Perl Mongerのための Goの始め方

id:karupanerura

Gotanda.pm #8

About me

- Kenta SATO (id:karupanerura)
- Perl/Go/Swift/Kotlin/Java/Crystal/C99/etc..
- Gotanda.pm Author
- Senior Engineer at Mobile Factory, Inc.
  - WebApp/iOS/Android/Operation/etc..

おことわり

- ごりごり書いている人向けではありません
    - 暖かいマサカリをお願いします
- PerlとはPerl6ではなくPerl5を指します
    - Perl6とPerl5は別の言語です（本当に）

おことわり

- 原理的の説明をするためにベストプラクティスから外れた説明をする場合があります
- 鵜呑みにして愚直に書かないでください
- この点にはマサカリを投げないでください
- やられたら泣きます（泣きません）

見た目の違い


$_='ev
al("seek\040D
0;");foreach(1..3)
@camel1hump;my$camel;
<DATA>){$_=sprintf("%-6
1=split(//);if(defined($
p=split(//);}while(@dromeda
;my$CAMEL=3;if(defined($_=shif
))&&/\S/){$camel1hump+=1<<$CAMEL;}
efined($_=shift(@dromedary1))&&/\S/){
<<$CAMEL;}$CAMEL--;if(defined($_=shift(
ATA,0,
{<DATA>;}my
my$Camel ;while(
9s",$_);my@dromedary
_=<DATA>)){@camel1hum
ry1){my$camel1hump=0
t(@dromedary1
$CAMEL--;if(d
$camel1hump+=1
@camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL--;if(
defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
L;;}$camel.=(split(//,"\040..m`{/J\047\134}L^7FX"))[$camel1h
ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
064\065\066\067\070/;y/12345678/JL7F\175\173\047`/;$_=reverse;
print"$_\040$Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y
/LJF7\173\175`\047/12345678/;y/12345678/JL7F\175\173\0 47`/;
$_=reverse;print"\040$_$Camel\n";}';;s/\s*//g;;eval; eval
("seek\040DATA,0,0;");undef$/;$_=<DATA>;s/\s*//g;( );;s
;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__ \124
\1 50\145\040\165\163\145\040\157\1 46\040\1 41\0
40\143\141 \155\145\1 54\040\1 51\155\ 141
\147\145\0 40\151\156 \040\141 \163\16 3\
157\143\ 151\141\16 4\151\1 57\156
\040\167 \151\164\1 50\040\ 120\1
45\162\ 154\040\15 1\163\ 040\14
1\040\1 64\162\1 41\144 \145\
Perl

```go
func (c *HttpClient) makeRequest(method, path string, headers map[string]string, content interface{}) (*
	uri, body, err := c.preformContent(method, path, content)
	if err != nil {
		return nil, err
	}

	req, err := http.NewRequest(method, uri, body)
	if err != nil {
		return nil, err
	}

	if content != nil {
		contentType := c.Config.RequestEncoder.GetContentType()
		req.Header.Set("Content-Type", contentType)
	}

	req.Header.Set("User-Agent", c.Config.UserAgent)
	for k, v := range c.Config.Headers {
		req.Header.Set(k, v)
	}
	if headers != nil {
		for k, v := range headers {
			req.Header.Set(k, v)
		}
	}
	return req, nil
}

func (c *HttpClient) preformContent(method, path string, content interface{}) (uri string, body io.Reade
	uri = c.Config.Server.MakeUrl(path)

	if content != nil {
		if isContentMethod(method) {
			body, err = c.Config.RequestEncoder.Encode(content)
			if err != nil {
				return
			}
```

Go

※冗談です

学ぶ

Perlを学ぶ

- 公式のドキュメント: perldoc perlintro
- 定番の書籍: 初めてのPerl、雅なPerl入門
    - WEB+DB PRESS: Perl Hackers Hub
- 勉強会: Perl入学式

Goを学ぶ

- 公式のドキュメント: A Tour of Go
- 定番の書籍: プログラミング言語Go (未発売)
    - WEB+DB PRESS: Vol.82 はじめてのGo
- 勉強会: (定期開催されているものはなさそう)

ドキュメント

Perlのドキュメント(読み方)

- perldocコマンドからぜんぶ読める
  - perldocのドキュメント: perldoc perldoc
  - perlのドキュメント: perldoc perl (目次)
- metacpan.org や perldoc.jp などでも読める

Goのドキュメント(読み方)

- 言語のドキュメント: https://golang.org/doc/
- モジュールのドキュメント: godoc.org
- godocコマンドでも読める

Perlのドキュメント(形式)

- 割と自由、DESCRIPTION/SYNOPSISは重要
    - DESCRIPTION: モジュールの説明
    - SYNOPSIS: 一貫的なサンプルコード
- POD形式で書かれる

Goのドキュメント(形式)

- 構造体/インターフェース/関数毎に書かれる
  - publicな関数などにドキュメントが書かれていないと警告が出る
  - 関数ごとにサンプルコード(Example)がある
- コメントとしてソースコード内に書かれる

環境

Perlの環境

- 処理系はperlのみ
- マルチOSサポート
- 実行環境に強く依存する
  - plenvやCatronなどでバージョンを固定する

Goの環境

- 処理系はgoのみ
- マルチOSサポート (クロスコンパイルも容易)
- ビルド環境に強く依存する
  - ビルド環境のバージョンがstatic linkされたバイナリが生成される

インストール

Perl 処理系のインストール

- anyenv install plenv
- plenv install -l # バージョン確認
- plenv install 5.22.1
- plenv global 5.22.1
- plenv install-cpanm

Perl モジュールのインストール

- cpanm App::revealup
  - テストがローカルで実行される
- cpanm -n App::revealup
  - テストなし
- cpanm –install-deps .
  - カレントの依存モジュールをインストール

Perl 環境の勘所

- 実行環境に合わせて柔軟に開発環境を変える
- OSにインストールするとroot権限でモジュールをインストールする必要がある
- パスの指定は可能
- CartonやLocal::libと併用する手もある

Go 処理系のインストール

- OS X: brew install go
- Debian/Ubuntu: apt-get install golang
- binary: https://golang.org/dl/
- /usr/bin とかに普通にインストールで十分

Go モジュールのインストール

- go get github.com/karupanerura/gostress
  - インストールのみ
- go get -t github.com/karupanerura/gostress
  - テストしつつインストール
- go get -d .
  - 依存モジュールをインストール

Go 環境の勘所

- 多様な実行環境を考えなくて良い
  - 実行環境のバージョンを気にしなくて良い
    - 必要が出たらgoenvなどを使うと良い
- モジュールはGOPATH以下にインストール
  - $HOME/go とかにしとけばroot権限不要

エディタ

Perlのエディタ事情

- emacs
  - cperl-mode + flycheck + perl-completion
- vim
  - vim-perl + quickrun + perlomni
  - くわしくない

Goのエディタ事情

- emacs
  - go-mode + + flycheck + gocode
- vim
  - vim-go + vim-godef + gocode
  - くわしくない

人人人人人人人人

> 僕も知りたい <

￣Y^Y^Y^Y^Y^Y^Y^Y^Y￣

言語仕様

名前空間

Perl 名前空間

- package = 名前空間 = クラス
- private/publicという概念は無い
  - 名前空間のレキシカルスコープは作れる
  - privateとして扱いたいサブルーチンの場合はアンダースコアを名前のprefixにするのが慣習

Perl 名前空間

- ファイルパスとpackageに強い関係がある
  - Foo::Bar = Foo/Bar.pm
- ロードするときはuseまたはrequireを使う
  - use Foo::Bar;

Perl 名前空間

- 他のパッケージにはフルネームでアクセス
    - Foo::Bar::baz() # Foo::Barのbuz()を実行
- データ(リファレンス)とpackageを紐付けることができる(後述)

Go 名前空間

- package = パッケージ
- 名前で勝手にアクセス制御が掛かる
  - 小文字で始まるものはpackage
  - 大文字で始まるものはpublic

Go 名前空間

- ファイルパスとpackageに強い関係がある
  - = Foo/Bar.pm
- 他のパッケージにはフルネームでアクセス
  - Foo::Bar::baz() # Foo::Barのbuz()を実行

型

Perlと型

- 動的型付け言語（コンテキストによる型付け）
- Scalar/Array/Hash 及びそれらのリファレンス
- 他の動的型付け言語とくらべても特殊
- 詳しく語ると60分くらい喋れてしまうので
  今回は詳しい言及を控える(Goメインなので)

Perlのデータ型 (Scalar)

- 単一の値を表すデータ型
- 整数/実数/文字列/リファレンスが入る
- 人間にとって $age が文字列かどうかは関心外
  - コンテキストによっては決めてかかりたい
  - $age > 20 # ageを整数で扱うコンテキスト

Perlのデータ型 (Array/Hash)

- 配列
  - 順序を持つデータの集合
- ハッシュ
  - 名前を持つデータの集合

Perlのデータ (リファレンス)

- データの実体への参照(Cのポインタに近い)
  - ファイルシステムでいうところのショートカットやシンボリックリンク
- リファレンス・カウントGC
  - 実体への参照がなくなると実体を破棄

Goと型

- 強い静的型付け言語
- 変数や関数(引数/返り値)、データが型を持つ
  - 変数とデータの型が一致する必要がある
  - 変数と関数の型が一致する必要がある
  - とにかく型が一致する必要がある

Goのデータ型 (基本)

- 整数: int, int32, int64
- 実数: float, float32, float64
- 文字列: string

Goのデータ型 (array)

- 順序を持ったデータの集合
- サイズ10の文字列の配列: [10]string
- 一度作ったら拡張することはできない

Goのデータ型 (slice)

- 配列の一部をスライスした参照
  - 分からない人はポインタの理解が足りない
- 以下のスライドの39ページ以降を参照すべし
  - http://www.slideshare.net/yasi_life/go-14075425

Goのデータ型 (map)

- 名前を持ったデータの集合
- stringをキーにした整数のmap: map[string]int
- サイズは気にしなくてもよしなにしてくれる

Goのデータ型 (struct)

- 単一のデータを表現するデータの集合
- Point{X: 1, Y: 1} みたいなノリのアレ
- 自由に定義できる
- C++のstruct/classに近い (後述)

Goのデータ型 (pointer)

- データの実体への参照(Cのポインタに近い)
    - Perlのリファレンスと同じ
- 全てのデータ型はポインタを作れる

Goの型 (interface)

- 特定の関数を呼び出せる型を示す
- duck typingを支援する
- 自由に定義できる
- Javaのinterfaceに近いがimplementsする必要はない

メモリ管理

Perlのメモリ管理

- 全部ヒープ領域に持つ
- メモリプールを持って動的に割り当てる
- 全てのデータはGCで破棄される
    - リファレンスカウントなので循環参照に弱い

Goのメモリ管理

- 関数内だけで使われる値: スタック領域
- 関数外でも使われる値: ヒープ領域
- ヒープ領域のデータはGCで管理される
  - Mark & Sweep なので循環参照に強いがGCされるものが多いとGCが重くなる

関数

Perl 関数

- サブルーチンが代替
- 仮引数は無い
  - @_を適当に代入して使う
- 複数の値を返すことができる

Go 関数

- 仮引数と返り値の型を明示する必要がある
  - @_ みたいな野蛮なことはできない
- 複数の値を返すことができる
  - 別々の変数で全ての返り値を受け取る

メソッド

Perl メソッド

- package(クラス)やpackageにbless(ひも付け)されたリファレンス(=インスタンス)からサブルーチンをメソッド呼び出しできる

- メソッド呼び出しするとサブルーチンの第一引数にコンテキストとしてクラスやインスタンスが渡される

Perl メソッド

- @PackageName::ISA が継承パッケージ
  - 見た目の通り多重継承が可能
- メソッド呼び出しでは継承を辿ってくれる
- このからくりをblessと組み合わせることでOOPが可能になる

Perl メソッド (蛇足)

- AUTOLOAD
  - rubyでいうところのmethod_missing
- UNIVERSAL
  - rubyでいうところのObjectクラス

Go メソッド

- 自分のpackageのデータ型にはメソッドのようなものを生やすことができる
- structやそのポインタに生やすのが一般的
    - 他の言語のクラスのインスタンスメソッドっぽくなる

Go メソッド (蛇足)

- structのmixinは可能だが実態はただの委譲
  - 継承ではないので直にメソッドは呼べない
    - 委譲なのでcontext objectが異なることに
- 型情報もisaの関係にならない

Go メソッド (超蛇足)

- 静的型付け言語なのでAUTOLOADできません
  - ちなみにCrystalだとmacroで実現している
- UNIVERSALみたいなやつはないのー？
  - 継承が無いのでお察しくださいな

前置き終わり

実用的な話

文字列操作

Perl の文字列操作

- 正規表現まじ最高まじ便利!!
  - 基本的にshell script由来の文化なので...
  - tr///とs///とかあればだいたいなんとかなる
  - なんとかならないところでuc/lcとかを使う

Go の文字列操作

- コアがサポートする機能は最小限
- stringsパッケージでだいたいなんとかなる
- 困ったらまずここを調べよ
- 正規表現を気軽に使うと可読性を損なう
    - 正規表現のコードがPerlとくらべて冗長

型変換

Perl の型変換

- コンテキストで型を明示するのでデータの型変換がそもそも不要
  - 強いて言えばコンテキスト毎に暗黙的に型変換が起こっているのに近いがこれは厳密には正確な表現ではない

Go の型変換

- 型が厳密なので型変換したいことが多い
- strconvパッケージでだいたいなんとかなる
- 型キャストは最終手段と心得よ
- interfaceをうまく活用すると多くの場合では型キャストの必要性がなくなる

エラー処理

Perl のエラー処理

- 例外
  - eval BLOCK の中でdieする
    - $@ にdieに渡したScalar値が入る
- 普通

Go のエラー処理

- 返り値でerror型を返すのが風習

- result, err := strconv.Atoi(“unko”)

- if err != nil { ... } で全部なんとかする

- 分岐書き忘れてもコンパイラは助けてくれん

Go のエラー処理

- 未使用の変数があるとコンパイルエラー
- 単一のスコープで単一のエラーを処理する
  - するとerror型の値を使わないとコンパイルエラーが発生する
- catch漏れがあるとコンパイルエラーする世界

構造設計

※個人の見解です

Perl の構造設計

- 小さいものは手続きっぽくゴリゴリっと書く
- 大きなものはOOPらしく分けて書く
- WebとかだとAmon2とか使ってMVCしたり
- package(クラス)でしっかりスコープを切る

Go の構造設計

- interfaceを中心に設計する
    - 振る舞いを受け渡していくイメージ
    - ポリモーフィズム的な考え方
- スタック領域をうまく使うように書くと高速

Go の構造設計

- 大きなものを作るときは内部的にpackageを分けるとよさそう
    - 相対パスでパッケージをロードできるのでそれをうまく活用する

- https://github.com/builderscon/octav

並列処理

※個人の見解です

Perl の並列処理

- fork一択
    - threadのようなものもなくはないがゴミ
        - 使ってはならない
- 同期はファイルを使ったりPIPEを使ったり

Perl の並列処理

- なるべくプロセスが使いまわせるようにする
  - つまりいわゆるpre-fork式
- プロセス間の同期が必要になったら負け
  - どうしても必要ならPerlには向いてないと割り切るかAnyEventなどを併用する

Go の並列処理

- goroutineというスレッドのようなものがある
    - 「並列で動いても良いもの」を作るイメージ
    - Goがいいかんじにスケジューリングする
- 基本的にはchannelで同期を取る

Go の並列処理

- 同期のコストより並列化のメリットが高ければばがんがんgoroutineを作ると良さそう
    - 同期が頻繁に必要になるなら考え方を変えてみる
- プロセスと違って気軽に殺せない点には注意

まとめ

まとめ

- GoとPerlは似ている！
  - とまでは言えないけどとっつきやすいと思う
- GoもPerlも適材適所で使っていけるとよさそうですね

おわり

質問まだあればどうぞ