**可配置化网络爬虫项目构建说明书**

**小组名称：**可配置化网络平台小组

**小组口号：**我们爱学习

**指导教师：**陈博老师

**文档撰写人：**吴祥龙 关玉婷

**文档撰写时间：**2017年5月20日

团队分工记录表

|  |  |  |  |
| --- | --- | --- | --- |
| **项目名称** | **学号** | **姓名** | **分工** |
| 可配置化网络爬虫 | SA17225405 | 吴祥龙 | 项目构建 |
| SA17225094 | 关玉婷 | 项目构建 |
| SA17225362 | 王润 | 项目构建 |
| SA17225362 | 唐冬鑫 | 项目构建 |

目录

[1 具体实现 2](#_Toc130)

[1.1 重要的数据结构的介绍 2](#_Toc12544)

[1.1 SpiderState介绍 2](#_Toc26501)

[1.2 SpiderModel介绍 3](#_Toc18923)

[1.3 WorkerZkMessage介绍 3](#_Toc14092)

[2 RM的实现 4](#_Toc27)

[2.1 RM的thrift服务定义 4](#_Toc19273)

[2.2 worker的发现与worker宕机的处理 4](#_Toc12194)

[2.3 RM资源分配过程的实现 5](#_Toc17948)

[3 Worker的实现 6](#_Toc14477)

[3.1 worker的thrift服务的定义 6](#_Toc32330)

[3.2 worker在ZK的注册 6](#_Toc22995)

[3.3 Worker对爬虫任务的管理 6](#_Toc29994)

[4 Spider的实现 8](#_Toc15552)

[4.1 扩展PageProcessor 8](#_Toc28384)

[4.2 重写Scheduler 8](#_Toc31944)

[4.3 重写Spider 8](#_Toc26985)

[5 客户端的实现 9](#_Toc5863)

[6 本章小结 10](#_Toc20735)

# 1 具体实现

## 1.1 重要的数据结构的介绍

### 1.1 SpiderState介绍

SpiderState是对爬虫节点的状态的枚举，下面是SpiderState在thrift中的定义，allocate代表这个爬虫的资源被分配，这个状态代表爬虫资源已经分配好了，但是爬虫节点还未开始运行、running代表这个爬虫正在运行中，wait代表这个爬虫正在等待其它爬虫将网页链接写到队列里，stop代表爬虫正常结束，dead代表爬虫异常结束。

**enum** SpiderState{  
 allocate,  
 running,  
 wait,  
 stop,  
 dead  
}

### 1.2 SpiderModel介绍

/\*\*  
\*  
\* SpiderModel 是对启动爬虫参数的抽象  
\*\*/  
**struct** SpiderModel{  
 1: **required string** classPath;  
 2: **required binary** classData;  
 3: **required i32** taskId;  
 4: **required i32** nodeNumbers;  
 5:**required i32** maxMemoryPerNode;  
}

SpiderModel是启动爬虫的一个必要参数。ClassPath是爬虫逻辑的实现的类的路径、classData是爬虫逻辑的实现的类的一个对象、taskId是爬虫的id号，每个爬虫都有一个唯一的id号与之对应、nodeNumbers是这个爬虫需要运行的爬虫节点数目、maxMemoryPerNode是每个节点分配的内存数目，单位是字节

### 1.3 WorkerZkMessage介绍

*/\*\*  
 \*  
 \* 每个Worker在zk里注册的信息  
 \** ***@author*** *xuxue 2017-5-2  
 \*/*public class WorkerZkMessage {  
  
 //worker的名字  
 private String name;  
   
 //worker的地址  
 private String address;  
   
 //worker允许分配的最大内存  
 private long allocateMemory;  
   
 //worker服务的端口  
 private int port;  
   
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getAddress() {  
 return address;  
 }  
  
 public void setAddress(String address) {  
 this.address = address;  
 }  
  
 public long getAllocateMemory() {  
 return allocateMemory;  
 }  
  
 public void setAllocateMemory(long allocateMemory) {  
 this.allocateMemory = allocateMemory;  
 }  
  
 public int getPort() {  
 return port;  
 }  
  
 public void setPort(int port) {  
 this.port = port;  
 }  
}

WorkerZkMessage是每个Worker节点在ZK集群里注册时写入的信息，name是worker的名字、address是worker的地址，allocateMemory是这个worker最大分配的内存数量单位是字节，port是这个Worker服务所在的端口。

## 2 RM的实现

RM是整个爬虫集群的控制中心，RM的本质是一个thrift服务，下面详细介绍RM是如何实现的。

### 2.1 RM的thrift服务定义

1. /\*\*  
   \* ResourceManager 是整个Spider集群的资源管理器 负责为spider的运行分配资源  
   \* 并监控每个Spider的运行状态  
   \* 1.启动Spider  
   \* 2.监控Spider的运行状态  
   \* 3.为Spider动态的加减节点  
   \* 4.停止Spider  
   \*\*/  
   **service** ResourceManagerService{  
    **void** startSpider(1:SpiderModel application);//启动爬虫  
    **void** addNode(1:**i32** taskId); //增加爬虫的节点  
    **void** reduceNode(1:**i32** taskId);//减少爬虫的节点  
    **void** killSpider(1:**i32** taskId);//杀死爬虫  
    **void** reportSpiderStateChange(1:**i32** taskId,  
    2:**i32** subId,  
    3:SpiderState oldState,  
    4:SpiderState newState);//上报爬虫的状态  
    **string** getSpiderNodesState(1:**i32** taskId);//获取一个爬虫节点的状态  
   }

### 2.2 worker的发现与worker宕机的处理

**val** zkClient = **new** ZkClient(rMConfig.*zkConnectString*,5000)  
zkClient.setZkSerializer(**new** ZkSerializer {//设置zkClient的解码编码器  
 **override def** serialize(data: scala.Any): Array[Byte] = data.asInstanceOf[String].getBytes("utf-8")  
  
 **override def** deserialize(bytes: Array[Byte]): AnyRef = **new** String(bytes)  
})  
zkClient.subscribeChildChanges("/spider",(parentPath:String,currentChilds:java.util.List[String])=>{  
 **if**(currentChilds.size()< *workers*.keySet().size()){ //如果worker的数目减少  
 **val** name = findReduceWorkerName(parentPath,currentChilds).getOrElse("")  
 **this** synchronized{  
 **val** worker = *workers*.remove(name)//将增加的死去的worker找到  
 **if**(worker != **null**){//如果死去的worker存在  
 worker.list().forEach((container)=>{//找到这个worker里运行的spider  
 **val** taskId = container.asInstanceOf[ResourceManagerContainer].taskId  
 **val** subId = container.asInstanceOf[ResourceManagerContainer].subId  
 **val** runSpiders = *runSpidersMap*.get(taskId)  
 **if**(runSpiders!=**null**){  
 runSpiders.reAllocate(subId)//重新启动  
 }  
 })  
 }  
 }  
 }**else**{ //如果worker的数目增加  
 **this** synchronized{  
 **val** name = findAddWorkerName(parentPath,currentChilds).getOrElse("")  
 **if**(!*workers*.contains(name)) {//把worker加入资源列表  
 **val** message: String = zkClient.readData(parentPath + "/" + name)  
 **val** workerZkMessage = G.*gson*.fromJson(message, *classOf*[WorkerZkMessage])  
 **val** resourceManagerWorker = **new** ResourceManagerWorker(workerZkMessage)  
 *workers*.put(name,resourceManagerWorker)  
 }  
 }  
 }  
})  
zkClient

RM通过监控Zk的/spider节点的子节点增加来发现worker的增加，当worker启动后worker会在/spider下面创建一个名字是worker的名字的临时节点并将这个worker的信息写入这个临时节点。RM通过监控/spider的子节点的变化来获取worker的增加事件与worker的宕机事件。当worker宕机后RM会将之前分配在这个worker的任务重新启动，分配到其它的Wroker中，当新增加worker时RM会读取这个worker的配置信息，放到列表里。

### 2.3 RM资源分配过程的实现

在分配之前RM会根据Worker剩余的内存大小判断出哪个Worker最适合分配，如果最适合分配的Worker的可用内存大于需要分配的内存，则返回一个存放运行爬虫节点的容器的option分配成功，否者返回一个空option分配失败

**def** maxFreeMemory():ResourceManagerWorker={  
 **this** synchronized{  
 **val** allWorker = workers.values()  
 **var** maxMemoryWorker:ResourceManagerWorker = **null  
 for**(worker <- allWorker){  
 **if**(maxMemoryWorker ==**null** || maxMemoryWorker.availableMemory<worker.availableMemory){  
 maxMemoryWorker = worker  
 }  
 }  
 maxMemoryWorker  
 }  
}

**def** allocate(taskId:Int,subId:Int,memory:Long):Option[Container]={  
 **this** synchronized{  
 **val** maxMemoryWorker =maxFreeMemory()  
 **if**(maxMemoryWorker.availableMemory<memory) **return** Option.empty  
 **else** Option(maxMemoryWorker.allocate(taskId,subId,memory))  
 }  
}

## 3 Worker的实现

Worker主要负责启动爬虫，管理运行在这个Worker上的爬虫节点并向Worker上报节点的状态变化。下面详细介绍Worker的实现

### 3.1 worker的thrift服务的定义

Worker本质是一个thrift服务，下面是这个thrift服务定义的一系列方法

/\*\*  
\* Worker  
\* 1.接收ResourceManager的信号 启动Spider  
\* 2.接收每个Spider上报的状态(每个Spider会隔一段时间向woker上报状态)  
\* 3.接收ResourceManager 的信号 杀死Spider  
\* 4.检测Spider的内存使用情况 如果超出分配的值 则杀死Spider  
\*\*/  
**service** WorkerService{  
 **void** startSpider(1:**binary** classData,2:**i64** allocateMemory,3:**i32** taskId,4:**i32** subId);  
 **void** killSpider(1:**i32** taskId,2:**i32** subId);  
 **void** reportSpiderState(1:**i32** taskId,2:**i32** subId,3:SpiderState state);  
 **list**<SpiderNodeState> getSpiderNodesState(1:**i32** taskId);  
}

startSpider方法由RM调用返回启动一个爬虫节点，killSpider方法也是由Spider调用杀死一个爬虫节点，reportSpiderState方法在Spider向Worker汇报状态时调用，getSpiderNodesState由Spider调用获取爬虫的状态信息。

### 3.2 worker在ZK的注册

**def** register(workerConfig: WorkerConfig,zkCli:ZkClient):Unit={  
 **val** workerZkMessage = **new** WorkerZkMessage  
 workerZkMessage.setPort(workerConfig.*port*)  
 workerZkMessage.setName(workerConfig.*name*)  
 workerZkMessage.setAddress(workerConfig.*ip*)  
 workerZkMessage.setAllocateMemory(workerConfig.*configMemory*)  
 zkCli.create("/spider",G.*gson*.toJson(workerZkMessage),CreateMode.*EPHEMERAL*)  
}

worker在初始化完毕后，会向ZK集群注册一个临时节点，把自身的配置信息写到这个节点里，以上是具体实现的代码

### 3.3 Worker对爬虫任务的管理

每个Woker都管理着运行在这个worker上的爬虫节点，运行在这个worker上的爬虫节点被抽象成一个NodeManagerContainer对象。Wroker通过对NodeManagerContainer对象的管理来实现对运行在这个Worker上的爬虫节点的管理。

*/\*\*  
 \* Worker上运行的爬虫节点的抽象  
 \** ***@author*** *xuxue 2017-5-5  
 \*/***class** NodeManagerContainer(workerConfig: WorkerConfig,**val** allocateMemory:Long,**val** taskId:Int,**val** subId:Int) **extends** Container{  
 */\*\*  
 \* 运行的进程  
 \*/* **var** *process*:Process = **null** */\*\*  
 \* 爬虫的状态  
 \*/* **var** *spiderState*:SpiderState = SpiderState.*allocate  
  
 /\*\*  
 \* 与RM的连接  
 \*/* **val** *resourceManagerClient*:ResourceManagerClient = **new** ResourceManagerClient(workerConfig.*resourceManagerHost*,workerConfig.*resourceManagerPort*)  
 */\*\*  
 \* 这个爬虫进程的pid  
 \*/* **var** *pid*:Long=0  
  
 */\*\*  
 \* 启动爬虫  
 \** ***@param model*** *启动的参数  
 \*/* **override def** runSpider(model: SpiderModel): Unit = {  
 **val** bytes = model.*classData*.array()  
 *use*(**new** FileOutputStream("spider.model")){  
 stream=>  
 stream.write(bytes,0,bytes.length)  
 }  
 *process* = Runtime.*getRuntime*.exec("nohoup java -jar "+workerConfig.*spiderJarFile*)  
 *pid*=SystemUtil.*getPidFromProcess*(*process*)  
 }  
  
 */\*\*  
 \* 杀死爬虫进程  
 \*/* **override def** kill(): Unit = {  
 *process*.destroy()  
 }  
  
 */\*\*  
 \* 这个爬虫进程的状态更改  
 \** ***@param spiderState*** *更改后的状态  
 \*/* **def** changeState(spiderState: SpiderState): Unit ={  
 **if**(spiderState!=**this**.*spiderState*){  
 *resourceManagerClient*.reciveSpiderStateChange(taskId,subId,**this**.*spiderState*,spiderState)  
 **this**.*spiderState* = spiderState  
 }  
 }  
  
 **override def** getState: SpiderState = *spiderState* **override def** setSpiderNodeState(spiderNodeState: SpiderState): Unit = changeState(*spiderState*)  
}

## 4 Spider的实现

本系统的爬虫模块主要是对webmagic爬虫框架的扩展。webmagic不支持分布式爬虫，本系统重写了Spider与Scheduler模块使这个爬虫支持分布式。

### 4.1 扩展PageProcessor

*/\*\*  
 \*  
 \** ***@author*** *xuxue on 2017-5-4  
 \*/***trait** CrawlerProcessor **extends** PageProcessor **with** Serializable{  
 **def** initCrawler(distributedSpider: DistributedSpider):Unit  
}

因为爬虫是多节点在运行的，有些初始化操作只希望一个节点执行，其它节点在这个节点的初始化操作完成后才能运行。CrawlerProcessor的initCrawler方法就是应对这一场景设计出来的。

### 4.2 重写Scheduler

WebMagic默认的Scheduler是用java的list实现的，这显然不适用于多节点爬虫的场景，本系统对Scheduler重写，利用redis缓存实现了一个适用与多节点分布式场景的Scheduler

*/\*\*  
 \** ***@author*** *xuxue on 2017/4/29.  
 \*/***class** CrawlerScheduler(**val** spiderConfig: SpiderConfig,**val** taskId:Int) **extends** DuplicateRemovedScheduler{  
  
 **val** *redis* =**new** Jedis(spiderConfig.*redisHost*,spiderConfig.*redisPort*)  
  
 */\*\*  
 \* get an url to crawl  
 \*  
 \** ***@param task*** *the task of spider  
 \** ***@return*** *the url to crawl  
 \*/* **override def** poll(task: Task): Request = {  
 **val** value=*redis*.lpop(""+taskId)  
 **if**(value == **null**) **null  
 else** G.*gson*.fromJson(value,*classOf*[Request])  
 }  
  
 **override protected def** pushWhenNoDuplicate(request: Request, task: Task): Unit = {  
 *redis*.rpush(""+taskId,G.*gson*.toJson(request))  
 }  
  
}

### 4.3 重写Spider

Spider是WebMagic框架的核心类，它负责创建Spider。本系统爬虫运行的环境是多节点分布式的环境，WebMagic框架的Spider类不适用于这种场景。本系统增加了一种Spider类型DistributeSpider。DistributeSpider与Spider的主要不同之处在与执行的过程下面代码实现了DistributeSpider的执行流程

**override def** run(): Unit = {  
 initComponent() //各个组件的初始化  
 **if**(subId == 1){ //只有sub id 为1的节点执行 initCrawler 方法  
 *crawlerProcessor*.initCrawler(**this**)  
 }**else**{ //等待subid为1的节点初始化完毕  
 waitMasterNodeInit()  
 }  
 logInfo(getUUID+" initCrawler success!")  
 *stat* = SpiderState.*running* //上报节点状态变化  
 *workerClient*.reportSpiderState(spiderModel.*taskId*,subId,SpiderState.*running*)  
 *reportThread*.start() //启动上报状态的线程  
 **while**(!Thread.*currentThread*().isInterrupted &&   
 *stat*!= SpiderState.*dead* && *stat*!=SpiderState.*stop*){//循环从队列里去网页链接并下载  
 **val** request = *scheduler*.poll(**this**)  
 **if**(request !=**null**) {//如果取出的元素不为空  
 *threadPool*.execute(()=>{  
 **try** {  
 processRequest(request)  
 onSuccess(request)  
 } **catch** {  
 **case** e: Exception =>  
 onError(request)  
 logError("process request " + request + " error", e)  
 } **finally** {  
 *pageCount*.incrementAndGet  
 signalNewUrl()  
 }  
 })  
 }**else**{  
 **if**(haveRunningInSelf()){//如果为空，并且本节点有其它任务在运行  
 waitNewUrl()  
 }**else if**(haveOtherNodeRunning()){//如果为空，有其它节点在运行  
 *stat* = SpiderState.*wait* //更改状态  
 waitNewUrl()  
 }**else**{//否者  
 *stat* = SpiderState.*stop* }  
 }  
 }  
 //如果跳出循环 爬虫运行结束  
 *reportThread*.interrupt()  
 System.*exit*(0)  
}

## 5 客户端的实现

客户端主要是对RM的访问，远程调用RM的方法实现对整个集群资源的访问，这个客户端其实是一个访问thrift服务的客户端，这使得其他语言很方便的调用RM。本系统用java实现了一个命令行的客户端，方便集群的管理。这个客户端主要运用了java的反射机制。主要代码如下

**class** Client{  
  
 **val** *clientConf* = **new** ClientConf("conf/client.conf")  
  
 **var** *client*:ResourceManagerClient = **new** ResourceManagerClient(*clientConf*.*default\_server\_IP*,*clientConf*.*default\_server\_port*)  
  
  
 **def** list(args:Array[String]):Unit={  
 **if**(args.length == 1) *client*.listSpider()  
 }  
  
  **。**

**。**

**。**  
  
 **def** listWorker(args:Array[String]):Unit={  
 *println*(*client*.listWorkers())  
 }  
  
 **def** test(args:Array[String]):Unit=args.map(System.*out*.println)  
  
 **def** close():Unit=*client*.close()  
}  
  
**object** Client{  
  
 **def** main(args: Array[String]): Unit = {  
 **val** methodName = args(0)  
 **val** clientClass = *classOf*[Client]  
 **val** client = **new** Client()  
 **val** method = clientClass.getMethod(methodName,*classOf*[Array[String]])  
 method.invoke(client,args)  
 client.close()  
 }  
  
}

客户端通过命令行传入调用的方法，第一个参数是方法名，剩下的参数是调用这个方法的参数，通过反射机制，实现对方法的动态调用。

## 6 本章小结

本章详细节介绍了这篇文章实现的分布式的爬虫系统的一些实现细节。本章先介绍了分布式爬虫系统里用到的一些重要的数据结构，然后分别介绍了RM、Worker、Spider的实现细节