30 | 辅助任务管理：任务优先级、去重与失败处理

2022-12-17 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 07:40　大小 7.01M

你好，我是郑建勋。

这节课，让我们给系统加入一些辅助功能，把爬虫流程变得更完善一些。这些功能包括：爬虫最大深度、请求不重复、优先队列、以及随机的 User-Agent。

设置爬虫最大深度

当我们用深度和广度优先搜索爬取一个网站时，为了防止访问陷入到死循环，同时控制爬取的有效链接的数量，一般会给当前任务设置一个最大爬取深度。最大爬取深度是和任务有关的，因此我们要在 Request 中加上 MaxDepth 这个字段，它可以标识到爬取的最大深度。Depth 则表示任务的当前深度，最初始的深度为 0。

复制代码

```
type Request struct {
  Url       string
```

```
  Cookie    string
  WaitTime  time.Duration
    Depth     int
  MaxDepth  int
  ParseFunc func([]byte, *Request) ParseResult
}
```


那在异步爬取的情况下，我们怎么知道当前网站的深度呢？最好的时机是在采集引擎采集并解析爬虫数据，并将下一层的请求放到队列中的时候。以我们之前写好的 ParseURL 函数为例，在添加下一层的 URL 时，我们将 Depth 加 1，这样就标识了下一层的深度。

复制代码

```go
func ParseURL(contents []byte, req *collect.Request) collect.ParseResult {
  re := regexp.MustCompile(urlListRe)

  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,
        WaitTime: req.WaitTime,
        Cookie:   req.Cookie,
        Depth:    req.Depth + 1,
        MaxDepth: req.MaxDepth,
        ParseFunc: func(c []byte, request *collect.Request) collect.ParseResult
          return GetContent(c, u)
        },
      })
  }
  return result
}
```

最后一步，我们在爬取新的网页之前，判断最大深度。如果当前深度超过了最大深度，那就不再进行爬取。这部分的完整代码你可以查看分支 v0.1.7。

复制代码

```go
func (r *Request) Check() error {
  if r.Depth > r.MaxDepth {
    return errors.New("Max depth limit reached")
  }
  return nil
```

```
}

func (s *Schedule) CreateWork() {
  for {
    r := <-s.workerCh
    if err := r.Check(); err != nil {
      s.Logger.Error("check failed",
        zap.Error(err),
      )
      continue
    }
    ...
  }
}
```


避免请求重复

为了避免爬取时候的死循环，避免无效的爬取，我们常常需要检测请求是否重复，这时我们需要考虑 3 个问题：

- 用什么数据结构来存储数据才能保证快速地查找到请求的记录？
- 如何保证并发查找与写入时，不出现并发冲突问题？
- 在什么条件下，我们才能确认请求是重复的，从而停止爬取？

要解决第一个问题我们可以用一个简单高效的结构：哈希表。我们可以借助哈希表查找 o(1)。另外，由于 Go 语言中的哈希表是不支持并发安全的，为了解决第二个问题，我们还需要在此基础上加一个互斥锁。而第三个问题我们需要在爬虫采取之前进行检查。

在解决上面的三个问题之前，我们先优化一下代码。我们之前的 `Request` 结构体会在每一次请求时发生变化，但是我们希望有一个字段能够表示一整个网站的爬取任务，因此我们需要抽离出一个新的结构 Task 作为一个爬虫任务，而 `Request` 则作为单独的请求存在。有些参数是整个任务共有的，例如 Task 中的 Cookie、MaxDepth（最大深度）、WaitTime（默认等待时间）和 RootReq（任务中的第一个请求）。

复制代码

```
type Task struct {
  Url         string
  Cookie      string
  WaitTime    time.Duration
  MaxDepth    int
  RootReq     *Request
```

```
  Fetcher     Fetcher
}

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


由于抽象出了 Task，代码需要做对应的修改，例如我们需要把初始的 Seed 种子任务替换为 Task 结构。

复制代码

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

同时，在深度检查时，每一个请求的最大深度需要从 Task 字段中获取。

复制代码

```
func (r *Request) Check() error {
  if r.Depth > r.Task.MaxDepth {
    return errors.New("Max depth limit reached")
  }
  return nil
}
```

完整代码你可以查看 v0.1.8。

接下来，我们继续用一个哈希表结构来存储历史请求。由于我们希望随时访问哈希表中的历史请求，所以把它放在 Request、Task 中都不合适。 放在调度引擎中也不合适，因为调度引擎

从功能上讲，应该只负责调度才对。所以，我们还需要完成一轮抽象，将调度引擎抽离出来作为一个接口，让它只做调度的工作，不用负责存储全局变量等任务。

所以我们就构建一个新的结构 Crawler 作为全局的爬取实例，将之前 Schedule 中的 options 迁移到 Crawler 中，Schedule 只处理与调度有关的工作，并抽象为了 Scheduler 接口。


```
type Crawler struct {
  out chan collect.ParseResult
  options
}

type Scheduler interface {
  Schedule()
  Push(...*collect.Request)
  Pull() *collect.Request
}

type Schedule struct {
  requestCh chan *collect.Request
  workerCh  chan *collect.Request
  reqQueue  []*collect.Request
  Logger    *zap.Logger
}
```

在 Scheduler 中，Schedule 方法负责启动调度器，Push 方法会将请求放入到调度器中，而 Pull 方法则会从调度器中获取请求。我们也需要对代码做相应的调整，这里就不再赘述了，具体你可以参考v0.1.9。调度器抽象为接口后，如果我们有其他的调度器算法实现，也能够非常方便完成替换了。

现在，我们在 Crawler 中加入 Visited 哈希表，用它存储请求访问信息，增加 VisitedLock 来确保并发安全。


```
type Crawler struct {
  out         chan collect.ParseResult
  Visited     map[string]bool
  VisitedLock sync.Mutex
  options
}
```

Visited 中的 Key 是请求的唯一标识，我们现在先将唯一标识设置为 URL + method 方法，并使用 MD5 生成唯一键。后面我们还会为唯一标识加上当前请求的规则条件。

复制代码

```
// 请求的唯一识别码
func (r *Request) Unique() string {
  block := md5.Sum([]byte(r.Url + r.Method))
  return hex.EncodeToString(block[:])
}
```

接着，编写 HasVisited 方法，判断当前请求是否已经被访问过。StoreVisited 方法用于将请求存储到 Visited 哈希表中。

复制代码

```
func (e *Crawler) HasVisited(r *collect.Request) bool {
  e.VisitedLock.Lock()
  defer e.VisitedLock.Unlock()
  unique := r.Unique()
  return e.Visited[unique]
}

func (e *Crawler) StoreVisited(reqs ...*collect.Request) {
  e.VisitedLock.Lock()
  defer e.VisitedLock.Unlock()

  for _, r := range reqs {
    unique := r.Unique()
    e.Visited[unique] = true
  }
}
```

最后在 Worker 中，在执行 request 前，判断当前请求是否已被访问。如果请求没有被访问过，将 request 放入 Visited 哈希表中。

复制代码

```
func (s *Crawler) CreateWork() {
  for {
    r := s.scheduler.Pull()
    if err := r.Check(); err != nil {
      s.Logger.Error("check failed",
        zap.Error(err),
      )
```

```
        continue
      }
      // 判断当前请求是否已被访问
      if s.HasVisited(r) {
        s.Logger.Debug("request has visited",
          zap.String("url:", r.Url),
        )
        continue
      }
      // 设置当前请求已被访问
      s.StoreVisited(r)
      ...
    }
}
```


最后要注意的是，哈希表需要用 make 进行初始化，要不然在运行时访问哈希表会直接报错。（完整的代码位于v0.2.0）。

设置优先队列

我们要给项目增加的第三个功能就是优先队列。

爬虫任务的优先级有时并不是相同的，一些任务需要优先处理。因此，接下来我们就来设置一个任务的优先队列。优先队列还可以分成多个等级，不过在这里我将它简单地分为了两个等级，即优先队列和普通队列。优先级更高的请求会存储到 priReqQueue 优先队列中。


```
type Schedule struct {
  requestCh   chan *collect.Request
  workerCh    chan *collect.Request
  priReqQueue []*collect.Request
  reqQueue    []*collect.Request
  Logger      *zap.Logger
}
```

在调度函数 Schedule 中，我们会优先从优先队列中获取请求。而在放入请求时，如果请求的优先级更高，也会单独放入优先级队列。

最后我们还修复了之前遗留的一个 Bug，将变量 req、ch 放置到 for 循环外部，防止丢失请求的可能性。

复制代码

```
func (s *Schedule) Schedule() {
  var req *collect.Request
  var ch chan *collect.Request
  for {
    if req == nil && len(s.priReqQueue) > 0 {
      req = s.priReqQueue[0]
      s.priReqQueue = s.priReqQueue[1:]
      ch = s.workerCh
    }
    if req == nil && len(s.reqQueue) > 0 {
      req = s.reqQueue[0]
      s.reqQueue = s.reqQueue[1:]
      ch = s.workerCh
    }
    select {
    case r := <-s.requestCh:
      if r.Priority > 0 {
        s.priReqQueue = append(s.priReqQueue, r)
      } else {
        s.reqQueue = append(s.reqQueue, r)
      }
    case ch <- req:
      req = nil
      ch = nil
    }
  }
}
```


执行后输出结果为：

复制代码

```
{"level":"INFO","ts":"2022-11-05T21:40:18.339+0800","caller":"crawler/main.go:1
{"level":"INFO","ts":"2022-11-05T21:40:22.067+0800","caller":"engine/schedule.g
{"level":"INFO","ts":"2022-11-05T21:40:22.150+0800","caller":"engine/schedule.g
...
```

完整的代码位于v0.2.1。

设置随机 User-Agent

我们给项目增加的第四个功能是 User-Agent 随机性。 为了避免服务器检测到我们使用了同一个 User-Agent，继而判断出是同一个客户端在发出请求，我们可以为发送的 User-Agent 加

入随机性。这个操作的本质就是将浏览器的不同型号与不同版本拼接起来，组成一个新的 User-Agent。

随机生成 User-Agent 的逻辑位于 extensions/randomua.go 中，里面枚举了不同型号的浏览器和不同型号的版本，并且通过排列组合产生了不同的 User-Agent。

最后一步，我们要在采集引擎中调用 GenerateRandomUA 函数，将请求头设置为随机的 User-Agent，如下所示：

复制代码

```
func (b BrowserFetch) Get(request *spider.Request) ([]byte, error) {
    ...
    req.Header.Set("User-Agent", extensions.GenerateRandomUA())
    resp, err := client.Do(req)
```

完整的代码你可以参考 v0.2.2 分支。

进行失败处理

在课程的最后，我们来看一看失败处理。

我们在爬取网站时，网络超时等诸多潜在风险都可能导致爬取失败。这时，我们可以对失败的任务进行重试。但是如果网站多次失败，那就没有必要反复重试了，我们可以将它们放入单独的队列中。为了防止失败请求日积月久导致的内存泄露，同时也为了在程序崩溃后能够再次加载这些失败网站，我们最后还需要将这些失败网站持久化到数据库或文件中。

这节课我们先完成前半部分，即失败重试。后半部分会在第 32 讲存储引擎中详细介绍。我们要在全局 Crawler 中存储 failures 哈希表，设置 Key 为请求的唯一键，用于快速查找。failureLock 互斥锁用于并发安全。

复制代码

```
type Crawler struct {
    ...
    failures    map[string]*collect.Request // 失败请求id -> 失败请求
    failureLock sync.Mutex
}
```

当请求失败之后，调用 SetFailure 方法将请求加入到 failures 哈希表中，并且把它重新交由调度引擎进行调度。这里我们为任务 Task 引入了一个新的字段 Reload，标识当前任务的网页是否可以重复爬取。如果不可以重复爬取，我们需要在失败重试前删除 Visited 中的历史记录。

复制代码

```
func (e *Crawler) SetFailure(req *collect.Request) {
    if !req.Task.Reload {
        e.VisitedLock.Lock()
        unique := req.Unique()
        delete(e.Visited, unique)
        e.VisitedLock.Unlock()
    }
    e.failureLock.Lock()
    defer e.failureLock.Unlock()
    if _, ok := e.failures[req.Unique()]; !ok {
        // 首次失败时，再重新执行一次
        e.failures[req.Unique()] = req
        e.scheduler.Push(req)
    }
    // todo: 失败2次，加载到失败队列中
}
```

如果失败两次，就将请求单独加载到失败队列中，并在后续进行持久化，这一步操作我们之后再进行。失败重试的完整代码位于v0.2.3。

总结

这节课，我们为爬虫系统增加了丰富的辅助任务，包括：设置爬虫的最大深度，避免了重复爬取、设置优先队列、设置随机的 User-Agent，另外我们还进行了任务失败的处理。

随着爬虫系统的演进，我们还会有更加复杂的功能与需求。后面的内容，我们还会介绍限速器，并借助 JS 虚拟机实现更加灵活的爬虫系统。

课后题

学完这节课也给你留一道思考题。

之前我们实现了任务的优先队列，但我们目前只支持两种优先级。如果要支持多种优先级，你知道如何实现吗？

欢迎你在留言区与我交流讨论，我们下节课见。

上一篇　29 | 细节决定成败：切片与哈希表的陷阱与原理

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。