Chapitre 10. Énumérations et modèles

> *Étonnant à quel point les trucs informatiques ont du sens vus comme une privation tragique de types de somme (cf. privation de lambdas).*
>
> —Graydon Hoare

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

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

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

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

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

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

Énumérations

Énumérations simples de style Csont simples :

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

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

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

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

ou avec tous ses constructeurs :

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

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

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

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

Pour importer les constructeurs d'une énumération déclarée dans le module courant, utilisez un `self` import :

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

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

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

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

Sinon, Rust attribuera les numéros pour vous, en commençant par 0.

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

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

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

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

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

Cependant, la conversion dans l'autre sens, de l'entier à l'énumération, ne l'est pas. Contrairement à C et C++, Rust garantit qu'une valeur enum n'est jamais qu'une des valeurs énoncées dans la `enum` déclaration. Un transtypage non vérifié d'un type entier vers un type enum pourrait rompre cette garantie, il n'est donc pas autorisé. Vous pouvez soit écrire votre propre conversion vérifiée :

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

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

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

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

Les énumérations peuvent avoir des méthodes, tout comme les structures :

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

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

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

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

Énumérations avec des données

Certains programmestoujours besoin d'afficher des dates et des heures complètes jusqu'à la milliseconde, mais pour la plupart des applications, il est plus convivial d'utiliser une approximation approximative, comme "il y a deux mois". Nous pouvons écrire une énumération pour aider à cela, en utilisant l'énumération définie précédemment :

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

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

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

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

Les énumérations peuvent également avoir des *variantes* de structure , qui contiennent des champs nommés, tout comme les structures ordinaires :

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

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

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

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

Tous les constructeurs et champs d'une énumération partagent la même visibilité que l'énumération elle-même.

Énumérations en mémoire

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

Depuis Rust 1.56, `RoughTime` tient dans 8 octets, comme illustré à la Figure 10-1 .

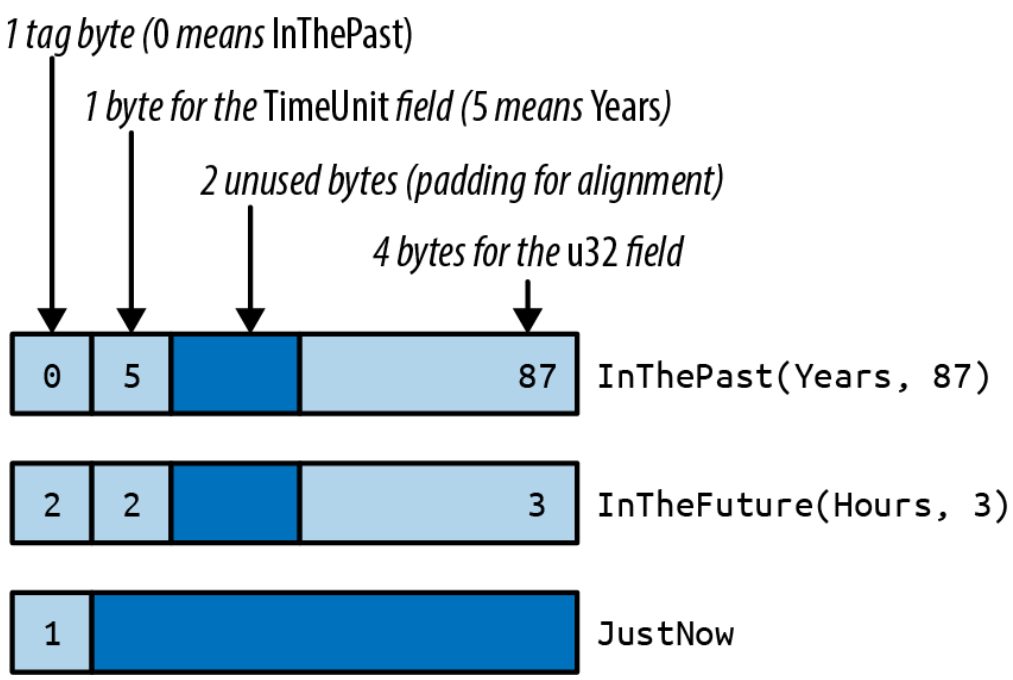

Illustration 10-1. `RoughTime` valeurs en mémoire

Cependant, Rust ne fait aucune promesse concernant la disposition des enums, afin de laisser la porte ouverte à de futures optimisations. Dans certains cas, il serait possible d'emballer une énumération plus efficacement que ne le suggère la figure. Par exemple, certaines structures génériques peuvent être stockées sans balise, comme nous le verrons plus tard.

Structures de données riches à l'aide d'énumérations

Énumérationssont également utiles pour implémenter rapidement des structures de données arborescentes. Par exemple, supposons qu'un programme Rust doive travailler avec des données JSON arbitraires. En mémoire, tout JSONdocument peut être représenté comme une valeur de ce type Rust :

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

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

L'explication de cette structure de données en anglais ne peut pas beaucoup améliorer le code Rust. La norme JSON spécifie les différents types

de données qui peuvent apparaître dans un document JSON : `null` , valeurs booléennes, nombres, chaînes, tableaux de valeurs JSON et objets avec des clés de chaîne et des valeurs JSON. L' `Json` énumération énonce simplement ces types.

Ceci n'est pas un exemple hypothétique. Une énumération très similaire peut être trouvée dans `serde_json` , une sérialisationbibliothèque pour les structures Rust qui est l'une des caisses les plus téléchargées sur crates.io.

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

A `HashMap` est encore plus grand. Si nous devions lui laisser de la place dans chaque `Json` valeur, elles seraient assez grandes, huit mots environ. Mais a `Box<HashMap>` est un mot unique : c'est juste un pointeur vers des données allouées par tas. On pourrait rendre `Json` encore plus compact en boxant plus de champs.

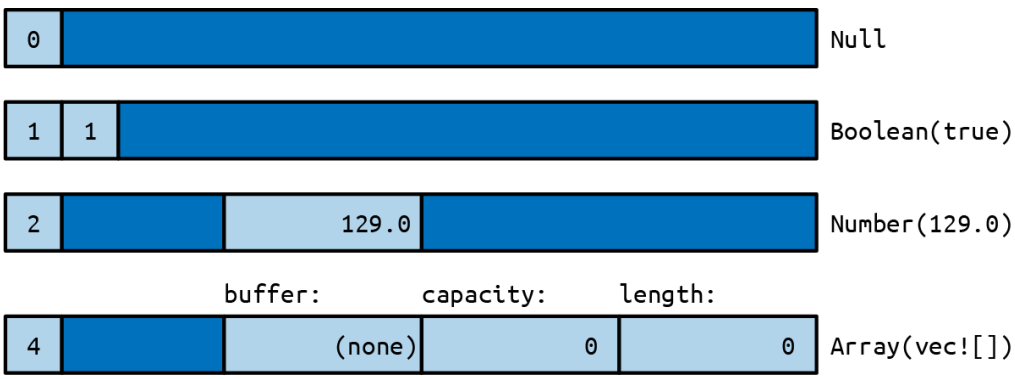

Illustration 10-2. `Json` valeurs en mémoire

Ce qui est remarquable ici, c'est la facilité avec laquelle il a été mis en place. En C++, on pourrait écrire une classe pour cela :

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

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

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

        Tag tag;
        Data data;

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

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

Énumérations génériques

Énumérationspeut être générique. Deux exemples de la bibliothèque standard font partie des types de données les plus utilisés dans le langage :

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

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

Ces types sont déjà assez familiers et la syntaxe des énumérations génériques est la même que celle des structures génériques.

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

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

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

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

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

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

Chaque `TreeNode` valeur contient un élément réel, ainsi que deux autres `BinaryTree` valeurs. Cela signifie qu'un arbre peut contenir des sous-

arbres, et donc un `NonEmpty` arbre peut avoir n'importe quel nombre de descendants.

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

Construire un nœud particulier dans cet arbre est simple :

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

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

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

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

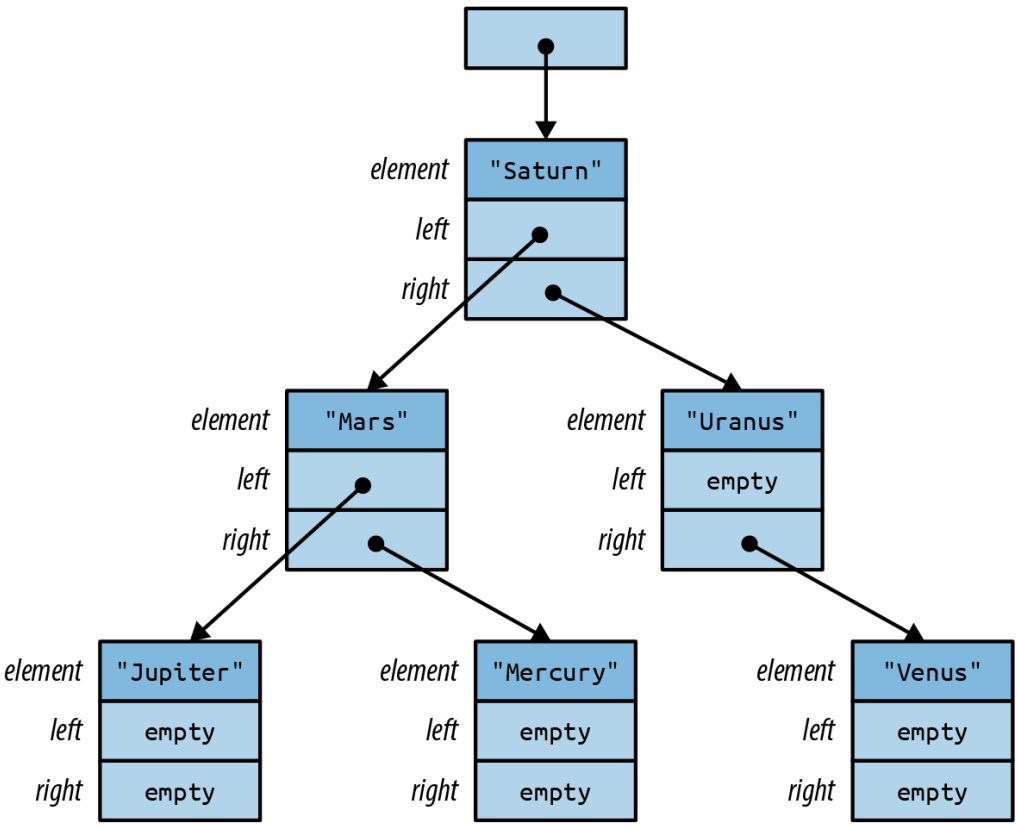

Illustration 10-3. A `BinaryTree` contenant six cordes

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

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

Plus loin dans ce chapitre, nous montrerons comment implémenter une `add` méthode sur le `BinaryTree` typepour que l'on puisse plutôt écrire :

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

Quelle que soit la langue d'où vous venez, la création de structures de données comme `BinaryTree` dans Rust nécessitera probablement un peu de pratique. Il ne sera pas évident au début où mettre les `Box` es. Une façon de trouver une conception qui fonctionnera consiste à dessiner une image comme la figure 10-3 qui montre comment vous voulez que les choses soient disposées en mémoire. Revenez ensuite de l'image au code. Chaque collection de rectangles est une structure ou un tuple ; chaque flèche est un `Box` ou un autre pointeur intelligent. Déterminer le type de chaque champ est un peu un casse-tête, mais gérable. La récompense pour résoudre le puzzle est le contrôle de l'utilisation de la mémoire de votre programme.

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

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

Le seul moyen d'accéder aux données d'une énumération est le moyen sûr : utiliser des modèles.

Motifs

Rappel de la définitionde notre `RoughTime` type au début de ce chapitre :

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

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

Vous avez besoin d'une `match` expression:

```
1  fn rough_time_to_english(rt: RoughTime) -> String {
2      match rt {
3          RoughTime:: InThePast(units, count) =>
4              format!("{} {} ago", count, units.plural()),
5          RoughTime:: JustNow =>
6              format!("just now"),
7          RoughTime::InTheFuture(units, count) =>
8              format!("{} {} from now", count, units.plural()),
9      }
10 }
```

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

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

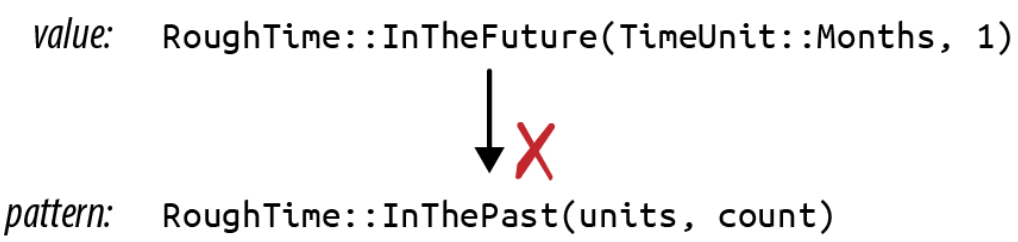

Illustration 10-4. Une `RoughTime` valeur et un modèle qui ne correspondent pas

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

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

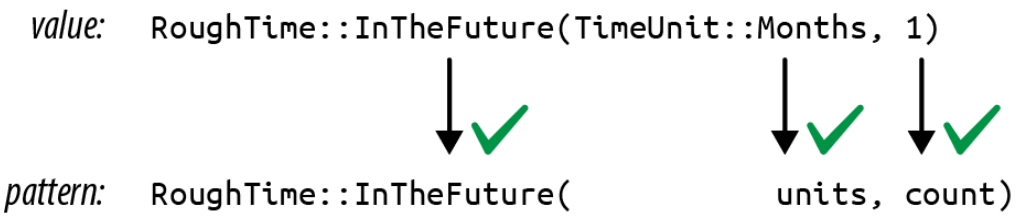

Illustration 10-5. Un match réussi

Lorsqu'un modèle contient des identifiants simples comme `units` et `count` , ceux-ci deviennent des variables locales dans le code suivant le modèle. Tout ce qui est présent dans la valeur est copié ou déplacé dans les nouvelles variables. Rust stocke `TimeUnit::Months` dans `units` et `1` dans `count` , exécute la ligne 8 et renvoie la chaîne `"1 months from now"` .

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

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

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

Même avec le nouveau code, `RoughTime::InTheFuture(TimeUnit::Hours, 1)` présente toujours un problème : le résultat `"a hour from now"` n'est pas tout à fait cor-

rect. Telle est la langue anglaise. Cela aussi peut être corrigé en ajoutant un autre bras au `match` .

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

Tableau 10-1. Motifs

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

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

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

Modèle d'énumération	`Some(value)`
	`None`
	`Pet::Orca`	

Modèle de tuple	`(key, value)` `(r, g, b)`	
Modèle de tableau	`[a, b, c, d, e, f, g]` `[heading, carom, correction]`	

Type de motif	Exemple	Remarques

Modèle 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)`	Ne correspond qu'aux valeurs de référence
Ou des motifs	`'a' | 'A'` `Some("left" | "right")`	
Expression de garde	`x if x * x <= r2`	En `match` seulement (non valide en `let`, etc.)

Littéraux, variables et caractères génériques dans les modèles

Jusqu'à présent, nous avons montré `match` des expressions fonctionnant avec des énumérations. D'autres types peuvent également être assortis. Lorsque vous avez besoin de quelque chose comme une `switch` instruction C, à utiliser `match` avec une valeur entière. Littéraux entiersaiment `0` et `1` peuvent servir de motifs :

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

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

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

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

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

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

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

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

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

```
        Shape::Rectangle(rect) => paint_rect_selection(rect),
        _ => panic!("unexpected selection type"),
    }
```

Modèles de tuple et de structure

Modèles de tuplecorrespondre à des tuples. Ils sont utiles chaque fois que vous souhaitez impliquer plusieurs éléments de données dans un seul `match` :

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

Structureles modèles utilisent des accolades, tout comme les expressions struct. Ils contiennent un sous-modèle pour chaque champ :

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

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

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

Illustration 10-6. Correspondance de modèles avec des structures

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

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

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

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

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

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

Modèles de tableau et de tranche

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

Par exemple, lors de la conversion des valeurs de couleur de teinte, de saturation et de luminosité (HSL) en valeurs de couleur rouge, vert, bleu (RVB), les couleurs avec une luminosité nulle ou totale sont simplement noires ou blanches. Nous pourrions utiliser une `match` expression pour traiter ces cas simplement.

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

```
            ...
        }
    }
```

Trancheles modèles sont similaires, mais contrairement aux tableaux, les tranches ont des longueurs variables, de sorte que les modèles de tranche correspondent non seulement sur les valeurs mais aussi sur la longueur. `..` dans un modèle de tranche correspond à n'importe quel nombre d'éléments :

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

Modèles de référence

Rouillerles modèles prennent en charge deux fonctionnalités pour travailler avec des références. `ref` les modèles empruntent des parties d'une valeur correspondante. `&` les modèles correspondent aux références. Nous couvrirons `ref` d'abord les modèles.

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

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

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

Si `name` et `language` étaient tous deux des valeurs copiables, Rust copierait les champs au lieu de les déplacer, et ce code irait bien. Mais supposons qu'il s'agisse de l' `String` art. Que pouvons-nous faire?

Nous avons besoin d'une sorte de modèle qui *emprunte* les valeurs correspondantes au lieu de les déplacer. Le `ref` mot-clé fait exactement cela :

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

Désormais, les variables locales `name` et `language` sont des références aux champs correspondants dans `account` . Puisque `account` n'est qu'emprunté, pas consommé, vous pouvez continuer à appeler des méthodes dessus.

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

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

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

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

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

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

La correspondance de motifs se déroule comme illustré à la Figure 10-7 .

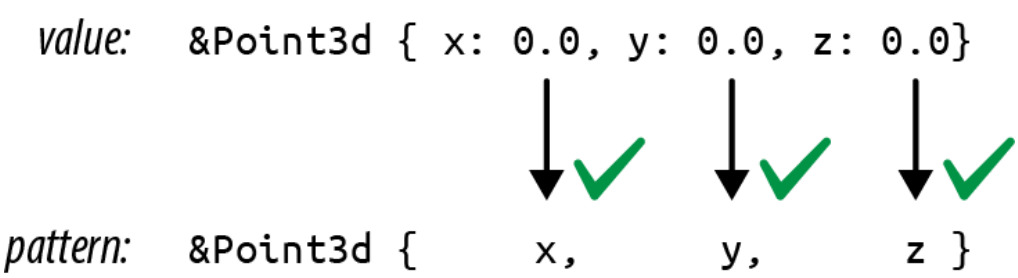

Illustration 10-7. Correspondance de modèle avec des références

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

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

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

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

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

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

Un programme peut utiliser un `&` motif pour obtenir le caractère pointé:

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

Gardes de match

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

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

Cela échoue car les identifiants dans les modèles introduisent de *nouvelles* variables. Le modèle `Some(current_hex)` ici crée une nouvelle variable locale `current_hex`, occultant l' argument `current_hex`. Rust émet plusieurs avertissements à propos de ce code, en particulier le dernier bras du `match` est inaccessible. Une façon de résoudre ce problème consiste simplement à utiliser une `if` expression dans le bras de correspondance :

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

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

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

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

Faire correspondre plusieurs possibilités

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

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

Dans une expression, `|` est l'opérateur OU au niveau du bit, mais ici il fonctionne plus comme le `|` symbole dans une expression régulière. `at_end` est défini sur `true` si `chars.peek()` est `None` , ou a `Some` contenant un retour chariot ou un saut de ligne.

Utilisez `..=` pour faire correspondre toute une plage de valeurs. Les modèles de plage incluent les valeurs de début et de fin, donc `'0' ..= '9'` correspondent à tous les chiffres ASCII :

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

Rust autorise également des modèles de plage comme `x..` , qui correspondent à n'importe quelle valeur `x` jusqu'à la valeur maximale du type. Cependant, les autres variétés de plages exclusives de fin, comme

`0..100` ou `..100`, et les plages illimitées comme `..` ne sont pas encore autorisées dans les modèles.

Reliure avec @ Patterns

Enfin, les matchs `x @ pattern` exactement comme le given `pattern`, mais en cas de succès, au lieu de créer des variables pour des parties de la valeur correspondante, il crée une seule variable `x` et y déplace ou copie la valeur entière. Par exemple, disons que vous avez ce code :

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

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

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

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

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

Où les modèles sont autorisés

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

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

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

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

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

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

Chacun d'entre eux enregistre deux ou trois lignes de code passe-partout. Le même concept existe dans d'autres langages : en JavaScript, cela s'appelle *déstructuration* , tandis qu'en Python, c'est *unpacking* .

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

Un *modèle réfutable*est un qui ne correspond pas, comme `Ok(x)` , qui ne correspond pas à un résultat d'erreur, ou `'0' ..= '9'` , qui ne correspond pas au caractère `'Q'` . Des modèles réfutables peuvent être utilisés dans `match` les armes, car `match` ils sont conçus pour eux : si un modèle ne correspond pas, ce qui se passe ensuite est clair. Les quatre exemples précédents sont des endroits dans les programmes Rust où un modèle peut être pratique, mais le langage ne permet pas l' échec de la correspondance.

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

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

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

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

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

Pour plus de détails sur ces expressions, voir « if let » et « Loops » .

Remplir un arbre binaire

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

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

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

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

```
impl<T: Ord> BinaryTree<T> {
    fn add(&mut self, value: T) {
        match *self {
            BinaryTree:: Empty => {
                *self = BinaryTree:: NonEmpty(Box:: new(TreeNode {
```

```
 6                         element: value,
 7                         left: BinaryTree:: Empty,
 8                         right: BinaryTree:: Empty,
 9                     }))
10                 }
11                 BinaryTree::NonEmpty(ref mut node) => {
12                     if value <= node.element {
13                         node.left.add(value);
14                     } else {
15                         node.right.add(value);
16                     }
17                 }
18             }
19         }
20     }
```

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

Si l'arborescence existante `*self` est vide, c'est le cas le plus simple. Les lignes 5 à 9 s'exécutent, changeant l' `Empty` arbre en `NonEmpty` un. L'appel à `Box::new()` here alloue un nouveau `TreeNode` dans le tas. Lorsque nous avons terminé, l'arbre contient un élément. Ses sous-arborescences gauche et droite sont toutes les deux `Empty` .

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

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

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

La nouvelle méthode peut être utilisée comme ceci :

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

La grande image

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

La programmation est un traitement de données. Mettre les données dans la bonne forme peut faire la différence entre un petit programme rapide et élégant et un enchevêtrement lent et gigantesque de ruban adhésif et d'appels de méthodes virtuelles.

Il s'agit de l'adresse des énumérations d'espace problématique. Ils sont un outil de conception pour mettre les données dans la bonne forme. Pour les cas où une valeur peut être une chose, ou une autre chose, ou peut-être rien du tout, les énumérations sont meilleures que les hiérarchies de classes sur chaque axe : plus rapides, plus sûres, moins de code, plus faciles à documenter.

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

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