Chapitre 10. Enums et motifs

> *Surprenant de voir à quel point les choses informatiques ont du sens vu comme une privation tragique de types de somme (cf. privation de lambdas).*
>
> —Graydon Hoare

Le premier sujet de ce chapitre est puissant, aussi vieux que les collines, heureux de vous aider à faire beaucoup en peu de temps (pour un prix), et connu sous de nombreux noms dans de nombreuses cultures. Mais ce n'est pas le diable. C'est une sorte de type de données défini par l'utilisateur, connu depuis longtemps par les pirates ML et Haskell comme des types de somme, des unions discriminées ou des types de données algébriques. Dans Rust, ils sont *appelés énumérations*, ou simplement *enums*. Contrairement au diable, ils sont tout à fait en sécurité, et le prix qu'ils demandent n'est pas une grande privation.

C++ et C# ont des enums ; vous pouvez les utiliser pour définir votre propre type dont les valeurs sont un ensemble de constantes nommées. Par exemple, vous pouvez définir un type nommé avec des valeurs , , , etc. Ce genre d'enum fonctionne aussi dans Rust. Mais Rust pousse les enums beaucoup plus loin. Un enum Rust peut également contenir des données, même des données de différents types. Par exemple, le type de Rust est un enum; une telle valeur est soit une valeur contenant un, soit une valeur contenant un . C'est au-delà de ce que les enums C++ et C# peuvent faire. C'est plus comme un C, mais contrairement aux unions, les enums Rust sont sans danger pour le
type. `Color Red Orange Yellow Result<String, io::Error> Ok String Err io::Error union`

Les enums sont utiles chaque fois qu'une valeur peut être une chose ou une autre. Le « prix » de leur utilisation est que vous devez accéder aux données en toute sécurité, en utilisant la correspondance de modèles, notre sujet pour la deuxième moitié de ce chapitre.

Les modèles peuvent également être familiers si vous avez utilisé le déballage en Python ou la déstructuration en JavaScript, mais Rust pousse les modèles plus loin. Les motifs de rouille sont un peu comme des expressions régulières pour toutes vos données. Ils sont utilisés pour tester si une valeur a ou non une forme souhaitée particulière. Ils peuvent extraire plusieurs champs d'une structure ou d'un tuple en variables locales

en une seule fois. Et comme les expressions régulières, elles sont concises, faisant généralement tout cela en une seule ligne de code.

Ce chapitre commence par les bases des enums, montrant comment les données peuvent être associées à des variantes enum et comment les enums sont stockés en mémoire. Ensuite, nous montrerons comment les modèles et les instructions de Rust peuvent spécifier de manière concise la logique basée sur des enums, des structs, des tableaux et des tranches. Les modèles peuvent également inclure des références, des mouvements et des conditions, ce qui les rend encore plus performants. `match if`

Enums

Les enums simples de style C sont simples:

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

Cela déclare un type avec trois valeurs possibles, *appelées variantes* ou *constructeurs* : , et . Cet enum particulier fait partie de la bibliothèque standard, de sorte que le code Rust peut l'importer, soit par lui-même: `Ordering Ordering::Less Ordering::Equal Ordering::Greater`

```
use std::cmp::Ordering;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Ordering::Less
    } else if n > m {
        Ordering::Greater
    } else {
        Ordering::Equal
    }
}
```

ou avec tous ses constructeurs :

```
use std::cmp::Ordering::{self, *};    // `*` to import all children

fn compare(n: i32, m: i32) -> Ordering {
```

```
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

Après avoir importé les constructeurs, nous pouvons écrire à la place de , et ainsi de suite, mais comme c'est moins explicite, il est généralement considéré comme un meilleur style de *ne pas* les importer, sauf lorsque cela rend votre code beaucoup plus lisible. `Less Ordering::Less`

Pour importer les constructeurs d'un énumération déclaré dans le module actif, utilisez une importation : `self`

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;
```

En mémoire, les valeurs des enums de style C sont stockées sous forme d'entiers. Parfois, il est utile de dire à Rust quels entiers utiliser:

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

Sinon, Rust vous attribuera les numéros, à partir de 0.

Par défaut, Rust stocke les énumérations de style C en utilisant le plus petit type entier intégré qui peut les accueillir. La plupart tiennent dans un seul octet:

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2);  // 404 doesn't fit in a u8
```

Vous pouvez remplacer le choix de représentation en mémoire de Rust en ajoutant un attribut à l'énumération. Pour plus d'informations, consultez « Recherche de représentations de données communes ». `#[repr]`

La conversion d'un enum de style C en un entier est autorisée :

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

Cependant, la coulée dans l'autre sens, de l'entier à l'enum, ne l'est pas. Contrairement à C et C++, Rust garantit qu'une valeur enum n'est jamais qu'une des valeurs énoncées dans la déclaration. Une conversion non cochée d'un type entier vers un type enum pourrait briser cette garantie, elle n'est donc pas autorisée. Vous pouvez soit écrire votre propre conversion cochée : `enum`

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None,
    }
}
```

ou utilisez la `caisse enum_primitive` . Il contient une macro qui génère automatiquement ce type de code de conversion pour vous.

Comme pour les structs, le compilateur implémentera des fonctionnalités telles que l'opérateur pour vous, mais vous devez demander: `==`

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years,
}
```

Les enums peuvent avoir des méthodes, tout comme les structs :

```
impl TimeUnit {
    /// Return the plural noun for this time unit.
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
```

```
                TimeUnit::Days => "days",
                TimeUnit::Months => "months",
                TimeUnit::Years => "years",
            }
        }

        /// Return the singular noun for this time unit.
        fn singular(self) -> &'static str {
            self.plural().trim_end_matches('s')
        }
    }
```

Voilà pour les enums de style C. Le type le plus intéressant de Rust enum est celui dont les variantes contiennent des données. Nous montrerons comment ceux-ci sont stockés en mémoire, comment les rendre génériques en ajoutant des paramètres de type et comment créer des structures de données complexes à partir d'enums.

Enums avec données

Certains programmes doivent toujours afficher les dates et les heures complètes jusqu'à la milliseconde, mais pour la plupart des applications, il est plus convivial d'utiliser une approximation approximative, comme « il y a deux mois ». Nous pouvons écrire un enum pour aider à cela, en utilisant l'enum défini précédemment:

```
/// A timestamp that has been deliberately rounded off, so our program
/// says "6 months ago" instead of "February 9, 2016, at 9:49 AM".
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

Deux des variantes de cet enum, et , prennent des arguments. Celles-ci sont *appelées variantes de tuple*. Comme les tuple structs, ces constructeurs sont des fonctions qui créent de nouvelles valeurs
: `InThePast InTheFuture RoughTime`

```
let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4 * 20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```

Les enums peuvent également avoir des *variantes struct*, qui contiennent des champs nommés, tout comme les structs ordinaires :

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2: Point3d },
}

let unit_sphere = Shape::Sphere {
    center: ORIGIN,
    radius: 1.0,
};
```

En tout, Rust a trois types de variante enum, faisant écho aux trois types de struct que nous avons montrés dans le chapitre précédent. Les variantes sans données correspondent à des structures de type unitaire. Les variantes de tuple ressemblent et fonctionnent comme des structures de tuple. Les variantes Struct ont des accolades et des champs nommés. Un seul enum peut avoir des variantes des trois types:

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem,
    },
}
```

Tous les constructeurs et champs d'un enum partagent la même visibilité que l'enum lui-même.

Enums en mémoire

En mémoire, les énumérations avec des données sont stockées sous la forme d'une petite *balise* entière, plus suffisamment de mémoire pour contenir tous les champs de la plus grande variante. Le champ de balise est destiné à l'usage interne de Rust. Il indique quel constructeur a créé la valeur et donc quels champs il possède.

À partir de Rust 1.56, tient dans 8 octets, comme illustré à la figure 10-1. `RoughTime`

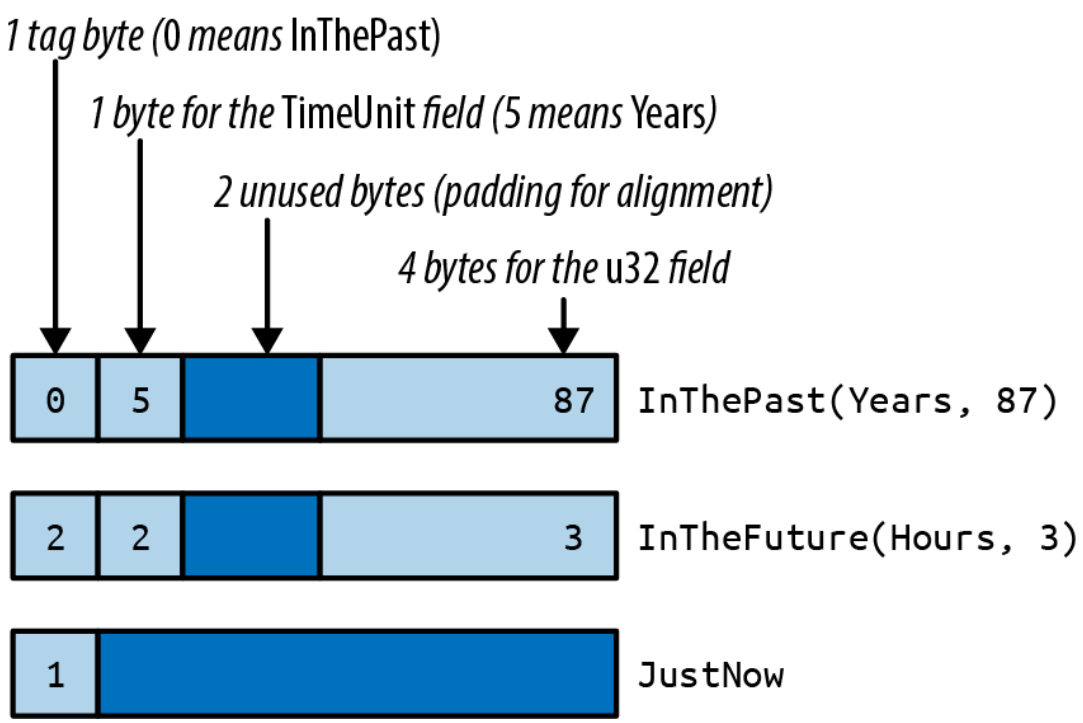


Figure 10-1. valeurs en mémoire `RoughTime`

Rust ne fait cependant aucune promesse sur la disposition de l'enum afin de laisser la porte ouverte à de futures optimisations. Dans certains cas, il serait possible d'emballer un enum plus efficacement que ne le suggère le chiffre. Par exemple, certaines structures génériques peuvent être stockées sans balise, comme nous le verrons plus tard.

Structures de données enrichies à l'aide d'enums

Les enums sont également utiles pour implémenter rapidement des structures de données arborescentes. Par exemple, supposons qu'un programme Rust doive fonctionner avec des données JSON arbitraires. En mémoire, tout document JSON peut être représenté sous la forme d'une valeur de ce type Rust :

```
use std::collections::HashMap;

enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>),
}
```

L'explication de cette structure de données en anglais ne peut pas améliorer beaucoup le code Rust. La norme JSON spécifie les différents types de données qui peuvent apparaître dans un document JSON : ,

valeurs booléennes, nombres, chaînes, tableaux de valeurs JSON et objets avec des clés de chaîne et des valeurs JSON. L'enum énonce simplement ces types. `null Json`

Ce n'est pas un exemple hypothétique. Un enum très similaire peut être trouvé dans , une bibliothèque de sérialisation pour Rust structs qui est l'une des caisses les plus téléchargées sur crates.io. `serde_json`

L'entourage de ce qui représente un ne sert qu'à rendre toutes les valeurs plus compactes. En mémoire, les valeurs de type occupent quatre mots machine. et les valeurs sont trois mots, et Rust ajoute un octet de balise. et les valeurs ne contiennent pas suffisamment de données pour utiliser tout cet espace, mais toutes les valeurs doivent avoir la même taille. L'espace supplémentaire n'est pas utilisé. La figure 10-2 montre quelques exemples de l'apparence réelle des valeurs en mémoire. `Box HashMap Object Json Json String Vec Null Boolean Json Json`

A est encore plus grand. Si nous devions laisser de la place pour cela dans chaque valeur, ils seraient assez grands, huit mots environ. Mais a est un seul mot : c'est juste un pointeur vers des données allouées en tas. Nous pourrions rendre encore plus compact en boxant plus de terrains. `HashMap Json Box<HashMap> Json`

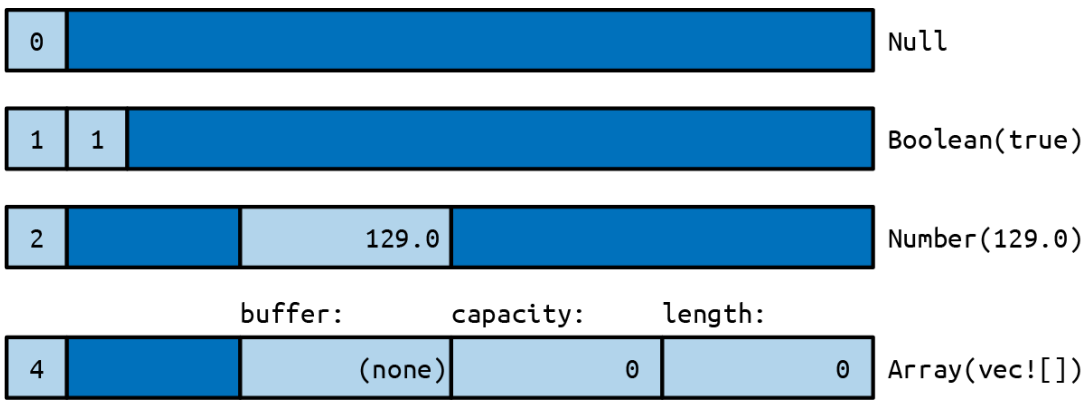


Figure 10-2. valeurs en mémoire `Json`

Ce qui est remarquable ici, c'est à quel point il a été facile de mettre cela en place. En C++, on peut écrire une classe pour ceci :

```
class JSON {
private:
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
```

```
            shared_ptr<string> str;
            shared_ptr<vector<JSON>> array;
            shared_ptr<unordered_map<string, JSON>> object;

            Data() {}
            ~Data() {}
            ...
        };

        Tag tag;
        Data data;

    public:
        bool is_null() const { return tag == Null; }
        bool is_boolean() const { return tag == Boolean; }
        bool get_boolean() const {
            assert(is_boolean());
            return data.boolean;
        }
        void set_boolean(bool value) {
            this->~JSON();  // clean up string/array/object value
            tag = Boolean;
            data.boolean = value;
        }
        ...
};
```

À 30 lignes de code, nous avons à peine commencé le travail. Cette classe aura besoin de constructeurs, d'un destructeur et d'un opérateur d'affectation. Une alternative serait de créer une hiérarchie de classes avec une classe de base et des sous-classes , , et ainsi de suite. Quoi qu'il en soit, lorsque cela sera fait, notre bibliothèque JSON C ++ aura plus d'une douzaine de méthodes. Il faudra un peu de lecture pour que d'autres programmeurs le prennent et l'utilisent. L'ensemble de Rust enum est composé de huit lignes de code. `JSON JSONBoolean JSONString`

Enums génériques

Enums peut être générique. Deux exemples tirés de la bibliothèque standard comptent parmi les types de données les plus utilisés dans la langue :

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

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Ces types sont maintenant assez familiers, et la syntaxe pour les enums génériques est la même que pour les structs génériques.

Un détail peu évident est que Rust peut éliminer le champ de balise lorsque le type est une référence, , ou un autre type de pointeur intelligent. Étant donné qu'aucun de ces types de pointeurs n'est autorisé à être nul, Rust peut représenter, disons, comme un seul mot machine: 0 pour et non nul pour pointeur. Cela rend ces types proches des analogues aux valeurs de pointeur C ou C ++ qui pourraient être nulles. La différence est que le système de type de Rust vous oblige à vérifier qu'un est avant de pouvoir utiliser son contenu. Cela élimine efficacement les déréférencements de pointeur
nuls. `Option<T>` `T` `Box` `Option<Box<i32>>` `None` `Some` `Option` `Option` `Some`

Les structures de données génériques peuvent être construites avec seulement quelques lignes de code :

```
// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}
```

Ces quelques lignes de code définissent un type qui peut stocker n'importe quel nombre de valeurs de type . `BinaryTree` `T`

Beaucoup d'informations sont emballées dans ces deux définitions, nous prendrons donc le temps de traduire le code mot à mot en anglais. Chaque valeur est soit ou . Si c'est , alors il ne contient aucune donnée du tout. Si , alors il a un , un pointeur vers un tas alloué
. `BinaryTree` `Empty` `NonEmpty` `Empty` `NonEmpty` `Box` `TreeNode`

Chaque valeur contient un élément réel, ainsi que deux autres valeurs. Cela signifie qu'un arbre peut contenir des sous-arbres, et donc un arbre peut avoir n'importe quel nombre de descendants. `TreeNode BinaryTree NonEmpty`

Une esquisse d'une valeur de type est illustrée à la figure 10-3. Comme avec , Rust élimine le champ de balise, de sorte qu'une valeur n'est qu'un mot de machine. `BinaryTree<&str> Option<Box<T>> BinaryTree`

La création d'un nœud particulier dans cette arborescence est simple :

```
use self::BinaryTree::*;
let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

Les arbres plus grands peuvent être construits à partir de plus petits:

```
let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
    right: mercury_tree,
}));
```

Naturellement, cette affectation transfère la propriété de et vers leur nouveau nœud parent. `jupiter_node mercury_node`

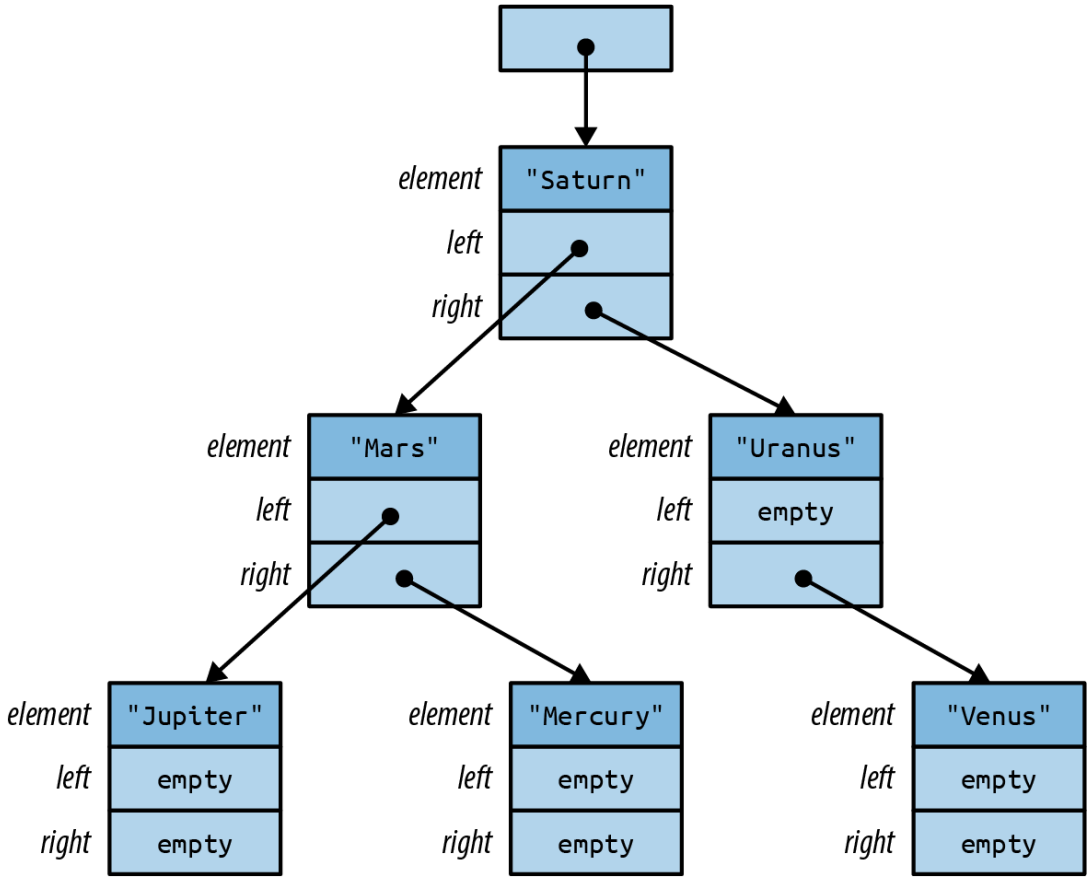


Figure 10-3. A contenant six chaînes `BinaryTree`

Les parties restantes de l'arbre suivent les mêmes schémas. Le nœud racine n'est pas différent des autres :

```
let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree,
}));
```

Plus loin dans ce chapitre, nous montrerons comment implémenter une méthode sur le type afin que nous puissions écrire à la place : `add BinaryTree`

```
let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

Quelle que soit la langue d'où vous venez, la création de structures de données comme dans Rust nécessitera probablement un peu de pratique. Il ne sera pas évident au début où mettre les es. Une façon de trouver un design qui fonctionnera est de dessiner une image comme la figure 10-3 qui montre comment vous voulez que les choses soient disposées en mémoire. Ensuite, revenez de l'image au code. Chaque collection de rectan-

gles est une structure ou un tuple; chaque flèche est un ou un autre pointeur intelligent. Déterminer le type de chaque champ est un peu un casse-tête, mais gérable. La récompense pour résoudre le puzzle est le contrôle de l'utilisation de la mémoire de votre programme. `BinaryTree Box Box`

Vient maintenant le « prix » que nous avons mentionné dans l'introduction. Le champ de balise d'un enum coûte un peu de mémoire, jusqu'à huit octets dans le pire des cas, mais c'est généralement négligeable. Le véritable inconvénient des enums (si on peut l'appeler ainsi) est que le code Rust ne peut pas jeter la prudence au vent et essayer d'accéder aux champs, qu'ils soient ou non réellement présents dans la valeur:

```
let r = shape.radius;  // error: no field `radius` on type `Shape`
```

La seule façon d'accéder aux données dans un enum est la manière sûre: utiliser des modèles.

Modèles

Rappelez-vous la définition de notre type de plus haut dans ce chapitre: `RoughTime`

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

Supposons que vous ayez une valeur et que vous souhaitiez l'afficher sur une page Web. Vous devez accéder aux champs et à l'intérieur de la valeur. Rust ne vous permet pas d'y accéder directement, en écrivant et , car après tout, la valeur pourrait être , qui n'a pas de champs. Mais alors, comment pouvez-vous extraire les données?
`RoughTime TimeUnit u32 rough_time.0 rough_time.1 RoughTime::JustNow`

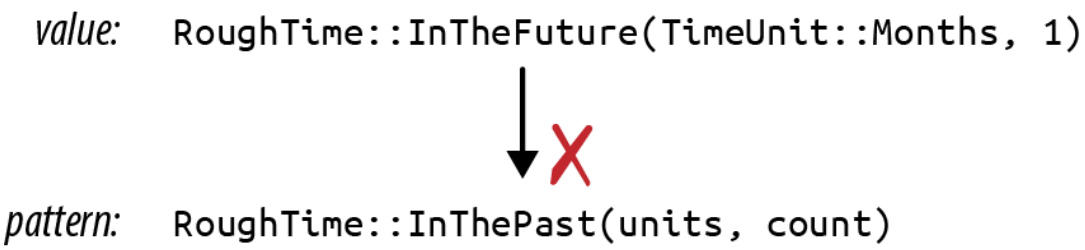
Vous avez besoin d'une expression : `match`

```
fn rough_time_to_english(rt: RoughTime) -> String {
    match rt {
        RoughTime::InThePast(units, count) =>
            format!("{} {} ago", count, units.plural()),
        RoughTime::JustNow =>
```

```
6                 format!("just now"),
7             RoughTime::InTheFuture(units, count) =>
8                 format!("{} {} from now", count, units.plural()),
9         }
10    }
```

`match` effectue l'appariement des modèles; dans cet exemple, les *motifs* sont les pièces qui apparaissent avant le symbole sur les lignes 3, 5 et 7. Les modèles qui correspondent aux valeurs ressemblent aux expressions utilisées pour créer des valeurs. Ce n'est pas une coïncidence. Les expressions *produisent des* valeurs; les modèles *consomment* des valeurs. Les deux utilisent beaucoup de la même syntaxe. `=> RoughTime RoughTime`

Passons en revue ce qui se passe lorsque cette expression s'exécute. Supposons que la valeur . Rust essaie d'abord de faire correspondre cette valeur au modèle de la ligne 3. Comme vous pouvez le voir à la figure 10-4, il ne correspond pas. `match rt RoughTime::InTheFuture(TimeUnit::Months, 1)`



Graphique 10-4. Une valeur et un modèle qui ne correspondent pas `RoughTime`

Le motif correspondant à un enum, une structure ou un tuple fonctionne comme si Rust effectuait une simple analyse de gauche à droite, en vérifiant chaque composant du motif pour voir si la valeur lui correspond. Si ce n'est pas le cas, Rust passe au modèle suivant.

Les motifs des lignes 3 et 5 ne correspondent pas. Mais le modèle de la ligne 7 réussit (Figure 10-5).

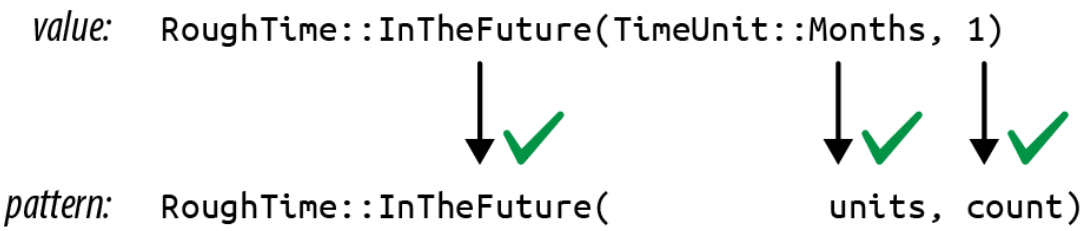


Figure 10-5. Un match réussi

Lorsqu'un modèle contient des identificateurs simples tels que et , ceux-ci deviennent des variables locales dans le code suivant le modèle. Tout ce qui est présent dans la valeur est copié ou déplacé dans les nouvelles variables. Rust stocke dans et dans , exécute la ligne 8 et renvoie la chaîne

`.units count TimeUnit::Months units 1 count "1 months from now"`

Cette sortie a un problème grammatical mineur, qui peut être résolu en ajoutant un autre bras au : `match`

```
    RoughTime::InTheFuture(unit, 1) =>
        format!("a {} from now", unit.singular()),
```

Ce bras ne correspond que si le champ est exactement 1. Notez que ce nouveau code doit être ajouté avant la ligne 7. Si nous l'ajoutons à la fin, Rust n'y arrivera jamais, car le motif de la ligne 7 correspond à toutes les valeurs. Le compilateur Rust vous avertira d'un « modèle inaccessible » si vous faites ce genre d'erreur. `count InTheFuture`

Même avec le nouveau code, présente toujours un problème: le résultat n'est pas tout à fait correct. Telle est la langue anglaise. Cela aussi peut être corrigé en ajoutant un autre bras au `.RoughTime::InTheFuture(TimeUnit::Hours, 1) "a hour from now" match`

Comme le montre cet exemple, la correspondance de motifs fonctionne main dans la main avec les enums et peut même tester les données qu'ils contiennent, ce qui constitue un remplacement puissant et flexible de l'instruction C. Jusqu'à présent, nous n'avons vu que des modèles qui correspondent aux valeurs enum. Il y a plus que cela. Les modèles de rouille sont leur propre petit langage, résumé dans le tableau 10-1. Nous passerons la majeure partie du reste du chapitre sur les fonctionnalités présentées dans ce tableau. `match switch`

Tableau 10-1. Modèles

Type de motif	Exemple	Notes
Littéral	`100` `"name"`	Correspond à une valeur exacte; le nom d'un est également autorisé `co nst`

Gamme	`0 ..= 100`
	`'a' ..= 'k'`
	`256..`
		Correspond à n'importe quelle valeur de plage, y compris la valeur finale si elle est donnée

Génériques	`_`	Correspond à n'importe quelle valeur et l'ignore
Variable	`name` `mut count`	J'aime mais déplace ou copie la valeur dans une nouvelle variable locale _
`ref` variable	`ref field` `ref mut fie ld`	Emprunte une référence à la valeur correspondante au lieu de la déplacer ou de la copier
Liaison avec sous-modèle	`val @ 0 ..= 99` `ref circle @ Shape::Circl e { .. }`	Correspond au motif à droite de , en utilisant le nom de la variable à gauche `@`

Motif Enum	`Some(value)`
	`None`
	`Pet::Orca`	

Modèle de tuple	`(key, valu e)` `(r, g, b)`	
Modèle de tableau	`[a, b, c, d, e, f, g]` `[heading, ca rom, correct ion]`	

Type de motif	Exemple	Notes

Motif de tranche	`[first, second]`
	`[first, _, third]`
	`[first, .., nth]`
	`[]`	

Modèle de structure	`Color(r, g, b)`
	`Point { x, y }`
	`Card { suit: Clubs, rank: n }`
	`Account { id, name, .. }`	

Référence	`&value` `&(k, v)`	Correspond uniquement aux valeurs de référence
Ou des modèles	`'a' | 'A'` `Some("left" | "right")`	
Expression de garde	`x if x * x <= r2`	En seulement (non valide en , etc.) `match let`

Littéraux, variables et caractères génériques dans les modèles

Jusqu'à présent, nous avons montré des expressions travaillant avec des enums. D'autres types peuvent également être appariés. Lorsque vous avez besoin d'une instruction C, utilisez avec une valeur entière. Les littéraux entiers aiment et peuvent servir de motifs: `match switch match 0 1`

```
match meadow.count_rabbits() {
    0 => {}  // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
```

```
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

Le motif correspond s'il n'y a pas de lapins dans le pré. correspond s'il n'y en a qu'un. S'il y a deux lapins ou plus, nous atteignons le troisième modèle, . Ce modèle n'est qu'un nom de variable. Elle peut correspondre à n'importe quelle valeur et la valeur correspondante est déplacée ou copiée dans une nouvelle variable locale. Donc, dans ce cas, la valeur de est stockée dans une nouvelle variable locale , que nous imprimons ensuite. `0 1 n meadow.count_rabbits() n`

D'autres littéraux peuvent également être utilisés comme motifs, y compris les booléens, les caractères et même les chaînes:

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese" => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other => return parse_error("calendar", other),
};
```

Dans cet exemple, sert de modèle fourre-tout comme dans l'exemple précédent. Ces modèles jouent le même rôle qu'un cas dans une instruction, correspondant à des valeurs qui ne correspondent à aucun des autres modèles. `other n default switch`

Si vous avez besoin d'un modèle fourre-tout, mais que vous ne vous souciez pas de la valeur correspondante, vous pouvez utiliser un seul trait de soulignement comme motif, le *modèle générique* : _

```
let caption = match photo.tagged_pet() {
    Pet::Tyrannosaur => "RRRAAAAAHHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // generic caption, works for any pet
};
```

Le modèle générique correspond à n'importe quelle valeur, mais sans le stocker n'importe où. Étant donné que Rust exige que chaque expression gère toutes les valeurs possibles, un caractère générique est souvent requis à la fin. Même si vous êtes très sûr que les cas restants ne peuvent pas se produire, vous devez au moins ajouter un bras de secours, peut-être un bras qui panique: `match`

```
// There are many Shapes, but we only support "selecting"
// either some text, or everything in a rectangular area.
// You can't select an ellipse or trapezoid.
match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start, end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type"),
}
```

Modèles Tuple et Struct

Les motifs de tuple correspondent aux tuples. Ils sont utiles chaque fois que vous souhaitez impliquer plusieurs données dans un seul : `match`

```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else",
    }
}
```

Les motifs Struct utilisent des accolades bouclées, tout comme les expressions struct. Ils contiennent un sous-modèle pour chaque champ :

```
match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}m, {}m)", x, y),
}
```

Dans cet exemple, si le premier bras correspond, alors est stocké dans la nouvelle variable locale . `balloon.location.y height`

Supposons que est . Comme toujours, Rust vérifie chaque composant de chaque motif à tour [de rôle Figure 10-6](). `balloon.location Point { x: 30, y: 40 }`

Figure 10-6. Correspondance de motifs avec des structures

Le deuxième bras correspond, de sorte que la sortie serait . `at (30m, 40m)`

Les modèles comme sont courants lors de la correspondance des structs, et les noms redondants sont un encombrement visuel, donc Rust a un raccourci pour cela: . Le sens est le même. Ce modèle stocke toujours le champ d'un point dans un nouveau local et son champ dans un nouveau local. `Point { x: x, y: y } Point {x, y} x x y y`

Même avec le raccourci, il est fastidieux de faire correspondre une grande structure lorsque nous ne nous soucions que de quelques champs:

```
match get_account(id) {
    ...
    Some(Account {
            name, language,  // <--- the 2 things we care about
            id: _, status: _, address: _, birthday: _, eye_color: _,
            pet: _, security_question: _, hashed_innermost_secret: _,
            is_adamantium_preferred_customer: _, }) =>
        language.show_custom_greeting(name),
}
```

Pour éviter cela, dites à Rust que vous ne vous souciez d'aucun des autres champs: `..`

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name),
```

Modèles de tableau et de tranche

Les modèles de tableau correspondent aux tableaux. Ils sont souvent utilisés pour filtrer certaines valeurs de cas spéciaux et sont utiles chaque fois que vous travaillez avec des tableaux dont les valeurs ont une signification différente en fonction de la position.

Par exemple, lors de la conversion des valeurs de couleur de teinte, de saturation et de luminosité (HSL) en valeurs de couleur rouge, vert, bleu (RVB), les couleurs sans luminosité ou avec une luminosité totale sont

simplement noires ou blanches. Nous pourrions utiliser une expression pour traiter ces cas simplement. `match`

```
fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {
    match hsl {
        [_, _, 0] => [0, 0, 0],
        [_, _, 255] => [255, 255, 255],
        ...
    }
}
```

Les modèles de tranches sont similaires, mais contrairement aux tableaux, les tranches ont des longueurs variables, de sorte que les tapotements de tranche correspondent non seulement aux valeurs, mais également à la longueur. dans un motif de tranche correspond à un nombre quelconque d'éléments : `..`

```
fn greet_people(names: &[&str]) {
    match names {
        [] => { println!("Hello, nobody.") },
        [a] => { println!("Hello, {}.", a) },
        [a, b] => { println!("Hello, {} and {}.", a, b) },
        [a, .., b] => { println!("Hello, everyone from {} to {}.", a, b)
    }
}
```

Modèles de référence

Les motifs de rouille prennent en charge deux fonctionnalités pour travailler avec des références. les motifs empruntent des parties d'une valeur correspondante. les modèles correspondent aux références. Nous allons d'abord couvrir les modèles. `ref & ref`

La correspondance d'une valeur non copiable déplace la valeur. En continuant avec l'exemple de compte, ce code ne serait pas valide :

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account);  // error: borrow of moved value: `a
    }
}
```

Ici, les champs et sont déplacés dans les variables locales et . Le reste est abandonné. C'est pourquoi nous ne pouvons pas emprunter une référence à cela par la suite. `account.name account.language name language account`

Si et étaient les deux valeurs copiables, Rust copierait les champs au lieu de les déplacer, et ce code serait correct. Mais supposons que ce soient des s. Que pouvons-nous faire? `name language String`

Nous avons besoin d'une sorte de modèle qui *emprunte des valeurs correspondantes* au lieu de les déplacer. C'est exactement ce que fait le mot-clé : `ref`

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account);  // ok
    }
}
```

Maintenant, les variables locales et sont des références aux champs correspondants dans . Étant donné qu'il n'est qu'emprunté, pas consommé, il est acceptable de continuer à appeler des méthodes dessus. `name language account account`

Vous pouvez utiliser pour emprunter des références : `ref mut mut`

```
match line_result {
    Err(ref err) => log_error(err),  // `err` is &Error (shared ref)
    Ok(ref mut line) => {            // `line` is &mut String (mut ref)
        trim_comments(line);         // modify the String in place
        handle(line);
    }
}
```

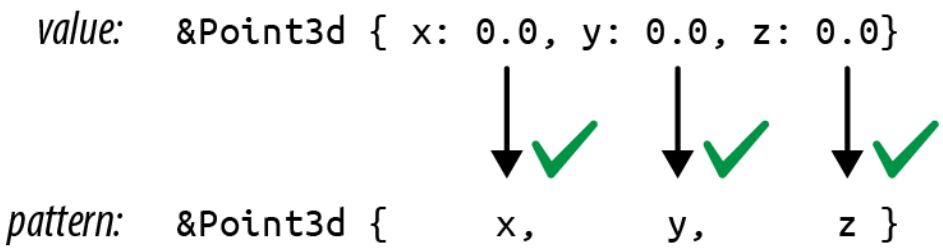
Le modèle correspond à n'importe quel résultat de réussite et emprunte une référence à la valeur de succès stockée à l'intérieur. `Ok(ref mut line) mut`

Le type opposé de modèle de référence est le modèle. Un modèle commençant par correspond à une référence : `& &`

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

Dans cet exemple, supposons qu'il renvoie une référence à un champ privé de , un motif commun dans Rust. La valeur renvoyée est l'adresse d'un fichier . Si le centre est à l'origine, renvoie `.sphere.center() sphere Point3d sphere.center() &Point3d { x: 0.0, y: 0.0, z: 0.0 }`

La correspondance des motifs se déroule comme illustré à la figure 10-7.



Graphique 10-7. Correspondance de motifs avec des références

C'est un peu délicat car Rust suit un pointeur ici, une action que nous associons généralement à l'opérateur, pas à l'opérateur. La chose à retenir est que les motifs et les expressions sont des opposés naturels. L'expression transforme deux valeurs en un nouveau tuple, mais le motif fait le contraire : il correspond à un tuple et décompose les deux valeurs. C'est la même chose avec . Dans une expression, crée une référence. Dans un modèle, correspond à une référence. `* & (x, y) (x, y) & & &`

Faire correspondre une référence suit toutes les règles auxquelles nous nous attendons. Les durées de vie sont appliquées. Vous ne pouvez pas y accéder via une référence partagée. Et vous ne pouvez pas déplacer une valeur hors d'une référence, même d'une référence. Lorsque nous faisons correspondre , les variables , et recevons des copies des coordonnées, en laissant la valeur d'origine intacte. Cela fonctionne parce que ces champs sont copiables. Si nous essayons la même chose sur une structure avec des champs non copiables, nous obtiendrons une erreur `:mut mut &Point3d { x, y, z } x y z Point3d`

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) =>  // error: can't move out of borrow
        ...
    None => {}
}
```

Mettre au rebut une voiture empruntée pour des pièces n'est pas agréable, et Rust ne le supportera pas. Vous pouvez utiliser un motif pour emprunter une référence à un article. Vous ne le possédez tout simplement pas: `ref`

```
        Some(&Car { ref engine, .. }) =>  // ok, engine is a reference
```

Regardons un autre exemple de modèle. Supposons que nous ayons un itérateur sur les caractères d'une chaîne, et qu'il ait une méthode qui renvoie une référence au caractère suivant, le cas échéant. (Les itérateurs peekables renvoient en fait un , comme nous le verrons au chapitre 15.) `& chars chars.peek() Option<&char> Option<&ItemType>`

Un programme peut utiliser un modèle pour obtenir le caractère pointu : `&`

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars"),
}
```

Gardes de match

Parfois, un bras d'allumette a des conditions supplémentaires qui doivent être remplies avant de pouvoir être considéré comme un match. Supposons que nous mettions en œuvre un jeu de société avec des espaces hexagonaux et que le joueur clique simplement pour déplacer une pièce. Pour confirmer que le clic était valide, nous pouvons essayer quelque chose comme ceci:

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) =>  // try to match if user clicked the curren
                              // (it doesn't work: see explanation below
            Err("You are already there! You must click somewhere else.")
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

Cela échoue car les identificateurs dans les modèles introduisent de *nouvelles* variables. Le modèle ici crée une nouvelle variable locale , ombrageant l'argument . Rust émet plusieurs avertissements à propos de ce code, en particulier, le dernier bras du est inaccessible. Une façon de résoudre ce problème consiste simplement à utiliser une expression dans le bras

d'allumette: `Some(current_hex) current_hex current_hex match i f`

```
    match point_to_hex(click) {
        None => Err("That's not a game space."),
        Some(hex) => {
            if hex == current_hex {
                Err("You are already there! You must click somewhere else")
            } else {
                Ok(hex)
            }
        }
    }
```

Mais Rust fournit également des *gardes d'allumettes*, des conditions supplémentaires qui doivent être vraies pour qu'un bras d'allumette s'applique, écrit comme , entre le motif et le jeton du bras: `if CONDITION =>`

```
    match point_to_hex(click) {
        None => Err("That's not a game space."),
        Some(hex) if hex == current_hex =>
            Err("You are already there! You must click somewhere else"),
        Some(hex) => Ok(hex)
    }
```

Si le modèle correspond, mais que la condition est fausse, la correspondance se poursuit avec le bras suivant.

Faire correspondre plusieurs possibilités

Un modèle du formulaire correspond si l'un ou l'autre des sous-modèles correspond à : `pat1 | pat2`

```
    let at_end = match chars.peek() {
        Some(&'\r' | &'\n') | None => true,
        _ => false,
    };
```

Dans une expression, est l'opérateur OR binaire, mais ici il fonctionne plus comme le symbole dans une expression régulière. est défini sur si est , ou un maintien d'un retour chariot ou d'un saut de ligne. `| | at_end true chars.peek() None Some`

Permet de faire correspondre toute une plage de valeurs. Les modèles de plage incluent les valeurs de début et de fin, ce qui correspond à tous les

chiffres ASCII : `..= '0' ..= '9'`

```
match next_char {
    '0'..='9' => self.read_number(),
    'a'..='z' | 'A'..='Z' => self.read_word(),
    ' ' | '\t' | '\n' => self.skip_whitespace(),
    _ => self.handle_punctuation(),
}
```

Rust autorise également des modèles de plage tels que , qui correspondent à n'importe quelle valeur allant jusqu'à la valeur maximale du type. Cependant, les autres variétés de gammes exclusives à la fin, comme ou , et les gammes illimitées comme ne sont pas encore autorisées dans les modèles. `x.. x 0..100 ..100 ..`

Liaison avec @ Patterns

Enfin, correspond exactement comme le donné , mais en cas de succès, au lieu de créer des variables pour des parties de la valeur correspondante, il crée une seule variable et déplace ou copie la valeur entière dans celle-ci. Par exemple, supposons que vous ayez ce code : `x @ pattern pattern x`

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) => {
        optimized_paint(&Shape::Rect(top_left, bottom_right))
    }
    other_shape => {
        paint_outline(other_shape.get_outline())
    }
}
```

Notez que le premier cas décompresse une valeur, uniquement pour reconstruire une valeur identique sur la ligne suivante. Cela peut être réécrit pour utiliser un modèle : `Shape::Rect Shape::Rect @`

```
    rect @ Shape::Rect(..) => {
        optimized_paint(&rect)
    }
```

`@` les motifs sont également utiles avec les plages:

```
match chars.next() {
    Some(digit @ '0'..='9') => read_number(digit, chars),
```

```
    ...
},
```

Où les modèles sont autorisés

Bien que les motifs soient les plus importants dans les expressions, ils sont également autorisés à plusieurs autres endroits, généralement à la place d'un identifiant. La signification est toujours la même : au lieu de simplement stocker une valeur dans une seule variable, Rust utilise la correspondance de motif pour séparer la valeur. `match`

Cela signifie que les modèles peuvent être utilisés pour...

```
// ...unpack a struct into three new local variables
let Track { album, track_number, title, .. } = song;

// ...unpack a function argument that's a tuple
fn distance_to((x, y): (f64, f64)) -> f64 { ... }

// ...iterate over keys and values of a HashMap
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// ...automatically dereference an argument to a closure
// (handy because sometimes other code passes you a reference
// when you'd rather have a copy)
let sum = numbers.fold(0, |a, &num| a + num);
```

Chacun d'entre eux permet d'économiser deux ou trois lignes de code standard. Le même concept existe dans d'autres langages : en JavaScript, on l'appelle *déstructuration*, tandis qu'en Python, c'est le *déballage*.

Notez que dans les quatre exemples, nous utilisons des modèles qui sont garantis pour correspondre. Le motif correspond à toutes les valeurs possibles du type struct, correspond à n'importe quelle paire, etc. Les motifs qui correspondent toujours sont spéciaux dans Rust. Ils sont *appelés modèles irréfutables*, et ce sont les seuls modèles autorisés aux quatre endroits montrés ici (après , dans les arguments de fonction, après et dans les arguments de fermeture). `Point3d { x, y, z } Point3d (x, y) (f64, f64) let for`

Un *modèle réfutable est un modèle* qui peut ne pas correspondre, comme , qui ne correspond pas à un résultat d'erreur, ou , qui ne correspond pas au caractère . Les motifs réfutables peuvent être utilisés dans les bras, car

ils sont conçus pour eux: si un motif ne correspond pas, il est clair ce qui se passe ensuite. Les quatre exemples précédents sont des endroits dans les programmes Rust où un modèle peut être pratique, mais le langage ne permet pas l'échec de la correspondance. `Ok(x) '0' ..= '9' 'Q' match match`

Les motifs réfutables sont également autorisés dans et les expressions, qui peuvent être utilisées pour... `if let while let`

```
// ...handle just one enum variant specially
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

// ...run some code only if a table lookup succeeds
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ...repeatedly try something until it succeeds
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // let the user try again (it might still be a human)
}

// ...manually loop over an iterator
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

Pour plus d'informations sur ces expressions, voir « if let » et « Loops ».

Remplissage d'un arbre binaire

Plus tôt, nous avons promis de montrer comment implémenter une méthode, , qui ajoute un nœud à un de ce type: `BinaryTree::add() BinaryTree`

```
// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
```

```
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}
```

Vous en savez maintenant assez sur les modèles pour écrire cette méthode. Une explication des arbres de recherche binaires dépasse le cadre de ce livre, mais pour les lecteurs déjà familiers avec le sujet, il vaut la peine de voir comment cela se passe dans Rust.

```
 1  impl<T: Ord> BinaryTree<T> {
 2      fn add(&mut self, value: T) {
 3          match *self {
 4              BinaryTree::Empty => {
 5                  *self = BinaryTree::NonEmpty(Box::new(TreeNode {
 6                      element: value,
 7                      left: BinaryTree::Empty,
 8                      right: BinaryTree::Empty,
 9                  }))
10              }
11              BinaryTree::NonEmpty(ref mut node) => {
12                  if value <= node.element {
13                      node.left.add(value);
14                  } else {
15                      node.right.add(value);
16                  }
17              }
18          }
19      }
20  }
```

La ligne 1 indique à Rust que nous définissons une méthode sur des types ordonnés. C'est exactement la même syntaxe que nous utilisons pour définir des méthodes sur des structures génériques, expliquée dans « Définition de méthodes avec impl ». `BinaryTree`

Si l'arbre existant est vide, c'est le cas facile. Les lignes 5 à 9 s'exécutent, changeant l'arbre en un. L'appel à ici alloue un nouveau dans le tas. Lorsque nous avons terminé, l'arbre contient un élément. Ses sous-arbres gauche et droit sont tous deux
. `*self Empty NonEmpty Box::new() TreeNode Empty`

Si n'est pas vide, nous faisons correspondre le modèle de la ligne
11: `*self`

```
    BinaryTree::NonEmpty(ref mut node) => {
```

Ce modèle emprunte une référence modifiable au , afin que nous puissions accéder aux données de ce nœud d'arborescence et les modifier. Cette référence est nommée , et elle est dans la portée de la ligne 12 à la ligne 16. Comme il y a déjà un élément dans ce nœud, le code doit appeler de manière récursive pour ajouter le nouvel élément à la sous-arborescence gauche ou droite. `Box<TreeNode<T>> node .add()`

La nouvelle méthode peut être utilisée comme ceci:

```
let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
```

Vue d'ensemble

Les enums de Rust sont peut-être nouveaux dans la programmation de systèmes, mais ils ne sont pas une idée nouvelle. Voyageant sous divers noms à consonance académique, comme les *types de données algébriques*, ils sont utilisés dans les langages de programmation fonctionnels depuis plus de quarante ans. On ne sait pas pourquoi si peu d'autres langues de la tradition C en ont jamais eu. Peut-être est-ce simplement que pour un concepteur de langage de programmation, combiner des variantes, des références, la mutabilité et la sécurité de la mémoire est extrêmement difficile. Les langages de programmation fonctionnels se passent de mutabilité. Les C, en revanche, ont des variantes, des pointeurs et une mutabilité, mais sont si spectaculairement dangereux que même en C, ils sont un dernier recours. Le vérificateur d'emprunt de Rust est la magie qui permet de combiner les quatre sans compromis. `union`

La programmation, c'est le traitement des données. Obtenir des données dans la bonne forme peut faire la différence entre un petit programme rapide et élégant et un enchevêtrement lent et gigantesque de ruban adhésif et d'appels de méthode virtuelle.

C'est le problème que les enums d'espace abordent. Ils sont un outil de conception pour obtenir des données dans la bonne forme. Pour les cas où une valeur peut être une chose, ou une autre chose, ou peut-être rien du tout, les enums sont meilleurs que les hiérarchies de classes sur

chaque axe : plus rapides, plus sûrs, moins de code, plus faciles à documenter.

Le facteur limitant est la flexibilité. Les utilisateurs finaux d'un enum ne peuvent pas l'étendre pour ajouter de nouvelles variantes. Les variantes ne peuvent être ajoutées qu'en modifiant la déclaration enum. Et lorsque cela se produit, le code existant se brise. Chaque expression qui correspond individuellement à chaque variante de l'enum doit être revisitée – elle a besoin d'un nouveau bras pour gérer la nouvelle variante. Dans certains cas, la flexibilité de trading pour la simplicité est juste du bon sens. Après tout, la structure de JSON ne devrait pas changer. Et dans certains cas, revisiter toutes les utilisations d'un enum quand il change est exactement ce que nous voulons. Par exemple, lorsque an est utilisé dans un compilateur pour représenter les différents opérateurs d'un langage de programmation, l'ajout d'un nouvel opérateur *doit* impliquer de toucher tout le code qui gère les opérateurs. `match enum`

Mais parfois, plus de flexibilité est nécessaire. Pour ces situations, Rust a des traits, le sujet de notre prochain chapitre.