
这次讲什么呢？
《Retry 可以怎么设计？》

🧐 如何是你，你会考虑什么哪些点？

来，每人说一个点。不说重复。

重试次数

- 简单指定次数
- 考虑无限重试

重试间隔

- 随机 (需要有个基数)
- 固定间隔
- 时间逐步拉长
- 手动指定重试的间隔
- 组合多种策略
- 没传的时候，需要有个 backoff
- 设定最大间隔时间

重试结果

- 多次重试的结果可能不尽相同（比如503、限流、账户欠费等），返回所有异常（存于数组）。
- 或者只需要返回最后错误

😯 定义执行的 Callback

- 执行前
- 执行后

比如可以打日志之类的。

出现什么情况才进行重试

- 指定异常出现，才会进行尝试

比如出现 429、503 等

 出现什么情况就终止重试

- 到达次数
- 遇到指定的异常（不再重试）

比如出现不可逆的业务错误，重试再多次都是没用的。

 配置项的问题

- 需要有一个全局的配置项（也就是默认项）
- 怎么方便让使用者进行赋值

实现: 基于 Golang

- 方法当做参数传递
- 变长的参数传递

```
type Option func(*Config)

// Attempts set count of retry. Setting to 0 will retry until the retried function succeeds.
// default is 10
// 返回一个 func 。接收一个 config 的地址。
func Attempts(attempts uint) Option {
  return func(c *Config) {
    c.attempts = attempts
  }
}

// MaxDelay set maximum delay between retry
// does not apply by default
// 返回一个 func 。接收一个 config 的地址。
func MaxDelay(maxDelay time.Duration) Option {
  return func(c *Config) {
    c.maxDelay = maxDelay
  }
}
```

生成默认的配置

```go
func newDefaultRetryConfig() *Config {
  return &Config{
    attempts:      uint(10),
    delay:         100 * time.Millisecond,
    maxJitter:     100 * time.Millisecond,
    onRetry:       func(n uint, err error) {},
    retryIf:       IsRecoverable,
    delayType:     CombineDelay(BackOffDelay, RandomDelay),
    lastErrorOnly: false,
    context:       context.Background(),
    timer:         &timerImpl{},
  }
}
```

```
func Do(retryableFunc RetryableFunc, opts ...Option) error {
  var n uint
  config := newDefaultRetryConfig()

  for _, opt := range opts {
    opt(config)
  }

  for n < config.attempts {
    err := retryableFunc()
    if err != nil {
      if !config.retryIf(err) {
        break
      }
      config.onRetry(n, err)
      if n == config.attempts-1 {
        break
      }
    } else {
      return nil
    }
    n++
  }

  return errorLog
}
```

调用的方式，如下：

```
retry.Do(
    func() error {
        return errors.New("special error")
    },
    retry.RetryIf(func(err error) bool {
        if err.Error() == "special error" {
            return false
        }
        return true
    })
)
```

实现: 基于 Ruby

- 参数可以直接用 hash，就可以达到很好的效果
- 至于执行函数，直接传个 block 就好了。

```ruby
def retriable(opts = {})
  # 新增默认配置，并对进行指定配置的合并
  local_config = opts.empty? ? config : Config.new(config.to_h.merge(opts))

  exception_list = on.is_a?(Hash) ? on.keys : on
  start_time = Time.now
  elapsed_time = -> { Time.now - start_time }

  if intervals
    tries = intervals.size + 1
  else
    intervals = ExponentialBackoff.new(
      tries:          tries - 1,
      base_interval:  base_interval,
      multiplier:     multiplier,
      max_interval:   max_interval,
      rand_factor:    rand_factor
    ).intervals
  end
```

```
  tries.times do |index|
    try = index + 1

    begin
      # 执行 block
      return Timeout.timeout(timeout) { return yield(try) } if timeout
      return yield(try)
    rescue *[*exception_list] => exception
      if on.is_a?(Hash)
        raise unless exception_list.any? do |e|
          exception.is_a?(e) && ([*on[e]].empty? || [*on[e]].any? { |pattern| exception.message =~ pattern })
        end
      end

      interval = intervals[index]
      on_retry.call(exception, try, elapsed_time.call, interval) if on_retry
      raise if try >= tries || (elapsed_time.call + interval) > max_elapsed_time
      sleep interval if sleep_disabled != true
    end
  end
end
```

调用的方式，如下：

```
retriable(on: Timeout::Error, tries: 3, base_interval: 1) do
  # code here...
end
```

一些更加具体的可以看一下

- retry-go
- retriable

鸡汤一下

- 简单聊了一下重试的设计点。
- 对比了一下 Ruby + Golang 实现方式的异同。

随着服务相互调用越来越频繁，需要更加重视失败的处理。

重试这个我们经常遇到，但也可深挖，从中发现乐趣。

慢慢积累自己的设计思路，对以后其他功能的设计，也会考虑得越来越全。