**6. 组件的使用和定制**

在第一章里，我们提到了WebMagic的组件。WebMagic的一大特色就是可以灵活的定制组件功能，实现你自己想要的功能。

在Spider类里，PageProcessor、Downloader、Scheduler和Pipeline四个组件都是Spider的字段。除了PageProcessor是在Spider创建的时候已经指定，Downloader、Scheduler和Pipeline都可以通过Spider的setter方法来进行配置和更改。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| setScheduler() | 设置Scheduler | spipder.setScheduler(new FileCacheQueueScheduler("D:\data\webmagic")) |
| setDownloader() | 设置Downloader | spipder.setDownloader(new SeleniumDownloader())) |
| addPipeline() | 设置Pipeline，一个Spider可以有多个Pipeline | spipder.addPipeline(new FilePipeline()) |

在这一章，我们会讲到如何定制这些组件，完成我们想要的功能。

### 6.1 使用和定制Pipeline

Pileline是抽取结束后，进行处理的部分，它主要用于抽取结果的保存，也可以定制Pileline可以实现一些通用的功能。在这一节中，我们会对Pipeline进行介绍，并用两个例子来讲解如何定制Pipeline。

#### 6.1.1 Pipeline介绍

Pipeline的接口定义如下：

public interface Pipeline {

// ResultItems保存了抽取结果，它是一个Map结构，

// 在page.putField(key,value)中保存的数据，可以通过ResultItems.get(key)获取

public void process(ResultItems resultItems, Task task);

}

可以看到，Pipeline其实就是将PageProcessor抽取的结果，继续进行了处理的，其实在Pipeline中完成的功能，你基本上也可以直接在PageProcessor实现，那么为什么会有Pipeline？有几个原因：

1. 为了模块分离。“页面抽取”和“后处理、持久化”是爬虫的两个阶段，将其分离开来，一个是代码结构比较清晰，另一个是以后也可能将其处理过程分开，分开在独立的线程以至于不同的机器执行。
2. Pipeline的功能比较固定，更容易做成通用组件。每个页面的抽取方式千变万化，但是后续处理方式则比较固定，例如保存到文件、保存到数据库这种操作，这些对所有页面都是通用的。WebMagic中就已经提供了控制台输出、保存到文件、保存为JSON格式的文件几种通用的Pipeline。

在WebMagic里，一个Spider可以有多个Pipeline，使用Spider.addPipeline()即可增加一个Pipeline。这些Pipeline都会得到处理，例如你可以使用

spider.addPipeline(new ConsolePipeline()).addPipeline(new FilePipeline())

实现输出结果到控制台，并且保存到文件的目标。

#### 6.1.2 将结果输出到控制台

在介绍PageProcessor时，我们使用了[GithubRepoPageProcessor](https://github.com/code4craft/webmagic/blob/master/webmagic-core/src/main/java/us/codecraft/webmagic/processor/example/GithubRepoPageProcessor.java)作为例子，其中某一段代码中，我们将结果进行了保存：

public void process(Page page) {

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+)").all());

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+)").all());

//保存结果author，这个结果会最终保存到ResultItems中

page.putField("author", page.getUrl().regex("https://github\\.com/(\\w+)/.\*").toString());

page.putField("name", page.getHtml().xpath("//h1[@class='entry-title public']/strong/a/text()").toString());

if (page.getResultItems().get("name")==null){

//设置skip之后，这个页面的结果不会被Pipeline处理

page.setSkip(true);

}

page.putField("readme", page.getHtml().xpath("//div[@id='readme']/tidyText()"));

}

现在我们想将结果保存到控制台，要怎么做呢？[ConsolePipeline](https://github.com/code4craft/webmagic/blob/master/webmagic-core/src/main/java/us/codecraft/webmagic/pipeline/ConsolePipeline.java)可以完成这个工作：

public class ConsolePipeline implements Pipeline {

@Override

public void process(ResultItems resultItems, Task task) {

System.out.println("get page: " + resultItems.getRequest().getUrl());

//遍历所有结果，输出到控制台，上面例子中的"author"、"name"、"readme"都是一个key，其结果则是对应的value

for (Map.Entry<String, Object> entry : resultItems.getAll().entrySet()) {

System.out.println(entry.getKey() + ":\t" + entry.getValue());

}

}

}

参考这个例子，你就可以定制自己的Pipeline了——从ResultItems中取出数据，再按照你希望的方式处理即可。

#### 6.1.3 将结果保存到MySQL

这里先介绍一个demo项目：[jobhunter](https://github.com/webmagic-io/jobhunter)。它是一个集成了Spring，使用WebMagic抓取招聘信息，并且使用Mybatis持久化到Mysql的例子。我们会用这个项目来介绍如何持久化到Mysql。

在Java里，我们有很多方式将数据保存到MySQL，例如jdbc、dbutils、spring-jdbc、MyBatis等工具。这些工具都可以完成同样的事情，只不过功能和使用复杂程度不一样。如果使用jdbc，那么我们只需要从ResultItems取出数据，进行保存即可。

如果我们会使用ORM框架来完成持久化到MySQL的工作，就会面临一个问题：这些框架一般都要求保存的内容是一个定义好结构的对象，而不是一个key-value形式的ResultItems。以MyBatis为例，我们使用[MyBatis-Spring](http://mybatis.github.io/spring/zh/)可以定义这样一个DAO：

public interface JobInfoDAO {

@Insert("insert into JobInfo (`title`,`salary`,`company`,`description`,`requirement`,`source`,`url`,`urlMd5`) values (#{title},#{salary},#{company},#{description},#{requirement},#{source},#{url},#{urlMd5})")

public int add(LieTouJobInfo jobInfo);

}

我们要做的，就是实现一个Pipeline，将ResultItems和LieTouJobInfo对象结合起来。

#### 注解模式

注解模式下，WebMagic内置了一个[PageModelPipeline](https://github.com/code4craft/webmagic/blob/master/webmagic-extension/src/main/java/us/codecraft/webmagic/pipeline/PageModelPipeline.java)：

public interface PageModelPipeline<T> {

//这里传入的是处理好的对象

public void process(T t, Task task);

}

这时，我们可以很优雅的定义一个[JobInfoDaoPipeline](https://github.com/webmagic-io/jobhunter/blob/master/src/main/java/us/codecraft/jobhunter/pipeline/JobInfoDaoPipeline.java)，来实现这个功能：

@Component("JobInfoDaoPipeline")

public class JobInfoDaoPipeline implements PageModelPipeline<LieTouJobInfo> {

@Resource

private JobInfoDAO jobInfoDAO;

@Override

public void process(LieTouJobInfo lieTouJobInfo, Task task) {

//调用MyBatis DAO保存结果

jobInfoDAO.add(lieTouJobInfo);

}

}

#### 基本Pipeline模式

至此，结果保存就已经完成了！那么如果我们使用原始的Pipeline接口，要怎么完成呢？其实答案也很简单，如果你要保存一个对象，那么就需要在抽取的时候，将它保存为一个对象：

public void process(Page page) {

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+)").all());

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+)").all());

GithubRepo githubRepo = new GithubRepo();

githubRepo.setAuthor(page.getUrl().regex("https://github\\.com/(\\w+)/.\*").toString());

githubRepo.setName(page.getHtml().xpath("//h1[@class='entry-title public']/strong/a/text()").toString());

githubRepo.setReadme(page.getHtml().xpath("//div[@id='readme']/tidyText()").toString());

if (githubRepo.getName() == null) {

//skip this page

page.setSkip(true);

} else {

page.putField("repo", githubRepo);

}

}

在Pipeline中，只要使用

GithubRepo githubRepo = (GithubRepo)resultItems.get("repo");

就可以获取这个对象了。

PageModelPipeline实际上也是通过原始的Pipeline来实现的，它将与PageProcessor进行了整合，在保存时，使用类名作为key，而对象则是value，具体实现见：[ModelPipeline](https://github.com/code4craft/webmagic/blob/master/webmagic-extension/src/main/java/us/codecraft/webmagic/model/ModelPipeline.java)。

#### 6.1.4 WebMagic已经提供的几个Pipeline

WebMagic中已经提供了将结果输出到控制台、保存到文件和JSON格式保存的几个Pipeline：

| **类** | **说明** | **备注** |
| --- | --- | --- |
| ConsolePipeline | 输出结果到控制台 | 抽取结果需要实现toString方法 |
| FilePipeline | 保存结果到文件 | 抽取结果需要实现toString方法 |
| JsonFilePipeline | JSON格式保存结果到文件 |  |
| ConsolePageModelPipeline | (注解模式)输出结果到控制台 |  |
| FilePageModelPipeline | (注解模式)保存结果到文件 |  |
| JsonFilePageModelPipeline | (注解模式)JSON格式保存结果到文件 | 想要持久化的字段需要有getter方法 |

### 6.2 使用和定制Scheduler

Scheduler是WebMagic中进行URL管理的组件。一般来说，Scheduler包括两个作用：

1. 对待抓取的URL队列进行管理（push和pop）。
2. 对已抓取的URL进行去重。

WebMagic内置了几个常用的Scheduler。如果你只是在本地执行规模比较小的爬虫，那么基本无需定制Scheduler，但是了解一下已经提供的几个Scheduler还是有意义的。

| **类** | **说明** | **备注** |
| --- | --- | --- |
| DuplicateRemovedScheduler | 抽象基类，提供一些模板方法 | 继承它可以实现自己的功能 |
| QueueScheduler | 使用内存队列保存待抓取URL |  |
| PriorityScheduler | 使用带有优先级的内存队列保存待抓取URL | 耗费内存较QueueScheduler更大，但是当设置了request.priority之后，只能使用PriorityScheduler才可使优先级生效 |
| FileCacheQueueScheduler | 使用文件保存抓取URL，可以在关闭程序并下次启动时，从之前抓取到的URL继续抓取 | 需指定路径，会建立.urls.txt和.cursor.txt两个文件 |
| RedisScheduler | 使用Redis保存抓取队列(me:redis的sortedset)，可进行多台机器同时合作抓取 | 需要安装并启动redis |

在0.5.1版本里，我对Scheduler的内部实现进行了重构，去重部分被单独抽象成了一个接口：DuplicateRemover，从而可以为同一个Scheduler选择不同的去重方式，以适应不同的需要，目前提供了两种去重方式。

| **类** | **说明** |
| --- | --- |
| HashSetDuplicateRemover | 使用HashSet来进行去重，占用内存较大 |
| BloomFilterDuplicateRemover | 使用BloomFilter来进行去重，占用内存较小，但是可能漏抓页面 |

所有默认的Scheduler都使用HashSetDuplicateRemover来进行去重，（除开RedisScheduler是使用Redis的set进行去重）。如果你的URL较多，使用HashSetDuplicateRemover会比较占用内存，所以也可以尝试以下BloomFilterDuplicateRemover[1](http://webmagic.io/docs/zh/posts/ch6-custom-componenet/scheduler.html#fn_1)，使用方式：

spider.setScheduler(new QueueScheduler()

.setDuplicateRemover(new BloomFilterDuplicateRemover(10000000)) //10000000是估计的页面数量

)

1. 0.6.0版本后，如果使用BloomFilterDuplicateRemover，需要单独引入Guava依赖包。

### 6.3 使用和定制Downloader

WebMagic的默认Downloader基于HttpClient。一般来说，你无须自己实现Downloader，不过HttpClientDownloader也预留了几个扩展点，以满足不同场景的需求。

另外，你可能希望通过其他方式来实现页面下载，例如使用SeleniumDownloader来渲染动态页面。