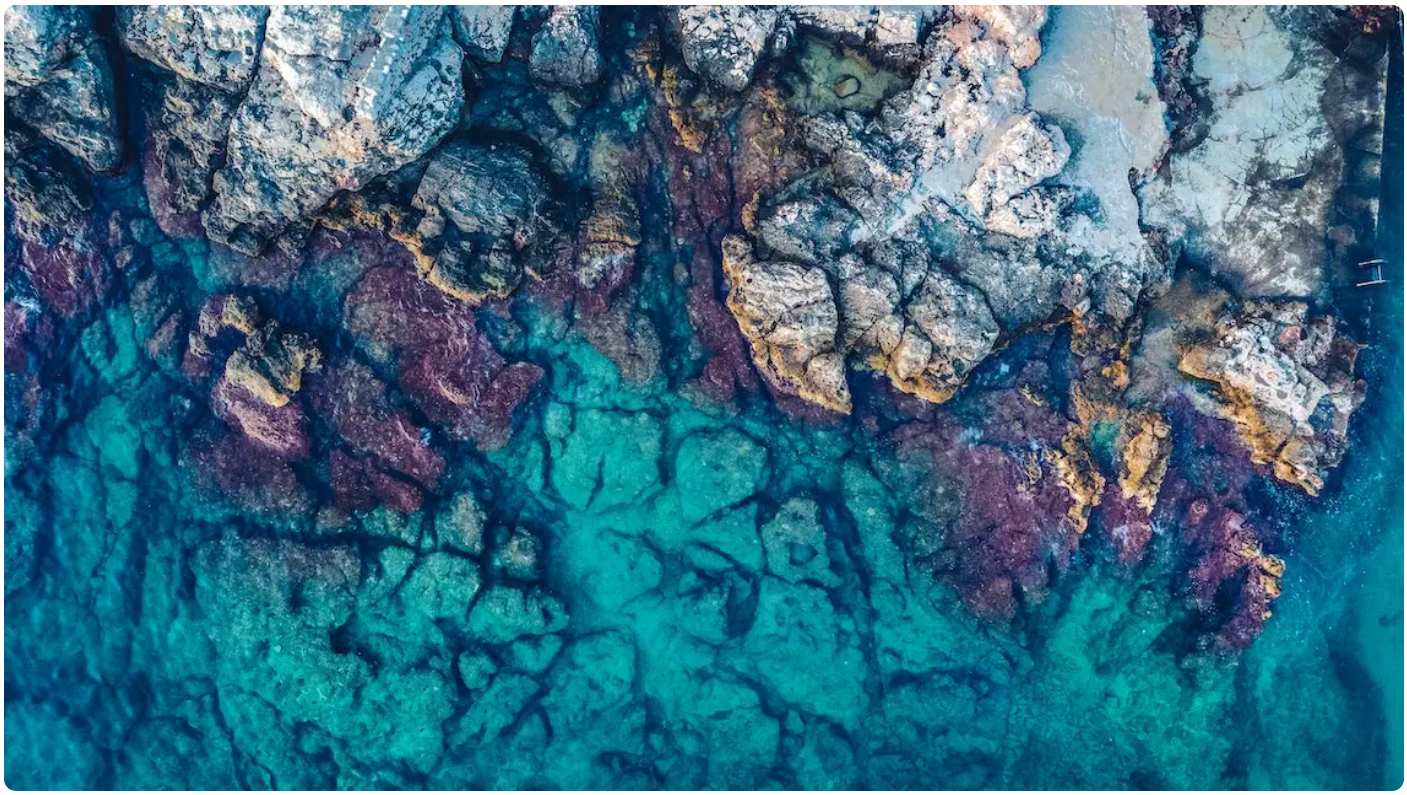
31 | 规则引擎：自定义爬虫处理规则

2022-12-20 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 10:28　大小 9.56M

你好，我是郑建勋。

这一节课，我们重点来看看如何更合理地设置爬虫任务规则。之前，我们在查找租房信息时，已经实现了有一定扩展性的程序。通过在每一个请求中加入 ParseFunc 函数，可以实现灵活的请求规则。

复制代码

```
// 单个请求
type Request struct {
  unique    string
  Task      *Task
  Url       string
  Method    string
  Depth     int
  Priority  int
  ParseFunc func([]byte, *Request) ParseResult
}
```

但我们现在仍然面临几个问题：

- 一个爬虫任务会针对不同的网站有不同的处理规则，但现在的处理方式导致多个规则之间是割裂的，不便于统一管理。
- 我们在添加初始爬虫网站 URL 时，这些种子任务是在 main 函数中注入进去的，与任务的规则之间是割裂的。但是我们需要将初始爬虫 URL 与处理规则进行统一的管理。
- 当前的爬虫任务还是需要手动初始化才能运行，可配置化程度比较低。我们希望这些写好的静态任务在程序初始化时能够自动加载。而通过外部接口，或者只要在配置文件中指定一个任务名就能将任务调度起来。
- 更进一步，我们当前的任务和规则都是静态的，静态指的是代码需要提前写好，重新编译运行才能够在运行中被调用。我们能否动态地增加任务和任务的规则，让程序能够动态地解析我们的规则呢？

为了解决上面这些问题，我们需要有专门负责规则管理的模块，并完成静态规则与动态规则的解析。

静态规则引擎

静态规则处理的确定性强，它适合对性能要求高的爬虫任务。我们可以把一个任务的规则抽象如下：

复制代码

```
type RuleTree struct {
  Root  func() []*Request // 根节点(执行入口)
  Trunk map[string]*Rule  // 规则哈希表
}

// 采集规则节点
type Rule struct {
  ParseFunc func(*Context) ParseResult // 内容解析函数
}

type Context struct {
  Body []byte
  Req  *Request
}
```

其实规则引擎就像一棵树。RuleTree.Root 是一个函数，用于生成爬虫的种子网站，而 RuleTree.Trunk 是一个规则哈希表，用于存储当前任务所有的规则，哈希表的 Key 为规则名，Value 为具体的规则。每一个规则就是一个 ParseFunc 解析函数。参数 Context 为自定义结构体，用于传递上下文信息，也就是当前的请求参数以及要解析的内容字节数组。后续还会添加请求中的临时数据等上下文数据。

下面我们沿用之前爬取租房信息的例子，将处理规则替换为使用新的静态规则引擎。

第一步，定义任务与规则。

我们在 Task 中加入了 Name 字段，将其作为一个任务唯一的标识。Task 里除了之前具有的最大深度、等待时间等属性，我们还加入了规则条件， 规则条件中 Root 生成了初始化的爬虫任务。Trunk 为爬虫任务中的所有规则。

复制代码

```
type Task struct {
    ...
    Rule        RuleTree
}

var DoubangroupTask = &collect.Task{
  Name:     "find_douban_sun_room",
  WaitTime: 1 * time.Second,
  MaxDepth: 5,
  Cookie:  "xxx",
  Rule: collect.RuleTree{
    Root: func() []*collect.Request {
      var roots []*collect.Request
      for i := 0; i < 25; i += 25 {
        str := fmt.Sprintf("<https://www.douban.com/group/szsh/discussion?start
        roots = append(roots, &collect.Request{
          Priority: 1,
          Url:      str,
          Method:   "GET",
          RuleName: "解析网站URL",
        })
      }
      return roots
    },
    Trunk: map[string]*collect.Rule{
      "解析网站URL": &collect.Rule{ParseURL},
      "解析阳台房":   &collect.Rule{GetSunRoom},
    },
  },
}
```

当前任务规则包括了“解析网站 URL” 和“解析阳台房” 这两个规则，分别对应了处理函数 ParseURL 和 GetSunRoom，如下所示。

复制代码

```
const urlListRe = `(<https://www.douban.com/group/topic/[0-9a-z]+/>)"[^>]*>([^<
const ContentRe = `<div class="topic-content">[\s\S]*?阳台[\s\S]*?<div class="as

func ParseURL(ctx *collect.Context) collect.ParseResult {
  re := regexp.MustCompile(urlListRe)

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

  for _, m := range matches {
    u := string(m[1])
    result.Requesrts = append(
      result.Requesrts, &collect.Request{
        Method:   "GET",
        Task:     ctx.Req.Task,
        Url:      u,
        Depth:    ctx.Req.Depth + 1,
        RuleName: "解析阳台房",
      })
  }
  return result
}

func GetSunRoom(ctx *collect.Context) collect.ParseResult {
  re := regexp.MustCompile(ContentRe)

  ok := re.Match(ctx.Body)
  if !ok {
    return collect.ParseResult{
      Items: []interface{}{},
    }
  }
  result := collect.ParseResult{
    Items: []interface{}{ctx.Req.Url},
  }
  return result
}
```

第二步，初始化任务与规则。

在 engine/schedule.go 文件中，init 函数中的 Store.Add 函数将任务加载到全局的任务队列中。在 Go 中，init 函数是一个特殊的函数，它会在 main 函数之前自动执行。注意，当添加的任务越来越多之后，代码会变得臃肿，这不是一种优雅的写法。后面我们还会优化它。

复制代码

```
// engine/schedule.go
func init() {
  Store.Add(doubangroup.DoubangroupTask)
}

func (c *CrawlerStore) Add(task *collect.Task) {
  c.hash[task.Name] = task
  c.list = append(c.list, task)
}

// 全局爬虫任务实例
var Store = &CrawlerStore{
  list: []*collect.Task{},
  hash: map[string]*collect.Task{},
}

type CrawlerStore struct {
  list []*collect.Task
  hash map[string]*collect.Task
}
```

第三步，启动任务。

启动爬虫任务的方式可以分为两种，一种是加载配置文件，另一种是在调用用户接口时，传递任务名称和参数。不过在这里我们先用硬编码的形式来实现。而通过配置文件和用户接口来操作任务的方式我们会有专门的课程来实现。

复制代码

```
func main(){
  ...
  seeds = append(seeds, &collect.Task{
      Name:    "find_douban_sun_room",
      Fetcher: f,
    })
  s := engine.NewEngine(
      engine.WithFetcher(f),
      engine.WithLogger(logger),
      engine.WithWorkCount(5),
      engine.WithSeeds(seeds),
```

```
        engine.WithScheduler(engine.NewSchedule()),
    )
    s.Run()
}
```


第四步，加载任务。

在调度器启动时，通过 task.Rule.Root() 获取初始化任务，并加入到任务队列中。

复制代码

```
func (e *Crawler) Schedule() {
  var reqs []*collect.Request
  for _, seed := range e.Seeds {
    task := Store.hash[seed.Name]
    // 获取初始化任务
    rootreqs := task.Rule.Root()
    reqs = append(reqs, rootreqs...)
  }
  go e.scheduler.Schedule()
  go e.scheduler.Push(reqs...)
}
```

在 Worker 处理请求时，需要从 Rule.Trunk 中获取当前请求的解析规则，并将内容和请求包装到 Context 中，调用 ParseFunc 对内容进行解析。

复制代码

```
func (s *Crawler) CreateWork() {
  for {
     ...
     //获取当前任务对应的规则
    rule := req.Task.Rule.Trunk[req.RuleName]
    // 内容解析
    result := rule.ParseFunc(&collect.Context{
      body,
      req,
    })
  // 新的任务加入队列中
    if len(result.Requesrts) > 0 {
      go s.scheduler.Push(result.Requesrts...)
    }
  }
}
```

完整的代码位于v0.2.4。

动态规则引擎

对 Go 语言这样的静态语言来说，我们需要在编译前就明确规则。这保证了程序的安全性与高性能，但是也失去了一些灵活性。像 Javascript、Python、Lua 这样的动态语言与 Go 有显著不同，它们不需要提前进行编译，能够比较灵活地书写并执行动态的规则。这一功能依赖于一种语言解释器，这种解释器一般是静态的语言编写的，例如 C/C++，解释器会解析这些动态语言的语法，然后执行相应规则。

和你分享一个实际企业中的例子。一家人工智能企业的核心产品之一是对视频流进行人脸识别。完成视频中人脸的解析涉及到视频的解码、 人脸的识别、人脸的矫正、特征的提取等多个阶段。这整个过程被称为一个 pipeline。pipeline 中的每一个阶段可能是串行的也可能是并行的。在过去，人脸、人群、物体的识别都需要单独来开发，这样开发的周期比较长，也缺乏灵活性。

为了应对未来灵活多变的检测需求，例如监测人是否摔倒，工人是否佩戴安全帽等，我们需要更短的开发周期，需要用更灵活的方式把这些阶段串联起来。这时这家公司就在 Go 中使用了 Lua 虚拟机。开发者遇到一个新的长尾需求时，通过书写 Lua 脚本来定义新的规则。在 Go 程序中通过动态加载 Lua 脚本来实现灵活性。

我们现在的爬虫项目其实也面邻着一样的问题。网站和规则是多种多样的，我们无法穷尽所有的规则，如果每次遇到新网站都要重新写代码，写爬取规则然后重启程序，这会比较繁琐，所以我们希望能够动态地在程序运行过程中加载规则。

动态规则带来的另一个好处是，降低了书写代码规则的门槛，它甚至可以让业务人员也能书写简单的规则。**说到在爬虫项目中实现动态规则的引擎，我们首先想到的就是使用 Javascript 虚拟机了。因为使用 JS 操作网页有天然的优势。**

自己要在短时间内实现一个工业级的虚拟机可能比较困难，我们可以使用一些开源的项目，例如otto。otto 是用 Go 编写的 JavaScript 虚拟机，用于在 Go 中执行 Javascript 语法。

下面是用 otto 编写的一个简单的例子。在这个例子中，script 字符串即为要执行的 Javascript 语法，console.log 是 JS 中的函数，用于打印变量。

```go
package main

import (
  "fmt"
  "github.com/robertkrimen/otto"
)

func main() {
  vm := otto.New()
  script := `
    var n = 100;
    console.log("hello-" + n);
    n;
  `
  value, _ := vm.Run(script)
  fmt.Println("value:", value.String())
}
```


最终结果为：


```
hello-100
value: 100
```

这样，我们就实现了在 Go 语言中执行 JS 脚本的目的。实际上，otto 内部解析了这一串字符串，并按照 JS 语法中对应的规则进行了相应的处理，例如脚本中的 console.log 函数最终其实也调用了 Go 中的 fmt 函数，实现将文本打印到控制台。不过我们要注意的是，不一定 JS 的所有语法 JS 虚拟机都是支持的，是否支持取决于当前虚拟机的实现。例如当前 otto 支持 JS5 语法，但是不支持 JS6 语法。

下面我们我们仍然以爬取租房信息为例，借助 otto 库实现的 JS 虚拟机来实现动态的规则引擎。

第一步，构建动态规则模型 TaskModle。


```go
type (
  TaskModle struct {
    Name     string        `json:"name"` // 任务名称，应保证唯一性
    Url      string        `json:"url"`
```

```
    Cookie   string        `json:"cookie"`
    WaitTime time.Duration `json:"wait_time"`
    Reload   bool          `json:"reload"` // 网站是否可以重复爬取
    MaxDepth int64         `json:"max_depth"`
    Root  string           `json:"root_script"`
    Rules []RuleModle      `json:"rule"`
  }
  RuleModle struct {
    Name      string `json:"name"`
    ParseFunc string `json:"parse_script"`
  }
)
```


为什么这里要单独构建一个任务的结构体呢？主要原因在于，现在我们的规则都是字符串了，这和之前的静态规则引擎有本质的不同。其中，TaskModle.Root 为初始化种子节点的 JS 脚本，TaskModle.Rules 为具体爬虫任务的规则树。

第二步，书写动态爬虫规则。

示例代码如下。其中，Root 脚本就是我们要生成的种子网站 URL。在这里我们构建了一个 JS 数组 arr，将生成的请求数组添加到 arr 之后，又调用了 AddJsReq 函数。AddJsReq 函数其实是一个 Go 函数，用于最终生成 Go 中的请求数组。在这里我们可以看到，在 Go 的 JS 虚拟机中，还可以灵活地调用原生的 Go 函数。

复制代码

```
var DoubangroupJSTask = &collect.TaskModle{
  Property: collect.Property{
    Name:     "js_find_douban_sun_room",
    WaitTime: 1 * time.Second,
    MaxDepth: 5,
    Cookie:   "xxx",
  },
  Root: `
    var arr = new Array();
     for (var i = 25; i <= 25; i+=25) {
      var obj = {
         Url: "<https://www.douban.com/group/szsh/discussion?start=>" + i,
         Priority: 1,
         RuleName: "解析网站URL",
         Method: "GET",
       };
      arr.push(obj);
    };
    console.log(arr[0].Url);
    AddJsReq(arr);
```

```
    `,
  Rules: []collect.RuleModle{
    {
      Name: "解析网站URL",
      ParseFunc: `
      ctx.ParseJSReg("解析阳台房","(<https://www.douban.com/group/topic/[0-9a-z]+
      `,
    },
    {
      Name: "解析阳台房",
      ParseFunc: `
      //console.log("parse output");
      ctx.OutputJS("<div class=\\"topic-content\\">[\\\\s\\\\S]*?阳台[\\\\s\\\\S
      `,
    },
  },
}
```


而在 Rules 脚本中，我们加入了两个爬虫规则，分别是“解析网站 URL”和 “解析阳台房”，他们都可以使用非常简单的规则来实现。在这里我们调用了 ctx.ParseJSReg 来解析请求，调用了 ctx.OutputJS 来解析并输出找到的内容，注意这里的 ctx.ParseJSReg 与 ctx.OutputJS 也是 Go 原生的函数，下面我们会看到他们的实现。

第三步，书写动态规则中的 Go 函数。

AddJsReqs 函数将在 JS 脚本中的请求数据变为 Go 结构中的数组[]*collect.Request，而 ctx.ParseJSReg 方法则会动态解析 JS 中传递的正则表达式并生成新的请求, ctx.OutputJS 负责解析传递过来的正则表达式并完成结果的输出。注意 JS 虚拟机会自动将 JS 脚本中的参数转换为函数参数中对应的结构。

复制代码

```go
// 用于动态规则添加请求。
func AddJsReqs(jreqs []map[string]interface{}) []*collect.Request {
  reqs := make([]*collect.Request, 0)

  for _, jreq := range jreqs {
    req := &collect.Request{}
    u, ok := jreq["Url"].(string)
    if !ok {
      return nil
    }
    req.Url = u
    req.RuleName, _ = jreq["RuleName"].(string)
    req.Method, _ = jreq["Method"].(string)
    req.Priority, _ = jreq["Priority"].(int64)
```

```
    reqs = append(reqs, req)
  }
  return reqs
}

// 动态解析JS中的正则表达式
func (c *Context) ParseJSReg(name string, reg string) ParseResult {
  re := regexp.MustCompile(reg)

  matches := re.FindAllSubmatch(c.Body, -1)
  result := ParseResult{}

  for _, m := range matches {
    u := string(m[1])
    result.Requesrts = append(
      result.Requesrts, &Request{
        Method:   "GET",
        Task:     c.Req.Task,
        Url:      u,
        Depth:    c.Req.Depth + 1,
        RuleName: name,
      })
  }
  return result
}

// 解析内容并输出结果
func (c *Context) OutputJS(reg string) ParseResult {
  re := regexp.MustCompile(reg)
  ok := re.Match(c.Body)
  if !ok {
    return ParseResult{
      Items: []interface{}{},
    }
  }
  result := ParseResult{
    Items: []interface{}{c.Req.Url},
  }
  return result
}
```


第四步，初始化任务与规则。

初始化动态规则这一步更复杂一些，因为我们需要将 JS 脚本放入 paesrFunc 函数中，供 otto 库解析，代码如下所示。


```
func init() {
```

```
    ...
  Store.AddJSTask(doubangroup.DoubangroupJSTask)
}

func (c *CrawlerStore) AddJSTask(m *collect.TaskModle) {
  task := &collect.Task{
    Property: m.Property,
  }

  task.Rule.Root = func() ([]*collect.Request, error) {
    vm := otto.New()
    vm.Set("AddJsReq", AddJsReqs)
    v, err := vm.Eval(m.Root)
    e, err := v.Export()
    return e.([]*collect.Request), nil
  }

  for _, r := range m.Rules {
    paesrFunc := func(parse string) func(ctx *collect.Context) (collect.ParseRe
      return func(ctx *collect.Context) (collect.ParseResult, error) {
        vm := otto.New()
        vm.Set("ctx", ctx)
        v, err := vm.Eval(parse)
        e, err := v.Export()
        return e.(collect.ParseResult), err
      }
    }(r.ParseFunc)
    if task.Rule.Trunk == nil {
      task.Rule.Trunk = make(map[string]*collect.Rule, 0)
    }
    task.Rule.Trunk[r.Name] = &collect.Rule{
      paesrFunc,
    }
  }

  c.hash[task.Name] = task
  c.list = append(c.list, task)
}
```


在这里，用于生成种子网站的 Root 函数中的 vm.Eval(m.Root) 执行了配置中的 root 脚本，然后返回了生成的请求数组。vm.Set(“AddJsReq”, AddJsReqs) 可以将 Go 原生函数注册到 JS 虚拟机中，这样我们才能在 JS 脚本中调用 Go 函数。paesrFunc 函数也是一样，在这里我们使用了闭包，方便后续执行 parse 脚本并最后返回解析后的 collect.ParseResult 结果。

第五步，启动并加载任务。

启动任务和加载任务与静态规则引擎代码完全一致，完全复用就可以了。到这里，我们只需要指定一个任务名 js_find_douban_sun_room 就可以利用动态规则引擎将爬虫任务跑起来了。


```
func main(){
  ...
  seeds = append(seeds, &collect.Task{
      Property: collect.Property{
        Name: "js_find_douban_sun_room",
      },
      Fetcher: f,
    })

    s := engine.NewEngine(
      engine.WithFetcher(f),
      engine.WithLogger(logger),
      engine.WithWorkCount(5),
      engine.WithSeeds(seeds),
      engine.WithScheduler(engine.NewSchedule()),
    )

    s.Run()
}
```

完整的代码位于v0.2.5。

总结

在本节课程中，为了更好地对爬虫任务进行管理，我们构建了规则引擎模块来管理静态和动态的规则。

静态的规则指的是需要在 Go 代码中提前写好的解析规则，这些规则固定、性能更高。但是由于我们无法穷尽所有的网站和规则，所以每加入一个网站都需要修改代码，还要重新启动程序，过程比较繁琐。动态的规则构建了 Javascript 的虚拟机，它会动态解析 JS 的语法规则。

其实从我们现在的实现来看，用这些简单的规则就可以支持许多简单的爬取规则。在 JS 语法中，我们实现了调用 Go 函数的能力，这让书写 JS 脚本变得更加简单。我们甚至可以想象这样一个场景，用户只要在页面中选择自己希望爬取的内容，就足够我们生成动态的规则满足客户的爬取需求了。这种商业模式是成立的，用我们现在的技术完全可以实现。

不过，动态的规则当前的性能确实比不上静态的规则，因为它内部有大量运算、内存分配还有反射的使用。但是在爬虫项目中，很多时候真正的瓶颈是来自于网络 I/O。因此通常动态规则带来的灵活性收益要大于其性能的损耗。

课后题

你认为下面这个最基本的 JS 脚本，在 JS 虚拟机内部是如何实现的？

复制代码

```
script := `
  var n = 1+2;
  console.log("hello-" + n);
  n;
```

欢迎你在留言区与我交流讨论，我们下节课再见！

上一篇 30 | 辅助任务管理：任务优先级、去重与失败处理

下一篇 32 | 存储引擎：数据清洗与存储

精选留言 (2)

Realm

2022-12-20 来自浙江

课程渐入佳境了，知识越来越有趣味了。老师，有时间可以把每节课的思考题答疑下，那就更好了。

思考题：

猜测是用AST抽象语法树+反射，解析js，转成Golang语法。

疑问：

```
type CrawlerStore struct {
	list []*collect.Task
	hash map[string]*collect.Task
}
```

这个list的设计是用于做什么？好像程序中没有提到。


翡翠虎

2022-12-20 来自广西

设计一个这样的引擎有什么好处？如果把xpath规则或者正则放到数据库，运行的时候随着任务传递给程序，程序按几个预设配置处理，会不会更好？