Chapitre 7. Gestion des erreurs

> *Je savais que si je restais assez longtemps, quelque chose comme ça arriverait.*
>
> —George Bernard Shaw sur la mort

L'approche de Rust pour la gestion des erreursest suffisamment inhabituel pour justifier un court chapitre sur le sujet. Il n'y a pas d'idées difficiles ici, juste des idées qui pourraient être nouvelles pour vous. Ce chapitre couvre les deux différents types de gestion des erreurs dans Rust : panique et `Result`s.

Les erreurs ordinaires sont gérées à l'aide du `Result` type. `Result`s représentent généralement des problèmes causés par des éléments extérieurs au programme, comme une entrée erronée, une panne de réseau ou un problème d'autorisations. Que de telles situations se produisent ne dépend pas de nous ; même un programme sans bogue les rencontrera de temps en temps. La majeure partie de ce chapitre est consacrée à ce type d'erreur. Nous couvrirons d'abord la panique, car c'est la plus simple des deux.

La panique concerne l'autre type d'erreur, celle qui *ne devrait jamais arriver* .

Panique

Un programme paniquelorsqu'il rencontre quelque chose de tellement confus qu'il doit y avoir un bogue dans le programme lui-même. Quelque chose comme:

- Accès à la baie hors limites
- Division entière par zéro
- Faire appel `.expect()` à un `Result` qui se trouve être `Err`
- Échec de l'assertion

(Il y a aussi la macro `panic!()` , pour les cas où votre propre code découvre qu'il a mal tourné, et vous devez donc déclencher directement une panique. `panic!()` accepte `println!()` arguments facultatifs de style -, pour la construction d'un message d'erreur.)

Ce que ces conditions ont en commun, c'est qu'elles sont toutes, sans trop insister là-dessus, la faute du programmeur. Une bonne règle d'or est : « Ne paniquez pas.

Mais nous faisons tous des erreurs. Lorsque ces erreurs qui ne devraient pas se produire se produisent, que se passe-t-il alors ? Remarquablement, Rust vous donne le choix. Rust peut soit dérouler la pile en cas de panique, soit interrompre le processus. Le déroulement est la valeur par défaut.

Se détendre

Lorsqueles pirates se partagent le butin d'un raid, le capitaine reçoit la moitié du butin. Les membres d'équipage ordinaires gagnent des parts égales de l'autre moitié. (Les pirates détestent les fractions, donc si l'une ou l'autre des divisions n'est pas égale, le résultat est arrondi à l'inférieur, le reste allant au perroquet du navire.)

```
fn pirate_share(total: u64, crew_size: usize) ->u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

Cela peut bien fonctionner pendant des siècles jusqu'au jour où il s'avère que le capitaine est le seul survivant d'un raid. Si nous passons a `crew_size` de zéro à cette fonction, elle divisera par zéro. En C++, ce serait un comportement indéfini. Dans Rust, cela déclenche une panique, qui se déroule généralement comme suit :

- Un message d'erreur est imprimé sur le terminal :

  ```
  thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
  note: Run with `RUST_BACKTRACE=1` for a backtrace.
  ```

  Si vous définissez la `RUST_BACKTRACE` variable d'environnement, comme le suggèrent les messages, Rust videra également la pile à ce stade.
- La pile est déroulée. Cela ressemble beaucoup à la gestion des exceptions C++.
  Toutes les valeurs temporaires, variables locales ou arguments que la fonction actuelle utilisait sont supprimés, dans l'ordre inverse de leur création. Supprimer une valeur signifie simplement nettoyer après elle : tous les `String` s ou `Vec` s que le programme utilisait sont libé-

rés, tous les `File`s ouverts sont fermés, et ainsi de suite. `drop` Les méthodes définies par l'utilisateur sont également appelées ; voir "Déposer" . Dans le cas particulier de `pirate_share()`, il n'y a rien à nettoyer.

Une fois l'appel de fonction en cours nettoyé, nous passons à son appelant, en supprimant ses variables et ses arguments de la même manière. Ensuite, nous passons à l'appelant de *cette* fonction, et ainsi de suite dans la pile.

- Enfin, le fil se termine. Si le thread paniqué était le thread principal, alors tout le processus se termine (avec un code de sortie différent de zéro).

La panique est peut-être un nom trompeur pour ce processus ordonné. Une panique n'est pas un crash. Ce n'est pas un comportement indéfini. C'est plus comme un `RuntimeException` en Java ou un `std::logic_error` en C++. Le comportement est bien défini ; ça ne devrait pas arriver.

La panique est sans danger. Il ne viole aucune des règles de sécurité de Rust ; même si vous parvenez à paniquer au milieu d'une méthode de bibliothèque standard, elle ne laissera jamais un pointeur pendant ou une valeur à moitié initialisée en mémoire. L'idée est que Rust intercepte l'accès invalide au tableau, ou quoi que ce soit, *avant* que quelque chose de mal ne se produise. Il serait dangereux de continuer, donc Rust déroule la pile. Mais le reste du processus peut continuer à s'exécuter.

La panique est par thread. Un thread peut paniquer tandis que d'autres threads poursuivent leurs activités normales. Au chapitre 19 , nous montrerons comment un thread parent peut savoir quand un thread enfant panique et gérer l'erreur avec élégance.

Il existe également un moyen d' *attraper* le déroulement de la pile, permettant au thread de survivre et de continuer à fonctionner. La fonction de bibliothèque standard `std::panic::catch_unwind()` est ce que ca. Nous n'expliquerons pas comment l'utiliser, mais c'est le mécanisme utilisé par le harnais de test de Rust pour récupérer lorsqu'une assertion échoue dans un test. (Cela peut également être nécessaire lors de l'écriture de code Rust pouvant être appelé depuis C ou C++, car le déroulement dans du code non-Rust est un comportement indéfini ; voir le chapitre 22 .)

Idéalement, nous aurions tous un code sans bogue qui ne panique jamais. Mais personne n'est parfait. Vous pouvez utiliser des threads et `catch_unwind()` gérer la panique, ce qui rend votre programme plus ro-

buste. Une mise en garde importante est que ces outils n'attrapent que les paniques qui déroulent la pile. Toutes les paniques ne se déroulent pas de cette façon.

Abandon

Empilerle déroulement est le comportement de panique par défaut, mais il existe deux circonstances dans lesquelles Rust n'essaie pas de dérouler la pile.

Si une `.drop()` méthode déclenche une deuxième panique alors que Rust essaie toujours de nettoyer après la première, cela est considéré comme fatal. Rust arrête de se dérouler et interrompt tout le processus.

De plus, le comportement de panique de Rust est personnalisable. Si vous compilez avec `-C panic=abort`, la *première* panique de votre programme interrompt immédiatement le processus. (Avec cette option, Rust n'a pas besoin de savoir comment dérouler la pile, ce qui peut réduire la taille de votre code compilé.)

Ceci conclut notre discussion sur la panique dans Rust. Il n'y a pas grand-chose à dire, car le code Rust ordinaire n'a aucune obligation de gérer la panique. Même si vous utilisez des threads ou `catch_unwind()`, tout votre code de gestion de panique sera probablement concentré à quelques endroits. Il est déraisonnable de s'attendre à ce que chaque fonction d'un programme anticipe et gère les bogues dans son propre code. Les erreurs causées par d'autres facteurs sont une autre paire de manches.

Résultat

Rouillern'a pas d'exceptions. Au lieu de cela, les fonctions qui peuvent échouer ont un type de retour qui le dit :

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

Le `Result` type indique une panne possible. Lorsque nous appelons la `get_weather()` fonction, elle renverra soit un *résultat de réussite* `Ok(weather)`, où `weather` est une nouvelle `WeatherReport` valeur, soit un *résultat d'erreur* `Err(error_value)`, où `error_value` est une `io::Error` explication de ce qui s'est mal passé.

Rust nous oblige à écrire une sorte de gestion des erreurs chaque fois que nous appelons cette fonction. Nous ne pouvons pas accéder au `WeatherReport` sans faire *quelque chose* au `Result`, et vous recevrez un avertissement du compilateur si une `Result` valeur n'est pas utilisée.

Au chapitre 10, nous verrons comment la bibliothèque standard définit `Result` et comment vous pouvez définir vos propres types similaires. Pour l'instant, nous allons adopter une approche de « livre de recettes » et nous concentrer sur la façon d'utiliser `Result` s pour obtenir le comportement de gestion des erreurs que vous souhaitez. Nous verrons comment intercepter, propager et signaler les erreurs, ainsi que les modèles courants d'organisation et d'utilisation des `Result` types.

Attraper les erreurs

LaLa façon la plus complète de traiter a `Result` est celle que nous avons montrée au chapitre 2 : utiliser une `match` expression.

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

C'est l'équivalent de Rust `try/catch` dans d'autres langages. C'est ce que vous utilisez lorsque vous voulez gérer les erreurs de front, et non les transmettre à votre interlocuteur.

`match` est un peu verbeux, `Result<T, E>` offre donc une variété de méthodes qui sont utiles dans des cas courants particuliers. Chacune de ces méthodes a une `match` expression dans son implémentation. (Pour la liste complète des `Result` méthodes, consultez la documentation en ligne. Les méthodes répertoriées ici sont celles que nous utilisons le plus.)

`result.is_ok()`, `result.is_err()`

Revenirun `bool` dire si `result` est un résultat de succès ou un résultat d'erreur.

`result.ok()`

Retourla valeur de réussite, le cas échéant, en tant que `Option<T>`. Si `result` est un résultat de réussite, cela renvoie `Some(success_value)` ;

sinon, il renvoie `None` , en supprimant la valeur d'erreur.

`result.err()`

Retourla valeur d'erreur, le cas échéant, sous forme de `Option<E>` .

`result.unwrap_or(fallback)`

Retourla valeur de réussite, si `result` est un résultat de réussite. Sinon, elle renvoie `fallback` , en supprimant la valeur d'erreur.

```
// A fairly safe prediction for Southern California.
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);

// Get a real weather report, if possible.
// If not, fall back on the usual.
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);
display_weather(los_angeles, &report);
```

C'est une bonne alternative à `.ok()` parce que le type de retour est `T` , pas `Option<T>` . Bien sûr, cela ne fonctionne que lorsqu'il existe une valeur de repli appropriée.

`result.unwrap_or_else(fallback_fn)`

C'est le même, mais au lieu de transmettre directement une valeur de secours, vous transmettez une fonction ou une fermeture. C'est pour les cas où il serait inutile de calculer une valeur de repli si vous n'allez pas l'utiliser. Le `fallback_fn` n'est appelé que si nous avons un résultat d'erreur.

```
let report =
    get_weather(hometown)
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

(Le chapitre 14 couvre les fermetures en détail.)

`result.unwrap()`

Retourne égalementla valeur de réussite, si `result` est un résultat de réussite. Cependant, s'il `result` y a un résultat d'erreur, cette méthode panique. Cette méthode a ses utilisations ; nous en reparlerons plus tard.

`result.expect(message)`

Cetteidentique à `.unwrap()` , mais vous permet de fournir un message qu'il imprime en cas de panique.

Enfin, les méthodes pour travailler avec des références dans un `Result` :

`result.as_ref()`

Convertitun `Result<T, E>` à un `Result<&T, &E>` .

`result.as_mut()`

Cetteest le même, mais emprunte une référence mutable. Le type de retour est `Result<&mut T, &mut E>` .

L'une des raisons pour lesquelles ces deux dernières méthodes sont utiles est que toutes les autres méthodes répertoriées ici, à l'exception de `.is_ok()` et `.is_err()` , *consomment* le `result` sur lequel elles opèrent. Autrement dit, ils prennent l' `self` argument par valeur. Parfois, il est assez pratique d'accéder aux données à l'intérieur d'un `result` sans les détruire, et c'est ce `.as_ref()` que `.as_mut()` nous faisons pour nous. Par exemple, supposons que vous vouliez appeler `result.ok()` , mais que vous deviez `result` rester intact. Vous pouvez écrire `result.as_ref().ok()` , qui emprunte simplement `result` , renvoyant un `Option<&T>` plutôt queun `Option<T>` .

Alias de type de résultat

quelquefoisvous verrez la documentation de Rust qui semble omettre le type d'erreur de a `Result` :

```
fn remove_file(path: &Path) ->Result<()>
```

Cela signifie qu'un `Result` alias de type est utilisé.

Un alias de type est une sorte de raccourci pour les noms de type. Les modules définissent souvent un `Result` alias de type pour éviter d'avoir à répéter un type d'erreur qui est utilisé de manière cohérente par presque toutes les fonctions du module. Par exemple, le module de la bibliothèque standard `std::io` inclut cette ligne de code :

```
pub type Result<T> = result::Result<T, Error>;
```

Cela définit un type public `std::io::Result<T>` . C'est un alias pour `Result<T, E>` , mais il est codé en dur `std::io::Error` comme type d'erreur. Concrètement, cela signifie que si vous écrivez `use std::io;` , alors Rust comprendra `io::Result<String>` comme un raccourci pour `Result<String, io::Error>` .

Lorsque quelque chose comme `Result<()>` apparaît dans la documentation en ligne, vous pouvez cliquer sur l'identifiant `Result` pour voir quel

alias de type est utilisé et connaître le type d'erreur. En pratique, c'est généralement évident d'après le contexte.

Erreurs d'impression

quelquefoisla seule façon de gérer une erreur est de la vider dans le terminal et de passer à autre chose. Nous avons déjà montré une façon de procéder :

```
println!("error querying the weather: {}", err);
```

La bibliothèque standard définit plusieurs types d'erreurs avec des noms ennuyeux : `std::io::Error`, `std::fmt::Error`, `std::str::Utf8Error`, etc. Tous implémentent une interface commune `std::error::Error`, le trait, ce qui signifie qu'ils partagent les fonctionnalités et méthodes suivantes :

`println!()`

Toutes les erreursles types sont imprimables en utilisant ceci. L'impression d'une erreur avec le `{}` spécificateur de format n'affiche généralement qu'un bref message d'erreur. Vous pouvez également imprimer avec le `{:?}` spécificateur de format pour obtenir une `Debug` vue de l'erreur. Ceci est moins convivial, mais comprend des informations techniques supplémentaires.

```
// result of `println!("error: {}", err);`
error: failed to look up address information: No address associated w
hostname

// result of `println!("error: {:?}", err);`
error: Error { repr: Custom(Custom { kind: Other, error: StringError(
"failed to look up address information: No address associated with
hostname") }) }
```

`err.to_string()`

Retourun message d'erreur sous forme de `String`.

`err.source()`

Retourune `Option` de l'erreur sous-jacente, le cas échéant, qui a causé `err`. Par exemple, une erreur de réseau peut entraîner l'échec d'une transaction bancaire, ce qui pourrait entraîner la reprise de possession de votre bateau. Si `err.to_string()` est `"boat was repossessed"`, alors `err.source()` peut renvoyer une erreur concernant la transaction ayant

> échoué. Cette erreur `.to_string()` peut être , et il peut s'agir d'un avec des détails sur la panne de réseau spécifique qui a causé tout ce remue-ménage. Cette troisième erreur est la cause première, donc sa méthode renverrait . Étant donné que la bibliothèque standard ne comprend que des fonctionnalités plutôt de bas niveau, la source des erreurs renvoyées par la bibliothèque standard est généralement . `"failed to transfer $300 to United Yacht Supply" .source() io::Error .source() None None`

L'impression d'une valeur d'erreur n'imprime pas également sa source. Si vous voulez être sûr d'imprimer toutes les informations disponibles, utilisez cette fonction :

```
use std:: error:: Error;
use std:: io::{Write, stderr};

/// Dump an error message to `stderr`.
///
/// If another error happens while building the error message or
/// writing to `stderr`, it is ignored.
fn print_error(mut err:&dyn Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {}", source);
        err = source;
    }
}
```

La `writeln!` macrofonctionne comme `println!` , sauf qu'il écrit les données dans un flux de votre choix. Ici, nous écrivons les messages d'erreur dans le flux d'erreur standard, `std::io::stderr` . Nous pourrions utiliser la `eprintln!` macro pour faire la même chose, mais `eprintln!` panique si une erreur se produit. Dans `print_error` , nous voulons ignorer les erreurs qui surviennent lors de l'écriture du message ; nous expliquons pourquoi dans « Ignorer les erreurs » , plus loin dans le chapitre.

Les types d'erreur de la bibliothèque standard n'incluent pas de trace de pile, mais le `anyhow` crate populaire fournit un type d'erreur prêt à l'emploi qui le fait, lorsqu'il est utilisé avec une version instable du compilateur Rust. (A partir de Rust 1.56, les fonctions de la bibliothèque standard pour capturer les backtraces n'étaient pas encore stabilisées.)

Propagation des erreurs

Dans la plupart des endroitsoù nous essayons quelque chose qui pourrait échouer, nous ne voulons pas attraper et gérer l'erreur immédiatement. C'est tout simplement trop de code pour utiliser une `match` instruction de 10 lignes à chaque endroit où quelque chose pourrait mal tourner.

Au lieu de cela, si une erreur se produit, nous voulons généralement laisser notre appelant s'en occuper. Nous voulons que les erreurs se *propagent* dans la pile des appels.

Rust a un `?` opérateurça fait ça. Vous pouvez ajouter un `?` à toute expression qui produit un `Result` , comme le résultat d'un appel de fonction :

```
let weather = get_weather(hometown)?;
```

Le comportement de `?` dépend du fait que cette fonction renvoie un résultat de réussite ou un résultat d'erreur :

- En cas de succès, il déballe le `Result` pour obtenir la valeur de succès à l'intérieur. Le type d' `weather` ici n'est pas `Result<WeatherReport, io::Error>` mais simplement `WeatherReport` .
- En cas d'erreur, il revient immédiatement de la fonction englobante, transmettant le résultat de l'erreur dans la chaîne d'appel. Pour s'assurer que cela fonctionne, `?` ne peut être utilisé que sur `Result` des fonctions in qui ont un `Result` type de retour.

`?` L' opérateur n'a rien de magique . Vous pouvez exprimer la même chose en utilisant une `match` expression, bien qu'elle soit beaucoup plus verbeuse :

```
let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err)
};
```

Les seules différences entre ceci et l' `?` opérateur sont quelques petits détails concernant les types et les conversions. Nous couvrirons ces détails dans la section suivante.

Dans le code plus ancien, vous pouvez voir la `try!()` macro, qui était la manière habituelle de propager les erreurs jusqu'à ce que l' `?` opérateur soit introduit dans Rust 1.13 :

```
let weather = try!(get_weather(hometown));
```

La macro se développe en une `match` expression, comme la précédente.

Il est facile d'oublier à quel point la possibilité d'erreurs est omniprésente dans un programme, en particulier dans le code qui s'interface avec le système d'exploitation. L' `?` opérateur apparaît parfois sur presque toutes les lignes d'une fonction :

```
use std:: fs;
use std:: io;
use std:: path::Path;

fn move_all(src: &Path, dst: &Path) -> io:: Result<()> {
    for entry_result in src.read_dir()? {  // opening dir could fail
        let entry = entry_result?;         // reading dir could fail
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?;  // renaming could fail
    }
    Ok(())  // phew!
}
```

`?` fonctionne également de la même manière avec le `Option` type. Dans une fonction qui retourne `Option` , vous pouvez utiliser `?` pour déballer une valeur et revenir tôt dans le cas de `None` :

```
let weather = get_weather(hometown).ok()?;
```

Travailler avec plusieurs types d'erreurs

Souvent, plusqu'une chose pourrait mal tourner.Supposons que nous lisons simplement des nombres à partir d'un fichier texte :

```
use std:: io::{self, BufRead};

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;          // reading lines can fail
        numbers.push(line.parse()?);      // parsing integers can fail
    }
    Ok(numbers)
}
```

Rust nous renvoie une erreur de compilation :

```
error: `?` couldn't convert the error to `std::io::Error`

  numbers.push(line.parse()?);     // parsing integers can fail
                              ^
              the trait `std::convert::From<std::num::ParseIntError>`
              is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait
```

Les termes de ce message d'erreur auront plus de sens lorsque nous atteindrons le chapitre 11 , qui couvre les traits. Pour l'instant, notez simplement que Rust se plaint que l' `?` opérateur ne peut pas convertir une `std::num::ParseIntError` valeur en type `std::io::Error` .

Le problème ici est que la lecture d'une ligne d'un fichier et l'analyse d'un entier produisent deux types d'erreurs potentiels différents. Le type de `line_result` est `Result<String, std::io::Error>` . Le type de `line.parse()` est `Result<i64, std::num::ParseIntError>` . Le type de retour de notre `read_numbers()` fonction n'accepte que `io::Error` s. Rust essaie de gérer le `ParseIntError` en le convertissant en un `io::Error` , mais il n'y a pas une telle conversion, nous obtenons donc une erreur de type.

Il y a plusieurs façons de traiter cela. Par exemple, le `image` crate que nous avons utilisé au chapitre 2 pour créer les fichiers image de l'ensemble de Mandelbrot définit son propre type d'erreur, `ImageError` , et implémente les conversions de `io::Error` et plusieurs autres types d'erreur vers `ImageError` . Si vous souhaitez emprunter cette voie, essayez la `thiserror` caisse, qui est conçue pour vous aider à définir de bons types d'erreurs avec seulement quelques lignes de code.

Une approche plus simple consiste à utiliser ce qui est intégré à Rust. Tous les types d'erreur de bibliothèque standard peuvent être convertis en type `Box<dyn std::error::Error + Send + Sync + 'static>` .C'est un peu long, mais `dyn std::error::Error` cela représente "n'importe quelle erreur" et `Send + Sync + 'static` permet de passer en toute sécurité entre les threads, ce que vous voudrez souvent. [1] Pour plus de commodité, vous pouvez définir des alias de type :

```
type GenericError = Box<dyn std:: error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Ensuite, changez le type de retour de `read_numbers()` en `GenericResult<Vec<i64>>`. Avec ce changement, la fonction compile. L'`?` opérateur convertit automatiquement l'un ou l'autre type d'erreur en a `GenericError` selon les besoins.

Incidemment, l'`?` opérateur effectue cette conversion automatique en utilisant une méthode standard que vous pouvez utiliser vous-même. Pour convertir toute erreur en `GenericError` type, appelez `GenericError::from()` :

```
let io_error = io:: Error:: new(          // make our own io::Error
    io:: ErrorKind:: Other, "timed out");
return Err(GenericError::from(io_error));  // manually convert to GenericE
```

Nous couvrirons entièrement le `From` trait et sa `from()` méthode au chapitre 13 .

L'inconvénient de l'`GenericError` approche est que le type de retour ne communique plus précisément à quels types d'erreurs l'appelant peut s'attendre. L'appelant doit être prêt à tout.

Si vous appelez une fonction qui renvoie a `GenericResult` et que vous souhaitez gérer un type particulier d'erreur mais laisser toutes les autres se propager, utilisez la méthode générique `error.downcast_ref::<ErrorType>()` . Il emprunte une référence à l'erreur, *s'il* s'agit du type particulier d'erreur que vous recherchez :

```
loop {
    match compile_project() {
        Ok(()) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>()
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue;  // try again!
            }
            return Err(err);
        }
    }
}
```

De nombreux langages ont une syntaxe intégrée pour ce faire, mais cela s'avère rarement nécessaire. Rust a une méthode pour cela à la place.

Traiter les erreurs qui « ne peuvent pas se produire »

quelquefoisnous *savons* juste qu'une erreur ne peut pas arriver. Par exemple, supposons que nous écrivions du code pour analyser un fichier de configuration et qu'à un moment donné, nous découvrions que la prochaine chose dans le fichier est une chaîne de chiffres :

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
```

Nous voulons convertir cette chaîne de chiffres en un nombre réel. Il existe une méthode standard qui fait cela:

```
let num = digits.parse::<u64>();
```

Maintenant le problème : la `str.parse::<u64>()` méthode ne renvoie pas un `u64` . Il renvoie un `Result` . Cela peut échouer, car certaines chaînes ne sont pas numériques :

```
"bleen".parse::<u64>()  // ParseIntError: invalid digit
```

Mais il se trouve que nous savons que dans ce cas, `digits` se compose entièrement de chiffres. Que devrions nous faire?

Si le code que nous écrivons renvoie déjà un `GenericResult` , nous pouvons ajouter un `?` et l'oublier. Sinon, nous sommes confrontés à la perspective irritante de devoir écrire du code de gestion des erreurs pour une erreur qui ne peut pas se produire. Le meilleur choix serait alors d'utiliser `.unwrap()` , une `Result` méthode qui panique si le résultat est un `Err` , mais renvoie simplement la valeur de succès d'un `Ok` :

```
let num = digits.parse::<u64>().unwrap();
```

C'est exactement comme `?` sauf que si nous nous trompons sur cette erreur, si cela *peut* arriver, alors dans ce cas, nous paniquerions.

En fait, nous nous trompons sur ce cas particulier. Si l'entrée contient une chaîne de chiffres suffisamment longue, le nombre sera trop grand pour tenir dans un `u64` :

```
"99999999999999999999".parse::<u64>()    // overflow error
```

L'utiliser `.unwrap()` dans ce cas particulier serait donc un bug. Une fausse entrée ne devrait pas provoquer de panique.

Cela dit, des situations surviennent où une `Result` valeur ne peut vraiment pas être une erreur. Par exemple, au chapitre 18 , vous verrez que le `Write` trait définit un ensemble commun de méthodes (`.write()` et d'autres) pour le texte et la sortie binaire. Toutes ces méthodes renvoient `io::Result` s, mais si vous écrivez dans un `Vec<u8>` , elles ne peuvent pas échouer. Dans de tels cas, il est acceptable d'utiliser `.unwrap()` ou `.expect(message)` de renoncer à l' `Result` art.

Ces méthodes sont également utiles lorsqu'une erreur indique une condition si grave ou bizarre que la panique est exactement la façon dont vous voulez la gérer :

```
fn print_file_age(filename: &Path, last_modified:SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
    ...
}
```

Ici, la `.elapsed()` méthodene peut échouer que si l'heure système est *antérieure* à la date de création du fichier. Cela peut se produire si le fichier a été créé récemment et que l'horloge système a été ajustée à l'envers pendant l'exécution de notre programme. Selon la façon dont ce code est utilisé, il est raisonnable de paniquer dans ce cas, plutôt que de gérer l'erreur ou de la propager à l'appelant.

Ignorer les erreurs

Parfois, nous voulons juste ignorerune erreur en somme. Par exemple, dans notre `print_error()` fonction, nous avons dû gérer la situation improbable où l'impression de l'erreur déclenche une autre erreur. Cela peut arriver, par exemple, si `stderr` est redirigé vers un autre processus et que ce processus est tué. L'erreur d'origine que nous essayions de signaler est probablement plus importante à propager, nous voulons donc simplement ignorer les problèmes avec `stderr` , mais le compilateur Rust avertit des `Result` valeurs inutilisées :

```
    writeln!(stderr(), "error: {}", err);  // warning: unused result
```

L'idiome `let _ = ...` est utilisé pour faire taire cet avertissement :

```
    let _ = writeln!(stderr(), "error: {}", err);  // ok, ignore result
```

Gestion des erreurs dans main()

Dans la plupart des endroitsoù a `Result` est produit, laisser l'erreur remonter jusqu'à l'appelant est le bon comportement. C'est pourquoi `?` est un seul personnage dans Rust. Comme nous l'avons vu, dans certains programmes, il est utilisé sur plusieurs lignes de code à la suite.

Mais si vous propagez une erreur assez longtemps, elle finit par atteindre `main()`, et quelque chose doit être fait avec. Normalement, `main()` ne peut pas utiliser `?` car son type de retour n'est pas `Result` :

```
fn main() {
    calculate_tides()?;  // error: can't pass the buck any further
}
```

La manière la plus simple de gérer les erreursen `main()` est d'utiliser `.expect()` :

```
fn main() {
    calculate_tides().expect("error");  // the buck stops here
}
```

Si `calculate_tides()` revientun résultat d'erreur, la `.expect()` méthode panique. Paniquer dans le thread principal imprime un message d'erreur, puis se termine avec un code de sortie différent de zéro, ce qui correspond à peu près au comportement souhaité. Nous l'utilisons tout le temps pour de petits programmes. C'est un début.

Le message d'erreur est un peu intimidant, cependant :

```
$tidecalc --planète mercure
thread 'main' panicked at 'error: "moon not found"', src/main.rs:2:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtr
```

Le message d'erreur se perd dans le bruit. Aussi, `RUST_BACKTRACE=1` est un mauvais conseil dans ce cas particulier.

Cependant, vous pouvez également modifier la signature de type de `main()` pour renvoyer un `Result` type, vous pouvez donc utiliser `?` :

```
fn main() ->Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

Cela fonctionne pour tout type d'erreur qui peut être imprimé avec le `{:?}` formateur, ce que peuvent être tous les types d'erreur standard, comme `std::io::Error` . Cette technique est facile à utiliser et donne un message d'erreur un peu plus agréable, mais ce n'est pas idéal :

```
$tidecalc --planète mercure
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

Si vous avez des types d'erreurs plus complexes ou si vous souhaitez inclure plus de détails dans votre message, il est préférable d'imprimer vous-même le message d'erreur :

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std:: process::exit(1);
    }
}
```

Ce code utilise une `if let` expression pour imprimer le message d'erreur uniquement si l'appel à `calculate_tides()` renvoie un résultat d'erreur. Pour plus de détails sur `if let` les expressions, voir Chapitre 10 . La `print_error` fonction est répertoriée dans "Erreurs d'impression" .

Maintenant, la sortie est belle et bien rangée :

```
$tidecalc --planète mercure
error: moon not found
```

Déclarer un type d'erreur personnalisé

Supposervous écrivez un nouvel analyseur JSON et vous souhaitez qu'il ait son propre type d'erreur. (Nous n'avons pas encore couvert les types

définis par l'utilisateur ; cela arrivera dans quelques chapitres. Mais les types d'erreur sont pratiques, nous allons donc inclure un petit aperçu ici.)

Approximativement, le code minimum que vous écririez est :

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column:usize,
}
```

Cette structure s'appellera `json::error::JsonError`, et lorsque vous voudrez lever une erreur de ce type, vous pourrez écrire :

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column:current_column
});
```

Cela fonctionnera bien. Cependant, si vous voulez que votre type d'erreur fonctionne comme les types d'erreur standard, comme les utilisateurs de votre bibliothèque s'y attendent, alors vous avez un peu plus de travail à faire :

```
use std::fmt;

// Errors should be printable.
impl fmt:: Display for JsonError {
    fn fmt(&self, f: &mut fmt:: Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// Errors should implement the std::error::Error trait,
// but the default definitions for the Error methods are fine.
impl std:: error::Error for JsonError { }
```

Encore une fois, la signification du mot-`impl` clé, `self`, et tout le reste seront expliqués dans les prochains chapitres.

Comme pour de nombreux aspects du langage Rust, les caisses existent pour rendre la gestion des erreurs beaucoup plus facile et plus concise. Il en existe une grande variété, mais l'un des plus utilisés est `thiserror`, qui fait tout le travail précédent pour vous, vous permettant d'écrire des erreurs comme celle-ci :

```
use thiserror:: Error;
#[derive(Error, Debug)]
#[error("{message:} ({line:}, {column})")]
pub struct JsonError {
    message: String,
    line: usize,
    column:usize,
}
```

La `#[derive(Error)]` directive indique `thiserror` de générer le code présenté précédemment, ce qui peut économiser beaucoup de temps et d'efforts.

Pourquoi Résultats ?

Maintenant, nous en savons assez pour comprendre ce que Rustest en train de choisir `Result` s plutôt que des exceptions. Voici les points clés de la conception :

- Rust oblige le programmeur à prendre une sorte de décision et à l'enregistrer dans le code, à chaque point où une erreur pourrait se produire. C'est une bonne chose car sinon, il est facile de se tromper dans la gestion des erreurs par négligence.
- La décision la plus courante est de permettre aux erreurs de se propager, et c'est écrit avec un seul caractère, `?`. Ainsi, la plomberie d'erreur n'encombre pas votre code comme elle le fait en C and Go. Pourtant, il est toujours visible : vous pouvez regarder un morceau de code et voir d'un coup d'œil tous les endroits où les erreurs se propagent.
- Étant donné que la possibilité d'erreurs fait partie du type de retour de chaque fonction, il est clair quelles fonctions peuvent échouer et lesquelles ne le peuvent pas. Si vous modifiez une fonction pour qu'elle soit faillible, vous modifiez son type de retour, de sorte que le compilateur vous obligera à mettre à jour les utilisateurs en aval de cette fonction.
- Rust vérifie que `Result` les valeurs sont utilisées, vous ne pouvez donc pas laisser passer accidentellement une erreur en silence (une erreur courante en C).

- Puisqu'il `Result` s'agit d'un type de données comme les autres, il est facile de stocker les résultats de réussite et d'erreur dans la même collection. Cela facilite la modélisation d'un succès partiel. Par exemple, si vous écrivez un programme qui charge des millions d'enregistrements à partir d'un fichier texte et que vous avez besoin d'un moyen de faire face au résultat probable que la plupart réussiront, mais que certains échoueront, vous pouvez représenter cette situation en mémoire à l'aide d'un vecteur de `Result` l'art.

Le coût est que vous vous retrouverez à penser et à gérer les erreurs d'ingénierie plus dans Rust que vous ne le feriez dans d'autres langages. Comme dans de nombreux autres domaines, la gestion des erreurs de Rust est un peu plus stricte que ce à quoi vous êtes habitué. Pour la programmation système, cela vaut la peinece.

1 Vous devriez également envisager d'utiliser la caisse populaire `anyhow`, qui fournit des types d'erreur et de résultat très similaires à nos `GenericError` et `GenericResult`, mais avec quelques fonctionnalités supplémentaires intéressantes.