

---

## retry

go report A+ license MIT \* used by 346 projects go report A+ GO reference codecov 0%

\* used by 346 projects

Simple library for retry mechanism

Slightly inspired by Try::Tiny::Retry

## SYNOPSIS

HTTP GET with retry:

```
1 url := "http://example.com"
2 var body []byte
3
4 err := retry.Do(
5     func() error {
6         resp, err := http.Get(url)
7         if err != nil {
8             return err
9         }
10        defer resp.Body.Close()
11        body, err = ioutil.ReadAll(resp.Body)
12        if err != nil {
13            return err
14        }
15        return nil
16    },
17 )
18
19 if err != nil {
20     // handle error
21 }
22
23 fmt.Println(string(body))
```

HTTP GET with retry with data:

```
1 url := "http://example.com"
2
3 body, err := retry.DoWithData(
4     func() ([]byte, error) {
5         resp, err := http.Get(url)
6         if err != nil {
7             return nil, err
8         }
9     })
```

---

```
 9      defer resp.Body.Close()
10      body, err := ioutil.ReadAll(resp.Body)
11      if err != nil {
12          return nil, err
13      }
14
15      return body, nil
16  },
17 )
18
19 if err != nil {
20     // handle error
21 }
22
23 fmt.Println(string(body))
```

More examples

## SEE ALSO

- giantswarm/retry-go - slightly complicated interface.
- sethgrid/pester - only http retry for http calls with retries and backoff
- cenkalti/backoff - Go port of the exponential backoff algorithm from Google's HTTP Client Library for Java. Really complicated interface.
- rafaeljesus/retry-go - looks good, slightly similar as this package, don't have 'simple' [Retry](#) method
- matryer/try - very popular package, nonintuitive interface (for me)

## BREAKING CHANGES

- 4.0.0
  - infinity retry is possible by set [Attempts](#) (0) by PR #49
- 3.0.0
  - [DelayTypeFunc](#) accepts a new parameter [err](#) - this breaking change affects only your custom Delay Functions. This change allow make delay functions based on error.
- 1.0.2 -> 2.0.0

- 
- argument of `retry.Delay` is final delay (no multiplication by `retry.Units` anymore)
  - function `retry.Units` are removed
  - more about this breaking change
- 0.3.0 -> 1.0.0
    - `retry.Retry` function are changed to `retry.Do` function
    - `retry.RetryCustom` (OnRetry) and `retry.RetryCustomWithOpts` functions are now implement via functions produces Options (aka `retry.OnRetry`)

## Usage

### **func BackOffDelay**

```
1 func BackOffDelay(n uint, _ error, config *Config) time.Duration
```

BackOffDelay is a DelayType which increases delay between consecutive retries

### **func Do**

```
1 func Do(retryableFunc RetryableFunc, opts ...Option) error
```

### **func DoWithData**

```
1 func DoWithData[T any](retryableFunc RetryableFuncWithData[T], opts ...Option) (T, error)
```

### **func FixedDelay**

```
1 func FixedDelay(_ uint, _ error, config *Config) time.Duration
```

FixedDelay is a DelayType which keeps delay the same through all iterations

### **func IsRecoverable**

```
1 func IsRecoverable(err error) bool
```

IsRecoverable checks if error is an instance of `unrecoverableError`

### **func RandomDelay**

```
1 func RandomDelay(_ uint, _ error, config *Config) time.Duration
```

RandomDelay is a DelayType which picks a random delay up to `config.maxJitter`

### **func Unrecoverable**

```
1 func Unrecoverable(err error) error
```

Unrecoverable wraps an error in `unrecoverableError` struct

---

### **type Config**

```
1 type Config struct {  
2 }
```

### **type DelayTypeFunc**

```
1 type DelayTypeFunc func(n uint, err error, config *Config) time.  
    Duration
```

DelayTypeFunc is called to return the next delay to wait after the retrieable function fails on `err` after `n` attempts.

### **func CombineDelay**

```
1 func CombineDelay(delays ...DelayTypeFunc) DelayTypeFunc
```

CombineDelay is a DelayType the combines all of the specified delays into a new DelayTypeFunc

### **type Error**

```
1 type Error []error
```

Error type represents list of errors in retry

### **func (Error) As**

```
1 func (e Error) As(target interface{}) bool
```

### **func (Error) Error**

```
1 func (e Error) Error() string
```

Error method return string representation of Error It is an implementation of error interface

### **func (Error) Is**

```
1 func (e Error) Is(target error) bool
```

### **func (Error) Unwrap**

```
1 func (e Error) Unwrap() error
```

Unwrap the last error for compatibility with `errors.Unwrap()`. When you need to unwrap all errors, you should use `WrappedErrors()` instead.

```
1 err := Do(  
2     func() error {  
3         return errors.New("original error")  
4     },  
5     Attempts(1),  
6 )  
7
```

---

```
8 fmt.Println(errors.Unwrap(err)) # "original error" is printed
```

Added in version 4.2.0.

#### **func (Error) WrappedErrors**

```
1 func (e Error) WrappedErrors() []error
```

WrappedErrors returns the list of errors that this Error is wrapping. It is an implementation of the `errwrap.Wrapper` interface in package `errwrap` so that `retry.Error` can be used with that library.

#### **type OnRetryFunc**

```
1 type OnRetryFunc func(attempt uint, err error)
```

Function signature of OnRetry function

#### **type Option**

```
1 type Option func(*Config)
```

Option represents an option for retry.

#### **func Attempts**

```
1 func Attempts(attempts uint) Option
```

Attempts set count of retry. Setting to 0 will retry until the retried function succeeds. default is 10

#### **func AttemptsForError**

```
1 func AttemptsForError(attempts uint, err error) Option
```

AttemptsForError sets count of retry in case execution results in given `err` Retries for the given `err` are also counted against total retries. The retry will stop if any of given retries is exhausted.

added in 4.3.0

#### **func Context**

```
1 func Context(ctx context.Context) Option
```

Context allow to set context of retry default are Background context

example of immediately cancellation (maybe it isn't the best example, but it describes behavior enough; I hope)

```
1 ctx, cancel := context.WithCancel(context.Background())
2 cancel()
3
4 retry.Do(
```

---

```
5 func() error {
6     ...
7 },
8 retry.Context(ctx),
9 )
```

#### **func Delay**

```
1 func Delay(delay time.Duration) Option
```

Delay set delay between retry default is 100ms

#### **func DelayType**

```
1 func DelayType(delayType DelayTypeFunc) Option
```

DelayType set type of the delay between retries default is BackOff

#### **func LastErrorOnly**

```
1 func LastErrorOnly(lastErrorOnly bool) Option
```

return the direct last error that came from the retried function default is false (return wrapped errors with everything)

#### **func MaxDelay**

```
1 func MaxDelay(maxDelay time.Duration) Option
```

MaxDelay set maximum delay between retry does not apply by default

#### **func MaxJitter**

```
1 func MaxJitter(maxJitter time.Duration) Option
```

MaxJitter sets the maximum random Jitter between retries for RandomDelay

#### **func OnRetry**

```
1 func OnRetry(onRetry OnRetryFunc) Option
```

OnRetry function callback are called each retry

log each retry example:

```
1 retry.Do(
2     func() error {
3         return errors.New("some error")
4     },
5     retry.OnRetry(func(n uint, err error) {
6         log.Printf("#%d: %s\n", n, err)
7     }),
8 )
```

---

### **func RetryIf**

```
1 func RetryIf(retryIf RetryIfFunc) Option
```

RetryIf controls whether a retry should be attempted after an error (assuming there are any retry attempts remaining)

skip retry if special error example:

```
1 retry.Do(  
2     func() error {  
3         return errors.New("special error")  
4     },  
5     retry.RetryIf(func(err error) bool {  
6         if err.Error() == "special error" {  
7             return false  
8         }  
9         return true  
10    })  
11 )
```

By default RetryIf stops execution if the error is wrapped using `retry.Unrecoverable`, so above example may also be shortened to:

```
1 retry.Do(  
2     func() error {  
3         return retry.Unrecoverable(errors.New("special error"))  
4     }  
5 )
```

### **func UntilSucceeded**

```
1 func UntilSucceeded() Option
```

UntilSucceeded will retry until the retried function succeeds. Equivalent to setting Attempts(0).

### **func WithTimer**

```
1 func WithTimer(t Timer) Option
```

WithTimer provides a way to swap out timer module implementations. This primarily is useful for mocking/testing, where you may not want to explicitly wait for a set duration for retries.

example of augmenting time.After with a print statement

```
1 type struct MyTimer {}  
2  
3 func (t *MyTimer) After(d time.Duration) <- chan time.Time {  
4     fmt.Print("Timer called!")  
5     return time.After(d)  
6 }  
7
```

---

```
8 retry.Do(  
9     func() error { ... },  
10     retry.WithTimer(&MyTimer{ })  
11 )
```

#### **func WrapContextErrorWithLastError**

```
1 func WrapContextErrorWithLastError(wrapContextErrorWithLastError bool)  
   Option
```

WrapContextErrorWithLastError allows the context error to be returned wrapped with the last error that the retried function returned. This is only applicable when Attempts is set to 0 to retry indefinitely and when using a context to cancel / timeout

default is false

```
1 ctx, cancel := context.WithCancel(context.Background())  
2 defer cancel()  
3  
4 retry.Do(  
5     func() error {  
6         ...  
7     },  
8     retry.Context(ctx),  
9     retry.Attempts(0),  
10    retry.WrapContextErrorWithLastError(true),  
11 )
```

#### **type RetryIfFunc**

```
1 type RetryIfFunc func(error) bool
```

Function signature of retry if function

#### **type RetryableFunc**

```
1 type RetryableFunc func() error
```

Function signature of retryable function

#### **type RetryableFuncWithData**

```
1 type RetryableFuncWithData[T any] func() (T, error)
```

Function signature of retryable function with data

#### **type Timer**

```
1 type Timer interface {  
2     After(time.Duration) <-chan time.Time  
3 }
```



---

Timer represents the timer used to track time for a retry.

## Contributing

Contributions are very much welcome.

## Makefile

Makefile provides several handy rules, like README.md [generator](#) , [setup](#) for prepare build/dev environment, [test](#), [cover](#), etc...

Try [make help](#) for more information.

## Before pull request

maybe you need [make setup](#) in order to setup environment

please try: \* run tests ([make test](#)) \* run linter ([make lint](#)) \* if your IDE don't automatically do [go fmt](#), run [go fmt](#) ([make fmt](#))

## README

README.md are generate from template `.godocdown.tmpl` and code documentation via `godoc-down`.

Never edit README.md direct, because your change will be lost.