# Gestion des erreurs en Go

*Andrew Gerrand*

*Traduction : Xavier Méhaut*

*V1.0*

# Sommaire

## Préambule

Si vous avez déjà écrit du code en Go, vous avez probablement été confrontés au type `os.Error`. Go utilise la valeur de type `os.Error` pour indiquer un état anormal. Par exemple, la fonction `os.Open` retourne une valeur de type `os.Error` non nulle quand elle échoue à ouvrir un fichier.

```
func Open(name string) (file *File, err Error)
```

La fonction suivante utilise `os.Open` pour ouvrir un fichier. Si une erreur survient, elle appelle `log.Fatal` pour afficher un message d'erreur et s'arrêter.

```
func main() {
    f, err := os.Open("filename.ext")
    if err != nil {
        log.Fatal(err)
    }
    // do something with the open *File f
}
```

Vous pouvez faire beaucoup de choses en Go en ne connaissant seulement que `os.Error`, mais dans cet article nous allons jeter un coup d'oeil plus détaillé sur `os.Error` et discuter des bonnes pratiques en Go à mettre en œuvre en matière de gestion des erreurs.

## Le type os.Error

Le type `os.Error` est un type interface. Une variable `os.Error` représente n'importe quelle valeur qui peut se décrire par une chaine de caractères (string). Ci-dessous la déclaration de cette interface :

```
package os

type Error interface {
    String() string
}
```

Il n'y a rien d'autre de spécial concernant ce type. Il s'agit juste d'une convention habituellement utilisée.

L'implémentation la plus commune de `os.Error` est le type `errorString`, non exporté, du package `os`.

```
type errorString string

func (s errorString) String() string { return string(s) }
```

Vous pouvez construire une de ces valeurs avec la fonction `os.NewError`. Elle prend une chaîne de caractères en entrée qui convertit une `os.errorString` puis renvoie une valeur de type `os.Error`.

```
func NewError(s string) Error { return errorString(s) }
```

Ci-dessous une façon d'utiliser `os.NewError`:

```
func Sqrt(f float64) (float64, os.Error) {
    if f < 0 {
        return 0, os.NewError("math: square root of negative number")
    }
    // implementation
}
```

Une fonction appelante passant un argument négatif à `Sqrt` recoit une valeur non nulle `os.Error` (dont la représentation concrète est une valeur de type `os.errorString`). Cette fonction appelante peut accéder à une chaîne erreur ("math: square root of...") en appelant la méthode String de `os.Error`, ou simplement en l'affichant :

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

Le package `fmt` package peut afficher n'importe quoi avec une méthode "`String() string`", ce qui inclut bien évidemment les valeurs de type `os.Error`.

C'est de la responsabilité de l'implémentation de récupérer le contexte. L'erreur renvoyée par `os.Open` est semblable à "open /etc/passwd: permission denied," et non pas juste comme "permission denied." Il manque néanmoins une information au sujet de l'argument invalide dans l'erreur renvoyée par `Sqrt`.

Pour ajouter cette information, une fonction utile est `Errorf` du package `fmt`. Elle formate une chaîne de caractères selon les règles de `Printf` et la renvoie comme une `os.Error` créée par `os.NewError`.

```
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g",
f)
}
```

Dans de nombreux cas, `fmt.Errorf` est sufisante, mais parce que `os.Error` est une interface , vous pouvez utiliser des structures de données élaborées comme valeur d'erreur, afin de permettre aux fonctions appelantes d'inspecter en détails l'erreur renvoyée.

Par exemple, notre hypothétique fonction appelante peut vouloir récupérer l'argument invalide passé par `Sqrt`. Nous pouvons permettre ceci en définissant une nouvelle erreur à la place de `os.errorString`:

```
type NegativeSqrtError float64

func (f NegativeSqrtError) String() string {
    return fmt.Sprintf("math: square root of negative number %g",
float64(f))
}
```

Une fonction appelante sophistiquée peut alors être un type assertion afin de vérifier une `NegativeSqrtError` et la traiter de manière spécifique, pendant que les fonctions appelantes

qui passent justes l'erreur à `fmt.Println` ou `log.Fatal` ne constaterons aucun changement dans leur comportement.

Un autre exemple ; le package `json` spécifie un type `SyntaxError` que la fonction `json.Decode` renvoie quand elle rencontre une erreur de syntaxe lors de du *parsing* un blob.

```
type SyntaxError struct {
    msg    string // description of error
    Offset int64  // error occurred after reading Offset bytes
}

func (e *SyntaxError) String() string { return e.msg }
```

Le champ `field` ne sera même pas affiché lors du formatage par défaut de l'erreur, mais les fonctions appelantes peuvent l'utiliser pour ajouter de l'information à leur messages d'erreur dans un fichier :

```
if err := dec.Decode(&val); err != nil {
    if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
    }
    return err
}
```

(Ceci est une version légèrement simplifiée de celle (actual code) du projet Camlistore )

L'interface `os.Error` requiert seulement une méthode `String`; les implémentations d'erreur spécifiques peuvent posséder des méthodes additionnelles. Par exemple, le package `net` renvoie des erreurs de `os.Error`, suivant la convention habituelle, mais certaines implémentations possèdent des méthodes supplémentaires définies par l'interface `net.Error`:

```
package net

type Error interface {
    os.Error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Le code client peut tester une `net.Error` avec un type assertion et même distinguer les erreurs de réseau transitoires. Par exemple, il se peut qu'un *web crawler* s'endorme et ne se réveille que quand il rencontre une erreur temporaire et n'en à rien à faire sinon.

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

## Simplifier la gestion des erreurs répétitives

En Go, la gestion des erreurs est quelque chose d'important. La conception du langage et les conventions adoptées vous encouragent à vérifier explicitement les erreurs à l'endroit où elle apparaissent (ce qui le distingue des conventions utilisées dans d'autres langages qui lèvent des exceptions pour les « trapper » par la suite à un niveau supérieur). Dans certains cas, ceci rend le code Go verbeux, mais heureusement il existe des techniques qui vous permettent de minimiser la gestion des erreurs répétitives.

Considérons une application App Engine avec un *handler* (gestionnaire) HTTP qui récupère un enregistrement (*record*) d'une base de doonées et le formate avec un *template*.

```
func init() {
    http.HandleFunc("/view", viewRecord)
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    key := datastore.NewKey("Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        http.Error(w, err.String(), 500)
        return
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        http.Error(w, err.String(), 500)
    }
}
```

Cette fonction gère des erreurs renvoyées par la fonction `datastore.Get` et la méthode `Execute` de `viewTemplate`. Dans les deux cas, elle présente une simple erreur à l'utilisateur avec un code HTTP de status 500 ("Internal Server Error"). Ceci semble gérable tel quel, mais ajoutez quelques *handlers* supplémentaires et vous arriverez rapidement à dupliquer de nombreuses fois un même code identique.

Pour réduire cette répétition, nous pouvons définir notre propre type `appHandler` HTTP qui inclut une valeur de type `os.Error` :

```
type appHandler func(http.ResponseWriter, *http.Request) os.Error
```

Nous pouvons ensuite modifier la fonction `viewRecord` pour renvoyer les erreurs :

```
func viewRecord(w http.ResponseWriter, r *http.Request) os.Error {
    c := appengine.NewContext(r)
    key := datastore.NewKey("Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return err
    }
    return viewTemplate.Execute(w, record)
}
```

C'est plus simple que la version originale, mais le package `http` ne comprend que les fonctions qui retournent `os.Error`. Pour corriger ceci, nous pouvons implémenter la méthode `ServeHTTP` de l' interface `http.Handler` sur `appHandler`:

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    if err := fn(w, r); err != nil {
        http.Error(w, err.String(), 500)
    }
}
```

La méthode `ServeHTTP` appelle la fonction `appHandler` et affiche l'erreur renvoyée (si présente) à l'utilisateur. Notez bien que le récepteur de la méthode est une fonction. (Go peut le faire !) ; la méthode invoque la fonction en appelant le récepteur dans l'expression `fn(w, r)`.

Maintenant lors de l'enregistrement de `viewRecord` avec le package http, nous utilisons la fonction `Handle` (à la place de `HandleFunc`) puisque `appHandler` est un `http.Handler` (et pas une `http.HandlerFunc`).

```
func init() {
    http.Handle("/view", appHandler(viewRecord))
}
```

Nous pouvons rendre encore plus efficace cette infrastructure de gestion des erreurs que nous avons mise en place. Plutôt que de juste afficher la chaîne d'erreur, il serait souhaitable de renvoyer à l'utilisateur un simple message d'erreur avec le code de status approprié, et de *logguer* le code d'erreur complet sur la console développeur de *App Engine* dans un but de *débuggage*.

Pour réaliser ceci, une structure `appError` contenant une `os.Error` et quelques autres champs :

```
type appError struct {
    Error os.Error
    Message string
    Code int
}
```

Ensuite nous modifions le type *appHandler* pour renvoyer des valeurs de type `*appError`:

```
type appHandler func(http.ResponseWriter, *http.Request) *appError
```

(Ce n'est habituellement pas judicieux de passer en retour le type concret d'une erreur plutôt qu'une `os.Error`, pour des raisons qui seront discutées dans un article futur, mais c'est une bonne chose de le faire ici parce que `ServeHTTP` est le seul endroit où l'on peut voir les valeurs et utiliser leur contenu)

Et créer la méthode `ServeHTTP` de `appHandler` qui affiche le `Message` de `appError` à l'utilisateur avec le `Code` HTTP de statu correct et *loggue* l' `Error` en entier sur la console développeur :

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    if e := fn(w, r); e != nil { // e is *appError, not os.Error.
        c := appengine.NewContext(r)
        c.Errorf("%v", e.Error)
        http.Error(w, e.Message, e.Code)
    }
}
```

Finalement, nous mettons à jour `viewRecord` avec la nouvelle signature de fonction et nous pouvons ainsi renvoyer un contexte plus large quand une erreur survient dans l'application :

```
func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
    c := appengine.NewContext(r)
    key := datastore.NewKey("Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

Cette version de `viewRecord` fait la même longueur que l'originale, mais maintenant chacune de ces lignes possède une signification spécifique et nous pouvons fournir à l'utilisateur une information d'erreur plus pertinente de son point de vue.

Ceci n'est pas terminé ; nous pouvons encore améliorer cette gestion des erreurs. Quelques idées :

- Fournir au gestionnaire (*handler*) d'erreurs un *template* HTML pour le formatage de l'erreur
- Rendre le *debuggage* plus aisé en écrivant la trace de la pile comme HTTP quand l'utilisateur est un administrateur
- Écrire un constructeur pour `appError` qui sauvegarde la trace de la pile (stack trace) pour améliorer le *débuggage*
- Récupérer des paniques (panics) à l'intérieur de `appHandler`, logguer l'erreur sur la console comme "Critical," pendant que l'on informe l'utilisateur qu'une "a serious error has occurred." C'est une attention délicate d'éviter d'exposer l'utilisateur à des messages d'erreur ésotériques causés par des erreurs de programmation. Voir l'article Defer, Panic, and Recover pour plus de détail.

## Conclusion

Une gestion des erreurs dans les règles est une exigence essentielle pour construire un bon logiciel. En employant les techniques décrites dans cet article, vous devriez pouvoir écrire du code plus fiable et néanmoins succinct.