

Если вы видите что-то необычное, просто сообщите мне.

# Первые шаги на Go — Построение простого веб приложения с Neo4j

## Цель:

Цель этого поста создать простое веб приложение которое снимает структуру вашего сайта получая все ссылки которые содержит и сохраняет их в neo4j базу данных. Идея проста - и в ней следующие шаги:

- Делаем запрос на URL
- парсим ответ
- Извлекаем ссылки из ответа
- Сохраняем извлеченные ссылки в neo4j
- Повторяем 1 шаг с полученными ссылками пока не исследуем весь сайт
- Наконец используем Neo4j веб интерфейс чтобы посмотреть на структуру.

## Требования:

Эта статья подойдет начинающим. Будут приведены ссылки, каждый раз, когда будет представлена новая идея. Для Neo4j, базовое знание графово ориентированной базы данных будет к месту. Предполагается что Go и Neo4j уже установлены на машинет.

## Создаем ползунок:

Теперь, когда у нас все есть. Начнем.

# Получение одной страницы из интернета:

Время написать сожный код, которы поможет нам получить определенную страницу из интернета.

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

type responseWriter struct{}

func main() {

    resp, err := http.Get("http://www.sfeir.com")

    if err != nil {
        fmt.Println("Error:", err)
        os.Exit(1)
    }

    rw := responseWriter{}
    io.Copy(rw, resp.Body)
}

func (responseWriter) Write(bs []byte) (int, error) {
    fmt.Printf(string(bs))

    return len(bs), nil
}
```

Мы начали с объявления главного пакета и импорта требуемых пакетов. Дальше, мы объявили структуру которая будет реализовывать `Writer` интерфейс. В `main` функции, мы собираемся присвоить множеству переменных значения. В основном, `http.Get` будет возвращать значения с ответом и некоторой ошибкой, если что-то пойдет не так. Это общий способ обработки ошибок в Go программах.

Если вы посмотрите на документацию, вы найдете `Writer` интерфейс с одной функцией. Для того, чтобы реализовать этот интерфейс, нам нужно добавить получателя функции к нашей `responseWriter` структуре которая совпадает с функцией `Writer`. Если вы пришли из Java вы должны ожидать синтаксис типа `реализация Writer`. Ну чтож, это не тот случай для Go, так как тут реализация происходит неявно.

Наконец, мы используем `io.Copy` для записи тела ответа в нашу переменную ответа. Следующий шаг это модификация нашего кода для извлечения ссылок из данного адреса страницы. После некоторого рефакторинга, у нас будет два файла. Это `main.go`:

```
package main

import (
    "fmt"
    "os"
)

func main() {

    if len(os.Args) < 2 {
        fmt.Println("Web site url is missing")
        os.Exit(1)
    }

    url := os.Args[1]

    retrieve(url)
}
```

И этот `retriever.go`:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

type responseWriter struct{}

func (responseWriter) Write(bs []byte) (int, error) {
    fmt.Printf(string(bs))

    return len(bs), nil
}

func retrieve(uri string) {
    resp, err := http.Get(uri)

    if err != nil {
        fmt.Println("Error:", err)
        os.Exit(1)
    }

    rw := responseWriter{}
    io.Copy(rw, resp.Body)
}
```

Мы можем запустить это для небольшого вебсайта:

```
go run main.go retriever.go http://www.sfeir.com
```

Теперь мы сделали наш первый шаг для создание ползунка. Есть возможность загрузить и спарсить данный адрес, открыть подключение прям к удлённому хосту, и получить html содержание.

# Создадим все гиперссылки для одной страницы

Теперь начинается часть где нам нужно извлечь все ссылки из html документа. К сожалению, нет доступного для этого помощника для обработки HTML в Go API. Поэтому мы должны посмотреть на стороннюю API. Давайте рассмотрим `goquery`. Как вы можете догадаться, она похожа на `jquery` только для Go.

Пакет `goquery` легко получается командой:

```
go get github.com/PuerkitoBio/goquery
```

```
package main

import (
    "fmt"
    "os"
)

func main() {

    if len(os.Args) < 2 {
        fmt.Println("Web site url is missing")
        os.Exit(1)
    }

    url := os.Args[1]

    links, err := retrieve(url)

    if err != nil {
        fmt.Println("Error:", err)
        os.Exit(1)
    }

    for _, link := range links {
        fmt.Println(link)
    }
}
```

```
}  
}
```

Я изменил нашу `retrieve` функцию, таким образом, чтобы она возвращала ссылки данные на странице.

```
package main  
  
import (  
    "fmt"  
    "net/http"  
    "net/url"  
    "strings"  
  
    "github.com/PuerkitoBio/goquery"  
)  
  
func retrieve(uri string) ([]string, error) {  
    resp, err := http.Get(uri)  
    if err != nil {  
        fmt.Println("Error:", err)  
        return nil, err  
    }  
  
    doc, readerErr := goquery.NewDocumentFromReader(resp.Body)  
    if readerErr != nil {  
        fmt.Println("Error:", readerErr)  
        return nil, readerErr  
    }  
    u, parseErr := url.Parse(uri)  
    if parseErr != nil {  
        fmt.Println("Error:", parseErr)  
        return nil, parseErr  
    }  
    host := u.Host  
  
    links := []string{}  
    doc.Find("a[href]").Each(func(index int, item *goquery.Selection) {  
        href, _ := item.Attr("href")  
        lu, err := url.Parse(href)
```

```

    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    if isInternalURL(host, lu) {
        links = append(links, u.ResolveReference(lu).String())
    }

})

return unique(links), nil
}

// insures that the link is internal
func isInternalURL(host string, lu *url.URL) bool {

    if lu.IsAbs() {
        return strings.EqualFold(host, lu.Host)
    }
    return len(lu.Host) == 0
}

// insures that there is no repetition
func unique(s []string) []string {
    keys := make(map[string]bool)
    list := []string{}
    for _, entry := range s {
        if _, value := keys[entry]; !value {
            keys[entry] = true
            list = append(list, entry)
        }
    }
    return list
}

```

Как можно увидеть, наша `retrieve` функция стала существенно улучшена. Я убрал `responseWriter` структуру так как она больше не нужна из-за `goquery` имеет свою реализацию интерфейса `Writer`. Я так же добавил две функции помощников. Первая - определяет URL указывающий на внутреннюю страницу. Вторая - проверяет, что список не содержит

дубликатов ссылок.

Вновь запустим программу для простого сайта:

```
go run main.go retriever.go http://www.sfeir.com
```

## Получаем все гиперссылки для всего сайта.

Ура! Мы сделали большую работу. Следующее, мы собираемся посмотреть, как улучшить `retrieve` функцию для того, чтобы получить ссылки с других страниц, в том числе. Я предлагаю рассмотреть использование рекурсии. Мы создадим другую функцию под названием `crawl` и эта функция будет вызывать себя рекурсивно, с каждой следующей полученной ссылкой. Так же, нам нужно отслеживать посещенные страницы, чтобы избежать повторных переходов по ссылкам. Проверим:

```
// part of retriever.go
var visited = make(map[string]bool)

func crawl(uri string) {

    links, _ := retrieve(uri)

    for _, l := range links {
        if !visited[l] {
            fmt.Println("Fetching", l)
            visited[l] = true
            crawl(l)
        }
    }
}
```

Теперь можно вызывать `crawl` вместо `retrieve` функции в `main.go`. Код будет следующим.

```
package main

import (
```

```
    []"fmt"  
    []"os"  
    )  
  
    func main() {  
  
        []if len(os.Args) < 2 {  
            [][]fmt.Println("Web site url is missing")  
            [][]os.Exit(1)  
            []}  
  
            []url := os.Args[1]  
  
            []crawl(url)  
  
        }  
    }  
}
```

Запустим нашу программу:

```
go run main.go retriever.go http://www.sfeir.com
```

## Реализуем слушателя событий через каналы.

В прошлой части, мы увидели как полученные URL отображаются внутри `crawl` функции. Это не лучшее решение особенно когда вам нужно делать больше чем просто вывод на экране. Чтобы это исправить, в основном, нам нужно реализовать слушателя событий для получения URL через каналы. Давайте посмотрим на это:

```
// same imports  
type link struct {  
    []source string  
    []target string  
}  
  
type retriever struct {  
    []events map[string][]chan link
```

```

    []visited map[string]bool
}

func (b *retriever) addEvent(e string, ch chan link) {
    []if b.events == nil {
        [][]b.events = make(map[string][]chan link)
    []}
    []if _, ok := b.events[e]; ok {
        [][]b.events[e] = append(b.events[e], ch)
    []} else {
        [][]b.events[e] = []chan link{ch}
    []}
}

func (b *retriever) removeEvent(e string, ch chan link) {
    []if _, ok := b.events[e]; ok {
        [][]for i := range b.events[e] {
            [][]if b.events[e][i] == ch {
                [][][]b.events[e] = append(b.events[e][:i], b.events[e][i+1:]...)
                [][][]break
            [][]}
        []}
    []}
}

func (b *retriever) emit(e string, response link) {
    []if _, ok := b.events[e]; ok {
        [][]for _, handler := range b.events[e] {
            [][]go func(handler chan link) {
                [][][]handler <- response
            [][]}(handler)
        []}
    []}
}

func (b *retriever) crawl(uri string) {

    []links, _ := b.retrieve(uri)

    []for _, l := range links {

```

```

    if !b.visited[l] {
        b.emit("newLink", link{
            source: uri,
            target: l,
        })
        b.visited[uri] = true
        b.crawl(l)
    }
}

func (b *retriever) retrieve(uri string) ([]string, error) {
    resp, err := http.Get(uri)
    if err != nil {
        fmt.Println("Error:", err)
        return nil, err
    }

    doc, readerErr := goquery.NewDocumentFromReader(resp.Body)
    if readerErr != nil {
        fmt.Println("Error:", readerErr)
        return nil, readerErr
    }

    u, parseErr := url.Parse(uri)
    if parseErr != nil {
        fmt.Println("Error:", parseErr)
        return nil, parseErr
    }

    host := u.Host

    links := []string{}
    doc.Find("a[href]").Each(func(index int, item *goquery.Selection) {
        href, _ := item.Attr("href")
        lu, err := url.Parse(href)
        if err != nil {
            fmt.Println("Error:", err)
            return
        }

        if isInternalURL(host, lu) {
            links = append(links, u.ResolveReference(lu).String())
        }
    })
}

```

```

    []}

    []})

    []return unique(links), nil
    }

    // same helper functions

```

Как можно увидеть, у нас есть дополнительные функции, для управления событиями для данного `retriever`. Для этого кода я использовал `go` ключевое слово. В основном, написание `go foo()` запустит функцию `foo` запуситься асинхронно. В нашем случае мы использовали `go` которая является анонимной функцией, чтобы послать параметр события(ссылку) всем слушателям через канал. Я указал тип канала данных как `link`, который содержит источник и целевую страницу.

Теперь давайте взглянем на `main` функцию:

```

package main

import (
    []"fmt"
    []"os"
)

func main() {

    []if len(os.Args) < 2 {
        [][]fmt.Println("Web site url is missing")
        [][]os.Exit(1)
    []}

    []url := os.Args[1]
    []ev := make(chan link)
    []r := retriever{visited: make(map[string]bool)}
    []r.addEvent("newLink", ev)

    []go func() {
        [][]for {

```

```
l := <-ev
fmt.Println(l.source + " -> " + l.target)
}
}()

r.crawl(url)

}
```

Вновь используя `go` ключевое слово, в этот раз чтобы получать параметры события отправленные `crawl` функцией. Если мы запустим нашу программу, теперь мы должны увидеть все внутренние ссылки для данного сайта.

Этого достаточно для ползунка.

## Интеграция с Neo4j

Мы закончили с ползунком, давайте перейдем к части с Neo4j. Первая вещь, которую мы собираемся сделать это установить драйвер.

```
go get github.com/neo4j/neo4j-go-driver/neo4j
```

После установки драйвера, нам нужно создать некую базовую функцию которая позволит нам работать с Neo4j. Создадим файл под названием `neo4j.go`:

```
package main

import (
    "github.com/neo4j/neo4j-go-driver/neo4j"
)

func connectToNeo4j() (neo4j.Driver, neo4j.Session, error) {

    configForNeo4j40 := func(conf *neo4j.Config) { conf.Encrypted = false }

    driver, err := neo4j.NewDriver("bolt://localhost:7687", neo4j.BasicAuth(
        "neo4j", "alice!in!wonderland", ""), configForNeo4j40)
```

```

    if err != nil {
        return nil, nil, err
    }

    sessionConfig := neo4j.SessionConfig{AccessMode: neo4j.AccessModeWrite}
    session, err := driver.NewSession(sessionConfig)
    if err != nil {
        return nil, nil, err
    }

    return driver, session, nil
}

func createNode(session *neo4j.Session, l *link) (neo4j.Result, error) {
    r, err := (*session).Run("CREATE (:WebLink{source: $source, target: $target})",
        map[string]interface{}{
            "source": l.source,
            "target": l.target,
        })

    if err != nil {
        return nil, err
    }

    return r, err
}

func createNodesRelationship(session *neo4j.Session) (neo4j.Result, error) {
    r, err := (*session).Run("MATCH (a:WebLink),(b:WebLink) WHERE a.target = b.source CREATE (a)-
        [r:point_to]->(b)", map[string]interface{}{})

    if err != nil {
        return nil, err
    }

    return r, err
}

```

В основе, мы имеет три функции отвечающие за инициализацию подключения Neo4j с базовым запросом. Вам нужно поменять Neo4j конфигурацию для работы с локальной установкой.

Чтобы создать `WebLink` ноду нам просто нужно запустить следующий запрос:

```
CREATE (:WebLink{source: "http://www.sfeir.com/", target: "http://www.sfeir.com/en/services"})
```

Как только нода будет создана, нам нужно создать отношение между ними запустив следующий запрос:

```
MATCH (a:WebLink),(b:WebLink)
WHERE a.target = b.source
CREATE (a)-[r:point_to]->(b)
```

Давайте обновим нашу `main` функцию.

```
package main

import (
    "fmt"
    "os"

    "github.com/neo4j/neo4j-go-driver/neo4j"
)

func main() {

    if len(os.Args) < 2 {
        fmt.Println("Web site url is missing")
        os.Exit(1)
    }

    driver, session, connErr := connectToNeo4j()

    if connErr != nil {
        fmt.Println("Error connecting to Database:", connErr)
        os.Exit(1)
    }
}
```

```

defer driver.Close()

defer session.Close()

url := os.Args[1]
ev := make(chan link)
r := retriever{visited: make(map[string]bool)}
r.addEvent("newLink", ev)

go func(session *neo4j.Session) {
    for {
        l := <-ev
        fmt.Println(l.source + " -> " + l.target)
        _, err := createNode(session, &l)

        if err != nil {
            fmt.Println("Failed to create node:", err)
        }

    }
}(&session)

r.crawl(url)

fmt.Println("Creation of relationship between nodes.. ")
_, qErr := createNodesRelationship(&session)

if qErr == nil {
    fmt.Println("Nodes updated")
} else {
    fmt.Println("Error while updating nodes:", qErr)
}
}

```

Используя три функцию объявленные в `neo4j.go` наша программа создаст подключение к neo4j, подпишется на `newLink` событие для вставки нод и наконец обновит связи нод. Я использовал `defer` ключево слово, чтобы сослаться на исполнение функции до тех пор пока

не завершится `main` функция. Давайте запустим в последний раз:

```
go run main.go retriever.go neo4j.go http://www.sfeir.com
```

Чтобы проверить результат в Neo4j вы можете запустить следующий запрос в вашем Neo4j браузере.

```
MATCH (n:WebLink) RETURN count(n) AS count
```

Или этот запрос отобразит все ноды:

```
MATCH (n:WebLink) RETURN n
```

## Выводы

В этом посте, мы изучили множество свойств языка Golang включая множественные присвоения переменным, реализацию интерфейсов и каналов и горутин. Так же мы использовали стандартную библиотеку. Спасибо за чтение.

---

Revision #8

Created 2021-12-17 09:39:30 UTC by gasick

Updated 2023-04-16 19:30:03 UTC by gasick