# 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 en matière de gestion des erreurs est suffisamment inhabituelle 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 s. `Result`

Les erreurs ordinaires sont gérées à l'aide du type. représentent généralement des problèmes causés par des éléments extérieurs au programme, tels qu'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 bug les rencontrera de temps en temps. La majeure partie de ce chapitre est consacrée à ce genre d'erreur. Nous allons d'abord couvrir la panique, cependant, parce que c'est le plus simple des deux. `Result Result`

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

# Panique

Un programme panique lorsqu'il rencontre quelque chose de si gâché qu'il doit y avoir un bogue dans le programme lui-même. Quelque chose comme :

- Accès au tableau hors limites
- Division des entiers par zéro
- Faire appel à un qui se trouve être `.expect() Result Err`
- Échec de l'assertion

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

Ce que ces conditions ont en commun, c'est qu'elles sont toutes, pour ne pas mettre un point trop fin, la faute du programmeur. Une bonne règle de base 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. La rouille peut soit dérouler la pile lorsqu'une panique se produit, soit interrompre le processus. Le dénouement est la valeur par défaut.

## Déroulement

Lorsque les pirates séparent le butin d'un raid, le capitaine obtient 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 division ne sort pas même, le résultat est arrondi vers le bas, 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 un de zéro à cette fonction, elle sera divisée par zéro. En C++, il s'agit d'un comportement non défini. Dans Rust, cela déclenche une panique, qui se déroule généralement comme suit: `crew_size`

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

  ```
  thread 'main' panicked at 'attempt to divide by zero', pirates.rs:378
  note: Run with `RUST_BACKTRACE=1` for a backtrace.
  ```

  Si vous définissez la variable d'environnement, comme le suggèrent les messages, Rust videra également la pile à ce stade. `RUST_BACKTRACE`
- La pile est déroulée. Cela ressemble beaucoup à la gestion des exceptions C ++.
  Toutes les valeurs temporaires, variables locales ou arguments utilisés par la fonction actuelle sont supprimés, à l'inverse de l'ordre dans lequel ils ont été créés. Laisser tomber une valeur signifie simplement nettoyer après elle: tous les s ou s que le programme utilisait sont libérés, tous les s ouverts sont fermés, etc. Les méthodes définies par

l'utilisateur sont également appelées ; voir « Déposer ». Dans le cas particulier de , il n'y a rien à nettoyer. `String Vec File drop pirate_share()`
Une fois l'appel de fonction actuel nettoyé, nous passons à son appelant, en déposant ses variables et ses arguments de la même manière. Ensuite, nous passons à l'appelant de *cette* fonction, et ainsi de suite en haut de la pile.

- Enfin, le thread se ferme. Si le thread de panique était le thread principal, alors l'ensemble du processus se ferme (avec un code de sortie non nul).

Peut-être que *la panique* est 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 en Java ou un en C ++. Le comportement est bien défini ; cela ne devrait tout simplement pas se produire. `RuntimeException std::logic_error`

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 attrape l'accès au tableau non valide, ou quoi que ce soit, *avant* que quelque chose de mauvais ne se produise. Il serait dangereux de continuer, alors Rust déroule la pile. Mais le reste du processus peut continuer à fonctionner.

La panique est par fil. Un fil peut paniquer tandis que d'autres fils de discussion parlent de leurs activités normales. Dans le 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 fil de survivre et de continuer à fonctionner. C'est ce que fait la fonction de bibliothèque standard. Nous ne verrons 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 qui peut être appelé à partir de C ou C ++, car le déroulement sur du code non-Rust est un comportement non défini; voir le chapitre 22.) `std::panic::catch_unwind()`

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

### Abandon

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

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

En outre, le comportement de panique de Rust est personnalisable. Si vous compilez avec , la *première* panique dans 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.) `-C panic=abort`

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 , 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 marmite de poisson. `catch_unwind()`

## Résultat

La rouille n'a pas d'exceptions. Au lieu de cela, les fonctions qui peuvent échouer ont un type de retour qui dit ceci :

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

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

Rust nous oblige à écrire une sorte de gestion des erreurs chaque fois que nous appelons cette fonction. Nous ne pouvons pas aller à la sans faire

*quelque chose* à la , et vous obtiendrez un avertissement du compilateur si une valeur n'est pas utilisée. `WeatherReport Result Result`

Dans le chapitre 10, nous verrons comment la bibliothèque standard définit 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 s pour obtenir le comportement de gestion des erreurs que vous souhaitez. Nous verrons comment détecter, propager et signaler les erreurs, ainsi que les modèles courants d'organisation et de travail avec les types. `Result Result Result`

## Détection des erreurs

La façon la plus complète de traiter a est la façon dont nous l'avons montré au chapitre 2 : utiliser une expression. `Result match`

```
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 dans d'autres langues. C'est ce que vous utilisez lorsque vous souhaitez gérer les erreurs de front, et non les transmettre à votre appelant. `try/catch`

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

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

Renvoyer un indiquant s'il s'agit d'un résultat de réussite ou d'un résultat d'erreur. `bool result`

*`result.ok()`*

Renvoie la valeur de réussite, le cas échéant, sous la forme d'un fichier . Si est un résultat réussi, cela renvoie ; sinon, il renvoie , en ignorant la valeur d'erreur. `Option<T> result Some(success_value) None`

*`result.err()`*

Renvoie la valeur d'erreur, le cas échéant, sous la forme d'un fichier `Option<E>`.

*`result.unwrap_or(fallback)`*

Renvoie la valeur de réussite, si est un résultat de réussite. Sinon, il renvoie , en ignorant la valeur d'erreur. `result fallback`

```
// 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 car le type de retour est , pas . Bien sûr, cela ne fonctionne que lorsqu'il existe une valeur de secours appropriée. `.ok() T Option<T>`

*`result.unwrap_or_else(fallback_fn)`*

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

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

(Le chapitre 14 traite en détail des fermetures.)

*`result.unwrap()`*

Renvoie également la valeur de réussite, si est un résultat de réussite. Cependant, s'il s'agit d'un résultat d'erreur, cette méthode panique. Cette méthode a ses utilités; nous en reparlerons plus tard. `result result`

*`result.expect(message)`*

C'est la même chose que , mais vous permet de fournir un message qu'il imprime en cas de panique. `.unwrap()`

Enfin, les méthodes de travail avec les références dans un : `Result`

*result.as_ref()*

Convertit a en fichier . `Result<T, E> Result<&T, &E>`

*result.as_mut()*

C'est la même chose, 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 énumérées ici, à l'exception et , *consomment* le sur lequel elles fonctionnent. C'est-à-dire qu'ils prennent l'argument par valeur. Parfois, il est très pratique d'accéder aux données à l'intérieur d'un sans les détruire, et c'est ce que nous faisons. Par exemple, supposons que vous souhaitiez appeler , mais que vous devez être laissé intact. Vous pouvez écrire , qui emprunte simplement , en renvoyant un plutôt qu'un `..is_ok() .is_err() result self result .as_ref() .as_mut() result.ok() result result.as_ref().ok() result Option<&T> Option<T>`

## Alias de type de résultat

Parfois, vous verrez la documentation Rust qui semble omettre le type d'erreur d'un : `Result`

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

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

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

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

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

Lorsque quelque chose comme apparaît dans la documentation en ligne, vous pouvez cliquer sur l'identifiant pour voir quel alias de type est utilisé et apprendre le type d'erreur. En pratique, c'est généralement évident d'après le contexte. `Result<()> Result`

## Erreurs d'impression

Parfois, la seule façon de gérer une erreur est de la déverser sur le terminal et de passer à autre chose. Nous avons déjà montré une façon de le faire:

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

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

*`println!()`*

Tous les types d'erreur sont imprimables à l'aide de ce document. 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 vue de l'erreur. Ceci est moins convivial, mais comprend des informations techniques supplémentaires. `{} {:?} Debug`

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

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

*`err.to_string()`*

Renvoie un message d'erreur sous la forme d'un fichier . `String`

*`err.source()`*

Renvoie une des erreurs sous-jacentes, le cas échéant, qui ont provoqué . Par exemple, une erreur de mise en réseau peut entraîner l'échec d'une transaction bancaire, ce qui peut entraîner la reprise de possession de votre bateau. Si est ,

peut renvoyer une erreur sur la transaction ayant échoué. Cette erreur peut être , et il peut s'agir de détails sur la panne de réseau spécifique qui a causé tout le tapage. Cette troisième erreur est la cause première, de sorte que sa méthode renverrait . Étant donné que la bibliothèque standard n'inclut que des fonctionnalités de niveau plutôt bas, la source des erreurs renvoyées par la bibliothèque standard est généralement

```
. Option err err.to_string() "boat was
repossessed" err.source() .to_string() "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 macro fonctionne comme , sauf qu'elle écrit les données dans un flux de votre choix. Ici, nous écrivons les messages d'erreur dans le flux d'erreur standard, . Nous pourrions utiliser la macro pour faire la même chose, mais panique si une erreur se produit. Dans , 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. `writeln! println! std::io::stderr eprintln! eprintln! print_error`

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

## Propagation des erreurs

Dans la plupart des endroits où 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 instruction de 10 lignes à chaque endroit où quelque chose pourrait mal tourner. `match`

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* vers le haut de la pile d'appels.

Rust a un opérateur qui fait cela. Vous pouvez ajouter un à n'importe quelle expression qui produit un , tel que le résultat d'un appel de fonction : `? ? Result`

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

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

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

Il n'y a rien de magique chez l'opérateur. Vous pouvez exprimer la même chose en utilisant une expression, bien que ce soit beaucoup plus verbeux: `? match`

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

Les seules différences entre celui-ci et l'opérateur sont quelques points fins impliquant des types et des conversions. Nous couvrirons ces détails dans la section suivante. `?`

Dans le code plus ancien, vous pouvez voir la macro, qui était le moyen habituel de propager les erreurs jusqu'à ce que l'opérateur soit introduit

dans Rust 1.13 : `try!()` `?`

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

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

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 type. Dans une fonction qui renvoie , vous pouvez utiliser pour décompresser une valeur et la renvoyer tôt dans le cas de : `Option Option ? None`

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

## Utilisation de plusieurs types d'erreurs

Souvent, plus d'une chose pourrait mal tourner. Supposons que nous lisions 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 donne une erreur de compilateur:

```
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 conversio
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 valeur en type . `? std::num::ParseIntError 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 potentielles différents. Le type de est . Le type de est . Le type de retour de notre fonction ne s'adapte qu'à s. Rust essaie de faire face à la en le convertissant en un , mais il n'y a pas une telle conversion, donc nous obtenons une erreur de type. `line_result Result<String, std::io::Error> line.parse() Result<i64, std::num::ParseIntError> read_numbers() io::Error ParseIntError io::Error`

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

Une approche plus simple consiste à utiliser ce qui est intégré dans Rust. Tous les types d'erreur de bibliothèque standard peuvent être convertis en type . C'est un peu une bouchée, mais représente « toute erreur » et permet de passer en toute sécurité entre les fils, ce que vous voudrez souvent. `Box<dyn std::error::Error + Send + Sync + 'static> dyn`

`std::error::Error Send + Sync + 'static`[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, remplacez le type de retour par . Avec ce changement, la fonction se compile. L'opérateur convertit automatiquement l'un ou l'autre type d'erreur en un au besoin. `read_numbers() GenericResult<Vec<i64>> ? GenericError`

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

```
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 Gener
```

Nous couvrirons le trait et sa méthode en détail dans le chapitre 13. `From from()`

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

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

```
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 il s'avère que cela est rarement nécessaire. Rust a une méthode pour cela à la place.

## Faire face aux erreurs qui « ne peuvent pas se produire »

Parfois, nous *savons* simplement qu'une erreur ne peut pas se produire. Par exemple, supposons que nous écrivions du code pour analyser un fichier de configuration et qu'à un moment donné, nous constations 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 ceci:

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

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

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

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

Si le code que nous écrivons renvoie déjà un , nous pouvons taper sur 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 , une méthode qui panique si le résultat est un , mais renvoie simplement la valeur de succès d'un : `GenericResult ? .unwrap() Result Err Ok`

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

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

En fait, nous nous trompons dans 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'utilisation dans ce cas particulier serait donc un bug. Une fausse entrée ne devrait pas provoquer de panique. `.unwrap()`

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

Ces méthodes sont également utiles lorsqu'une erreur indiquerait 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 méthode ne peut échouer que si l'heure système est *antérieure à* celle de la 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, c'est un appel raisonnable à la panique dans ce cas, plutôt que de gérer l'erreur ou de la propager à l'appelant. `.elapsed()`

## Ignorer les erreurs

Parfois, nous voulons simplement ignorer complètement une erreur. Par exemple, dans notre fonction, nous avons dû gérer la situation improbable où l'impression de l'erreur déclenche une autre erreur. Cela pourrait

se produire, par exemple, si est canalisé 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 , mais le compilateur Rust met en garde contre les valeurs inutilisées: `print_error() stderr stderr Result`

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

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

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

## Gestion des erreurs dans main()

Dans la plupart des endroits où un est produit, laisser la bulle d'erreur jusqu'à l'appelant est le bon comportement. C'est pourquoi il s'agit d'un seul personnage dans Rust. Comme nous l'avons vu, dans certains programmes, il est utilisé sur de nombreuses lignes de code d'affilée. `Result ?`

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

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

Le moyen le plus simple de gérer les erreurs est d'utiliser : `main() .expect()`

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

Si renvoie un résultat d'erreur, la méthode panique. Paniquer dans le thread principal imprime un message d'erreur, puis se ferme avec un code de sortie non nul, ce qui est à peu près le comportement souhaité. Nous l'utilisons tout le temps pour de minuscules programmes. C'est un début. `calculate_tides() .expect()`

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

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

Le message d'erreur est perdu dans le bruit. En outre, est un mauvais conseil dans ce cas particulier. `RUST_BACKTRACE=1`

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

```
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 tous les types d'erreur standard, comme , peuvent être. Cette technique est facile à utiliser et donne un message d'erreur un peu plus agréable, mais ce n'est pas idéal: `{:?} std::io::Error`

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

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

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

Ce code utilise une expression pour imprimer le message d'erreur uniquement si l'appel renvoie un résultat d'erreur. Pour plus d'informations sur les expressions, reportez-vous au chapitre 10. La fonction est répertoriée dans « Erreurs d'impression ». `if let calculate_tides() if let print_error`

Maintenant, la sortie est belle et bien rangée:

```
$ tidecalc --planet mercury
error: moon not found
```

## Déclaration d'un type d'erreur personnalisé

Supposons que vous écrivez un nouvel analyseur JSON et que vous souhaitez qu'il ait son propre type d'erreur. (Nous n'avons pas encore couvert les types définis par l'utilisateur; cela vient dans quelques chapitres. Mais les types d'erreur sont pratiques, nous allons donc inclure un 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 sera appelée , et lorsque vous souhaitez déclencher une erreur de ce type, vous pouvez écrire : `json::error::JsonError`

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

Cela fonctionnera bien. Toutefois, si vous souhaitez que votre type d'erreur fonctionne comme les types d'erreur standard, comme les utilisateurs de votre bibliothèque s'y attendent, vous avez encore un peu 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-clé, , et tout le reste seront expliqués dans les prochains chapitres. `impl self`

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

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

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

## Pourquoi les résultats?

Maintenant, nous en savons assez pour comprendre ce que Rust veut dire en choisissant s plutôt que des exceptions. Voici les points clés de la conception: `Result`

- Rust exige que le programmeur prenne une sorte de décision et l'enregistre dans le code, à chaque point où une erreur pourrait se produire. C'est bien parce que sinon, il est facile de se tromper de 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 en 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 les valeurs sont utilisées, de sorte que vous ne pouvez pas laisser accidentellement une erreur passer silencieusement (une erreur courante en C). `Result`
- Comme il s'agit d'un type de données comme un autre, il est facile de stocker les résultats de réussite et d'erreur dans la même collection. Cela facilite la modélisation du 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 s. `Result Result`

Le coût est que vous vous retrouverez à penser et à gérer les erreurs d'ingénierie plus dans Rust que dans d'autres langues. Comme dans beaucoup d'autres domaines, le point de vue de Rust sur la gestion des erreurs est un peu plus serré que ce à quoi vous êtes habitué. Pour la programmation de systèmes, cela en vaut la peine.

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