27 | 掘地三尺：实战深度与广度优先搜索算法

2022-12-10 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 09:21　大小 8.57M

你好，我是郑建勋。

上节课，我们看到了如何在 Go 中创建高并发模型，这节课让我们回到项目中来，为爬虫项目构建高并发的模型。

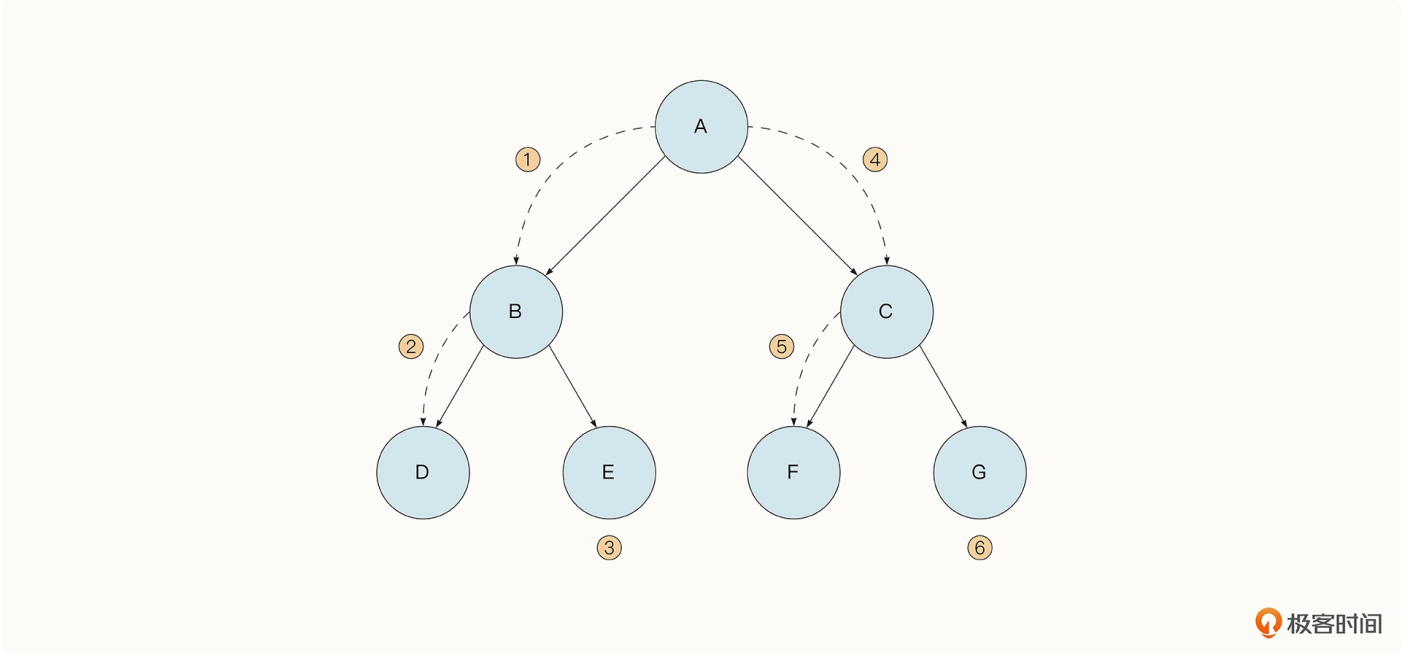
要想构建高并发模型，我们首先要做的就是将一个大任务拆解为许多可以并行的小任务。比方说在爬取一个网站时，这个网站中通常会有一连串的 URL 需要我们继续爬取。显然，如果我们把所有任务都放入到同一个协程中去处理，效率将非常低下。那么我们应该选择什么方式来拆分出可以并行的任务，又怎么保证我们不会遗漏任何信息呢？

要解决这些问题，我们需要进行爬虫任务的拆分、并设计任务调度的算法。首先让我们来看一看两种经典的爬虫算法：深度优先搜索算法（Depth-First-Search，DFS）和广度优先搜索算法（Breadth-First Search，BFS），他们也是图论中的经典算法。

深度优先搜索算法

深度优先搜索算法是约翰·霍普克洛夫特和罗伯特·塔扬共同发明的，他们也因此在 1986 年共同获得计算机领域的最高奖：图灵奖。

以下图中的拓扑结构为例，节点 A 标识的是爬取的初始网站，在网站 A 中，有 B、C 两个链接需要爬取，以此类推。深度优先搜索的查找顺序是： 从 A 查找到 B，紧接着查找 B 下方的 D，然后是 E。查找完之后，再是 C、F，最后是 G。可以看出，深度优先搜索的特点就是“顺藤摸瓜”，一路向下，先找最“深”的节点。



深度优先搜索在实践中有许多应用，例如查找图的最长路径、解决八皇后之类的迷宫问题等。**而在实现形式上，深度优先搜索可以采取递归与非递归两种形式。**其中，递归是一种非常经典的分层思想，但是如果函数调用时不断压栈，可能导致栈内存超出限制，这对于 Go 语言来说会有栈扩容的成本，并且在实践中也不太好调试。而深度优先搜索的非递归形式要更简单一些，我们可以借助堆栈先入后出的特性来实现它，不过需要开辟额外的空间来模拟堆栈。

《The Go Programming Language》这本书里有一个很恰当的案例，我们以它为基础进一步说明一下。

假设我们都是计算机系的大学生，需要选修一些课程。但是要选修有的课程必须要先学习它的前序课程。例如，学习网络首先要学习操作系统的知识，而要学习操作系统的知识必须首先学习数据结构的知识。如果我们现在只知道每门课程的前序课程，不清楚完整的学习路径，我们

要怎么设计这一系列课程学习的顺序，确保我们在学习任意一门课程的时候，都已经学完了它的前序课程呢？

这个案例非常适合使用深度优先搜索算法来处理。下面是这个案例的实现代码：

复制代码

```
// 计算机课程和其前序课程的映射关系
var prereqs = map[string][]string{
  "algorithms": {"data structures"},
  "calculus":   {"linear algebra"},

  "compilers": {
    "data structures",
    "formal languages",
    "computer organization",
  },

  "data structures":       {"discrete math"},
  "databases":             {"data structures"},
  "discrete math":         {"intro to programming"},
  "formal languages":      {"discrete math"},
  "networks":              {"operating systems"},
  "operating systems":     {"data structures", "computer organization"},
  "programming languages": {"data structures", "computer organization"},
}

func main() {
  for i, course := range topoSort(prereqs) {
    fmt.Printf("%d:\t%s\n", i+1, course)
  }
}

func topoSort(m map[string][]string) []string {
  var order []string
  seen := make(map[string]bool)
  var visitAll func(items []string)

  visitAll = func(items []string) {
    for _, item := range items {
      if !seen[item] {
        seen[item] = true
        visitAll(m[item])
        order = append(order, item)
      }
    }
  }

```

```
  var keys []string
  for key := range m {
    keys = append(keys, key)
  }

  sort.Strings(keys)
  visitAll(keys)
  return order
}
```


这里， prereqs 代表计算机课程和它的前序课程的映射关系，核心的处理逻辑则在 visitAll 这个匿名函数中。visitAll 会使用递归计算最前序的课程，并添加到列表的 order 中，这就保证了课程的先后顺序。

广度优先搜索算法

广度优先搜索指的是从根节点开始，逐层遍历树的节点，直到所有节点均被访问为止。我们还是以之前的拓扑结构为例，广度优先搜索会首先查找 A、接着查找 B、C，最后查找 D、E、F、G。Dijkstra 最短路径算法和 Prim 最小生成树算法都采用了和广度优先搜索类似的思想。

实现广度优先搜索最简单的方式是使用队列。这是由于队列具有先入先出的属性。以上面的拓扑结构为例，我们可以构造一个队列，然后先将节点 A 放入到队列中。接着取出 A 来处理，并将与 A 相关联的 B、C 放入队列末尾。接着取出 B，将 D、E 放入队列末尾，接着取出 C，将 F、G 放入队列末尾。以此类推。

广度优先搜索在实践中的应用也很广泛。例如。要计算两个节点之间的最短路径，即时策略游戏中的找寻路径问题都可以使用它。Go 语言垃圾回收在并发标记阶段也是用广度优先搜索查找当前存活的内存的。

下面是一段利用广度优先搜索爬取网站的例子。其中，urls 是一串 URL 列表，exactUrl 抓取每一个网站中要继续爬取的 URL，并放入到队列 urls 的末尾，用于后续的爬取。

复制代码

```
func breadthFirst(urls []string) {
  for len(urls) > 0 {
    items := urls
    urls = nil
    for _, item := range items {
      urls = append(urls, exactUrl(item)...)
    }
```

```
    }
}
```

用广度优先搜索实战爬虫

根据爬取目标的不同，可以灵活地选择广度优先和深度优先算法。但一般广度优先搜索算法会更加简单直观一些。下面我用广度优先搜索来实战爬虫，这一次我们爬取的是豆瓣小组中的数据。

首先，让我们在 collect 中新建一个 request.go 文件，对 request 做一个简单的封装。Request 中包含了一个 URL，表示要访问的网站。这里的 ParseFunc 函数会解析从网站获取到的网站信息，并返回 Requesrts 数组用于进一步获取数据。而 Items 表示获取到的数据。

复制代码

```go
type Request struct {
  Url       string
  ParseFunc func([]byte) ParseResult
}

type ParseResult struct {
  Requesrts []*Request
  Items     []interface{}
}
```

豆瓣小组是一个个的兴趣小组，小组内的组员可以发帖和评论。我们以“深圳租房”这个兴趣小组为例，这个网站里有许多的租房帖子。

假设我们希望找到带阳台的租房信息的帖子，第一步我们首先要将这个页面中所有帖子所在的网址爬取出来。

南山地铁站附近 业主直租两房，5800，可办理小区停...	猴面包树		10-15 21:23
9号线 南油西站 大次卧近服装城 科技生态园招舍友	冰淇淋		10-15 21:22
[新房源出租，碧海湾地铁口附近，富通城二期105平...	Life is like a		
1号线坪洲、西乡双地铁口/永丰社区28平大单间转租 ...	房小妹	1	10-15 21:20
福田口岸2000急转！！	爆米花__		10-15 21:10
南山区1号线大新站 阳台房 2300押一付一 房东直租	芭比小桑		10-15 21:06
（仅租女）南山一号线大新地铁站步行7分钟2200月租...	哔哩吧啦		10-15 21:05
个人转租，丹竹头地铁口	hui子妮		10-15 21:05
民治地铁口两室一厅合租1350+也可整套转租	豆友C791jftm5Y	2	10-15 21:00
近笋岗水贝小区大单间转租	向天再借500元		10-15 20:57
求一位合租室友 赤湾复式四室一厅 仅两户居住 低价...	小白白		10-15 20:56
蛇口转租单间3k4	粲君		10-15 20:51
求宝安区优质商铺资源	Bryan	1	10-15 20:39
灵芝洪浪北个人转租免中介电梯房公寓一房一厅	沐沐.		10-15 20:38
版田北 地铁口500米 1330	L I		10-15 20:35
南山地铁口H出口次卧出租	Helen	1	10-15 20:34
2300上水径地铁站好房转租	鲍慧东		10-15 20:30
西丽7号线求租	Context_ming	4	10-15 20:29
龙岗坂田1300大单间转租	Logimonster	3	10-15 20:26
南山/福田l求租 离车公庙10km以内 一房一厅单间...	工资还没发呢	1	10-15 20:17
龙华区近深圳北站单间 带阳台洗手间可做饭 910元一...	cab	1	10-15 20:16

<前页 1 2 3 4 5 6 7 8 9 ... 后页>

不过我们没法一次性将所有的帖子查找出来，因为每一页只会为我们展示 25 个帖子，要看后面的内容需要点击下方具体的页数，进入到第 2 页、第 3 页。

不过这难不倒我们，稍作分析就能发现，“第 1 页”的网站是：https://www.douban.com/group/szsh/discussion?start=0，“第 2 页”的网址是：https://www.douban.com/group/szsh/discussion?start=25，豆瓣是通过 HTTP GET 参数中 start 的变化来标识不同的页面的。所以我们可以用循环的方式把初始网站添加到队列中。如下所示，我们准备抓取前 100 个帖子：

```
func main(){
  var worklist []*collect.Request
    for i := 25; i <= 100; i += 25 {
      str := fmt.Sprintf("<https://www.douban.com/group/szsh/discussion?start=%>
      worklist = append(worklist, &collect.Request{
        Url:       str,
        ParseFunc: ParseCityList,
      })
    }
}
```


下一步，我们要解析一下抓取到的网页文本。这里我新建一个文件夹“parse”来专门存储对应网站的规则。对于首页样式的页面，我们需要获取所有帖子的 URL，这里我选择使用正则表达式的方式来实现。匹配到符合帖子格式的 URL 后，我们把它组装到一个新的 Request 中，用作下一步的爬取。

复制代码

```
const cityListRe = `(<https://www.douban.com/group/topic/[0-9a-z]+/>)"[^>]*>([^

func ParseURL(contents []byte) collect.ParseResult {
  re := regexp.MustCompile(cityListRe)

  matches := re.FindAllSubmatch(contents, -1)
  result := collect.ParseResult{}

  for _, m := range matches {
    u := string(m[1])
    result.Requesrts = append(
      result.Requesrts, &collect.Request{
        Url: u,
        ParseFunc: func(c []byte) collect.ParseResult {
          return GetContent(c, u)
        },
      })
  }
  return result
}
```

新的 Request 需要有不同的解析规则，这里我们想要获取的是正文中带有“阳台”字样的帖子（注意不要匹配到侧边栏的文字）。

查看 HTML 文本的规则会发现，正本包含在 `<div class="topic-content">xxxx <div>` 当中，所以我们可以用正则表达式这样书写规则函数，意思是当发现正文中有对应的

文字，就将当前帖子的 URL 写入到 Items 当中。


```
const ContentRe = `<div class="topic-content">[\s\S]*?阳台[\s\S]*?<div`

func GetContent(contents []byte, url string) collect.ParseResult {
  re := regexp.MustCompile(ContentRe)

  ok := re.Match(contents)
  if !ok {
    return collect.ParseResult{
      Items: []interface{}{},
    }
  }

  result := collect.ParseResult{
    Items: []interface{}{url},
  }

  return result
}
```

最后在 main 函数中，为了找到所有符合条件的帖子，我们使用了广度优先搜索算法。循环往复遍历 worklist 列表，完成爬取与解析的动作，找到所有符合条件的帖子。


```
var worklist []*collect.Request
  for i := 0; i <= 100; i += 25 {
    str := fmt.Sprintf("<https://www.douban.com/group/szsh/discussion?start=%d>
    worklist = append(worklist, &collect.Request{
      Url:       str,
      ParseFunc: doubangroup.ParseURL,
    })
  }

  var f collect.Fetcher = collect.BrowserFetch{
    Timeout: 3000 * time.Millisecond,
    Proxy:   p,
  }

  for len(worklist) > 0 {
    items := worklist
    worklist = nil
    for _, item := range items {
      body, err := f.Get(item.Url)
      time.Sleep(1 * time.Second)
      if err != nil {
```

```
        logger.Error("read content failed",
          zap.Error(err),
        )
        continue
      }
      res := item.ParseFunc(body)
      for _, item := range res.Items {
        logger.Info("result",
          zap.String("get url:", item.(string)))
      }
      worklist = append(worklist, res.Requesrts...)
    }
  }
```

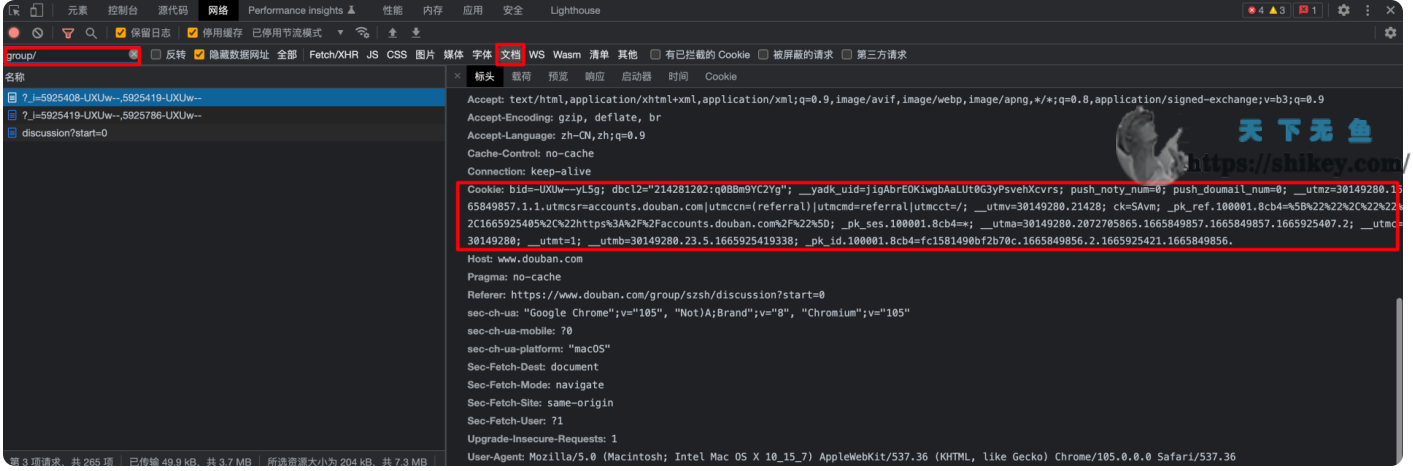

用 Cookie 突破反爬封锁

在爬取豆瓣网站时，我们会利用 time.Sleep 休眠 1 秒钟尽量减缓服务器的压力。但是，如果爬取速度太快，我们还是有可能触发服务器的反爬机制，导致我们的 IP 被封。如果出现了这种情况应该怎么办呢？

这个问题完全可以用我们之前介绍的代理来解决，通过代理我们可以假装来自不同的地方。除此之外，我还想再介绍一种突破反爬封锁的机制：Cookie。 我们实操的时候会发现，豆瓣 IP 被封锁后，会提示我们 IP 异常，需要我们重新登录。所以我们可以在浏览器中登录一下，并获得网站的 Cookie。

Cookie 是由服务器建立的文本信息。用户在浏览网站时，网页浏览器会将 Cookie 存放在电脑中。Cookie 可以让服务器在用户的浏览器上储存状态信息（如添加到购物车中的商品）或跟踪用户的浏览活动（如点击特定按钮、登录时间或浏览历史等）。

以谷歌浏览器为例，要获取当前页面的 Cookie，我们可以在当前页面中打开浏览器的开发者工具，依次选择网络 -> 文档。查找到当前页面对应的请求，就会发现一长串的 Cookie。



如果我们在 HTTP 请求头中设置了 Cookie，服务器就会认为我们是已经登录过的用户，会解除对我们的封锁。

因为参数 URL 已经无法解决像 Cookie 这样特殊的请求了，所以我们要修改一下之前的 Fetcher 接口。

```
type Fetcher interface {
  Get(url string) ([]byte, error)
}
```

修改参数为 Request，同时在 Request 中添加 Cookie。

```
type Fetcher interface {
  Get(url *Request) ([]byte, error)
}

func (b BrowserFetch) Get(request *Request) ([]byte, error) {
  if len(request.Cookie) > 0 {
      req.Header.Set("Cookie", request.Cookie)
  }
}

```

这样我们就能顺利爬取多个网站了，完整的代码可以查看 v0.1.3 分支。这里要注意一下，在学习的时候，如果我们要用相同的 IP 大量获取网站数据，最好加上长一些的休眠时间，防止把目标网站搞崩溃，也尽量避免被目标网站封禁。

总结

为了保证能够按照规则爬取完整的网站，我们需要用到一些爬取的策略。其中，深度与广度优先搜索算法就是两种经典的算法策略。

深度优先搜索就像是在“顺藤摸瓜”，它会首先查找“深”的节点。广度优先搜索则是逐层遍历树的节点，直到访问完所有节点为止。深度优先搜索需要采用递归或者模拟堆栈的形式，而广度优先搜索更简单，通过一个队列即可实现。这节课，我们就利用广度优先搜索爬取了一个豆瓣小组。由于爬取数据的过程中容易触发网站的反爬机制，我们还使用了休眠与 Cookie 来突破服务器的封锁。

使用算法爬取数据保证了爬取的完整性，它也将一个爬虫任务拆分为了多个可以并发执行的不同的爬取任务。不过，在我们这节课的案例中，爬取工作仍然是在单个协程中完成的。下节课，我们会更进一步，看看如何使用调度器并发调度这些任务。

课后题

学完这节课，请你思考下面两个问题。

1. 递归是一种非常经典的思想，但是为什么在实践中我们还是会尽量避免使用递归呢？
2. 爬虫机器人有许多特征，并不是切换 IP 就一定能骗过目标服务器，举一个例子，相同的 User-Agent 有时会被认为是同一个用户发出来的请求。如何解决这一问题?

欢迎你在留言区与我交流讨论，我们下节课见！

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。