Programando em Go

Crie aplicações com a linguagem do Google

Casa do Código

CAIO FILIPINI

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-66250-49-7

EPUB: 978-85-5519-131-2

MOBI: 978-85-5519-132-9

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

CRÉDITOS

O Gopher utilizado na capa deste livro é criação de Renee French (http://reneefrench.blogspot.com/) e licenciado sob Creative Commons Attributions 3.0 (https://creativecommons.org/licenses/by/3.0/).

PREFÁCIO

Go tem sido fundamental no meu dia a dia escrevendo programas concorrentes e plataformas (*systems programming*). É uma linguagem que favorece a criação de programas paralelos, leves, rápidos, simples de entender, de distribuir (um simples binário compilado estaticamente) e de manter.

Honestamente, porém, minhas primeiras impressões gerais sobre a linguagem não foram das melhores. A ausência de funcionalidades encontradas em outras linguagens mais sofisticadas me incomodava. Algumas bem polêmicas e controversas, como a de tipos genéricos e o tratamento de erros simples sem exceções. Não é incomum ter que escrever uma ou outra linha de código a mais em Go do que seria necessário em outra linguagem. Não é uma linguagem otimizada para programas curtos, mas sim para programas escaláveis.

No início, o modo com que Go lida com concorrência e paralelismo foi o que despertou meu interesse na linguagem. *Channels* e *goroutines* são primitivas extremamente poderosas que continuam influenciando bastante a forma com que escrevo programas concorrentes, inclusive em outras linguagens. E eu tenho certeza de que vão influenciar a forma com que você escreve programas também.

O resto não parecia ter nada de especial quando comparado com outras linguagens. Mas, com o tempo, fui aprendendo a apreciar a simplicidade de Go. Demorou um pouco até me cair a ficha de que a ausência de funcionalidades que trariam

complexidade foi (e continua sendo) uma decisão explícita de seus principais mantenedores. Isso faz com que Go seja uma linguagem relativamente fácil de aprender e geralmente existe apenas uma (ou poucas) forma(s) clara(s) de se resolver os problemas e escrever código em Go, a maioria delas usando apenas o que já está disponível na biblioteca padrão.

Quando me dá vontade de reclamar de que Go não tem essa ou aquela funcionalidade, lembro-me o quão simples é ler e entender programas escritos em Go. Lembro-me também de quanta discussão desnecessária se evita no dia a dia com outros programadores sobre como código deveria ser escrito. Uma vez que se entende e aceita a mentalidade por trás da linguagem, times conseguem focar em escrever código que resolve problemas de verdade de forma simples de manter, em vez de gastar tempo discutindo preferências pessoais e de estilo de código.

Go, na minha opinião, representa um ótimo balanço entre pragmatismo e funcionalidades. É uma escolha perfeita para programas que precisam sobreviver por muito tempo, na mão de muitas pessoas e times diferentes.

Já tive o prazer de trabalhar diretamente com o Caio, que é um excelente programador e faz realmente um bom trabalho em apresentar a linguagem.

Bom proveito e feliz programação!

Fabio Kung

AGRADECIMENTOS

À minha esposa Gabriela, pela paciência e todo o apoio.

Ao Francisco Capuano, pelo exemplo e inspiração.

À minha avô Arlinda e ao meu tio Washington, por permitirem que eu pudesse estudar e me apaixonar pela minha profissão.

A toda a minha família por acreditar e confiar em mim, sempre.

E a todos que, de forma direta ou indireta, contribuíram para a escrita deste livro; em especial: Guilherme Conte, Anderson Leite, Reinaldo Braga, Bruno Grasselli, Luca Pette, Adriano Almeida e Paulo Silveira.

Sumário

Versão: 20.9.6

Capítulo 1

INTRODUÇÃO

Desenvolver um *software* nunca foi uma tarefa fácil. Escrever um programa, por menor que ele seja, exige altas doses de concentração, paciência e atenção aos detalhes. Por vezes os problemas parecem não ter solução. E nada disso parece ter mudado muito com o passar do tempo: novas linguagens de programação e novas ferramentas são criadas todos os dias por programadores ao redor do mundo, enquanto o número de desafios não diminui.

Ao contrário, numa época em que processadores com múltiplos núcleos (*multi-core processors*) estão presentes até mesmo em dispositivos móveis, programadores e administradores de sistemas sofrem para tentar usar ao máximo os recursos da máquina sem que haja impacto para seus usuários – tarefa que se torna mais difícil conforme o número de usuários cresce.

Não existe nenhuma linguagem de programação capaz de resolver todos estes problemas de forma fácil. No entanto, a linguagem Go surgiu num ambiente onde tais problemas são ainda mais extremos: o Google.

Apesar de bastante inspirada na linguagem C, Go possui características de mais alto nível, como abstrações para algumas

estruturas de dados, coleta de lixo (*garbage collection*) e *duck typing*, além de trazer uma abordagem moderna e elegante para a criação de aplicações concorrentes. Go também inclui uma extensa biblioteca padrão com ferramentas para comunicação em redes, servidores HTTP, expressões regulares, leitura e escrita de arquivos e muito mais.

O objetivo deste livro é apresentar ao leitor os recursos da linguagem Go e importantes partes da biblioteca padrão, sempre trazendo exemplos relevantes que demonstram o uso de cada recurso.

Resolver os problemas citados anteriormente é um desafio enorme, mas Go veio para tornar esta tarefa um pouco mais prazerosa.

1.1 POR QUE GO?

Go nasceu como um projeto interno no Google, iniciado em 2007 por Rob Pike, Ken Thompson e Robert Griesemer, e posteriormente lançado como um projeto de código aberto em novembro de 2009. Pike e Thompson já haviam trabalhado juntos no Bell Labs, o berço do Unix e do Plan 9 – o sistema operacional que deu origem às ferramentas de compilação usadas como base do desenvolvimento da linguagem Go.

No artigo (em inglês) "Go at Google: Language Design in the Service of Software Engineering" (http://talks.golang.org/2012/splash.article), Rob Pike atribui os longos períodos de compilação e a dificuldade em escalar o desenvolvimento de grandes aplicações como sendo os principais

motivadores para a criação da nova linguagem. Segundo ele, a maior parte do problema é a forma com que as linguagens C e C++ tratam o gerenciamento de dependências entre os diversos arquivos-fonte.

Dentro deste cenário, Go foi concebida com o objetivo de tornar o desenvolvimento de servidores no Google uma tarefa mais produtiva e eficiente.

Para evitar os problemas de compilação, Go implementa um controle rigoroso e inteligente de dependências, baseado na definição e uso de *packages* (pacotes).

Com uma sintaxe bastante limpa, se comparada com outras linguagens inspiradas em C – como C++ e Java –, Go permite a escrita de programas concisos e legíveis, além de facilitar bastante a escrita de ferramentas que interagem com o código-fonte. Bons exemplos de tais ferramentas são `go fmt` (formata o código de acordo com o guia de estilo da linguagem) e `go fix` (reescreve partes do código que usa APIs depreciadas para que usem as novas APIs introduzidas em versões mais recentes).

Go possui tipagem forte e estática, porém introduz uma forma curta de declaração de variáveis baseada em inferência de tipos, evitando redundância e produzindo código muito mais sucinto do que linguagens estaticamente tipadas tradicionais. Além disso, Go traz uma implementação de *duck typing* (se faz "quack" como um pato e anda como um pato, então provavelmente *é* um pato) baseada em interfaces, permitindo a criação de tipos bastante flexíveis.

Alguns tipos de coleção de dados como *slices* (listas de

tamanho dinâmico) e *maps* (dicionários de dados associativos) são nativos à linguagem, que também fornece um conjunto de funções embutidas para a manipulação destes tipos; *arrays* (listas de tamanho fixo) também estão disponíveis para casos em que um nível maior de controle é necessário.

Go suporta o uso de ponteiros (referências a endereços de memória), o que torna ainda mais fácil a criação de poderosos tipos customizados. Entretanto, aritmética sobre ponteiros não é suportada. Apesar de toda a flexibilidade, através do uso de um coletor de lixo (ou *garbage collector*), Go reduz drasticamente a complexidade no gerenciamento de memória das aplicações, tornando-as mais robustas e evitando vazamentos descuidados.

Go também fornece recursos que permitem a escrita de programas totalmente procedurais, orientados a objetos (através da definição de novos tipos e de funções que operam sobre tais tipos) ou funcionais – funções são membros de primeira classe em Go, que também suporta a criação de *closures* (funções que herdam o contexto de onde elas foram definidas).

A abordagem de Go para concorrência é um dos maiores diferenciais da linguagem. Inspirada no famoso *paper* de C. A. R. Hoare *Communicating Sequential Processes* [ref csp-hoare], Go implementa *goroutines* – processos extremamente leves que se comunicam através de *channels*, evitando o uso de memória compartilhada e dispensando o uso de travas, semáforos e outras técnicas de sincronização de processos.

Com todos estes recursos disponíveis, chegou a hora de ver como Go funciona na prática.

1.2 INSTALAÇÃO

O primeiro passo para começar a escrever programas em Go é instalar a linguagem e suas ferramentas. Os pacotes de distribuição estão disponíveis no site http://golang.org/dl/. No momento da escrita deste livro, a versão estável disponível é a `1.4`. Faça o download do pacote referente ao seu sistema operacional. A seguir você encontrará as instruções específicas para cada plataforma.

Linux

Abra um terminal, vá para o diretório onde você salvou o arquivo da distribuição (no caso do Linux, o nome do arquivo será parecido com `go1.4.linux-amd64.tar.gz`) e execute o comando a seguir:

```
sudo tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz
```

As ferramentas da linguagem Go agora estarão disponíveis no diretório `/usr/local/go`. Adicione o diretório `/usr/local/go/bin` à variável de ambiente `PATH` para ter acesso às ferramentas, independente do diretório onde você estiver no sistema. Você pode fazer isso adicionando a seguinte linha ao seu arquivo `$HOME/.profile`:

```
export PATH=$PATH:/usr/local/go/bin
```

Para verificar se a instalação está correta, execute o seguinte comando no terminal:

```
$ go version
```

Este comando deverá produzir uma saída similar à seguinte:

```
go version go1.4 linux/amd64
```

Mac OS

No Mac OS há duas formas de instalação. A mais simples é idêntica à no Linux, sendo a única diferença o nome do arquivo de distribuição:

```
sudo tar -C /usr/local -xzf go1.4.darwin-amd64-osx10.8.tar.gz
```

A outra opção é baixar o instalador automático `go1.4.darwin-amd64-osx10.8.pkg`. Execute-o e siga as instruções.

As duas formas disponibilizam as ferramentas instaladas no diretório `/usr/local/go`. Após a instalação, assim como no Linux, adicione o diretório `/usr/local/go/bin` à variável de ambiente `PATH` no seu arquivo `$HOME/.profile`:

```
export PATH=$PATH:/usr/local/go/bin
```

E verifique se a instalação está correta, executando o seguinte comando no terminal:

```
$ go version
```

Este comando deverá produzir uma saída similar ao que segue:

```
go version go1.4 darwin/amd64
```

Windows

No Windows, assim como no Mac OS, há duas formas de instalação. A forma recomendada é a utilização do instalador automático. Faça o download do arquivo `go1.4.windows-amd64.msi`, execute-o e siga as instruções passo a passo. Após a instalação, as variáveis de ambiente serão automaticamente configuradas e você poderá seguir para as instruções de pós-

instalação.

A forma alternativa é fazer o download do pacote `go1.4.windows-amd64.zip` e descompactá-lo no diretório `C:\Go`. Em seguida, adicione o diretório `C:\Go\bin` à variável de ambiente `PATH`.

Para ter certeza de que a instalação foi bem-sucedida, execute o seguinte comando no prompt:

```
C:\>go version
```

Este comando deverá produzir uma saída similar à seguinte:

```
go version go1.4 windows/amd64
```

Instalando Go em um diretório não-padrão

Todas as instruções apresentadas anteriormente partem do princípio de que a instalação foi feita no diretório padrão – `/usr/local/go` no Linux e Mac OS, e `C:\Go` no Windows. Caso você não tenha esta opção ou prefira realizar a instalação em um diretório diferente, algumas configurações adicionais são necessárias.

Para que tudo funcione como esperado, é preciso configurar a variável de ambiente `GOROOT` para referenciar o diretório escolhido para a instalação. Por exemplo, se a instalação foi feita no diretório `$HOME/go` no Linux, adicione a seguinte configuração ao seu `$HOME/.profile`:

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

No Windows, uma configuração semelhante deve ser feita.

Entretanto, para evitar problemas, é recomendável que o diretório padrão seja utilizado.

Pós-instalação

A partir deste momento, todos os exemplos de comandos serão escritos utilizando a notação Unix, assumindo que você tem uma instalação funcional da linguagem Go e que o diretório `GOROOT/bin` foi adicionado à variável `PATH` (sendo que `GOROOT` deve ser o diretório no qual Go foi instalada).

Para extrair o máximo das ferramentas disponíveis na instalação, precisamos criar um diretório de trabalho (*workspace*) onde todos os programas devem residir. A estrutura do workspace inclui três subdiretórios:

- `src` contém o código-fonte de todos os seus programas escritos em Go (organizados em pacotes), e também o código-fonte de pacotes de terceiros instalados em seu sistema;
- `pkg` contém objetos referentes aos pacotes;
- `bin` contém arquivos executáveis gerados no seu sistema.

Crie um workspace e adicione-o à variável de ambiente `GOPATH`:

```
$ mkdir $HOME/go
```

Para maior conveniência, configure a variável `GOPATH` no seu arquivo `$HOME/.profile` (ou nas configurações avançadas do seu Windows conforme mencionado anteriormente), e adicione também o diretório `GOPATH/bin` à variável de ambiente `PATH`:

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

1.3 O PRIMEIRO PROGRAMA EM GO

Programas em Go podem ser escritos em qualquer editor de textos que tenha suporte à codificação `UTF-8`. Os arquivos-fontes **devem** necessariamente ser salvos nesta codificação, caso contrário o compilador não será capaz de processá-los.

Agora podemos finalmente escrever nosso primeiro programa em Go. Crie um novo diretório chamado `cap1-ola` dentro de `$GOPATH/src`. No diretório recém-criado, utilize seu editor de textos favorito e crie um novo arquivo chamado `ola.go` com o seguinte conteúdo e salve o arquivo:

```
package main

import "fmt"

func main() {
    fmt.Println("Olá, Go!")
}
```

Para executar o programa, utilizaremos a ferramenta `go`. Vá para o diretório `$GOPATH/src` e execute o seguinte comando:

```
$ go run cap1-ola/ola.go
```

Após a execução do programa, o texto "Olá, Go!" deverá ser impresso no terminal.

Arquivos-fonte contendo código Go devem seguir sempre a mesma estrutura, dividida em três seções: primeiro, a declaração do pacote, seguida da declaração dos pacotes externos dos quais o arquivo-fonte depende, e por fim, o código referente ao programa

ou pacote sendo escrito.

Todo código Go deve obrigatoriamente existir dentro de um pacote (*package*), e todo programa em Go deve ter um pacote `main` contendo uma função `main()` que serve como ponto de partida do programa.

No exemplo anterior, importamos o pacote `fmt`, que contém funções para a formatação de `strings`, e utilizamos a função `Println` para mostrar o texto "Olá, Go!" para o usuário.

Repare que a função `main()` não recebe nenhum argumento e não retorna nenhum valor. Diferente de linguagens como Java e C, argumentos passados para um programa Go através da linha de comando não são automaticamente disponibilizados ao programa. Em vez disso, precisamos utilizar um pacote chamado `os`, que será apresentado posteriormente.

Agora já temos um ambiente de desenvolvimento funcional, e acabamos de escrever o primeiro programa em Go.

Nos capítulos *Explorando a sintaxe básica* e *Indo além: mais exemplos*, veremos alguns exemplos que apresentarão a sintaxe básica e alguns dos recursos da linguagem. Nos capítulos posteriores, entraremos em detalhes sobre as funcionalidades mais importantes.

CAPÍTULO 2

EXPLORANDO A SINTAXE BÁSICA

Agora que você já tem um ambiente capaz de compilar e executar programas em Go, vamos fazer um pequeno passeio sobre as principais características da linguagem com alguns exemplos guiados. O objetivo dos exemplos a seguir é familiarizar o leitor com os elementos mais comuns da sintaxe, as principais palavras reservadas e estruturas de controle, além de apresentar alguns idiomas comuns no desenvolvimento de aplicações em Go.

Todos os exemplos de código deste livro estão disponíveis para consulta no endereço https://github.com/caiofilipini/casadocodigo-go.

Os recursos apresentados nos exemplos serão explicados em maiores detalhes nos próximos capítulos.

2.1 ESTRUTURA DO CAPÍTULO

Este capítulo é baseado em dois exemplos.

O primeiro é um simples conversor de medidas que apresentará o uso de alguns elementos básicos da linguagem, como

as estruturas de controle `if` e `for`, o conceito de *slices* e tratamento básico de erros. Este exemplo pode ser encontrado na seção *Exemplo 1: conversor de medidas* mais adiante.

O segundo, encontrado na seção *Exemplo 2: quicksort e funções*, é uma implementação do algoritmo de ordenação *quicksort*, e introduz o uso de funções, recursividade e a função `append()`.

Antes de apresentar os exemplos, porém, veremos o funcionamento básico das expressões `if` e o que são *arrays* e *slices*.

2.2 IF E EXPRESSÕES LÓGICAS

Em Go, diferente de algumas linguagens, somente expressões de valor lógico verdadeiro ou falso podem ser utilizadas em conjunto com instruções `if`.

Estas expressões devem necessariamente ser do tipo `bool` – cujos únicos valores possíveis são `true` e `false` – e podem ser variáveis ou mesmo funções ou métodos, desde que retornem um valor do tipo `bool`.

2.3 ARRAYS E SLICES

Go possui dois tipos padrões para listas de dados: `arrays` e `slices`.

Um `array` é uma lista de tamanho fixo, similar a um `array` nas linguagens C ou Java.

Já um `slice` é uma camada de abstração criada com base nos `arrays` e pode crescer indefinidamente, proporcionando uma flexibilidade muito maior.

As diferenças entre eles serão discutidas em detalhes no capítulo *Coleções: arrays, slices e maps*.

2.4 EXEMPLO 1: CONVERSOR DE MEDIDAS

O primeiro exemplo é um simples conversor de medidas. Ele aceita como entrada uma lista de valores com sua unidade de medida, e produz uma lista de valores convertidos. Por questões didáticas, o conversor trabalha apenas com dois tipos de conversões: de graus Celsius para Fahrenheit, e de quilômetros pra milhas.

```
package main

import (
    "fmt"
    "os"
    "strconv"
)
```

A primeira seção do programa não é muito diferente do que já vimos anteriormente. Porém, como este exemplo é um pouco mais complexo, importamos o pacote `fmt`, mas também precisamos dos pacotes `os` e `strconv`.

O pacote `os` possui uma série de operações que lidam com o sistema operacional hospedeiro de forma independente, facilitando a escrita de aplicações multiplataformas. No exemplo a seguir, utilizaremos este pacote para ter acesso aos argumentos passados ao nosso programa via linha de comando, e também para

instruir o programa a interromper sua execução e retornar um código de erro adequado em casos excepcionais.

Já o pacote `strconv` fornece uma grande variedade de funções para a conversão de `strings` para outros formatos e vice-versa. Os argumentos recebidos via linha de comando são sempre tratados como `strings`, portanto, para que o nosso conversor seja capaz de realizar operações matemáticas sobre os valores, utilizaremos o pacote `strconv` para convertê-los em números decimais.

A seguir precisamos declarar a função `main()` que, conforme visto no capítulo anterior, não recebe argumentos e não retorna nenhum valor. O conteúdo desta função será discutido passo a passo.

```
func main() {
    // ...
}
```

Como em qualquer programa que depende da entrada de dados de usuários, a primeira coisa que o conversor faz é garantir que os argumentos passados via linha de comando estão em um formato minimamente são. Para isso, verificamos a quantidade de argumentos recebidos – presentes em `os.Args` – através do uso da função embutida `len()`.

`os.Args` contém uma lista (tecnicamente um *slice*) de todos os argumentos passados para o programa, sendo que por padrão o primeiro argumento **sempre** será o próprio nome do programa executado. Portanto, verificamos se `os.Args` possui pelo menos três argumentos – o nome do programa; pelo menos um valor a ser convertido; e a unidade referente aos valores – através da instrução

de controle `if`.

Caso a quantidade de argumentos fornecidos seja menor do que três, utilizamos a já conhecida função `Println` do pacote `fmt` para apresentar ao usuário uma mensagem de ajuda com o formato de entrada do programa, e logo depois utilizamos a função `os.Exit()` para abortar a execução do programa. Note que a função `os.Exit()` foi chamada com o valor `1` como argumento; o valor especificado é retornado como código de erro para o sistema operacional. Seguindo os padrões de programas Unix, qualquer valor diferente de `0` indica uma execução anormal.

```
if len(os.Args) < 3 {
    fmt.Println("Uso: conversor <valores> <unidade>")
    os.Exit(1)
}
```

Seguindo a execução quando três ou mais argumentos foram fornecidos, encontramos a declaração e atribuição de duas variáveis: `unidadeOrigem` e `valoresOrigem`. Em Go, quando uma variável é declarada e atribuída na mesma operação, não é preciso informar seu tipo – o compilador é inteligente o suficiente para adivinhá-lo baseado na operação de atribuição; ou seja, caso o valor `10` seja atribuído à variável `x`, o compilador sabe que o tipo de `x` é um número inteiro, ou `int` em Go.

Atribuímos à `unidadeOrigem` o último argumento passado, de acordo com o formato esperado. Para isso, acessamos a última posição do *slice* `os.Args`. Slices são listas indexadas e de tamanho variável em Go, sendo que o primeiro elemento possui índice `0`. Sendo assim, o índice do último elemento é sempre `n - 1`, sendo `n` o número total de elementos presentes no slice – no caso, `len(os.Args)-1`.

```
unidadeOrigem := os.Args[len(os.Args)-1]
```

A variável `valoresOrigem` recebe uma sublista dos argumentos, descartando o primeiro (o nome do programa) e o último (a unidade já atribuída à `unidadeOrigem`). Em Go, este tipo de operação é trivial e conhecida como *slicing* (fatiar, separar) – daí vem o nome dado ao tipo de listas dinâmicas. Assim, fatiamos a lista completa de argumentos, obtendo somente os valores que devemos converter: `os.Args[1 : len(os.Args)-1]`. O índice `1` refere-se ao segundo elemento, e `len(os.Args)-1` ao último elemento, que é desconsiderado no slice retornado.

```
valoresOrigem := os.Args[1 : len(os.Args)-1]
```

Uma vez que identificamos a unidade-origem e os valores a serem convertidos, precisamos descobrir qual é a unidade-destino e, consequentemente, qual fórmula de conversão deverá ser usada. Conforme mencionado anteriormente, nosso programa só sabe converter graus Celsius para Fahrenheit e quilômetros para milhas.

Declaramos a variável `unidadeDestino` como sendo do tipo `string`. Como seu conteúdo depende de uma verificação e, portanto, não podemos atribuir a ela um valor no momento da declaração, precisamos utilizar a palavra-chave `var` antes do nome da variável, e neste caso precisamos também informar seu tipo logo após o nome. Se você está acostumado com linguagens como C e Java, pode achar a ordem da declaração um pouco estranha – nestas linguagens, declara-se primeiro o tipo e depois o nome da variável. Repare como em Go a leitura da declaração fica mais fluida: declare uma `var` iável com nome `unidadeDestino` do tipo `string`.

```
var unidadeDestino string
```

Para atribuir o valor correto à `unidadeDestino`, verificamos o conteúdo de `unidadeOrigem`: caso seja a `string` `"celsius"`, atribuímos `"fahrenheit"`; caso seja `"quilometros"`, atribuímos `"milhas"`; caso seja qualquer outro valor desconhecido, informamos este fato ao usuário e interrompemos a execução do programa. Note que, desta vez, utilizamos a função `fmt.Printf`, que imprime na saída padrão uma `string` de acordo com o formato especificado, similar à função `printf` da linguagem C. Neste caso, especificamos o formato `%s não é uma unidade conhecida!`, onde `%s` será substituído pelo valor da variável `unidadeDestino`.

```
if unidadeOrigem == "celsius" {
    unidadeDestino = "fahrenheit"
} else if unidadeOrigem == "quilometros" {
    unidadeDestino = "milhas"
} else {
    fmt.Printf("%s não é uma unidade conhecida!",
        unidadeOrigem)
    os.Exit(1)
}
```

Agora que conhecemos a unidade-origem, os valores a serem convertidos e também a unidade-destino, vamos à conversão propriamente dita. Para converter todos os valores informados, precisamos percorrer o slice `valoresOrigem`, transformar cada valor em um número decimal, aplicar a fórmula sobre este número e imprimir o resultado.

Go possui apenas uma estrutura de repetição, `for`, que pode ser usada de diferentes formas de acordo com o contexto; no nosso caso, utilizamos `for` em conjunto com o operador `range` para obter acesso a cada elemento do slice `valoresOrigem`. O operador `range`, quando aplicado a um slice, retorna dois valores

para cada elemento: primeiro, o índice do elemento no slice, e depois, o elemento propriamente dito. Atribuímos o índice à variável `i` e o elemento à `v`.

```
for i, v := range valoresOrigem {
    // ...
}
```

Em seguida, utilizamos a função `ParseFloat()` do pacote `strconv` para converter a `string` em um número de ponto flutuante. Esta função recebe dois argumentos – o valor a ser convertido e a precisão do valor retornado (32 ou 64 bits) – e retorna dois valores: o valor convertido e um erro de conversão (que é `nil` quando o valor é convertido com sucesso). Caso um erro aconteça, informamos ao usuário (novamente utilizando `fmt.Printf` para mostrar corretamente o valor – `string`, representado como `%s` no formato – e sua posição na lista de argumentos – `int`, representado como `%d`) e interrompemos a execução do programa.

```
valorOrigem, err := strconv.ParseFloat(v, 64)
if err != nil {
    fmt.Printf(
        "O valor %s na posição %d não é um número válido!\n",
        v, i)
    os.Exit(1)
}
```

Com o valor correto em mãos, declaramos a variável `valorDestino` como sendo do tipo `float64`, verificamos qual é a unidade-origem e, aplicando a fórmula correspondente à unidade, atribuímos a ela o valor convertido.

```
var valorDestino float64

if unidadeOrigem == "celsius" {
    valorDestino = valorOrigem*1.8 + 32
```

```
} else {
    valorDestino = valorOrigem / 1.60934
}
```

Por fim, utilizamos novamente a função `fmt.Printf` para apresentar o valor convertido e sua unidade ao usuário. Repare que utilizamos `%.2f` para informar à função que o valor é do tipo `float` e deve ser arredondado e formatado com duas casas decimais.

```
fmt.Printf("%.2f %s = %.2f %s\n",
    valorOrigem, unidadeOrigem, valorDestino, unidadeDestino)
```

A seguir, você encontra o código completo do conversor. Para executá-lo, crie um novo diretório chamado `cap2-conversor` em `$GOPATH/src` e, dentro dele, crie um arquivo chamado `conversor.go` com o conteúdo:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func main() {
    if len(os.Args) < 3 {
        fmt.Println("Uso: conversor <valores> <unidade>")
        os.Exit(1)
    }

    unidadeOrigem := os.Args[len(os.Args)-1]
    valoresOrigem := os.Args[1 : len(os.Args)-1]

    var unidadeDestino string

    if unidadeOrigem == "celsius" {
        unidadeDestino = "fahrenheit"
    } else if unidadeOrigem == "quilometros" {
        unidadeDestino = "milhas"
```

```
    } else {
        fmt.Printf("%s não é uma unidade conhecida!",
            unidadeOrigem)
        os.Exit(1)
    }

    for i, v := range valoresOrigem {
        valorOrigem, err := strconv.ParseFloat(v, 64)
        if err != nil {
            fmt.Printf(
                "O valor %s na posição %d " +
                    "não é um número válido!\n",
                v, i)
            os.Exit(1)
        }

        var valorDestino float64

        if unidadeOrigem == "celsius" {
            valorDestino = valorOrigem*1.8 + 32
        } else {
            valorDestino = valorOrigem / 1.60934
        }

        fmt.Printf("%.2f %s = %.2f %s\n",
            valorOrigem, unidadeOrigem,
            valorDestino, unidadeDestino)
    }
}
```

Um exemplo da execução do programa:

```
$ go run cap2-conversor/conversor.go 32 27.4 -3 0 celsius
32.00 celsius = 89.60 fahrenheit
27.40 celsius = 81.32 fahrenheit
-3.00 celsius = 26.60 fahrenheit
0.00 celsius = 32.00 fahrenheit
```

Algumas das técnicas apresentadas neste exemplo também serão utilizadas nos próximos e, portanto, não serão explicadas novamente.

2.5 CRIANDO FUNÇÕES BÁSICAS

Antes de implementar o próximo exemplo, vejamos algumas formas básicas para a definição de funções:

```
func imprimirDados(nome string, idade int) {
    fmt.Printf("%s tem %d anos.", nome, idade)
}

func main() {
    imprimirDados("Fernando", 29)
}
```

A função `imprimirDados()` recebe dois argumentos, uma `string` representando o `nome`, e um `int` representando a idade, e imprime os dois argumentos através do uso da função `fmt.Printf()`. Assim como a função especial `main()`, `imprimirDados()` não retorna nenhum valor.

```
func soma(n, m int) int {
    return n + m
}

func main() {
    s := soma(3, 5)
    fmt.Println("A soma é", s)
}
```

Esta outra função simples, denominada `soma()`, recebe dois argumentos – `n` e `m` – do tipo `int` e retorna a soma dos dois, também do tipo `int`. Note que, como os dois argumentos são do mesmo tipo, podemos simplificar sua declaração separando os nomes dos argumentos com uma vírgula e especificando seu tipo uma única vez.

Agora que conhecemos o básico sobre funções, vamos à implementação do próximo exemplo.

2.6 EXEMPLO 2: QUICKSORT E FUNÇÕES

O segundo exemplo é uma implementação simples do famoso algoritmo de ordenação *quicksort*. A ideia básica deste algoritmo é:

1. eleger um elemento da lista como pivô e removê-lo da lista;
2. particionar a lista em duas listas distintas: uma contendo elementos menores que o pivô, e outra contendo elementos maiores;
3. ordenar as duas listas recursivamente;
4. retornar a combinação da lista ordenada de elementos menores, o próprio pivô, e a lista ordenada de elementos maiores.

Como a recursividade é naturalmente parte deste algoritmo, nossa implementação será baseada em uma função recursiva. Este exemplo é mais complexo do que o anterior, porém muito mais expressivo, e demonstra como algumas construções da linguagem ajudam a escrever programas bastante sucintos.

Implementando o quicksort

Nosso programa limita-se a ordenar números inteiros. Os números são recebidos como argumentos via linha de comando de forma muito similar ao exemplo do conversor de unidades; a diferença é que desta vez utilizamos a função `strconv.Atoi()` para convertê-los em números inteiros em vez de decimais.

Repare também que a declaração do slice `numeros`, que armazenará os números inteiros, é bastante diferente da forma que foi utilizada no conversor: aqui utilizamos a função nativa `make()` para criar e inicializar um slice do tipo `[]int` – um slice

de números inteiros – especificando também seu tamanho inicial como sendo o mesmo da lista recebida como argumento, o que é sempre uma boa ideia quando sabemos de antemão qual será o tamanho final do slice.

Depois de converter a entrada do programa para uma lista de números inteiros, chamamos uma função denominada `quicksort()`, passando como argumento a lista de números a ser ordenada e imprimindo a lista resultante.

```
func main() {
    entrada := os.Args[1:]
    numeros := make([]int, len(entrada))

    for i, n := range entrada {
        numero, err := strconv.Atoi(n)
        if err != nil {
            fmt.Printf("%s não é um número válido!\n", n)
            os.Exit(1)
        }
        numeros[i] = numero
    }

    fmt.Println(quicksort(numeros))
}
```

A função `quicksort()` é responsável pela implementação do algoritmo. Ela recebe como argumento um slice de números inteiros e retorna um slice ordenado, como podemos ver na assinatura a seguir:

```
func quicksort(numeros []int) []int {
    // ...
}
```

O primeiro passo é verificar se a lista de entrada está vazia ou contém apenas um número e, em caso positivo, retornar a própria lista. Esta condição é extremamente importante em funções

recursivas; é a chamada *condição de parada*, que previne que a função seja executada eternamente ou até que aconteça uma sobrecarga na pilha de execução (*stack overflow*).

```
if len(numeros) <= 1 {
    return numeros
}
```

O próximo passo é criar uma cópia do slice original para evitar que ele seja modificado. Utilizamos a função embutida `copy()` para copiar o conteúdo do slice `numeros` para o `n`, que é criado com o mesmo tamanho do slice original:

```
n := make([]int, len(numeros))
copy(n, numeros)
```

A partir deste ponto, manipularemos somente o novo slice `n`. Veremos a função `copy()` em maiores detalhes na seção *Slices* do capítulo *Coleções: arrays, slices e maps*.

Em seguida, fazemos a escolha do pivô. Neste caso, utilizamos uma das técnicas mais simples e escolhemos o elemento que se encontra mais ou menos no meio da lista. Armazenamos o índice – que será importante em breve – e o próprio pivô:

```
indicePivo := len(n) / 2
pivo := n[indicePivo]
```

Com o pivô em mãos, precisamos removê-lo da lista original. Faremos isso através do uso da função nativa `append()`. Ela adiciona um elemento ao final de um slice, e sua forma geral é:

```
novoSlice := append(slice, elemento)
```

Isso pode parecer um tanto estranho quando queremos de fato **remover** um elemento do slice. Entretanto, combinando o uso

`append()` com operações de *slice*, temos uma construção bastante poderosa e idiomática em Go:

```
n = append(n[:indicePivo], n[indicePivo+1:]...)
```

Primeiro, fatiamos o slice `n` do primeiro elemento até o pivô – `n[:indicePivo]` – e utilizamos este novo slice como base para a operação de `append()`; depois, fatiamos novamente `n`, partindo do elemento imediatamente posterior ao pivô até o último elemento disponível – `n[indicePivo+1:]` – e utilizamos este slice como valor a ser adicionado ao slice-base. É muito importante notar o uso das reticências ao final do segundo argumento: estamos informando que todos os elementos do segundo slice devem ser adicionados ao slice-base.

O resultado desta operação é uma lista que contém todos os elementos anteriores ao pivô, e todos os elementos posteriores a ele, **exceto** o próprio pivô. Missão cumprida!

O próximo passo do algoritmo é particionar o slice de números em dois novos slices – um contendo todos os elementos menores ou iguais ao pivô, e outro contendo somente os elementos maiores. Esta tarefa é delegada a uma outra função chamada `particionar()`, que será explicada em breve. Note que tiramos proveito do fato de que a função `particionar()` retorna dois valores distintos – dois slices:

```
menores, maiores := particionar(n, pivo)
```

O último passo é ordenar estes dois slices recursivamente e combinar os resultados com o pivô, e é exatamente o que a última linha da função `quicksort()` faz – novamente através do uso da função `append` ():

```
return append(
    append(quicksort(menores), pivo),
    quicksort(maiores)...)
```

Primeiro, chamamos `quicksort()` recursivamente para ordenar o slice `menores`; depois adicionamos o `pivo` ao resultado desta ordenação; em seguida, fazemos outra chamada recursiva a `quicksort()` para ordenar o slice `maiores`; e por fim, combinamos as duas listas ordenadas com `append()` de maneira similar à apresentada anteriormente e retornamos.

A seguir a listagem completa da função `quicksort()`:

```
func quicksort(numeros []int) []int {
    if len(numeros) <= 1 {
        return numeros
    }

    n := make([]int, len(numeros))
    copy(n, numeros)

    indicePivo := len(n) / 2
    pivo := n[indicePivo]
    n = append(n[:indicePivo], n[indicePivo+1:]...)

    menores, maiores := particionar(n, pivo)

    return append(
        append(quicksort(menores), pivo),
        quicksort(maiores)...)
}
```

Para finalizar, vamos analisar a assinatura da função `particionar()`:

```
func particionar(
    numeros []int,
    pivo int) (menores []int, maiores []int) {

    // ...
}
```

Esta função recebe dois argumentos – um slice `numeros []int` e um `pivo int` – e retorna dois valores – um slice `menores []int` contendo todos os números menores ou iguais ao `pivo` e um slice `maiores []int` contendo os números maiores que o `pivo`. A implementação completa é bastante trivial e também faz uso da função `append()` apresentada anteriormente.

```
func particionar(
    numeros []int,
    pivo int) (menores []int,maiores []int) {

    for _, n := range numeros {
        if n <= pivo {
            menores = append(menores, n)
        } else {
            maiores = append(maiores, n)
        }
    }

    return menores, maiores
}
```

Um detalhe importante e novo que aparece na função `particionar()` é a presença do símbolo `_`, conhecido como *identificador vazio* (ou *blank identifier*). Como já vimos anteriormente, o operador `range`, quando utilizado para iterar sobre um slice, retorna sempre dois valores: o índice do elemento e o próprio elemento. Muitas vezes não precisamos dos dois valores, porém em Go uma variável declarada e não utilizada causa um erro de compilação. Para resolver este problema, utiliza-se o identificador vazio para ignorar um valor que não será usado. Isto se aplica a qualquer operação que retorne múltiplos valores e não se restringe apenas ao operador `range`.

A seguir você encontra a listagem completa da nossa

implementação de quicksort. Crie um diretório chamado `cap2-quicksort` logo abaixo do diretório `$GOPATH/src`, e dentro dele crie o arquivo `quicksort.go` com o conteúdo a seguir:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func main() {
    entrada := os.Args[1:]
    numeros := make([]int, len(entrada))

    for i, n := range entrada {
        numero, err := strconv.Atoi(n)
        if err != nil {
            fmt.Printf("%s não é um número válido!\n", n)
            os.Exit(1)
        }
        numeros[i] = numero
    }

    fmt.Println(quicksort(numeros))
}

func quicksort(numeros []int) []int {
    if len(numeros) <= 1 {
        return numeros
    }

    n := make([]int, len(numeros))
    copy(n, numeros)

    indicePivo := len(n) / 2
    pivo := n[indicePivo]
    n = append(n[:indicePivo], n[indicePivo+1:]...)

    menores, maiores := particionar(n, pivo)

    return append(
```

```
        append(quicksort(menores), pivo),
        quicksort(maiores)...)
}

func particionar(
    numeros []int,
    pivo int) (menores []int, maiores []int) {

    for _, n := range numeros {
        if n <= pivo {
            menores = append(menores, n)
        } else {
            maiores = append(maiores, n)
        }
    }

    return menores, maiores
}
```

Para ver este exemplo em execução, utilize o comando a seguir:

```
$ go run cap2-quicksort/quicksort.go 10 30 41 53 78 12 19 22
[10 12 19 22 30 41 53 78]
```

Neste capítulo, aprendemos a utilizar o básico das estruturas de controle `if` e `for`. Também vimos o que são `arrays` e `slices`, e aprendemos a criar nossas próprias funções.

Os tipos de coleção disponíveis em Go, incluindo `arrays` e `slices`, serão discutidos em detalhes no capítulo *Coleções: arrays, slices e maps*. As funções e todos os seus recursos serão apresentados no capítulo *Funções*.

CAPÍTULO 3

INDO ALÉM: MAIS EXEMPLOS

No capítulo anterior, aprendemos alguns elementos básicos da sintaxe e algumas das construções mais fundamentais em Go. Neste capítulo avançaremos um pouco mais através de exemplos que utilizam `maps` (dicionários de dados) e tipos customizados.

3.1 EXEMPLO 3: MAPAS E ESTATÍSTICAS

Um `map` – também conhecido como array associativo ou dicionário – é uma coleção de pares *chave/valor* sem ordem definida, em que cada chave é única e armazena um único valor.

O exemplo a seguir utiliza um `map` para contar a frequência de palavras com as mesmas iniciais. Recebemos a lista de palavras via linha de comando, assim como já fizemos nos exemplos anteriores. Armazenamos a lista em um slice denominado `palavras`, chamamos a função `colherEstatisticas()` utilizando o slice `palavras` como argumento, armazenamos as estatísticas numa variável e, por fim, chamamos a função `imprimir()` para imprimi-las.

```
package main
```

```
import (
    "fmt"
    "os"
    "strings"
)

func main() {
    palavras := os.Args[1:]

    estatisticas := colherEstatisticas(palavras)

    imprimir(estatisticas)
}
```

Nenhuma novidade até aqui, exceto pelo fato de que importamos o pacote `strings`. Este pacote contém uma gama de funções para manipulação de `strings`.

A função `colherEstatisticas()` possui a seguinte assinatura:

```
func colherEstatisticas(palavras []string) map[string]int
```

Repare que o tipo de retorno é `map[string]int`: um `map` com chaves do tipo `string` e valores do tipo `int`, ideal para armazenar as estatísticas que desejamos. Para cada item armazenado, a chave é a letra inicial da palavra e o valor é a quantidade de palavras com esta inicial.

Dentro da função declaramos o `map` que irá armazenar as estatísticas e damos a ele o nome `estatisticas`. Para inicializar um `map` utilizamos a função `make()`, similar à inicialização de slices:

```
estatisticas := make(map[string]int)
```

Em seguida, para cada palavra, extraímos sua letra inicial – `palavra[0]` – e, utilizando a função `ToUpper()` do pacote

`strings`, convertemos a inicial para maiúscula e a armazenamos na variável `inicial`. Desta forma, garantimos que não haja distinção de palavras escritas em caixa alta ou baixa.

Com a inicial em mãos, procuramos no mapa `estatisticas` uma entrada cuja chave seja esta inicial:

```
contador, encontrado := estatisticas[inicial]
```

Para acessar um valor em um `map`, utilizamos uma notação similar ao acesso de valores em um slice, porém utilizamos a *chave* do valor entre os colchetes – no nosso caso, a `string` representando a letra inicial. Esta operação retorna dois valores: o valor armazenado sob aquela chave e um `bool` indicando se a chave existe ou não no `map`. Atribuímos estes dois valores às variáveis `contador` e `encontrado`, respectivamente. Caso já exista um contador para a `inicial` (`encontrado` possui o valor `true`), incrementamos a contagem (`contador + 1`) e a atualizamos no mapa; caso contrário, é a primeira ocorrência de uma palavra com esta inicial e, portanto, armazenamos a contagem inicial `1`.

Após iterar sobre todas as palavras, nosso mapa `estatisticas` possui a contagem completa, e então ele é usado como valor de retorno para a função `colherEstatisticas()`.

```
if encontrado {
    estatisticas[inicial] = contador + 1
} else {
    estatisticas[inicial] = 1
}

return estatisticas
```

A função `imprimir()` recebe o mapa contendo as estatísticas

e simplesmente itera sobre todas as entradas, imprimindo as estatísticas para cada inicial:

```
func imprimir(estatisticas map[string]int) {
    fmt.Println("Contagem de palavras iniciadas em cada letra:")

    for inicial, contador := range estatisticas {
        fmt.Printf("%s = %d\n", inicial, contador)
    }
}
```

Veja como iterar sobre todas as entradas de um `map` é uma tarefa trivial quando utilizamos o operador `range`: como cada entrada no mapa é, por definição, um par *chave, valor*, o retorno de `range` se encaixa perfeitamente. É importante ressaltar que, em Go, a ordem de iteração sobre `maps` é **aleatória** e, portanto, não se deve confiar nela. No capítulo seguinte, estudaremos `maps` mais a fundo e veremos como garantir a ordem desejada nestes casos.

A seguir você encontra a listagem completa do programa. Crie o arquivo `cap3-maps/maps.go` com o seguinte conteúdo:

```
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {
    palavras := os.Args[1:]

    estatisticas := colherEstatisticas(palavras)

    imprimir(estatisticas)
}
```

```
func colherEstatisticas(palavras []string) map[string]int {
    estatisticas := make(map[string]int)

    for _, palavra := range palavras {
        inicial := strings.ToUpper(string(palavra[0]))
        contador, encontrado := estatisticas[inicial]
        if encontrado {
            estatisticas[inicial] = contador + 1
        } else {
            estatisticas[inicial] = 1
        }
    }

    return estatisticas
}

func imprimir(estatisticas map[string]int) {
    fmt.Println("Contagem de palavras iniciadas em cada letra:")

    for inicial, contador := range estatisticas {
        fmt.Printf("%s = %d\n", inicial, contador)
    }
}
```

E um exemplo da entrada e saída:

```
$ go run cap3-maps/maps.go \
Lorem ipsum dolor sit amet leo eu velit ante sagittis dolor \
turpis dis
Contagem de palavras iniciadas em cada letra:
L = 2
I = 1
D = 3
S = 2
A = 2
E = 1
V = 1
T = 1
```

3.2 EXEMPLO 4: PILHAS E TIPOS CUSTOMIZADOS

Definir tipos customizados traz inúmeros benefícios à legibilidade e robustez de um programa. Go possui algumas funcionalidades que facilitam esta tarefa, e veremos mais sobre este assunto no capítulo *Criando novos tipos*.

O exemplo a seguir apresenta uma implementação simples de uma pilha de objetos através de um tipo customizado.

Desta vez não receberemos argumentos via linha de comando, simplesmente empilhamos e desempilhamos alguns valores e interagimos com a pilha chamando alguns outros métodos utilitários.

Para facilitar o entendimento da interface da pilha, veremos cada passo da função `main()` que interage com ela, e posteriormente, veremos a implementação de cada método.

Inicialmente, criamos uma instância da pilha e atribuímos o objeto retornado à variável `pilha`. Em seguida, imprimimos o resultado da chamada de métodos: `pilha.Tamanho()` e `pilha.Vazia()`, que retornam `0` e `true`, respectivamente.

```
pilha := Pilha{}

fmt.Println("Pilha criada com tamanho ", pilha.Tamanho())
fmt.Println("Vazia? ", pilha.Vazia())
```

Se você já programou em qualquer outra linguagem com suporte a orientação a objetos, não encontrou nenhuma grande novidade até aqui. Entretanto, repare que os nomes de todos os métodos que chamamos se iniciam com uma letra maiúscula. Go possui uma forma muito simples de controle de acesso: todo identificador iniciado com uma letra maiúscula dentro de um pacote é automaticamente exportado e visível fora do pacote. Na

prática, já utilizamos esta funcionalidade quando chamamos funções como `fmt.Println()`, por exemplo.

Uma pilha vazia não é muito útil. Vamos agora empilhar quatro objetos e verificar o `Tamanho()` da pilha e se ela continua `Vazia()`:

```
pilha.Empilhar("Go")
pilha.Empilhar(2009)
pilha.Empilhar(3.14)
pilha.Empilhar("Fim")

fmt.Println("Tamanho após empilhar 4 valores: ",
    pilha.Tamanho())
fmt.Println("Vazia? ", pilha.Vazia())
```

Veja que somos capazes de empilhar valores de tipos completamente diferentes: duas `strings`, um `int` e um `float64`. Sem nenhuma surpresa, as duas últimas linhas do trecho anterior imprimirão `4` e `false`, respectivamente. E agora que temos uma pilha cheia, que tal desempilhar todos os objetos?

Como vimos anteriormente, `for` é a única estrutura de repetição disponível em Go. Mas isto não significa que ela seja limitada. A seguir utilizamos uma forma diferente da iteração com `for`, similar à construção `while` em outras linguagens. Enquanto a pilha não estiver vazia (`for !pilha.Vazia()`), o bloco será executado. Dentro do bloco, desempilhamos um valor e o atribuímos à variável `v`, cujo valor é mostrado ao usuário, junto com os resultados de `pilha.Tamanho()` e `pilha.Vazia()` após cada operação de remoção. Note que o método `pilha.Desempilhar()` retorna dois valores. O segundo valor indica um erro que, neste momento, optamos por ignorar, atribuindo-o ao identificador vazio `_`.

```
for !pilha.Vazia() {
    v, _ := pilha.Desempilhar()
    fmt.Println("Desempilhando ", v)
    fmt.Println("Tamanho: ", pilha.Tamanho())
    fmt.Println("Vazia? ", pilha.Vazia())
}
```

Ao final da iteração, a pilha está vazia novamente. Poderíamos nos certificar deste fato chamando a função `pilha.Vazia()`. Porém, decidimos chamar o método `pilha.Desempilhar()` mais uma vez. Agora, no entanto, ignoramos o primeiro valor retornado e atribuímos o segundo à variável `err`. Por fim, se um erro foi retornado, ele é apresentado ao usuário.

```
_, err := pilha.Desempilhar()
if err != nil {
    fmt.Println(err)
}
```

Agora vamos à definição do tipo `Pilha` e seus métodos.

Diferente de outras linguagens com suporte à programação orientada a objetos, Go não possui o conceito de classes. Em vez disso, definimos estruturas de dados em forma de `structs`, construções semelhantes às `structs` da linguagem C, e um conjunto de funções – métodos, neste caso – que manipulam estes dados. A seguir temos a definição do novo tipo `Pilha`:

```
type Pilha struct {
    valores []interface{}
}
```

O novo tipo possui apenas um membro: um slice denominado `valores`, que armazena objetos do tipo `interface{}`. Este tipo é conhecido como *interface vazia* e descreve uma interface sem nenhum método. Qualquer tipo em Go implementa pelo menos zero métodos, portanto satisfaz a interface vazia. Na prática, isto

faz com que a nossa implementação de pilha seja capaz de armazenar objetos de **qualquer** tipo Go válido, conforme pudemos ver na implementação da função `main()`.

O slice `valores` foi intencionalmente declarado com a inicial minúscula, garantindo que ele não seja acessível em outro pacote.

Tendo definido o tipo `Pilha`, podemos começar a definir seus métodos. Começaremos com o simples método `Tamanho()`:

```
func (pilha Pilha) Tamanho() int {
    return len(pilha.valores)
}
```

A definição de um método é muito semelhante à definição de uma função, como já vimos anteriormente. A diferença marcante é que métodos definem um objeto *receptor*, que neste caso foi chamado de `pilha` e é do tipo `Pilha`, que deve ser especificado entre parênteses antes do nome do método. Assim, o método `Tamanho()` acessa o slice `pilha.valores` e retorna seu tamanho utilizando a função `len()`.

É importante observar que, como Go não possui o conceito de classes, seus métodos também não possuem um receptor implícito – conhecido como `self` ou `this` em outras linguagens. Isto facilita muito a implementação do *runtime* da linguagem e aumenta a clareza do código, já que dentro da definição do método só existe uma forma de acessar seu receptor.

O método `pilha.Vazia()` é trivial e simplesmente verifica se o tamanho atual da pilha é igual a zero:

```
func (pilha Pilha) Vazia() bool {
    return pilha.Tamanho() == 0
}
```

Os dois métodos apresentados até aqui são imutáveis – não possuem nenhum efeito colateral – e, portanto, bastante simples.

No caso dos métodos `Empilhar()` e `Desempilhar()`, desejamos alterar a pilha na qual tais métodos foram chamados. Em Go, argumentos de funções e métodos são **sempre** passados por cópia (com a exceção de slices, maps e channels, que são conhecidos como *reference types*, como veremos nos próximos capítulos). Por isso, quando precisamos alterar qualquer argumento – incluindo receptores de métodos – devemos declará-los como *ponteiros*. Vejamos a definição do método `Empilhar()`:

```
func (pilha *Pilha) Empilhar(valor interface{}) {
    pilha.valores = append(pilha.valores, valor)
}
```

Repare que o tipo do receptor foi definido como `*Pilha`, e indica que a variável `pilha` é um *ponteiro* para um objeto do tipo `Pilha`. Para empilhar um novo objeto, adicionamo-lo ao slice `pilha.valores` através do uso da função `append()` e alteramos o valor atual de `pilha.valores` para guardar o novo slice que contém o objeto adicionado.

De forma similar, o método `Desempilhar()` também define o receptor como sendo um ponteiro. Como vimos na definição da função `main()`, este método possui dois valores de retorno: o objeto desempilhado e um valor do tipo `error` que é retornado quando a pilha está vazia. Para isso, utilizamos o método `pilha.Vazia()` e, em caso de retorno positivo, criamos um novo erro através da função `errors.New()` e retornamos `nil` no lugar do objeto desempilhado, junto com o erro recém-criado.

Caso a pilha não esteja vazia, atribuímos o último objeto

empilhado à variável `valor`. Em seguida, atualizamos o slice `pilha.valores` com uma fatia do slice atual, incluindo todos os objetos empilhados com a exceção do último – que acabou de ser desempilhado. Finalmente, retornamos o objeto removido e `nil` no lugar do erro, indicando que o objeto foi desempilhado com sucesso.

```
func (pilha *Pilha) Desempilhar() (interface{}, error) {
    if pilha.Vazia() {
        return nil, errors.New("Pilha vazia!")
    }
    valor := pilha.valores[pilha.Tamanho()-1]
    pilha.valores = pilha.valores[:pilha.Tamanho()-1]
    return valor, nil
}
```

Confira a listagem completa da implementação da pilha. Utilize o conteúdo a seguir para criar o arquivo `cap3-pilha/pilha.go`:

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    pilha := Pilha{}
    fmt.Println("Pilha criada com tamanho ", pilha.Tamanho())
    fmt.Println("Vazia? ", pilha.Vazia())

    pilha.Empilhar("Go")
    pilha.Empilhar(2009)
    pilha.Empilhar(3.14)
    pilha.Empilhar("Fim")
    fmt.Println("Tamanho após empilhar 4 valores: ",
        pilha.Tamanho())
    fmt.Println("Vazia? ", pilha.Vazia())

    for !pilha.Vazia() {
```

```
        v, _ := pilha.Desempilhar()
        fmt.Println("Desempilhando ", v)
        fmt.Println("Tamanho: ", pilha.Tamanho())
        fmt.Println("Vazia? ", pilha.Vazia())
    }

    _, err := pilha.Desempilhar()
    if err != nil {
        fmt.Println("Erro: ", err)
    }
}

type Pilha struct {
    valores []interface{}
}

func (pilha Pilha) Tamanho() int {
    return len(pilha.valores)
}

func (pilha Pilha) Vazia() bool {
    return pilha.Tamanho() == 0
}

func (pilha *Pilha) Empilhar(valor interface{}) {
    pilha.valores = append(pilha.valores, valor)
}

func (pilha *Pilha) Desempilhar() (interface{}, error) {
    if pilha.Vazia() {
        return nil, errors.New("Pilha vazia!")
    }
    valor := pilha.valores[pilha.Tamanho()-1]
    pilha.valores = pilha.valores[:pilha.Tamanho()-1]
    return valor, nil
}
```

Um exemplo deste programa em execução:

```
$ go run cap3-pilha/pilha.go
Pilha criada com tamanho  0
Vazia?  true
Tamanho após empilhar 4 valores:  4
Vazia?  false
```

```
Desempilhando  Fim
Tamanho:  3
Vazia?  false
Desempilhando  3.14
Tamanho:  2
Vazia?  false
Desempilhando  2009
Tamanho:  1
Vazia?  false
Desempilhando  Go
Tamanho:  0
Vazia?  true
Erro:  Pilha vazia!
```

Neste capítulo, melhoramos nosso conhecimento a respeito dos recursos básicos da linguagem Go. Vimos a utilização básica de mapas para classificar valores sob chaves, e aprendemos a definir tipos customizados para criar abstrações mais complexas.

Mapas serão discutidos em detalhes no capítulo a seguir, na seção *Maps*, e aprenderemos ainda mais sobre outras formas de criar tipos customizados no capítulo *Criando novos tipos*.

Capítulo 4

COLEÇÕES: ARRAYS, SLICES E MAPS

Nos capítulos anteriores, vimos como coleções são essenciais para a solução de vários problemas. Agora veremos as coleções disponíveis em Go em maiores detalhes.

4.1 ARRAYS

Arrays em Go são coleções indexadas de valores do mesmo tipo e de tamanho fixo e invariável. O primeiro elemento do array possui índice `0`, e o último elemento é sempre `len(array) - 1`.

Para declarar um array, podemos utilizar uma das seguintes formas:

```
var a [3]int
numeros := [5]int{1, 2, 3, 4, 5}
primos := [...]int{2, 3, 5, 7, 11, 13}
nomes := [2]string{}

fmt.Println(a, numeros, primos, nomes)
```

O resultado da última linha seria o seguinte:

```
[0 0 0] [1 2 3 4 5] [2 3 5 7 11 13] [ ]
```

O array `a` foi declarado como `[3]int`, ou uma lista de 3 números inteiros. O tamanho de um array deve **sempre** ser especificado na declaração e faz parte do tipo do array – `[3]int` e `[5]int` são considerados tipos diferentes, ainda que ambos carreguem valores do mesmo tipo.

Apesar de não termos inserido nenhum valor em `a`, obtivemos `[0 0 0]` quando imprimimos seu conteúdo. Quando um array é declarado, seus elementos ganham automaticamente um valor inicial conhecido como *zero value* em Go. Este valor inicial varia de acordo com o tipo de dados definido para o array, da seguinte forma:

- `false` para valores do tipo `bool`;
- `0` para `ints`;
- `0.0` para `floats`;
- `""` (ou `string` vazia) para `strings`;
- `nil` para ponteiros, funções, interfaces, slices, maps e channels.

Algumas vezes sabemos de antemão quais serão os valores contidos num array, e podemos declará-los utilizando a forma literal, como em `numeros := [5]int{1, 2, 3, 4, 5}`. Nestes casos, para facilitar a vida do programador, pode-se substituir o tamanho do array na declaração pelo operador *ellipsis* (reticências), instruindo o compilador a calcular o tamanho do array com base na quantidade de elementos declarados, como fizemos em `primos := [...]int{2, 3, 5, 7, 11, 13}`.

Por fim, declaramos `nomes := [2]string{}` – um array de duas `strings` – utilizando a forma literal, porém sem nenhum valor declarado entre as chaves, fazendo com que os elementos

sejam inicializados com `""`. Esta declaração é idêntica à primeira forma utilizada (para declarar o array `a`) e poderia ser escrita também da seguinte forma:

```
var nomes [2]string
```

O tamanho de um array pode ser obtido através do uso da função `len()`. Por exemplo, a execução do código a seguir nos daria `3 5 6 2` como resultado:

```
fmt.Println(len(a), len(numeros), len(primos), len(nomes))
```

Podemos também criar arrays cujos valores são também arrays, ou arrays multidimensionais, de forma similar a arrays simples:

```
var multiA [2][2]int
multiA[0][0], multiA[0][1] = 3, 5
multiA[1][0], multiA[1][1] = 7, -2

multiB := [2][2]int{{2, 13}, {-1, 6}}

fmt.Println("Multi A:", multiA)
fmt.Println("Multi B:", multiB)
```

Repare que, na declaração literal do array `multiB`, não precisamos especificar o tipo dos arrays internos.

Imprimindo os valores de `multiA` e `multiB` teríamos o seguinte resultado:

```
Multi A: [[3 5] [7 -2]]
Multi B: [[2 13] [-1 6]]
```

Os arrays têm sua importância e seu papel, mas não são muito flexíveis, especialmente nos casos em que precisamos aumentar seu tamanho dinamicamente. É possível fazer isso com arrays, mas exige um grande trabalho manual de verificação de limites, alocação de novos arrays e cópia de conteúdo.

Em quase todos os casos, em Go é mais comum utilizar slices. A própria biblioteca padrão da linguagem utiliza slices em vez de arrays como parâmetros e tipos de retorno em sua API pública. A seguir veremos os benefícios e facilidades de se trabalhar com eles.

4.2 SLICES

Um slice é uma poderosa abstração criada em cima de arrays que introduz uma série de facilidades. Diferente de um array, no entanto, um slice possui tamanho variável e pode crescer indefinidamente.

Na prática, a utilização de slices é muito similar ao que acabamos de aprender sobre arrays. Para declarar um slice, podemos utilizar quase a mesma sintaxe da declaração de arrays, incluindo a forma literal, com a diferença de que não especificamos o tamanho do slice:

```
var a []int
primos := []int{2, 3, 5, 7, 11, 13}
nomes := []string{}

fmt.Println(a, primos, nomes)
```

Teríamos a seguinte saída impressa:

```
[] [2 3 5 7 11 13] []
```

Como não especificamos o tamanho dos slices `a` e `nomes` nem adicionamos nenhum valor a eles durante a declaração, ambos estão vazios após a inicialização.

Arrays e slices em Go possuem duas propriedades importantes: tamanho e capacidade – `len()` e `cap()`, respectivamente, são as funções utilizadas para inspecionar estas propriedades.

Um slice pode ser criado também através da função `make()`, que internamente aloca um array e retorna uma referência para o slice criado. Ela possui a seguinte assinatura:

```
func make([]T, len, cap) []T
```

`T` representa o tipo dos elementos do slice, `len` o tamanho inicial do array alocado e `cap` o tamanho total da área de memória reservada para o crescimento do slice. Por conveniência, pode-se omitir o último argumento, e neste caso Go assume por padrão o mesmo valor do tamanho. Vejamos alguns exemplos:

```
b := make([]int, 10)
fmt.Println(b, len(b), cap(b))

c := make([]int, 10, 20)
fmt.Println(c, len(c), cap(c))
```

Esse código imprimiria:

```
[0 0 0 0 0 0 0 0 0 0] 10 10
[0 0 0 0 0 0 0 0 0 0] 10 20
```

Note que os 10 primeiros elementos – 10 foi o tamanho especificado – dos dois slices foram inicializados com `0`, o *zero value* para `ints`, conforme vimos anteriormente.

Uma grande vantagem de utilizar slices – de fato, qualquer tipo criado através da função `make()` – em vez de arrays é que, quando usados como argumentos ou no retorno de funções, são passados por referência e não por cópia. Isto torna estas chamadas muito mais eficientes, pois o tamanho da referência será sempre o mesmo, independente do tamanho do slice.

Iteradores

Já iteramos sobre slices nos exemplos anteriores, e agora veremos todas as diferentes formas de iteração disponíveis em Go.

Sabemos que a única estrutura de repetição em Go é o `for`. Em sua forma mais básica, podemos especificar uma condição lógica e o bloco será executado enquanto a condição for verdadeira – similar à construção `while` em outras linguagens. Por exemplo:

```
a, b := 0, 10

for a < b {
    a += 1
}

fmt.Println(a)
```

Podemos ler esse código como *enquanto `a` for menor que `b`, incremente o valor de `a`*. Seu resultado seria a impressão do valor `10`, o valor de `a` após a execução do bloco `for`.

Também podemos utilizar o `for` com uma cláusula de inicialização, uma condição lógica e uma cláusula de incremento – a construção tradicional da linguagem C:

```
for i := 0; i < 10; i++ {
    // ...
}
```

É importante notar que, como a variável `i` não existia antes do `for`, seu escopo é limitado ao bloco; se tentarmos acessar seu valor após a execução do bloco, teremos um erro de compilação informando que `i` não foi definida. Para resolver esses casos, precisamos declarar a variável **antes** do `for`:

```
var i int

for i = 0; i < 10; i++ {
    // ...
```

```
}

fmt.Println(i)
```

Desta forma, `i` continua existindo após a execução do bloco `for`, e o programa imprimiria o valor `10`.

Qualquer elemento da cláusula `for` pode ser omitido, mas o ponto e vírgula é obrigatório – a não ser que o único elemento presente seja a condição lógica, como já vimos no primeiro exemplo de uso do `for`. Desconsiderando o escopo da variável de controle, todas as formas a seguir são equivalentes:

```
i := 0
for i < 10 {
    i += 1
}

for j := 0; j < 10; j++ {
    // ...
}

var k int
for k = 0; k < 10; {
    k += 1
}

l := 0
for ; l < 10; l++ {
    // ...
}
```

A forma mais comum de iterar sobre slices, porém, é utilizando o operador `range`, como já vimos nos capítulos anteriores. A sintaxe geral é:

```
for indice, valor := range slice {
    // ...
}
```

O operador `range` retorna o índice de cada elemento, começando em `0`, e uma **cópia** de cada valor presente no slice. Quando precisamos modificar os valores em um slice, ou estamos interessados somente nos índices, podemos simplesmente omitir o segundo valor na atribuição e acessar cada elemento através de seu índice:

```
numeros := []int{1, 2, 3, 4, 5}

for i := range numeros {
    numeros[i] *= 2
}

fmt.Println(numeros)
```

O código anterior itera sobre um slice chamado `numeros`, multiplicando cada valor por 2. Após a iteração, imprimimos o conteúdo de `numeros` e obtemos `[2 4 6 8 10]` como resultado.

Ao contrário, quando não precisamos dos índices dos valores, podemos ignorá-los atribuindo-os ao identificador vazio:

```
for _, elemento := range slice {
    // ...
}
```

Existem casos em que não estamos interessados nos valores de um laço de repetição, apenas na iteração propriamente dita. Até a versão 1.3 da linguagem Go, a única forma de atingir esse objetivo era ignorando o valor retornado pelo operador `range`:

```
numeros := []int{1, 2, 3, 4, 5}

for _ := range numeros {
    // ...
}
```

Para facilitar a escrita de laços como esse, a versão 1.4 introduziu uma nova forma simplificada para o `for` combinado com o operador `range`:

```
numeros := []int{1, 2, 3, 4, 5}

for range numeros {
    // ...
}
```

Esse código faria com que a iteração fosse executada exatamente 5 vezes (a quantidade de elementos no slice `numeros`).

A última forma de utilização do `for` é a forma conhecida como *loop infinito*, que em Go é simplesmente uma cláusula `for` sem nenhuma condição – Go assume `true` como valor padrão para a condição:

```
for {
    // loop infinito
}
```

Para sair da execução de um loop infinito, podemos utilizar o comando `break`.

A seguir, temos um exemplo que inicia um loop infinito, gera números aleatórios e sai do loop somente quando o número gerado for divisível por 42, ou seja, o resto da divisão do valor por 42 é 0 – `i%42 == 0`. Crie um novo arquivo chamado `cap4-loop-infinito/loop_infinito.go` com o seguinte conteúdo:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
```

```
)

func main() {
    rand.Seed(time.Now().UnixNano())
    n := 0

    for {
        n++

        i := rand.Intn(4200)
        fmt.Println(i)

        if i%42 == 0 {
            break
        }
    }

    fmt.Printf("Saída após %d iterações.\n", n)
}
```

Quando geramos números aleatórios, é sempre importante configurar o valor conhecido como *seed* (semente) do gerador. No exemplo anterior, utilizamos o *timestamp* atual no formato padrão do Unix – o número de nano segundos desde 1º de janeiro de 1970 – para garantir que, a cada execução do programa, o gerador de números aleatórios produza números diferentes da vez anterior.

Um exemplo da saída do programa:

```
$ go run cap4-loop-infinito/loop_infinito.go
1458
3816
2413
1179
594
1562
93
2911
1131
3008
2235
3910
```

```
2928
1470
Saída após 14 iterações.
```

Por padrão, `break` sai do loop mais próximo ao ponto em que o comando foi executado. Há casos nos quais temos loops `for` aninhados e desejamos quebrar o loop externo em vez do interno. Para resolver este problema, Go também dá suporte a blocos `for` nomeados. Vejamos um exemplo:

```
var i int

externo:
for {
    for i = 0; i < 10; i++ {
        if i == 5 {
            break externo
        }
    }
}

fmt.Println(i)
```

Este recurso também é importante quando temos, por exemplo, um bloco `switch` dentro de um bloco `for` . Como o comando `break` também é usado para sair do `switch` , precisamos especificar o nome dado ao bloco `for` se quisermos sair do loop. Vamos criar um programa para demonstrar o uso de laços nomeados. Crie um arquivo chamado `cap4-loop-nomeado/loop_nomeado.go` definindo um pacote `main` , importando o pacote `fmt` , e adicione o seguinte conteúdo à função `main()` :

```
var i int

loop:
for i = 0; i < 10; i++ {
    fmt.Printf("for i = %d\n", i)
```

```
    switch i {
    case 2, 3:
        fmt.Printf("Quebrando switch, i == %d.\n", i)
        break
    case 5:
        fmt.Println("Quebrando loop, i == 5.")
        break loop
    }
}

fmt.Println("Fim.")
```

A saída do programa seria a seguinte:

```
$ go run cap4-loop-nomeado/loop_nomeado.go
for i = 0
for i = 1
for i = 2
Quebrando switch, i == 2.
for i = 3
Quebrando switch, i == 3.
for i = 4
for i = 5
Quebrando loop, i == 5.
Fim.
```

Fatiando slices

Fatiar (do inglês *to slice*) é o nome dado a operações que extraem partes de um slice ou de um array.

Para fatiar um slice ou array, utilizamos a seguinte forma:

```
novoSlice := slice[inicio : fim]
```

Sendo que `0 <= inicio <= fim <= len(slice)`. Qualquer combinação de índices inicial e final que não atenda a esta regra resulta em um erro de compilacão (*slice bounds out of range*, ou *limites do slice fora de alcance*).

Por questões práticas, podemos omitir o índice inicial e/ou final. Se o índice inicial for omitido, `0` é assumido como padrão; de maneira similar, `len(slice)-1` é assumido como valor padrão para o índice final:

```
fib := []int{1, 1, 2, 3, 5, 8, 13}
fmt.Println(fib)
fmt.Println(fib[:3])
fmt.Println(fib[2:])
fmt.Println(fib[:])
```

Esse código imprimiria:

```
[1 1 2 3 5 8 13]
[1 1 2]
[2 3 5 8 13]
[1 1 2 3 5 8 13]
```

Como citado anteriormente neste capítulo, um slice é uma abstração criada com base nos arrays. Entender a relação entre eles é essencial para trabalhar efetivamente com slices.

Sabemos que, quando um slice é criado, um array é alocado internamente. Quando fatiamos este slice e o atribuímos a uma nova variável, temos um novo slice que compartilha o **mesmo array interno** do original. Isto quer dizer que, quando um elemento comum aos dois slices é modificado, esta modificação é refletida no outro. Vejamos um exemplo:

```
original := []int{1, 2, 3, 4, 5}
fmt.Println("Original:", original)

novo := original[1:3]
fmt.Println("Novo:", novo)

original[2] = 13

fmt.Println("Original pós modificação:", original)
fmt.Println("Novo pós modificação:", novo)
```

A saída deste programa seria a seguinte:

```
Original: [1 2 3 4 5]
Novo: [2 3]
Original pós modificação: [1 2 13 4 5]
Novo pós modificação: [2 13]
```

Veja que, ao alterarmos o valor de `original[2]` para 13, o conteúdo dos dois slices foi modificado. Isto é verdade para **qualquer** slice criado fatiando um outro slice, independente da quantidade de indireções criadas. Veja mais um exemplo:

```
a := []string{"Paulo", "almoça", "em", "casa", "diariamente."}
b := a[:len(a)-1]
c := b[:len(b)-1]
d := c[:len(c)-1]
e := d[:len(d)-1]

e[0] = "Tiago"
fmt.Printf("%v\n%v\n%v\n%v\n%v\n", a, b, c, d, e)
```

E o resultado impresso seria:

```
[Tiago almoça em casa diariamente.]
[Tiago almoça em casa]
[Tiago almoça em]
[Tiago almoça]
[Tiago]
```

Todas as operações apresentadas anteriormente também são válidas em um array, e é importante mencionar que fatiar um array **sempre** resulta em um slice, nunca em outro array.

Inserindo valores

Todas as operações realizadas sobre slices são baseadas na função `append()`. Já vimos alguns exemplos de como usá-la. Sua assinatura é:

```
func append(slice []Tipo, elementos ...Tipo) []Tipo
```

Para inserir um novo valor *ao final* de um slice, utilizamos `append()` em sua forma mais básica:

```
s := make([]int, 0)
s = append(s, 23)

fmt.Println(s)
```

Executando este código, veríamos o valor `[23]` impresso.

Para inserir um novo valor *no começo* de um slice, precisamos inicialmente criar um novo slice contendo o valor que desejamos inserir, e depois adicionar todos os elementos do slice inicial ao recém-criado. Parece complicado, mas na prática é bastante simples:

```
s := []int{23, 24, 25}
n := []int{22}
s = append(n, s...)

fmt.Println(s)
```

Uma outra opção mais sucinta é criar o novo slice utilizando a forma literal na própria chamada à função `append()`:

```
s := []int{23, 24, 25}
s = append([]int{22}, s...)

fmt.Println(s)
```

As duas formas são semanticamente equivalentes e imprimiriam `[22 23 24 25]`. Repare que utilizamos novamente o operador *ellipsis* (reticências) para expandir o conteúdo do slice, garantindo que todos os valores sejam passados de forma individual à função `append()`.

Também é possível inserir um ou mais valores em qualquer posição *no meio* de um slice. Para isso, precisamos fatiar o slice até a posição onde desejamos inserir os novos valores – `s[:3]` – e utilizar esta fatia do slice original como primeiro argumento na chamada à `append()`. O segundo argumento deve ser, por sua vez, o resultado de uma segunda operação de `append()`, onde inserimos os valores restantes do slice original – `s[3:]...` – ao slice que contém os novos valores – `append(v, s[3:]...)...`. Repare novamente o uso do operador *ellipsis*, aparecendo duas vezes neste exemplo:

```
s := []int{11, 12, 13, 16, 17, 18}
v := []int{14, 15}
s = append(s[:3], append(v, s[3:]...)...)

fmt.Println(s)
```

Esse código imprimiria `[11 12 13 14 15 16 17 18]`.

Removendo valores

Para remover valores do começo de um slice não precisamos da função `append()`. Basta fatiar o slice ignorando os índices dos elementos que desejamos remover, e atribuir o novo slice à mesma variável. Por exemplo, podemos remover o primeiro valor de um slice da seguinte forma:

```
s := []int{20, 30, 40, 50, 60}
s = s[1:]

fmt.Println(s)
```

Assim, esse código imprimiria o slice `[30 40 50 60]`.

De maneira análoga, para remover valores do final de um slice, fatiamos este slice ignorando os índices dos elementos finais:

```
s := []int{20, 30, 40, 50, 60}
s = s[:3]

fmt.Println(s)
```

E agora o slice `s` teria o conteúdo `[20 30 40]`.

Por fim, para remover valores do meio de um slice, precisamos recorrer novamente à função `append()`, utilizando duas fatias do slice original como argumentos – a primeira incluindo elementos do início até o índice desejado, e a segunda começando do próximo elemento que nos interessa. Por exemplo, dado um slice `s := []int{10, 20, 30, 40, 50, 60}`, podemos remover os valores 30 e 40 da forma a seguir:

```
s := []int{10, 20, 30, 40, 50, 60}
s = append(s[:2], s[4:]...)

fmt.Println(s)
```

Ao executar esse código, teríamos o slice `[10 20 50 60]` impresso conforme o esperado.

Copiando slices

Até agora, todas as vezes em que manipulamos slices, utilizamos a função `append()` para **modificar** o slice original. No entanto, muitas vezes precisamos manter o estado do slice original intacto e manipular uma **cópia** dele. Para isso, Go também possui uma função embutida chamada `copy()`, com a seguinte assinatura:

```
func copy(destino, origem []Tipo) int
```

Para criar uma cópia de um slice, chamamos a função `copy()` da seguinte forma:

```
numeros := []int{1, 2, 3, 4, 5}
dobros := make([]int, len(numeros))

copy(dobros, numeros)

for i := range dobros {
    dobros[i] *= 2
}

fmt.Println(numeros)
fmt.Println(dobros)
```

Imprimindo o conteúdo dos dois slices, teríamos `[1 2 3 4 5]` e `[2 4 6 8 10]` como resultado, provando que o slice original não sofreu nenhuma alteração.

4.3 MAPS

Um *map*, ou mapa, é uma coleção de pares *chave-valor* sem nenhuma ordem definida. É a implementação em Go de uma estrutura de dados também conhecida como *hashtable*, dicionário de dados ou array associativo, entre outros nomes.

Qualquer tipo que suporte os operadores de igualdade (i.e. `==` e `!=`) pode ser usado como chave em um mapa. No entanto, as chaves em um mesmo mapa devem necessariamente ser do mesmo tipo. É importante mencionar também que as chaves são únicas – se armazenarmos dois valores distintos sob uma mesma chave, o primeiro valor será **sobrescrito** pelo segundo.

Os valores em um mapa também devem sempre ser do mesmo tipo, embora a interface vazia `interface{}` seja um tipo válido neste caso. Isto possibilita armazenar virtualmente **qualquer** valor sob uma chave, porém requer uma *asserção de tipo* (*type assertion*) quando os valores são recuperados.

Podemos declarar mapas utilizando a forma literal ou a função `make()`, de maneira muito semelhante à declaração de slices. Por exemplo, para declarar um mapa vazio com chaves do tipo `int` e valores do tipo `string`, as duas opções a seguir são equivalentes:

```
vazio1 := map[int]string{}
vazio2 := make(map[int]string)
```

A quantidade de valores armazenados em um mapa é flexível e pode crescer indefinidamente durante a execução de um programa, sendo que podemos especificar sua capacidade inicial quando sabemos de antemão quantos valores precisaremos armazenar. Isso pode ser importante quando sabemos que o mapa irá armazenar muitos valores, tornando o uso de memória mais eficiente e evitando problemas de performance. É recomendável especificá-la sempre que possível, e podemos fazê-lo simplesmente passando um segundo argumento à função `make()`:

```
mapaGrande := make(map[int]string, 4096)
```

A qualquer momento podemos inspecionar a quantidade de elementos que um mapa possui através da função `len()`. Por exemplo:

```
capitais := map[string]string{
    "GO": "Goiânia",
    "PB": "João Pessoa",
    "PR": "Curitiba"}

fmt.Println(len(capitais))
```

O código imprimiria o valor 3.

Populando mapas

Podemos popular um mapa utilizando literais no momento da

declaração e/ou atribuindo valores individualmente após a declaração:

```
capitais := map[string]string{
    "GO": "Goiânia",
    "PB": "João Pessoa",
    "PR": "Curitiba"}

capitais["RN"] = "Natal"
capitais["AM"] = "Manaus"
capitais["SE"] = "Aracaju"

fmt.Println(capitais)

populacao := make(map[string]int, 6)
populacao["GO"] = 6434052
populacao["PB"] = 3914418
populacao["PR"] = 10997462
populacao["RN"] = 3373960
populacao["AM"] = 3807923
populacao["SE"] = 2228489

fmt.Println(populacao)
```

O resultado da execução desse código seria o seguinte:

```
map[GO:Goiânia PB:João Pessoa PR:Curitiba RN:Natal
    AM:Manaus SE:Aracaju]
map[GO:6434052 PB:3914418 PR:10997462 RN:3373960
    AM:3807923 SE:2228489]
```

Para tornar o exemplo anterior um pouco mais real, vamos implementá-lo utilizando um novo tipo, chamado `Estado`. Crie um arquivo chamado `cap4-estados/estados.go` e adicione a definição do tipo `Estado` com a seguinte estrutura:

```
type Estado struct {
    nome      string
    populacao int
    capital   string
}
```

Agora, na função `main()`, podemos popular um mapa de estados desta forma:

```
estados := make(map[string]Estado, 6)

estados["GO"] = Estado{"Goiás", 6434052, "Goiânia"}
estados["PB"] = Estado{"Paraíba", 3914418, "João Pessoa"}
estados["PR"] = Estado{"Paraná", 10997462, "Curitiba"}
estados["RN"] = Estado{"Rio Grande do Norte", 3373960, "Natal"}
estados["AM"] = Estado{"Amazonas", 3807923, "Manaus"}
estados["SE"] = Estado{"Sergipe", 2228489, "Aracaju"}

fmt.Println(estados)
```

Esse código imprimiria o seguinte conteúdo (as quebras de linha e alguns espaços em branco foram deliberadamente adicionados para facilitar a visualização):

```
$ go run cap4-estados/estados.go
map[GO:{Goiás 6434052 Goiânia}
    PB:{Paraíba 3914418 João Pessoa}
    PR:{Paraná 10997462 Curitiba}
    RN:{Rio Grande do Norte 3373960 Natal}
    AM:{Amazonas 3807923 Manaus}
    SE:{Sergipe 2228489 Aracaju}]
```

Lookup: recuperando valores

A operação de recuperação de um valor em um mapa é conhecida como *lookup* e é muito similar à recuperação de valores de um índice específico em um slice – basta especificar a chave desejada entre colchetes:

```
sergipe := estados["SE"]

fmt.Println(sergipe)
```

Desta forma, a variável `sergipe` receberia o `Estado` presente no mapa e o valor `{Sergipe 2228489 Aracaju}` seria

impresso.

E o que aconteceria se tentássemos acessar o valor de um estado que não está presente no mapa?

```
fmt.Println(estados["SP"])
```

O valor `{ 0 }` seria impresso. Note que existe um espaço em branco antes e outro depois do valor 0. Quando tentamos recuperar o valor de uma chave que não está presente no mapa, o *zero value* do tipo armazenado é retornado. Muitas vezes isso pode causar comportamentos inesperados em um programa. Para evitar tais problemas, podemos testar se uma chave existe ou não:

```
saoPaulo, encontrado := estados["SP"]
if encontrado {
    fmt.Println(saoPaulo)
}
```

O segundo valor retornado pela operação de lookup é um `bool` que receberá o valor `true` caso a chave esteja presente no mapa, ou `false` caso contrário. Algumas vezes precisamos testar se uma dada chave existe, mas não necessariamente precisamos do valor correspondente. Neste caso, podemos ignorar o valor atribuindo-o ao identificador vazio:

```
_, encontrado := estados["RJ"]
if encontrado {
    // ...
}
```

Atualizando valores

Para atualizar valores existentes em um mapa, utilizamos a mesma sintaxe da inserção de um novo valor. Porém, como as chaves são únicas, o valor armazenado será atualizado. Por

exemplo:

```
idades := map[string]int{
    "João":    37,
    "Ricardo": 26,
    "Joaquim": 41,
}

idades["Joaquim"] = 42

fmt.Println(idades["Joaquim"])
```

O valor 42 seria impresso.

Removendo valores

Podemos remover valores presentes em um mapa utilizando a função embutida `delete()`, que possui a seguinte assinatura:

```
func delete(m map[TipoChave]TipoValor, chave TipoChave)
```

Por exemplo, caso desejássemos remover os dados do Estado do Amazonas no mapa de estados utilizado anteriormente, poderíamos fazê-lo desta forma:

```
delete(estados, "AM")
```

Iterando sobre mapas

Podemos utilizar o operador `range` para iterar sobre todas as entradas de um mapa:

```
for sigla, estado := range estados {
    fmt.Printf("%s (%s) possui %d habitantes.\n",
        estado.nome, sigla, estado.populacao)
}
```

Teríamos a saída:

```
Goiás (GO) possui 6434052 habitantes.
Paraíba (PB) possui 3914418 habitantes.
Paraná (PR) possui 10997462 habitantes.
Rio Grande do Norte (RN) possui 3373960 habitantes.
Amazonas (AM) possui 3807923 habitantes.
Sergipe (SE) possui 2228489 habitantes.
```

Um detalhe importante que já foi mencionado é que a ordem dos elementos em um mapa não é garantida. Em mapas pequenos como os que apresentamos até aqui, a ordem de inserção parece ter sido mantida, mas se tivermos um mapa com uma quantidade maior de elementos, veremos que isto não é sempre verdade. Por exemplo:

```
quadrados := make(map[int]int, 15)

for i := 1; i <= 15; i++ {
    quadrados[i] = i * i
}

for n, quadrado := range quadrados {
    fmt.Printf("%d^2 = %d\n", n, quadrado)
}
```

Um exemplo da execução desse código seria:

```
2^2 = 4
6^2 = 36
10^2 = 100
14^2 = 196
1^2 = 1
5^2 = 25
9^2 = 81
13^2 = 169
4^2 = 16
8^2 = 64
12^2 = 144
3^2 = 9
7^2 = 49
11^2 = 121
15^2 = 225
```

Muitas vezes precisamos apresentar os dados seguindo uma ordem definida. No caso do exemplo anterior, a leitura ficaria muito mais fácil se apresentássemos os valores ordenadamente. Em Go, a forma recomendada é manter uma estrutura de dados separada contendo as chaves ordenadas, iterando sobre esta estrutura e obtendo os valores correspondentes no mapa. Vamos adaptar o exemplo anterior para utilizar esta técnica, aproveitando as facilidades do pacote `sort` para ordenar a lista de chaves. Crie um arquivo chamado `cap4-mapa-ordenado/mapa_ordenado.go` com o seguinte conteúdo:

```
package main

import (
    "fmt"
    "sort"
)

func main() {
    quadrados := make(map[int]int, 15)

    for n := 1; n <= 15; n++ {
        quadrados[n] = n * n
    }

    numeros := make([]int, 0, len(quadrados))

    for n := range quadrados {
        numeros = append(numeros, n)
    }
    sort.Ints(numeros)

    for _, numero := range numeros {
        quadrado := quadrados[numero]
        fmt.Printf("%d^2 = %d\n", numero, quadrado)
    }
}
```

Agora sim, a saída do programa apareceria ordenada:

```
$ go run cap4-mapa-ordenado/mapa_ordenado.go
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
6^2 = 36
7^2 = 49
8^2 = 64
9^2 = 81
10^2 = 100
11^2 = 121
12^2 = 144
13^2 = 169
14^2 = 196
15^2 = 225
```

Repare no uso da função `sort.Ints()` para ordenar o slice contendo as chaves do mapa. Para maiores detalhes sobre as funções de ordenação disponíveis neste pacote, acesse a documentação oficial (em inglês) em http://golang.org/pkg/sort/.

Neste capítulo, vimos em detalhes os tipos nativos de coleção de dados, abstrações extremamente importantes e muito utilizadas no desenvolvimento de praticamente qualquer programa.

No capítulo a seguir, veremos como criar abstrações ainda mais poderosas através de tipos customizados.

CAPÍTULO 5

CRIANDO NOVOS TIPOS

Criar tipos customizados é uma ferramenta de abstração muito poderosa em linguagens de programação. Go, comparada a linguagens puramente orientadas a objetos, tem um suporte limitado às características deste paradigma. Por exemplo, não é possível criar uma hierarquia de tipos baseada em herança. Composição de tipos, no entanto, é suportada e encorajada em Go.

Dois tipos podem também implementar uma ou mais interfaces em comum, tornando-os ainda mais flexíveis. Veremos todas estas características nas próximas seções.

5.1 NOVOS NOMES PARA TIPOS EXISTENTES

Algumas linguagens dinâmicas – como Ruby – possibilitam que o programador altere o comportamento de qualquer classe ou objeto durante a execução de um programa. Em Go isto não é possível, mas podemos estender tipos existentes através da criação de novos tipos, por conveniência ou simplesmente para melhorar a legibilidade de um programa.

Para demonstrar esta técnica, vamos criar um novo tipo chamado `ListaDeCompras` baseado em um slice de `strings` :

```
type ListaDeCompras []string

func main() {
    lista := make(ListaDeCompras, 6)
    lista[0] = "Alface"
    lista[1] = "Pepino"
    lista[2] = "Azeite"
    lista[3] = "Atum"
    lista[4] = "Frango"
    lista[5] = "Chocolate"

    fmt.Println(lista)
}
```

O código anterior imprimiria `[Alface Pepino Azeite Atum Frango Chocolate]`.

Inicialmente, o tipo `ListaDeCompras` não parece muito útil. Porém, a grande vantagem em criar tipos customizados é que podemos estendê-los, algo impossível de se fazer com os tipos padrões da linguagem. Vamos estender o tipo `ListaDeCompras` para facilitar a nossa vida quando formos ao supermercado, definindo um método que separa os elementos da lista em categorias:

```
func (lista ListaDeCompras) Categorizar() (
    []string, []string, []string) {

    var vegetais, carnes, outros []string

    for _, e := range lista {
        switch e {
        case "Alface", "Pepino":
            vegetais = append(vegetais, e)
        case "Atum", "Frango":
            carnes = append(carnes, e)
        default:
            outros = append(outros, e)
        }
    }
```

```
    return vegetais, carnes, outros
}
```

Considerando a mesma lista de compras do exemplo anterior, poderíamos categorizá-la da seguinte forma:

```
vegetais, carnes, outros := lista.Categorizar()
fmt.Println("Vegetais:", vegetais)
fmt.Println("Carnes:", carnes)
fmt.Println("Outros:", outros)
```

Agora temos três slices distintos contendo os elementos da lista de compras categorizados, e executando este código teríamos o seguinte resultado:

```
Vegetais: [Alface Pepino]
Carnes: [Atum Frango]
Outros: [Azeite Chocolate]
```

Repare na forma com que o comando `switch` foi utilizado na implementação do método `Categorizar()`, verificando múltiplos valores – separados por vírgulas – em algumas das cláusulas `case`. Também utilizamos a cláusula `default` para criar um slice chamado `outros`, contendo qualquer valor que não foi reconhecido pelas cláusulas `case` especificadas anteriormente.

5.2 CONVERSÃO ENTRE TIPOS COMPATÍVEIS

Apesar do tipo `ListaDeCompras` ter sido criado com base no tipo `[]string`, na prática eles são diferentes e não são automaticamente intercambiáveis. Desta forma, não é possível utilizar um valor `ListaDeCompras` em que um `[]string` é esperado e vice-versa.

Para contornar este problema, precisamos realizar uma conversão de tipos (operação conhecida em Go como *type conversion*) manualmente. Podemos converter um valor para outro tipo utilizando o formato `T(x)`, onde `T` é o tipo destino e `x` o valor a ser convertido. Vejamos um exemplo convertendo valores entre `ListaDeCompras` e `[]string`. Crie um arquivo chamado `cap5-conversao/conversao.go` com o conteúdo:

```
package main

import "fmt"

type ListaDeCompras []string

func imprimirSlice(slice []string) {
    fmt.Println("Slice:", slice)
}

func imprimirLista(lista ListaDeCompras) {
    fmt.Println("Lista de compras:", lista)
}

func main() {
    lista := ListaDeCompras{"Alface", "Atum", "Azeite"}
    slice := []string{"Alface", "Atum", "Azeite"}

    imprimirSlice([]string(lista))
    imprimirLista(ListaDeCompras(slice))
}
```

Podemos agora executar este programa e verificar o resultado da conversão entre os tipos:

```
$ go run cap5-conversao/conversao.go
Slice: [Alface Atum Azeite]
Lista de compras: [Alface Atum Azeite]
```

5.3 CRIANDO ABSTRAÇÕES MAIS

PODEROSAS

Já vimos no capítulo anterior que podemos facilmente manipular slices utilizando apenas operações de *slicing* e a função `append()`. Podemos também criar uma camada de abstração sobre os slices e implementar uma lista genérica que armazena valores de qualquer tipo e possui operações para remover valores do início, do meio e do fim da lista. Primeiro vejamos a definição do tipo propriamente dito:

```
type ListaGenerica []interface{}
```

Criamos um tipo chamado `ListaGenerica` baseado em um slice que armazena valores do tipo `interface{}`, o que permite que a lista armazene valores de qualquer tipo.

Agora podemos criar os métodos, começando pelo método que remove valores de um índice específico:

```
func (lista *ListaGenerica) RemoverIndice(
    indice int) interface{} {

    l := *lista
    removido := l[indice]
    *lista = append(l[0:indice], l[indice+1:]...)
    return removido
}
```

Repare que definimos o tipo do receptor como `*ListaGenerica` – um ponteiro para um valor do tipo `ListaGenerica` – pois desejamos alterar o conteúdo da lista. Inicialmente definimos uma variável `l` para facilitar as operações sobre o ponteiro. Depois atribuímos o valor presente no índice especificado da lista à variável `removido`, que será retornada ao final do método. Tendo o valor desejado em mãos, alteramos a

lista para desconsiderar o elemento removido, e finalmente retornamos o valor removido.

Para implementar o método que remove valores do início da lista, podemos simplesmente chamar o método `RemoverIndice()` passando `0` como argumento:

```
func (lista *ListaGenerica) RemoverInicio() interface{} {
    return lista.RemoverIndice(0)
}
```

Isto funciona porque, substituindo o `indice` na operação que altera a lista por `0`, obtemos a expressão `append(l[0:0], l[1:]...)`, ou seja, adicionamos os elementos da lista iniciando no índice `1` à lista vazia retornada pela operação `l[0:0]`.

De maneira análoga, podemos implementar o método que remove valores do final da lista, novamente recorrendo ao método `RemoverIndice()`, agora passando o último índice da lista como argumento:

```
func (lista *ListaGenerica) RemoverFim() interface{} {
    return lista.RemoverIndice(len(*lista)-1)
}
```

O que ocorre neste caso é o inverso do que vimos com o método `RemoverInicio()`. Assumindo que o último índice presente na lista seja `5`, podemos substituir o valor de `indice` para obter a expressão `append(l[0:5], l[6:])`, ou seja, adicionamos a lista vazia retornada por `l[6:]` à lista da qual o valor do índice `5` foi removido.

A seguir está a listagem completa do programa. Crie um novo arquivo chamado `cap5-lista-generica/lista.go` com o conteúdo:

```
package main

import "fmt"

type ListaGenerica []interface{}

func (lista *ListaGenerica) RemoverIndice(
    indice int) interface{} {

    l := *lista
    removido := l[indice]
    *lista = append(l[0:indice], l[indice+1:]...)
    return removido
}

func (lista *ListaGenerica) RemoverInicio() interface{} {
    return lista.RemoverIndice(0)
}

func (lista *ListaGenerica) RemoverFim() interface{} {
    return lista.RemoverIndice(len(*lista)-1)
}

func main() {
    lista := ListaGenerica{
        1, "Café", 42, true, 23, "Bola", 3.14, false,
    }

    fmt.Printf("Lista original:\n%v\n\n", lista)

    fmt.Printf(
        "Removendo do início: %v, após remoção:\n%v\n",
        lista.RemoverInicio(), lista)
    fmt.Printf(
        "Removendo do fim: %v, após remoção:\n%v\n",
        lista.RemoverFim(), lista)
    fmt.Printf(
        "Removendo do índice 3: %v, após remoção:\n%v\n",
        lista.RemoverIndice(3), lista)
    fmt.Printf(
        "Removendo do índice 0: %v, após remoção:\n%v\n",
        lista.RemoverIndice(0), lista)
    fmt.Printf(
        "Removendo do último índice: %v, após remoção:\n%v\n",
        lista.RemoverIndice(len(lista)-1), lista)
```

```
}
```

Ao executar este programa, veremos o seguinte resultado:

```
$ go run cap5-lista-generica/lista.go
Lista original:
[1 Café 42 true 23 Bola 3.14 false]

Removendo do início: 1, após remoção:
[Café 42 true 23 Bola 3.14 false]

Removendo do fim: false, após remoção:
[Café 42 true 23 Bola 3.14]

Removendo do índice 3: 23, após remoção:
[Café 42 true Bola 3.14]

Removendo do índice 0: Café, após remoção:
[42 true Bola 3.14]

Removendo do último índice: 3.14, após remoção:
[42 true Bola]
```

5.4 STRUCTS

Uma *struct* (estrutura) é simplesmente uma coleção de variáveis que formam um novo tipo. Structs são importantes para agrupar dados relacionados, criando a noção de registros. Já vimos um exemplo básico disso na seção *Exemplo 4: pilhas e tipos customizados* do capítulo *Indo além: mais exemplos*, quando implementamos uma pilha baseada numa struct simples.

Se estivéssemos desenvolvendo um programa para extrair estatísticas de um arquivo de texto, poderíamos definir a struct `Arquivo` da seguinte forma:

```
type Arquivo struct {
    nome       string
    tamanho    float64
```

```
    caracteres int
    palavras   int
    linhas     int
}
```

Com a struct definida, podemos criar uma instância especificando o valor dos campos na ordem em que eles foram definidos e separados por vírgulas:

```
arquivo := Arquivo{"artigo.txt", 12.68, 12986, 1862, 220}

fmt.Println(arquivo)
```

Desta forma, teríamos o valor `{artigo.txt 12.68 12986 1862 220}` impresso.

Por questões de clareza, podemos especificar os nomes dos campos ao inicializar uma struct. Isto pode ser especialmente útil quando não precisamos atribuir valores a todos os campos no momento da inicialização, e também elimina a necessidade de declarar os valores na ordem em que foram definidos. Por exemplo:

```
codigoFonte := Arquivo{tamanho: 1.12, nome: "programa.go"}

fmt.Println(codigoFonte)
```

Este código produziria `{programa.go 1.12 0 0 0}` como resultado. Note que os valores que não foram especificados (`caracteres` , `palavras` e `linhas`) foram inicializados com `0` , o *zero value* para o tipo `int` .

Para acessar os valores em uma struct, utilizamos o operador `.` separando o nome da variável e o nome do campo acessado:

```
arquivo := Arquivo{"artigo.txt", 12.68, 12986, 1862, 220}
codigoFonte := Arquivo{tamanho: 1.12, nome: "programa.go"}
```

```
fmt.Printf("%s\t%.2fKB\n", arquivo.nome, arquivo.tamanho)
fmt.Printf("%s\t%.2fKB\n",
    codigoFonte.nome,
    codigoFonte.tamanho)
```

Esse programa imprimiria o seguinte resultado:

```
artigo.txt    12.68KB
programa.go    1.12KB
```

Em programas maiores, é muito comum inicializar uma struct e armazenar um ponteiro para ela numa variável que será manipulada posteriormente. Para isso, utilizamos o operador `&`, que retorna um ponteiro para a struct criada. Por conveniência, ao manipular um ponteiro para uma struct podemos omitir o operador `*`. Na prática, isso não influencia a forma como utilizamos a variável, como podemos ver a seguir:

```
ponteiroArquivo := &Arquivo{tamanho: 7.29, nome: "arquivo.txt"}

fmt.Printf("%s\t%.2fKB\n",
    ponteiroArquivo.nome,
    ponteiroArquivo.tamanho)
```

Qualquer valor armazenado em uma struct pode ser alterado – structs são tipos mutáveis:

```
codigoFonte := Arquivo{tamanho: 1.12, nome: "programa.go"}
fmt.Printf("%s\t%.2fKB\n",
    codigoFonte.nome,
    codigoFonte.tamanho)

codigoFonte.tamanho = 23.42
fmt.Printf("%s\t%.2fKB\n",
    codigoFonte.nome,
    codigoFonte.tamanho)
```

Executando esse código veríamos os seguintes valores impressos:

```
Antes: programa.go 1.12KB
Depois: programa.go 23.42KB
```

Em linguagens orientadas a objetos tradicionais, definimos novos tipos usando classes, que reunem dados (estado) e métodos (comportamento) em uma mesma unidade – a própria classe.

Em Go, utilizamos structs para definir novos tipos. Assim como fizemos anteriormente quando definimos tipos que estendem os tipos padrão da linguagem, podemos também definir métodos cujos receptores são structs, atingindo na prática um efeito muito semelhante à definição de uma classe.

Vamos enriquecer nosso tipo `Arquivo` criando dois métodos – `TamanhoMedioDePalavra()` e `MediaDePalavrasPorLinha()`:

```
func (arq *Arquivo) TamanhoMedioDePalavra() float64 {
    return float64(arq.caracteres) / float64(arq.palavras)
}

func (arq *Arquivo) MediaDePalavrasPorLinha() float64 {
    return float64(arq.palavras) / float64(arq.linhas)
}

func main() {
    arquivo := Arquivo{"artigo.txt", 12.68, 12986, 1862, 220}

    fmt.Printf("Média de palavras por linha: %.2f\n",
        arquivo.MediaDePalavrasPorLinha())
    fmt.Printf("Tamanho médio de palavra: %.2f",
        arquivo.TamanhoMedioDePalavra())
}
```

A implementação desses métodos é bastante trivial. Executando a função `main()` teríamos o seguinte resultado:

```
Média de palavras por linha: 8.46
Tamanho médio de palavra: 6.97
```

É importante notar a conversão de `int` para `float64` no momento da divisão, pois queremos que o resultado seja um número decimal. Quando dividimos dois `ints`, o resultado produzido é outro `int` e a parte decimal do número é truncada. Por exemplo, a divisão `2 / 3` produziria `0` como resultado.

5.5 INTERFACES

Uma interface é a definição de um conjunto de métodos comuns a um ou mais tipos. É o que permite a criação de tipos polimórficos em Go.

Java possui um conceito muito parecido, também chamado de interface. A grande diferença é que, em Go, um tipo implementa uma interface **implicitamente** – basta que este tipo defina **todos** os métodos desta interface, não havendo a necessidade de palavras-chave como *implements*, *extends* etc. Por exemplo, vamos criar uma interface chamada `Operacao`, que define um único método `Calcular()`:

```
type Operacao interface {
    Calcular() int
}
```

Podemos agora criar um tipo chamado `Soma` contendo dois operandos, e implementar o método `Calcular()`. Para facilitar a leitura dos resultados, vamos também implementar o método `String()`, que retorna uma representação textual da operação de soma. Este método é invocado automaticamente quando queremos apresentar um valor utilizando o pacote `fmt`. Para implementá-lo, vamos utilizar a função `Sprintf()` do próprio pacote `fmt`, que retorna uma `string` formatada:

```
type Soma struct {
    operando1, operando2 int
}

func (s Soma) Calcular() int {
    return s.operando1 + s.operando2
}

func (s Soma) String() string {
    return fmt.Sprintf("%d + %d", s.operando1, s.operando2)
}
```

Para utilizar este tipo na prática, poderíamos simplesmente declarar e instanciar uma variável do tipo `Soma`. Entretanto, prestando um pouco mais de atenção, podemos ver que a assinatura do método `Calcular()` do tipo `Soma` satisfaz a definição da interface `Operacao`, e isto é suficiente para dizer que a `Soma` *é uma* `Operacao`. Desta forma, podemos atribuir um valor `Soma` a uma variável definida como sendo do tipo `Operacao`:

```
var soma Operacao
soma = Soma{10, 20}

fmt.Printf("%v = %d\n", soma, soma.Calcular())
```

Repare na forma como utilizamos o marcador `%v` para obter a representação `string` de uma `Soma` – neste caso, `10 + 20`. Assim, executando este código teríamos o resultado `10 + 20 = 30`.

Um único tipo não é suficiente para demonstrar a flexibilidade de uma interface. Então, vamos agora criar um novo tipo chamado `Subtracao`, também implementando a interface `Operacao`. Em seguida criaremos uma lista de operações que serão calculadas independente do tipo. Primeiro, vamos ao código do tipo `Subtracao`:

```
type Subtracao struct {
    operando1, operando2 int
}

func (s Subtracao) Calcular() int {
    return s.operando1 - s.operando2
}

func (s Subtracao) String() string {
    return fmt.Sprintf("%d - %d", s.operando1, s.operando2)
}
```

Vamos agora criar uma lista de operações e calcular o valor acumulado de todas elas. Crie um novo arquivo chamado `cap5-operacoes/operacoes.go`. Adicione a ele as definições dos tipos `Operacao`, `Soma` e `Subtracao` da forma como foram apresentados anteriormente e, por fim, crie a seguinte função `main()`:

```
func main() {
    operacoes := make([]Operacao, 4)
    operacoes[0] = Soma{10, 20}
    operacoes[1] = Subtracao{30, 15}
    operacoes[2] = Subtracao{10, 50}
    operacoes[3] = Soma{5, 2}

    acumulador := 0
    for _, op := range operacoes {
        valor := op.Calcular()
        fmt.Printf("%v = %d\n", op, valor)
        acumulador += valor
    }
    fmt.Println("Valor acumulado =", acumulador)
}
```

Veja um exemplo da execução deste programa:

```
$ go run cap5-operacoes/operacoes.go
10 + 20 = 30
30 - 15 = 15
10 - 50 = -40
5 + 2 = 7
```

```
Valor acumulado = 12
```

5.6 DUCK TYPING E POLIMORFISMO

Duck typing (ou *tipagem pato*, em português literal) é a capacidade de um sistema de tipos de determinar a semântica de um dado tipo baseado em seus métodos e não em sua hierarquia. O nome tem origem no chamado *duck test* (ou *teste do pato*): se faz "quack" como um pato e anda como um pato, então provavelmente *é* um pato.

É mais comum encontrar esta propriedade em linguagens dinamicamente tipadas como Ruby ou Python. Porém, como Go não permite herança mas os tipos implementam interfaces implicitamente, na prática considera-se que Go implementa uma forma de *duck typing*, com a grande vantagem de ter, em tempo de compilação, a garantia de que o tipo esperado como argumento implementa a interface desejada.

Para facilitar o entendimento, vamos alterar o exemplo anterior e extrair o código que calcula o valor acumulado da lista de operações da função `main()` para uma nova função:

```
func acumular(operacoes []Operacao) int {
    acumulador := 0

    for _, op := range operacoes {
        valor := op.Calcular()
        fmt.Printf("%v = %d\n", op, valor)
        acumulador += valor
    }

    return acumulador
}
```

Desta forma, poderíamos reutilizar a função `acumular()`

passando como argumento um slice contendo quaisquer objetos que implementassem o método `Calcular()`, retornando um `int`.

Para demonstrar este fato, vamos adicionar ao arquivo `cap5-operacoes/operacoes.go` a definição de um novo tipo `Idade` implementando o método `Calcular()`. Vamos alterar também a função `main()` para chamar a função `acumular()` utilizando objetos deste tipo para obter a idade acumulada:

```
type Idade struct {
    anoNascimento int
}

func (i Idade) Calcular() int {
    return time.Now().Year() - i.anoNascimento
}

func (i Idade) String() string {
    return fmt.Sprintf("Idade desde %d", i.anoNascimento)
}

func main() {
    operacoes := make([]Operacao, 4)
    operacoes[0] = Soma{10, 20}
    operacoes[1] = Subtracao{30, 15}
    operacoes[2] = Subtracao{10, 50}
    operacoes[3] = Soma{5, 2}

    fmt.Println("Valor acumulado =", acumular(operacoes))

    idades := make([]Operacao, 3)
    idades[0] = Idade{1969}
    idades[1] = Idade{1977}
    idades[2] = Idade{2001}

    fmt.Println("Idades acumuladas =", acumular(idades))
}
```

Assumindo 2014 como o ano atual, um exemplo da execução

deste programa seria:

```
$ go run cap5-operacoes/operacoes.go
10 + 20 = 30
30 - 15 = 15
10 - 50 = -40
5 + 2 = 7
Valor acumulado = 12
Idade desde 1969 = 45
Idade desde 1977 = 37
Idade desde 1999 = 15
Idades acumuladas = 97
```

5.7 UM EXEMPLO DA BIBLIOTECA PADRÃO: IO.READER

A biblioteca padrão é sempre uma boa referência para boas práticas. Portanto, vamos recorrer a uma interface muito importante presente nela: `io.Reader`. Esta interface define um único método `Read()` com a seguinte assinatura:

```
func Read(p []byte) (n int, err error)
```

Esta função lê até `len(p)` bytes, armazena estes bytes em `p` e retorna o número de bytes lidos em caso de sucesso, ou um `error` em caso de erro na leitura.

Como pudemos ver anteriormente, qualquer objeto que implemente o método `Read()` com a mesma assinatura poderá ser usado em um contexto onde um `io.Reader` é esperado.

Vamos criar um exemplo simples de `io.Reader`. Para isso, vamos criar um novo arquivo chamado `cap5-leitor/leitor.go` declarando as seguintes dependências:

```
import (
    "fmt"
```

```
    "io"
)
```

Agora vamos definir um novo tipo simples chamado `LeitorDeStrings`, que não possui nenhum atributo e implementa somente o método `Read()`, retornando um slice de `bytes` – `[]byte` – contendo uma string ASCII simples, porém respeitando a interface `io.Reader`:

```
type LeitorDeStrings struct{}

func (l LeitorDeStrings) Read(p []byte) (int, error) {
    p[0] = 'A'
    p[1] = 'B'
    p[2] = 'C'
    p[3] = 'D'

    return len(p), nil
}
```

Em seguida vamos implementar uma função chamada `lerString()`, que recebe como argumento um `io.Reader`, chama seu método `Read()`, converte o slice de `bytes` resultante em uma `string` propriamente dita e a retorna:

```
func lerString(r io.Reader) string {
    p := make([]byte, 4)
    r.Read(p)
    return string(p)
}
```

Por fim, na função `main()`, vamos criar uma instância de `LeitorDeStrings` e utilizá-la como um `io.Reader` ao chamar a função `lerString()`, imprimindo seu resultado:

```
func main() {
    leitor := LeitorDeStrings{}
    fmt.Println(lerString(leitor))
}
```

A seguir você encontra a listagem completa do arquivo `cap5-leitor/leitor.go`:

```
package main

import (
    "fmt"
    "io"
)

type LeitorDeStrings struct{}

func (l LeitorDeStrings) Read(p []byte) (int, error) {
    p[0] = 'A'
    p[1] = 'B'
    p[2] = 'C'
    p[3] = 'D'

    return len(p), nil
}

func lerString(r io.Reader) string {
    p := make([]byte, 4)
    r.Read(p)
    return string(p)
}

func main() {
    leitor := LeitorDeStrings{}
    fmt.Println(lerString(leitor))
}
```

Um exemplo da saída produzida por este programa:

```
$ go run cap5-leitor/leitor.go
ABCD
```

Além do uso da interface `io.Reader`, há uma outra novidade no exemplo anterior: o uso de caracteres literais – que em Go são do tipo `rune`. Este tipo foi criado para representar um código Unicode em 32 bits – `rune` é um *alias* para o tipo `int32`.

De fato, a codificação Unicode é tão importante em Go que toda `string` e o próprio código-fonte são codificados como UTF-8 por padrão. Ken Thompson e Rob Pike, dois dos criadores da linguagem, foram também responsáveis pela implementação original do padrão UTF-8. Desta forma, qualquer caractere UTF-8 pode ser utilizado livremente em programas escritos em Go.

Neste capítulo, aprendemos a criar novos tipos e abstrações baseados em `structs` e `interfaces`. Também conhecemos um pouco sobre a importante interface `io.Reader`, presente na biblioteca padrão.

No capítulo seguinte, veremos em detalhes os recursos para a criação e manipulação de funções.

Capítulo 6

FUNÇÕES

Até agora já criamos diversas funções em diferentes contextos. Criamos inclusive uma função recursiva quando implementamos o algoritmo *quicksort* no capítulo*Explorando a sintaxe básica*. Também já vimos como criar objetos e adicionar métodos que definem seu comportamento.

Neste capítulo veremos em detalhes todos os recursos disponíveis em Go para escrever funções.

6.1 A FORMA BÁSICA

Apenas para relembrar, utilizamos a palavra-chave `func` para declarar uma nova função. Por exemplo, uma função que não recebe nenhum argumento e não retorna nenhum valor pode ser declarada da seguinte forma:

```
func ImprimirVersao() {
    fmt.Println("1.12")
}
```

Argumentos são declarados de maneira similar a variáveis, portanto, especificamos primeiro seu nome, e depois seu tipo:

```
func ImprimirSaudacao(nome string) {
    fmt.Printf("Olá, %s!\n", nome)
}
```

Múltiplos argumentos são separados por vírgulas:

```
func ImprimirDados(nome string, idade int) {
    fmt.Printf("%s, %d anos.\n", nome, idade)
}
```

E, caso os argumentos sejam do mesmo tipo, podemos especificá-lo uma única vez:

```
func ImprimirSoma(a, b int) {
    fmt.Println(a + b)
}
```

Isto funciona inclusive quando a função recebe mais argumentos de outros tipos. A declaração pode ser feita da seguinte forma:

```
func ImprimirSoma(a, b int, texto string) {
    fmt.Printf("%s: %d\n", texto, a + b)
}
```

Para retornar um valor, especificamos seu tipo ao final da declaração e utilizamos a palavra-chave `return` para retorná-lo:

```
func Somar(a, b int) int {
    return a + b
}
```

Podemos também especificar múltiplos valores de retorno, de forma bastante similar à lista de argumentos:

```
func PrecoFinal(precoCusto float64) (float64, float64) {
    fatorLucro := 1.33
    taxaConversao := 2.34

    precoFinalDolar := precoCustoDolar * fatorLucro

    return precoFinalDolar, precoFinalDolar * taxaConversao
}
```

6.2 VALORES DE RETORNO NOMEADOS

Os valores retornados por uma função podem ser nomeados no momento da declaração, fazendo com que eles estejam automaticamente disponíveis como variáveis no corpo da função. Desta forma, podemos utilizar a palavra-chave `return` sem especificar nenhum valor; os valores atuais das variáveis declaradas na definição da função serão retornados. Vejamos um exemplo, reescrevendo a função `PrecoFinal()` apresentada anteriormente:

```
func PrecoFinal(precoCusto float64) (
    precoDolar float64,
    precoReal float64) {

    fatorLucro := 1.33
    taxaConversao := 2.34

    precoDolar = precoCusto * fatorLucro
    precoReal = precoDolar * taxaConversao

    return
}

func main() {
    precoDolar, precoReal := PrecoFinal(34.99)

    fmt.Printf("Preço final em dólar: %.2f\n" +
            "Preço final em reais: %.2f\n",
        precoDolar, precoReal)
}
```

Repare que no corpo da função atribuímos os valores às variáveis `precoDolar` e `precoReal` diretamente utilizando o operador `=`, pois elas já foram declaradas na definição da função.

Nomear os valores de retorno é uma ótima maneira de documentar uma função. No exemplo modificado, definimos que a função retorna dois valores específicos – `precoDolar` e

`precoReal` – tornando sua intenção muito mais clara. No entanto, utilizar a palavra-chave `return` sem especificar os valores retornados dificulta o entendimento do código. Por isso é sempre recomendável especificar os valores retornados explicitamente, mesmo que eles já tenham sido nomeados na assinatura da função.

A seguir, vamos modificar novamente a função `PrecoFinal()` para deixar explícito quais são os valores retornados. Crie um novo arquivo chamado `cap6-retornos-nomeados/precos.go` com o conteúdo:

```
package main

import "fmt"

func PrecoFinal(precoCusto float64) (
    precoDolar float64,
    precoReal float64) {

    fatorLucro := 1.33
    taxaConversao := 2.34

    precoDolar = precoCusto * fatorLucro
    precoReal = precoDolar * taxaConversao

    return precoDolar, precoReal
}

func main() {
    precoDolar, precoReal := PrecoFinal(34.99)

    fmt.Printf("Preço final em dólar: %.2f\n" +
            "Preço final em reais: %.2f\n",
        precoDolar, precoReal)
}
```

Executando este programa, teríamos o seguinte resultado:

```
$ go run cap6-retornos-nomeados/precos.go
```

```
Preço final em dólar: 46.54
Preço final em reais: 108.90
```

6.3 ARGUMENTOS VARIÁVEIS

Uma função pode receber um número variável de argumentos. Funções deste tipo são conhecidas em Go como *variadic functions*. Já utilizamos algumas delas, como por exemplo `fmt.Printf()` e `append()`.

Para criar uma *variadic function*, devemos preceder o tipo do último (ou único) argumento com reticências. Na prática, as reticências indicam que a função pode receber zero ou mais argumentos do tipo especificado.

Vamos definir uma função que recebe uma lista variável de nomes de arquivos e cria cada um deles em um diretório temporário. Para isto, vamos utilizar algumas funções presentes no pacote `os`:

```
func CriarArquivos(dirBase string, arquivos ...string) {
    for _, nome := range arquivos {
        caminhoArquivo := fmt.Sprintf(
            "%s/%s.%s", dirBase, nome, "txt")

        arq, err := os.Create(caminhoArquivo)

        defer arq.Close()

        if err != nil {
            fmt.Printf("Erro ao criar arquivo %s: %v\n",
                nome, err)
            os.Exit(1)
        }

        fmt.Printf("Arquivo %s criado.\n", arq.Name())
    }
}
```

Note o uso das reticências na declaração do argumento `arquivos : arquivos ...string` .

A implementação da função é trivial. Percorremos a lista de nomes de arquivos recebida e, para cada nome de arquivo, compilamos seu caminho completo a partir do diretório especificado em `dirBase` e utilizamos a função `os.Create()` para criá-lo em disco. Essa função recebe um caminho completo de arquivo e cria um arquivo novo no caminho especificado. Caso o arquivo em questão já exista, seu conteúdo será totalmente apagado, resultando em um arquivo vazio.

Além de efetivamente criar o arquivo, a função `os.Create()` retorna um ponteiro para um objeto do tipo `os.File` representando o descritor do arquivo criado. Este descritor pode ser usado para realizar outras operações no arquivo. Neste exemplo, porém, apenas chamamos o método `Name()` para obter seu caminho completo.

A função `os.Create()` também pode retornar um erro caso não seja possível criar o arquivo. Se isto acontecer, a execução será interrompida e o programa será encerrado através da chamada à função `os.Exit()` .

Por fim, é importante notar o uso da palavra-chave `defer` . Ela é utilizada para instruir o ambiente de execução Go a realizar uma tarefa imediatamente antes de a função atual retornar. É bastante comum utilizar `defer` para garantir que os recursos alocados pela função sejam liberados ao final de sua execução. Neste exemplo, chamamos o método `Close()` no descritor do arquivo criado para ter certeza de que ele será devidamente fechado e liberado.

Como dito anteriormente, uma função com argumentos variáveis pode receber zero ou mais argumentos do tipo especificado. Para entender melhor, crie um novo arquivo chamado `cap6-variadic-functions/arquivos.go`. Adicione a ele a função `CriarArquivos()` apresentada anteriormente e, então, adicione a seguinte função `main()` para realizar algumas chamadas à função `CriarArquivos()` com listas de argumentos diferentes:

```
func main() {
    tmp := os.TempDir()

    CriarArquivos(tmp)
    CriarArquivos(tmp, "teste1")
    CriarArquivos(tmp, "teste2", "teste3", "teste4")
}
```

Inicialmente criamos a variável `tmp` e atribuímos a ela o valor retornado por `os.TempDir()`. Essa função retorna o caminho do diretório temporário utilizado pelo ambiente de execução Go. Em seguida, chamamos a função `CriarArquivos()` em três cenários diferentes: inicialmente sem nenhum nome de arquivo; depois com um único nome de arquivo; e por fim, com três nomes de arquivos distintos. Ao final da execução, o resultado esperado é que os arquivos `teste1.txt`, `teste2.txt`, `teste3.txt` e `teste4.txt` sejam criados em disco.

A seguir encontramos um exemplo da saída da execução deste programa no Linux:

```
$ go run cap6-variadic-functions/arquivos.go
Arquivo /tmp/teste1.txt criado.
Arquivo /tmp/teste2.txt criado.
Arquivo /tmp/teste3.txt criado.
Arquivo /tmp/teste4.txt criado.
```

Para verificarmos que os arquivos foram realmente criados, podemos listar os arquivos do diretório temporário:

```
$ ls -l /tmp/
-rw-r--r--  1 user  group     0B May  1 10:56 teste1.txt
-rw-r--r--  1 user  group     0B May  1 10:56 teste2.txt
-rw-r--r--  1 user  group     0B May  1 10:56 teste3.txt
-rw-r--r--  1 user  group     0B May  1 10:56 teste4.txt
```

6.4 FUNÇÕES DE PRIMEIRA CLASSE

Funções em Go são *first-class citizens* (cidadãos de primeira classe). Isso significa que funções podem ser passadas como parâmetro para outras, ou utilizadas como valores de retorno, e até mesmo como membros de uma *struct* ou coleção.

Se você já trabalhou com funções e *callbacks* em JavaScript, *lambdas* em Ruby, Python ou Java (que finalmente introduziu expressões *lambda* na versão 8), ou escreveu programas em qualquer linguagem puramente funcional – Lisp, Haskell, Scala, Erlang ou Clojure, para citar alguns exemplos –, já deve estar familiarizado com os conceitos que serão apresentados a seguir.

6.5 FUNÇÕES ANÔNIMAS

Durante o desenvolvimento de uma aplicação é muito comum encontrarmos problemas que exigem algum tipo de manipulação de textos: substituição de caracteres acentuados, processamento de *templates* (modelos de documentos) etc.

Uma das formas mais poderosas de resolver este tipo de problema é o uso de expressões regulares. Go disponibiliza uma série de facilidades para trabalhar com elas no pacote `regexp` ,

basta adicioná-lo à lista de pacotes especificados na cláusula `import`.

Veja o exemplo a seguir:

```
func main() {
    texto := "Anderson tem 21 anos"
    expr := regexp.MustCompile("\\d")

    fmt.Println(expr.ReplaceAllString(texto, "3"))
}
```

Inicialmente criamos a expressão regular `expr` através da função `regexp.MustCompile()`. Essa função recebe a expressão desejada em formato `string`, compila-a e retorna um ponteiro para um objeto do tipo `regexp.Regexp`, utilizado para interagir com a expressão compilada. Neste exemplo, criamos uma expressão regular que irá capturar qualquer caractere numérico (`\d`, que precisa ser utilizado como uma sequência de escape dentro da `string`, por isso utilizamos a dupla barra inversa).

Com a expressão regular compilada em mãos, chamamos seu método `ReplaceAllString()`, especificando a `string` que deverá ser processada (`Anderson tem 21 anos`) e uma segunda `string` indicando o texto que irá substituir os números encontrados – neste caso, desejamos substituir os números 2 e 1 pelo número 3. Desta forma, executando a função `main()` apresentada, teríamos o texto `Anderson tem 33 anos` impresso.

Esta é a forma mais comum de substituição de textos utilizando expressões regulares. No entanto, em alguns casos precisamos construir processadores mais complexos. Por exemplo, desejamos transformar a primeira letra de cada palavra em maiúscula. É impossível resolver este problema usando uma

substituição simples como a do exemplo anterior.

Para resolver esses casos, objetos do tipo `regexp.Regexp` possuem um método similar ao `ReplaceAllString()`, chamado `ReplaceAllStringFunc()` que, em vez de uma `string`, aceita uma função como segundo argumento. Ela deve, por sua vez, receber uma `string` como argumento e retornar outra `string` transformada. Neste caso, utilizamos a função `ToUpper()` do pacote `strings` para transformar a `string` recebida em maiúsculas:

```
func main() {
    expr := regexp.MustCompile("\\b\\w")
    texto := "antonio carlos jobim"

    processado := expr.ReplaceAllStringFunc(
        texto,
        func(s string) string {
            return strings.ToUpper(s)
        })


    fmt.Println(processado)
}
```

Repare que definimos a função transformadora na própria chamada à função `ReplaceAllStringFunc()`. Repare também que a função definida não possui um nome. Funções deste tipo – definidas no momento em que são utilizadas – são conhecidas como **funções anônimas**.

Também podemos armazenar uma função anônima em uma variável, e nos referirmos a esta variável posteriormente. Vamos modificar o exemplo anterior para utilizar esta técnica. Adicione o seguinte código a um novo arquivo chamado `cap6-funcoes-anonimas/regexp.go`:

```
package main

import (
    "fmt"
    "regexp"
    "strings"
)

func main() {
    expr := regexp.MustCompile("\\b\\w")

    transformadora := func(s string) string {
        return strings.ToUpper(s)
    }

    texto := "antonio carlos jobim"
    fmt.Println(transformadora(texto))
    fmt.Println(expr.ReplaceAllStringFunc(
        texto, transformadora))
}
```

Extraímos a mesma função anônima utilizada no exemplo anterior, agora a atribuindo à variável `transformadora`. É importante notar que `transformadora` **não é** o nome da função, é apenas um identificador para uma posição de memória que armazena uma função, que então utilizamos em dois contextos distintos: primeiro, fazemos uma chamada explícita para transformar o texto `antonio carlos jobim` em maiúsculas; por fim, passamos a função `transformadora` como argumento para a função `regexp.ReplaceAllStringFunc()`, que irá então substituir apenas os caracteres capturados pela expressão regular `expr`.

Executando esse exemplo, teríamos o seguinte resultado:

```
$ go run cap6-funcoes-anonimas/regexp.go
ANTONIO CARLOS JOBIM
Antonio Carlos Jobim
```

Para saber mais sobre os recursos do pacote `regexp`, visite http://golang.org/pkg/regex/.

6.6 CLOSURES

Funções definidas de forma anônima dentro de outra função herdam o contexto de onde elas foram criadas. Para demonstrar este fato, vamos criar uma implementação da sequência de Fibonacci utilizando uma função anônima. Crie um novo arquivo `cap6-closures/fibonacci.go` com o seguinte código:

```
package main

import "fmt"

func main() {
    a, b := 0, 1

    fib := func() int {
        a, b = b, a+b

        return a
    }

    for i := 0; i < 8; i++ {
        fmt.Printf("%d ", fib())
    }
}
```

Inicialmente, declaramos e inicializamos as variáveis `a` e `b` com os valores 0 e 1, respectivamente, dentro da própria função `main()`.

Em seguida, definimos a função anônima – atribuída à variável `fib` – que calcula a sequência de Fibonacci. Repare que esta função manipula diretamente as variáveis `a` e `b`, atualizando seus valores a cada chamada. Esta propriedade de uma função

poder manipular o contexto onde ela foi originalmente definida é conhecida como *closure* (*clausura* em português, embora este termo seja raramente traduzido).

Executando o programa criado anteriormente, teríamos o seguinte resultado:

```
$ go run cap6-closures/fibonacci.go
1 1 2 3 5 8 13 21
```

6.7 HIGHER-ORDER FUNCTIONS

Como já foi dito, funções em Go podem retornar outras funções ou recebê-las como argumentos. As funções que possuem estas características são conhecidas como *higher-order functions* (*funções de ordem superior* em português – novamente um termo raramente utilizado) e são parte fundamental de qualquer linguagem de programação funcional. Apesar de a linguagem Go não ser uma linguagem funcional pura, ela disponibiliza algumas das poderosas abstrações introduzidas por este paradigma.

Vamos extrair a função que calcula a sequência de Fibonacci – apresentada no exemplo anterior – para uma função externa chamada `GerarFibonacci()`, alterando-a para que receba a quantidade de números que deverão ser gerados. Também precisamos fazer com que esta função **retorne outra função** capaz de gerar a sequência de Fibonacci. Adicione o seguinte código a um novo arquivo chamado `cap6-higher-order-functions/cronometro.go`:

```
package main

import (
    "fmt"
```

```
    "time"
)

func GerarFibonacci(n int) func() {
    return func() {
        a, b := 0, 1

        fib := func() int {
            a, b = b, a+b

            return a
        }

        for i := 0; i < n; i++ {
            fmt.Printf("%d ", fib())
        }
    }
}
```

É importante notar que, como queremos que a função `GerarFibonacci()` retorne outra função, definimos seu tipo de retorno como sendo `func()` – uma função sem argumentos e que não retorna nenhum valor.

Em seguida, vamos criar uma nova função chamada `Cronometrar()`, que receberá como argumento uma função qualquer e calculará seu tempo de execução, utilizando recursos do pacote `time`:

```
func Cronometrar(funcao func()) {
    inicio := time.Now()

    funcao()

    fmt.Printf("\nTempo de execução: %s\n\n",
        time.Since(inicio))
}
```

Assim como definimos que a função `GerarFibonacci()` retorna uma função, agora especificamos que a `Cronometrar()`

recebe também uma função sem argumentos e sem valor de retorno chamada `funcao` do tipo `func()`.

Primeiro, armazenamos o tempo atual – `time.Now()` – na variável `inicio`.

Em seguida, chamamos a função recebida como argumento. Aqui fica claro o motivo pelo qual `GerarFibonacci()` retorna outra função: para que possamos calcular seu tempo de execução, precisamos controlar **quando** ela será chamada. Assim, passamos a função retornada por `GerarFibonacci()` como argumento para `Cronometrar()`, que por sua vez decide o momento apropriado para realizar a chamada.

Por fim, utilizamos a função `time.Since()` para calcular o intervalo de tempo entre o momento atual e o momento armazenado em `inicio`. Ela retorna um valor do tipo `time.Duration`, que representa um período de tempo. Este tipo possui vários métodos úteis. Neste exemplo, porém, utilizamos apenas a representação `string` deste objeto para imprimir o tempo de execução da `funcao`.

Vamos agora criar uma função `main()` para cronometrar a geração da sequência de Fibonacci em três situações diferentes:

```
func main() {
    Cronometrar(GerarFibonacci(8))
    Cronometrar(GerarFibonacci(48))
    Cronometrar(GerarFibonacci(88))
}
```

A seguir encontramos um exemplo da saída deste programa. Parte das sequências geradas foram omitidas para facilitar a visualização:

```
$ go run cap6-higher-order-functions/cronometro.go
1 1 2 3 5 8 13 21
Tempo de execução: 51.647us

1 1 2 3 5 8 13 21 34 55 ... 701408733 1134903170 1836311903
2971215073 4807526976
Tempo de execução: 66.315us

1 1 2 3 5 8 ... 420196140727489673 679891637638612258
1100087778366101931
Tempo de execução: 130.485us
```

Repare na representação do tempo de execução de cada função: `51.647us` para gerar 8 números, `66.315us` para gerar 48 números e `130.485us` para gerar uma sequência de 88 números. A função `String()` do tipo `time.Duration` decide automaticamente a unidade de tempo utilizada de acordo com o tamanho do período representado. Neste caso, a unidade utilizada foi o microssegundo (µs).

Para maiores informações sobre as funções e tipos disponíveis no pacote `time`, visite http://golang.org/pkg/time/.

6.8 TIPOS DE FUNÇÃO

No exemplo anterior, criamos uma função `Cronometrar()` que recebe como argumento uma outra função. Esta, por sua vez, não recebe nenhum argumento e não retorna nenhum valor. Algumas vezes, no entanto, precisamos receber como argumento funções mais complexas e flexíveis.

Uma das formas de obter tal flexibilidade é através da definição de tipos para funções (*function types*).

Considere uma função de agregação que recebe uma lista de

valores, um valor inicial e uma função agregadora:

```
func Agregar(
    valores []int,
    inicial int,
    fn func(n, m int) int,
) int {
    agregado := inicial

    for _, v := range valores {
        agregado = fn(v, agregado)
    }

    return agregado
}
```

Utilizando-a como base, podemos agora escrever uma função para calcular a soma de uma série numérica:

```
func CalcularSoma(valores []int) int {
    soma := func(n, m int) int {
        return n + m
    }

    return Agregar(valores, 0, soma)
}
```

E também podemos escrever uma função para calcular o produto de uma série numérica:

```
func CalcularProduto(valores []int) int {
    multiplicacao := func(n, m int) int {
        return n * m
    }

    return Agregar(valores, 1, multiplicacao)
}
```

Podemos agora criar a função `main()` e utilizar as duas novas funções:

```
func main() {
```

```
    valores := []int{3, -2, 5, 7, 8, 22, 32, -1}

    fmt.Println(CalcularSoma(valores))
    fmt.Println(CalcularProduto(valores))
}
```

Para esta série, o programa imprimirá os valores 74 e 1182720 para a soma e o produto, respectivamente.

Repare, porém, como o trecho `func(n, m int) int` aparece três vezes e causa confusão especialmente quando utilizado como argumento na definição da função `Agregar()`.

Vamos extrair esta declaração comum para um tipo de função chamado `Agregadora`:

```
type Agregadora func(n, m int) int
```

Veja como a definição de um tipo de função não é muito diferente da definição de outros tipos.

Na prática, um *function type* é muito semelhante a uma interface: qualquer função que receba dois argumentos do tipo `int` e retorne um valor `int` satisfaz a assinatura definida pelo tipo `Agregadora` e, portanto, pode ser utilizada como uma função agregadora.

Agora vamos modificar a função `Agregar()` para utilizar o novo tipo na declaração da lista de argumentos:

```
func Agregar(
    valores []int,
    valorInicial int,
    fn Agregadora,
) int {
    agregado := valorInicial

    for _, v := range valores {
```

```
        agregado = fn(v, agregado)
    }

    return agregado
}
```

Assim tornamos muito mais legível a definição da função `Agregar()` e especialmente do argumento `fn`. Além disso, `CalcularSoma()` e `CalcularProduto()` não precisam sofrer nenhuma alteração, pois as funções que ambas passam como argumento para `Agregar()` satisfazem implicitamente a assinatura de `Agregadora`.

6.9 SERVINDO HTTP ATRAVÉS DE FUNÇÕES

No capítulo anterior, na seção *Um exemplo da biblioteca padrão: io.Reader*, vimos um exemplo do uso de interfaces na própria biblioteca padrão da linguagem Go. Agora vamos recorrer novamente a ela para demonstrar o uso de funções que recebem outras como argumento através de um *function type*. Para isso vamos utilizar um dos recursos mais importantes da biblioteca padrão: escrever um servidor HTTP.

A maior parte dos recursos disponíveis para a escrita de servidores HTTP pode ser encontrada no pacote `net/http` (http://golang.org/pkg/net/http/).

Nosso servidor será bastante simples e apenas retornará a data atual do sistema.

A forma mais fácil de escrever este servidor em Go é através da função `http.HandleFunc()`, que recebe dois argumentos: primeiro, uma `string` definindo o padrão da URL atendida por

este serviço, e por último, uma função do tipo `http.HandlerFunc`, que define a seguinte assinatura:

```
type http.HandlerFunc func(http.ResponseWriter, *http.Request)
```

Isso significa que qualquer função que possua a mesma assinatura pode ser usada como um servidor HTTP!

A implementação do serviço é bastante simples. Crie um novo arquivo chamado `cap6-servidor-tempo/servidor_tempo.go` com o conteúdo:

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

func main() {
    http.HandleFunc(
        "/tempo",
        func(w http.ResponseWriter, r *http.Request) {
            fmt.Fprintf(w, "%s",
                time.Now().Format("2006-01-02 15:04:05"))
        })

    http.ListenAndServe(":8080", nil)
}
```

É muito importante notar que, na prática, um servidor HTTP em Go é um programa como outro qualquer, portanto, precisa definir um pacote `main` onde a função `main()` é o ponto de partida. Sendo assim, `main()` é o lugar ideal para configurarmos nosso serviço.

Inicialmente, utilizamos a função `http.HandleFunc()` para registrá-lo, especificando que ele atenderá requisições recebidas na

URL `/tempo`.

Em seguida, especificamos uma função anônima que satisfaz a assinatura definida pelo tipo `http.HandlerFunc`, recebendo um `http.ResponseWriter` e um ponteiro para um objeto `http.Request`.

A implementação desta função apresenta algumas novidades: utilizamos a função `fmt.Fprintf()` para imprimir a data e hora atual como resposta à requisição HTTP. Esta função é muito similar à `fmt.Printf`, que já usamos muitas vezes até aqui. A diferença é que `Printf()` escreve na saída padrão, enquanto `Fprintf()` recebe como primeiro argumento um valor do tipo `io.Writer`, onde a saída será escrita – neste caso, utilizamos o valor recebido em `w`. Este valor é, na verdade, do tipo `http.ResponseWriter`, mas satisfaz a interface `io.Writer` e, portanto, pode ser utilizado neste contexto.

Para formatar a data e hora atual, utilizamos o método `Format()` definido pelo tipo `time.Time`. Este método recebe como argumento uma `string` que define o formato desejado. A data escolhida para definir o formato no nosso exemplo não é aleatória: o método `Format()` utiliza esta data preestabelecida (2 de janeiro de 2006, 15:04:05) para analisar e reconhecer o formato recebido.

Por fim, utilizamos a função `http.ListenAndServe()` para especificar que o servidor deverá aceitar conexões na porta 8080. Esta função inicia um servidor HTTP, bloqueia a execução do programa e delega as conexões recebidas no endereço definido para os serviços registrados. O segundo argumento é um valor do tipo `http.Handler` e, neste caso, foi especificado como `nil` pois

já registramos um serviço através do método `http.HandleFunc()`.

Podemos executar o servidor da mesma forma que executamos todos os outros programas:

```
$ go run cap6-servidor-tempo/servidor_tempo.go
```

Caso o servidor tenha sido iniciado com sucesso, nenhuma saída será impressa. Para testar o serviço, abra seu navegador e direcione-o à URL `http://localhost:8080/tempo`. O resultado será uma página similar à demonstrada a seguir, com a data e a hora atuais:

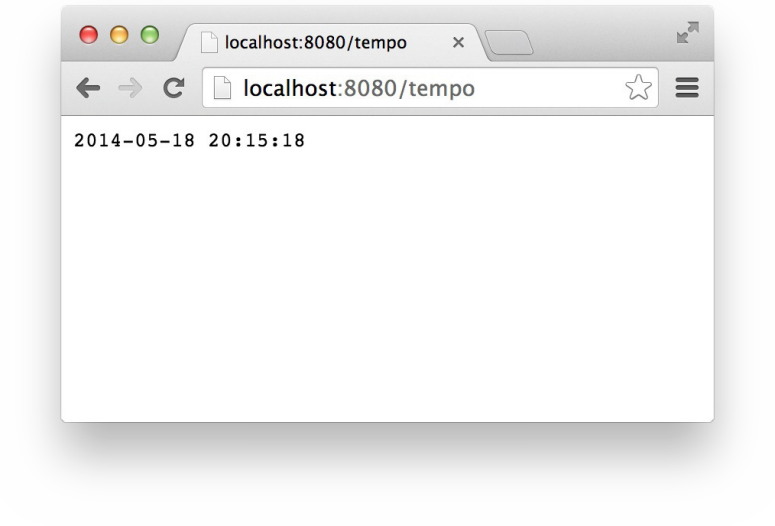

Figura 6.1: Exemplo de data e hora retornados pelo servidor de tempo

Neste capítulo, vimos todos os recursos disponíveis para a criação e manipulação de funções, incluindo funções anônimas,

closures e funções de primeira classe.

Também vimos como criar um servidor HTTP simples utilizando uma função para atender às requisições.

No capítulo a seguir, veremos os recursos para a escrita de programas concorrentes em Go.

Capítulo 7

CONCORRÊNCIA COM GOROUTINES E CHANNELS

"Do not communicate by sharing memory; instead, share memory by communicating." - Effective Go

Programação concorrente é um tema bastante delicado e complexo. Se você já escreveu programas que dependiam de múltiplas *threads* ou múltiplos processos sabe que é uma tarefa que exige muita atenção e paciência, especialmente quando é necessário compartilhar estado entre as diferentes linhas de execução.

A maior parte das linguagens de programação implementa *threads* de alguma forma, seja através de um escalonador próprio ou delegando o controle para o sistema operacional hospedeiro.

O compartilhamento de estado em um programa *multi-threaded* é normalmente implementado através de variáveis globais e/ou compartilhadas e exige algum mecanismo de trava ou semáforo para evitar condições de corrida (*race conditions*).

Para evitar este tipo de problema, Go implementa um modelo

de concorrência baseado em *goroutines* que se comunicam através de *channels*, sendo que o próprio ambiente de execução garante que apenas uma *goroutine* acesse um *channel* em um determinado momento.

Existe um documento (em inglês) no site oficial da linguagem chamado *Effective Go* (http://golang.org/doc/effective_go.html) que descreve uma série de boas práticas para garantir a escrita de programas com código limpo e idiomático em Go. Este documento contém a definição de um slogan que resume a ideologia por trás do modelo de concorrência escolhido: *Do not communicate by sharing memory; instead, share memory by communicating* (não comunique através de memória compartilhada; em vez disso, compartilhe memória através da comunicação).

Nas próximas seções veremos como atingir este objetivo.

7.1 GOROUTINES

Uma *goroutine* é um tipo de processo extremamente leve. Na prática, uma goroutine é muito similar a uma thread. No entanto, goroutines são gerenciadas pelo ambiente de execução da linguagem, que decide quando e como associá-las a threads do sistema operacional.

Para iniciar uma goroutine, utilizamos a palavra-chave `go` seguida de uma chamada de função. O ambiente de execução irá executar a função chamada sem bloquear a linha de execução principal.

Considere a seguinte função `imprimir()`, que recebe um

número inteiro e o imprime três vezes com um espaço de 200 milissegundos entre cada impressão:

```
func imprimir(n int) {
    for i := 0; i < 3; i++ {
        fmt.Printf("%d ", n)
        time.Sleep(200 * time.Millisecond)
    }
}
```

Agora considere a seguinte função `main()` que chama a função `imprimir()` duas vezes com números diferentes:

```
func main() {
    imprimir(2)
    imprimir(3)
}
```

Executando esta função `main()` obteríamos `2 2 2 3 3 3` como resultado.

Vamos alterar a primeira chamada para que seja executada em uma goroutine:

```
func main() {
    go imprimir(2)
    imprimir(3)
}
```

Através da simples adição da palavra-chave `go`, esta nova versão executa as duas chamadas de forma concorrente e produz um resultado completamente diferente: `3 2 3 2 3 2`.

É importante ressaltar que as goroutines dependem da função `main()` para que continuem sua execução. Em outras palavras, as goroutines **morrem** quando a função `main()` finaliza sua execução.

Podemos provar este fato com o simples programa a seguir, que inicia uma goroutine que dorme por 5 segundos, enquanto a função `main()` dorme por apenas 3 segundos:

```
func dormir() {
    fmt.Println("Goroutine dormindo por 5 segundos...")
    time.Sleep(5 * time.Second)
    fmt.Println("Goroutine finalizada.")
}

func main() {
    go dormir()

    fmt.Println("Main dormindo por 3 segundos...")
    time.Sleep(3 * time.Second)
    fmt.Println("Main finalizada.")
}
```

O resultado da execução deste programa seria o seguinte:

```
Main dormindo por 3 segundos...
Goroutine dormindo por 5 segundos...
Main finalizada.
```

A goroutine executando a função `dormir()` nunca termina, pois a função `main()` termina sua execução antes e o programa é finalizado.

7.2 CHANNELS

A capacidade de executar diferentes goroutines concorrentemente é muito importante e abre diversas possibilidades para a solução de problemas que exigem alta performance e melhor eficiência no uso de recursos de processamento.

Entretanto, é muito rara a situação em que várias goroutines

são disparadas independentemente, sem que haja comunicação entre elas. Os *channels* foram criados como uma abstração para viabilizar esta comunicação.

Um *channel* é um canal que conduz informações de um determinado tipo – qualquer tipo válido em Go.

Para criar um novo canal, utilizamos a função `make()`. Por exemplo, para criar um canal capaz de trafegar valores do tipo `int`, podemos utilizar o seguinte comando:

```
c := make(chan int)
```

Para interagir com um canal, utilizamos o operador `<-` (conhecido como *arrow operator*, ou operador seta). A posição do canal em relação à seta indica a direção do fluxo da comunicação. Por exemplo, para enviar valores `int` para o canal `c`, utilizamos a seguinte notação:

```
c <- 33
```

E para receber um valor enviado para o canal `c`:

```
valor := <-c
```

A seguir temos um exemplo simples que combina todos os passos anteriores para demonstrar o fluxo de comunicação completo:

```
func main() {
    c := make(chan int)
    go produzir(c)

    valor := <-c
    fmt.Println(valor)
}

func produzir(c chan int) {
```

```
    c <- 33
}
```

Inicialmente criamos um canal para trafegar valores do tipo `int`.

Em seguida, disparamos uma goroutine executando a função `produzir()`, que recebe um canal como argumento e simplesmente envia um número inteiro para o canal recebido.

Por padrão, operações de envio e recebimento em um canal bloqueiam até que o outro lado esteja pronto. Este fato permite que a própria comunicação entre duas goroutines garanta a sincronização entre elas, sem que nenhum mecanismo de travas seja necessário.

Por este motivo, a próxima linha da função `main()` – que recebe um valor do canal – fará com que a linha de execução principal fique bloqueada até que algum valor seja enviado para o canal `c`. Assim que o valor 33 for enviado pela função `produzir()`, a linha de execução principal será então desbloqueada, o valor 33 será consumido, atribuído à variável `valor` e impresso no console.

7.3 BUFFERS

Canais podem ser criados com um buffer. Por exemplo:

```
c := make(chan int, 5)
```

O canal `c` foi criado com um buffer de tamanho 5. Isto quer dizer que operações de envio não serão bloqueadas enquanto o buffer não estiver cheio, e operações de recebimento não serão bloqueadas enquanto o buffer não estiver vazio. Vejamos mais um

exemplo para facilitar o entendimento:

```
func main() {
    c := make(chan int, 3)
    go produzir(c)

    fmt.Println(<-c, <-c, <-c, <-c)
}

func produzir(c chan int) {
    c <- 1
    c <- 2
    c <- 3
}
```

Criamos um canal com buffer de tamanho 3 e imediatamente disparamos uma goroutine que envia três valores pelo canal criado. Em seguida, recebemos quatro valores do canal e os imprimimos no console. Qual seria o resultado desse programa?

```
fatal error: all goroutines are asleep - deadlock!
```

Causamos um *deadlock*! A goroutine produtora encerrou sua execução logo após produzir os três valores; por causa do buffer, nenhuma das operações de envio fez com que a execução fosse bloqueada. No entanto, ao tentarmos receber um quarto valor – que nunca foi produzido – pelo canal, a linha principal ficou bloqueada. O ambiente de execução detectou o *deadlock* e encerrou a execução do programa com um erro.

O produtor precisa indicar de alguma forma que não enviará mais nenhum valor pelo canal. Para isso existe a função embutida `close()`. Vamos alterar a função `produzir()` para fechar o canal após produzir os três valores:

```
func produzir(c chan int) {
    c <- 1
    c <- 2
```

```
    c <- 3

    close(c)
}
```

É muito importante que o lado produtor feche o canal sempre que possível quando não houver mais valores a serem produzidos.

Agora, precisamos também detectar que o canal foi fechado do lado do consumidor.

O operador `<-` retorna sempre dois valores: o valor lido e um valor `bool` indicando se o valor foi lido com sucesso ou não; este valor será `false` quando o canal tiver sido fechado.

Até o momento, ignoramos o segundo valor retornado. Vamos alterar também a função `main()` para checá-lo:

```
func main() {
    c := make(chan int, 3)
    go produzir(c)

    for {
        valor, ok := <-c
        if ok {
            fmt.Println(valor)
        } else {
            break
        }
    }
}
```

Qual seria o resultado do programa agora?

```
1
2
3
```

Resolvemos o *deadlock*! Porém, o processo de checar sempre se o canal continua aberto ou não é bastante tedioso.

Felizmente, há uma outra forma de fazê-lo: podemos utilizar o operador `range` para ler valores de um canal conforme eles forem sendo produzidos. Vamos criar um exemplo para utilizá-lo. Em um novo arquivo chamado `cap7-buffers/buffers.go`, adicione a função `produzir()` apresentada no exemplo anterior:

```
package main

import "fmt"

func produzir(c chan int) {
    c <- 1
    c <- 2
    c <- 3

    close(c)
}
```

Em seguida, crie a seguite função `main()`, parecida com a do exemplo anterior, porém utilizando o operador `range`:

```
func main() {
    c := make(chan int, 3)
    go produzir(c)

    for valor := range c {
        fmt.Println(valor)
    }
}
```

O código resultante é menor e muito mais claro.

Execute o programa e veja que o resultado é idêntico:

```
$ go run cap7-buffers/buffers.go
1
2
3
```

7.4 CONTROLANDO A DIREÇÃO DO FLUXO

Por padrão, a comunicação em um canal é bidirecional. Algumas vezes, porém, desejamos controlar a direção do fluxo quando passamos um canal como argumento para outra função, ou quando temos uma função que retorne um canal.

Considerando ainda o exemplo anterior, vamos alterar a função `produzir()` para definir a direção da comunicação:

```
func produzir(c chan<- int) {
    // ...
}
```

Repare que, ao receber o canal como argumento, definimos que a função poderá somente enviar valores para o canal (`chan<-`). Desta forma, caso a função tente receber valores pelo mesmo canal, causará o seguinte erro:

```
invalid operation: <-c (receive from send-only type chan<- int)
```

De maneira similar, podemos definir um canal *read-only* (somente leitura):

```
func consumir(c <-chan int) {
    // ...
}
```

Vale ressaltar que, como Go é uma linguagem com tipos fortes e os canais são tipados, esse erro é causado em tempo de **compilação**.

7.5 SELECT

É muito comum encontrar um cenário em que uma goroutine precisa interagir com múltiplos canais de comunicação. Por exemplo, uma função pode disparar goroutines que escrevem

valores em canais diferentes, e a função original depende dos valores de todos estes canais para produzir seu resultado final.

Para evitar que a execução de uma goroutine seja bloqueada esperando por operações em algum dos canais dos quais ela depende, Go fornece um comando chamado `select`, que é muito semelhante ao `switch`. Sua forma geral é:

```
select {
case v1 := <-canal1:
    // ...
case v2 := <-canal2:
    // ...
default:
    // ...
}
```

Caso exista algum valor a ser lido no `canal1`, ele será atribuído à variável `v1` e o bloco associado ao primeiro comando `case` será executado. Caso contrário, o `canal2` será checado por valores recebidos e, em caso positivo, seu valor será atribuído à variável `v2` e o bloco associado ao segundo comando `case` será executado. Se nenhum dos canais especificados possuírem valores prontos para serem lidos, o bloco `default` será executado.

Uma das formas mais comuns do uso do `select` é dentro de um laço que controla a comunicação com as goroutines. Vamos escrever um programa que, dada uma lista de números, separa-os em duas listas distintas de pares e ímpares. O algoritmo é bastante trivial, porém, vamos implementá-lo usando uma goroutine e canais separados para enviar números pares e ímpares. Vamos utilizar também um terceiro canal para indicar o final da execução da goroutine.

Primeiro, vamos criar a função `separar()`, que recebe a lista

de números e três canais: `i`, `p` e `pronto`, todos unidirecionais:

```
func separar(nums []int, i, p chan<- int, pronto chan<- bool) {
    for _, n := range nums {
        if n%2 == 0 {
            p <- n
        } else {
            i <- n
        }
    }
    pronto <- true
}
```

Para cada número presente em `nums`, verificamos se é par (divisível por 2 – `n%2 == 0`) e, em caso positivo, enviamos o número para o canal `p`; caso seja um número ímpar, ele será enviado para o canal `i`. Ao final da iteração, enviamos o valor `true` para o canal `pronto`, indicando o fim do processamento.

Vamos agora seguir a função `main()` passo a passo. Inicialmente, criamos os canais e a lista de números:

```
i, p := make(chan int), make(chan int)
pronto := make(chan bool)
nums := []int{1, 23, 42, 5, 8, 6, 7, 4, 99, 100}
```

Em seguida, disparamos uma *goroutine* para separar os números:

```
go separar(nums, i, p, pronto)
```

Com tudo preparado e a goroutine já separando os valores, precisamos coletar os resultados enviados para cada canal. Primeiro, criamos as duas listas e uma variável para controlar o laço:

```
var impares, pares []int
fim := false
```

Agora precisamos popular as listas `impares` e `pares` de acordo com a separação que está sendo feita na goroutine. Para isso, criamos um laço que executará até que o valor da variável `fim` seja verdadeiro, e utilizamos um comando `select` para ler os valores de cada um dos canais:

```
for !fim {
    select {
    case n := <-i:
        impares = append(impares, n)
    case n := <-p:
        pares = append(pares, n)
    case fim = <-pronto:
    }
}
```

Repare na última cláusula `case`: ela só será executada quando a goroutine enviar o valor `true` para o canal `pronto` e, então, a variável `fim` assumirá o valor `true` e a condição de saída do laço será atendida. Utilizar um canal para sincronizar goroutines que envolvem múltiplos canais é uma técnica muito comum em Go.

Por fim, imprimimos os valores separados presentes nas listas `impares` e `pares`:

```
fmt.Printf("Ímpares: %v | Pares: %v\n", impares, pares)
```

A seguir podemos ver o código completo da função `main()`:

```
func main() {
    i, p := make(chan int), make(chan int)
    pronto := make(chan bool)
    nums := []int{1, 23, 42, 5, 8, 6, 7, 4, 99, 100}

    go separar(nums, i, p, pronto)

    var impares, pares []int
    fim := false
```

```
    for !fim {
        select {
        case n := <-i:
            impares = append(impares, n)
        case n := <-p:
            pares = append(pares, n)
        case fim = <-pronto:
        }
    }

    fmt.Printf("Ímpares: %v | Pares: %v\n", impares, pares)
}
```

Executando este programa, obteríamos o seguinte resultado:

```
$ go run select.go
Ímpares: [1 23 5 7 99] | Pares: [42 8 6 4 100]
```

3# Temporizadores e timeouts

Controlar a execução de múltiplas goroutines através de um *select* é uma técnica bastante simples e muito utilizada em Go. Porém, existem casos em que a execução de uma determinada tarefa precisa acontecer em um período limitado de tempo.

O pacote `time` fornece uma função `After()` que ajuda a resolver estes casos. Sua assinatura é:

```
func After(d Duration) <-chan Time
```

Repare que esta função convenientemente retorna um canal. Assim, podemos facilmente utilizá-la dentro de um *select* para controlar o tempo de execução de uma goroutine, e tomar alguma atitude caso ela não produza resultados dentro do tempo esperado.

Vamos simular esta situação com uma goroutine que simplesmente dorme por 5 segundos e sinaliza o final de sua execução enviando o valor `true` para um canal. Crie um arquivo

chamado `cap7-timeout/timeout.go` e defina uma nova função `executar()` com o código da simulação:

```
package main

import (
    "fmt"
    "time"
)

func executar(c chan<- bool) {
    time.Sleep(5 * time.Second)
    c <- true
}
```

Crie então o esqueleto da função `main()`, cujo conteúdo será apresentado em partes logo em seguida:

```
func main() {
    // ...
}
```

Primeiramente, devemos criar um canal e, então, disparar uma goroutine executando a tarefa demorada:

```
c := make(chan bool, 1)
go executar(c)

fmt.Println("Esperando...")
```

Suponha que o tempo máximo aceitável para a execução desta tarefa seja de 2 segundos. Vamos implementar um mecanismo de *timeout* utilizando `time.After()` dentro de um `select` enquanto esperamos que a tarefa seja finalizada:

```
fim := false
for !fim {
    select {
    case fim = <-c:
        fmt.Println("Fim!")
    case <-time.After(2 * time.Second):
```

```
        fmt.Println("Timeout!")
        fim = true
    }
}
```

Executando este programa teremos o seguinte resultado:

```
$ go run cap7-timeout/timeout.go
Esperando...
Timeout!
```

Simples, não?

Se alterarmos a função `executar()` para dormir por apenas 1 segundo em vez de 5, teremos como resultado:

```
$ go run cap7-timeout/timeout.go
Esperando...
Fim!
```

7.6 SINCRONIZANDO MÚLTIPLAS GOROUTINES

Anteriormente, vimos um exemplo de como esperar que uma goroutine finalize sua execução através da utilização de um canal. No entanto, quando precisamos esperar pela execução de múltiplas goroutines, controlar manualmente quantas delas já terminaram pode se tornar uma tarefa sujeita a falhas.

Go fornece um tipo chamado `WaitGroup`, presente no pacote `sync`, que torna esta tarefa bem mais simples. Para vê-lo em ação, crie um arquivo chamado `cap7-sincronizador/sincronizador.go`, declarando o pacote `sync` entre suas dependências:

```
package main
```

```
import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)
```

Crie também a função `main()` com o conteúdo:

```
func main() {
    inicio := time.Now()
    rand.Seed(inicio.UnixNano())

    var controle sync.WaitGroup

    for i := 0; i < 5; i++ {
        controle.Add(1)
        go executar(&controle)
    }

    controle.Wait()

    fmt.Printf("Finalizado em %s.\n", time.Since(inicio))
}
```

Inicialmente, armazenamos o *timestamp* atual e configuramos a semente para a geração de números aleatórios, garantindo que números diferentes sejam gerados a cada execução.

Em seguida, criamos uma variável chamada `controle` do tipo `sync.WaitGroup` que utilizaremos para sincronizar a execução das goroutines. Antes de disparar cada goroutine, chamamos o método `controle.Add(1)`, indicando que uma nova goroutine deverá ser sincronizada, e então iniciamos sua execução chamando a função `executar()`.

Repare que passamos como argumento um ponteiro para a variável `controle`. Como os argumentos em Go são passados como cópias, é importante que utilizemos um ponteiro neste caso,

pois o mesmo `WaitGroup` **deve** ser utilizado por todas as goroutines.

Por fim, chamamos o método `controle.Wait()`, que neste caso irá bloquear a execução da função `main()` até que todas as goroutines tenham finalizado. Quando isto acontecer, o método `Wait()` irá retornar, e então imprimiremos o tempo percorrido entre o início e o fim da execução do programa.

Agora, vamos adicionar o código da função `executar()`:

```
func executar(controle *sync.WaitGroup) {
    defer controle.Done()

    duracao := time.Duration(1+rand.Intn(5)) * time.Second
    fmt.Printf("Dormindo por %s...\n", duracao)
    time.Sleep(duracao)
}
```

Primeiro, garantimos que o método `controle.Done()` será chamado quando a função terminar sua execução, notificando o `WaitGroup` deste fato. Em seguida, criamos um valor do tipo `time.Duration` baseado em um número aleatório entre 1 e 5 segundos. Então chamamos a função `time.Sleep()` para que a goroutine atual durma pelo período especificado.

Um exemplo da execução deste programa:

```
$ go run cap7-sincronizador/sincronizador.go
Dormindo por 5s...
Dormindo por 4s...
Dormindo por 1s...
Dormindo por 4s...
Dormindo por 3s...
Finalizado em 5.001185837s.
```

7.7 CONCORRÊNCIA, PARALELISMO E

GOMAXPROCS

Todos os recursos apresentados neste capítulo fazem parte do suporte nativo de Go para programas concorrentes. Porém, é importante ressaltar que concorrência não é paralelismo, e nenhum destes recursos fará com que um programa execute em paralelo utilizando todos os CPUs disponíveis no ambiente de execução.

Alguns programas, no entanto, realizam operações que exigem um poder de processamento maior e podem se beneficiar da utilização dos múltiplos *cores* disponíveis.

Por padrão, o *runtime* da linguagem Go executa uma única goroutine em um dado momento, mas disponibiliza uma forma de modificar esta configuração.

Por exemplo, ao iniciar a execução de um programa, pode-se especificar a variável de ambiente `GOMAXPROCS`:

```
$ GOMAXPROCS=2 go run paralelo.go
```

É possível também configurar este valor programaticamente através do pacote `runtime`:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    // ...
}
```

O código anterior instrui o *runtime* a utilizar todos os *cores* disponíveis.

A seguir veremos um exemplo de programa que se beneficia do uso de múltiplos *cores*.

```
package main

import (
    "fmt"
    "math"
    "sync"
    "time"
)

func calcular(base float64, controle *sync.WaitGroup) {
    defer controle.Done()
    n := 0.0

    for i := 0; i < 100000000; i++ {
        n += base / math.Pi * math.Sin(2)
    }

    fmt.Println(n)
}

func main() {
    inicio := time.Now()
    var controle sync.WaitGroup
    controle.Add(3)

    go calcular(9.37, &controle)
    go calcular(6.94, &controle)
    go calcular(42.57, &controle)

    controle.Wait()
    fmt.Printf("Finalizado em %s.\n", time.Since(inicio))
}
```

O código é bastante simples e apenas dispara 3 goroutines que realizam algum tipo de cálculo que exige alto uso da CPU.

Os resultados a seguir foram colhidos em um computador com 4 cores. Inicialmente, um exemplo de execução utilizando a configuração padrão do *runtime*:

```
$ go run paralelo.go
2.712037442294368e+08
2.0087022191177934e+08
```

```
1.2321391006719112e+09
Finalizado em 4.515132372s.
```

Em seguida, instruímos o *runtime* a utilizar dois *cores*:

```
$ GOMAXPROCS=2 go run paralelo.go
2.0087022191177934e+08
2.712037442294368e+08
1.2321391006719112e+09
Finalizado em 2.419171346s.
```

Veja que o tempo de execução caiu quase pela metade! Por fim, vamos instruir o *runtime* a utilizar 3 *cores*. É esperado que esta configuração seja a ideal, já que temos exatamente 3 goroutines executando. O resultado:

```
$ GOMAXPROCS=3 go run paralelo.go
2.712037442294368e+08
2.0087022191177934e+08
1.2321391006719112e+09
Finalizado em 2.165083253s.
```

Agora tivemos um ganho menor, mas ainda significativo.

É importante ressaltar que este recurso deve ser utilizado com bastante cautela e pode produzir resultados muito diferentes dependendo da natureza do programa em questão. É recomendável que cada caso seja estudado com cuidado.

7.8 RECAPITULANDO

Este capítulo apresentou os principais conceitos de concorrência disponíveis em Go, dando um foco especial a goroutines e channels. No entanto, há muito a ser explorado sobre esse assunto.

Programação concorrente é um tema de alta complexidade. O

modelo de concorrência da linguagem foi criado com o objetivo de tornar a escrita de programas concorrentes uma tarefa mais fácil.

Crie seus próprios programas utilizando os recursos aprendidos, modifique os exemplos apresentados e pratique bastante. Assim, você absorverá os conceitos mais facilmente.

CAPÍTULO 8

MÃO NA MASSA: ENCURTADOR DE URLS

Nosso passeio pelos principais recursos da linguagem Go chegou ao fim. Para consolidar todo o conhecimento acumulado até aqui, chegou a hora de colocar a mão na massa e desenvolver uma aplicação real!

Vamos desenvolver um servidor HTTP que provê uma API (*application programming interface*) para criar URLs curtas e redirecioná-las para as URLs originais. Para isso, utilizaremos os recursos do pacote `net/http` e implementaremos um repositório em memória.

8.1 ESTRUTURA DO PROJETO

Até agora, todos os exemplos apresentados definiam um único pacote `main`. Projetos maiores e mais complexos, porém, necessitam de pacotes adicionais para organizar melhor o código.

No capítulo *Introdução*, na seção *Pós-instalação*, aprendemos como configurar uma área de trabalho através da variável de ambiente `$GOPATH`. Este passo é especialmente importante para que projetos que definem múltiplos pacotes possam ser

corretamente compilados.

Todo projeto em Go segue uma convenção semelhante: a estrutura de diretórios e pacotes é definida de acordo com o caminho do repositório onde o código reside. Por exemplo, projetos hospedados no GitHub incluem `github.com/usuario/projeto/nome-do-pacote` no caminho completo dos pacotes. Veremos a importância desta convenção no capítulo a seguir.

Considerando que o nosso projeto será hospedado em https://github.com/caiofilipini/encurtador, a seguinte estrutura de diretórios deverá ser criada em `$GOPATH/src`:

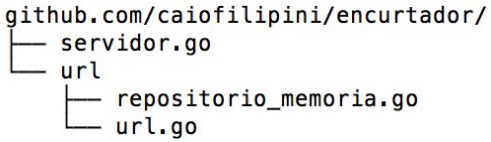

Figura 8.1: Estrutura do projeto

Vamos começar pela definição do servidor propriamente dito.

8.2 CRIANDO O SERVIDOR

O servidor será definido no arquivo `servidor.go` e será o ponto de partida da nossa aplicação. Neste arquivo, definiremos a função `main()` que será responsável por configurar e iniciar o servidor HTTP.

Primeiro, vamos definir o pacote `main` e especificar as dependências:

```
package main
```

```
import (
    "fmt"
    "log"
    "net/http"
    "strings"

    "github.com/caiofilipini/encurtador/url"
)
```

Nenhuma grande novidade até aqui, exceto pelo caminho do pacote interno `url`.

Precisamos de uma variável para armazenar a porta onde o serviço irá aceitar novas conexões, e também uma variável para representar a URL base do serviço. Como elas serão utilizadas em mais de um lugar, vamos criá-las no nível do pacote para que sejam acessíveis por todas as funções definidas nele:

```
var (
    porta   int
    urlBase string
)
```

Agora precisamos inicializá-las. Go disponibiliza um mecanismo padrão para inicialização de variáveis e recursos utilizados em um pacote: a função `init()`. Um pacote pode definir várias funções `init()` diferentes (prática comum, caso o código do pacote seja dividido em múltiplos arquivos, mas a ordem de chamada é indefinida). No nosso caso, precisamos de uma única:

```
func init() {
    porta = 8888
    urlBase = fmt.Sprintf("http://localhost:%d", porta)
}
```

E chegamos finalmente à definição da função `main()`, onde

configuramos as rotas para o nosso servidor HTTP:

```
func main() {
    http.HandleFunc("/api/encurtar", Encurtador)
    http.HandleFunc("/r/", Redirecionador)

    log.Fatal(http.ListenAndServe(
        fmt.Sprintf(":%d", porta), nil))
}
```

Definimos duas rotas para o nosso servidor:

- `/api/encurtar` : responsável por receber uma URL, criar um identificador curto e retornar a URL curta;
- `/r/<id-curto>` : responsável por receber um identificador curto e redirecionar para a URL original.

Configuramos as duas rotas utilizando a função `http.HandleFunc()` . A rota `/api/encurtar` será tratada pela função `Encurtador` , enquanto a função `Redirecionador` cuidará das requisições recebidas em `/r/` .

A seguir veremos em detalhes cada uma destas funções. A implementação do pacote interno `url` será apresentada posteriormente. Por enquanto, veremos somente a definição do tipo `Url` , que será usado nos exemplos a seguir:

```
type Url struct {
    Id      string
    Criacao time.Time
    Destino string
}
```

Este tipo é uma `struct` simples, contendo um campo para armazenar o identificador curto (`Id`), a data e hora da criação da URL curta (`Criacao`), e a URL original (`Destino`).

8.3 CRIANDO URLS CURTAS

A função `Encurtador()` irá tratar as requisições recebidas em `/api/encurtar`.

Como esta função será registrada através do método `http.HandleFunc()`, ela precisa ser do tipo `http.HandlerFunc`. Portanto, sua assinatura é:

```
func Encurtador(w http.ResponseWriter, r *http.Request)
```

Como estamos implementando uma API sobre HTTP, tentaremos seguir o padrão arquitetural REST (*Representational State Transfer*) (FIELDING, 2000). Assim, o primeiro passo é verificar se estamos recebendo uma requisição do tipo POST e, caso contrário, retornaremos um erro HTTP 405 (*Method Not Allowed*), indicando no cabeçalho `Allow` que este serviço aceita apenas requisições com o método POST:

```
if r.Method != "POST" {
    responderCom(w, http.StatusMethodNotAllowed, Headers{
        "Allow": "POST",
    })
    return
}
```

Para formatar a resposta incluindo o cabeçalho `Allow`, chamamos uma função utilitária `responderCom()` que recebe o objeto `http.ResponseWriter`, o código de resposta desejado e um mapa de cabeçalhos do tipo `Headers`. Este tipo é apenas um *type alias* para o tipo `map[string]string` e foi criado para tornar o código mais claro. Sua definição é a seguinte:

```
type Headers map[string]string
```

A função `responderCom()` itera sobre os cabeçalhos

recebidos, configurando cada um deles através do método `http.ResponseWriter.Header().Set()` antes de, finalmente, ajustar o código de resposta chamando o método `WriteHeader()`, também disponível para objetos do tipo `http.ResponseWriter`:

```
func responderCom(
    w http.ResponseWriter,
    status int,
    headers Headers,
) {
    for k, v := range headers {
        w.Header().Set(k, v)
    }
    w.WriteHeader(status)
}
```

Continuando o fluxo de execução, caso tenhamos recebido uma requisição POST válida, chamamos a função `extrairUrl()` para ler e retornar a URL recebida no corpo da requisição. Esta função é bastante simples:

```
func extrairUrl(r *http.Request) string {
    url := make([]byte, r.ContentLength, r.ContentLength)
    r.Body.Read(url)
    return string(url)
}
```

Primeiro, criamos um slice de `bytes` chamado `url`, especificando seu tamanho inicial como sendo o tamanho do corpo da requisição – presente no campo `ContentLength` do tipo `http.Request`. Em seguida, utilizamos o método `Body.Read()`, também do tipo `http.Request`, para ler o conteúdo do corpo da requisição e copiá-lo para o slice `url`. Por fim, retornamos uma `string` criada a partir da conversão dos bytes presentes no slice `url`.

Com a URL recebida em mãos, chamamos o método `BuscarOuCriarNovaUrl()` do nosso próprio pacote `url`, passando como argumento a URL extraída da requisição. Este método retorna três valores: um objeto do tipo `url.Url`, representando a URL curta recém-criada; um valor `bool`, sendo `true` caso uma nova URL curta tenha sido criada, ou `false` caso a URL recebida já tenha sido encurtada anteriormente; e um `error` caso a URL recebida seja inválida:

```
url, nova, err := url.BuscarOuCriarNovaUrl(extrairUrl(r))
```

Caso tenhamos recebido um erro, retornamos um erro HTTP 400 (*Bad Request*):

```
if err != nil {
    responderCom(w, http.StatusBadRequest, nil)
    return
}
```

Em caso de sucesso, verificamos o valor da variável `nova` para decidir o código de resposta. Responderemos com o código HTTP 201 (*Created*) caso a URL tenha sido criada, ou HTTP 200 (*OK*), caso a URL já tenha sido encurtada anteriormente:

```
var status int
if nova {
    status = http.StatusCreated
} else {
    status = http.StatusOK
}
```

Finalmente, formatamos a URL curta utilizando a `urlBase` definida na inicialização do pacote e apontando para a rota `/r/`, e então respondemos com o código decidido anteriormente (201 ou 200) e a URL curta no cabeçalho `Location`:

```
urlCurta := fmt.Sprintf("%s/r/%s", urlBase, url.Id)
```

```
responderCom(w, status, Headers{"Location": urlCurta})
```

Para demonstrar alguns exemplos de requisições para o nosso servidor, a partir de agora utilizaremos uma ferramenta chamada `cURL`. Se você é usuário Linux ou Mac OS, provavelmente já possui esta ferramenta instalada por padrão. Caso contrário, verifique as instruções de instalação para o seu sistema operacional em http://curl.haxx.se/download.html.

Utilizaremos duas opções do comando `curl`: `-v` para visualizarmos todos os cabeçalhos e o conteúdo da requisição e da resposta; e `-d`, que é utilizada para fazer uma requisição do tipo POST, sendo que a string especificada será enviada no corpo desta requisição. Para facilitar a visualização, algumas linhas produzidas pelo comando serão omitidas em todos os exemplos de requisições.

A seguir, veremos um exemplo de requisição para encurtar uma URL nova:

```
$ curl -v http://localhost:8888/api/encurtar \
-d "http://casadocodigo.com.br/products/livro-vraptor"
> POST /api/encurtar HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
> Content-Length: 64
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 201 Created
< Location: http://localhost:8888/r/oZYm9
< Date: Tue, 03 Jun 2014 21:45:41 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
```

Repare na URL curta retornada no cabeçalho `Location`.

Agora, se repetirmos a mesma requisição, receberemos uma resposta com código 200 em vez do 201, conforme esperado:

```
$ curl -v http://localhost:8888/api/encurtar \
-d "http://casadocodigo.com.br/products/livro-vraptor"
> POST /api/encurtar HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
> Content-Length: 64
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 200 OK
< Location: http://localhost:8888/r/oZYm9
< Date: Tue, 03 Jun 2014 21:48:24 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
```

8.4 REDIRECIONANDO PARA AS URLS ORIGINAIS

Analisando a URL curta retornada pelo exemplo anterior, percebemos que ela aponta para a rota `/r/`, que foi registrada para ser tratada pela função `Redirecionador()`. Esta função também deve ser do tipo `http.HandlerFunc` e, portanto, deve seguir a mesma assinatura:

```
func Redirecionador(w http.ResponseWriter, r *http.Request)
```

Apesar de o mecanismo de redirecionamento ser o coração de um serviço encurtador de URLs, não precisamos de muito código para realizar esta tarefa. O padrão das URLs curtas geradas pelo serviço anterior é `/r/<id-curto>`. No caso da requisição de exemplo, geramos a URL curta `/r/oZYm9`; o identificador curto é `oZYm9`.

Com isso em mente, o primeiro passo para redirecionar para a URL original é extrair seu identificador do caminho da URL curta (obtido em `http.Request.URL.Path`). Podemos chamar a função `strings.Split()`, que separa uma string em várias partes baseado em um caractere especificado e retorna as partes separadas em um slice. Vamos utilizar `"/"` como caractere separador, armazenar o slice retornado na variável `caminho`, e então assumir que o último elemento do slice contém o identificador esperado:

```
caminho := strings.Split(r.URL.Path, "/")
id := caminho[len(caminho)-1]
```

Agora precisamos verificar se este identificador corresponde a uma URL armazenada. Para isso, vamos utilizar a função `Buscar()` do nosso pacote `url`, que recebe um identificador e retorna a URL encontrada ou `nil`, caso a mesma não exista. Desta forma, verificamos se o retorno é diferente de `nil`. Em caso positivo, retornamos uma resposta HTTP 301 (*Moved Permanently*) que irá redirecionar para a URL original (`url.Destino`); caso o identificador não corresponda a uma URL encurtada anteriormente, simplesmente retornamos um erro HTTP 404 (*Not Found*):

```
if url := url.Buscar(id); url != nil {
    http.Redirect(w, r, url.Destino,
        http.StatusMovedPermanently)
} else {
    http.NotFound(w, r)
}
```

Repare que, como em nenhum dos casos precisamos responder com cabeçalhos especiais, utilizamos as funções predefinidas `http.Redirect()` e `http.NotFound()` para formatar as

respostas. Outro detalhe importante é a criação da variável `url` no bloco `if` ; isto faz com que seu escopo seja limitado somente ao próprio bloco e é uma prática bastante comum quando a variável não será utilizada em nenhum outro lugar.

Utilizando a URL encurtada anteriormente, podemos fazer uma requisição GET e visualizar a resposta:

```
$ curl -v http://localhost:8888/r/oZYm9
> GET /r/oZYm9 HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Location: http://casadocodigo.com.br/...programador-apaixonado
< Date: Tue, 03 Jun 2014 22:15:40 GMT
< Content-Length: 99
< Content-Type: text/html; charset=utf-8
<
```

Caso a URL desejada não exista, receberemos um erro HTTP 404, como no exemplo a seguir:

```
curl -v http://localhost:8888/r/naoexiste
> GET /r/naoexiste HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Type: text/plain; charset=utf-8
< Date: Tue, 03 Jun 2014 22:17:02 GMT
< Content-Length: 19
<
```

8.5 APRESENTANDO O PACOTE URL

Até aqui, o pacote interno `url` foi apresentado mais ou

menos como uma caixa preta. Interagimos com o tipo `url.Url` e as funções `url.Buscar()`, `url.BuscarOuCriarNovaUrl()`, `url.ConfigurarRepositorio()` e `url.NovoRepositorioMemoria()`.

Agora veremos em detalhes cada um destes elementos.

Criar um pacote é a melhor forma de separar – física e logicamente – certos elementos em uma aplicação Go. No pacote `url`, definimos toda a lógica de criação e armazenamento das URLs curtas, sendo que apenas uma parte destas funcionalidades precisa ser exposta. O servidor não precisa conhecer detalhes de como tudo funciona; precisa apenas conhecer a interface pública com a qual ele deve interagir para criar e redirecionar URLs.

Todos os exemplos apresentados até aqui definiam apenas o pacote `main`. Para criar um pacote diferente, basta atribuir a ele um outro nome. Por isso, o arquivo `url/url.go` começa com a seguinte declaração:

```
package url
```

Todo o código definido dentro deste arquivo será compilado em um pacote chamado `github.com/caiofilipini/encurtador/url`, e apenas os elementos cujos nomes iniciarem com uma letra maiúscula serão visíveis por outros pacotes.

O pacote `url`, por sua vez, depende de alguns outros pacotes:

```
import (
    "math/rand"
    "net/url"
    "time"
)
```

Além das dependências, nosso pacote também define algumas constantes utilizadas internamente (note que seus nomes iniciam com letras minúsculas):

```
const (
    tamanho  = 5
    simbolos = "abcdefghijklmnopqr...STUVWXYZ1234567890_-+"
)
```

A constante `tamanho` define o tamanho do identificador curto que deverá ser gerado. Já a constante `simbolos` lista todos os caracteres permitidos em um identificador; veremos em breve como eles são gerados.

Para inicializar suas dependências, um pacote pode definir uma função `init()`, como a que foi apresentada no arquivo `servidor.go`. O pacote `url` também define sua própria função de inicialização; neste caso, ela é utilizada apenas para configurar a semente para geração de números aleatórios:

```
func init() {
    rand.Seed(time.Now().UnixNano())
}
```

O pacote `url` define também alguns tipos, como o já conhecido tipo `Url`:

```
type Url struct {
    Id      string
    Criacao time.Time
    Destino string
}
```

8.6 ESPECIFICANDO A IMPLEMENTAÇÃO DO REPOSITÓRIO

Para padronizar a implementação do repositório onde as URLs serão armazenadas, definimos também uma interface chamada `Repositorio`:

```
type Repositorio interface {
    IdExiste(id string) bool
    BuscarPorId(id string) *Url
    BuscarPorUrl(url string) *Url
    Salvar(url Url) error
}
```

O objetivo desta interface é especificar todas as operações que um repositório de URLs deverá ser capaz de implementar. Neste exercício, implementaremos um repositório em memória. Porém, poderíamos substituir esta implementação por um banco de dados relacional, por exemplo, desde que o contrato definido pela interface `Repositorio` fosse atendido. Desta forma, precisaríamos alterar apenas a linha que configura o repositório a ser utilizado no arquivo `servidor.go`.

Como a implementação concreta do repositório é configurada externamente, precisamos definir uma função que seja capaz de registrar esta configuração. Para isso, definimos primeiramente a variável `repo` para armazenar o repositório propriamente dito, e fornecemos a função `ConfigurarRepositorio()`:

```
var repo Repositorio

func ConfigurarRepositorio(r Repositorio) {
    repo = r
}
```

Repare que o tipo do argumento `r` é definido como `Repositorio`, que é a interface definida anteriormente. Desta forma, não importa qual é a implementação configurada, deste que a interface seja obedecida; o próprio compilador garante esta

restrição. Repare também que a única forma de configurar o repositório é através desta função, já que a variável `repo` não é visível fora do pacote.

A implementação do repositório em memória será apresentada posteriormente.

8.7 CRIANDO IDENTIFICADORES CURTOS

Uma das funcionalidades principais de um serviço encurtador de URLs é a capacidade de gerar identificadores curtos. Para expor esta funcionalidade, o pacote `url` implementa a função `BuscarOuCriarNovaUrl()`. Sua assinatura é a seguinte:

```
func BuscarOuCriarNovaUrl(destino string) (
    u *string,
    nova bool,
    err error,
)
```

Recapitulando, esta função recebe a URL original como argumento e retorna três valores: um ponteiro para um objeto do tipo `Url`, representando a URL curta recém-criada; um valor `bool`, sendo `true` caso uma nova URL curta tenha sido criada, ou `false` caso a URL recebida já tenha sido encurtada anteriormente; e um `error` caso a URL recebida seja inválida.

O primeiro passo é verificar se a URL já foi encurtada anteriormente, evitando duplicações. Para isso, chamamos o método `repo.BuscarPorUrl()` passando a URL `destino` como argumento. Caso uma URL tenha sido encontrada, ela será imediatamente retornada, acompanhada do valor `false`, indicando que a URL já existia, e `nil` como erro:

```
if u = repo.BuscarPorUrl(destino); u != nil {
    return u, false, nil
}
```

Caso a URL especificada ainda não tenha sido encurtada, precisamos ter certeza de que recebemos uma URL válida para evitar problemas de redirecionamento. A biblioteca padrão fornece um pacote chamado `net/url`, responsável por identificar e manipular URLs de todos os tipos. Este pacote define uma função `ParseRequestURI()`, que trata especificamente de URLs HTTP, e retorna um erro caso a URL especificada não seja válida – exatamente o que precisamos fazer. Caso um erro tenha sido encontrado, ele será retornado:

```
if _, err = url.ParseRequestURI(destino); err != nil {
    return nil, false, err
}
```

Se chegamos até aqui, temos a certeza de que a URL recebida é válida e não foi encurtada anteriormente. Portanto, criamos um novo objeto `Url` e o populamos com um identificador curto, uma data de criação (a data/hora atual) e a URL destino. Em seguida, chamamos o método `repo.Salvar()` para persisti-lo no repositório e retornamos o endereço do objeto recém-criado e o valor `true`, indicando a criação de uma nova URL:

```
url := Url{gerarId(), time.Now(), destino}
repo.Salvar(url)
return &url, true, nil
```

Agora vamos acompanhar passo a passo o algoritmo para gerar um novo identificador curto, implementado pela função `gerarId()`. A ideia geral é extrair cinco caracteres aleatórios (como definido pela constante `tamanho`) da lista de caracteres permitidos (definida na constante `simbolos`) e juntá-los em uma

`string` – o identificador; caso ele já exista no repositório, repetimos o processo até que um identificador totalmente novo seja gerado:

```
func gerarId() string {
    novoId := func() string {
        id := make([]byte, tamanho, tamanho)
        for i := range id {
            id[i] = simbolos[rand.Intn(len(simbolos))]
        }
        return string(id)
    }

    for {
        if id := novoId(); !repo.IdExiste(id) {
            return id
        }
    }
}
```

Como cada caractere da `string simbolos` é acessado individualmente, eles são representados como `bytes`, por isso a variável `id` foi definida como um slice de `bytes`, que é convertido para uma string (`string(id)`) antes de ser retornado.

Outro detalhe interessante é que, para facilitar a repetição do processo caso um identificador já exista, criamos uma função anônima e a armazenamos na variável `novoId`. Assim, podemos chamá-la mais de uma vez em um laço infinito, que é quebrado quando a palavra-chave `return` força o retorno da função assim que um identificador totalmente novo é gerado.

Para finalizar, como o repositório `repo` é um objeto visível apenas ao pacote `url`, precisamos expor a funcionalidade de buscar uma URL pelo seu identificador curto, da qual o redirecionador depende. Para isso, definimos a função `Buscar()`, que apenas delega a busca ao próprio repositório:

```
func Buscar(id string) *Url {
    return repo.BuscarPorId(id)
}
```

8.8 IMPLEMENTANDO O REPOSITÓRIO EM MEMÓRIA

O repositório em memória, parte do pacote `url`, será definido no arquivo `url/repositorio_memoria.go`, que também começa com a definição do pacote:

```
package url
```

Na prática, este repositório nada mais é do que um mapa cujas chaves são os identificadores curtos, e os valores são ponteiros para os objetos do tipo `Url`. Em outras palavras, um `map[string]*Url`:

```
type repositorioMemoria struct {
    urls map[string]*Url
}
```

Repare que o nome do tipo definido é `repositorioMemoria`, iniciando com uma letra minúscula. Isto garante que a implementação não será conhecida fora do próprio pacote, assegurando o encapsulamento.

No entanto, precisamos fornecer uma maneira de criar objetos deste tipo. Este é o objetivo da função `NovoRepositorioMemoria()`, utilizada pelo servidor para criar e configurar um repositório no início da execução do programa:

```
func NovoRepositorioMemoria() *repositorioMemoria {
    return &repositorioMemoria{make(map[string]*Url)}
}
```

Como a implementação é baseada em um mapa, seus métodos tiram proveito das funcionalidades básicas dos mapas em Go.

Por exemplo, o método `IdExiste()` simplesmente verifica se o identificador especificado é uma chave que existe no mapa:

```
func (r *repositorioMemoria) IdExiste(id string) bool {
    _, existe := r.urls[id]
    return existe
}
```

De maneira semelhante, o método `BuscarPorId()` retorna o valor armazenado sob o identificador recebido, que será `nil` caso a chave não esteja presente no mapa:

```
func (r *repositorioMemoria) BuscarPorId(id string) *Url {
    return r.urls[id]
}
```

Já o método `BuscarPorUrl()` precisa iterar pelos valores armazenados no mapa e compará-los à URL recebida como argumento, retornando o objeto correspondente caso encontre a URL desejada, ou `nil` caso contrário:

```
func (r *repositorioMemoria) BuscarPorUrl(url string) *Url {
    for _, u := range r.urls {
        if u.Destino == url {
            return u
        }
    }
    return nil
}
```

Por fim, o método `Salvar()` armazena uma nova entrada no mapa utilizando o identificador da `Url` como chave, e o endereço do próprio objeto como valor:

```
func (r *repositorioMemoria) Salvar(url Url) error {
    r.urls[url.Id] = &url
```

```
    return nil
}
```

Como esta operação em um mapa não produz nenhum erro (a não ser que não exista mais memória disponível para armazenar este valor), o método `Salvar()` simplesmente retorna `nil`.

Missão cumprida! No próximo capítulo, veremos como executar, compilar e instalar o projeto localmente.

Capítulo 9

COMPILANDO E EXECUTANDO O PROJETO

Go é uma linguagem compilada, assim como C ou C++. Isto significa que não existe uma linguagem intermediária nem uma máquina virtual envolvidas no momento da execução de um programa. Código Go é compilado diretamente para a linguagem da máquina; o artefato gerado pelo processo de compilação é um arquivo binário executável, compatível apenas com o sistema operacional alvo da compilação.

Todos os exemplos apresentados até aqui foram executados através do comando `go run` .

Por exemplo, agora que temos um código funcional para o servidor de URLs curtas, podemos executá-lo utilizando o mesmo comando. Primeiro, certifique-se de que você está no diretório que contém o código do projeto (`$GOPATH/src/github.com/caiofilipini/encurtador`), e então execute o comando a seguir:

```
$ go run servidor.go
```

Se o programa foi compilado com sucesso e o servidor foi capaz de abrir um ::socket:: na porta 8888, nenhuma saída será

gerada e o processo ficará bloqueado. Para ter certeza de que o servidor está no ar, podemos, em uma outra instância do terminal, utilizar o seguinte comando:

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Se você obteve uma saída similar ao executar o comando `telnet`, significa que o servidor está no ar e fomos capazes de nos conectar a ele.

9.1 ENTENDENDO O PROCESSO DE COMPILAÇÃO

O comando `go run` executa o processo completo de compilação sem que você precise se preocupar com o que está acontecendo. Por isso, é um utilitário muito importante durante o desenvolvimento de programas em Go.

Na prática, este comando realiza três operações distintas sequencialmente: compila os arquivos-fonte em arquivos-objeto; realiza o processo de linkedição dos arquivos-objeto, gerando (em um diretório temporário) um executável binário; executa o binário gerado. Este fluxo é bastante similar ao utilizado para compilar programas em C e C++.

As ferramentas distribuídas com a linguagem Go fornecem comandos distintos para compilação e linkedição. Estes comandos possuem nomes diferentes de acordo com a arquitetura do processador alvo da compilação. Por exemplo, caso a arquitetura alvo seja `amd64`, precisamos utilizar o comando `6g` para

compilar os arquivos-fonte, e em seguida o comando `6l` para realizar a linkedição; de maneira análoga, para realizar o mesmo procedimento para processadores com arquitetura `x86`, os comandos são `8g` e `8l`, respectivamente. Essa nomenclatura é herança das ferramentas de compilação do sistema operacional Plan 9, que foram a base para a criação das ferramentas disponíveis em Go.

É importante conhecer estes detalhes para entender o que acontece quando compilamos e executamos um programa em Go. No entanto, é possível realizar todo este processo com um único comando: `go build`.

Podemos gerar um executável do nosso serviço encurtador de URLs executando este comando. Como o código está hospedado em um diretório chamado `encurtador`, o arquivo executável será gerado com o mesmo nome. Assumindo que ainda temos um terminal aberto no diretório `$GOPATH/src/github.com/caiofilipini/encurtador`, podemos gerar um executável para o nosso projeto e executá-lo da seguinte forma:

```
$ go build

$ ls -l ./encurtador
-rwxr-xr-x  1 caio  xxxxx   6.6M Jun  5 22:35 ./encurtador

$ ./encurtador
```

Estes passos reproduzem exatamente o processo realizado quando utilizamos o comando `go run`, com a diferença de que o executável gerado não é descartado quando sua execução é finalizada. O comando `ls` foi adicionado apenas para facilitar a visualização.

9.2 INSTALANDO O EXECUTÁVEL NO SISTEMA

Go também fornece um comando chamado `go install`, que executa o mesmo processo de compilação apresentado anteriormente, porém faz a instalação do programa no ambiente Go local, conforme configurado na variável de ambiente `$GOPATH`.

Por exemplo, considerando que estamos compilando o projeto em um Linux 64 bits, quando executamos `go install` no diretório `encurtador`, dois artefatos são gerados e instalados:

- o pacote `url` é compilado e disponibilizado em `$GOPATH/pkg/linux_amd64/github.com/.../encurtador/url.a` para que possa ser utilizado por outros programas;
- o executável `encurtador`, compatível apenas com Linux 64 bits, é gerado no diretório `$GOPATH/bin`.

Assim, caso o diretório `$GOPATH/bin` tenha sido adicionado à variável de ambiente `$PATH`, é possível executar o servidor de URLs curtas de qualquer lugar do seu sistema.

9.3 APRENDENDO MAIS

O processo de compilação de programas em Go é bastante prático, porém apenas o básico foi apresentado neste capítulo. É possível, por exemplo, gerar executáveis compatíveis com Windows em um computador com Mac OS ou Linux e vice-versa.

Para finalizar, vamos fazer um pequeno exercício. O comando

`go build` possui uma opção `-x` que mostra, passo a passo, todo o processo de compilação. Execute este comando no seu computador, analise os resultados e tente entender o que acontece em cada um dos passos.

```
$ go build -x
```

A documentação completa do comando `go` pode ser encontrada no endereço http://golang.org/cmd/go/.

Capítulo 10

COLHENDO ESTATÍSTICAS

Sistemas encurtadores de URLs foram criados para facilitar o compartilhamento de URLs. Isto se tornou especialmente importante após a popularização de serviços como o Twitter, onde o número de caracteres é limitado, impossibilitando o compartilhamento de URLs muito grandes.

Hoje em dia, os serviços encurtadores de URLs mais populares provêm também uma série de funcionalidades complementares, sendo a coleta e visualização de estatísticas uma das mais importantes.

Vamos criar um mecanismo para coletar estatísticas a respeito das URLs curtas criadas pelo nosso próprio serviço: implementaremos uma simples contagem de quantas vezes cada URL foi acessada.

10.1 REALIZANDO A CONTAGEM NO REPOSITÓRIO

Existem várias formas de implementar a contagem de acessos. Como já implementamos o repositório em memória utilizando um

mapa, podemos reaproveitar a ideia e utilizar também um mapa para armazenar a contagem.

Para facilitar a implementação, vamos adicionar um método à interface `Repositorio` já existente, definida no arquivo `url/url.go`:

```
type Repositorio interface {
    // ...
    RegistrarClick(id string)
}
```

Precisamos também alterar o tipo `repositorioMemoria` para implementar o novo método. Mas antes de adicionar a implementação, precisamos criar um novo mapa para armazenar a contagem de acessos. Para isso, vamos adicioná-lo à `struct repositorioMemoria` no arquivo `url/repositorio_memoria.go`:

```
type repositorioMemoria struct {
    // ...
    clicks map[string]int
}
```

Como adicionamos um novo mapa, precisamos garantir que ele será inicializado na criação do objeto. Portanto, vamos alterar a função `NovoRepositorioMemoria()` para inicializá-lo:

```
func NovoRepositorioMemoria() *repositorioMemoria {
    return &repositorioMemoria{
        make(map[string]*Url),
        make(map[string]int),
    }
}
```

Para facilitar a leitura, quebramos a inicialização em várias linhas.

Finalmente, podemos adicionar a implementação do método `RegistrarClick()` ao tipo `repositorioMemoria` da seguinte forma:

```
func (r *repositorioMemoria) RegistrarClick(id string) {
    r.clicks[id] += 1
}
```

Como o valor armazenado sob uma chave `string` é um `int`, que possui `0` como valor padrão, não precisamos verificar se já existe algum valor armazenado para a chave `id` recebida; quando o primeiro acesso for registrado, a chamada `r.clicks[id]` retornará `0` e, portanto, a expressão completa `r.clicks[id] += 1` resultará no valor `1` sendo armazenado para o `id` recebido.

Para que possamos chamá-lo de fora do pacote `url`, precisamos expor este método. Assim, podemos registrar as requisições recebidas pela função `Redirecionador()` definida no arquivo `servidor.go`. Para isso, vamos alterar o arquivo `url/url.go` e criar uma nova função `RegistrarClick()`. Ela irá apenas delegar a chamada ao repositório, de maneira semelhante à forma como implementamos a função `Buscar()`:

```
func RegistrarClick(id string) {
    repo.RegistrarClick(id)
}
```

Agora o repositório está preparado para realizar a contagem de acessos.

10.2 REGISTRANDO OS ACESSOS NO SERVIDOR

Para registrar cada acesso, precisamos alterar o `servidor.go` para chamar a função `url.RegistrarClick()` a cada requisição recebida.

Como no momento registraremos apenas a quantidade de acessos, poderíamos simplesmente alterar a função `Redirecionador()` e adicionar a chamada imediatamente após a chamada à `http.Redirect()`, resultando no seguinte código:

```
func Redirecionador(w http.ResponseWriter, r *http.Request) {
    // ...

    if url := url.Buscar(id); url != nil {
        http.Redirect(w, r, url.Destino,
            http.StatusMovedPermanently)

        url.RegistrarClick(id)
    }

    // ...
}
```

No entanto, normalmente este tipo de redirecionamento deve ser tratado da maneira mais rápida possível. Caso desejássemos registrar outras métricas no futuro, precisaríamos adicionar diversas outras chamadas a diferentes funções ou métodos, atrasando ainda mais a resposta e piorando muito a experiência do usuário.

Este tipo de problema é muito comum em servidores HTTP. Uma técnica bastante utilizada para resolvê-lo é processar apenas o necessário no momento da requisição, e utilizar algum mecanismo para realizar o restante do trabalho em um momento posterior.

Implementar este tipo de solução em Go é trivial: vamos utilizar uma *goroutine*!

Precisamos de um canal para realizar a comunicação com a goroutine. Como tudo o que precisamos para registrar o acesso é do identificador da URL curta – uma `string` –, criaremos um canal capaz de trafegar `strings`. Este canal deve ser acessível pela própria goroutine, mas também pela função `Redirecionador()`. Podemos declará-lo como uma variável global, da mesma forma que fizemos com as variáveis `porta` e `urlBase`:

```
var (
    porta   int
    urlBase string
    stats   chan string
)
```

Vamos criar uma função que, utilizando o operador `range`, bloqueia a execução até que alguma mensagem seja recebida no canal, e então registra o acesso no repositório e imprime uma linha na saída padrão indicando o sucesso da operação:

```
func registrarEstatisticas(ids <-chan string) {
    for id := range ids {
        url.RegistrarClick(id)
        fmt.Printf("Click registrado com sucesso para %s.\n", id)
    }
}
```

Para que possamos registrar as estatísticas para cada redirecionamento, precisamos disparar a goroutine assim que o servidor for iniciado. Vamos alterar a função `main()` para criar o canal e imediatamente iniciar uma goroutine executando a função `registrarEstatisticas()`:

```
func main() {
    stats = make(chan string)
    defer close(stats)
    go registrarEstatisticas(stats)

    // ...
```

```
}
```

Repare que, assim que o canal foi criado, garantimos que ele será fechado ao final da execução da função `main()` através da chamada `defer close(stats)`.

Para finalizar, vamos alterar a função `Redirecionador()` para enviar o identificador da URL acessada para o canal `stats`, desencadeando o processo que irá registrar este acesso:

```
func Redirecionador(w http.ResponseWriter, r *http.Request) {
    // ...

    if url := url.Buscar(id); url != nil {
        http.Redirect(w, r, url.Destino,
            http.StatusMovedPermanently)

        stats <- id
    }

    // ...
}
```

Agora já temos todo o código necessário para registrar as estatísticas de acesso!

Vamos iniciar o servidor executando a nova versão do código:

```
$ go run servidor.go
```

Em uma outra janela do terminal, vamos criar uma nova URL curta:

```
$ curl -v "http://localhost:8888/api/encurtar" \
-d "http://casadocodigo.com.br/products/livro-jogos-android"
> POST /api/encurtar HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
> Content-Length: 55
> Content-Type: application/x-www-form-urlencoded
```

```
>
< HTTP/1.1 201 Created
< Location: http://localhost:8888/r/9aOFf
< Date: Sat, 07 Jun 2014 19:09:12 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
```

A partir de agora, cada requisição à URL curta deverá incrementar o contador. Vamos realizar algumas requisições e verificar se o registro está sendo feito:

```
$ curl http://localhost:8888/r/9aOFf
$ curl http://localhost:8888/r/9aOFf
$ curl http://localhost:8888/r/9aOFf
```

Para facilitar a visualização, os detalhes destas requisições foram omitidos por completo.

Para cada uma destas requisições, na janela onde o servidor foi iniciado, podemos ver uma linha relacionada indicando o registro do acesso:

```
Click registrado com sucesso para 9aOFf.
Click registrado com sucesso para 9aOFf.
Click registrado com sucesso para 9aOFf.
```

Mas como saber se estamos de fato registrando cada acesso?

Precisamos fornecer alguma maneira de visualizar as estatísticas colhidas. Idealmente, uma API deste tipo deve prover estes dados de uma maneira flexível, permitindo que o cliente decida a melhor forma de apresentá-los.

Como estamos desenvolvendo uma API sobre HTTP, uma forma simples de atingir este resultado é retornar os dados no formato JSON (*JavaScript Object Notation*).

10.3 SERIALIZANDO JSON

Para serializar objetos no formato JSON, Go fornece o pacote `encoding/json`. Basicamente, quase qualquer tipo válido em Go pode ser serializado, incluindo `maps` e `structs`, com a exceção de canais, funções e o tipo `complex` (que representa números complexos).

Serializar uma `struct` utilizando o pacote `encoding/json` é uma tarefa trivial; basta chamar a função `json.Marshal()`, que possui a seguinte assinatura:

```
func Marshal(v interface{}) ([]byte, error)
```

Repare que, para ser capaz de serializar qualquer tipo, a função `Marshal()` recebe um argumento do tipo `interface{}`, denominado `v`. Esta função converte todos os campos **públicos** da `struct v` em suas representações JSON e retorna um slice de `bytes` contendo o JSON serializado, e um `error`, que será `nil` caso a serialização tenha sido realizada com sucesso.

Por exemplo, para serializar um valor do tipo `url.Url`, podemos utilizar o seguinte trecho de código:

```
url := Url{
    "9aOFf",
    time.Now(),
    "http://casadocodigo.com.br/products/livro-jogos-android",
}

json, err := json.Marshal(url)
if err == nil {
    fmt.Println(string(json))
}
```

Assim, o objeto `url` seria convertido para a seguinte

representação JSON (a formatação foi adicionada para facilitar a visualização):

```
{
    "Id":"9aOFf",
    "Criacao":"2009-11-10T23:00:00Z",
    "Destino":
      "http://casadocodigo.com.br/products/livro-jogos-android"
}
```

Simples, não?

O único problema com o exemplo anterior é o nome dos campos – como apenas os campos públicos são serializados, seus nomes são refletidos no resultado gerado e começam com letras maiúsculas; dificilmente encontraremos uma API retornando valores JSON desta forma.

Felizmente, Go provê uma solução para este problema: marcadores de `structs` (ou *struct tags*). Podemos controlar a forma como os campos de uma `struct` serão serializados através da utilização destes marcadores.

Por exemplo, para instruir o método `json.Marshal()` a serializar os campos da `struct Url` com nomes começando em letras minúsculas, precisamos adicionar a ela os seguintes marcadores:

```
type Url struct {
    Id      string    `json:"id"`
    Criacao time.Time `json:"criacao"`
    Destino string    `json:"destino"`
}
```

Desta forma, executando o mesmo trecho de código apresentado anteriormente, obteríamos como resultado o JSON:

```
{
    "id":"9aOFf",
    "criacao":"2009-11-10T23:00:00Z",
    "destino":
      "http://casadocodigo.com.br/products/livro-jogos-android"
}
```

Utilizaremos marcadores deste tipo para serializar as estatísticas de acesso das URLs.

10.4 VISUALIZANDO AS ESTATÍSTICAS COMO JSON

Agora que já sabemos o básico sobre serialização JSON em Go, vamos adicionar uma nova rota ao nosso servidor para retornar as estatísticas de acesso de uma URL neste formato.

Antes de mais nada, precisamos definir uma nova `struct` para representar as estatísticas, já incluindo os marcadores para converter os nomes dos campos para letras minúsculas. Vamos adicionar a definição deste tipo ao arquivo `url/url.go`:

```
type Stats struct {
    Url    *Url `json:"url"`
    Clicks int  `json:"clicks"`
}
```

A `struct Stats` define dois campos: uma referência à `Url` correspondente e um valor `int` que contém a quantidade de acessos a esta `Url`.

Precisamos também adicionar os marcadores à `struct Url`, conforme o exemplo anterior:

```
type Url struct {
    Id      string    `json:"id"`
    Criacao time.Time `json:"criacao"`
```

```
    Destino string    `json:"destino"`
}
```

Para poder retornar o número de acessos armazenado para uma determinada URL, precisamos adicionar um método à interface `Repositorio`:

```
type Repositorio interface {
    // ...
    BuscarClicks(id string) int
}
```

Também precisamos adicionar a implementação deste método ao arquivo `url/repositorio_memoria.go`:

```
func (r *repositorioMemoria) BuscarClicks(id string) int {
    return r.clicks[id]
}
```

Este método retornará a quantidade de acessos registrados para a URL identificada por `id`. Caso nenhum acesso tenha sido registrado para esta URL, a expressão `r.clicks[id]` retornará `0` – o valor padrão do tipo `int`.

Para criar uma interface mais natural, vamos agora adicionar um método à `struct Url` para retornar as estatísticas referentes a uma URL. Primeiro, obtemos o número de acessos registrados para esta URL chamando o recém-criado método `repo.BuscarClicks()`; depois, criamos um novo objeto `Stats`, associando a `Url` e injetando o número de acessos:

```
func (u *Url) Stats() *Stats {
    clicks := repo.BuscarClicks(u.Id)
    return &Stats{u, clicks}
}
```

Agora precisamos criar uma nova rota ao `servidor.go` para retornar o JSON contendo as estatísticas de uma determinada

URL. Primeiro, vamos adicionar a nova rota à configuração do servidor na função `main()`, antes da chamada à `http.ListenAndServe()`:

```
func main() {
    // ...

    http.HandleFunc("/api/stats/", Visualizador)

    // ...
}
```

A função `Visualizador()` é muito similar à `Redirecionador()`, portanto, vamos direto à implementação:

```
func Visualizador(w http.ResponseWriter, r *http.Request) {
    caminho := strings.Split(r.URL.Path, "/")
    id := caminho[len(caminho)-1]

    if url := url.Buscar(id); url != nil {
        json, err := json.Marshal(url.Stats())

        if err != nil {
            w.WriteHeader(http.StatusInternalServerError)
            return
        }

        responderComJSON(w, string(json))
    } else {
        http.NotFound(w, r)
    }
}
```

Começamos extraindo o identificador da URL do caminho da requisição, da mesma forma que fizemos na função `Redirecionador()`. Em seguida, chamamos a função `url.Buscar()` para obter os dados da URL. Se a URL for encontrada, utilizamos a função `json.Marshal()` para serializar o resultado da chamada `url.Stats()` (i.e. um valor do tipo

`url.Stats`).

Caso um erro ocorra durante a serialização, interrompemos o processamento e retornamos um erro HTTP 500.

Já em caso de sucesso, convertemos o slice `json` em uma `string` e chamamos a função `responderComJSON()` para formatar a resposta.

Esta função simplesmente chama a função `responderCom()`, configurando o valor `application/json` para o cabeçalho `Content-Type` e escrevendo o JSON recebido no corpo da resposta:

```
func responderComJSON(w http.ResponseWriter, resposta string) {
    responderCom(w, http.StatusOK, Headers{
        "Content-Type": "application/json",
    })
    fmt.Fprintf(w, resposta)
}
```

Por fim, se o identificador recebido não corresponder a uma URL curta existente, retornamos um erro HTTP 404.

O último passo necessário para completar esta funcionalidade é criar alguma ligação entre a URL criada e suas estatísticas. Em uma API *RESTful*, quando um recurso é criado, a resposta à requisição que o criou deve incluir de alguma forma este tipo de ligação. Por exemplo, através do cabeçalho `Link`.

Vamos alterar a função `Encurtador()` para incluir este cabeçalho com o endereço das estatísticas da URL recém-criada. Precisamos modificar a chamada à função `responderCom()` da seguinte forma:

```
func Encurtador() {
```

```
    // ...

    responderCom(w, http.StatusCreated, Headers{
        "Location": urlCurta,
        "Link": fmt.Sprintf("<%s/api/stats/%s>; rel=\"stats\"",
            urlBase, url.Id),

    })
}
```

Agora podemos reiniciar o servidor e visualizar algumas estatísticas:

```
$ go run servidor.go
```

Vamos repetir o processo realizado anteriormente para criar uma nova URL curta, fazer algumas requisições a esta URL e, então, visualizar suas estatísticas:

```
$ curl -v "http://localhost:8888/api/encurtar" \
-d "http://casadocodigo.com.br/products/livro-devops"
> POST /api/encurtar HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8888
> Accept: */*
> Content-Length: 48
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 201 Created
< Link: <http://localhost:8888/api/stats/Cp+Hf>; rel="stats"
< Location: http://localhost:8888/r/Cp+Hf
< Date: Sun, 08 Jun 2014 22:08:24 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
```

Repare que retornamos o cabeçalho `Link` com o endereço para acessar as estatísticas da URL recém-criada. Requisitando esta URL, teremos o seguinte resultado (mais uma vez formatado para facilitar a visualização):

```
$ curl http://localhost:8888/api/stats/Cp+Hf
{
    "url":{
        "id":"Cp+Hf",
        "criacao":"2014-06-09T00:08:24.899606706+02:00",
        "destino":
          "http://casadocodigo.com.br/products/livro-devops"
    },
    "clicks":0
}
```

Nenhum acesso foi registrado por enquanto. Vamos fazer algumas requisições para a URL curta:

```
$ curl http://localhost:8888/r/Cp+Hf
$ curl http://localhost:8888/r/Cp+Hf
$ curl http://localhost:8888/r/Cp+Hf
$ curl http://localhost:8888/r/Cp+Hf
```

Agora, se acessarmos novamente as estatísticas desta URL, veremos a seguinte resposta:

```
$ curl http://localhost:8888/api/stats/Cp+Hf
{
    "url":{
        "id":"Cp+Hf",
        "criacao":"2014-06-09T00:08:24.899606706+02:00",
        "destino":
          "http://casadocodigo.com.br/products/livro-devops"
    },
    "clicks":4
}
```

Registramos corretamente os 4 acessos. Nossa API está finalmente completa!

Um detalhe importante é que, como requisições de redirecionamento podem ser (e normalmente são) armazenadas em cache, acessar as URLs curtas em um navegador pode produzir resultados diferentes na contagem de acessos.

No próximo capítulo, veremos como melhorar o código produzido até aqui.

Capítulo 11

REFATORANDO O CÓDIGO

Depois de implementar todas as funcionalidades no nosso servidor de URLs curtas, podemos analisar um pouco o código produzido e identificar alguns pontos de melhoria, como por exemplo:

- Substituir variáveis globais onde possível;
- Reduzir duplicação de código;
- Introduzir logs;
- Tornar a inicialização mais flexível.

Vamos abordar cada um dos itens desta lista de forma separada.

11.1 SUBSTITUINDO VARIÁVEIS GLOBAIS

Utilizar variáveis globais é uma forma muito prática de compartilhar recursos entre diferentes funções, objetos e métodos em um programa. No entanto, alguns destes valores podem ser bastante sensíveis, e permitir acesso global a eles pode introduzir uma série de problemas.

Nosso servidor define três variáveis globais: `porta`, `urlBase` e o canal `stats`. As variáveis `porta` e `urlBase` poderiam ser tratadas como constantes, pois são atribuídas durante a inicialização do programa e não sofrem alterações durante sua execução. Porém, em Go, não é possível declarar uma constante sem inicializá-la imediatamente, e um dos pontos de melhoria é tornar a inicialização do servidor mais flexível, recebendo a porta como argumento, por exemplo. Isto impede que a variável `porta` seja uma constante e, como a variável `urlBase` depende do valor definido para `porta`, ela também não pode ser uma constante.

Por outro lado, o canal `stats` foi declarado como global apenas por conveniência. Atualmente, ele já é passado como argumento para a função `registrarEstatisticas()`, o que nos deixa com apenas um ponto de mudança: precisamos injetá-lo de alguma forma no contexto da função `Redirecionador()`.

Como esta função é utilizada para atender requisições HTTP e registrada através da função `http.HandleFunc()`, ela deve obrigatoriamente seguir a assinatura definida pelo tipo `http.HandlerFunc`. Isso impede que adicionemos qualquer novo argumento a esta função.

No entanto, o pacote `http` fornece uma outra forma para registrar *handlers*: a função `http.Handle()`. Esta função possui a seguinte assinatura:

```
func Handle(pattern string, handler Handler)
```

Veja que o segundo argumento é do tipo `http.Handler`, e não mais `http.HandlerFunc`. Este tipo é uma interface que define apenas um método:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Podemos então criar um novo tipo que implemente o método `ServeHTTP()` e, desta forma, registrá-lo através da função `http.Handle()`. Vamos introduzir o tipo `Redirecionador` ao arquivo `servidor.go` como uma `struct` vazia:

```
type Redirecionador struct{}
```

Agora vamos alterar a função `Redirecionador` para torná-la compatível com o tipo `http.Handler`. Para isso, precisamos defini-la como um método no tipo `Redirecionador` (i.e. definir o tipo `*Redirecionador` como receptor do método), e também a renomear para `ServeHTTP`. O corpo da função deve permanecer inalterado:

```
func (red *Redirecionador) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request,
) {
    // ...
}
```

Por fim, precisamos alterar a função `main()` para registrar o tipo `Redirecionador`. Vamos alterar a linha que registra a rota `/r/` da seguinte forma:

```
func main() {
    // ...

    http.Handle("/r/", &Redirecionador{})

    // ...
}
```

Na prática, essas alterações não afetam o funcionamento do servidor. Reinicie-o e faça algumas requisições para ter certeza de

que tudo continua funcionando.

Esta nova estrutura, porém, permite que nos livremos facilmente do canal como variável global. Primeiro, vamos introduzir um campo na `struct Redirecionador` para armazenar o canal:

```
type Redirecionador struct {
    stats chan string
}
```

Agora podemos remover a declaração do canal global `stats`.

Como este canal é criado na função `main()`, exatamente o mesmo local onde registramos o tipo `Redirecionador` para atender requisições, podemos injetá-lo no momento da criação do objeto do tipo `Redirecionador`. Também precisamos alterar a criação do canal para declará-lo como uma variável local à função `main()`:

```
func main() {
    stats := make(chan string)
    // ...

    http.Handle("/r/", &Redirecionador{stats})

    // ...
}
```

Por fim, precisamos alterar o método `ServeHTTP()` para acessar o canal através do objeto receptor:

```
func (red *Redirecionador) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request,
) {
    // ...

    if url := url.Buscar(id); url != nil {
```

```
        http.Redirect(w, r, url.Destino,
            http.StatusMovedPermanently)

        red.stats <- id
    } else {

    // ...
}
```

Repare que, ao enviar o identificador da URL para o canal, agora utilizamos a forma `red.stats <- id`, acessando o canal através do receptor `red`.

Livramo-nos da variável global!

Nosso próximo desafio é reduzir a duplicação de código.

11.2 REDUZINDO A DUPLICAÇÃO DE CÓDIGO

Quando introduzimos a função `Visualizador()` para retornar as estatísticas em formato JSON, acabamos com alguns trechos de código duplicados entre esta nova função e a função `ServeHTTP` do tipo `Redirecionador`.

Analisando as duas com cuidado, encontramos um padrão comum: ambas extraem o identificador recebido no caminho da requisição e buscam por uma URL correspondente àquele identificador; caso a URL seja encontrada, alguma ação é executada baseada no objeto `url` retornado; caso contrário, um erro HTTP 404 é devolvido.

Podemos extrair uma função que abstrai toda esta lógica e, caso a URL buscada seja encontrada, executa uma função passando o objeto retornado como argumento. Vamos chamá-la de

`buscarUrlEExecutar()`:

```
func buscarUrlEExecutar(
    w http.ResponseWriter,
    r *http.Request,
    executor func(*url.Url),
) {
    caminho := strings.Split(r.URL.Path, "/")
    id := caminho[len(caminho)-1]

    if url := url.Buscar(id); url != nil {
        executor(url)
    } else {
        http.NotFound(w, r)
    }
}
```

Repare que esta função implementa exatamente a mesma lógica descrita anteriormente. O terceiro argumento recebido é uma função denominada `executor`, que recebe um ponteiro para um valor do tipo `url.Url` (i.e., o valor retornado pela função `url.Buscar()`). Esta função será chamada caso a URL seja encontrada.

Vamos agora alterar o método `ServeHTTP()` do tipo `Redirecionador` para utilizar esta nova função:

```
func (red *Redirecionador) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request,
) {
    buscarUrlEExecutar(w, r, func(url *url.Url) {
        http.Redirect(w, r, url.Destino,
            http.StatusMovedPermanently)
        red.stats <- url.Id
    })
}
```

Por fim, vamos alterar também a função `Visualizador`, de maneira similar:

```
func Visualizador(w http.ResponseWriter, r *http.Request) {
    buscarUrlEExecutar(w, r, func(url *url.Url) {
        json, err := json.Marshal(url.Stats())

        if err != nil {
            w.WriteHeader(http.StatusInternalServerError)
            return
        }

        responderComJSON(w, string(json))
    })
}
```

O resultado final é um código mais focado e melhor fatorado, permitindo que alterações na maneira como recebemos o identificador da URL, ou até mesmo na forma como retornamos erros HTTP 404, sejam feitas em um único lugar.

11.3 ESCREVENDO LOGS

Escrever logs para cada ação realizada por uma aplicação é um recurso extremamente importante para identificar problemas ou mesmo rastrear requisições.

Go fornece alguns recursos simples para manipulação de logs no pacote `log`.

Antes de mais nada, precisamos adicionar o pacote `log` à lista de dependências do nosso pacote `main`:

```
import (
    // ...
    "log"
    // ...
)
```

Para facilitar, vamos criar uma função utilitária no arquivo `servidor.go` para nos ajudar a logar algumas ações:

```
func logar(formato string, valores ...interface{}) {
    log.Printf(fmt.Sprintf("%s\n", formato), valores...)
}
```

Todas as mensagens impressas através das funções do pacote `log` serão precedidas pela data e hora na qual foram escritas. Vamos introduzir algumas chamadas a funções desse pacote ao nosso servidor, começando pela `main()`:

```
func main() {
    // ...

    logar("Iniciando servidor na porta %d...", *porta)
    log.Fatal(http.ListenAndServe(
        fmt.Sprintf(":%d", *porta), nil))
}
```

Primeiro chamamos a função `logar()` para escrever uma mensagem indicando a porta na qual o servidor aguardará por conexões.

A função `http.ListenAndServe()` bloqueia a execução quando é capaz de iniciar o servidor com sucesso. Em caso de erro, no entanto, ela retorna um `error` que estávamos ignorando até agora. Envolvê-la em uma chamada à `log.Fatal()` é importante por dois motivos: primeiro, caso um erro ocorra ao iniciar o servidor, saberemos exatamente qual foi a causa do erro; segundo, a função `log.Fatal()` imprime uma entrada no log contendo a data e hora atuais e a mensagem de erro especificada (neste caso, o erro ocorrido ao iniciar o servidor), e imediatamente encerra a execução do programa retornando um código de erro para o sistema operacional.

Por exemplo, caso um outro processo servidor já tenha sido iniciado na porta 8888, se tentarmos iniciá-lo novamente, veremos

o seguinte erro no log:

```
$ go run servidor.go
2014/06/09 15:50:50 Iniciando servidor na porta 8888...
2014/06/09 15:50:50 listen tcp :8888: bind: address already in
use
exit status 1
```

Vamos agora adicionar uma mensagem de log quando uma nova URL curta é criada. Para isso, vamos incluir a seguinte chamada ao final da função `Encurtador()`:

```
func Encurtador(w http.ResponseWriter, r *http.Request) {
    // ...

    logar("URL %s encurtada com sucesso para %s.",
        url.Destino, urlCurta)
}
```

A partir de agora, quando encurtarmos uma URL, veremos uma nova entrada no log, similar a esta:

```
2014/06/09 15:56:43 URL
http://casadocodigo.com.br/products/livro-ruby
encurtada com sucesso para http://localhost:8888/r/rsHj_.
```

Finalmente, quando registramos um acesso a uma determinada URL, na função `registrarEstatisticas()` fazemos uma chamada à `fmt.Printf()` que pode ser substituída por uma chamada à função `logar()`:

```
func registrarEstatisticas(ids <-chan string) {
    for id := range ids {
        url.RegistrarClick(id)

        logar("Click registrado com sucesso para %s.", id)
    }
}
```

Desta forma, quando registrarmos um acesso, a mensagem

logada incluirá também a data e hora de quando o acesso ocorreu:

```
2014/06/09 16:08:43 Click registrado com sucesso para rsHj_.
```

Nosso servidor agora imprime mensagens de log para cada ação importante executada.

11.4 FLEXIBILIZANDO A INICIALIZAÇÃO DO SERVIDOR

Para tornar nosso servidor mais flexível, é importante que algumas configurações possam ser modificadas sem que o código precise ser alterado. Por exemplo, o número da porta onde o servidor aceitará conexões. Além disso, agora que introduzimos logs, podemos também incluir uma configuração para ligar ou desligar a geração de logs.

Existem diversas formas de atingir este objetivo, como o uso de variáveis de ambiente, arquivos de configuração externos ou argumentos de linha de comando.

Já vimos como receber e tratar argumentos de linha de comando anteriormente através do pacote `os`. No entanto, existe um outro pacote que fornece recursos que facilitam o tratamento destes argumentos: o pacote `flag`.

Para utilizá-lo, precisamos adicioná-lo à lista de dependências:

```
import (
    // ...
    "flag"
    // ...
)
```

O pacote `flag` provê uma série de funções utilitárias para

tratar as opções de linha de comando. Por exemplo, para definir uma opção `-p` que receberá o número da porta, podemos utilizar o código a seguir:

```
porta := flag.Int("p,", 8888, "porta")
```

Desta forma, definimos uma opção `-p` que deve ser um número inteiro. Caso nenhum valor seja especificado, o valor 8888 será usado. O último argumento define o nome da opção, e será utilizado para gerar uma mensagem de ajuda. Veremos um exemplo em breve.

Um detalhe importante é que as funções utilitárias do pacote `flag` para definir opções retornam **ponteiros**. Portanto, para transformarmos nossa variável global `porta` em uma `flag`, precisamos alterar seu tipo para `*int`. Vamos aproveitar e criar uma nova variável que irá definir se a geração de logs deverá ser ligada ou desligada:

```
var (
    porta     *int
    logLigado *bool
    urlBase   string
)
```

Em seguida, vamos alterar a função `init()` para configurar as opções:

```
func init() {
    porta = flag.Int("p", 8888, "porta")
    logLigado = flag.Bool("l", true, "log ligado/desligado")

    flag.Parse()

    urlBase = fmt.Sprintf("http://localhost:%d", *porta)
}
```

Repare que incluímos uma chamada à função `flag.Parse()`.

Ela efetivamente lê os argumentos recebidos na inicialização do programa e os valida de acordo com as opções definidas **antes** da chamada à `flag.Parse()`. É importante notar também que, de agora em diante, quando precisarmos dos valores reais armazenados nas variáveis `porta` e `logLigado`, precisamos utilizar o operador de indireção, pois elas são ponteiros. Por este motivo, ao configurar a `urlBase`, usamos `*porta` como o número da porta.

Se iniciarmos o servidor especificando a opção `-h`, veremos a mensagem de ajuda gerada pelo pacote `flag`:

```
$ go build
$ ./encurtador -h
Usage of ./encurtador:
-l=true: log ligado/desligado
-p=8888: porta
```

Agora, se tentarmos especificar uma porta inválida, veremos o seguinte erro:

```
$ ./encurtador -p=abcd
invalid value "abcd" for flag -p: strconv.ParseInt:
    parsing "abcd": invalid syntax
Usage of ./encurtador:
-l=true: log ligado/desligado
-p=8888: porta
```

Antes de continuarmos, precisamos garantir que aplicamos o operador de indireção em todos os lugares onde a variável `porta` é utilizada. Precisamos alterar as duas últimas linhas da função `main()` da seguinte forma:

```
func main() {
    // ...

    logar("Iniciando servidor na porta %d...", *porta)
    log.Fatal(http.ListenAndServe(
```

```
        fmt.Sprintf(":%d", *porta), nil))
}
```

Por fim, vamos alterar a função `logar()` para escrever as mensagens apenas se a opção `-l` foi configurada com valor `true`:

```
func logar(formato string, valores ...interface{}) {
    if *logLigado {
        log.Printf(fmt.Sprintf("%s\n", formato), valores...)
    }
}
```

Inicie o servidor com a opção `-l=false` e repare que nenhuma mensagem de log será gerada. De maneira similar, especifique diferentes portas através da opção `-p` e veja se tudo continua funcionando.

Nosso servidor de URLs curtas está finalmente completo!

Capítulo 12

PRÓXIMOS PASSOS

Go é uma linguagem moderna, criada com o objetivo principal de melhorar a produtividade no desenvolvimento em larga escala de servidores, baseada nas experiências de alguns times dentro do Google.

Hoje, mais de quatro anos após ter sido anunciada publicamente, cada vez mais empresas têm adotado a linguagem para escrever diversos tipos diferentes de aplicações, muito além dos servidores.

Enquanto o conteúdo apresentado neste livro cobre algumas de suas funcionalidades básicas, incluindo partes importantes da biblioteca padrão, ainda há muito a ser aprendido.

Para discutir sobre o conteúdo e os exemplos deste livro, participe da lista de discussão disponível em http://forum.casadocodigo.com.br/.

Além disso, todos os exemplos podem ser encontrados no GitHub, basta acessar https://github.com/caiofilipini/casadocodigo-go.

O projeto do encurtador de URLs desenvolvido nos capítulos finais também está disponível em

https://github.com/caiofilipini/encurtador.

12.1 APRENDENDO MAIS

Para continuar aprendendo sobre a linguagem Go, você encontrará a seguir alguns links selecionados (todos em inglês):

- Site oficial: http://golang.org/
- Índice de pacotes da biblioteca padrão: http://golang.org/pkg/
- Especificação da linguagem: http://golang.org/ref/spec
- *Effective Go*: http://golang.org/doc/effective_go.html
- Blog oficial, contendo diversos artigos interessantes: http://blog.golang.org/
- Índice geral da documentação oficial: http://golang.org/doc/

Capítulo 13

REFERÊNCIAS BIBLIOGRÁFICAS

FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures.* 2000. Disponível em: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

HOARE, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.