网络爬虫

Colly网络爬虫框架

Colly是用Go实现的网络爬虫框架。Colly快速优雅，在单核上每秒可以发起1K以上请求；以回调函数的形式提供了一组接口，可以实现任意类型的爬虫。它的主要特点是轻量、快速，设计非常优雅，并且分布式的支持也非常简单，易于扩展。

github地址: github.com/gocolly/colly

colly官网地址： http://go-colly.org/

Colly 特性

- 清晰的API
- 快速（单个内核上的请求数大于1k）
- 管理每个域的请求延迟和最大并发数
- 自动cookie 和会话处理
- 同步/异步/并行抓取
- 高速缓存
- 自动处理非Unicode的编码
- 支持Robots.txt
- 定制Agent信息
- 定制抓取频次

特性如此多，引无数程序员竞折腰。下面开始我们的Colly之旅：

Colly使用

1.首先，下载安装第三方包：

```
go get -u github.com/gocolly/colly/…
```

2.接下来在代码中导入包：

```
import "github.com/gocolly/colly"
```

准备工作已经完成，接下来就看看Colly的使用方法和主要的用途。

colly的主体是Collector对象，管理网络通信和负责在作业运行时执行附加的回调函数。使用colly需要先初始化Collector：

```
c := colly.NewCollector()
```

3.调用`colly.NewCollector()`创建一个类型为`*colly.Collector`的爬虫对象。

由于每个网页都有很多指向其他网页的链接。如果不加限制的话，运行可能永远不会停止。所以也可以通过传入一个选项`colly.AllowedDomains("www.baidu.com")`限制只爬取域名为`www.baidu.com`的网页。

我们看看NewCollector，它也是变参函数，参数类型为函数类型func(*Collector)，主要是用来初始化一个&Collector{}对象。

而在Colly中有好些函数都返回这个函数类型func(*Collector)，如UserAgent(us string)用来设置UA。所以，这里其实是一种初始化对象，设置对象属性的一种模式。相比使用方法（set方法）这种传统方式来初始设置对象属性，采用回调函数的形式在Go语言中更灵活更方便：

```
NewCollector(options ...func(*Collector)) *Collector
UserAgent(ua string) func(*Collector)
```

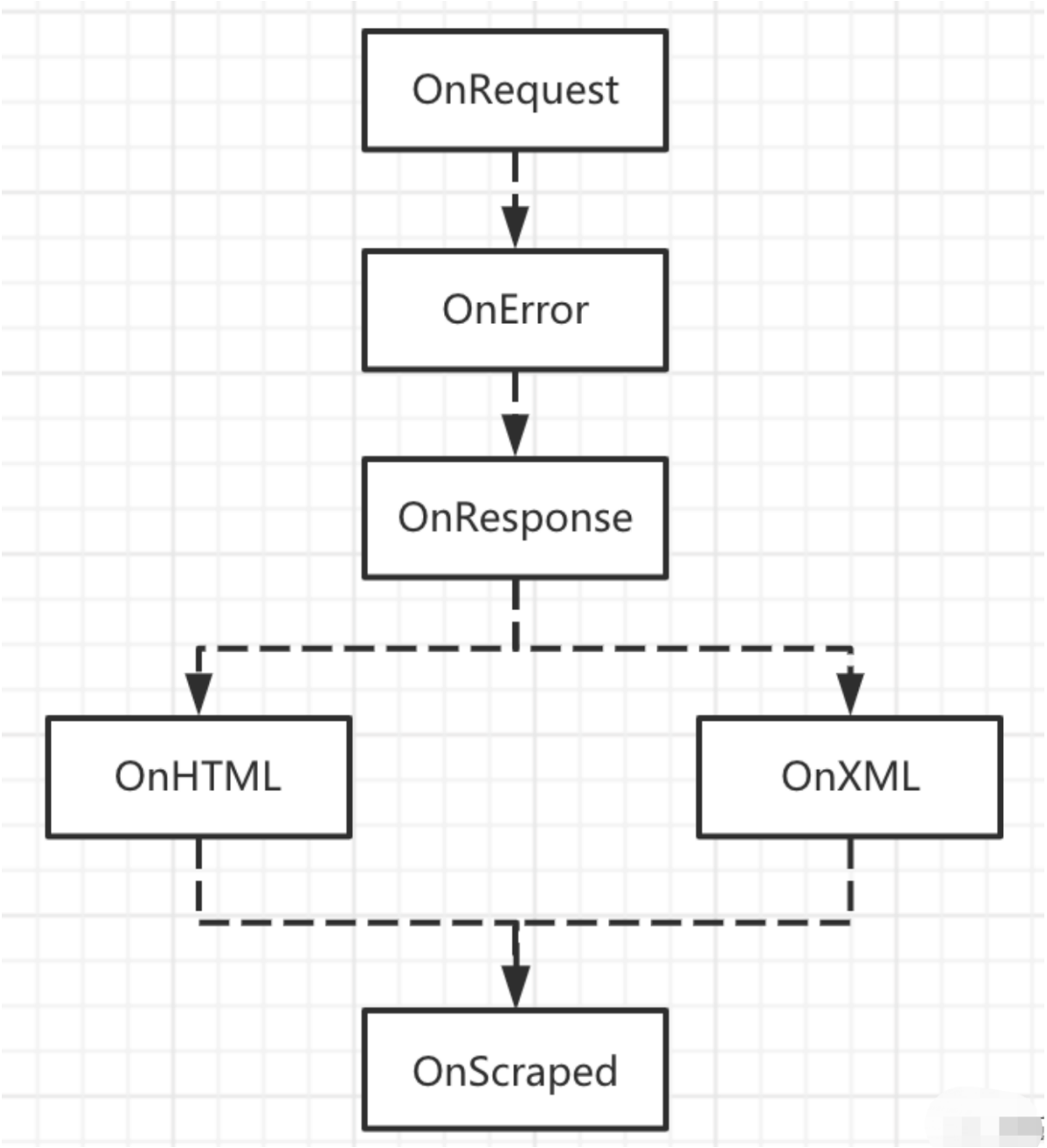
4.事件监听，通过 callback 执行事件处理。

一旦得到一个colly对象，可以向colly附加各种不同类型的回调函数（回调函数在Colly中广泛使用），来控制收集作业或获取信息，回调函数的调用顺序如下：

- OnRequest 请求执行之前调用
- OnResponse 响应返回之后调用
- OnHTML 监听执行 selector，在`OnResponse`之后调用。
- OnXML 监听执行 selector
- OnHTMLDetach，取消监听，参数为 selector 字符串
- OnXMLDetach，取消监听，参数为 selector 字符串
- OnScraped，完成抓取后执行，完成所有工作后执行，在OnXML/OnHTML回调完成后调用。不过官网写的是`Called after OnXML callbacks`，实际上对于OnHTML也有效，大家可以注意一下。
- OnError，错误回调

最后c.Visit() 正式启动网页访问。

执行顺序如下：



下面我们看一个例子：

```
package main

import (
    "fmt"

    "github.com/gocolly/colly/v2"
)

func main() {
    // NewCollector(options ...func(*Collector)) *Collector
    // 声明初始化NewCollector对象时可以指定Agent，连接递归深度，URL过滤以及domain限制等
    c := colly.NewCollector(
        //colly.AllowedDomains("news.baidu.com"),
        colly.UserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45 Safari/537.36"))

    // 发出请求时附的回调
    c.OnRequest(func(r *colly.Request) {
```

```
        // Request头部设定
        r.Headers.Set("Host", "baidu.com")
        r.Headers.Set("Connection", "keep-alive")
        r.Headers.Set("Accept", "*/*")
        r.Headers.Set("Origin", "")
        r.Headers.Set("Referer", "http://www.baidu.com")
        r.Headers.Set("Accept-Encoding", "gzip, deflate")
        r.Headers.Set("Accept-Language", "zh-CN, zh;q=0.9")

        fmt.Println("Visiting", r.URL)
    })

    // 对响应的HTML元素处理
    c.OnHTML("title", func(e *colly.HTMLElement) {
        //e.Request.Visit(e.Attr("href"))
        fmt.Println("title:", e.Text)
    })

    c.OnHTML("body", func(e *colly.HTMLElement) {
        // <div class="hotnews" alog-group="focustop-hotnews"> 下所有的a解析
        e.ForEach(".hotnews a", func(i int, el *colly.HTMLElement) {
            band := el.Attr("href")
            title := el.Text
            fmt.Printf("新闻 %d : %s - %s\n", i, title, band)
            // e.Request.Visit(band)
        })
    })

    // 发现并访问下一个连接
    //c.OnHTML(`.next a[href]`, func(e *colly.HTMLElement) {
    //    e.Request.Visit(e.Attr("href"))
    //})

    // extract status code
    c.OnResponse(func(r *colly.Response) {
        fmt.Println("response received", r.StatusCode)
        // 设置context
        // fmt.Println(r.Ctx.Get("url"))
    })

    // 对visit的线程数做限制，visit可以同时运行多个
    c.Limit(&colly.LimitRule{
        Parallelism: 2,
        //Delay:      5 * time.Second,
    })

    c.Visit("http://news.baidu.com")
}
```

上面代码在开始处对Colly做了简单的初始化，增加UserAgent用户代理和域名限制，其他的设置可根据实际情况来设置，Url过滤，抓取深度等等都可以在此设置，也可以后运行时在具体设置。

部分配置说明

- `AllowedDomains`: 设置收集器使用的域白名单，设置后不在白名单内链接，报错: `Forbidden domain`。
- `AllowURLRevisit`: 设置收集器允许对同一 URL 进行多次下载。
- `Async`: 设置收集器为异步请求，需跟`Wait()`配合使用。
- `Debugger`: 开启Debug,开启后会打印请求日志。
- `MaxDepth`: 设置爬取页面的深度。
- `UserAgent`: 设置收集器使用的用户代理。
- `MaxBodySize`: 以字节为单位设置检索到的响应正文的限制。
- `IgnoreRobotsTxt`: 忽略目标机器中的`robots.txt`声明。

设置UserAgent

```
//设置UserAgent的两种方式:
//方式一 :
c2 := colly.NewCollector()
c2.UserAgent = "xy"
c2.AllowURLRevisit = true*/ AllowURLRevisit 允许重复访问

//方式二 :
c2 := colly.NewCollector(
    colly.UserAgent("xy"),
    colly.AllowURLRevisit(),
)
```

collector 的配置可以在爬虫执行到任何阶段改变。一个经典的例子，通过随机改变 user-agent，可以帮助我们实现简单的反爬。

```
const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

func RandomString() string {
    b := make([]byte, rand.Intn(10)+10)
    for i := range b {
        b[i] = letterBytes[rand.Intn(len(letterBytes))]
    }
    return string(b)
}

c := colly.NewCollector()

c.OnRequest(func(r *colly.Request) {
    r.Headers.Set("User-Agent", RandomString())
})
```

设置Cookie

```
//设置cookie的两种方式
//方式一：通过手动网页添加cookies
c.OnRequest(func(r *colly.Request) {
    r.Headers.Add("cookie","_ga=GA1.2.1611472128.1650815524;
_gid=GA1.2.2080811677.1652022429; __atuvc=2|17,0|18,5|19")
})
// 方式二 :通过url 添加cookies
siteCookie := c.Cookies("url")
c.SetCookies("",siteCookie)
```

HTTP配置

Colly使用Golang的默认http客户端作为网络层。可以通过更改默认的HTTP roundtripper来调整HTTP选项。都是些常用的配置，比如代理、各种超时时间等。

```
c := colly.NewCollector()
c.WithTransport(&http.Transport{
    Proxy: http.ProxyFromEnvironment,
    DialContext: (&net.Dialer{
        Timeout:   30 * time.Second,          // 超时时间
        KeepAlive: 30 * time.Second,          // keepAlive 超时时间
        DualStack: true,
    }).DialContext,
    MaxIdleConns:          100,               // 最大空闲连接数
    IdleConnTimeout:       90 * time.Second,  // 空闲连接超时
    TLSHandshakeTimeout:   10 * time.Second,  // TLS 握手超时
    ExpectContinueTimeout: 1 * time.Second,
}
```

collector 默认已经为我们选择了较优的配置，其实它们也可以通过环境变量改变。这样，我们就可以不用为了改变配置，每次都得重新编译了。环境变量配置是在 collector 初始化时生效，正式启动后，配置是可以被覆盖的。

```
ALLOWED_DOMAINS (字符串切片)，允许的域名，比如 []string{"segmentfault.com",
"zhihu.com"}
CACHE_DIR (string) 缓存目录
DETECT_CHARSET (y/n) 是否检测响应编码
DISABLE_COOKIES (y/n) 禁止 cookies
DISALLOWED_DOMAINS (字符串切片)，禁止的域名，同 ALLOWED_DOMAINS 类型
IGNORE_ROBOTSTXT (y/n) 是否忽略 ROBOTS 协议
MAX_BODY_SIZE (int) 响应最大
MAX_DEPTH (int - 0 means infinite) 访问深度
PARSE_HTTP_ERROR_RESPONSE (y/n) 解析 HTTP 响应错误
USER_AGENT (string)
```

该例只是简单说明了Colly在爬虫抓取，调度管理方面的优势，对此如有兴趣可更深入了解。大家在深入学习Colly时，可自行选择更合适的URL。

程序运行后，开始根据news.baidu.com抓取页面结果，通过OnHTML回调函数分析首页中的热点新闻标题及链接，并可不断地抓取更深层次的新链接进行访问，每个链接的访问结果我们可以通过OnHTML来进行分析，也可通过OnResponse来进行处理。

我们来看看OnHTML这个方法的定义：

```
func (c *Collector) OnHTML(goquerySelector string, f HTMLCallback)
```

直接在参数中标明了 goquerySelector ，上例中我们有简单尝试。这和我们下面要介绍的goquery HTML解析框架有一定联系，我们也可以使用goquery，通过goquery 来更轻松分析HTML代码。

colly语法模板

```
// 简单使用模板示例
func collyUseTemplate() {
 // 创建采集器对象
 collector := colly.NewCollector()
 // 发起请求之前调用
 collector.OnRequest(func(request *colly.Request) {
  fmt.Println("发起请求之前调用...")
 })
 // 请求期间发生错误,则调用
 collector.OnError(func(response *colly.Response, err error) {
  fmt.Println("请求期间发生错误,则调用:",err)
 })
 // 收到响应后调用
 collector.OnResponse(func(response *colly.Response) {
  fmt.Println("收到响应后调用:",response.Request.URL)
 })
 //OnResponse如果收到的内容是HTML ,则在之后调用
 collector.OnHTML("#position_shares table", func(element *colly.HTMLElement) {
  // todo 解析html内容
 })
 // url：请求具体的地址
 err := collector.Visit("请求具体的地址")
 if err != nil {
  fmt.Println("具体错误:",err)
 }
}
```

常用解析函数

`colly` 爬取到页面之后，又该怎么解析 `Html` 内容呢？实际上使用goquery包解析这个页面,而 `colly.HTMLElement` 其实就是对 `goquery.Selection` 的简单封装：

```
// colly.HTMLElement结构体
type HTMLElement struct {
 Name       string
 Text       string
 attributes []html.Attribute
 Request *Request
 Response *Response
 // 对goquery封装
 DOM *goquery.Selection
 Index int
}
```

Attr

返回当前元素的属性

```
func (h *HTMLElement) Attr(k string) string
```

```
// 返回当前元素的属性
func TestUseAttr(t *testing.T) {
 collector := colly.NewCollector()
  // 定位到div[class='nav-logo'] > a标签元素
 collector.OnHTML("div[class='nav-logo'] > a", func(element *colly.HTMLElement)
{
  fmt.Printf("href:%v\n",element.Attr("href"))
 })
 _ = collector.Visit("https://book.douban.com/tag/小说")
}
```

ChildAttr&ChildAttrs

```
// 返回`goquerySelector`选择的第一个子元素的`attrName`属性；
func (h *HTMLElement) ChildAttr(goquerySelector, attrName string) string
// 返回`goquerySelector`选择的所有子元素的`attrName`属性，以`[]string`返回；
func (h *HTMLElement) ChildAttrs(goquerySelector, attrName string) []string
```

```
func TestChildAttrMethod(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnError(func(response *colly.Response, err error) {
  fmt.Println("OnError",err)
 })
 // 解析Html
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // 获取第一个子元素(div)的class属性
  fmt.Printf("ChildAttr:%v\n",element.ChildAttr("div","class"))
  // 获取所有子元素(div)的class属性
  fmt.Printf("ChildAttrs:%v\n",element.ChildAttrs("div","class"))
 })
 err := collector.Visit("a.html")
 if err != nil {
  fmt.Println("err",err)
 }
}
```

ChildText & ChildTexts

```
// 拼接goquerySelector选择的子元素的文本内容并返回
func (h *HTMLElement) ChildText(goquerySelector string) string
// 返回goquerySelector选择的子元素的文本内容组成的切片，以[]string返回。
func (h *HTMLElement) ChildTexts(goquerySelector string) []string
```

```
<html>
<head>
  <title>测试</title>
```

```
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
 <div class="div1">
  <div class="sub1">内容1</div>
 </div>
 <div class="div2">内容2</div>
 <div class="div3">内容3</div>
</body>
</html>

// 测试使用ChildText和ChildTexts
func TestChildTextMethod(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnError(func(response *colly.Response, err error) {
  fmt.Println("OnError",err)
 })
 //
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // 获取第一个子元素(div)的class属性
  fmt.Printf("ChildText:%v\n",element.ChildText("div"))
  // 获取所有子元素(div)的class属性
  fmt.Printf("ChildTexts:%v\n",element.ChildTexts("div"))
 })
 err := collector.Visit("a.html")
 if err != nil {
  fmt.Println("err",err)
 }
}
```

ForEach

```
//对每个`goquerySelector`选择的子元素执行回调`callback`
func (h *HTMLElement) ForEach(goquerySelector string, callback func(int,
*HTMLElement))
```

```


<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
 <ul class="demo">
  <li>
   <span class="name">张三</span>
   <span class="age">18</span>
   <span class="home">北京</span>
  </li>
  <li>
   <span class="name">李四</span>
   <span class="age">22</span>
   <span class="home">南京</span>
  </li>
```

```
  <li>
   <span class="name">王五</span>
   <span class="age">29</span>
   <span class="home">天津</span>
  </li>
 </ul>
</body>
</html>


func TestForeach(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("ul[class='demo']", func(element *colly.HTMLElement) {
  element.ForEach("li", func(_ int, el *colly.HTMLElement) {
   name := el.ChildText("span[class='name']")
   age := el.ChildText("span[class='age']")
   home := el.ChildText("span[class='home']")
   fmt.Printf("姓名: %s  年龄:%s 住址: %s \n",name,age,home)
  })
 })
 _ = collector.Visit("a.html")
}
```

Unmarshal

```
// 通过给结构体字段指定 goquerySelector 格式的 tag，可以将HTMLElement对象Unmarshal 到一个结构体实例中
func (h *HTMLElement) Unmarshal(v interface{}) error
```

```
<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
 <div class="book">
  <span class="title"><a href="https://liuqh.icu">红楼梦</a></span>
  <span class="autor">曹雪芹 </span>
  <ul class="category">
   <li>四大名著</li>
   <li>文学著作</li>
   <li>古典长篇章回小说</li>
   <li>四大小说名著之首</li>
  </ul>
  <span class="price">59.70元</span>
 </div>
</body>
</html>

// 定义结构体
type Book struct {
 Name string `selector:"span.title"`
 Link string `selector:"span > a" attr:"href"`
 Author string `selector:"span.autor"`
```

```
 Reviews []string `selector:"ul.category > li"`
 Price string `selector:"span.price"`
}

func TestUnmarshal(t *testing.T) {
 // 声明结构体
 var book Book
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  err := element.Unmarshal(&book)
  if err != nil {
   fmt.Println("解析失败:",err)
  }
  fmt.Printf("结果:%+v\n",book)
 })
 _ = collector.Visit("a.html")
}
```

常用选择器

html内容

```
<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
  <!-- 演示ID选择器 -->
  <div id="title">标题ABC</div>
  <!-- 演示class选择器 -->
  <div class="desc">这是一段描述</div>
  <!-- 演示相邻选择器 -->
  <span>好好学习! </span>
  <!-- 演示父子选择器 -->
  <div class="parent">
    <!-- 演示兄弟选择器 -->
    <p class="childA">老大</p>
 ```<p class="childB">老二</p>
  </div>
  <!-- 演示同时筛选多个选择器 -->
  武松上山
  打老虎
</body>
</html>
```

### 使用示例

```
// 常用选择器使用
func TestSelector(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // ID属性选择器,使用#
```
```

```
  fmt.Printf("ID选择器使用: %v \n",element.ChildText("#title"))
  // Class属性选择器,使用
  fmt.Printf("class选择器使用1: %v \n",element.ChildText("div[class='desc']"))
  fmt.Printf("class选择器使用2: %v \n",element.ChildText(".desc"))
  // 相邻选择器 prev +  next: 提取 <span>好好学习! </span>
  fmt.Printf("相邻选择器: %v \n",element.ChildText("div[class='desc'] + span"))
  // 父子选择器:  parent > child,提取:<div class="parent">下所有子元素
  fmt.Printf("父子选择器: %v \n",element.ChildText("div[class='parent'] > p"))
  // 兄弟选择器 prev ~ next , 提取:<p class="childB">老二</p>
  fmt.Printf("兄弟选择器: %v \n",element.ChildText("p[class='childA'] ~ p"))
  // 同时选中多个,用,
  fmt.Printf("同时选中多个1: %v
\n",element.ChildText("span[class='context1'],span[class='context2']"))
  fmt.Printf("同时选中多个2: %v \n",element.ChildText(".context1,.context2"))
 })
 _ = collector.Visit("a.html")
}
```

过滤器

第一个子元素(`first-child & first-of-type`)

过滤器	说明
`:first-child`	筛选出父元素的第一个子元素,不分区子元素类型
`:first-of-type`	筛选出父元素的第一个指定类型子元素

```
<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
  <div class="parent">
    <!--因为这里的p不是第一个子元素，不会被first-child筛选到-->
   <span>第一个不是p标签</span>
    <p>老大</p>
    <p>老二</p>
  </div>
  <div class="name">
    <p>张三</p>
    <p>小米</p>
  </div>
</body>
</html>

// 常用过滤器使用first-child & first-of-type
func TestFilterFirstChild(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // 只会筛选父元素下第一个子元素是<p>..</p>
  fmt.Printf("first-child: %v \n",element.ChildText("p:first-child"))
  // 会筛选父元素下第一个子元素类型是<p>..</p>
```

```
  fmt.Printf("first-of-type: %v \n",element.ChildText("p:first-of-type"))
 })
 _ = collector.Visit("a.html")
}
```

最后一个子元素(`last-child & last-of-type`)

过滤器	说明
`:last-child`	筛选出父元素的最后一个子元素,不分区子元素类型
`:last-of-type`	筛选出父元素的最后一个指定类型子元素

使用方法和上面`筛选第一个子元素`一样，不在啰嗦。

第x个子元素(`nth-child & nth-of-type`)

过滤器	说明
`nth-child(n)`	筛选出的元素是其父元素的第`n`个元素，n以`1`开始。 所以`:nth-child(1) = :first-child`
`nth-of-type(n)`	和`nth-child`一样，只不过它筛选的是同类型的第n个元素， 所以`:nth-of-type(1) = :first-of-type`
`nth-last-child(n)`	和`nth-child(n)`一样，顺序是倒序
`nth-last-of-type(n)`	和`nth-of-type(n)`一样，顺序是倒序

```
// 过滤器第x个元素
func TestFilterNth(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  //<div class="parent">下的第一个子元素
  nthChild := element.ChildText("div[class='parent'] > :nth-child(1)")
  fmt.Printf("nth-child(1): %v \n",nthChild)

  //<div class="parent">下的第一个p子元素
  nthOfType := element.ChildText("div[class='parent'] > p:nth-of-type(1)")
  fmt.Printf("nth-of-type(1): %v \n",nthOfType)

  // div class="parent">下的最后一个子元素
  nthLastChild := element.ChildText("div[class='parent'] > :nth-last-child(1)")
  fmt.Printf("nth-last-child(1): %v \n",nthLastChild)

  //<div class="parent">下的最后一个p子元素
  nthLastOfType := element.ChildText("div[class='parent'] > p:nth-last-of-
type(1)")
  fmt.Printf("nth-last-of-type(1): %v \n",nthLastOfType)

 })
 _ = collector.Visit("a.html")
}
```

仅有一个子元素(`only-child & only-of-type`)

过滤器	说明
`:only-child`	筛选其父元素下只有个子元素
`:on-of-type`	筛选其父元素下只有个指定类型的子元素

html内容

```
<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
  <div class="parent">
   <span>我是span标签</span>
  </div>
  <div class="name">
    <p>我是p标签</p>
  </div>
</body>
</html>
```

```
// 过滤器只有一个元素
func TestFilterOnly(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // 匹配其子元素：有且只有一个标签的
  onlyChild := element.ChildTexts("div > :only-child")
  fmt.Printf("onlyChild: %v \n",onlyChild)
  // 匹配其子元素：有且只有一个 p 标签的
  nthOfType := element.ChildTexts("div > p:only-of-type")
  fmt.Printf("nth-of-type(1): %v \n",nthOfType)

 })
 _ = collector.Visit("a.html")
}
```

内容匹配(`contains`)

html内容

```
<html>
<head>
  <title>测试</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
  <a href="https://www.baidu.com">百度</a>
  <a href="https://cn.bing.com">必应</a>
</body>
</html>
```

使用示例

```
func TestFilterContext(t *testing.T) {
 collector := colly.NewCollector()
 collector.OnHTML("body", func(element *colly.HTMLElement) {
  // 内容匹配
  attr1 := element.ChildAttr("a:contains(百度)", "href")
  attr2 := element.ChildAttr("a:contains(必应)", "href")
  fmt.Printf("百度: %v \n",attr1)
  fmt.Printf("必应: %v \n",attr2)
 })
 _ = collector.Visit("a.html")
}
```

goquery HTML解析库

简介

Colly框架可以快速发起请求，接收服务器响应。但如果我们需要分析返回的HTML代码，这时候仅仅使用Colly就有点吃力。而goquery库是一个使用Go语言写成的HTML解析库，功能更加强大。

goquery将jQuery的语法和特性引入进来，所以可以更灵活地选择采集内容的数据项，就像jQuery那样的方式来操作DOM文档，使用起来非常的简便。

goquery是基于 Go net/htm包和css选择器库 cascadia。由于net/htm解析器返回的是DOM节点,而不是完整的DOM树,因此, jQuery的状态操作函数没有实现(像height(), css(), detach)

> 注意：goquery只支持utf-8编码，其他编码需要转换。

goquery可以用来实现如下功能：

- 在HTML文档中查找、筛选和遍历元素
- 获取元素的属性、文本内容、HTML内容等信息
- 对元素进行添加、修改、删除等操作
- 在HTML文档中执行CSS选择器操作
- 支持链式调用，可以方便地进行多个操作组合

总的来说，goquery是一款非常实用的HTML解析工具，它可以大大简化开发者在Go语言中进行HTML解析的工作。

goquery 暴露了两个结构体：`Document` 和 `Selection`，`Document` 表示一个 HTML 文档，`Selection` 用于像 jQuery 一样操作，支持链式调用。goquery 需要指定一个 HTML 文档才能继续后续的操作。

goquery主要的结构：

```
type Document struct {
    *Selection
    Url      *url.URL
    rootNode *html.Node
}
```

Document 嵌入了Selection 类型，因此，Document 可以直接使用 Selection 类型的方法。我们可以通过下面四种方式来初始化得到*Document对象。

```
func NewDocumentFromNode(root *html.Node) *Document  // 传入 *html.Node 对象，也就是根节点。

func NewDocument(url string) (*Document, error)  // 传入 URL，内部用 http.Get 获取网页。

func NewDocumentFromReader(r io.Reader) (*Document, error) // 传入 io.Reader，内部从 reader 中读取内容并解析。返回了一个*Document和error。Document代表一个将要被操作的HTML文档。

func NewDocumentFromResponse(res *http.Response) (*Document, error) // 传入 HTTP 响应，内部拿到res.Body(实现了 io.Reader) 后的处理方式类似 NewDocumentFromReader
```

通过源码可以知道 Document 继承了 Selection，除此之外最重要的是rootNode，它是 HTML 的根节点，Url这个字段作用不大，在使用NewDocument和NewDocumentFromResponse时会对该字段赋值。

Selection 是重要的一个结构体，解析中最重要，最核心的方法方法都由它提供。

```
type Selection struct {
    Nodes    []*html.Node
    document *Document
    prevSel  *Selection
}
```

goquery使用

下面我们开始了解下怎么使用goquery：

首先，要确定已经下载安装这个第三方包：

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

接下来在代码中导入包：

```
import "github.com/PuerkitoBio/goquery"
```

goquery的主要用法是选择器，需要借鉴jQuery的特性，多加练习就能很快掌握。限于篇幅，这里只能简单介绍了goquery的大概情况。

goquery可以直接发送url请求，获得响应后得到HTML代码。但goquery主要擅长于HTML代码分析，而Colly在爬虫抓取管理调度上有优势，所以下面以Colly作为爬虫框架，goquery作为HTML分析器，看看怎么抓取并分析页面内容：

```
package main

import (
    "bytes"
    "fmt"
    "log"
    "net/url"
    "time"

    "github.com/PuerkitoBio/goquery"
    "github.com/gocolly/colly/v2"
)

func main() {
    urlstr := "https://news.baidu.com"
    u, err := url.Parse(urlstr)
    if err != nil {
        log.Fatal(err)
    }
    c := colly.NewCollector()
    // 超时设定
    c.SetRequestTimeout(100 * time.Second)
    // 指定Agent信息
    c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/63.0.3239.108 Safari/537.36"
    c.OnRequest(func(r *colly.Request) {
        // Request头部设定
        r.Headers.Set("Host", u.Host)
        r.Headers.Set("Connection", "keep-alive")
        r.Headers.Set("Accept", "*/*")
        r.Headers.Set("Origin", u.Host)
        r.Headers.Set("Referer", urlstr)
        r.Headers.Set("Accept-Encoding", "gzip, deflate")
        r.Headers.Set("Accept-Language", "zh-CN, zh;q=0.9")
    })

    c.OnHTML("title", func(e *colly.HTMLElement) {
        fmt.Println("title:", e.Text)
    })

    c.OnResponse(func(resp *colly.Response) {
        fmt.Println("response received", resp.StatusCode)

        // goquery直接读取resp.Body的内容
        htmlDoc, err :=
goquery.NewDocumentFromReader(bytes.NewReader(resp.Body))

        // 读取url再传给goquery，访问url读取内容，此处不建议使用
        // htmlDoc, err := goquery.NewDocument(resp.Request.URL.String())

        if err != nil {
```

```
            log.Fatal(err)
        }

        // 找到抓取项 <div class="hotnews" alog-group="focustop-hotnews"> 下所有的a
解析
        htmlDoc.Find(".hotnews a").Each(func(i int, s *goquery.Selection) {
            band, _ := s.Attr("href")
            title := s.Text()
            fmt.Printf("热点新闻 %d: %s - %s\n", i, title, band)
            c.Visit(band)
        })

    })

    c.OnError(func(resp *colly.Response, errHttp error) {
        err = errHttp
    })

    err = c.Visit(urlstr)
}
```

上面代码中，goquery先通过 goquery.NewDocumentFromReader生成文档对象htmlDoc。有了htmlDoc就可以使用选择器，而选择器的目的主要是定位：htmlDoc.Find(“.hotnews a”).Each(func(i int, s *goquery.Selection)，找到文档中的

。

有关选择器Find()方法的使用语法，是不是有些熟悉的感觉，没错就是jQuery的样子。

选择器

在goquery中，常用大概有以下选择器：

选择器类型	说明
HTML Element	元素的选择器Find(“a”)
Element ID 选择器	Find(element#id)
Class选择器	Find(“.class”)
属性选择器	

选择器	说明
Find(“div[lang]“)	筛选含有lang属性的div元素
Find(“div[lang=zh]“)	筛选lang属性为zh的div元素
Find(“div[lang!=zh]“)	筛选lang属性不等于zh的div元素
Find(“div[lang|=zh]“)	筛选lang属性为zh或者zh-开头的div元素
Find(“div[lang*=zh]“)	筛选lang属性包含zh这个字符串的div元素
Find(“div[lang~=zh]“)	筛选lang属性包含zh这个单词的div元素，单词以空格分开的

选择器	说明
Find(“div[lang$=zh]“)	筛选lang属性以zh结尾的div元素，区分大小写
Find(“div[lang^=zh]“)	筛选lang属性以zh开头的div元素，区分大小写

parent>child选择器
如果我们想筛选出某个元素下符合条件的子元素，我们就可以使用子元素筛选器，它的语法为Find(“parent>child”),表示筛选parent这个父元素下，符合child这个条件的最直接（一级）的子元素。

prev+next相邻选择器
假设我们要筛选的元素没有规律，但是该元素的上一个元素有规律，我们就可以使用这种下一个相邻选择器来进行选择。

prev~next选择器
有相邻就有兄弟，兄弟选择器就不一定要求相邻了，只要他们共有一个父元素就可以。

示例：对 微博热搜 进行一个简单的解析，打印当日的热搜排名标题以及热度。

```
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/PuerkitoBio/goquery"
)

type Data struct {
    number string
    title  string
    heat   string
}

func main() {
    // 爬取微博热搜网页
    res, err := http.Get("https://s.weibo.com/top/summary")
    if err != nil {
        log.Fatal(err)
    }

    defer res.Body.Close()
    if res.StatusCode != 200 {
        log.Fatalf("status code error: %d %s", res.StatusCode, res.Status)
    }

    //将html生成goquery的Document
    dom, err := goquery.NewDocumentFromReader(res.Body)
    if err != nil {
        log.Fatalln(err)
    }

    var data []Data
    // 筛选class为td-01的元素
    dom.Find(".td-01").Each(func(i int, selection *goquery.Selection) {
        data = append(data, Data{number: selection.Text()})
    })
```

```
    // 筛选class为td-02的元素下的a元素
    dom.Find(".td-02>a").Each(func(i int, selection *goquery.Selection) {
        data[i].title = selection.Text()
    })
    // 筛选class为td-02的元素下的span元素
    dom.Find(".td-02>span").Each(func(i int, selection *goquery.Selection) {
        data[i].heat = selection.Text()
    })

    fmt.Println(data)
}
```

过滤器示例

基于HTML Element 元素的选择器

使用Element名称作为选择器，如 dom.Find("div")

```
dom.Find("div").Each(func (i int, selection *goquery.Selection) {
    fmt.Println(selection.Text())
})
```

ID选择器

以#加id值作为选择器,如 dom.Find("#id")

```
dom.Find("#id").Each(func (i int, selection *goquery.Selection) {
    fmt.Println(selection.Text())
})
```

Class选择器

以.加class值为选择器,如 dom.Find(".class")

```
dom.Find(".class").Each(func (i int, selection *goquery.Selection) {
    fmt.Println(selection.Text())
})
```

由上面的示例可以看出，goquery的选择器与jQuery的选择器用法无异，在这里就不继续赘述了，同学们可以自行探索。

常用节点属性值

Html() 获取该节点的html

```
dom.Find("table").Each(func (i int, selection *goquery.Selection) {
    fmt.Println(selection.Html())
})
```

Text() 获取该节点的文本值

```
dom.Find(".td-02>a").Each(func (i int, selection *goquery.Selection) {
    fmt.Println(selection.Text())
})
```

Attr() 返回节点的属性值以及该属性是否存在的布尔值

```
dom.Find("#execution").Each(func (i int, selection *goquery.Selection) {
    value[i], ok = selection.Attr("value")
})
```

AttrOr 和上一个方法类似，区别在于如果属性不存在，则返回给定的默认值

Length() 返回该Selection的元素个数

```
dom.Find("td").Length()
```

Remove() 删除节点

AfterHtml() 插入 HTML

迭代

goquery 提供了三个用于迭代的方法，都接受一个匿名函数作为参数：

Each

```
Each(f func(int, *Selection)) *Selection
```

其中函数 `f` 的第一个参数是当前的下标，第二个参数是当前的节点

EachWithBreak

```
EachWithBreak(f func(int, *Selection) bool) *Selection
```

和 `Each` 类似，增加了中途跳出循环的能力，当 `f` 返回 `false` 时结束迭代

Map

```
Map(f func(int, *Selection) string) (result []string)
```

`f` 的参数与上面一样，返回一个 string 类型，最终返回 []string.

总结

简单来说就是

1、创建文档

```
d,e := goquery.NewDocumentFromReader(reader io.Reader)

d,e := goquery.NewDocument(url string)
```

2、查找内容

```
ele.Find("#title") //根据id查找

ele.Find(".title") //根据class查找

ele.Find("h2").Find("a") //链式调用
```

3、获取内容

```
ele.Html()

ele.Text()
```

4、获取属性

```
ele.Attr("href")

ele.AttrOr("href", "")
```

5、遍历

```
ele.Find(".item").Each(func(index int, ele *goquery.Selection){

})
```