

# Goらしい APIを求める旅路

GoCon 2018 Spring Apr 15 2018  
Daisuke Maki @lestrrat (HDE inc)



- **@lestrrat**
- Perl/Go hacker, author, father
- Author of [github.com/peco/peco](https://github.com/peco/peco)
- Organizer for builderscon

# <宣伝>



## スポンサー募集、今月末まで！

<http://blog.builderscon.io/entry/call-for-sponsors-2018>

# お題

# 「Goっぽい」コード

**Goを使うなら  
なるべくGoっぽいAPIがいい**

cenkalti/backoff: The exponent x

GitHub, Inc. [US] | <https://github.com/cenkalti/backoff>

This repository Search Pull requests Issues Marketplace Explore

cenkalti / backoff Watch 16 Star 942

Code Issues 1 Pull requests 0 Projects 0 Wiki Insights

The exponential backoff algorithm in Go (Golang). <https://godoc.org/github.com/cenkalti/backoff>

87 commits 1 branch 3 releases 12 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or fork

cenkalti committed on 25 Dec 2017 rename WithMaxTries to WithMaxRetries; fix #52; fix #54 Latest commit 2ea60e5 on 25 Dec 2017

.gitignore	Initial commit	4
.travis.yml	travis: add 1.x go version	7 m
LICENSE	Initial commit	4
README.md	update repo path	2
backoff.go	ticker: document it's unsafe to access backoff policy while ticker ru...	6 m
backoff_test.go	fixes #15 - added tons of new documentation, godoc, and small cosmeti...	3
context.go	Support for Go < 1.7	

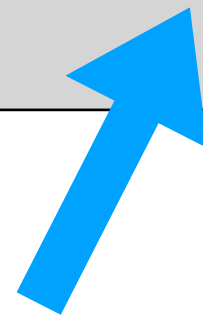
**[github.com/cenkalti/backoff](https://github.com/cenkalti/backoff)**

```
backoff.Retry(f, strategy)
```



**[github.com/cenkalti/backoff](https://github.com/cenkalti/backoff)**

```
backoff.Retry(f, strategy)
```



**[func() error]**

# Callback/Closure

- 関数を受け取るAPIでは、シグネチャがポイントになる
- 関数を許すからにはadhocな関数呼び出しができるべき

# net/http

Goっばい!

```
type HandleFunc func(ResponseWriter, *Request)
```

- net/httpでこの関数の副作用は気にしない  
でよい
- AdhocなHandlerをポンポン登録できる

```
backoff.Retry(f, stretegy)
```

# What About...

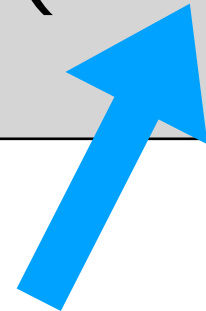
```
func Foo() (Result, Result, Result, error)
```

# This no worky...

```
backoff.Retry(Foo, ...)
```

# This no worky...

```
backoff.Retry(Foo, ...)
```



## signatureが合わない


# Capture Return Values

```
backoff.Retry(func() error {  
    a, b, c, err := Foo()  
    if err != nil {  
        return err  
    }  
}, ...)
```



# Capture Return Values

```
backoff.Retry(func() error {  
    a, b, c, err := Foo()  
    if err != nil {  
        return err  
    }  
}, ...)
```



この戻り値をRetryの外側で使うには？

# Upper Scope

```
var a, b, c Result
backoff.Retry(func() error {
    a, b, c, err := Foo()
    if err != nil {
        return err
    }
}, ...)
```

# Upper Scope

```
var a, b, c Result  
backoff.Retry(func() error {  
    a, b, c, err := Foo()  
    if err != nil {  
        return err  
    }  
}, ...)
```

あ！

# Final Result

```
var a, b, c Result
backoff.Retry(func() error {
    a, b, c, err = Foo()
    if err != nil {
        return err
    }
}, ...)
```

# ちょっとまった！

- スコープを気にしながらコードを書かないといけない
- `:=` と `=` を間違ってもコンパイラが怒ってくれない
- 毎回ムダ？にクロージャを作ってて **Goっぽくない...**

どんなAPIだと **Goっぽい!?**

+

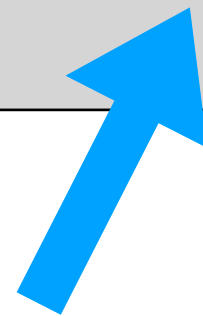
拡張性を持たせられる？

# Use Interface?

```
backoff.Retry(f, stretegy)
```

# Use Interface?

```
backoff.Retry(f, stretegy)
```



```
type Operation interface {  
    Do() error  
}
```



# Store The Results

```
type FooWrapper struct {  
    A Result  
    B Result  
    C Result  
}  
  
func (f *FooWrapper) Do() error {  
    f.A, f.B, f.C, err = Foo()  
    ...  
}
```

# Final Result

```
var f FooWrapper  
backoff.Retry(f, ...)  
// Use f.A, f.B, f.C
```

# Final Result

```
var f FooWrapper  
backoff.Retry(f, ...)  
// Use f.A, f.B, f.C
```

「これじゃない感」…  
毎回型を作る必要がある…



**たとえばadhocな関数は？**

# Adhoc Functions?

```
type OperationFunc func() error  
func (f OperationFunc) Do() error {  
    return f()  
}
```

# Adhoc Functions?

```
type OperationFunc func() error  
func (f OperationFunc) Do() error {  
    return f()  
}
```

**http.HandlerFuncと同じ  
形にすればキレイになる？**



# あ、これ前と同じだ

```
var a, b, c Result
backoff.Retry(OperationFunc(func() error {
    a, b, c, err = Foo()
    if err != nil {
        return err
    }
}), ...)
```



小細工では同じような形に  
なってしまう



**一歩引いて考え直す...**

# backoffは何をするのか

- 関数を実行する
- 成功したら終わる
- 失敗したら、待つ
- `optional: context.Context`で途中停止

# First pass

```
for {  
    if err := f(); err == nil {  
        return  
    }  
    time.Sleep(delay)  
}
```

# First pass

```
for {  
  if err := f(); err == nil {  
    return  
  }  
  time.Sleep(delay)  
}
```

関数を実行する

成功したら終わる

失敗したら待つ

# backoffは何をするのか

- ~~関数を実行する~~
- ~~成功したら終わる~~
- 失敗したら、待ち時間を増やしながら待つ
- optional: context.Contextで途中停止

# Second pass

```
for {  
    if err := f(); err == nil {  
        return  
    }  
  
    select {  
    case <-ctx.Done():  
        return  
    case <-time.After(nextDelay()):  
    }  
}
```

# Second pass

```
for {  
  if err := f(); err == nil {  
    return  
  }  
}
```

関数を実行する

成功したら終わる

```
select {  
case <-ctx.Done():  
  return  
case <-time.After(nextDelay()):  
}  
}
```

# Second pass

```
for {  
  if err := f(); err == nil {  
    return  
  }  
}
```

関数を実行する

成功したら終わる

```
select {  
case <-ctx.Done():  
  return  
case <-time.After(nextDelay()):  
}  
}
```

**context.Contextで  
途中停止**

**失敗したら待ち時間を  
増やしながら待つ**



**この形を簡単に書けるように  
すればいいはず...**

**[github.com/lestrrat-go/backoff](https://github.com/lestrrat-go/backoff)**

# Backoff Policy Object

```
policy := backoff.NewExponential(...)
```

# Backoff object

```
b, cancel := policy.Start(ctx)
```

```
b, cancel := policy.Start(ctx)
defer cancel()
for {
    if err := f(); err == nil {
        return
    }
    select {
    case <-b.Done():
        return
    case <-b.Next():
    }
}
```

```
b, cancel := policy.Start(ctx)
defer cancel()
for {
    if err := f(); err == nil {
        return
    }
    select {
    case <-b.Done():
        return
    case <-b.Next():
    }
}
```

<-chan struct{}

<-chan time.Time

**ただのコードブロックなので  
戻り値も自由自在**

```
b, cancel := policy.Start(ctx)
defer cancel()
for {
    a, b, c, err := f()
    if err == nil {
        // use or return a, b, c here
    }
    select {
    case <-b.Done():
        return
    case <-b.Next():
    }
}
```



```
b, cancel := policy.Start(ctx)
defer cancel()
for {
```

```
a, b, c, err := f()
if err == nil {
    // use or return a, b, c here
}
```

```
select {
case <-b.Done():
    return
case <-b.Next():
```

```
}
```



**Bonus: separate Policy/Backoff**

# bを共有せざるを得ない

```
b := cenkalti.NewExponential()  
for ... {  
    go func(b cenkalti.Backoff) {  
        // b is shared... be careful!  
    }(b)  
}
```

# 共有されるpolicyとbは別

```
for ... {  
    go func(b backoff.Backoff, cancel func()) {  
        // b is safe to be used in a separate goroutine  
    }(policy.Start(ctx))  
}
```

# **Problem:**

# **Boilerplate Code is Long**

```
b, cancel := policy.Start(ctx)
defer cancel()
for {
    a, b, c, err := f()
    if err == nil {
        return
    }
    select {
    case <-b.Done():
        return
    case <-b.Next():
    }
}
```

```
b, cancel := policy.Start(ctx)
defer cancel()
for {
    a, b, c, err := f()
    if err == nil {
        return
    }
    select {
    case <-b.Done():
        return
    case <-b.Next():
    }
}
```



```
for {  
  select {  
    case <-b.Done():  
      return  
    case <-b.Next():  
  }  
}
```



```
for {  
  select {  
    case <-b.Done():  
      return  
    case <-b.Next():  
  }  
}
```



```
for {  
  select {  
    case <-b.Done():  
      return  
    case <-b.Next():  
  }  
}
```

bail out of loop



```
for {  
  select {  
    case <-b.Done():  
      return  
    case <-b.Next():  
  }  
}
```

bail out of loop

continue loop



これ、 Loop Condition だ

# func to determine loop condition

```
func Continue(b *Backoff) bool {  
    select {  
    case <-b.Done():  
        return false  
    case <-b.Next():  
        return true  
    }  
    return false // never reached  
}
```

# キョツとした！

```
b, cancel := policy.Start(ctx)
defer cancel()
for backoff.Continue(b) {
    a, b, c, err := f()
    if err == nil {
        return
    }
}
```

# キョツとした！

```
b, cancel := policy.Start(ctx)
defer cancel()
for backoff.Continue(b) {
    a, b, c, err := f()
    if err == nil {
        return
    }
}
```



use as condition

# 確認



# 確認



contextで途中停止

# 確認



contextで途中停止



channelでタイミング制御

# 確認

- Goっぽい! contextで途中停止
- Goっぽい! channelでタイミング制御
- Goっぽい! for backoff.Continue() でループ制御

# 確認

- Goっぽい! contextで途中停止
- Goっぽい! channelでタイミング制御
- Goっぽい! for backoff.Continue() でループ制御
- Goっぽい! goroutine safe by design

# 確認

Goっぽい!

contextで途中停止

Goっぽい!

channelでタイミング制御

Goっぽい!

for backoff.Continue() でループ制御

Goっぽい!

goroutine safe by design

Goっぽい!

クロージャに頼る必要がない

採用



APIデザインをする時は**その言語の自然なフローを許す形を**  
**考えるのがお勧め**

**（なお、コードをよく読むと、  
cenkalti/backoffも実装はほぼ同じ仕  
組みになっているが、公開しているイ  
ンターフェースがまったく違う）**



# おまけ

Webアプリケーション開発のためのプログラミング技術情報誌  
FOR ALL WEB APPLICATION DEVELOPERS ウェブDBプレス

新人さん  
大歓迎!!

モダンなコードを  
×ユツと凝縮!

# WEB+DB PRESS Python

イマドキ vol. 104  
2018

入門

[文法][機械学習][Web開発]を一気に学ぼう

いきなり iPhone アプリ  
開発

カメラの写真を加工してTwitter投稿!

はじめての Unity

シューティングゲームを作ろう!

Google APIから学ぶ自動生成  
PHPの継続的バージョンアップ  
HashiCorp Vaultで秘密情報管理









# 「Google APIから学ぶ自動生成」

※ Goの連載記事です

※ GoのためのAPIデザインの話です



# End