Chapitre 14. Fermetures

> *Sauver l'environnement! Créez une fermeture aujourd'hui !*
>
> —Cormac Flanagan

Trier un vecteurd'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 a `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 fermeturepeut 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-miettesjoue 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, répétonsl'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 référenceà `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 solutionaux 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 fonctionset 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 minuscules : 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 appelable, 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 Rustsont 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.

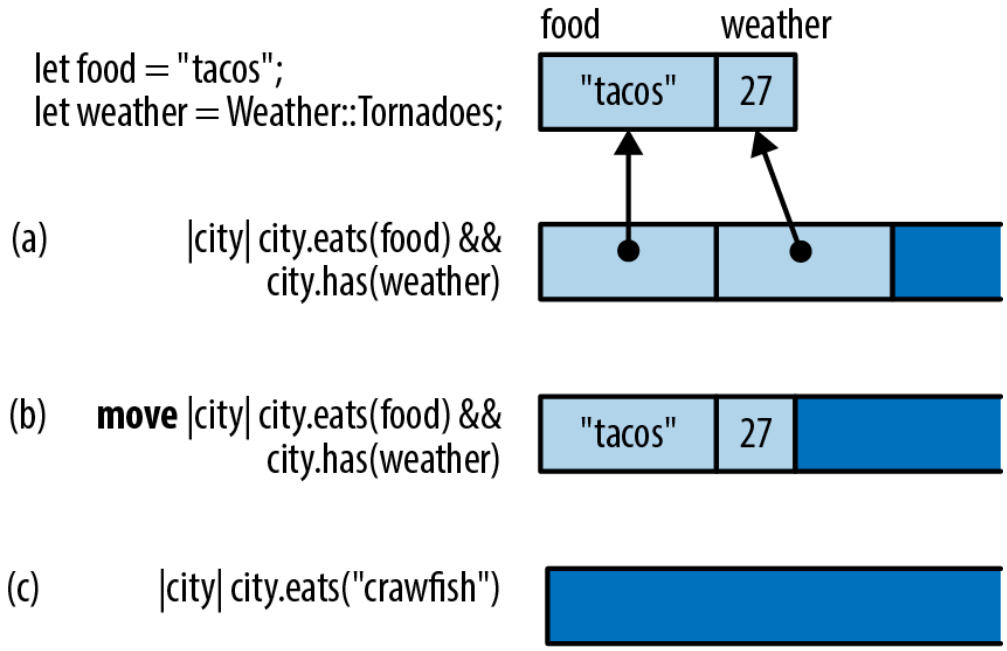

Illustration 14-1. Disposition des fermetures en mémoire

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 fermeturesqui 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 foisplus 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 famillede 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.

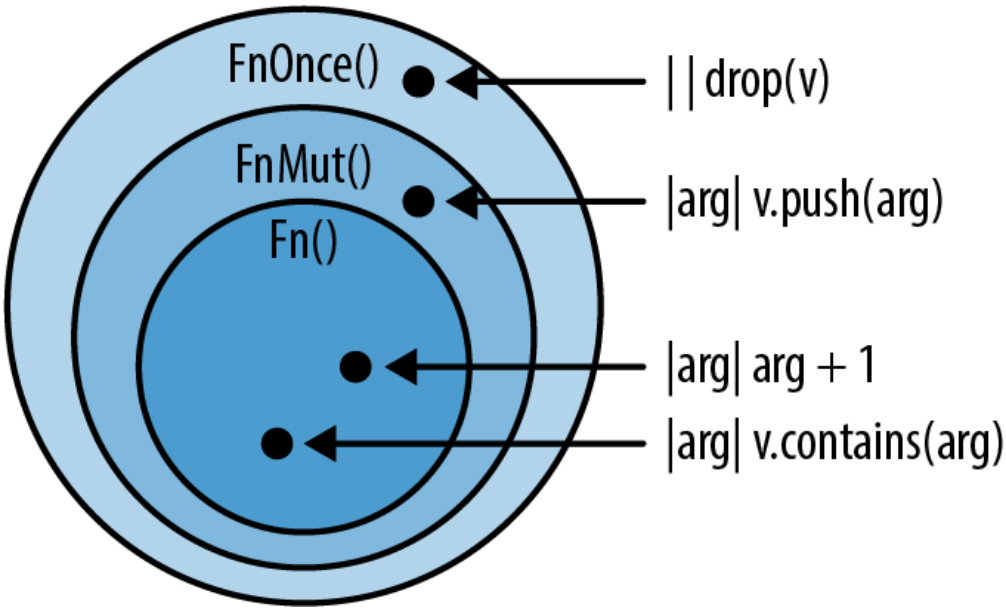

Illustration 14-2. Diagramme de Venn des trois catégories de fermeture

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 fermeturespeut 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` cadrepour 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-`fn` clé. 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 closu
   |
   = 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 économe 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 Rustsont 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-Contrôleurmodèle (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.

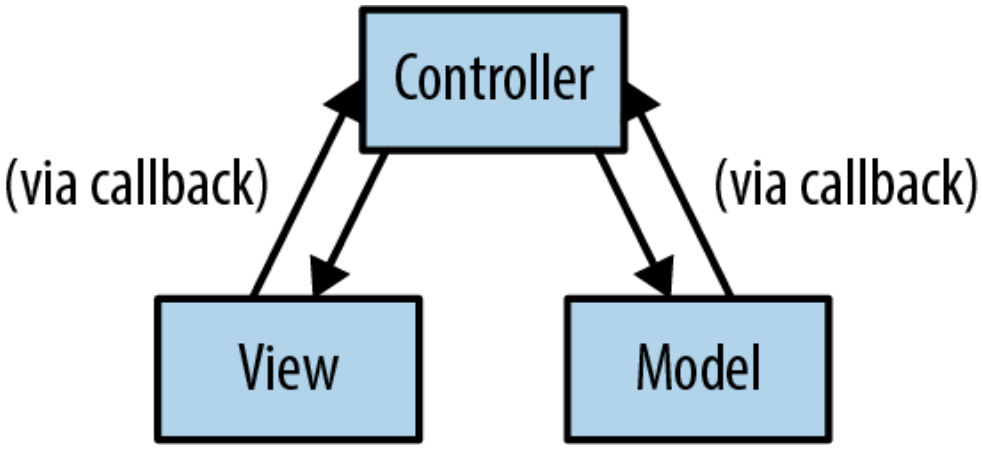

Illustration 14-3. Le modèle de conception Modèle-Vue-Contrôleur

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 .

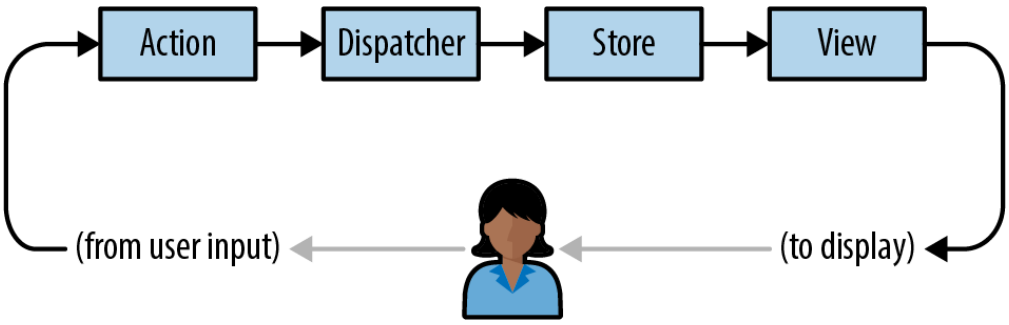

Illustration 14-4. L'architecture Flux, une alternative à MVC

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.