Chapitre 14. Fermetures

Sauvez l'environnement! Créez une clôture dès aujourd'hui!

—Cormac Flanagan

Le tri d'un vecteur d'entiers est facile :

```
integers.sort();
```

Il est donc triste de constater que lorsque nous voulons que certaines données soient triées, ce n'est presque jamais un vecteur d'entiers. Nous avons généralement des enregistrements d'une sorte ou d'une autre, et la méthode intégrée ne fonctionne généralement pas: `sort`

```
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 ne pas mettre en œuvre . Nous devons spécifier l'ordre de tri, comme ceci: `City std::cmp::Ord`

```
/// 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, , prend un enregistrement et extrait la *clé*, le champ par lequel nous voulons trier nos données. (Il renvoie un nombre négatif parce qu'il organise les nombres dans l'ordre croissant, et nous voulons un ordre décroissant: la ville la plus peuplée d'abord.) La méthode prend cette fonction clé comme paramètre. `city_population_descending City sort sort_by_key`

Cela fonctionne bien, mais il est plus concis d'écrire la fonction d'assistance comme 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 . Il prend un argument et renvoie . Rust déduit le type d'argument et le type de retour de la façon dont la fermeture est utilisée. `|city| -city.population` `city -city.population`

Voici d'autres exemples de fonctionnalités de bibliothèque standard qui acceptent les fermetures :

- `Iterator` méthodes telles que et , pour travailler avec des données séquentielles. Nous couvrirons ces méthodes au chapitre 15. `map` `filter`
- API de threading comme , qui démarre un nouveau thread système. La simultanéité consiste à déplacer le travail vers d'autres threads, et les fermetures représentent commodément les unités de travail. Nous aborderons ces fonctionnalités au chapitre 19. `thread::spawn`
- Certaines méthodes qui doivent conditionnellement calculer une valeur par défaut, comme la méthode des entrées. Cette méthode obtient ou crée une entrée dans un , et elle est utilisée lorsque la valeur par défaut est coûteuse à calculer. La valeur par défaut est transmise en tant que fermeture appelée uniquement si une nouvelle entrée doit être créée. `or_insert_with` `HashMap` `HashMap`

Bien sûr, les fonctions anonymes sont partout de nos jours, même dans des langages comme Java, C #, Python et C ++ qui ne les 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 les fermetures avec des méthodes de bibliothèque standard, comment une fermeture peut « capturer » des variables dans son étendue, comment écrire vos propres fonctions et méthodes qui prennent les fermetures comme arguments, et comment stocker les fermetures pour une utilisation ultérieure comme rappels. Nous expliquerons également comment les fermetures Rust sont mises en œuvre et pourquoi elles sont plus rapides que prévu.

Capture de variables

Une fermeture peut utiliser des données appartenant à une fonction englobante. Par exemple:

```
/// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

La fermeture utilise ici , qui appartient à la fonction d'enceinte, . On dit que la fermeture « capture ». C'est l'une des caractéristiques classiques des fermetures, donc naturellement, Rust le soutient; mais dans Rust, cette fonctionnalité est livrée avec une ficelle attachée. `stat sort_by_statistic stat`

Dans la plupart des langues avec fermetures, le ramassage des ordures joue un rôle important. Par exemple, considérez ce code JavaScript :

```
// Start an animation that rearranges the rows in a table of cities.
function startSortingAnimation(cities, stat) {
    // Helper function that we'll use to sort the table.
    // Note that this function refers to stat.
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // Now kick off an animation, passing keyfn to it.
    // The sorting algorithm will call keyfn later.
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

La fermeture est stockée dans le nouvel objet. Il est destiné à être appelé après les retours. Maintenant, normalement, lorsqu'une fonction est renvoyée, toutes ses variables et tous ses arguments sortent de la portée et sont ignorés. Mais ici, le moteur JavaScript doit rester d'une manière ou d'une autre, puisque la fermeture l'utilise. La plupart des moteurs JavaScript le font en allouant dans le tas et en laissant le garbage collector le récupérer plus tard. `keyfn SortingAnimation startSortingAnimation stat stat`

Rust n'a pas de ramassage des ordures. Comment cela fonctionnera-t-il? Pour répondre à cette question, nous allons examiner deux exemples.

Fermetures qui empruntent

Tout d'abord, répétons l'exemple d'ouverture de cette section:

```
/// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

Dans ce cas, lorsque Rust crée la fermeture, il emprunte automatiquement une référence à . Cela va de soi : la fermeture fait référence à , elle doit donc y avoir une référence. `stat stat`

Le reste est simple. La clôture est assujettie aux règles sur les emprunts et les durées de vie que nous avons décrites au chapitre 5. En particulier, puisque la fermeture contient une référence à , Rust ne la laissera pas survivre . Étant donné que la fermeture n'est utilisée que pendant le tri, cet exemple convient parfaitement. `stat stat`

En bref, Rust assure la sécurité en utilisant des durées de vie au lieu de la collecte des ordures. Le chemin de Rust est plus rapide: même une allocation GC rapide sera plus lente que le stockage sur la pile, comme rust le fait dans ce cas. `stat`

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 : prendre une fermeture et l'appeler dans un nouveau thread système. Notez qu'il s'agit de la liste d'arguments vide de la fermeture. `thread::spawn ||`

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 valeur. Nous couvrirons cela au chapitre 19.) `JoinHandle`

Encore une fois, la fermeture contient une référence à . Mais cette fois, Rust ne peut pas garantir que la référence est utilisée en toute sécurité. Rust rejette donc ce programme : `key_fn stat`

```
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 borro
   |              may outlive borrowed value `stat`
```

En fait, il y a deux problèmes ici, parce qu'il est également partagé de manière dangereuse. Tout simplement, on ne peut pas s'attendre à ce que le nouveau thread créé par finisse son travail avant et soit détruit à la fin de la fonction. `cities thread::spawn cities stat`

La solution aux deux problèmes est la même : dire à Rust de *déménager* et d'entrer dans les fermetures qui les utilisent au lieu d'emprunter des références à celles-ci. `cities stat`

```
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-clé avant chacune des deux fermetures. Le mot-clé indique à Rust qu'une fermeture n'emprunte pas les variables qu'elle utilise : elle les vole. `move move`

La première fermeture, , prend possession de . Ensuite, la deuxième fermeture prend possession des deux et . `key_fn stat cities key_fn`

Rust offre donc deux façons pour les fermetures d'obtenir des données à partir des scopes d'enclos: les déménagements et les emprunts. En réalité,

il n'y a rien de plus à dire que cela; les fermetures suivent les mêmes règles concernant les déménagements et les emprunts que celles que nous avons déjà abordées aux chapitres 4 et 5. Quelques exemples :

- Comme partout ailleurs dans la langue, si une fermeture est une valeur de type copiable, comme , elle copie la valeur à la place. Donc, s'il s'agissait d'un type copiable, nous pourrions continuer à l'utiliser même après avoir créé une fermeture qui l'utilise. `move i32 Statistic stat move`
- Les valeurs de types non pouvant être copiés, comme , sont vraiment déplacées : le code précédent est transféré vers le nouveau thread, par le biais de la fermeture. Rust ne nous permettrait pas d'accéder par nom après la création de la fermeture. `Vec<City> cities move cities`
- En l'occurrence, ce code n'a pas besoin d'être utilisé après le point où la fermeture le déplace. Si nous le faisions, cependant, la solution de contournement serait facile: nous pourrions dire à Rust de cloner et de stocker la copie dans une variable différente. La fermeture ne volerait qu'une seule des copies, quelle que soit celle à laquelle elle se réfère. `cities cities`

Nous obtenons quelque chose d'important en acceptant les règles strictes de Rust : la sécurité des fils. 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 des fonctions et des 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) et renvoie un . Il a le type . `&City i64 fn(&City) -> i64`

Vous pouvez faire toutes les mêmes choses avec des fonctions qu'avec d'autres valeurs. Vous pouvez les stocker dans des variables. Vous pouvez

utiliser toute la syntaxe Rust habituelle pour calculer les valeurs de fonction :

```
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 typés de fonction. Les types génériques comme peuvent stocker des scads de fonctions, tant qu'ils partagent tous le même type. Et les valeurs de fonction sont minuscules : une valeur est l'adresse mémoire du code machine de la fonction, tout comme un pointeur de fonction en C++. `Vec fn fn`

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 , pas le corps. La nouvelle version est générique. Il faut un de n'importe quel type aussi longtemps que met en œuvre le trait spécial . Ce trait est automatiquement implémenté par toutes les fonctions et la plupart des fermetures qui prennent un seul comme argument et renvoient une valeur booléenne
: `count_selected_cities test_fn F F Fn(&City) -> bool &City`

```
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 accepte soit une fonction, soit une fermeture
: `count_selected_cities`

```
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 . La fermeture a son propre type qui n'est pas un type. `fn |city| city.monster_attack_risk > limit fn`

En fait, chaque fermeture que vous écrivez a son propre type, car une fermeture peut contenir des données : des valeurs empruntées ou volées dans des étendues 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 qui ont exactement le même type. Mais chaque fermeture met en œuvre un trait; la fermeture dans notre exemple implémente . `Fn Fn(&City) -> i64`

Étant donné que chaque fermeture a son propre type, le code qui fonctionne avec les fermetures doit généralement être générique, comme . C'est un peu maladroit d'épeler les types génériques à chaque fois, mais pour voir les avantages de cette conception, lisez la suite. `count_selected_cities`

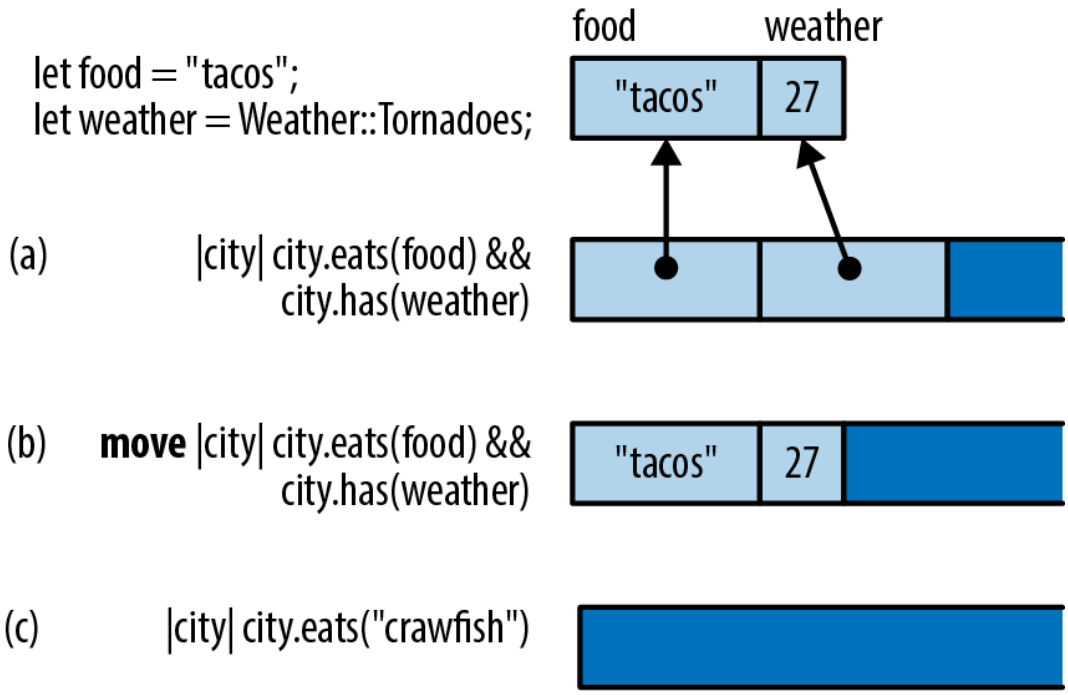
Performance de fermeture

Les fermetures de Rust sont conçues pour être rapides : plus rapides que les pointeurs de fonction, suffisamment rapides pour que vous puissiez les utiliser même dans un code chaud 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 langues, les fermetures sont allouées dans le tas, distribuées dynamiquement et les ordures collectées. 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 les contours*, une technique clé utilisée par les compilateurs pour éliminer la surcharge d'appel de fonction et permettre une série d'autres optimisations. Tout compte fait, les fermetures sont suffisamment lentes dans ces langues pour qu'il puisse être utile de les retirer manuellement des boucles internes serrées.

Les fermetures antirouille n'ont aucun de ces inconvénients de performance. Ce ne sont pas des ordures ramassées. Comme tout le reste dans Rust, ils ne sont pas alloués sur le tas à moins que vous ne les mettiez dans un , , ou un autre conteneur. Et puisque chaque fermeture a un type distinct, chaque fois que le compilateur Rust connaît le type de fermeture que vous appelez, il peut insérer le code pour cette fermeture particulière. Cela permet d'utiliser des fermetures en boucles serrées, et les programmes Rust le font souvent, avec enthousiasme, comme vous le verrez au chapitre 15. `Box` `Vec`

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 se réfèreront : une chaîne et un enum simple, dont la valeur numérique se trouve être 27. `food` `weather`



Graphique 14-1. Disposition des fermetures en mémoire

La fermeture (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 qu'il s'agit d'une fermeture, elle contient donc des valeurs au lieu de références. `move`

La fermeture (c) n'utilise aucune variable de son environnement. La structure est vide, donc cette fermeture ne prend aucune mémoire du

tout.

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 insérer tous les appels à une fermeture, puis même les petites structures illustrées dans cette figure sont optimisées.

Dans « Rappels », nous allons montrer comment allouer des fermetures dans le tas et les appeler dynamiquement, en utilisant des objets de trait. C'est un peu plus lent, mais c'est toujours aussi rapide que n'importe quelle autre méthode d'objet de trait.

Fermetures et sécurité

Tout au long du chapitre jusqu'à présent, nous avons parlé de la façon dont 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.

Fermetures qui tuent

Nous avons vu des fermetures qui empruntent des valeurs et des fermetures qui les volent; ce n'était qu'une question de temps avant qu'ils aillent jusqu'au bout.

Bien sûr, *tuer n'est* pas vraiment la bonne terminologie. Dans Rust, nous *supprimons des* 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 est appelé, est abandonné. `f my_str`

Alors, que se passe-t-il si nous l'appelons deux fois?

```
f();
f();
```

Réfléchissons-y. La première fois que nous appelons , il tombe , ce qui signifie que la mémoire où la chaîne est stockée est libérée, renvoyée au sys-

tème. La deuxième fois que nous appelons, la même chose se produit. C'est un *double gratuit*, une erreur classique dans la programmation C ++ qui déclenche un comportement indéfini. `f my_str f`

Laisser tomber un deux fois serait une idée tout aussi mauvaise dans Rust. Heureusement, Rust ne peut pas être trompé si facilement: `String`

```
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 une chose assez extraordinaire, mais nous avons parlé tout au long de ce livre de la propriété et des vies. L'idée que les valeurs sont épuisées (c'est-à-dire déplacées) est l'un des concepts fondamentaux de Rust. Cela fonctionne de la même manière avec les fermetures qu'avec tout le reste.

FnOnce

Essayons une fois de plus de tromper Rust pour qu'il en laisse tomber deux fois. Cette fois, nous allons utiliser cette fonction générique : `String`

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

Cette fonction générique peut être passée toute fermeture qui implémente le trait : c'est-à-dire les fermetures qui ne prennent aucun argument et renvoient . (Comme pour les fonctions, le type de retour peut être omis s'il est ; est un raccourci pour .) `Fn() () () Fn() Fn() -> ()`

Maintenant, que se passe-t-il si nous passons notre fermeture dangereuse à cette fonction générique?

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

Encore une fois, la fermeture tombera quand elle sera appelée. L'appeler deux fois serait un double gratuit. Mais encore une fois, Rust n'est pas dupe : `my_str`

```
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
    |         |       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 complètement bannis de la langue, mais les fermetures de nettoyage sont parfois utiles. Donc, au lieu de cela, Rust restreint leur utilisation. Les fermetures qui suppriment des valeurs, comme , ne sont pas autorisées à avoir . Ils sont, littéralement, pas du tout. Ils mettent en œuvre un trait moins puissant, le trait des fermetures que l'on peut appeler une fois. `f Fn Fn FnOnce`

La première fois que vous appelez une fermeture, *la fermeture elle-même est épuisée.* C'est comme si les deux traits, et , étaient définis comme ceci: `FnOnce Fn FnOnce`

```
// Pseudocode for `Fn` and `FnOnce` traits with no arguments.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Tout comme une expression arithmétique comme est un raccourci pour un appel de méthode, , Rust traite comme un raccourci pour l'une des deux méthodes de trait montrées dans l'exemple précédent. Pour une fermeture, s'étend à . Cette méthode prend par référence, de sorte que la fermeture n'est pas déplacée. Mais si la fermeture n'est sûre à appeler qu'une seule fois, elle s'étend à . Cette méthode prend par valeur, de sorte que la fermeture est épuisée. `a + b Add::add(a, b) closure() Fn closure() closure.call() self closure() closure.call_once() self`

Bien sûr, nous avons délibérément semé le trouble ici en utilisant . En pratique, vous vous retrouverez principalement dans cette situation par

accident. Cela n'arrive pas souvent, mais de temps en temps, vous écrirez du code de fermeture qui utilise involontairement une valeur: drop()

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict {  // oops!
        println!("{:?} - {:?}", key, value);
    }
};
```

Ensuite, lorsque vous appelez plus d'une fois, vous obtiendrez un message d'erreur comme celui-ci : debug_dump_dict()

```
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 vari
`dict` out of its environment
   |
13 |         for (key, value) in dict {
   |                             ^^^^
```

Pour déboguer cela, nous devons comprendre pourquoi la fermeture est un fichier . Quelle valeur est utilisée ici? Le compilateur souligne utilement que c'est , ce qui dans ce cas est le seul auquel nous faisons référence. Ah, il y a le bug : on l'épuise en itérant dessus directement. Nous devrions tourner en boucle sur , plutôt que simplement – pour accéder aux valeurs par référence : FnOnce dict dict &dict dict

```
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 et peut être appelée n'importe quel nombre de fois. Fn

FnMut

Il existe un autre type de fermeture, celui qui contient des données ou des références modifiables. `mut`

Rust considère que les valeurs non-sûres peuvent être partagées entre les threads. Mais il ne serait pas sûr de partager des non-fermetures qui contiennent des données : appeler une telle fermeture à partir de plusieurs threads pourrait conduire à toutes sortes de conditions de course car plusieurs threads essaient de lire et d'écrire les mêmes données en même temps. `mut mut mut`

Par conséquent, Rust a une autre catégorie de fermeture, , la catégorie de fermetures qui écrivent. les fermetures sont appelées par référence, comme si elles étaient définies comme suit : `FnMut FnMut mut`

```
// Pseudocode for `Fn`, `FnMut`, and `FnOnce` traits.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Toute fermeture qui nécessite l'accès à une valeur, mais ne supprime aucune valeur, est une fermeture. Par exemple: `mut FnMut`

```
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, cela nécessite un . Puisque est un et non un , ce code ne parvient pas à compiler. 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. `call_twice Fn incr FnMut Fn`

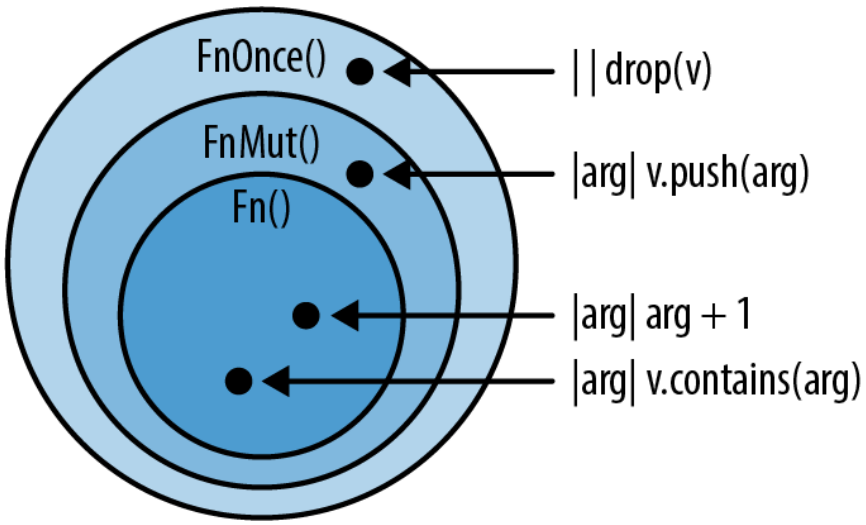
- `Fn` est la famille de fermetures et de fonctions que vous pouvez appeler plusieurs fois sans restriction. Cette catégorie la plus élevée com-

prend également toutes les fonctions. `fn`

- `FnMut` est la famille des fermetures qui peuvent être appelées plusieurs fois si la fermeture elle-même est déclarée . `mut`
- `FnOnce` est la famille de fermetures qui peuvent être appelées une fois, si l'appelant est propriétaire de la fermeture.

Chaque répond aux exigences de , et chaque répond aux exigences de . Comme le montre la figure 14-2, il ne s'agit pas de trois catégories distinctes. `Fn FnMut FnMut FnOnce`

Au lieu de cela, est un sous-trait de , qui est un sous-trait de . Cela fait la catégorie la plus exclusive et la plus puissante. et sont des catégories plus larges qui incluent les fermetures avec des restrictions d'utilisation. `Fn() FnMut() FnOnce() Fn FnMut FnOnce`



Graphique 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 la plus large bande possible de fermetures, notre fonction devrait vraiment accepter toutes les fermetures, comme ceci: `call_twice FnMut`

```
fn call_twice<F>(mut closure: F) where F: FnMut() {
    closure();
    closure();
}
```

La limite sur la première ligne était , et maintenant c'est . Avec ce changement, nous acceptons toujours toutes les fermetures, et nous pouvons également utiliser sur les fermetures qui mutent les données: `F: Fn() F: FnMut() Fn call_twice`

```
let mut i = 0;
call_twice(|| i += 1);  // ok!
```

```
assert_eq!(i, 2);
```

Copier et cloner pour les fermetures

Tout comme Rust détermine automatiquement quelles fermetures ne peuvent être appelées qu'une seule fois, il peut déterminer quelles fermetures peuvent être mises en œuvre et , et lesquelles ne le peuvent pas. `Copy` `Clone`

Comme nous l'avons expliqué précédemment, les fermetures sont représentées sous la forme de structures qui contiennent soit les valeurs (pour les fermetures), soit des références aux valeurs (pour les non-fermetures) des variables qu'elles capturent. Les règles pour et sur les fermetures sont tout comme les règles pour les structures régulières. Une non-fermeture qui ne mute pas les variables ne contient que des références partagées, qui sont à la fois et , de sorte que la fermeture est à la fois et aussi bien
: `move` `move` `Copy` `Clone` `Copy` `Clone` `move` `Clone` `Copy` `Clone` `Copy`

```
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'autre part, une non-fermeture qui *mute des* valeurs a des références mutables dans sa représentation interne. Les références mutables ne sont ni ni , donc pas plus qu'une fermeture qui les utilise : `move` `Clone` `Copy`

```
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 copi
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // error: use of moved val
```

Pour une fermeture, les règles sont encore plus simples. Si tout ce qu'une fermeture capture est , c'est . Si tout ce qu'il capture est , c'est . Par exemple: `move` `move` `Copy` `Copy` `Clone` `Clone`

```
let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{}", greeting);
};
```

```
greet.clone()("Alfred");
greet.clone()("Bruce");
```

Cette syntaxe est un peu bizarre, mais cela signifie simplement que nous clonons la fermeture et appelons ensuite le clone. Les résultats de ce programme sont les suivants : `.clone()(...)`

```
Hello, Alfred
Hello, Bruce
```

Lorsqu'il est utilisé dans , il est déplacé dans la structure qui représente en interne, car il s'agit d'une fermeture. Ainsi, lorsque nous clonons, tout ce qui s'y trouve est également cloné. Il existe deux copies de , qui sont chacune modifiées séparément lorsque les clones de 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. `greeting greet greet move greet greeting greet`

Rappels

Beaucoup de bibliothèques utilisent des *rappels* dans le cadre de leur API : des fonctions fournies par l'utilisateur, pour que la bibliothèque appelle plus tard. En fait, vous avez déjà vu des API comme celle-ci dans ce livre. Au chapitre 2, nous avons utilisé le framework pour écrire un serveur Web simple. Une partie importante de ce programme était le routeur, qui ressemblait à ceci: `actix-web`

```
App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

Le but du routeur est d'acheminer les demandes entrantes d'Internet vers le bit de code Rust qui gère ce type particulier de demande. Dans cet exemple, et étaient les noms des fonctions que nous avons déclarées ailleurs dans le programme, en utilisant le mot-clé. Mais nous aurions pu adopter des fermetures à la place, comme ceci: `get_index post_gcd fn`

```
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 qu'il a été écrit pour accepter n'importe quel thread-safe comme argument. `actix-web` `Fn`

Comment pouvons-nous le faire dans nos propres programmes? Essayons d'écrire notre propre routeur très simple à partir de zéro, sans utiliser de code de . Nous pouvons commencer par déclarer quelques types pour représenter les requêtes et les réponses HTTP : `actix-web`

```
struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

Maintenant, 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 à créer que des itinéraires 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 compile et s'exécute. Malheureusement, si nous ajoutons un autre itinéraire:

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

puis nous obtenons des erreurs:

```
error: mismatched types
   |
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                              expected closure, found a different cl
   |
   = 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 type: `BasicRouter`

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

Nous avons involontairement déclaré que chacun a un seul type de rappel , et tous les rappels dans le sont de ce type. De retour dans « Lequel utiliser », nous avons montré un type qui avait le même problème: `BasicRouter C HashMap Salad`

```
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, de sorte qu'une seule peut contenir toutes sortes de rappels. Notez que le paramètre type a disparu. `HashMap C`

Cela nécessite quelques ajustements aux 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));
    }
}
```

NOTE

Notez les deux limites dans la signature de type pour : un trait particulier et la durée de vie. La rouille nous fait ajouter cette liaison. Sans cela, l'appel à 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. `C add_route Fn 'static 'static Box::new(callback)`

Enfin, notre routeur simple est prêt à gérer les demandes 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 traits, utilise des *pointeurs de fonction ou des* types. Ces types, tels que , agissent beaucoup comme des fermetures: `fn fn(u32) -> u32`

```
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 type approprié, soit dans une liaison, soit dans une signature de fonction, le compilateur vous permet de les utiliser de cette façon : `fn`

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

Contrairement à la capture des fermetures, ces pointeurs de fonction ne prennent qu'un seul . `usize`

Une table de routage contenant des pointeurs de fonction ressemblerait à ceci :

```
struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) -> Response>
}
```

Ici, le ne stocke qu'un seul par , et critiquement, il n'y a pas de . Mis à part le lui-même, il n'y a pas d'allocation dynamique du tout. Bien sûr, les méthodes doivent également être
ajustées: `HashMap usize String Box HashMap`

```
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) -> Respons
        {
            self.routes.insert(url.to_string(), callback);
        }
    }
```

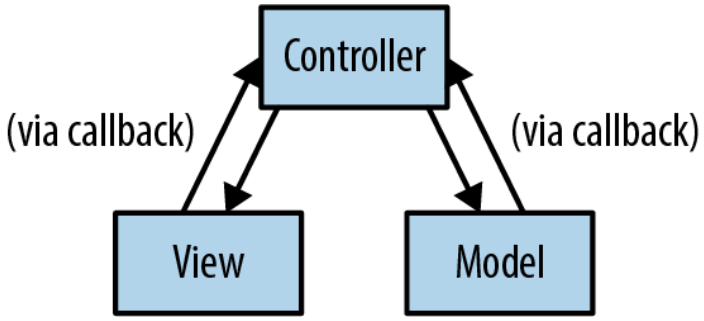
Comme le montre la figure 14-1, les fermetures ont des types uniques car chacune capture des variables différentes, de sorte qu'entre autres choses, elles ont chacune une taille différente. S'ils ne capturent rien, cependant, il n'y a rien à stocker. En utilisant des pointeurs dans les fonctions qui prennent des rappels, vous pouvez limiter un appelant à utiliser uniquement ces fermetures non capturantes, ce qui permet de gagner en performance et en flexibilité dans le code à l'aide de rappels au détriment de la flexibilité pour les utilisateurs de votre API. `fn`

Utilisation efficace des fermetures

Comme nous l'avons vu, les fermetures de Rust sont différentes des fermetures dans la plupart des autres langues. La plus grande différence est que dans les langues 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 le modèle de conception Model-View-Controller (MVC en abrégé), illustré à la figure 14-3. Pour chaque élément d'une interface utilisateur, une infrastructure 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 utilisateur. D'innombrables variantes de MVC ont été implémentées au fil des ans, mais l'idée générale est que trois objets répartissent les responsabilités de l'interface utilisateur d'une manière ou d'une autre.

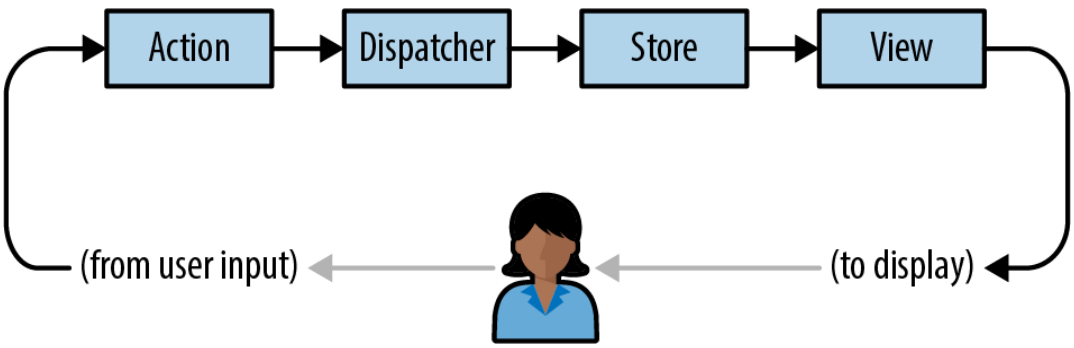
Voici le problème. En règle générale, chaque objet a une référence à l'un des autres ou aux deux, directement ou par le biais d'un rappel, comme illustré à la figure 14-3. Chaque fois que quelque chose arrive à l'un des objets, il avertit les autres, de sorte que tout se met à jour rapidement. La question de savoir quel objet « possède » les autres ne se pose jamais.



Graphique 14-3. Modèle de conception Model-View-Controller

Vous ne pouvez pas implémenter ce modèle dans Rust sans apporter quelques modifications. La propriété doit être 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 bons modèles alternatifs. Parfois, vous pouvez résoudre un problème de propriété et de durée de vie de la fermeture 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 de références. Ou vous pouvez implémenter l'une des nombreuses variantes sur MVC où les objets n'ont pas tous de références les uns aux autres. Ou modélisez votre boîte à outils d'après un système non-MVC avec un flux de données unidirectionnel, comme l'architecture Flux de Facebook, illustrée à la figure 14-4.



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

En bref, si vous essayez d'utiliser des 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 autour des alternatives de toute façon, 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 vitesse et de l'efficacité des fermetures Rust et qui est amusant à écrire, facile à lire et éminemment pratique. Suivant : les itérateurs de rouille.