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).

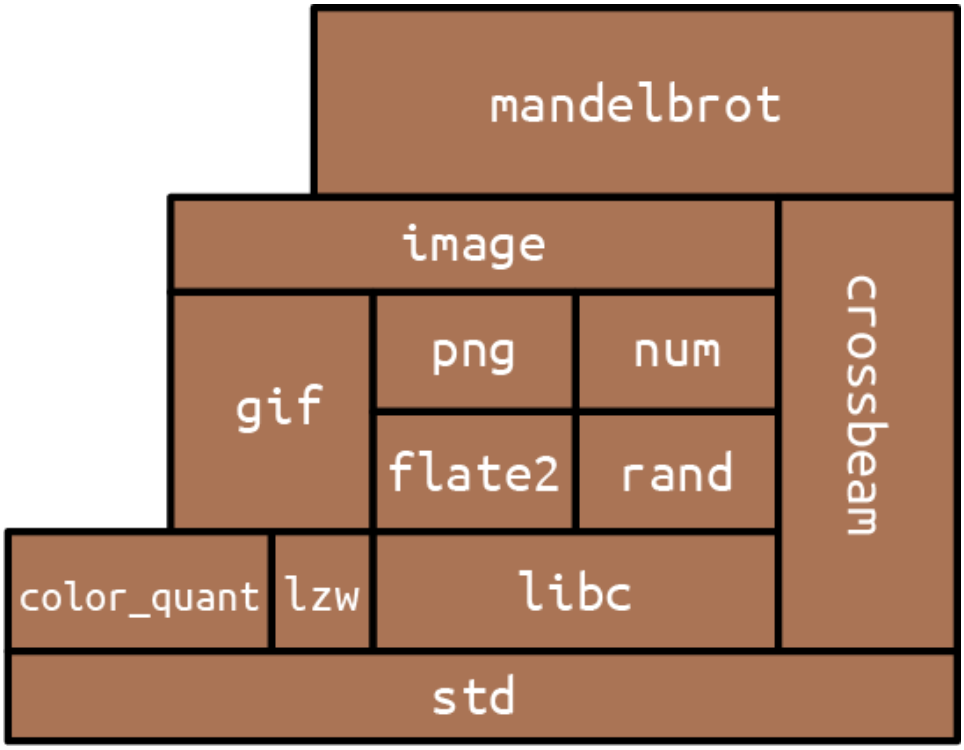

Illustration 8-1. Une caisse et ses dépendances

La façon la plus simple de voir ce que sont les caisses et comment elles fonctionnent ensemble est d'utiliser `cargo build` avecle `--verbose` drapeau pour construire un projet existant qui a des dépendances. Nous l'avons fait en utilisant "Un programme Mandelbrot simultané" 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=l
    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, télécharge *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 *transitif*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 génèrent 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 configurationsparamètres que vous pouvez mettre dans votre fichier *Cargo.toml* qui affectent les `rustc` lignes de commande `cargo` générées (Tableau 8-1).

Tableau 8-1. Sections de paramètres de configuration Cargo.toml

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 préludem*odules, qui réduisent le passe-partout en rassemblant les importations courantes dont presque tous les utilisateurs auront besoin.
- Nous présentons *des constantes*et *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 modulepeut 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' `::` opérateurest 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 normebibliothèque. 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` déclarationfait 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 roo
```

```
    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 modulecommence 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 desstructuretypes, 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. Statiquepeut 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 fougèrecommence à décoller, vous décidez que vous avez besoin de plus d'un programme. Supposons que vous disposiez 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@exemple.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/efern.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 articledans 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_revspec` should have a camel case name
    such as `GitRevspec`, #[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]` attributsur le genre :

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

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

`#[cfg(...)]` option	Activé lorsque
`test`	Les tests sont activés (compilation avec `cargo test` ou `rustc --test`).
`debug_assertions`	Les assertions de débogage sont activées (généralement dans les versions non optimisées).
`unix`	Compilation pour Unix, y compris macOS.
`windows`	Compilation pour Windows.
`target_pointer_width = "64"`	Cibler une plate-forme 64 bits. L'autre valeur possible est `"32"`.
`target_arch = "x86_64"`	Ciblant x86-64 en particulier. Autres valeurs : `"x86"`, `"arm"`, `"aarch64"`, `"powerpc"`, `"powerpc64"`, `"mips"`.
`target_os = "macos"`	Compilation pour macOS. Autres valeurs : `"windows"`, `"ios"`, `"android"`, `"linux"`, `"freebsd"`, `"openbsd"`, `"netbsd"`, `"dragonfly"`.
`feature = "robots"`	La fonctionnalité définie par l'utilisateur nommée `"robots"` est activée (compilation avec `cargo build --feature robots` ou `rustc --cfg feature='"robots"'`). Les fonctionnalités sont déclarées dans la `[features]` section *Cargo.toml*.
`not(`*UN*`)`	*A* n'est pas satisfait. Pour fournir deux implémentations différentes d'une fonction, marquez l'une avec `#[cfg(X)]` et l'autre avec `#[cfg(not(X))]`.
`all(`*Un*`,`*B*`)`	A et *B sont satisfaits* (l'équivalent de `&&`).

`#[cfg(...)]`	**Activé lorsque option**
`any(`*Un*`,`*B*`)`	A ou *B est satisfait* (l'équivalent de `||`).

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]` attributpour 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, etécrivez 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 bac 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` courttous les tests de votre projet :

```
$cargaisontest
   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. Sinon, 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 placepour écrire des assertions qui ne sont vérifiées que dans les versions de débogage.

Pour tester les cas d'erreur, ajoutez l' `#[should_panic]` attributà 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 a `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 fougèrecontinuer 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` créeDocuments HTMLpour 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* .

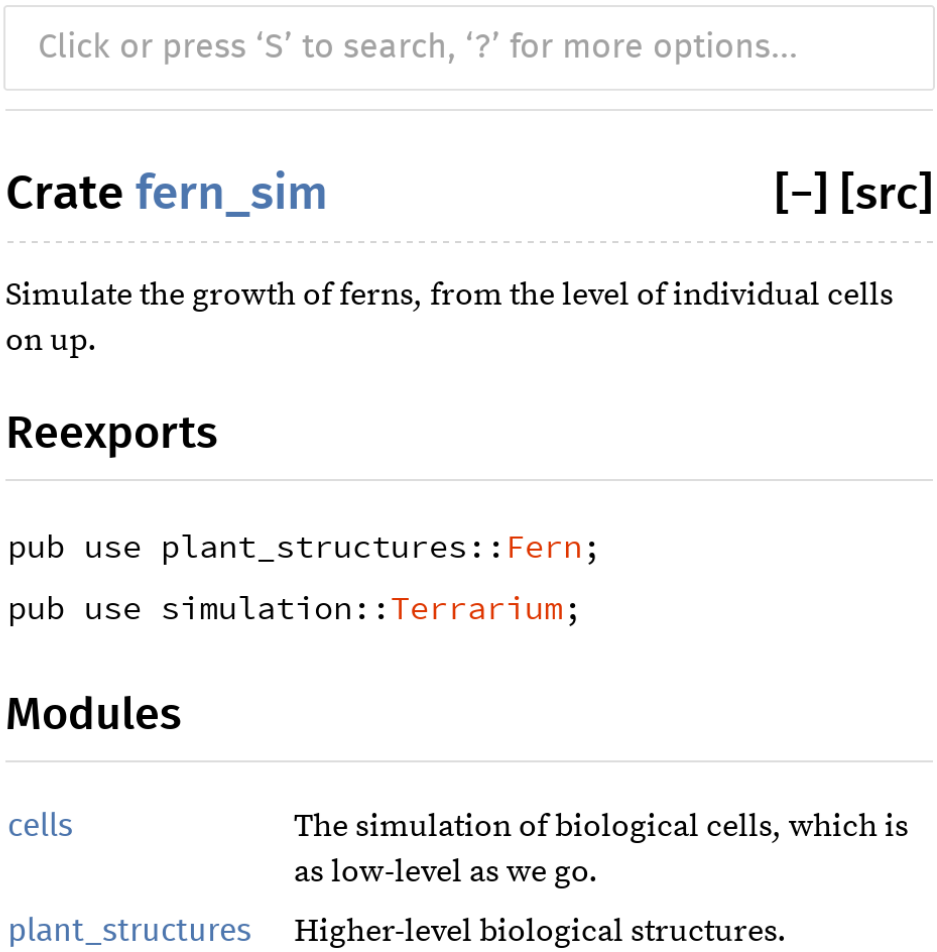

Illustration 8-2. Exemple de documentation générée par `rustdoc`

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
.

Function ranges::overlap [−] [src]

```
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 :

```
$cargaisontest
   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();
/// ```
pub fn upload_all(&mut self) {
    ...
}
```

Si le code n'est même pas censé être compilé, utilisez à la `ignore` place 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 s'ils 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 centaines 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 Cargooù 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 .

Tableau 8-3. Spécification des versions dans un fichier Cargo.toml

Ligne Cargo.toml	Sens
`image = "=0.10.0"`	Utilisez uniquement la version exacte 0.10.0
`image = ">=1.0.5"`	Utilisez 1.0.5 ou *toute* version supérieure (même 2.9, si elle est disponible)
`image = ">1.0.5 <1.1.9"`	Utilisez une version supérieure à 1.0.5, mais inférieure à 1.1.9
`image = "<=2.7.10"`	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 versionles 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 modifierez 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 caissepour toi.

```
$colis de fret
warning: manifest has no description, license, license-file, documentation,
homepage or repository. See http://doc.crates.io/manifest.html#package-metad
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
```

La `cargo package` commandecrée 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@exemple.com>"]
licence = "MIT"
page d'accueil = "https://fernsim.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsim.exemple.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 a `path` , qui est prioritaire pour vos propres builds locaux, et a `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 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

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 projetcontinue de croître, vous finissez par écrire de nombreuses caisses. Ils vivent côte à côte 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 CIpeut 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.