbrot , [écriture de fichiers image](#) - [écriture de fichiers image](#)
espace image, mappage au plan des nombres complexes , [mappage des pixels aux nombres complexes](#)
références immuables , [Références](#)
bloc impl , [Définir des méthodes avec impl](#) - [Fonctions associées au type](#)
trait impl , trait [impl](#) - trait [impl](#)
importations , [chemins et importations](#)
Méthode inbound.lines , [Réception de paquets : Plus de flux asynchrones](#)
comprendre! macro , [Macros intégrées](#)
inclure_octets ! macro , [Macros intégrées](#)
include_str ! macro , [Macros intégrées](#)
Index trait , [Index et IndexMut](#) - [Index et IndexMut](#)
contenu indexé , [Déplacements et contenu indexé](#) - [Déplacements et contenu indexé](#) , [Référence aux arguments par index ou nom](#) , [Envoi de valeurs](#) - [Exécution du pipeline](#) , [Transfert de presque tous les itérateurs vers un canal](#) - [Transfert de presque tous les itérateurs vers un canal](#)
IndexMut trait , [Index et IndexMut](#) - [Index et IndexMut](#)
boucles infinies , [Boucles](#)
Attribut #[inline] , [Attributs](#)
doublure , [Performance de fermeture](#)
entrée et sortie , [Entrée et sortie](#) - Mise en [réseau](#)
fichiers et répertoires , [Fichiers et répertoires](#) - [Fonctionnalités spécifiques à la plate-forme](#)
réseautage , [réseautage](#) - [réseautage](#)
lecteurs et écrivains , [Lecteurs et écrivains](#) - [Données binaires, compression et sérialisation](#)
inspecter l'adaptateur , [inspecter](#)
installation, Rust , [rustup et Cargo](#) - [rustup et Cargo](#)
Littéraux entiers , [Types entiers](#) , Types à virgule [flottante](#) , [Littéraux, Variables et Caractères génériques dans les modèles](#)
types entiers , [Types entiers](#) - Types [entiers](#) , [Formatage des nombres](#)

entiers , [Fonctions de Rust](#)

conversion de caractères vers/de , [Conversions vers et depuis des entiers](#)

conversion en pointeurs bruts , [Pointeurs bruts](#)

division par zéro panique , [opérateurs arithmétiques, binaires, de comparaison et logiques](#)

Points communs de type Rust et C , [Trouver des représentations de données communes](#)

tests d'intégration , [Tests d'intégration](#)

mutabilité intérieure , [Rc et Arc : Propriété partagée](#) , [Mutabilité intérieure - Mutabilité intérieure](#)

types d'adresses de protocole Internet, formatage , [Formatage d'autres types](#)

Dans le trait , [De et Dans - De et Dans](#)

IntoIter, type associé de , [The Iterator et IntoIterator Traits](#)

Trait IntoIterator , [Les traits Iterator et IntoIterator - Les traits Iterator et IntoIterator](#) , [Implémentations IntoIterator - Implémentations IntoIterator](#) , [Implémentation de vos propres itérateurs](#)

into_iter iterator , [Un programme Mandelbrot concurrent](#)

Méthode into_iter , [Les traits Iterator et IntoIterator](#)

erreurs d'invalidation , [Collections](#) , [Rust exclut les erreurs d'invalidation](#)

invariants, mutex et , [Qu'est-ce qu'un mutex ?](#) , [mutex empoisonnés](#)

index inversé , [envoi de valeurs - exécution du pipeline](#) , [transfert de presque tous les itérateurs vers un canal - transfert de presque tous les itérateurs vers un canal](#)

invoking wakers, in spawn_blocking , [Invoking Wakers: spawn blocking - Invoking Wakers: spawn_blocking](#)

module io , [Lecteurs et écrivains](#)

Type IpAddr , [Analyse d'autres types à partir de chaînes](#) , [Formatage d'autres types](#)

modèles irréfutables , [où les modèles sont autorisés](#)

type isize , [Types entiers](#)

déclarations d'articles , [Déclarations](#)

éléments , [Modules](#) , [Attributs - Attributs](#)

méthode iter , méthodes [iter et iter_mut](#) , [implémentations IntoIterator](#)

Méthode iter.collect , [HashMap<K, V> et BTreeMap<K, V>](#) , [HashSet<T> et BTreeSet<T>](#) , [Création de valeurs de chaîne](#)

type itérable , [les traits Iterator et IntoIterator](#)

itérer

emprunt et , [by ref](#)

sur une carte , [Itération de carte](#)

sur des ensembles , [itération d'ensemble](#)

sur texte , [Itération sur texte - Itération sur texte](#)

adaptateurs d'itérateur (voir méthodes d'adaptateur)

Méthodes Iterator , [Les traits Iterator et IntoIterator](#)

Trait Iterator , [Traits qui définissent les relations entre les types](#) , [Les traits Iterator et IntoIterator](#) , [Implémentation de vos propres itérateurs](#)

itérateurs , [Gestion des arguments de la ligne de commande](#) , [Itérateurs - Implémentation de vos propres itérateurs](#)

méthodes d'adaptateur , [adaptateurs d'itérateur - cycle](#)

types associés et , [Types associés \(ou Fonctionnement des itérateurs\) - Types associés \(ou Fonctionnement des itérateurs\)](#)

consommer (voir consommer des itérateurs)

création , [Création d'itérateurs - Autres sources d'itérateurs](#)

implémentation pour vos propres types , [Implémentation de vos propres itérateurs - Implémentation de vos propres itérateurs](#)

dans la bibliothèque standard , [Autres sources d'itérateurs](#)

traits , [Les traits Iterator et IntoIterator - Les traits Iterator et IntoIterator](#)

méthode iter_mut , méthodes [iter et iter_mut](#) , [implémentations IntoIterator](#)

J

Java

ConcurrentModificationException , [partage contre mutation](#)

relation objet-mutex dans [Qu'est-ce qu'un mutex ?](#)

JavaScript, fonction asynchrone , [Comparaison de conceptions asynchrones](#)

méthode de jointure

combinaison de chaînes , [Chaîne](#)

sur les itérateurs parallèles rayon , [Rayon](#)

en attente de thread , [spawn et join](#)

JSON (JavaScript Object Notation) , [Structures de données riches à l'aide d'énumérations](#)

json ! macro , [Construire le json! Macro - Importation et exportation de macros](#)

types de fragments , [Types](#) de fragments - [Types de fragments](#)
importation et exportation , [Importation et exportation de macros](#) -
[Importation et exportation de macros](#)
récurivité dans , Récursivité [dans les macros](#)
cadrage et hygiène , [cadrage et hygiène](#) - [cadrage et hygiène](#)
utiliser des traits avec , [Utiliser des traits avec des macros](#) - [Utiliser des traits avec des macros](#)

K

Garde, Daniel
Le petit livre des macros Rust , Au- [delà des macro règles !](#)
argument clé, map , [HashMap<K, V>](#) et [BTreeMap<K, V>](#)

L

traits d' extension du langage , traits [utilitaires](#)
dernière méthode , [dernière](#)
Jeu de caractères Latin-1 , [ASCII, Latin-1 et Unicode](#)
lazy_static crate , [Création de valeurs Regex paresseusement](#) , [Variables globales](#)
len method , [Vectors](#) , [Strings in Memory](#) , [position](#), [rposition](#) et
[ExactSizeIterator](#)
Instruction let , [Fonctions Rust](#) , [Plus d'opérations qui bougent](#) ,
[Déclarations](#)
Li, Peng , [les programmeurs systèmes peuvent avoir de belles choses](#)
libgit2 , [Fonctions étrangères](#) , [Utilisation des fonctions des bibliothèques](#) - [Utilisation des fonctions des bibliothèques](#)
interface brute vers , [Une interface brute vers libgit2](#) - [Une interface brute vers libgit2](#)
interface sécurisée vers , [Une interface sécurisée vers libgit2](#) - [Une interface sécurisée vers libgit2](#)
bibliothèques , [Transformer un programme en bibliothèque](#) - [Transformer un programme en bibliothèque](#)
doc-tests , [Doc-Tests](#) - [Doc-Tests](#)
documentation , [Documentation](#) - [Doc-Tests](#)
fonctions étrangères de , [Utilisation des fonctions des bibliothèques](#)
- [Utilisation des fonctions des bibliothèques](#)
Répertoire src/bin , [Le répertoire src/bin](#) - [Le répertoire src/bin](#)
tiers (voir caisses)

durée de vie

paramètres pour les fonctions génériques , les fonctions [génériques et les paramètres de type](#)

paramètres pour les références , [Sécurité](#) des références , [Réception de références en tant qu'arguments de fonction - Omission des paramètres de durée de vie](#)

contraintes de référence , [pointeurs bruts](#)

structs with , [Generic Structs with Lifetime Parameters](#)

ligne! macro , [Macros intégrées](#)

méthodes lines, sur les flux d'entrée , [try_fold et try_rfold](#) , [Prise d'entrée utilisateur : flux asynchrones](#)

Attribut #[link] , [Utilisation des fonctions des bibliothèques](#)

Linux

Paquet de rouille pour , [rustup et Cargo](#)

Utilisation des fonctions des bibliothèques , [Utilisation des fonctions des bibliothèques](#)

Littéraux , dans les modèles , [Littéraux, Variables et Caractères génériques dans les modèles](#)

Le Petit Livre des Macros Rust (Keep) , Au- [delà des macro règles !](#)

méthode de verrouillage , [Mutex<T>](#)

verrouillage des données

mutex , [Mutex<T> - Canaux multiconsommateurs utilisant des mutex](#)

verrous de lecture/écriture , verrous de [lecture/écriture \(RwLock<T>\)](#)

enregistrement

canaux pour , Au- [delà des pipelines](#)

pointeurs de formatage pour , [Utilisation du langage de formatage dans votre propre code](#)

valeurs de formatage pour , [Valeurs de formatage pour le débogage](#)

opérateurs logiques , opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

log_syntax ! macro , [Macros de débogage](#)

calculs de longue durée, programmation asynchrone, calculs de longue durée : [yield now et spawn blocking](#) - calculs de [longue durée : yield now et spawn blocking](#)

boucle (pour les boucles infinies) , [Boucles](#)

expressions de bouclage , [Boucles - Flux de contrôle dans les boucles lvalues](#) , [champs et éléments](#)

M

langage machine , [Trouver des représentations de données communes](#)

types de machines, types entiers , [Types](#) entiers - [Types entiers](#)

mot machine , [Types numériques à largeur fixe](#)

macOS

Paquet de rouille pour , [rustup et Cargo](#)

Utilisation des fonctions des bibliothèques , [Utilisation des fonctions des bibliothèques](#)

macros , [Macros](#) - Au- [delà des macro règles !](#)

intégré , Macros [intégrées](#) - [Macros intégrées](#)

débogage , [Macros](#) de débogage - [Macros de débogage](#)

expansion , [Macros](#) , [Principes de base de l'expansion de macro](#) - [Principes de base de l'expansion de macro](#)

types de fragments , [Types](#) de fragments - [Types de fragments](#)

importation et exportation , [Importation et exportation de macros](#) - [Importation et exportation de macros](#)

json ! , [Construire le json! Macro](#) - [Importation et exportation de macros](#)

procédural , Au- [delà des macro règles !](#)

récursivité dans , Récursivité [dans les macros](#)

répétition , [Répétition](#) - [Répétition](#)

cadrage et hygiène , [cadrage et hygiène](#) - [cadrage et hygiène](#)

conséquences imprévues , [Conséquences](#) imprévues - [Conséquences imprévues](#)

utiliser des traits avec , [Utiliser des traits avec des macros](#) - [Utiliser des traits avec des macros](#)

macro_règles ! , Principes de [base des macros](#) , [Types de fragments](#)

fonction main , [Gestion des arguments de la ligne de commande](#) , [S servir des pages sur le Web](#) , [Gestion des erreurs dans main\(\)](#)

Ensemble de Mandelbrot , [Concurrency](#) - [Ce qu'est réellement l'ensemble de Mandelbrot](#)

implémentation simultanée , [Concurrency](#) - [Safety Is Invisible](#)

mappage des pixels aux nombres complexes , [Mappage des pixels aux nombres complexes](#)

analyse des arguments de ligne de commande d'une paire , [Analyse d'arguments de ligne de commande d'une paire](#) - [Analyse d'arguments de ligne de commande d'une paire](#)

traçage , [Tracé de l'ensemble](#)

rendu avec parallélisme fork-join , [Revisiting the Mandelbrot Set](#) -

[Revisiting the Mandelbrot Set](#)

exécution du traceur , [Exécution du traceur de Mandelbrot](#)

écriture de fichiers image , [Écriture de fichiers image](#) - [Écriture de fichiers image](#)

méthodes map (HashMap et BTreeMap)

append method , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode btree_map.split_off , [HashMap<K, V> et BTreeMap<K, V>](#)

clear method , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode contains_key , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode d'entrée (clé) , [Entrées](#)

méthode entry(key).and_modify , [Entrées](#)

méthode entry(key).or_default , [Entrées](#)

méthode entry(key).or_insert , [Entrées](#)

méthode entry(key).or_insert_with , [Entrées](#)

méthode d'extension , [HashMap<K, V> et BTreeMap<K, V>](#)

get method , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode get_mut , [HashMap<K, V> et BTreeMap<K, V>](#)

insert method , [HashMap<K, V> et BTreeMap<K, V>](#)

Méthode into_iter , [Itération de la carte](#)

méthode into_keys , [itération de la carte](#)

Méthode into_values , [itération de la carte](#)

méthode is_empty , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode keys , [Map Itération](#)

méthode len , [HashMap<K, V> et BTreeMap<K, V>](#)

remove method , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode remove_entry , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode de conservation , [HashMap<K, V> et BTreeMap<K, V>](#)

méthode des valeurs , [Map Itération](#)

méthode values_mut , [itération de la carte](#)

adaptateur de carte , [carte et filtre](#) - [carte et filtre](#)

carte et cartographie , [Mappage des pixels aux nombres complexes](#)

BTreeMap<K, V> , [Overview](#) , [HashMap<K, V> et BTreeMap<K, V>](#) -

[Itération de carte](#)

adaptateurs filter_map et flat_map , [map et filter](#) - [filter map et flat map](#)

méthode find_map , [find, rfind et find map](#)

Trait HashMap , [Structures de données riches à l'aide](#)

[d'énumérations](#)

HashMap<K, V> , [Vue](#) d'ensemble , [HashMap<K, V> et BTreeMap<K, V>](#)

[- Itération de carte](#)

mapper et filtrer , [mapper et filtrer](#) - [mapper et filtrer](#)

traits de marqueur , traits [utilitaires](#) , [sécurité des threads : envoyer et synchroniser](#) - [sécurité des threads : envoyer et synchroniser](#) , [mais votre futur implémente-t-il des envois ?](#) - [Mais votre futur outil envoie-t-il ?](#) , [Le trait de détachement](#) - [Le trait de détachement](#) , [Traits dangereux](#)

expression de correspondance , [analyse d'arguments de ligne de commande de paire](#) , [si et correspondance](#) , [modèles](#) - [modèles](#)

déclaration de correspondance , [écriture de fichiers image](#)

allumettes! macro , [Macros intégrées](#)

unions assorties , unions [assorties](#)

Matsakis, Niko , [Rayonne](#)

méthode max , [max, min](#)

méthode max_by , [max_by, min_by](#)

méthode max_by_key , [max_by_key, min_by_key](#)

Type MaybeUninit , [une interface brute vers libgit2](#)

mémoire , [propriété et déménagements](#)

(voir aussi propriété)

disposition de la fermeture dans , [Performances de fermeture](#)

enums in , [Enums in Memory](#)

pointeurs bruts et , [Entrer et sortir de la mémoire](#) - [Entrer et sortir de la mémoire](#)

réinterpréter avec les syndicats , [Réinterpréter la mémoire avec les syndicats](#) - [Syndicats d'emprunt](#)

chaînes en , [chaînes en mémoire](#) - [chaînes en mémoire](#)

types pour représenter une séquence de valeurs dans , des [tableaux, des vecteurs et des tranches](#) - [tranches](#)

ordonnancement de la mémoire, pour les opérations atomiques ,

[Atomics](#)

méthodes

appelant , [Appels de fonction et de méthode](#)

définir avec impl , [Définir des méthodes avec impl](#) - [Fonctions associées au type](#)

Appels de méthode entièrement qualifiés , Appels de méthode entièrement qualifiés - [Appels de méthode entièrement qualifiés](#)

méthode min , [max, min](#)

méthode min_by , [max_by, min_by](#)

méthode min_by_key , [max by key, min by key](#)

Modèle-Vue-Contrôleur (voir MVC)

modules , [Modules - Statique et constantes](#)

bibliothèques et , [Transformer un programme en bibliothèque](#) -

[Transformer un programme en bibliothèque](#)

imbriqués , [modules imbriqués](#)

chemins et importations , [Chemins et importations](#) - [Chemins et importations](#)

prélude , [Modules](#)

dans des fichiers séparés , [Modules dans des fichiers séparés](#)

prélude standard , [Le prélude standard](#)

monomorphisation , [fonctions génériques et paramètres de type](#)

Ver Morris , les [programmeurs systèmes peuvent avoir de belles choses](#)

déplacements , [déplacements - déplacements et contenu indexé](#)

fermetures et , [Fermetures qui volent](#)

construire de nouvelles valeurs , [plus d'opérations qui bougent](#)

flux de contrôle et , [mouvements et flux de contrôle](#)

Types de copie comme exception à , [Types de copie : l'exception](#)

[aux mouvements](#) - [Types de copie : l'exception aux mouvements](#)

contenu indexé et , [Déplacements et contenu indexé](#) - [Déplacements et contenu indexé](#)

passer des valeurs à une fonction , [plus d'opérations qui bougent](#)

renvoyer des valeurs à une fonction , [plus d'opérations qui bougent](#)

assignation à une variable , [Plus d'opérations qui bougent](#)

module mpsc (multiproducteur, monoconsommateur) , [envoi de valeurs , fonctionnalités et performances](#) des canaux , [canaux multiconsummateurs utilisant des mutex](#)

Mul (trait de multiplication) , [Traits génériques \(ou Comment fonctionne la surcharge de l'opérateur\)](#)

plusieurs lecteurs , [Références aux valeurs](#)

programmation multithread , [Concurrence](#) - [Concurrence](#) , [Sécurité des threads : envoi et synchronisation](#)

(voir aussi programmation asynchrone ; concurrence)

Mot- clé mut (mutable) , [Fonctions Rust](#)

référence mut (mutable) , [mut et Mutex](#)

mutabilité, intérieur , [Mutabilité](#) intérieure - [Mutabilité intérieure](#)

références mutables (&mut T) , [Références](#) , [Références aux valeurs](#)

FnMut , [FnMut](#) - [FnMut](#)

Implémentation IntoIterator , [Implémentations IntoIterator](#)

Mutex et , [mut et Mutex](#)

règles pour , [partage contre mutation](#)

références partagées versus , [Références aux valeurs](#) , [Partage versus mutation](#) - [Partage versus mutation](#)

Fractionnement et , [Fractionnement](#) - [Fractionnement](#)

tranche mutable , [tableaux, vecteurs et tranches](#)

état mutable, partagé , [État mutable partagé](#) - [Variables globales](#)

statique mutable , [réception de références en tant qu'arguments de fonction](#) , [statiques et constantes](#) , [blocs non sécurisés](#)

Type de mutex , [La table de groupe : mutex synchrones](#) - [La table de groupe : mutex synchrones](#)

Mutex::new , [Mutex<T>](#)

mutex , [Qu'est-ce qu'un mutex ?](#) - [Qu'est-ce qu'un mutex ?](#)

connexions de chat avec des mutex asynchrones , [Gestion des connexions de chat](#) : mutex asynchrones - [Gestion des connexions de chat : mutex asynchrones](#)

création avec Mutex<T> , [Mutex<T>](#) - [Mutex<T>](#)

impasses et , [impasse](#)

invariants et , [Qu'est-ce qu'un mutex ?](#) , [mutex empoisonnés](#)

limitations , [pourquoi les mutex ne sont pas toujours une bonne idée](#)

canaux multiconsommateurs utilisant , [Canaux multiconsommateurs utilisant des mutex](#)

référence mut et , [mut et Mutex](#)

empoisonné , [mutex empoisonnés](#)

MVC (Model-View-Controller) , [Utilisation efficace des fermetures](#)

N

Structures de champs nommés , [Structures de champs nommés](#) -

Structures [de champs nommés](#)

espaces de noms (voir modules)

Valeurs NaN (not-a-number) , [Comparaisons d'équivalence](#)

Caisse native_tls , Mise en [réseau](#)

modules imbriqués , [Modules imbriqués](#)

module net , Mise en [réseau](#)

réseautage , [réseautage](#) - [réseautage](#)

newtypes , [Tuple-Like Structs](#) , [Exemple : Un type de chaîne ASCII efficace](#)

méthode suivante , [Types associés \(ou fonctionnement des itérateurs\)](#) ,

[by ref](#) , [Prise d'entrée utilisateur : flux asynchrones](#)

références non mut, fractionnement , [Fractionnement](#) -

[Fractionnement](#)

normalisation, Unicode , [Normalisation](#) - [La caisse de normalisation](#)

[unicode](#)

valeurs not-a-number (NaN) , [comparaisons d'équivalence](#)

Méthodes nth et nth_back , [nth](#),[nth back](#)

nième numéro de triangle , [itérateurs](#)

pointeurs nuls , [les références ne sont jamais nulles](#)

Pointeurs bruts nuls , [Pointeurs bruts](#) , Pointeurs [nullables](#)

références nulles non autorisées , [Références](#) , [Les références ne sont jamais nulles](#)

nombres, complexes , [Analyse d'arguments de ligne de commande de paires](#)

types numériques

Fixed -width , [Types numériques](#) à largeur fixe - Types à virgule

[flottante](#)

types à virgule flottante , Types à virgule [flottante](#) - [Types à virgule](#)

[flottante](#)

types entiers , [Types](#) entiers - Types [entiers](#) , [Formatage des nombres](#)

O

OccupiedEntry type, HashMap et BTreeMap , [Entries](#)

littéral numérique octal , [types entiers](#)

octal, formatage des nombres dans , [Formatage des nombres](#)

méthode offset , [Dangereux de quoi ?](#) , [Pointeurs bruts](#) , [Déréférence-ment des pointeurs bruts en toute sécurité](#) , [Arithmétique des pointeurs](#)

Une règle de définition , [les traits et les autres types de personnes](#)

Structure OpenOptions , [Fichiers](#)

surcharge d'opérateur , [Surcharge](#) d'opérateur - [Autres opérateurs](#)

opérateurs arithmétiques/au niveau du bit , Opérateurs [arithmétiques et au niveau du bit](#) - [Opérateurs d'affectation composés](#)

opérateurs binaires , [Opérateurs binaires](#)

opérateurs d'affectation composés , [Opérateurs d'affectation](#) composés - Opérateurs d' [affectation composés](#)

tests d'égalité , [Comparaisons d' équivalence](#) - [Comparaisons d'équivalence](#)

traits génériques et , [Traits génériques \(ou Fonctionnement de la surcharge de l'opérateur\)](#)

Index et IndexMut , [Index et IndexMut](#) - [Index et IndexMut](#)
limitations sur , [autres opérateurs](#)

comparaisons ordonnées , [Comparaisons ordonnées](#) - [Comparaisons ordonnées](#)

opérateurs unaires , [opérateurs unaires](#)

priorité des opérateurs , [Priorité et Associativité](#)

les opérateurs

arithmétique , [Opérateurs arithmétiques, binaires, de comparaison et logiques](#) , Opérateurs [arithmétiques et binaires](#) - Opérateurs [arithmétiques et binaires](#) , Opérateurs d' [affectation](#) composés - Opérateurs [d'affectation composés](#)

en tant qu'opérateur , [Conversions vers et à partir d'entiers](#)

Opérateurs binaires , [arithmétiques, binaires, de comparaison et logiques](#) , Opérateurs [binaires](#)

opérateurs au niveau du bit , [arithmétiques, au niveau du bit, de comparaison et logiques](#) , opérateurs [d'affectation](#) composés - opérateurs [d'affectation composés](#)

comparaison , [Type booléen](#) , [Utilisation de chaînes](#) , [Comparaison de références](#) , Opérateurs arithmétiques, binaires, de [comparaison et logiques](#) , [max by, min by](#)

égalité , [Comparaisons d' équivalence](#) - [Comparaisons d'équivalence](#)

référence , [Opérateurs de référence](#)

unaire , [opérateurs unaires](#)

Option<&T> , [les références ne sont jamais nulles](#)

option_env ! macro , [Macros intégrées](#)

opérateurs de comparaison ordonnés , [Comparaisons](#) ordonnées - [Comparaisons ordonnées](#)

Ordre ::SeqCst, ordre de la mémoire atomique , [Atomique](#)

règle orpheline , [traits et autres types de personnes](#)

module os , [Fonctionnalités spécifiques à la plate-forme](#)

Type de chaîne OsStr , [OsStr et Path](#) - [OsStr et Path](#)

Type sortant , [gestion des connexions de chat : mutex asynchrones](#)

opérations de débordement , [Arithmétique vérifiée, enveloppante, saturée et débordante](#)

Propriété , [Propriété et Déménagements](#) - [Rc et Arc : Copropriété](#)

Arc , [Rc et Arc : Copropriété](#) - [Rc et Arc : Copropriété](#)

C++ versus Rust , [Propriété](#) - [Propriété](#)

Vache , [Emprunter et Propriétaire au travail : la vache humble](#)

itération et , [Gestion des arguments de la ligne de commande](#)

déplacements , [déplacements](#) - [déplacements et contenu indexé](#)

Rc , [Rc et Arc : Copropriété](#) - [Rc et Arc : Copropriété](#)

partagé , [Rc et Arc : Propriété partagée](#) - [Rc et Arc : Propriété](#)

[partagée](#)

type propriétaire , [OsStr et Path](#)

P

panique , [Fonctions Rust](#) , [Panique](#) - [Abandon](#)

abandon , [Abandon](#)

mutex empoisonnés , mutex [empoisonnés](#)

sécurité dans un code dangereux , [Sécurité panique dans un code](#)
[dangereux](#)

dérouler , [dérouler](#) - [dérouler](#)

panique! macro , [Panique](#) , [Valeurs de formatage](#) , [Macros](#)

programmation parallèle , la programmation [parallèle est apprivoisée](#)
(voir aussi concurrence)

ParallèleItérateur , [Rayon](#)

paramètres

formatage , [Valeurs](#) de formatage , Valeurs [de formatage pour le débogage](#) , [Largeurs et précisions dynamiques](#)

lifetime , [Reference Safety](#) , [Recevoir des références comme arguments de fonction](#) - [Omettre les paramètres](#) de durée de vie , les [fonctions génériques et les paramètres de type](#)

type , [Analyse d'arguments de ligne de commande de paires](#) , [Structures](#) génériques , [Fonctions génériques et paramètres de type](#) , [Comparaisons d'équivalence](#)

méthode d' analyse , [analyse d'autres types à partir de chaînes](#)

fonction parse_args , [lecture et écriture de fichiers](#)

fonction parse_complex , [analyse des arguments de ligne de commande de la paire](#)

fonction parse_pair , [Analyse des arguments de la ligne de commande d'une paire](#)

Trait PartialEq , [Comparaisons](#) d'équivalence - [Comparaisons](#)
[d'équivalence](#)

Trait PartialOrd , [Comparaisons](#) ordonnées - [Comparaisons ordonnées](#)
méthode de partition , [partition](#)
méthode part_iter , [Rayon](#)
Type de chemin , [méthodes iter et iter_mut](#) , [formatage des valeurs de texte](#) , [OsStr et Path](#) - [méthodes Path et PathBuf](#)
 méthode des ancêtres , [Méthodes Path et PathBuf](#)
 méthode des composants , [Méthodes Path et PathBuf](#)
 méthode d'affichage , [Méthodes Path et PathBuf](#)
 méthode nom_fichier , [Méthodes Path et PathBuf](#)
 méthode is_absolute , [Méthodes Path et PathBuf](#)
 méthode is_relative , [Méthodes Path et PathBuf](#)
 join method , [Path et PathBuf Methods](#)
 méthode parent , [Méthodes Path et PathBuf](#)
 méthode to_str , méthodes [Path et PathBuf](#)
 méthode to_string_lossy , [Méthodes Path et PathBuf](#)
Path::new method , [Méthodes Path et PathBuf](#)
Type PathBuf , [Méthodes Path et PathBuf](#)
chemins, bibliothèque standard , [Chemins et importations](#) - [Chemins et importations](#)
patrons , [Patrons](#) - [La vue d'ensemble](#)
 @ Patterns , [Reliure avec @ Patterns](#)
 array , [Array et Slice Patterns](#)
 éviter les erreurs de syntaxe lors de la correspondance dans les macros , [Éviter les erreurs de syntaxe lors de la correspondance](#)
 gardes , gardes de [match](#)
 littéraux dans , [littéraux, variables et caractères génériques dans les modèles](#)
 correspondance des expressions et , [si et correspondance](#)
 faire correspondre plusieurs possibilités avec , [Faire correspondre plusieurs possibilités](#)
 remplir un arbre binaire , [Remplir un arbre binaire](#)
 référence , [Motifs de référence](#) - [Motifs de référence](#)
 recherche et remplacement , [Modèles de recherche de texte](#) - [Recherche et remplacement](#)
 situations qui permettent , [Où les modèles sont autorisés](#)
 slice , [Array et Slice Patterns](#)
 Modèles struct , [tuple et struct](#)
 Tuple , [Tuple et Struct Patterns](#)

variables dans , [littéraux, variables et caractères génériques dans les modèles](#)

caractères génériques dans , [littéraux, variables et caractères génériques dans les modèles](#)

méthode peek , [peekable](#)

Itérateur peekable , [peekable](#)

Type de broche , [pointeurs épingleés](#)

pointeur épingle , Pointeurs épingleés - [Pointeurs épingleés](#)

épingler des contrats à terme , [Épingler](#) - [Le trait de détachement](#)

approche pipeline

programmation simultanée , [Concurrence](#)

étapes de l'itérateur , [Itérateurs](#) , [Création d'itérateurs](#)

plusieurs threads , [envoi de valeurs](#) - [exécution du pipeline](#) , [fonctionnalités et performances du canal](#) , [transfert de presque tous les itérateurs vers un canal](#) - [transfert de presque tous les itérateurs vers un canal](#)

fonction pixel_to_point , [Un programme de Mandelbrot concurrent](#)

tracer, ensemble de Mandelbrot , [tracer l'ensemble](#) , [exécuter le traiteur de Mandelbrot](#)

Caractéristique de formatage du pointeur , [Pointeurs bruts](#)

types de pointeurs , [Types](#) de pointeurs - [Pointeurs bruts](#)

boîtes , [Boîtes](#)

non-propriétaire , [Références aux valeurs](#)

pointeur épingle , Pointeurs épingleés - [Pointeurs épingleés](#)

pointeurs bruts , [Pointeurs bruts](#) , Pointeurs [bruts](#) - [Sécurité anti-panique dans le code dangereux](#)

références (voir références (type pointeur))

pointeurs, restrictions de Rust sur , [propriété et déplacements](#)

PoisonError::into_inner , [mutex empoisonnés](#)

poll method , [Futures](#) , [Long Running Calculs: yield now et spawn blocking](#)

interface d'interrogation, programmation asynchrone , [Futurs](#) - [Futurs](#) , [Comparaison de conceptions asynchrones](#) , [Futurs primitifs et exécuteurs : quand un futur mérite-t-il à nouveau d'être interrogé ?](#) - [Implémentation de block on](#)

polymorphisme , [Traits et Génériques](#)

position method , [position, rposition et ExactSizeIterator](#)

Fonction post_gcd , [Servir des pages sur le Web](#) , [Concurrence](#)

priorité, opérateur , [priorité et associativité](#)

module prelude , [Modules](#) , [Fonctionnalités spécifiques à la plate-forme](#) , [Prise d'entrée utilisateur : flux asynchrones](#)
imprimer! macro , [Valeurs de formatage](#) , [Writers](#)
imprimez ! macro , [Servir des pages sur le Web](#) , [Chaîne](#) , [Valeurs de formatage](#) , [Rédacteurs](#)
imprimez ! méthode , [Erreurs d'impression](#)
fonction print_error , [Ignorer les erreurs](#)
Fonction print_padovan , [Propriété](#)
fonction print_usage , [L'interface de ligne de commande](#)
macros procédurales , Au- [delà des macro règles !](#)
fonction process_files , [Parallélisme Fork-Join](#)
méthode du produit , [Accumulation simple : compte, somme, produit](#)
profileur , [Construire des profils](#)
propagation des erreurs , [propagation des erreurs](#)
protocole
 client et serveur comme asynchrones , [Le protocole](#) - [Le protocole](#)
 types d'adresses de protocole Internet, formatage , [Formatage](#)
 [d'autres types](#)
module ptr , [Entrer et sortir de la mémoire](#)
ptr.copy_to , Entrer [et sortir de la mémoire](#)
ptr.copy_to_nonoverlapping , [Déplacement dans et hors de la mémoire](#)
ptr::copy , [Entrer et sortir de la mémoire](#)
ptr::copy_nonoverlapping , Entrer [et sortir de la mémoire](#)
ptr::read , [Entrer et sortir de la mémoire](#)
ptr::write , [Entrer et sortir de la mémoire](#)
traits de vocabulaire public , traits [utilitaires](#)
Python, affectation dans , [Moves](#) - [Moves](#)

R

méthode de course , [la fonction principale du client](#)
caisse rand , [éléments aléatoires](#)
méthode rand :: thread_rng , [éléments aléatoires](#)
gammes
 fermé , [champs et éléments](#)
 fin-exclusif , [Possibilités Multiples Assorties](#)
 semi-ouvert , [Champs et éléments](#) , [Correspondance de multiples possibilités](#)
 dans les expressions de boucle , [Boucles](#)
 illimité , [correspondant à de multiples possibilités](#)

module raw , [Trouver des représentations de données communes](#) ,
[Une interface brute vers libgit2](#) - [Une interface brute vers libgit2](#)
pointeurs bruts , [Pointeurs bruts](#) , [Code non sécurisé](#) , [Pointeurs bruts](#) -
[Sécurité anti-panique dans le code non sécurisé](#)

déréférencement , [pointeurs bruts](#) , [dangereux de quoi ?](#) , [Blocs non sécurisés](#) , [Déréférencement des pointeurs bruts en toute sécurité](#) -
[Exemple : RefWithFlag](#)

Exemple de GapBuffer , [Exemple : GapBuffer](#) - [Exemple : GapBuffer](#)
entrer/sortir de la mémoire , [entrer et sortir de la mémoire](#) - [entrer et sortir de la mémoire](#)

pointeurs nullables , [pointeurs nullables](#)

sécurité anti-panique dans le code non sécurisé , [Sécurité anti-panique dans le code non sécurisé](#)

arithmétique du pointeur , [Arithmétique du pointeur](#) - [Arithmétique du pointeur](#)

Exemple RefWithFlag , [Exemple : RefWithFlag](#) - [Exemple : RefWithFlag](#)

Tailles et alignements des caractères , [Tailles et alignements des caractères](#)

chaînes brutes , [Littéraux de chaîne](#)

Bibliothèque Rayon (Matsakis et Stone) , [Rayon](#) - [Rayon](#)

Type de pointeur Rc , [Rc et Arc : Propriété partagée](#) - [Rc et Arc : Propriété partagée](#) , [Se passant comme une boîte, Rc ou Arc](#) - [Se passant comme une boîte, Rc ou Arc](#)

Trait de lecture , [entrée et sortie](#) , [lecteurs et écrivains](#)

méthode des octets , [Lecteurs](#)

méthode de la chaîne , [Lecteurs](#)

méthode des lignes , [lecteurs tamponnés](#)

méthode de lecture , [Lecteurs](#)

méthode read_exact , [Lecteurs](#)

méthode read_to_end , [Lecteurs](#)

méthode read_to_string , [Lecteurs](#)

prendre la méthode , [Lecteurs](#)

accès en lecture seule, accès partagé en tant que , [partage contre mutation](#)

verrous de lecture/écriture (RwLock) , verrous de [lecture/écriture](#) ([RwLock<T>](#))

Trait ReadBytesExt , [données binaires, compression et sérialisation](#)
lecteurs , [Lecteurs](#) - [Collectionner les lignes](#)

données binaires, compression, sérialisation , [Données binaires, compression et sérialisation](#) - [Données binaires, compression et sérialisation](#)

tamponné , [Lecteurs tamponnés - Lignes de collecte](#)

lignes de collecte , [Lignes de collecte](#)

fichiers , [Fichiers](#)

autres types , [Autres types de lecteurs et d'enregistreurs](#) - [Autres types de lecteurs et d'enregistreurs](#)

lignes de lecture , [Lignes de lecture](#) - [Lignes de lecture](#)

Chercher le trait , [Chercher](#)

Méthode read_dir , [Lecture des répertoires](#)

Fonction read_numbers , [Utilisation de plusieurs types d'erreurs](#)

fonction read_to_string , [lecture et écriture de fichiers](#)

read_unaligned , Entrer [et sortir de la mémoire](#)

read_volatile , Entrer [et sortir de la mémoire](#)

Type de récepteur , [canaux multiconsommateurs utilisant des mutex](#)

réception de paquets, chat asynchrone , [Réception de paquets : Plus de flux asynchrones](#) - [Réception de paquets : Plus de flux asynchrones](#)

récursivité, macros , Récursivité [dans les macros](#)

Type de cellule de référence

emprunt de méthode , [mutabilité intérieure](#)

méthode emprunter_mut , [mutabilité intérieure](#)

méthode try_borrow , [mutabilité intérieure](#)

méthode try_borrow_mut , [mutabilité intérieure](#)

RefCell::new(value) , [Mutabilité intérieure](#)

Structure RefCell<T> , [Mutabilité intérieure](#)

motifs de référence (réf) , [Motifs](#) de référence - [Motifs de référence](#)

opérateurs de référence , [Opérateurs de référence](#)

type de pointeur à référence comptée (Rc) , [Rc et Arc : Propriété partagée](#) - [Rc et Arc : Propriété partagée](#)

références (type pointeur) , [Références](#) , [Références](#) - [Prendre les armes contre une mer d'objets](#)

affectation , [affectation de références](#)

emprunt , [Emprunter des références à des expressions arbitraires](#) ,

[Emprunter une variable locale](#) - [Emprunter une variable locale](#) ,

[Fermetures qui empruntent](#)

C++ versus Rust , [Références Rust versus Références C++](#)

comparer , [Comparer des références](#)

contraintes sur , [Sécurité de référence](#) - [Structures contenant des références](#) , [Pointeurs bruts](#)

immuable , [Références](#)

Implémentation IntoIterator , [Implémentations IntoIterator](#) itération et , [Gestion des arguments de la ligne de commande](#) paramètres de durée de vie et , [Sécurité des références](#) , [Réception de références en tant qu'arguments de fonction](#) - [Omission des paramètres de durée de vie](#)

mutable (voir références mutables)

null , [Références](#)

pointeurs nuls et , [les références ne sont jamais nulles](#)

passage de références à des fonctions , [passage de références à des fonctions](#)

réception en tant qu'arguments de fonction , [Réception de références en tant qu'arguments de fonction](#) - [Réception de références en tant qu'arguments de fonction](#)

retour , [retour de références](#)

sécurité de , [Sécurité de référence](#) - [Omettre les paramètres de durée de vie](#)

"mer d'objets" et , [Prendre les armes contre une mer d'objets](#) - [Prendre les armes contre une mer d'objets](#)

partagé versus mutable , [Références aux valeurs](#) , [Partage versus mutation](#) - [Partage versus mutation](#)

structures contenant , [Structures contenant des références](#) - [Structures contenant des références](#)

aux références , [Références aux références](#)

aux tranches et aux objets de trait , [Références aux tranches et aux objets de trait](#)

aux valeurs , [Références aux valeurs](#) - [Références aux valeurs](#)

modèles réfutables , [Où les modèles sont autorisés](#)

RefWithFlag<'a, T> , [Exemple : RefWithFlag](#) - [Exemple : RefWithFlag](#)

Structure Regex , [Rechercher et remplacer](#)

Méthode Regex ::captures , [Utilisation de base de Regex](#)

Regex::new constructor , [Construire des valeurs Regex paresseusement](#) - [Construire des valeurs Regex paresseusement](#)

expressions régulières (regex) , [Expressions régulières](#) - [Création de valeurs régulières paresseusement](#)

utilisation de base , Utilisation de base de Regex - [Utilisation de base de Regex](#)

construire des valeurs à la demande , [construire des valeurs régulières paresseusement](#)

macros versus , [Principes de base de l'expansion des macros](#)

opérateurs relationnels , opérateurs [arithmétiques, binaires, de comparaison et logiques](#)

méthode replace_all , [Rechercher et remplacer](#)

Attribut #[repr(C)] , [Recherche de représentations de données communes](#)

Attribut #[repr(i16)] , [Recherche de représentations de données communes](#)

caisse reqwest , [Réseautage](#)

programmation avec contraintes de ressources , [Préface](#)

Type de résultat , [Gestion des arguments de la ligne de commande](#) , [Résultat - Pourquoi des résultats ?](#)

Méthode as_mut , Capture des [erreurs](#)

méthode as_ref , capture des [erreurs](#)

attraper les erreurs , [Gestion](#) des erreurs , [Attraper les erreurs - Attraper les erreurs](#)

traiter les erreurs qui "ne peuvent pas arriver" , [Traiter les erreurs qui "ne peuvent pas arriver"](#)

déclaration d'un type d'erreur personnalisé , [Déclaration d'un type d'erreur personnalisé](#)

méthode err , [Attraper les erreurs](#)

gestion des erreurs sur les threads , [Gestion des erreurs sur les threads](#)

expect method , [Attraper les erreurs](#)

gestion des erreurs dans la fonction main , [Gestion des erreurs dans main\(\)](#)

ignorer les erreurs , [Ignorer les erreurs](#)

Méthode is_err , Capture des [erreurs](#)

Méthode is_ok , Capture des [erreurs](#)

points clés de la conception , [Pourquoi des résultats ?](#)

avec plusieurs types d'erreurs , [Utilisation de plusieurs types d'erreurs - Utilisation de plusieurs types d'erreurs](#)

méthode ok , [Attraper les erreurs](#)

erreurs d'impression , [Erreurs](#) d'impression - [Erreurs d'impression](#)

propagation des erreurs , [propagation des erreurs](#)

alias de type , [Alias de type de résultat](#)

méthode unwrap , [Capture des erreurs](#)

méthode unwrap_or , Capture des [erreurs](#)
méthode unwrap_or_else , capture des [erreurs](#)
expressions de retour , [Fonctions Rust](#) , Fonctions [Rust](#) , [Expressions de retour](#)
adaptateur rev , [Itérateurs réversibles et rev](#) - [Itérateurs réversibles et rev](#)
méthode inverse , [Vecteurs](#)
itérateurs réversibles , [Itérateurs réversibles et rev](#) - [Itérateurs réversibles et rev](#)
méthode rfind , [find, rfind et find_map](#)
méthode rfold , [fold et rfold](#)
Paramètre de type Rhs , [Comparaisons d'équivalence](#)
module racine , [Transformer un programme en bibliothèque](#)
méthode route , [diffusion de pages sur le Web](#)
routeurs, rappels et , [Rappels](#) - [Rappels](#)
méthode rposition , [position, rposition et ExactSizeIterator](#)
Rust , [Un tour de Rust](#) - [Rechercher et remplacer](#)
arguments de ligne de commande , [Gestion des arguments de ligne de commande](#)
de commande - [Gestion des arguments de ligne de commande](#)
interface de ligne de commande , [Systèmes de fichiers et outils de ligne de commande](#) - [L'interface de ligne de commande](#)
communauté , [Plus de belles choses](#)
simultanéité , [Concurrence](#) - La sécurité est invisible
systèmes de fichiers , [Systèmes de fichiers et outils de ligne de commande](#) - [Rechercher et remplacer](#)
rechercher et remplacer , [rechercher et remplacer](#)
fonctions dans , [Fonctions Rust](#) - [Fonctions Rust](#)
installation , [rustup et Cargo](#) - [rustup et Cargo](#)
lecture de fichiers , [Lecture et écriture de fichiers](#)
raisons d'utiliser , [Rust assume la charge pour vous](#) - [Rust facilite la collaboration](#)
règles pour un programme qui se comporte bien , [Comportement indéfini](#)
serveur Web simple , [diffusion de pages sur le Web](#) - [diffusion de pages sur le Web](#)
tests unitaires en , [écriture et exécution de tests unitaires](#)
site web , [rustup et Cargo](#)
commande rustc , [rustup et cargaison](#) , [caisses](#) , [macros de débogage](#)
Commande rustdoc , [rustup et Cargo](#)

rustup , [rustup et Cargo](#) - [rustup et Cargo](#)

RwLock , Verrous en [lecture/écriture \(RwLock<T>\)](#)

S

interface sécurisée vers libgit2 , [Une interface sécurisée vers libgit2](#) -

[Une interface sécurisée vers libgit2](#)

sécurité

fermetures et , [Fermetures et sécurité](#) - [Copier et cloner pour les fermetures](#)

l' invisibilité de , [La sécurité est invisible](#)

avec références , [Sécurité](#) de référence - [Paramètres de durée de vie omis](#)

sécurité des threads avec envoi et synchronisation , [sécurité des threads : envoi et synchronisation](#) - [sécurité des threads : envoi et synchronisation](#)

opérations de saturation , [Arithmétique vérifiée, enveloppante, saturante et débordante](#)

fonction say_hello , fonctions [génériques et paramètres de type](#)

portées et cadrage , [Rayonne](#) , [Cadrage et hygiène](#) - [Cadrage et hygiène](#) recherche

tranches , [Trier et rechercher](#) , [Rechercher et remplacer](#) text , [Conventions de recherche et d'itération](#) - [Recherche et remplacement](#)

Chercher le trait , [Chercher](#)

self argument , [Passing Self as a Box, Rc, or Arc](#) - [Passing Self as a Box, Rc, or Arc](#)

mot- clé self , [chemins et importations](#)

Type de soi , [soi dans les traits](#) - [soi dans les traits](#)

points-virgules suivant les expressions , [blocs et points-virgules](#)

Variable SEMVER , [création de valeurs régulières paresseusement](#)

Send marker trait , [Thread Safety : Send and Sync](#) - [Thread Safety : Send and Sync](#) , [Mais votre futur implémente-t-il Send ?](#) - [Mais votre futur outil envoie-t-il ?](#) , [Traits dangereux](#)

send_as_json , [Envoi de paquets](#)

serde library/crate , [Traits et autres types de personnes](#) , [Données binaires, compression et sérialisation](#) , [Un client et un serveur asynchrones](#)

serde_json crate , [Structures de données riches utilisant des énumérations](#) , [Données binaires, compression et sérialisation](#) , [Un client et un](#)

serveur asynchrones

sérialisation , données binaires, compression et sérialisation

méthode ensembliste , mutabilité intérieure

définir les types (HashMap et BTreemap)

contient method , HashSet<T> et BTreemap<T>

méthode des différences , Opérations sur l'ensemble

méthode get , lorsque des valeurs égales sont différentes

insert method , HashSet<T> et BTreemap<T>

méthode d'intersection , Opérations sur l'ensemble

Méthode is_disjoint , Opérations sur l'ensemble

méthode is_empty , HashSet<T> et BTreemap<T>

Méthode is_subset , Opérations sur l'ensemble

Méthode is_superset , Opérations sur l'ensemble

méthode iter , Définir l'itération

méthode len , HashSet<T> et BTreemap<T>

remove method , HashSet<T> et BTreemap<T>

méthode de remplacement , lorsque des valeurs égales sont différentes

méthode de conservation , HashSet<T> et BTreemap<T>

méthode symmetric_difference , Opérations sur l'ensemble

méthode take , lorsque des valeurs égales sont différentes

méthode union , Opérations sur l'ensemble

ensembles , HashSet<T> et BTreemap<T>

(voir aussi l'ensemble de Mandelbrot)

BTreeSet<T> , HashSet<T> et BTreemap<T> - Opérations sur l'ensemble

HashSet<T> type , HashSet<T> et BTreemap<T> - Opérations sur l'ensemble

ombrage , Déclarations

accès partagé , Partage contre mutation

partagé des données immuables entre les threads , Partage de données immuables entre les threads - Partage de données immuables entre les threads

état mutable partagé , État mutable partagé - Variables globales

atomique , Atomique , Variables globales

Variables de condition (Condvar) , Variables de condition (Condvar)

impasse , impasse

variables globales , Variables globales - Variables globales

canaux multiconsommateurs utilisant mutex , [Canaux multicon-sommateurs utilisant mutex](#)

mut et Mutex , [mut et Mutex](#)

limitations mutex , [pourquoi les mutex ne sont pas toujours une bonne idée](#)

Mutex<T> , [Mutex<T>](#) - [Mutex<T>](#)

mutex empoisonnés , mutex [empoisonnés](#)

verrous de lecture/écriture (RwLock) , verrous de [lecture/écriture](#) ([RwLock<T>](#))

références partagées (&T) , [Références aux valeurs](#)

Les pointeurs de C vers les valeurs const contre , [Partage Versus Mutation](#)

Implémentation IntoIterator , [Implémentations IntoIterator](#)

références mutables versus , [Partage Versus Mutation](#) - [Partage Versus Mutation](#)

règles pour , [Partage contre mutation](#) , [Partage contre mutation](#)

tranche partagée de Ts , [Arrays, Vectors et Slices](#)

Structure partagée, spawn_blocking , [Invocation de Wakers : spawn_blocking](#)

Attribut #[should_panic] , [Tests et Documentation](#)

fonction show_it , [Deref et DerefMut](#)

types entiers signés , [Types entiers](#)

auteur unique, règle de plusieurs lecteurs , [Références aux valeurs](#)

SipHash-1-3 , [Utilisation d'un algorithme de hachage personnalisé](#)

Trait calibré , [Dimensionné](#) - [Dimensionné](#) , [Tailles de caractères et alignements](#)

méthode size_hint , [by_ref](#) , Création [de collections : collect et FromIterator](#)

Fonction size_of_val , [Tailles et alignements de type](#)

adaptateurs skip et skip_while , [skip et skip_while](#)

modèles de tranche , Modèles de [tableau et de tranche](#)

tranches , [tranches](#)

&str (tranche de chaîne) , [Chaîne](#)

méthode binary_search , [Tri et recherche](#)

méthode binary_search_by , [tri et recherche](#)

méthode binary_search_by_key , [tri et recherche](#)

emprunt sous d'autres types de texte , [Emprunt sous d'autres types de texte](#)

méthode bytes , [itération sur le texte](#)

conversion de casse pour les chaînes , [Conversion de casse pour les chaînes](#)

Méthode chars , [Itération sur le texte](#)

Méthode char_indices , [Itération sur le texte](#)

choisir la méthode , [éléments aléatoires](#)

méthode des morceaux , [Fractionnement](#)

Méthode chunks_exact , [Fractionnement](#)

Méthode chunks_exact_mut , [Fractionnement](#)

Méthode chunks_mut , [Fractionnement](#)

comparer , [Comparer des tranches](#)

méthode concat , [Joindre](#)

contient la méthode , [Trier et rechercher](#) , [Rechercher et remplacer](#)

méthodes ends_with , [Comparaison de tranches](#) , [Recherche et remplacement](#)

méthode de recherche , [Recherche et remplacement](#)

première méthode , [Accéder aux éléments](#)

Méthode first_mut , [Accès aux éléments](#)

méthode get , [Accéder aux éléments](#)

Méthode get_mut , [Accès aux éléments](#)

Implémentation IntoIterator , [Implémentations IntoIterator](#)

méthode is_char_boundary , [inspection simple](#)

méthode is_empty , [vecteurs croissants et rétrécissants](#) , [inspection simple](#)

méthode iter , [Fractionnement](#)

itérer sur du texte , [Itérer sur du texte](#) - [Itérer sur du texte](#)

Méthode iter_mut , [Fractionnement](#)

méthode join , [Joindre](#)

jointure dans des tableaux de tableaux , [Joindre](#)

dernière méthode , [Accéder aux éléments](#)

Méthode last_mut , [Accès aux éléments](#)

méthode len , [vecteurs croissants et rétrécissants](#) , [inspection simple](#)

méthode des lignes , [Itération sur le texte](#)

correspond à la méthode , [Itération sur le texte](#)

Méthode match_indices , [Itération sur le texte](#)

sortie aléatoire , [éléments aléatoires](#)

Méthode rchunks , [Fractionnement](#)

Méthode rchunks_exact , [Fractionnement](#)

Méthode rchunks_exact_mut , [Fractionnement](#)

Méthode rchunks_mut , [Fractionnement](#)
références à , [références aux tranches et aux objets de trait](#)
méthode de remplacement , [Recherche et remplacement](#)
méthode replacen , [Recherche et remplacement](#)
méthode inverse , [tri et recherche](#)
Méthode rfind , [Recherche et remplacement](#)
Méthode rmatch_indices , [Itération sur le texte](#)
Méthode rsplit , [Fractionnement](#) , [Itération sur le texte](#)
Méthode rspltn , [Fractionnement](#) , [Itération sur le texte](#)
Méthode rspltn_mut , [Fractionnement](#)
Méthode rsplit_mut , [Fractionnement](#)
Méthode rsplit_terminator , [Itération sur le texte](#)
rechercher , [trier et rechercher](#) , [rechercher et remplacer](#)
méthode shuffle , [éléments aléatoires](#)
slice[range] , [Inspection simple](#)
méthode de tri , [Tri et recherche](#)
tri , [tri et recherche](#)
Méthode sort_by , [Tri et recherche](#)
Méthode sort_by_key , [Tri et recherche](#)
méthode split , [Fractionnement](#) , [Itération sur le texte](#)
méthode spltn , [Fractionnement](#) , [Itération sur le texte](#)
Méthode spltn_mut , [Fractionnement](#)
fractionnement des références non mut , [Fractionnement](#) -
[Fractionnement](#)
Méthode split_ascii_whitespace , [Itération sur le texte](#)
Méthode split_at , [Fractionnement](#) , [Inspection simple](#)
Méthode split_at_mut , [Fractionnement](#)
Méthode split_first , [Fractionnement](#)
Méthode split_first_mut , [Fractionnement](#)
Méthode split_last , [Fractionnement](#)
Méthode split_last_mut , [Fractionnement](#)
Méthode split_mut , [Fractionnement](#)
Méthode split_terminator , [Itération sur le texte](#)
Méthode split_whitespace , [Itération sur le texte](#)
méthodes starts_with , [Comparer les tranches](#) , [Rechercher et remplacer](#)
Méthode strip_prefix , [Trimming](#)
Méthode strip_suffix , [Trimming](#)
méthode d'échange , [Échange](#)

échanger le contenu de , [Échange](#)

Méthode to_lowercase , [Conversion de casse pour les chaînes](#)

Méthode to_owned , [Création de valeurs de chaîne](#)

Méthode to_string , [Création de valeurs de chaîne](#)

Méthode to_uppercase , [Conversion de casse pour les chaînes](#)

Méthode to_vec , [Accès aux éléments](#)

méthode de trim , [Trimming](#)

tailler les cordes , [tailler](#)

méthode trim_matches , [Trimming](#)

UTF-8 et , [Accès au texte en UTF-8 - Production de texte à partir de données UTF-8](#)

méthode windows , [Fractionnement](#)

cas de serpent , [Structures de champ nommé](#)

SocketAddr type , [Mise en forme d'autres types](#)

tri des tranches , [Tri et recherche](#)

fonction d'apparition

pour les tâches asynchrones , [Programmation asynchrone , Création](#) de tâches asynchrones , Création de tâches asynchrones [sur un pool de threads](#)

pour créer des threads , [spawn and join - spawn and join , Error Handling Across Threads](#)

générant des tâches asynchrones , [Générant](#) des tâches asynchrones -

[Générant](#) des tâches asynchrones , Générant des tâches asynchrones [sur un pool de threads](#)

spawn_blocking , Calculs de [longue durée : yield now et spawn blocking](#) - Calculs de [longue durée : yield now et spawn blocking](#) , Appel de [Wakers : spawn blocking - Appel de Wakers : spawn blocking](#)

spawn_local , Génération de [tâches](#) asynchrones - Génération de tâches asynchrones , Génération de [tâches asynchrones sur un pool de threads](#)

méthode d'épissage , [Self in Traits](#)

Répertoire src/bin , [Le répertoire src/bin - Le répertoire src/bin](#)

dérouler la pile , [Dérouler - Dérouler](#)

prélude standard , [Le prélude standard](#)

déclarations, expressions versus , [un langage d'expression](#)

mot- clé statique , [Statique et constantes , Blocs non sécurisés](#)

méthodes statiques , [appels de fonction et de méthode](#)

valeurs statiques (statics) , [Recevoir des références en tant qu'arguments de fonction , Modules](#)

std (bibliothèque standard) , [Chemins et importations](#)

Type Stderr , [autres types de lecteurs et d'enregistreurs](#)

Type Stdin , [autres types de lecteurs et d'enregistreurs](#)

Type StdinLock , [autres types de lecteurs et d'enregistreurs](#)

Type de sortie standard , [autres types de lecteurs et d'enregistreurs](#)

Pierre, Josh , [Rayonne](#)

str::from_utf8 , [Production de texte à partir de données UTF-8](#)

str::from_utf8_unchecked , [Production de texte à partir de données UTF-8](#)

Caractéristique de flux , [prise d'entrée utilisateur : flux asynchrones ruisseaux](#)

flux asynchrones , [Réception de paquets : plus de flux asynchrones](#)
- [Réception de paquets : plus de flux asynchrones](#)

client et serveur en tant qu'asynchrone , [Prise d'entrée utilisateur : flux asynchrones](#)

TcpStream , [Autres types de lecteurs et d'enregistreurs](#) , [Réception de paquets : davantage de flux asynchrones](#)

Types String et str , [Analyse d'arguments de ligne de commande de paires](#) , [Types String - Autres types](#) de type chaîne , [String et str - Chaînes en tant que collections génériques](#)

ajouter du texte , [Ajouter et insérer du texte](#) - [Ajouter et insérer du texte](#)

Ascii , [Exemple : Un type de chaîne ASCII efficace](#) - [Fonctions non sécurisées](#)

emprunt du contenu de la tranche , [Emprunter sous d'autres types de texte](#)

chaînes d'octets , chaînes d' [octets](#)

conversion de casse , [Conversion de casse pour les chaînes](#)

clear method , [Suppression et remplacement de texte](#)

conversion de valeurs non textuelles en , [Conversion d'autres types en chaînes](#) - [Conversion d'autres types en chaînes](#)

création de valeurs de chaîne , [Création de valeurs de chaîne](#)

méthode de vidange , [Suppression et remplacement de texte](#)

méthode extend , [Ajout et insertion de texte](#)

types de noms de fichiers , [OsStr et Path](#)

from_utf8 , [Production de texte à partir de données UTF-8](#)

from_utf8_lossy , [Production de texte à partir de données UTF-8](#)

from_utf8_unchecked , [Production de texte à partir de données UTF-8](#)

en tant que collections génériques , [Chaînes en tant que collections génériques](#)

méthode d'insertion , [Ajout et insertion de texte](#)

insertion de texte , [Ajout et insertion de texte - Ajout et insertion de texte](#)

Méthode insert_str , Ajout [et insertion de texte](#)

itérer sur du texte , [Conventions de recherche et d'itération](#) , [Itérer sur du texte - Itérer sur du texte](#)

chaînes non Unicode , [Autres types de type chaîne](#)

analyse des valeurs de , [analyse d'autres types de chaînes](#)

méthode pop , [Suppression et remplacement de texte](#)

produire du texte à partir de données UTF-8 , [Produire du texte à partir de données UTF-8](#)

méthode push , [Ajout et insertion de texte](#)

Méthode push_str , Ajout [et insertion de texte](#)

Reporter l'allocation , [Reporter l'allocation - Reporter l'allocation](#)

méthode de suppression , [Suppression et remplacement de texte](#)

suppression et remplacement de texte , [Suppression et remplacement de texte](#)

Méthode replace_range , [Suppression et remplacement de texte](#)

recherche de texte , [Conventions de recherche et d'itération - Recherche et remplacement](#)

inspection simple , inspection [simple](#)

chaînes en mémoire , [Chaînes en mémoire - Chaînes en mémoire](#)

rognage du texte , [rognage](#)

méthode truncate , [Suppression et remplacement de texte](#)

UTF-8 et , [Caractères , Accès au texte en UTF-8 - Production de texte à partir de données UTF-8](#)

Littéraux de chaîne , [Littéraux](#) de chaîne , [Chaînes d'octets](#) , [Chaîne](#)

tranche de chaîne (&str) , [Chaîne](#)

String::new , [Création de valeurs de chaîne](#)

String::with_capacity , [Création de valeurs](#) de chaîne , [Ajout et insertion de texte](#)

stringifier ! macro , [Macros intégrées](#)

chaînes et texte , [Chaînes et texte - La caisse de normalisation unicode](#)

caractères (char) , [Caractères \(char\) - Conversions vers et depuis des entiers](#)

valeurs de formatage , [Valeurs](#) de formatage - [Utilisation du langage de formatage dans votre propre code](#)

normalisation , [Normalisation - La caisse de normalisation unicode](#)
passant entre Rust et C , [Trouver des représentations de données communes](#)

expressions régulières , [Expressions régulières - Création de valeurs régulières paresseusement](#)

Arrière-plan Unicode , [Certains](#) arrière-plans Unicode - [Directionnalité du texte](#)

code non sécurisé pour la conversion d'Ascii en chaîne , [Exemple : Un type de chaîne ASCII efficace - Fonctions non sécurisées](#)

Stroustrup, Bjarne

"L'abstraction et le modèle de machine C++" , [et pourtant la rouille est toujours rapide](#)

expression de structure , [Structures de champ nommé](#)

struct patterns , [Tuple et Struct Patterns](#)

structs , [Création de champs Struct pub](#) , [Structs - Mutabilité intérieure](#)

définir des méthodes avec impl , [Définir des méthodes avec impl - Fonctions associées au type](#)

Dérivation de traits communs pour les types struct , [Dérivation de traits communs pour les types struct](#)

générique , [Structures](#) génériques - [Structures génériques](#)

implémentation de hachage , [Hachage](#)

mutabilité intérieure , [Mutabilité](#) intérieure - [Mutabilité intérieure](#)

mise en page , mise en [page de la structure](#)

avec paramètres de durée de vie , [structures génériques avec paramètres de durée de vie](#)

champ- nommé , [Structures de champ nommé](#) - Structures [de champ nommé](#)

références dans , [Structures contenant des références](#) - [Structures contenant des références](#)

tuple-like , [Tuple-Like Structs](#)

unit-like , [Unit-Like Structs](#)

sous- modules , [Modules imbriqués](#)

sous- traits , sous- [traits](#)

méthode des successeurs , [from fn et successeurs](#) - [from fn et successeurs](#)

méthode somme , [Accumulation simple : compte, somme, produit](#)

types de somme , [énumérations et modèles](#)

supertrait , Sous- [traits](#)

instruction switch , [littéraux, variables et caractères génériques dans les modèles](#)

méthode de lien symbolique , Fonctionnalités spécifiques à la [plate-forme](#) - [Fonctionnalités spécifiques à la plate-forme](#)

Type de synchronisation , [Sécurité des threads : envoi et synchronisation](#) - [Sécurité des threads : envoi et synchronisation](#) , [Caractéristiques non sécurisées](#)

objets synchronisés , [Concurrence](#)

canal synchrone, simultanéité , [fonctionnalités et performances du canal](#)

programmation synchrone à asynchrone , [De synchrone à asynchrone](#) - [Un vrai client HTTP asynchrone](#)

blocs asynchrones , [Blocs asynchrones - Création de fonctions asynchrones à partir de blocs asynchrones](#)

fonctions asynchrones , fonctions asynchrones [et expressions d'attente](#) - [Appel de fonctions asynchrones à partir de code synchrone : block_on](#) , [création de fonctions asynchrones à partir de blocs asynchrones](#)

caisse de client HTTP asynchrone , [un vrai client HTTP asynchrone](#)

attendre l'expression , [les fonctions asynchrones et les expressions d'attente](#)

comparaison de conceptions asynchrones , [Comparaison de conceptions asynchrones](#)

contrats à terme (voir contrats à terme)

implémente Send , [mais votre futur implémente-t-il Send ?](#) - [Mais votre futur outil envoie-t-il ?](#)

calculs de longue durée , calculs de [longue durée : yield now et spawn blocking](#) - calculs de [longue durée : yield now et spawn blocking](#)

générant des tâches asynchrones , [Générant](#) des tâches asynchrones - [Générant](#) des tâches asynchrones , Générant des tâches asynchrones [sur un pool de threads](#)

pool de threads, génération de tâches asynchrones à partir de , génération de tâches asynchrones sur [un pool de threads](#)

erreurs de syntaxe, macros et , [Éviter les erreurs de syntaxe lors de la correspondance](#)

appels système , [De synchrone à asynchrone](#)

programmation système , [Préface](#) , [Les programmeurs système peuvent avoir de belles choses](#)

J

<T> , [Tuples](#)

[T] tranches , [Tranches](#)

(voir aussi tranches)

adaptateurs take et take_while , [take et take_while](#) , [by_ref](#)

tâche_local ! macro , [Création de tâches asynchrones sur un pool de threads](#)

TcpStream , [Autres types de lecteurs et](#) d'enregistreurs , [Réception de paquets : davantage de flux asynchrones](#)

modèles, macro , [Principes de base de l'extension de macro](#)

#[test] attribut , [Attributs](#)

tests , [Tests et Documentation - Doc-Tests](#)

doc-tests , [Doc-Tests - Doc-Tests](#)

tests d'intégration , [Tests d'intégration](#)

texte , [ASCII, Latin-1 et Unicode](#)

(voir aussi chaînes et texte ; UTF-8)

ajouter et insérer , [Ajouter et insérer du texte - Ajouter et insérer du texte](#)

ASCII , [ASCII, Latin-1 et Unicode](#) , [Classifier les caractères](#) , [Gérer les chiffres](#) , [Exemple : Un type de chaîne ASCII efficace - Fonctions non sécurisées](#)

conversion de casse , [Conversion de casse pour les caractères](#) ,

[Conversion de casse pour les chaînes](#)

conventions de recherche/itération , [Conventions de recherche et d'itération](#)

directionnalité de , directionnalité du [texte](#)

Exemple de GapBuffer , [Exemple : GapBuffer - Sécurité anti-pa-nique dans le code non sécurisé](#)

itération sur , [Itération sur le texte](#)

supprimer et remplacer , [supprimer et remplacer du texte](#)

recherche , [Conventions de recherche et d'itération - Recherche et remplacement](#)

rognage , [rognage](#)

valeurs de texte, formatage , [Formatage des valeurs de texte - Formatage des valeurs de texte](#)

pool de threads, générant des tâches asynchrones sur , Générant des tâches asynchrones [sur un pool de threads](#)

fils

tâches asynchrones versus , [Programmation asynchrone](#)
fil d'arrière -plan , [Concurrence](#)
canaux et , [Canaux](#) - Au- [delà des pipelines](#)
impasse , [impasse](#)
gestion des erreurs dans , [Gestion des erreurs dans les threads](#)
sécurité avec envoi et synchronisation , [Thread Safety : envoi et synchronisation](#) - [Thread Safety : envoi et synchronisation](#)
partagé des données immuables entre , [Partage de données immuables entre les threads](#) - [Partage de données immuables entre les threads](#)
faire! macro , [Macros intégrées](#)
arbre à jetons , [types de fragments](#)
jetons, modèles de macro , [Principes de base de l'expansion de macro](#)
tokio crate , [Comparaison de conceptions asynchrones](#) , [Un client et un serveur asynchrones](#) , [Groupes de discussion : canaux de diffusion de tokio](#) - [Groupes de discussion : canaux de diffusion de tokio](#)
Trait ToOwned , [ToOwned](#) - [Emprunter et ToOwned au travail : la vache humble](#)
méthode to_owned , [chaîne](#)
méthode to_string , [Rechercher et remplacer](#) , [Chaîne](#)
trace_macros ! macro , [Macros de débogage](#)
objets de trait , [Objets](#) de trait - [Disposition des objets de trait](#)
code générique versus , [lequel utiliser](#) - [lequel utiliser](#)
mise en page , mise en [page de l'objet Trait](#)
références à , [références aux tranches et aux objets de trait](#)
types non calibrés et , [calibrés](#)
traits , [Gestion des arguments de la ligne de commande](#) , [Types de copie : l'exception aux mouvements](#) , [Traits et génériques](#) - [Traits comme base](#)
Const associés , Const [associés](#)
définir et mettre en œuvre , [Définir et mettre en œuvre les traits](#) - [Fonctions associées au type](#)
pour définir les relations entre les types , [Traits qui définissent les relations entre les types](#) - [Consts associés](#)
Appels de méthode entièrement qualifiés , Appels de méthode entièrement qualifiés - [Appels de méthode entièrement qualifiés](#)
impl , [impl Trait](#) - [impl Trait](#)
implémentation pour vos propres types , [Formatage de vos propres types](#) - [Formatage de vos propres types](#)

itérateurs et types associés , [Types associés \(ou fonctionnement des itérateurs\)](#) - [Types associés \(ou fonctionnement des itérateurs\)](#)

avec des macros , [Utilisation de caractéristiques avec des macros](#) -

[Utilisation de caractéristiques avec des macros](#)

pour la surcharge de l'opérateur , [Caractéristiques génériques \(ou Fonctionnement de la surcharge de l'opérateur\)](#) , [Surcharge de l'opérateur](#)

types d'autres personnes et , [Caractéristiques et types d'autres personnes](#) - [Caractéristiques et types d'autres personnes](#)

limites de la rétro-ingénierie , [Limites de la](#) rétro-ingénierie - Limites [de la rétro-ingénierie](#)

Soi en tant que type , [Soi en traits](#) - [Soi en traits](#)

pour les types struct , [Dérivation des traits communs pour les types struct](#)

sous- traits , sous- [traits](#)

fonctions associées au type , Fonctions associées au [type](#) dangereux , [Traits](#) dangereux - [Traits dangereux](#)

utilité (voir traits d'utilité)

dépendances transitives , [Crates](#)

Travis CI , [plus de belles choses](#)

arbres , [Propriété](#)

rognage du texte de la chaîne , [rognage](#)

essayer! macro , [propagation des erreurs](#)

Trait TryFrom , [TryFrom et TryInto](#)

Trait TryInto , [TryFrom et TryInto](#)

méthodes try_fold et try_rfold , [try_fold et try_rfold](#) - [try_fold et try_rfold](#)

méthode try_for_each , [for_each et try_for_each](#)

modèles de tuple , modèles de [tuple et de struct](#)

structures de type tuple , [Structures de type tuple](#)

tuples , [Tuples](#) - [Tuples](#)

alias de type , alias de type , alias de [type de résultat](#) , [utilisation des déclarations pub](#)

alignement des caractères, pointeurs bruts et , [tailles et alignements des caractères](#)

inférence de type , [Types fondamentaux](#)

paramètres de type , [analyse d'arguments de ligne de commande de paires](#) , structures génériques , [fonctions génériques et paramètres de type](#) , [comparaisons d'équivalence](#)

taille des caractères, pointeurs bruts et , [tailles et alignements des caractères](#)

fonctions associées au type , Fonctions associées au type , [Fonctions associées au type](#)

types , [Types fondamentaux](#) - Au- [delà des bases](#)

tableaux , [tableaux](#)

associés , [Types associés \(ou Fonctionnement des itérateurs\)](#) - [Types associés \(ou Fonctionnement des itérateurs\)](#)

moulages et , [Type Moulages](#)

des fermetures et des fonctions , Types de fonctions et de fermetures - [Types de fonctions et de fermetures](#)

gestion des erreurs , [Résultat](#) - [Pourquoi des résultats ?](#) , [Formatage d'autres types](#) , Types [d'erreur et de résultat](#)

nom de fichier , [OsStr et chemin](#)

virgule flottante , Types à virgule [flottante](#) - [Types](#) à virgule flottante , [max,min](#) , [Formatage des nombres](#)

valeurs de mise en forme , [Mise en forme d'autres types](#) , [Mise en forme de vos propres types](#) - [Mise en forme de vos propres types](#)

implémenter vos propres itérateurs , [Implémenter vos propres itérateurs](#) - [Implémenter vos propres itérateurs](#)

Implémentation IntoIterator , [Implémentations IntoIterator](#) - [Implémentations IntoIterator](#)

numeric , [Types numériques à largeur fixe](#) - Types à virgule [flottante](#) , [Formatage des nombres](#)

surcharge de l'opérateur et , [surcharge de l'opérateur](#)

Paramètres , [Structures](#) génériques , [Fonctions génériques et paramètres de type](#) , [Comparaisons d'équivalence](#)

pointeurs (voir types de pointeurs)

pour représenter une séquence de valeurs en mémoire , des [tableaux, des vecteurs et des tranches](#) - [tranches](#)

Dimensionné , [Dimensionné](#) - [Dimensionné](#)

tranches , [tranches](#)

String et str (voir Types String et str)

traits pour ajouter des méthodes à , [Traits et Types d'autres personnes](#)

traits pour définir les relations entre , [Traits qui définissent les relations entre les types](#) - [Consts associés](#)

tuples , [Tuples](#) - [Tuples](#)

non calibré , [calibré](#) - [calibré](#)

défini par l'utilisateur , [Making Struct Fields pub](#)

vecteurs , [Vecteurs - Vecteurs](#)

tu

opérateurs unaires , [opérateurs unaires](#)

gammes illimitées , [possibilités multiples assorties](#)

comportement indéfini , les [programmeurs systèmes peuvent avoir de belles choses](#) , [dangereux de quoi ?](#) , [Comportement indéfini - Comportement indéfini](#)

Unicode , [Certains arrière -plans Unicode - Directionnalité du texte](#)

ASCII et , [ASCII, Latin-1 et Unicode](#)

caractères littéraux , [Caractères](#)

Latin-1 et , [ASCII, Latin-1 et Unicode](#)

normalisation , [Normalisation - La caisse de normalisation unicode](#)

OsStr et , [OsStr et Chemin](#)

directionnalité du texte , [Directionnalité du texte](#)

UTF-8 , [UTF-8 - UTF-8](#)

caisse de normalisation unicode , [La caisse de normalisation unicode - La caisse de normalisation unicode](#)

non implémenté ! macro , [Macros intégrées](#)

unions , [Enums and Patterns](#) , [Unsafe Blocks](#) , [Réinterpréter la mémoire avec les unions - Emprunter des unions](#)

tests unitaires , [écriture et exécution de tests unitaires](#)

type d'unité , [Tuples](#)

structures de type unité , [Structures de type unité](#)

Unix

fichiers et répertoires , Fonctionnalités spécifiques à la [plate-forme](#)

- [Fonctionnalités spécifiques à la plate-forme](#)

canaux , [Canaux - Envoi de valeurs](#)

Trait de marqueur de détachement , [Le trait de détachement - Le trait de détachement](#)

Détacher le trait , [Envoi de paquets](#)

blocs non sécurisés , [pointeurs bruts](#) , [code non sécurisé](#) , [blocs non sécurisés - Exemple : un type de chaîne ASCII efficace](#) , un [bloc non sécurisé ou une fonction non sécurisée ?](#)

code dangereux , [Code dangereux - Syndicats d'emprunt](#)

fonctions étrangères (voir fonctions étrangères)

interface brute libgit2 , [Une interface brute vers libgit2 - Une interface brute vers libgit2](#)

pointeurs bruts (voir pointeurs bruts)

comportement indéfini , [Comportement](#) indéfini - [Comportement indéfini](#)

syndicats , [Réinterpréter la mémoire avec les syndicats](#) - [Emprunter des syndicats](#)

blocs non sécurisés , [pointeurs bruts](#) , [code](#) non sécurisé , [blocs non sécurisés](#) - [Exemple : un type de chaîne ASCII efficace](#) , un [bloc non sécurisé ou une fonction non sécurisée ?](#)

fonctionnalité non sécurisée , [Dangereux de quoi ?](#) - [Pas à l'abri de quoi ?](#)

fonctions non sécurisées , [Code](#) non sécurisé , [Fonctions](#) non sécurisées - Fonctions [non sécurisées](#)

traits dangereux , [Traits](#) dangereux - [Traits dangereux](#)

fonctions non sécurisées , [Code](#) non sécurisé , [Fonctions](#) non sécurisées - Fonctions [non sécurisées](#)

traits dangereux , [Traits](#) dangereux - [Traits dangereux](#)

types entiers non signés , [Types entiers](#)

types non calibrés , [calibrés](#) - [calibrés](#)

dérouler , [dérouler](#) - [dérouler](#)

méthode unwrap , [écriture de fichiers image](#) , gestion des [erreurs dans les threads](#)

use declarations , [Paths and Imports](#) , [Making use Declarations pub](#)

types définis par l'utilisateur , [Making Struct Fields pub](#)

usize type , [Types entiers](#) , [Accès aux éléments](#)

UTF-8 , [UTF-8](#) - [UTF-8](#)

accéder au texte en tant que , [Accéder au texte en tant qu'UTF-8](#)

Méthodes ASCII avec , [Caractères de classification](#)

type de caractère et , [Caractères](#)

OsStr et , [OsStr et Chemin](#) - [OsStr et Chemin](#)

produire du texte à partir de données , [Produire du texte à partir de données UTF-8](#)

Gestion des chaînes et des chaînes , [Chaîne et chaîne](#)

chaînes en mémoire , [chaînes en mémoire](#)

unsafe code and , [Exemple : Un type de chaîne ASCII efficace](#) - [Exemple : Un type de chaîne ASCII efficace](#)

Traits d'utilité , [Traits](#) d'utilité - [Emprunter et posséder au travail : la vache humble](#)

AsRef et AsMut , [AsRef et AsMut](#)

Emprunter et EmprunterMut , [Emprunter et EmprunterMut](#) - [Emprunter et EmprunterMut](#)

Cloner , [cloner](#) , [copier et cloner pour les fermetures](#) , [accéder aux éléments](#)

Copier , [copier](#) , [copier et cloner pour les fermetures](#)

Vache , [Emprunter et Propriétaire au Travail : La Vache Humble](#) , [Différer l'Allocation](#) - [Différer l'Allocation](#)

Par défaut , [Par défaut](#) - [Par défaut](#) , [partition](#)

Deref et Dereferent , [Deref et Dereferent](#) - [Deref et Dereferent](#)

Goutte , [Goutte](#) - [Goutte](#)

De et vers , [De et vers](#) - [De et vers](#) , [Utilisation de caractéristiques avec des macros](#)

Dimensionné , [Dimensionné](#) - [Dimensionné](#)

ToOwned , [ToOwned](#) - [Emprunter et ToOwned au travail : la vache humble](#)

TryFrom et TryInto , [TryFrom et TryInto](#)

module utils , [Types d'erreur et de résultat](#)

V

Type VacantEntry, HashMap et BTreeMap , [Entrées](#)
valeurs

construire à la demande , [construire des valeurs régulières paresseusement](#)

chute , [propriété](#) , [propriété](#) , [fermetures qui tuent](#) - [FnOnce](#)
formatage , [Valeurs](#) de formatage - [Utilisation du langage de formatage dans votre propre code](#) , [Pointeurs bruts](#)

types fondamentaux pour représenter , [Types fondamentaux](#) - [Au-delà des bases](#)

se déplace , [plus d'opérations qui se déplacent](#)

en passant par , [Références aux valeurs](#) , [Implémentations IntoIterator](#) , [Collections](#)

recevoir via les canaux , [recevoir des valeurs](#)

références et , [Références aux valeurs](#) - [Références aux tranches et aux objets de trait](#)

envoi via les canaux , [Envoi de valeurs](#) - [Envoi de valeurs](#)

ensembles et différences de valeurs "égales" , [lorsque des valeurs égales sont différentes](#)

statique , [Recevoir des références en tant qu'arguments de fonction](#) , [Modules](#) , [Statique et Constantes](#)

Types String et str , [Analyse d'autres types à partir de chaînes](#) -

[Conversion d'autres types en chaînes](#)

capture de variables , [Capture de variables](#)

variables

affectation à , [Plus d'opérations qui bougent](#)

emprunt local , [Emprunter une variable locale](#) - [Emprunter une variable locale](#)

condition , [Variables de condition \(Condvar\)](#)

déclaration à partir de bibliothèques étrangères , [Déclaration de fonctions et de variables étrangères](#) - [Déclaration de fonctions et de variables étrangères](#)

global , [Variables globales](#) - [Variables globales](#)

propriété , [Gestion des arguments de la ligne de commande](#) , [Propriété et mouvements](#) - [Rc et Arc : Propriété partagée](#) , [Emprunter et ToOwned at Work : The Humble Cow](#)

dans les modèles , les [littéraux, les variables et les caractères génératifs dans les modèles](#)

statique , [variables globales](#)

Type Vec , [Gestion des arguments de ligne de commande](#)

append method , [Croissance et rétrécissement des vecteurs](#)

bâtimennt de VecDeque , [VecDeque<T>](#)

méthode de la capacité , [vecteurs croissants et rétrécissants](#)

méthode claire , [vecteurs croissants et rétrécissants](#)

méthode de déduplication , [vecteurs croissants et rétrécissants](#)

méthode dedup_by , [vecteurs croissants et rétrécissants](#)

méthode dedup_by_key , [vecteurs croissants et rétrécissants](#)

méthode de drainage , [vecteurs croissants et rétrécissants](#)

méthode d'extension , [vecteurs croissants et rétrécissants](#)

méthode d'insertion , [vecteurs croissants et rétrécissants](#)

méthode pop , [vecteurs croissants et rétrécissants](#)

méthode push , [vecteurs croissants et rétrécissants](#)

supprimer la méthode , [Croissance et rétrécissement des vecteurs](#)

méthode de réserve , [vecteurs croissants et rétrécissants](#)

méthode reserve_exact , [vecteurs croissants et rétrécissants](#)

méthode de redimensionnement , [vecteurs croissants et rétrécissants](#)

méthode resize_with , [vecteurs croissants et rétrécissants](#)

méthode de conservation , [vecteurs croissants et rétrécissants](#)

méthode shrink_to_fit , [vecteurs croissants et rétrécissants](#)

méthode split_off , [vecteurs croissants et rétrécissants](#)

Méthode swap_remove , [Permutation](#)

méthode tronquée , [vecteurs croissants et rétrécissants](#)

méthode with_capacity , [vecteurs croissants et rétrécissants](#)

vec! macro , [Vecteurs](#) , [Vec<T>](#) , [Répétition](#) - [Répétition](#)

Type de collection [Vec<T>](#) , [Tableaux, vecteurs et tranches](#) , [Vecteurs](#) ,

[Vue](#) d'ensemble , [Vec<T>](#) - [Rust exclut les erreurs d'invalidation](#)

accéder aux éléments , [Accéder aux éléments](#) - [Accéder aux éléments](#)

comparaison de tranches , [Comparaison de tranches](#)

vecteurs de croissance/rétrécissement , Vecteurs de croissance et de

rétrécissement - [Vecteurs de croissance et de rétrécissement](#)

erreurs d'invalidation, exclusion , [Rust exclut les erreurs d'invalidation](#)

itération , [itération](#)

rejoindre , [joindre](#)

éléments aléatoires , [Éléments aléatoires](#)

recherche , [tri et recherche](#)

tri , [tri et recherche](#)

Fractionnement , [Fractionnement](#) - [Fractionnement](#)

échange , [échange](#)

[Vec<u8>](#) , [Autres types de lecteurs et d'enregistreurs](#)

[VecDeque](#) , [Aperçu](#) , [VecDeque<T>](#) - [VecDeque<T>](#)

méthode retour , [VecDeque<T>](#)

méthode back_mut , [VecDeque<T>](#)

méthode avant , [VecDeque<T>](#)

méthode front_mut , [VecDeque<T>](#)

méthode make_contiguous , [VecDeque<T>](#)

méthode pop_back , [VecDeque<T>](#)

méthode pop_front , [VecDeque<T>](#)

méthode push_back , [VecDeque<T>](#)

méthode push_front , [VecDeque<T>](#)

[VecDeque::from\(vec\)](#) , [VecDeque<T>](#)

vecteur de Ts , [tableaux, vecteurs et tranches](#)

vecteurs , [vecteurs](#) - [vecteurs](#) , vecteurs [croissants et rétrécissants](#) -

vecteurs [croissants et rétrécissants](#)

versions, fichier , [Spécification des dépendances](#)

barre verticale (|) , [Correspondance de plusieurs possibilités](#)

table virtuelle (vtable) , [mise en page de l'objet Trait](#)

O

waker , [Futures](#) , [Primitive Futures et Executors : Quand un futur mérite-t-il à nouveau d'être interrogé ?](#) - [Implémentation de block_on](#)
pointeurs faibles , [Rc et Arc : propriété partagée](#)
serveur Web, création avec Rust , [Servir des pages sur le Web](#) - [Servir des pages sur le Web](#)
programme bien conduit, Règles de Rust pour , [Comportement indéfini](#)
boucle while let , [Boucles](#)
boucle while , [Fonctions Rust](#) , [Boucles](#) , [Flux de contrôle dans les boucles](#)
opérations sur un ensemble [entier](#) , opérations sur un ensemble entier
caractères génériques , [chemins et importations](#) , [versions](#) , [littéraux](#),
[variables et caractères génériques dans les modèles](#)
les fenêtres
fichiers et répertoires , [Fonctionnalités spécifiques à la plate-forme](#)
[OsStr et](#) , [OsStr et Chemin](#)
Paquet de rouille pour , [rustup et Cargo](#)
Utilisation des fonctions des bibliothèques , [Utilisation des fonctions des bibliothèques](#)
voleur de travail , [rayonne](#)
pools de nœuds de calcul , [simultanéité](#)
espaces de travail , Espaces de [travail](#)
opérations d'habillage , [Arithmétique vérifiée, encapsulée, saturée et débordante](#) , [Arithmétique vérifiée](#) , [encapsulée](#) , [saturée](#) et [débordante](#)
méthode wrap_offset , [pointeurs bruts](#) , [déréférencement des pointeurs bruts en toute sécurité](#)
fonction d'écriture , [lecture et écriture de fichiers](#)
méthode d'écriture , [Caractéristiques et génériques](#) , [Méthodes par défaut](#)
Caractéristique d' écriture , [entrée et sortie](#)
ajouter et insérer du texte , [Ajouter et insérer du texte](#)
méthode flush , [écrivains](#)
utiliser le langage de formatage dans votre code , [Utiliser le langage de formatage dans votre propre code](#)
méthode d'écriture , [écrivains](#)
méthode write_all , [écrivains](#)
écrivez! macro , [Opérateurs binaires](#) , [Ajout et insertion de texte](#) , [Valeurs de formatage](#) , [Rédacteurs](#)

Trait WriteBytesExt , [données binaires, compression et sérialisation](#)
écris ! macro , [Erreurs d'impression](#) , [Ajout et insertion de texte](#) , [Valeurs de formatage](#) , [Rédacteurs](#)

écrivains , [lecteurs et écrivains](#) , [écrivains](#) - [écrivains](#)

données binaires, compression, sérialisation , [Données binaires, compression et sérialisation](#) - [Données binaires, compression et sérialisation](#)

fichiers , [Fichiers](#)

autres types , [Autres types de lecteurs et](#) d'enregistreurs - [Autres types de lecteurs et d'enregistreurs](#)

Chercher le trait , [Chercher](#)

Fonction write_image , [Écriture de fichiers image](#)

write_unaligned , Entrer [et sortir de la mémoire](#)

write_volatile , Entrer [et sortir de la mémoire](#)

Oui

yield_now , Calculs de [longue durée : yield now et spawn blocking](#) -
Calculs de [longue durée : yield now et spawn blocking](#)

Z

principe de zéro frais généraux , [et pourtant la rouille est toujours rapide](#)

zéro-uplet , [Tuples](#)

Trait pouvant être mis à zéro , [Traits dangereux](#)

adaptateur zip , [zip](#)

Soutien Se déconnecter

Programmation de Rust

de Jim Blandy , Jason Orendorff et Leonora FS Tindall : 2-3 minutes

Programmation Rust, 2e édition

Programmation de Rust

Copyright © 2021 Jim Blandy, Leonora FS Tindall, Jason Orendorff. Tous les droits sont réservés.

Imprimé aux États-Unis d'Amérique.

Publié par O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Les livres O'Reilly peuvent être achetés à des fins éducatives, commerciales ou promotionnelles. Des éditions en ligne sont également disponibles pour la plupart des titres (<http://oreilly.com>). Pour plus d'informations, contactez notre département des ventes corporatives/institutionnelles : 800-998-9938 ou corporate@oreilly.com .

- Rédactrice des acquisitions : Suzanne McQuade
- Éditeur du développement : Jeff Bleiel
- Rédactrice en chef : Beth Kelly
- Réacteur en chef : Charles Roumeliotis
- Correcteur : Kim Wimpsett
- Indexeur : Potomac Indexing, LLC
- Architecte d'intérieur : David Futato
- Concepteur de la couverture : Karen Montgomery
- Illustrateur : Kate Dullea
- Juin 2021 : deuxième édition

Historique des révisions pour la deuxième édition

- 11/06/2021 : première version
- 2021-11-05 : deuxième version

Voir <http://oreilly.com/catalog/errata.csp?isbn=9781492052593> pour les détails de la version.

Le logo O'Reilly est une marque déposée d'O'Reilly Media, Inc. *Programming Rust* , l'image de couverture et l'habillage commercial associé sont des marques déposées d'O'Reilly Media, Inc.

Les opinions exprimées dans cet ouvrage sont celles des auteurs et ne représentent pas les opinions de l'éditeur. Bien que l'éditeur et les auteurs aient déployé des efforts de bonne foi pour s'assurer que les informations et les instructions contenues dans cet ouvrage sont exactes, l'éditeur et les auteurs déclinent toute responsabilité pour les erreurs ou omissions, y compris, sans s'y limiter, la responsabilité pour les dommages résultant de l'utilisation ou s'appuyer sur ce travail. L'utilisation des informations et des instructions contenues dans cet ouvrage se fait à vos risques et périls. Si des exemples de code ou d'autres technologies que ce travail contient ou décrit sont soumis à des licences open source ou aux droits de propriété intellectuelle d'autrui, il est de votre responsabilité de vous assurer que votre utilisation est conforme à ces licences et/ou droits.

Colophon

L'animal sur la couverture de *Programming Rust* est un crabe de Montagu (*Xantho hydrophilus*). Le crabe de Montagu a été trouvé dans le nord-est de l'océan Atlantique et dans la mer Méditerranée. Il vit sous les rochers et les rochers à marée basse. Si quelqu'un est exposé lorsqu'un rocher est soulevé, il tiendra agressivement ses pinces et les écartera largement pour se faire paraître plus grand.

Ce crabe d'apparence robuste a une apparence musclée avec une large carapace d'environ 70 mm de large. Le bord de la carapace est sillonné et la couleur est jaunâtre ou brun rougeâtre. Il a 10 pattes : la paire avant (les chélipèdes) est de taille égale avec des griffes ou des pinces à bout noir ; puis il y a trois paires de pattes de marche qui sont robustes et relativement courtes ; et la dernière paire de pattes sert à nager. Ils marchent et nagent de côté.

Ce crabe est omnivore. Ils mangent principalement des algues, des escargots et des crabes d'autres espèces. Ils sont surtout actifs la nuit. Les femelles portant des œufs se trouvent de mars à juillet et les larves sont présentes dans le plancton pendant la majeure partie de l'été.

De nombreux animaux sur les couvertures O'Reilly sont en voie de disparition; tous sont importants pour le monde.

L'illustration de la couverture est de Karen Montgomery, basée sur une image de *Wood's Natural History*. Les polices de couverture sont Gilroy Semibold et Guardian Sans. La police du texte est Adobe Minion Pro ; la police d'en-tête est Adobe Myriad Condensed ; et la police de code est Ubuntu Mono de Dalton Maag.

