该文档为爬虫基于Redis调度方式的架构版本

若有任何疑问可以邮件我

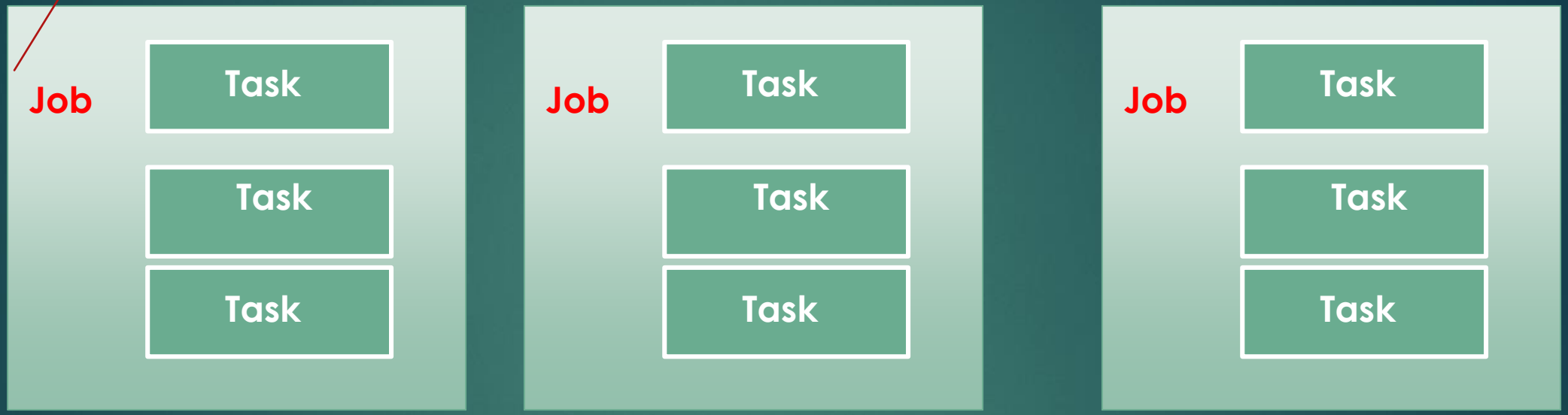
wengbenjue@163.com

- 分布式爬虫系统构架
- Zookeeper
- 爬虫模块逻辑
- 资源管理器
- 爬虫解析引擎
- 更新爬虫爬取周期
- 反被封
- 爬虫类图
- 日志记录模块
- 美团外卖爬取（二次开发方式）
- 任务层次批次
- Job管理
- Schema抽取
- 任务跟踪系统
- 监控系统

分布式爬虫系统构架

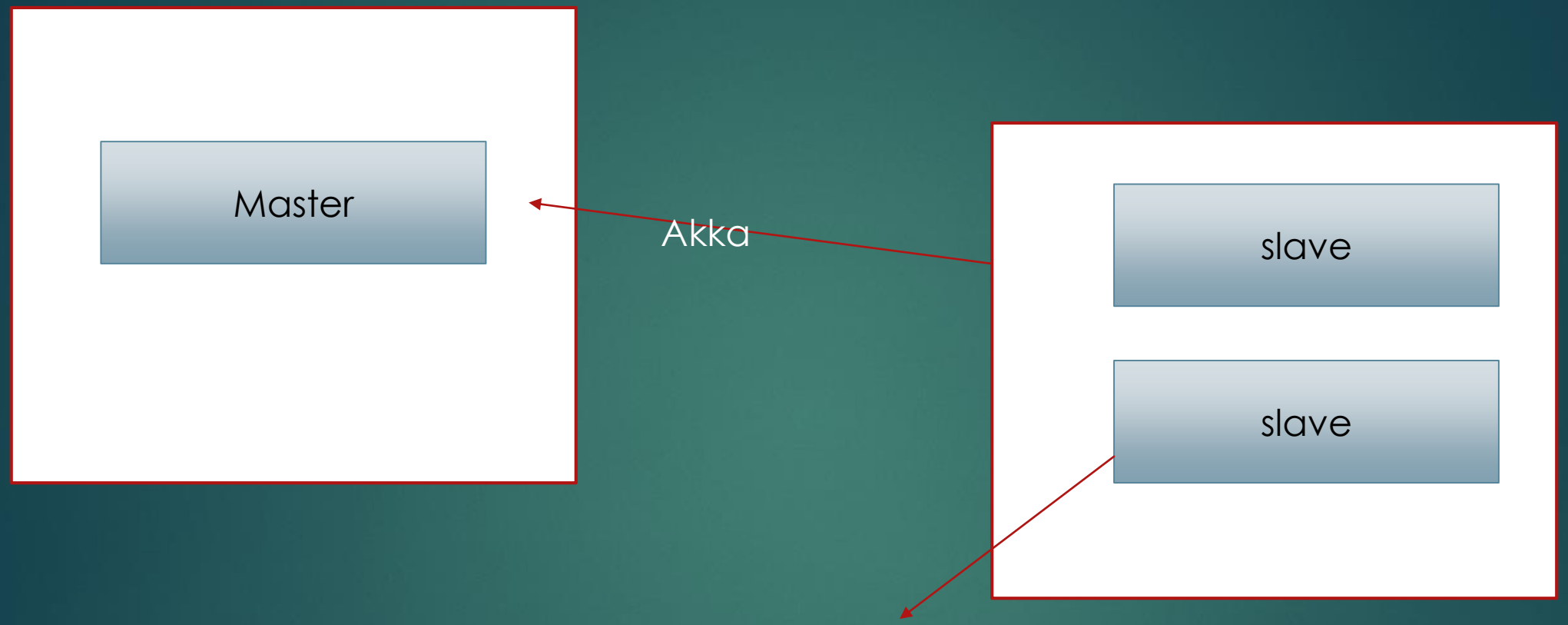
Job、Task

URL过滤规则接口



通用爬虫　　主题爬虫　　指定页爬虫

规则过滤接口拓扑



对于每个job只需要第一次去Master取规则，然后自己做缓存，当Job退出的时候，首先清掉本机对应Jobid缓存，然后告诉master完成，清理掉缓存

种子入队列

解耦

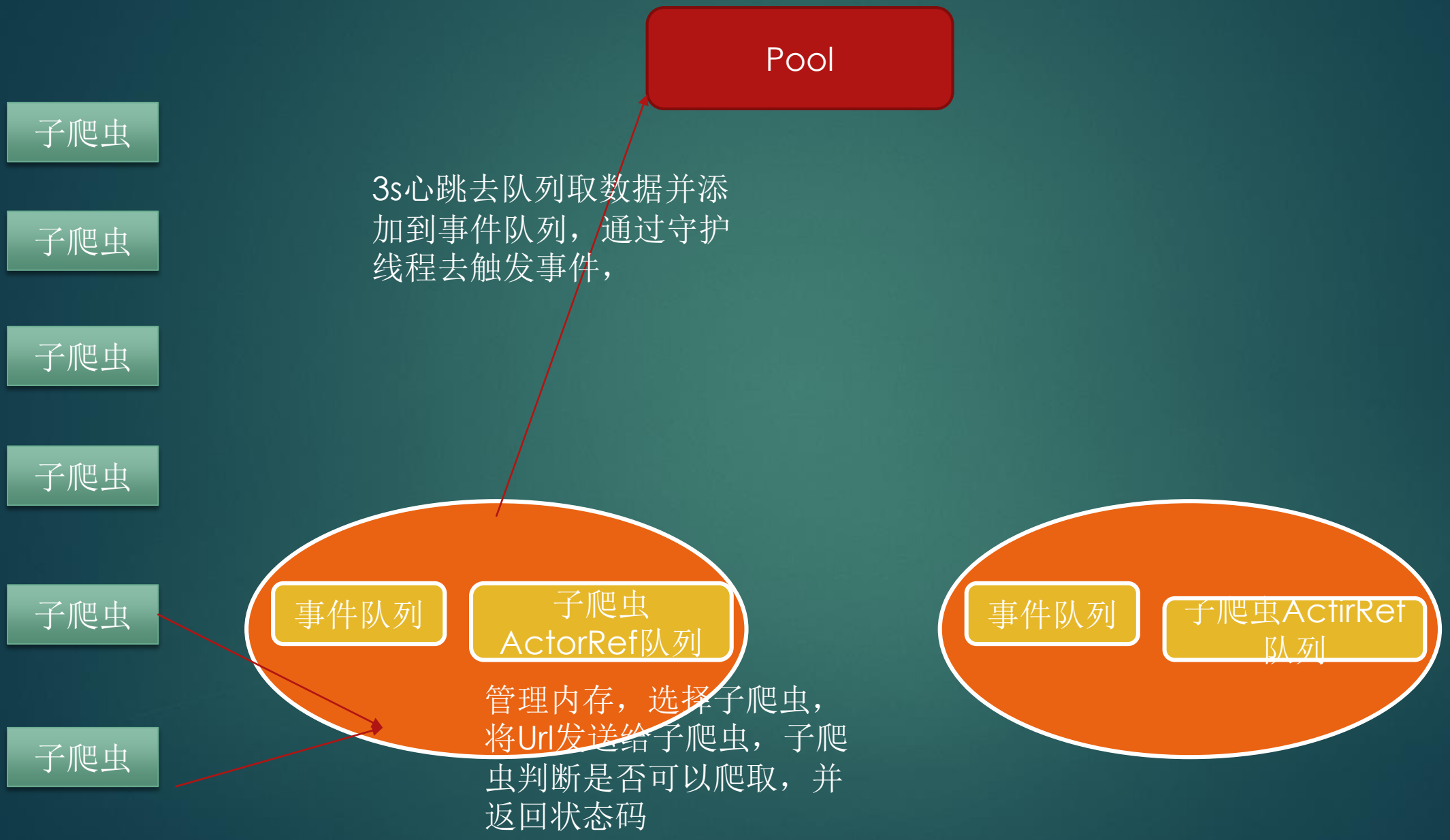
需安装Redis

如果到时间点了将任务移到任务队列中，如果是主题爬虫不取出，将时间加上定期爬取时间

爬虫从队列领任务爬取

解耦

爬虫子爬虫架构


Pool
子爬虫
子爬虫
子爬虫
子爬虫
子爬虫
子爬虫
3s心跳去队列取数据并添加到事件队列，通过守护线程去触发事件，
事件队列
子爬虫ActorRef队列
管理内存，选择子爬虫，将Url发送给子爬虫，子爬虫判断是否可以爬取，并返回状态码
事件队列
子爬虫ActirRef队列

子爬虫日志记录及爬取成功后操作

子爬虫

子爬虫

子爬虫

子爬虫

子爬虫

子爬虫

解耦

将上次爬取时间，被封次数，爬取站域名次数，同一网页上次爬取时间等，写入文件或者Hbase等，方便通过统计分析，然后通过机器学习建模，Spark+Streaming或者Hive+Spark+Storm

将爬取后原始数据存入Hbase，因内网，提供服务接口

然后将存好后的key放入Kafka

需安装Kafka依赖于zookeeper,最好单独安装zookeeper

机器学习建模的两个应用目标

子爬虫用于判断是否能接受给予的url,进行爬取

优先级队列中，score的建模排序

解析前准备

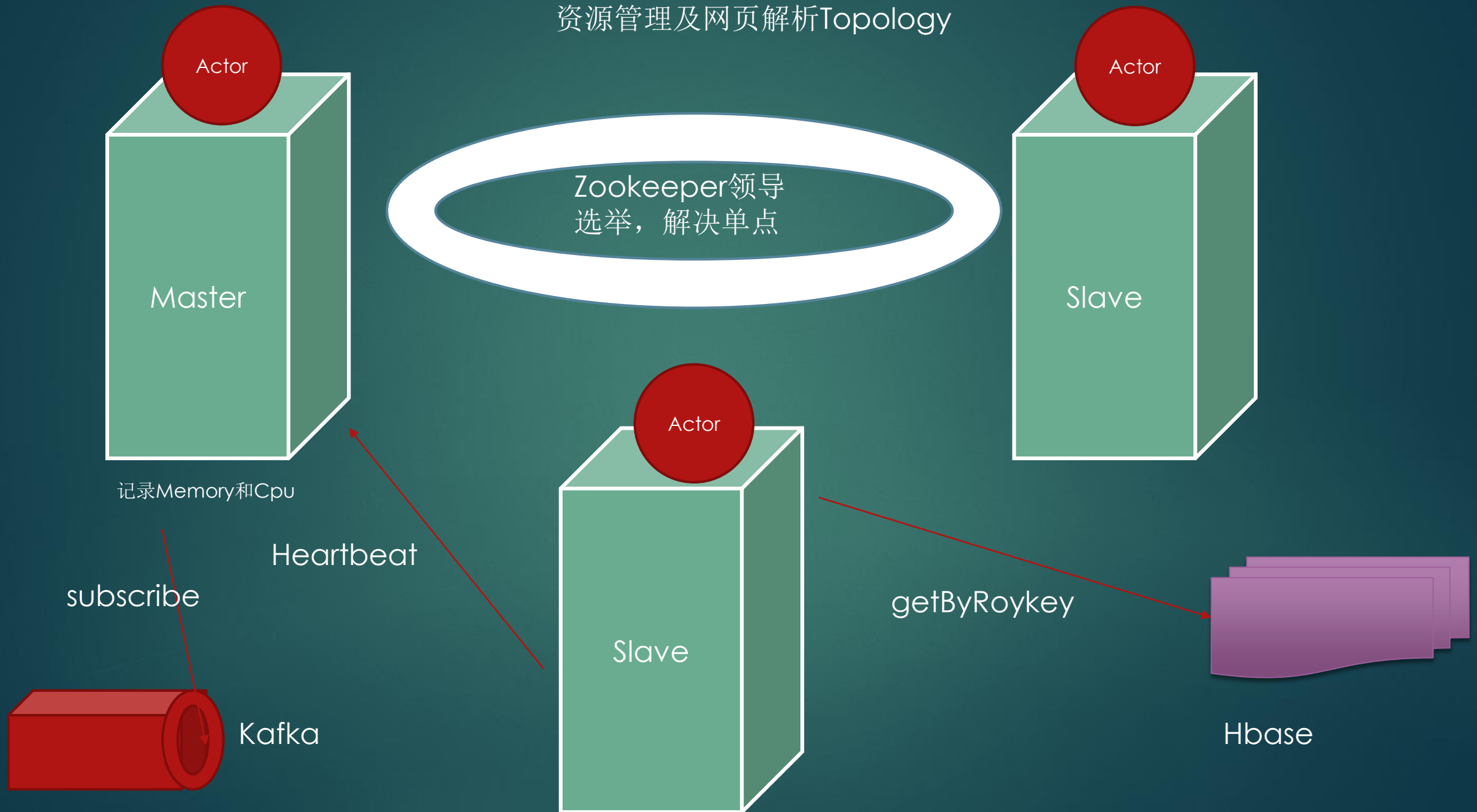
一、判断自己的资源，如内存、CPU是否够用。（通过主从分布式资源管理）

二、如果够用去Kafka拉取一条rowkey

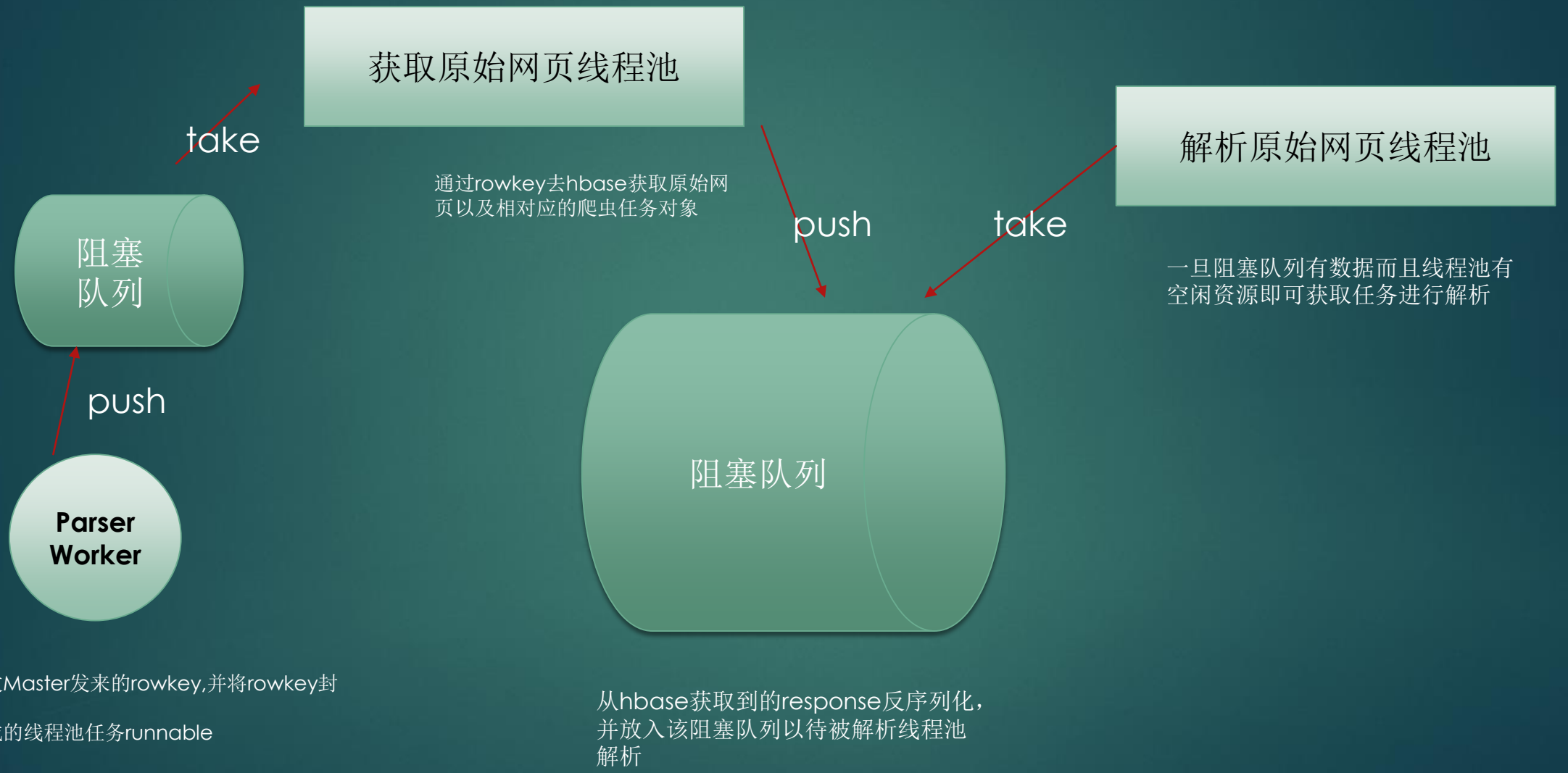
三、根据rowkey去hbase拉取该原始网页进行解析

解耦

资源管理及网页解析Topology



解析线程池逻辑结构


获取原始网页线程池
take
通过rowkey去hbase获取原始网页以及相对应的爬虫任务对象
push
take
解析原始网页线程池
一旦阻塞队列有数据而且线程池有空闲资源即可获取任务进行解析
阻塞队列
push
阻塞队列
Parser Worker
接收Master发来的rowkey,并将rowkey封装成的线程池任务runnable
从hbase获取到的response反序列化，并放入该阻塞队列以待被解析线程池解析

解析网页

解析的url重新入队列

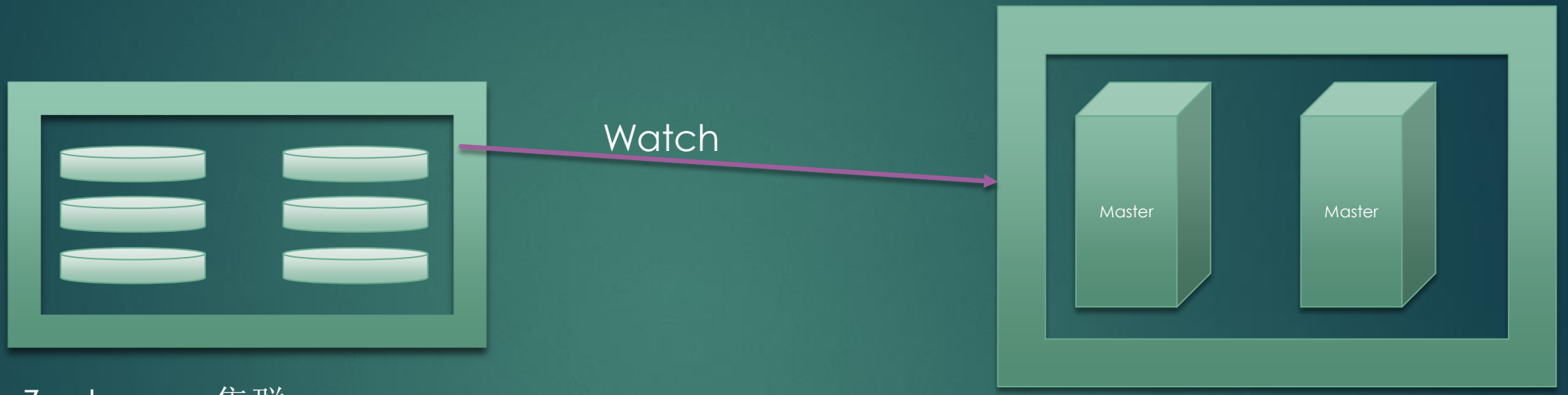
解析的网页内容进入MongDB

解析可以用Xpath，解析的架构和技术需要花时间梳理下

解耦

Zookeeper

Zookeeper Leadership election 图解

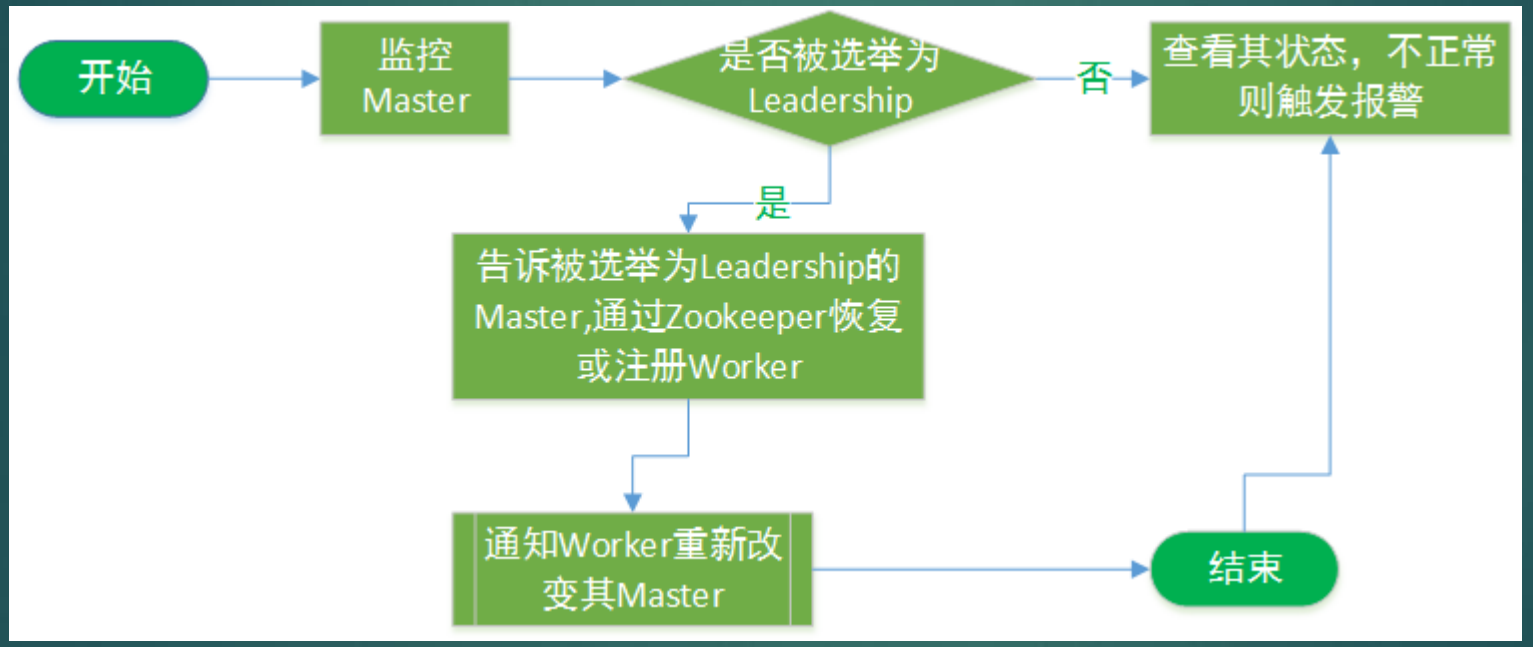


Zookeeper集群

Master,主备

Zookeeper Leadership election 图解

领导选举流程图



Zookeeper

ZooKeeperLeaderElectionAgent

```
private[spark] class ZooKeeperLeaderElectionAgent(val masterActor:
ActorRef,
    masterUrl: String, conf: SparkConf)
  extends LeaderElectionAgent with LeaderLatchListener with
Logging  {

  val WORKING_DIR = conf.get("spark.deploy.zookeeper.dir",
"/spark") + "/leader_election"

  private var zk: CuratorFramework = _
  private var leaderLatch: LeaderLatch = _
  private var status = LeadershipStatus.NOT_LEADER

  override def preStart() {

    logInfo("Starting ZooKeeper LeaderElection agent")
    zk = SparkCuratorUtil.newClient(conf)
    leaderLatch = new LeaderLatch(zk, WORKING_DIR)
    leaderLatch.addListener(this)

    leaderLatch.start()
  }

  override def preRestart(reason: scala.Throwable, message:
scala.Option[scala.Any]) {
    logError("LeaderElectionAgent failed...", reason)
    super.preRestart(reason, message)
  }

  override def postStop() {
    leaderLatch.close()
    zk.close()
  }
```

LeaderElectionAgent

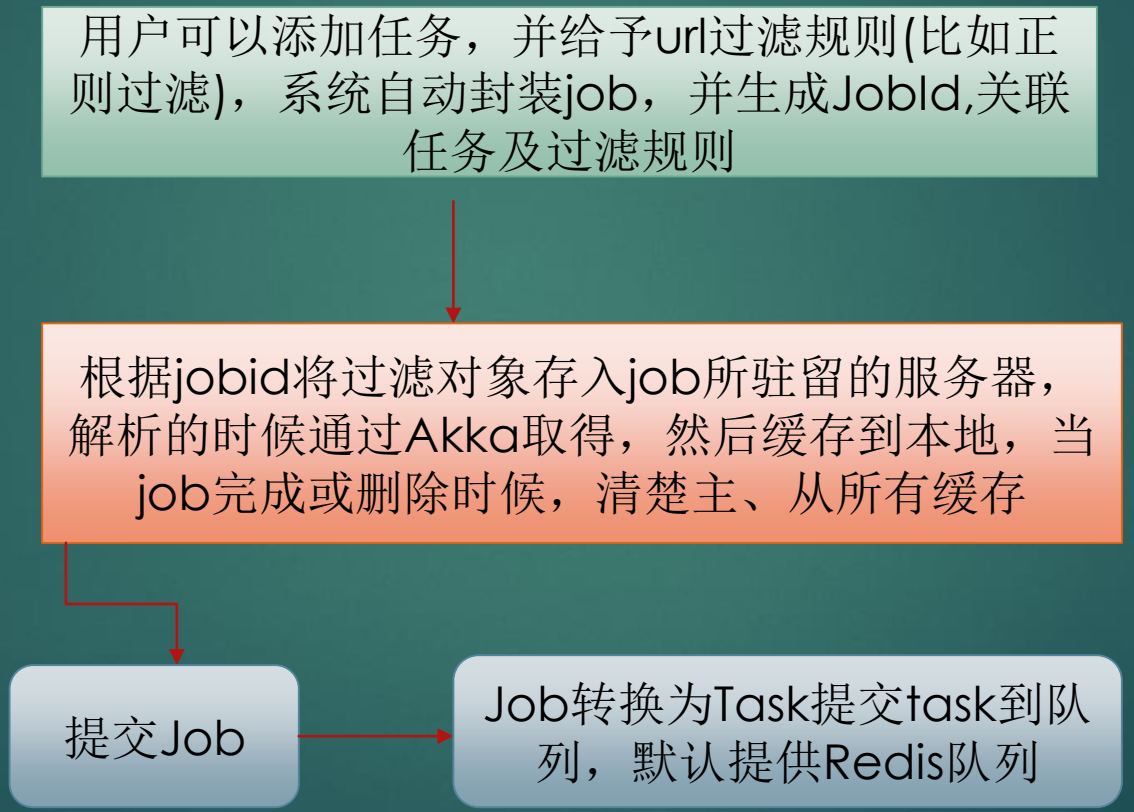
Zookeeper

```
/**
 * A LeaderElectionAgent keeps track of whether the current Master is the leader, meaning it
 * is the only Master serving requests.
 * In addition to the API provided, the LeaderElectionAgent will use of the following messages
 * to inform the Master of leader changes:
 * [[org.apache.spark.deploy.master.MasterMessages.ElectedLeader ElectedLeader]]
 * [[org.apache.spark.deploy.master.MasterMessages.RevokedLeadership RevokedLeadership]]
 */
private[spark] trait LeaderElectionAgent extends Actor {
  // TODO: LeaderElectionAgent does not necessary to be an Actor anymore, need refactoring.
  val masterActor: ActorRef
}

/** Single-node implementation of LeaderElectionAgent -- we're initially and always the leader. */
private[spark] class MonarchyLeaderAgent(val masterActor: ActorRef) extends LeaderElectionAgent {
  override def preStart() {
    masterActor ! ElectedLeader
  }

  override def receive = {
    case _ =>
  }
}
```

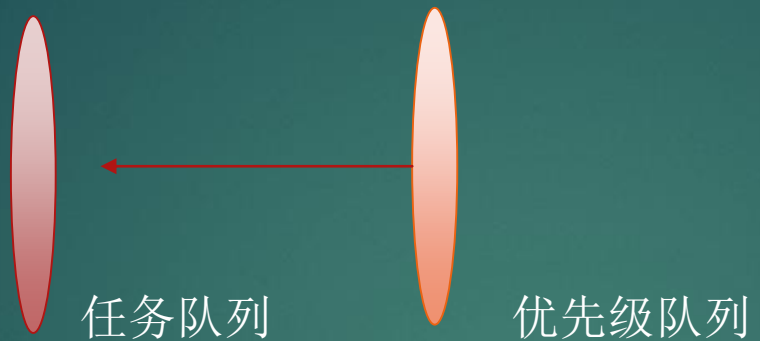
爬虫模块逻辑

爬虫入口逻辑

入口可以通过Web或者命令行进入



爬虫任务队列转移逻辑



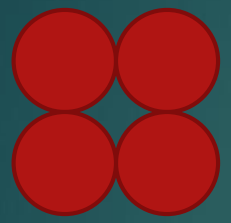
优先级队列，按执行时间排序，时间早的先执行
判断爬虫类型，对于主题爬虫，判断爬取时间与现在时间的时差，如果过了或刚好为当前时间就取出来放入任务队列，但是不需要删除，只需要改变时间为下次执行时间，方便下次继续执行；【使用双扫线程】

爬虫到子爬虫逻辑

1.爬虫从redis取得任务并放入本地Blocking Queue中[使用双扫线程]
2.使用死循环从blocking queue中取得任务并通过子爬虫传过来的资源状况（如内存，cpu，以及等待任务数，空闲线程数等）进行判断调度
3.将任务进行序列化并通过Akka将任务传给对应的子爬虫

另外
1.子爬虫通过心跳将自身资源传给爬虫，同时爬虫通过这个来判断子爬虫是否活着
2.爬虫集群的单点问题通过zookeeper
3.资源管理抽象出来

子爬虫逻辑



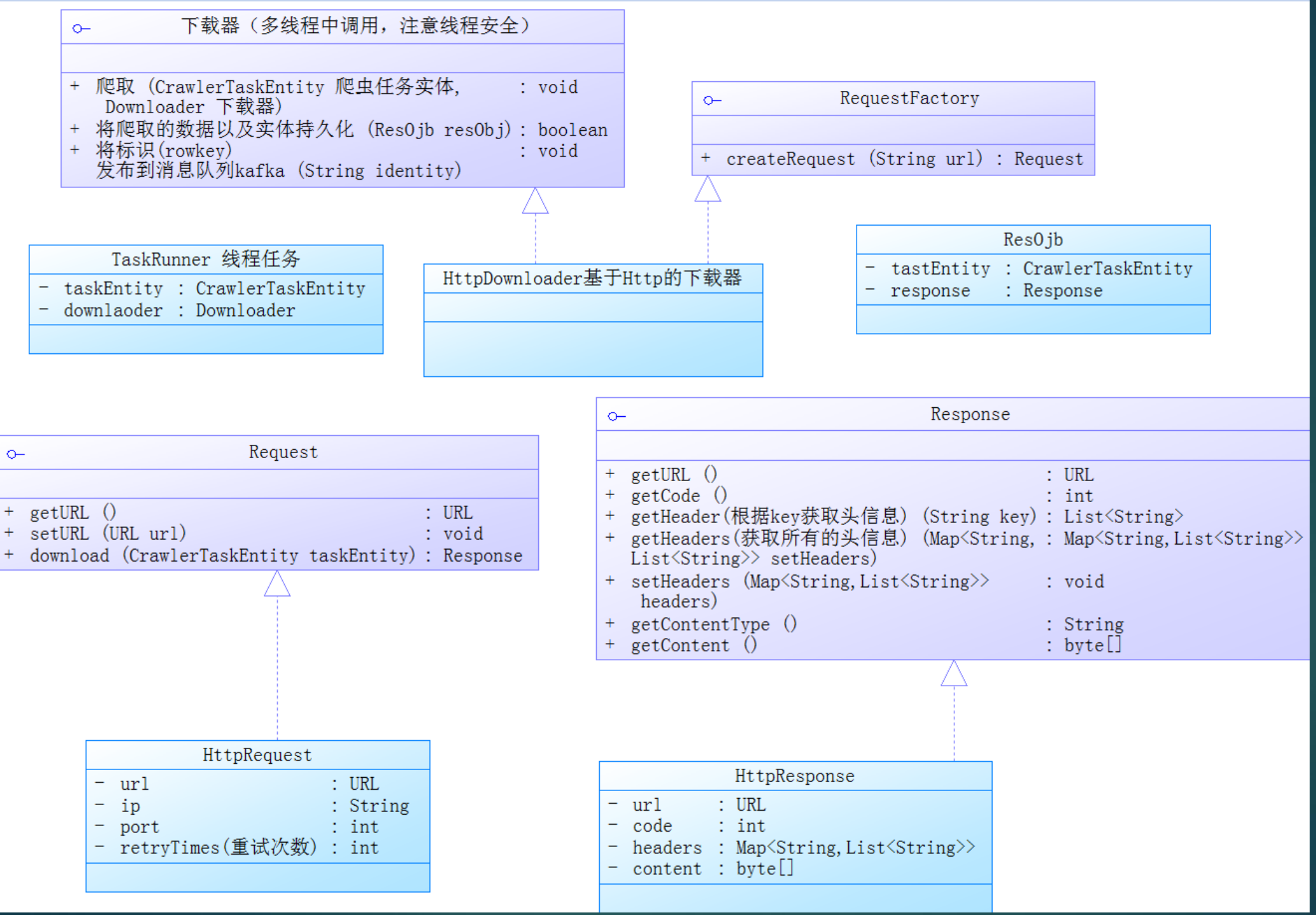
子爬虫
【Akka Actor 】

Executor

1.接收到爬虫传过来的uri后将uri进行反序列化操作，然后封装成TaskRunner传给线程池（注意线程安全问题）
线程池设置为守护线程，一直执行

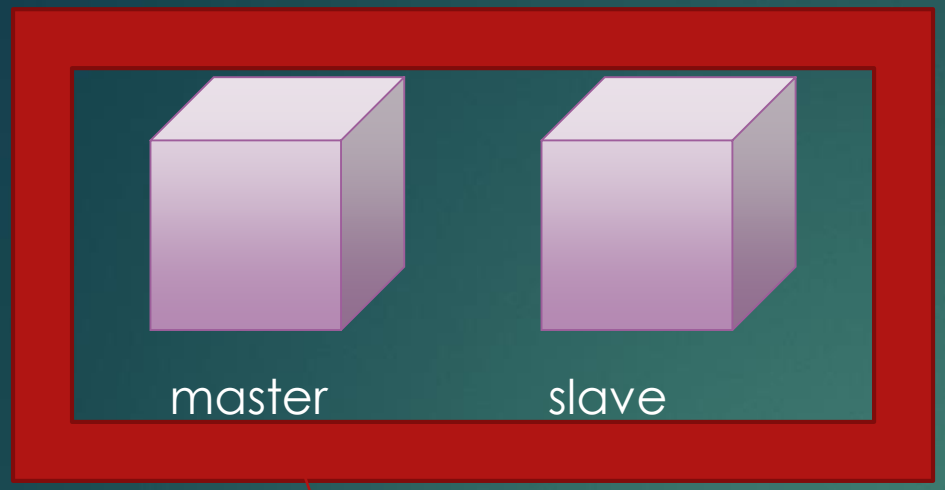
2.线程池内部，即TaskRunner将uri交给下载器，下载器的初始化在TaskRUnner中进行

3.下载器实现类（比如Http下载器使用HttpRequest下载网页，并将返回信息封装为HttpResponse）下载完后将实体类和Response封装为为ResOjb序列化并放入Hbase中，同时将rowkery放入kafka中

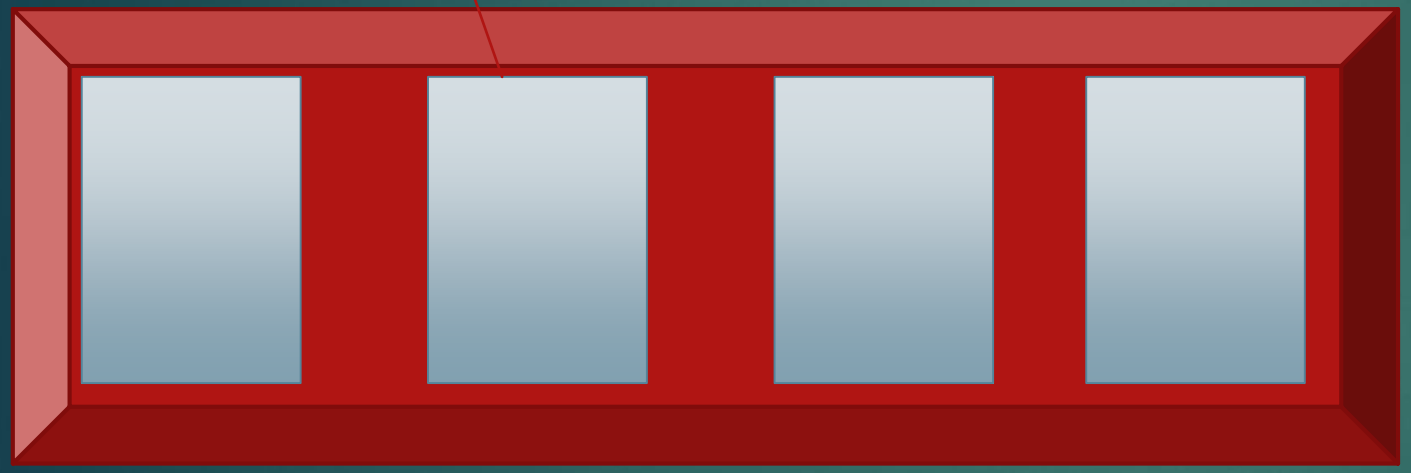

下载器（多线程中调用，注意线程安全）
+ 爬取 (CrawlerTaskEntity 爬虫任务实体, Downloader 下载器) : void
+ 将爬取的数据以及实体持久化 (ResObj resObj) : boolean
+ 将标识(rowkey)发布到消息队列kafka (String identity) : void
RequestFactory
+ createRequest (String url) : Request
TaskRunner 线程任务
- taskEntity : CrawlerTaskEntity
- downlaoder : Downloader
HttpDownloader基于Http的下载器
ResObj
- tastEntity : CrawlerTaskEntity
- response : Response
Request
+ getURL () : URL
+ setURL (URL url) : void
+ download (CrawlerTaskEntity taskEntity) : Response
Response
+ getURL () : URL
+ getCode () : int
+ getHeader(根据key获取头信息) (String key) : List<String>
+ getHeaders(获取所有的头信息) (Map<String, List<String>> setHeaders) : Map<String,List<String>>
+ setHeaders (Map<String,List<String>> headers) : void
+ getContentType () : String
+ getContent () : byte[]
HttpRequest
- url : URL
- ip : String
- port : int
- retryTimes(重试次数) : int
HttpResponse
- url : URL
- code : int
- headers : Map<String,List<String>>
- content : byte[]

资源管理器

资源管理



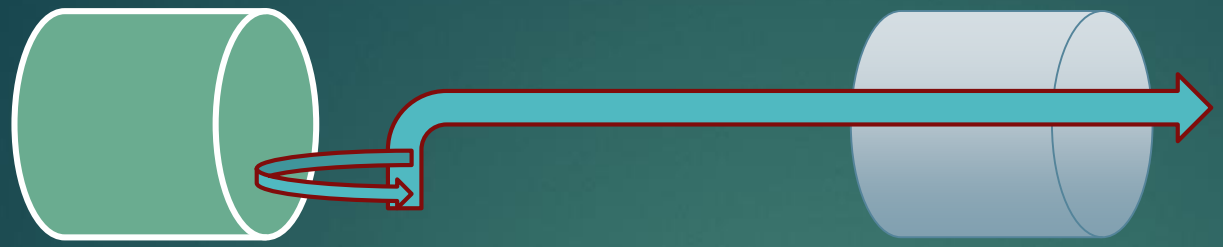
优先级队列



Workers

```
case class Res(workerId:
String,memUsage: Double,cpuUsage:
Double,cpuCores: Int,totalMem:
Double) extends TransferMsg
```

Worker定时发送本机资源信息到master，master通过优先级队列进行资源重排，若要求高性能，可采用事件机制



优先级队列

普通队列

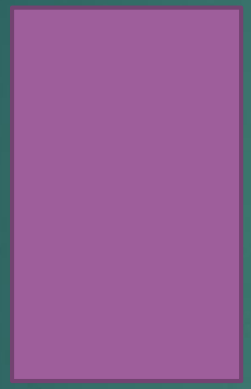
从服务器定时发送心跳报告资源情况
主服务器从优先级取得并删除对头元素
同时主服务器将从服务器缓存到普通队列中，普通队列远远小于优先级队列
当优先级队列为空的时候从普通队列取出

爬虫解析引擎

解析逻辑

解析
Master
Akka

解析
Slave
Akka



Job服务器

Pull

注意

ParserManager首先判断是否为第一层，如果是第一层则将爬虫实体更改下次爬取时间后重新放回爬虫队列

Master解析逻辑

1.Master通过Akka收到子爬虫发过来的rowkey
2.调用资源管理进行对比判断Slave的资源情况并将数据发给对应的Slave

另外
1.Master通过主备解决单点问题
2.Slave会心跳给Master（统一封装到资源管理中），更新资源情况并以此判断Slave是否挂掉

Slave解析逻辑

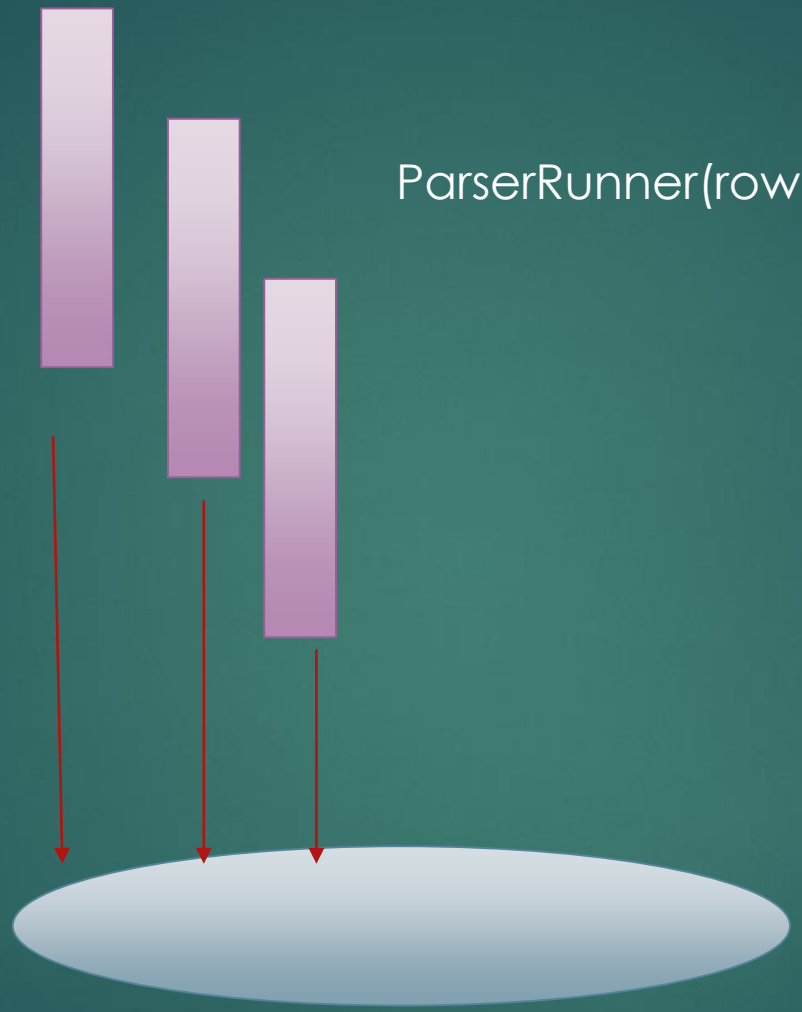
1.启动线程池
2.将rowkey传给ParserRunner，并进入多线程
3.调用抽象类ParserManager，解析对象在ParserRunner中初始化
4. ParserManager首先通过rowkey委托给**存储管理器(StorageManager)**去hbase获取数据，并进行反序列化
5. ParserManager将从存储管理器取得的数据委托给解析者Parser的工厂ParserFactory 工厂模式根据爬虫类型创建Parser）,工厂创建Parser调用parse进行解析.
6.如果是通用爬虫,ParserManager解析者**Parser**调用通用爬虫引擎General Parser，如果是API爬虫调用APIParser,如果是主题爬虫TopicParser,解析方法内部根据调用主题爬虫解析者域名来通过反射（使用控制反转，缓存对象）调用具体的解析对象（这里使用抽象类，url抽取使用的是相同的）的解析方法，也就是解析规则可以根据域名进行二次开发，形成通用库
7.解析方法调用url分析器URLAnalyzer，根据url生成Jsoup Document
8.使用Jsoup抽取ur和内容，委托给内容解析器ContentParser进行分析，提取所需的内容根据数据结构封装好返回
9.抽取的url委托给url过滤器ITaskFilter过滤url（url过滤规则通过JobId去Job服务器取得）
10.过滤后的url以及实体交给url组装器URLAssembler器根据参数组装url
11.调用URL去重管理器URLDistinctManager（如果是重复的url不调用调度器入库Redis）
12.然后剩下的不重复的调用调度器，将url重新交给redis队列
13. 将网页内容委托给存储管理器持久化到MongoDB

ParserMaster为任务id分配资源

1.通过阻塞队列去kafka获得rowkey
2.然后通过资源管理器判断是否有充足的资源
3.如果有就分配，如果都没有则休眠5s然后重新将任务放入阻塞队列

需要工作线程，一个双扫监控线程

解析
Slave
Akka

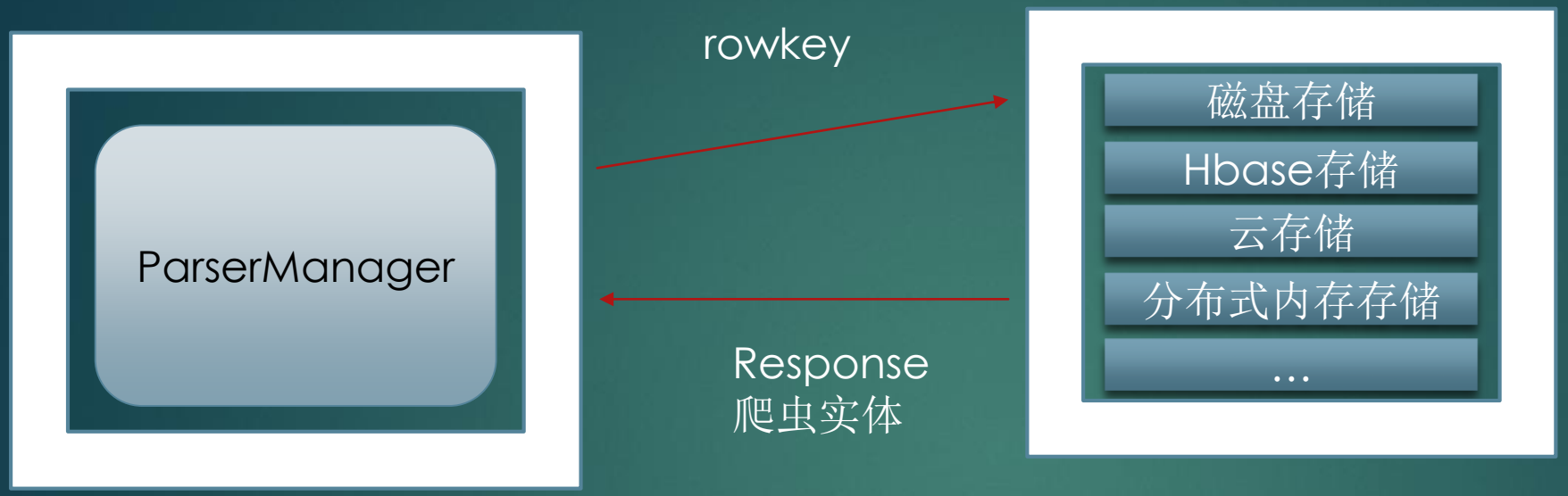


ParserRunner(rowkey)

守护线程池，执行
ParserRunner

1.启动线程池
2.将rowkey传给ParserRunner，并进入多线程

线程池数量由woker机器的CPU核数决定，也可以自行配置



ParserRunner

存储管理器

3.调用抽象类ParserManager，解析对象在ParserRunner中初始化

4. ParseManager首先通过rowkey委托给**存储管理器(StorageManager)**去hbase获取数据，并进行反序列化

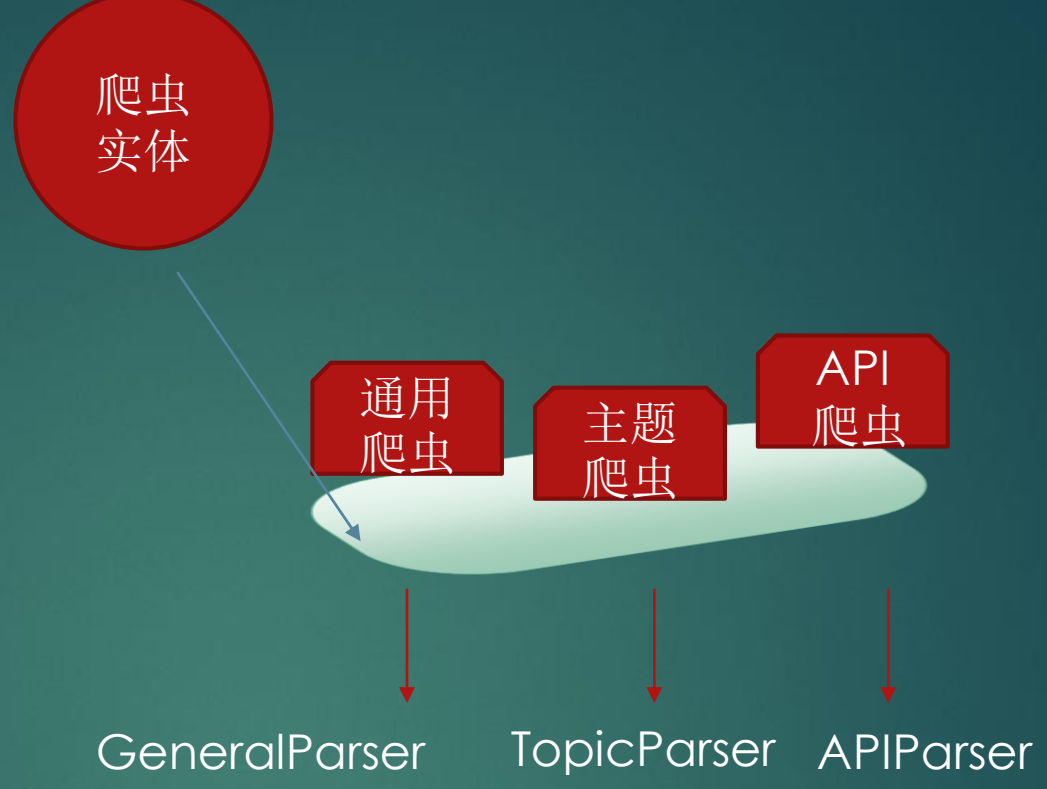
存储管理器

getByKey(key:String): T

getByKey(key:String): Seq[T]

putByKey(key:String,value:T): Boolean

put (entity:T): Boolean



5. ParserManager将从存储管理器取得的数据委托给解析者Parser的工厂ParserFactory（crawlerTaskEntity）（工厂模式根据爬虫类型创建Parser,实际系统中根据域名进行创建[注意从缓存中取，保证有且仅有一个对象，因为在多线程环境下，需要加锁]）,工厂创建Parser调用parse进行解析.

根据域名创建对象通过从映射内存表中获取类全名，如果没有，爬虫类型为主题爬虫时候，抛出异常，反之使用通用解析引擎

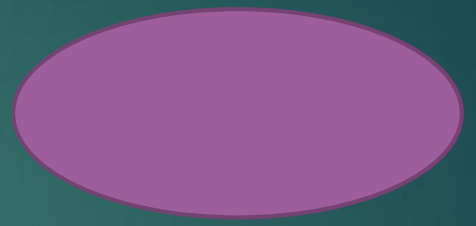
主题爬虫根据域名选择解析实体

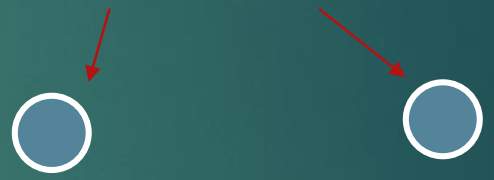
1.通过解析类工厂创建传入参数域名

2.通过域名从本地缓存取对应的实体HashMap（domain->主题爬虫实体）

3.如果没有取到实体则根据爬虫实体中的属性，主题爬虫类路径反射创建实体并放入 本地缓存（注意捕获异常）

工厂模式
根据域名产生具体解析





RatePublicParser

...

Version0.1

6.如果是通用爬虫， ParserManager调用是 al Parser，如果是API爬虫调用APIParser,如果是主题爬虫TopicParser,解析方法内部根据调用主题爬虫解析者域名来通过反射（可以使用解析对象池，使用队列）调用具体的解析对象ParserByDomain.parse（这里使用抽象类，url抽取使用的是相同的）的解析方法，也就是解析规则可以根据域名进行二次开发，形成通用库

7.解析方法调用url分析器URLAnalyzer，根据url生成Jsoup Document
8.使用Jsoup抽取ur和内容，委托给内容解析器ContentParser进行分析，提取所需的内容根据数据结构封装好返回

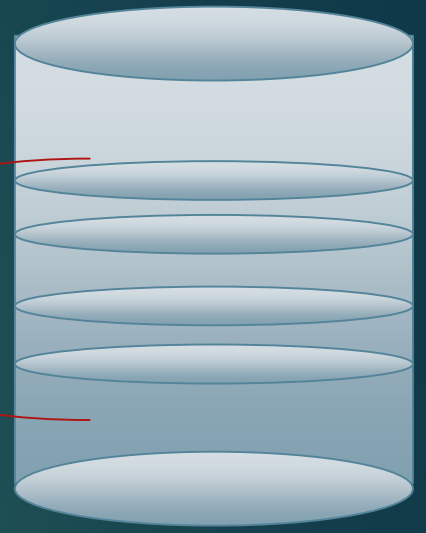
TopicParser.parse()

TopicParser.geneNewTaskEntitySet()

比如说分页，没有返回null,产生新任务，但是批次不变
还有，如果深度是第一级别的需要重新放回队列

ITaskFilter. matchRule

rules



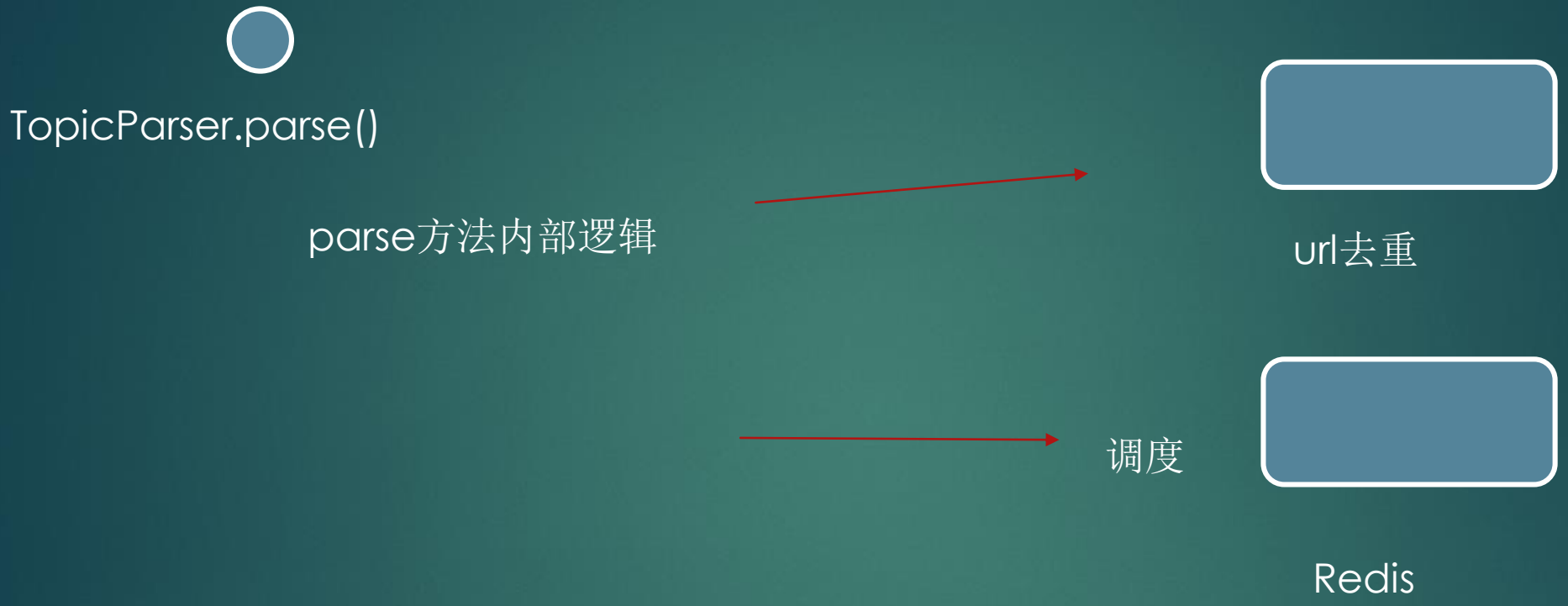
Job服务器

parse方法内部逻辑

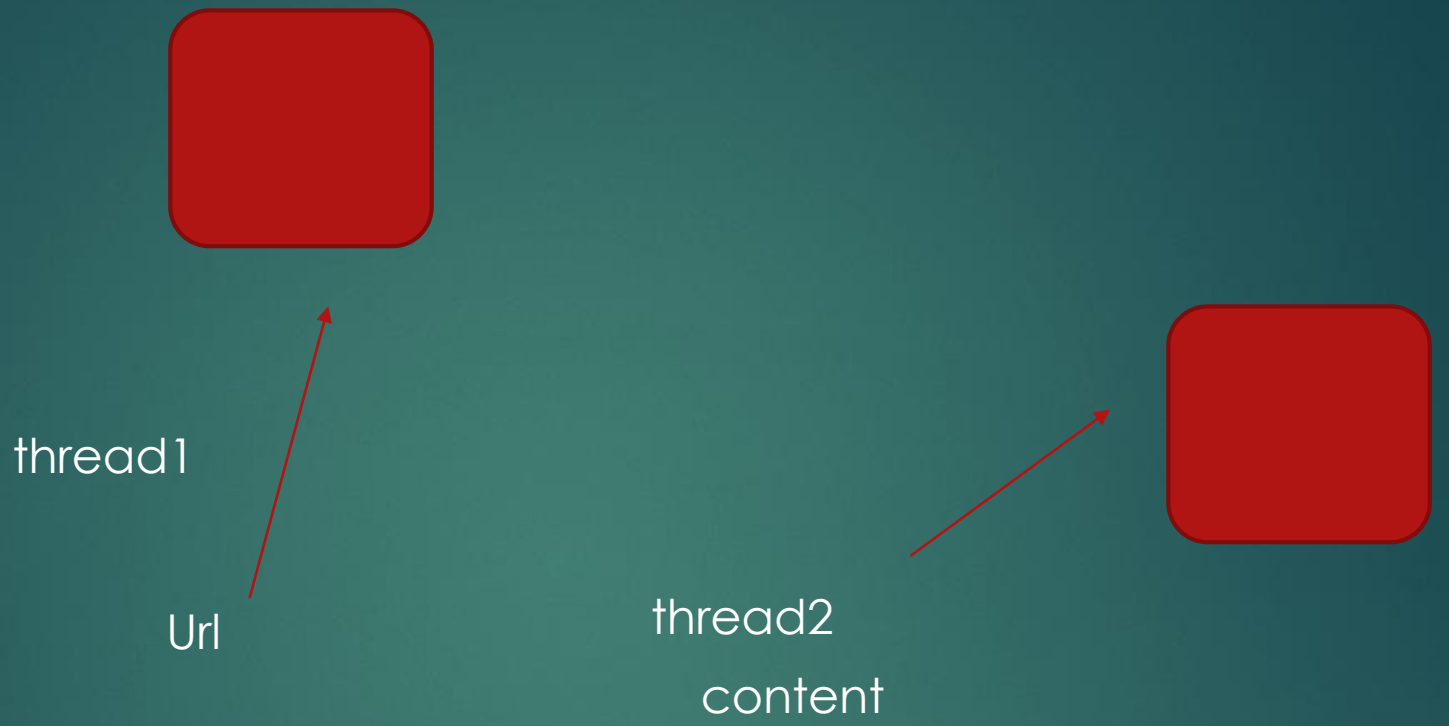
taskFilter通过工厂创建，默认从本地缓存取Map(jobid->TaskFilter)　TaslFilter是pipleline形式，使用装饰模式，一个job对应一个嵌套TaskFilter，只需要一个触发条件

9.抽取的url委托给url过滤器ITaskFilter过滤url（url过滤规则通过JobId去Job服务器取得）

10.过滤后的url以及实体交给url组装器URLAssembler器根据参数组装url



11.调用URL去重管理器URLDistinctManager（如果是重复的url不调用调度器入库Redis）
12.然后剩下的不重复的调用调度器，将url重新交给redis队列



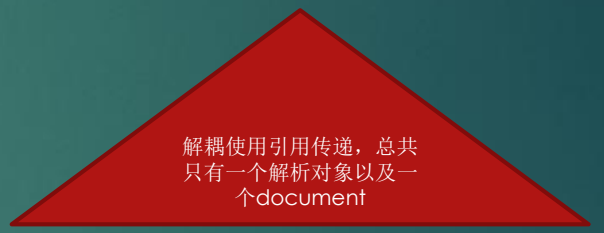
13. 将网页内容委托给存储管理器持久化到MongoDB

关于内容和url解析在parse内部再开两个线程分别同时进行处理

url解析与内容解析（入库）解耦

Parse(resobj){
1.生成parser对象
2.自动判断任务终止条件
3.抽取url（CralwerTaskEntity）
4.异步抽取内容并入库
5.对抽取的urls(CralwerTaskEntity)进行过滤，去重…
6.将新的url(CralwerTaskEntity)重新放入队列，等待调度
}

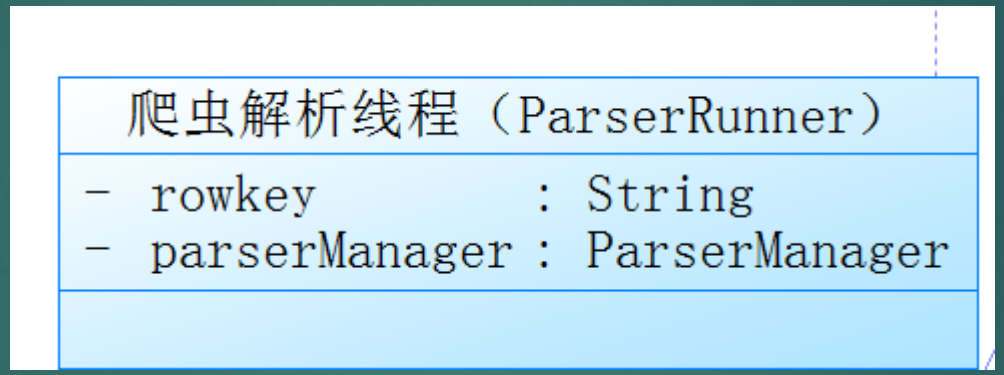
使用阻塞队列BlockingQueue,插入不阻塞，读取进行阻塞，读取的时候使用双扫线程，将读入的数据放入另外一个解析线程池进行解析

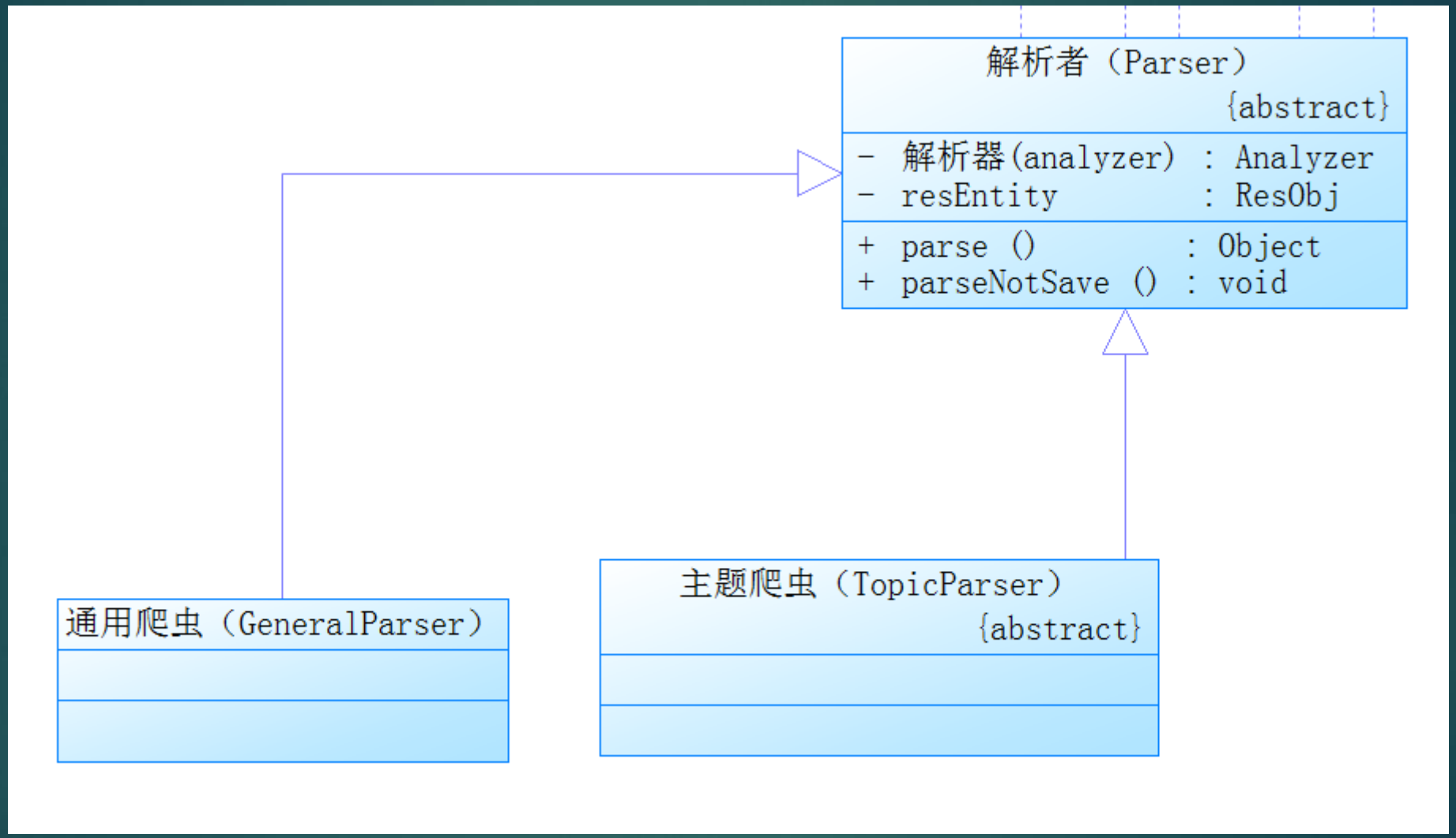


解析url并重新调度线程池

解析内容并封装成对象，交给存储管理器（默认使用mongo）


解析管理者（ParserManager）
{abstract}
- storageManager : StorageManager
- parser : Parser
- 解析器(analyzer) : Analyzer
- Url过滤器（taskFilter） : ITaskFilter
- url组装器(urlAssembler) : URLAssembler
- url去重（urllDistinctManager） : URLDistinctManager
- 调度器（ITaskQueue） : ITaskQueue
+ 启动（String taskKey） : void
+ 获取爬虫待分析（） : ResObj
+ 抽取utl连接（ResObj 待分析实体） : List<Link>
+ 抽取href Urls（ResObj 待分析实体） : List<String>
+ url过滤（ResObj 待分析实体, List<String> 抽取的url链接） : List<String>
+ url去重（根据类型，通用爬虫需要去重）（ResObj 待分析实体, List<String> 抽取的url链接） : List<String>
+ 组装url（ResObj 待分析实体, List<String> 抽取的url链接） : List<CrawlerTaskEnti
+ 调度，重新放入队列（CrawlerTaskEntity 爬虫实体对象） : boolean
+ 保存实体（） : boolean


爬虫解析线程（ParserRunner）
- rowkey : String
- parserManager : ParserManager


解析者（Parser）
{abstract}
- 解析器(analyzer) : Analyzer
- resEntity : ResObj
+ parse () : Object
+ parseNotSave () : void
通用爬虫（GeneralParser）
主题爬虫（TopicParser）
{abstract}

更新爬虫爬取周期

爬虫端实时更新

通过http header探测

异步更新周期

通过异步线程通过事件队列进行事件触发（比如可以有定时触发事件和异步模型更新完成事件）

实现方式通过机器学习

反被封

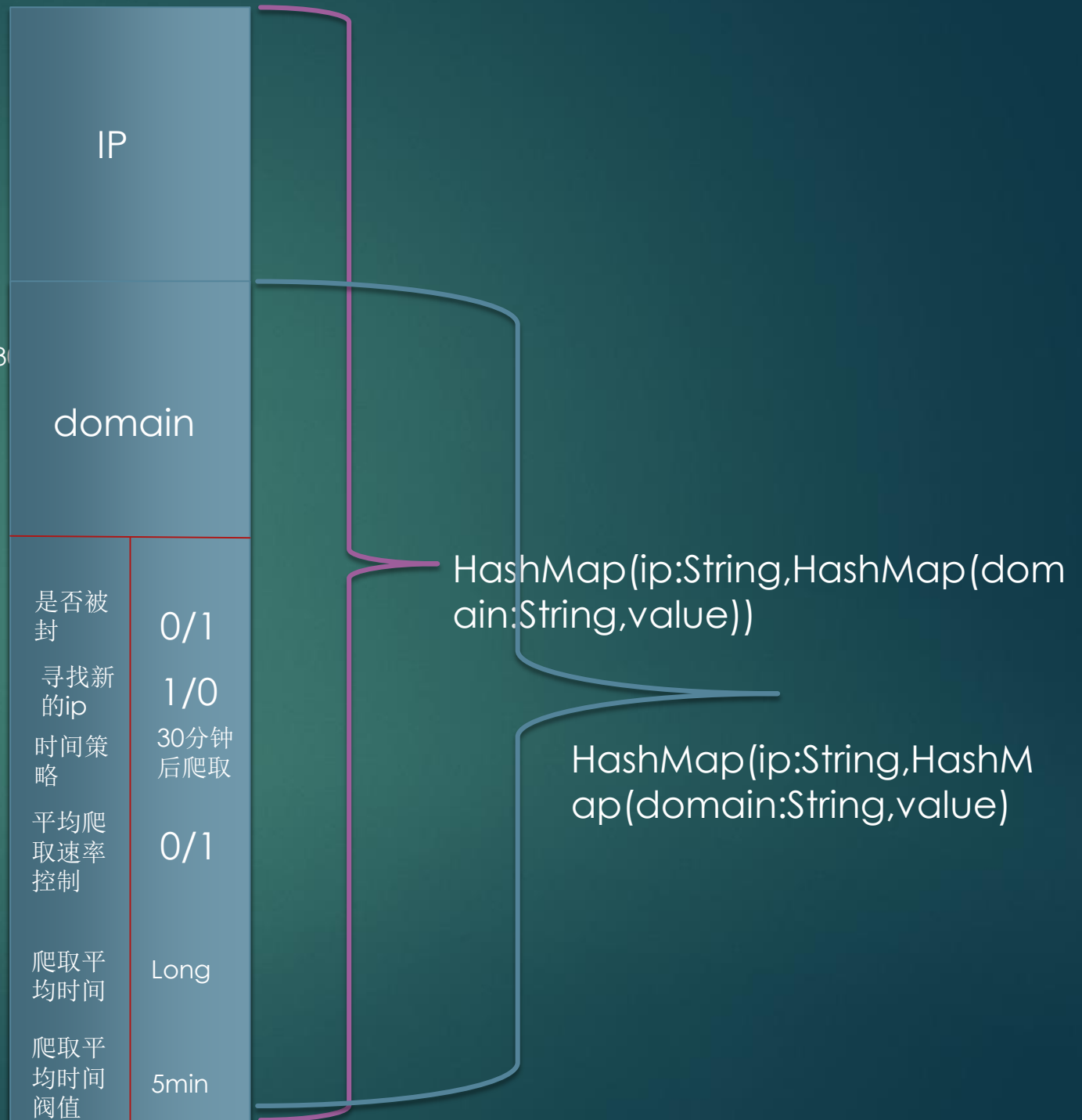
爬取什么

什么时候爬取

能不能爬取，什么时候能爬取

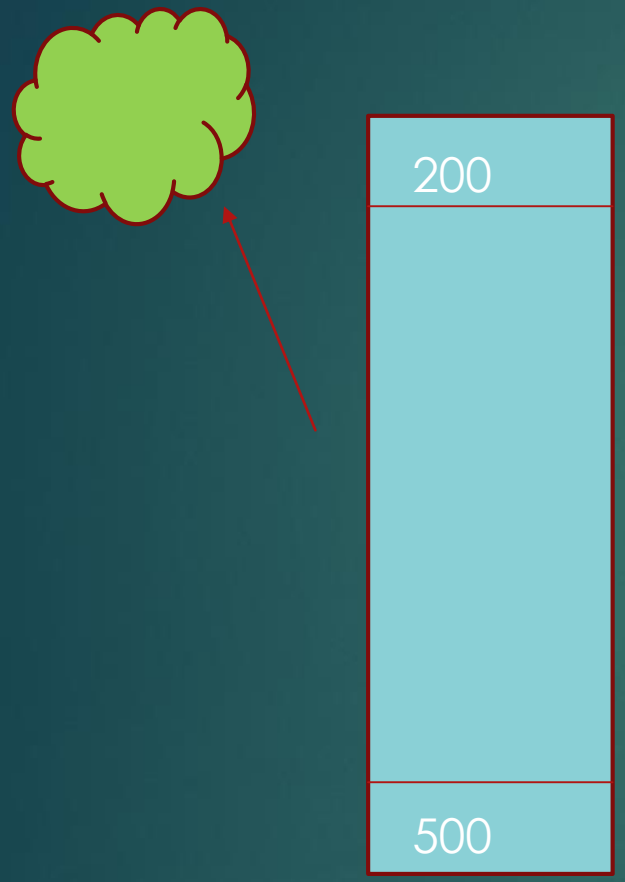
@1.爬取IP被远程域给封了 -》 1）重新选取IP爬取 2）3
@2.爬取IP爬取远程域很慢（由于地域分布问题）

怎么爬取



机器学习训练网页是否被封

（类似垃圾邮件过滤）



2.离线抽取网页源码，人工进行训练集标注，然后通过机器学习（可以使用native bayes、SVM或者logistic regression）

日志记录器

1.爬虫请求页面，如果是404则直接丢掉，如果是在300到500中间，则将网页内容记录下来，如果是302（开启默认不让浏览器跳转），则重新手动请求Location并记录下跳转后的页面源码

3.每次请求的时候如果是200到500之间，则得到源码并分词抽取向量输入模型进行判断

4.如果是被封状态则将redis的状态改为被封，方便分配资源的时候进行ip轮训

一般网站封ip的维度为

domain	IP	userAgent

关于浏览器的动态分配

怎么判断是否被封，通过机器学习

domain	当前浏览器	当前使用的是第几个浏览器

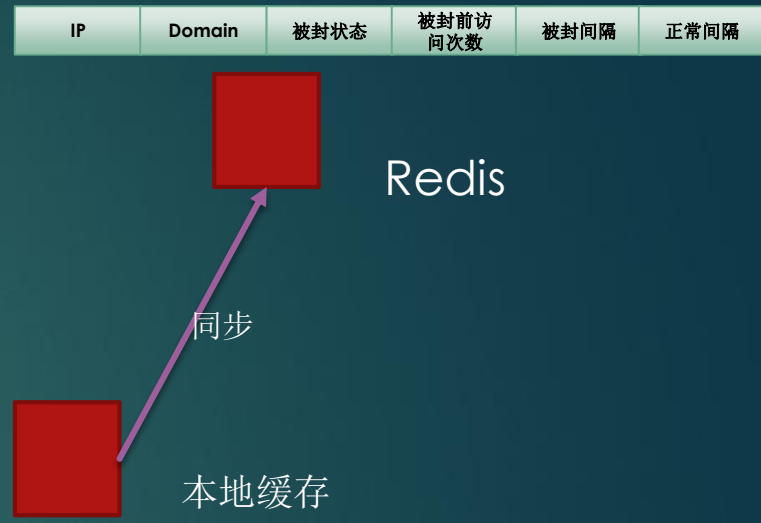
指针

被封指标----单位时间内访问域的次数　　次数/单位时间

目标，通过一台机器上因为被封学习到的经验分享到其他机器，以免其他机器被封掉

1.怎么才能算被封
2.被封后该怎么处理
3.怎么预测被封（一台机器学习到的规则，资源共享）

一台机器学习的经验，全局共享



AgentWorker

domain	(被封前请求开始时间，被封前请求结束时间，访问次数，开始被封时间，结束被封时间，状态（被封或者没有被封）)

可以简化为下面，被封前请求开始时间就等于被封结束时间，同理，被封前请求结束时间就等于开始被封时间

domain	(访问次数，开始被封时间，结束被封时间，状态（被封或者没有被封）)

在callback里面更新，当ip被封的时候，将本地缓存放入redis,放之前判断（请求结束时间-请求开始时间）是不是小于redis中已经存在的时间，如果是则更新，反之不更新；
然后将本地缓存的请求开始时间和请求结束时间归零，记录开始被封时间，同时将被封状态改为被封；每次请求200的时候判断是否被封状态为被封如果是则更新结束被封时间，同时更新被封状态为解封

Callback中，每次请求完进行判断，同一域名，时间是不是接近于redis中的时间，如果是则取出请求次数，判断本地请求次数是否接近于redis中记录的，基于性能考虑可以每天去redis中获取该记录同步到本地缓存，如果请求次数接近Redis则将任务重新放入redis sortset，并设置好相应的Interval阀值等待调度，可以设置为3分钟

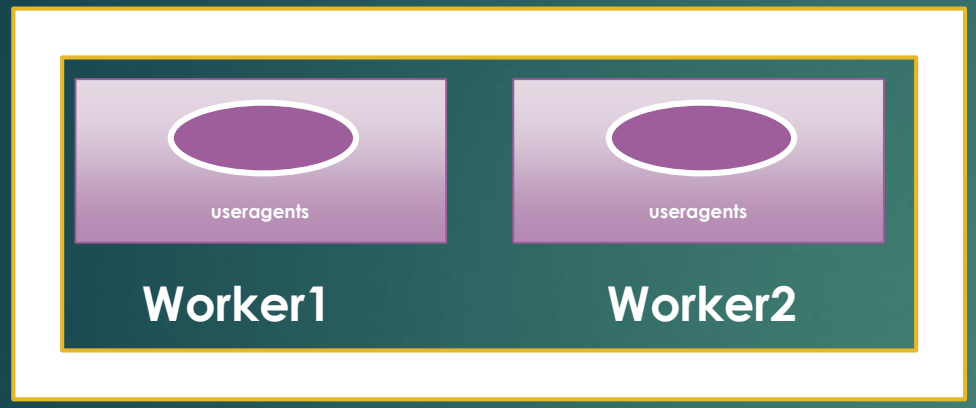
在资源调度的基础上，基于反封策略进行调度

1.Ip轮询
2.时间间隔策略(如果所有ip轮询完后都处于被封状态则启用该策略)

domain	(请求开始时间，请求结束时间，访问次数，开始被封时间，结束被封时间，状态（被封或者没有被封）)

Redis里面取得该对象，得到(被封结束时间-被封开始时间），为了性能考虑，可以定时异步从redis获取与本地缓存同步

代理Proxy模块



被封

如果所有Worker上的物理IP都被封了，没加入代理Proxy之前是将任务放进sortset等待调度，等待时间是被封的间隔时间；
　加入代理Proxy之后则不是直接等待调度，而是首先启用代理，代理的有效期只在物理ip被封的时间段内，如果在该时间段，所有的Proxy也被封了，则重新放入sortset等待调度

AgentMaster

proxyIp+domain	(proxy启用时间，当前proxy索引,已经执行的proxy个数)

(proxyHost,Port)

Proxy数组，定期与redis set合并

Logic

1.如果所有worker ip被封，启用proxy,并选择第一个proxy,更新缓存
2.如果被封时间间隔(当前时间-proxy启用时间 与 实体里面被封时间字段对比)到了，关掉proxy开关，启用worker 物理ip
3.如果所有的proxy都被封了，而还在被封状态，重新放入sortset score为(实体里面被封时间字段-当前时间-proxy启用时间)

爬虫类图

模块结构

爬虫控制

爬虫任务实体

爬虫处理

分布式调度模块

内容去重模块

DNS缓存

任务爬取处理类型

任务过滤规则

调度模块(任务队列封装)

解析处理

存储管理

网页内容管理模块

URL去重模块

爬虫URL生成器

规则更新相关处理

机器学习分析基础数据

爬虫代理资源池管理

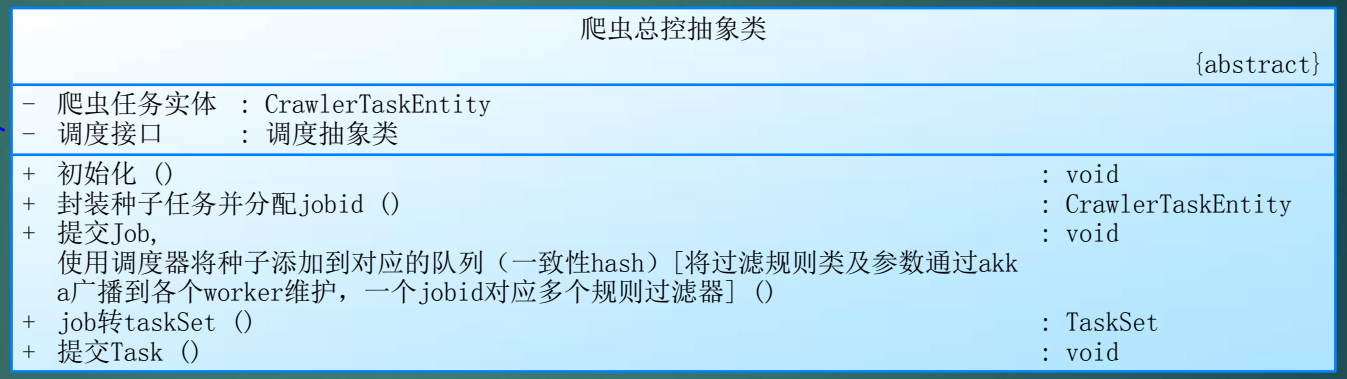
通用底层通信模块

资源管理

实时监控

总控

去工作(){
分析url
匹配规则
一致性hash调度
}



任务实体

Job类

- jobId	: String
- job名称	: String
- Job类型(主题爬虫、通用爬虫、指定ur爬虫)	: int
- 过滤规则列表(提交种子任务的时候使用Akka广播规则，可以使用反射，先实例化，然后传入规则)	: List<ITaskFilter>
- 爬虫任务实体列表	: List<CrawlerTaskEntity>
- 规则更新	: RuleUpdateType
- URL过滤规则（定义接口，可以使用主从管理方式，通过Akka通信）	: ITaskFilter
- 父job批次id	: int
- job批次id	: int

爬虫任务实体 {abstract}

- 爬取任务类型	: int

爬虫类型（枚举）

- API爬虫	: int
- 通用网络爬虫	: int
- 主题爬虫	: int

任务类型实体（使用单例模式） {abstract}

- id	: int
- 任务类型名	: String
- 规则过滤创建工厂	: RuleCreateFactory
- 爬虫创建工厂	: TaskFetchHandlerFactory
- 规则更新创建工厂	: TaskRuleUpdateTypeFactory
+ 创建伪规则过滤器 ()	: ITaskFilter
+ 创建伪爬虫 ()	: TaskFetchHandler
+ 创建伪更新规则类型 ()	: TaskRuleUpdateType

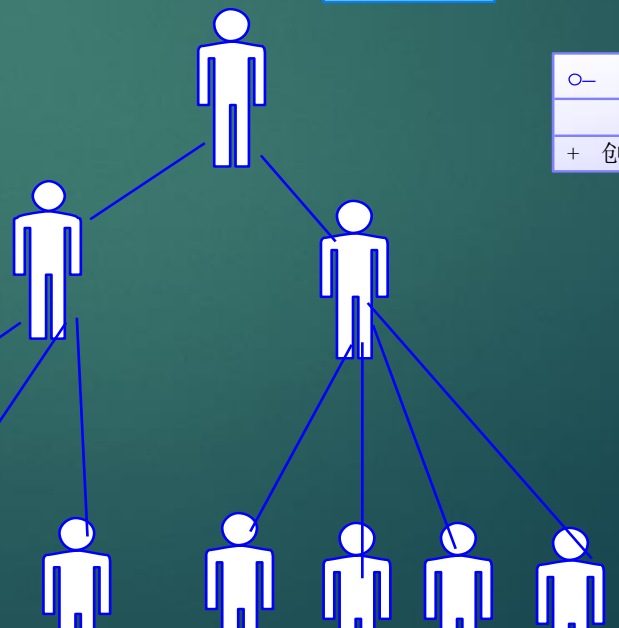
URL实体bean

- Job唯一id	:
- job名称	:
- 父job批次id	:
- job批次id	:
- 父任务Id	:
- 父任务携带参数	:
- 任务Id	:
- 任务URL	:
- userAgent	:
- cookies	:
- charset	:
- 任务domain	:
- 是否使用agent爬取	:
- 爬取状态	:
- 重试次数	:
- 是否使用代理	:
- 规则更新类别	:
- 下载类型,如http（解析的时候判断）	:
- 爬取深度级别	:
- 上次爬取时间	:
- 上次爬取所用时间	:
- 上次爬取完成时间	:
- 针对主题爬虫再次运行中的间隔时间	:
- 什么时候停止爬取（默认3次，跟爬虫类型有关）	:
- 创建时间	:
- 更新时间	:

Api任务实体

任务类型实体创建工厂

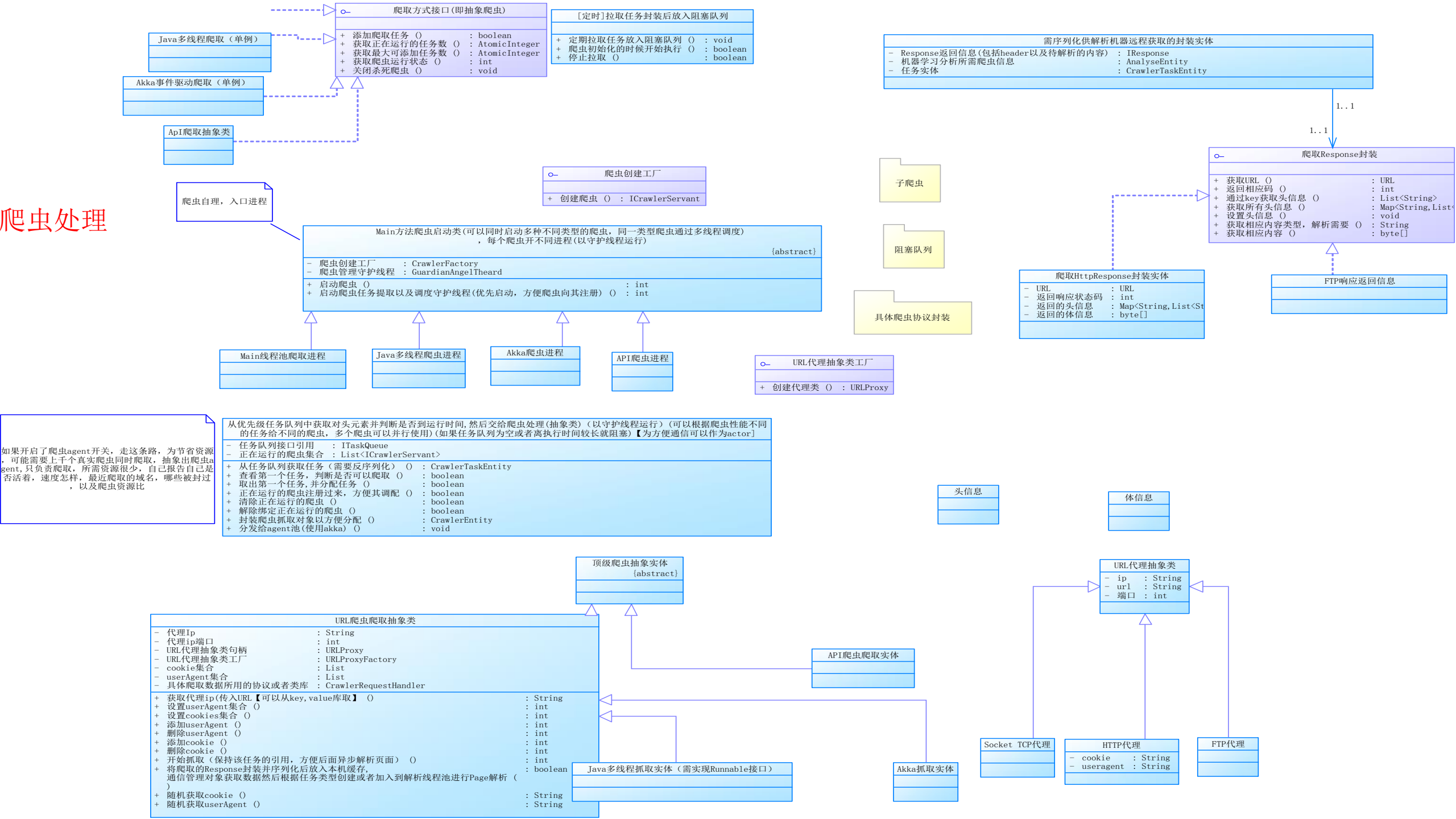
+ 创建任务类型（单例） ()	: TaskType



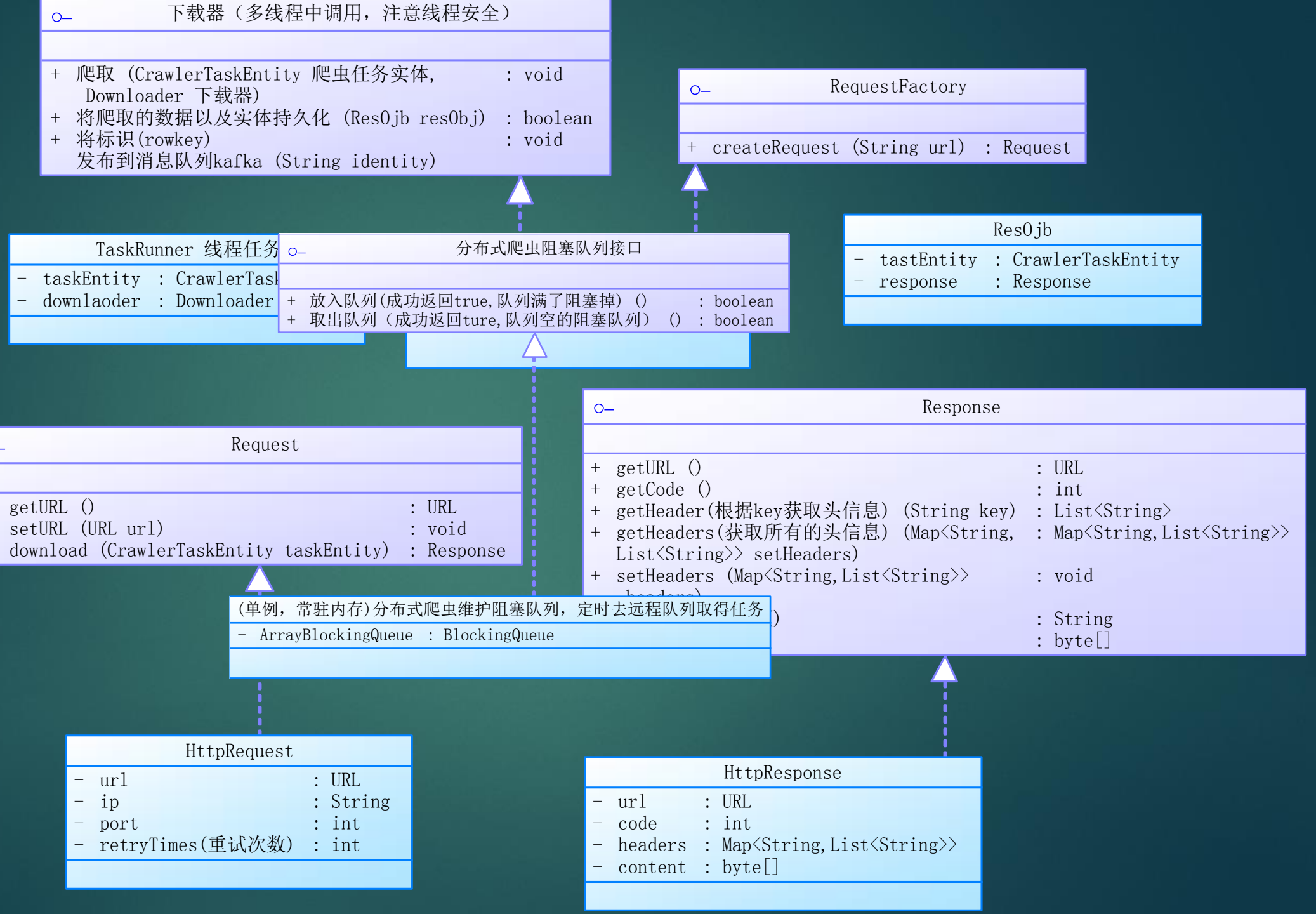
Site

- domain	: String
- userAgent	: String
- cookies	: Map
- charset	: String

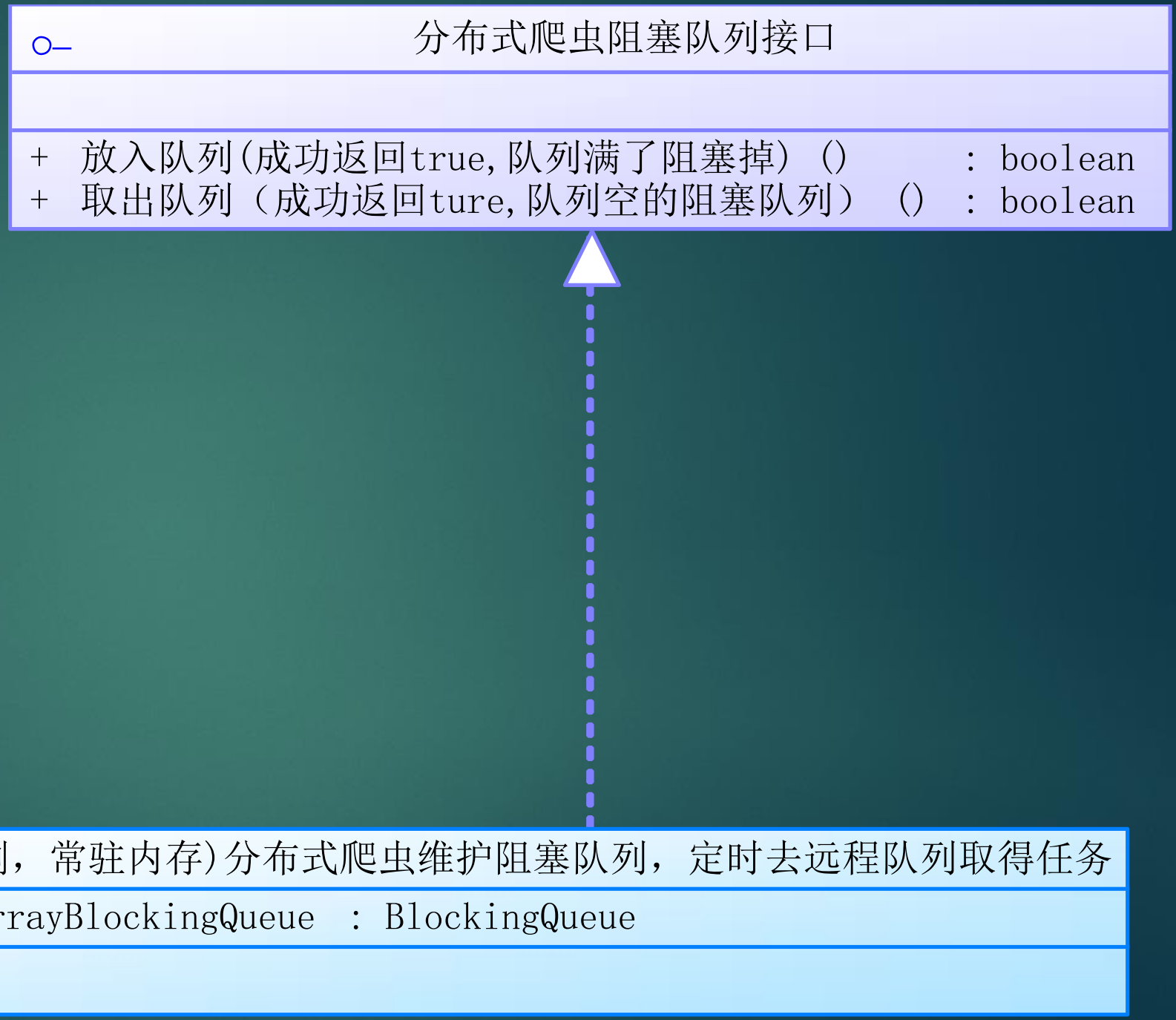
爬虫处理


爬取方式接口(即抽象爬虫)
+ 添加爬取任务 () : boolean
+ 获取正在运行的任务数 () : AtomicInteger
+ 获取最大可添加任务数 () : AtomicInteger
+ 获取爬虫运行状态 () : int
+ 关闭杀死爬虫 () : void
[定时]拉取任务封装后放入阻塞队列
+ 定期拉取任务放入阻塞队列 () : void
+ 爬虫初始化的时候开始执行 () : boolean
+ 停止拉取 () : boolean
Java多线程爬取（单例）
Akka事件驱动爬取（单例）
ApI爬取抽象类
爬虫自理，入口进程
爬虫创建工厂
+ 创建爬虫 () : ICrawlerServant
Main方法爬虫启动类(可以同时启动多种不同类型的爬虫，同一类型爬虫通过多线程调度)
，每个爬虫开不同进程(以守护线程运行)
{abstract}
- 爬虫创建工厂 : CrawlerFactory
- 爬虫管理守护线程 : GuardianAngelTheard
+ 启动爬虫 () : int
+ 启动爬虫任务提取以及调度守护线程(优先启动，方便爬虫向其注册) () : int
Main线程池爬取进程
Java多线程爬虫进程
Akka爬虫进程
API爬虫进程
子爬虫
阻塞队列
具体爬虫协议封装
URL代理抽象类工厂
+ 创建代理类 () : URLProxy
需序列化供解析机器远程获取的封装实体
- Response返回信息(包括header以及待解析的内容) : IResponse
- 机器学习分析所需爬虫信息 : AnalyseEntity
- 任务实体 : CrawlerTaskEntity
1..1
1..1
爬取Response封装
+ 获取URL () : URL
+ 返回相应码 () : int
+ 通过key获取头信息 () : List<String>
+ 获取所有头信息 () : Map<String,List<
+ 设置头信息 () : void
+ 获取相应内容类型，解析需要 () : String
+ 获取相应内容 () : byte[]
爬取HttpResponse封装实体
- URL : URL
- 返回响应状态码 : int
- 返回的头信息 : Map<String,List<St
- 返回的体信息 : byte[]
FTP响应返回信息
如果开启了爬虫agent开关，走这条路，为节省资源，可能需要上千个真实爬虫同时爬取，抽象出爬虫agent,只负责爬取，所需资源很少，自己报告自己是否活着，速度怎样，最近爬取的域名，哪些被封过，以及爬虫资源比
从优先级任务队列中获取对头元素并判断是否到运行时间,然后交给爬虫处理(抽象类)（以守护线程运行）(可以根据爬虫性能不同的任务给不同的爬虫，多个爬虫可以并行使用)(如果任务队列为空或者离执行时间较长就阻塞)【为方便通信可以作为actor】
- 任务队列接口引用 : ITaskQueue
- 正在运行的爬虫集合 : List<ICrawlerServant>
+ 从任务队列获取任务（需要反序列化） () : CrawlerTaskEntity
+ 查看第一个任务，判断是否可以爬取 () : boolean
+ 取出第一个任务,并分配任务 () : boolean
+ 正在运行的爬虫注册过来，方便其调配 () : boolean
+ 清除正在运行的爬虫 () : boolean
+ 解除绑定正在运行的爬虫 () : boolean
+ 封装爬虫抓取对象以方便分配 () : CrawlerEntity
+ 分发给agent池(使用akka) () : void
头信息
体信息
项级爬虫抽象实体
{abstract}
URL爬虫爬取抽象类
- 代理Ip : String
- 代理ip端口 : int
- URL代理抽象类句柄 : URLProxy
- URL代理抽象类工厂 : URLProxyFactory
- cookie集合 : List
- userAgent集合 : List
- 具体爬取数据所用的协议或者类库 : CrawlerRequestHandler
+ 获取代理ip(传入URL【可以从key,value库取】 () : String
+ 设置userAgent集合 () : int
+ 设置cookies集合 () : int
+ 添加userAgent () : int
+ 删除userAgent () : int
+ 添加cookie () : int
+ 删除cookie () : int
+ 开始抓取（保持该任务的引用，方便后面异步解析页面） () : int
+ 将爬取的Response封装并序列化后放入本机缓存, 通信管理对象获取数据然后根据任务类型创建或者加入到解析线程池进行Page解析 ( ) : boolean
+ 随机获取cookie () : String
+ 随机获取userAgent () : String
API爬虫爬取实体
Java多线程抓取实体（需实现Runnable接口）
Akka抓取实体
URL代理抽象类
- ip : String
- url : String
- 端口 : int
Socket TCP代理
HTTP代理
- cookie : String
- useragent : String
FTP代理

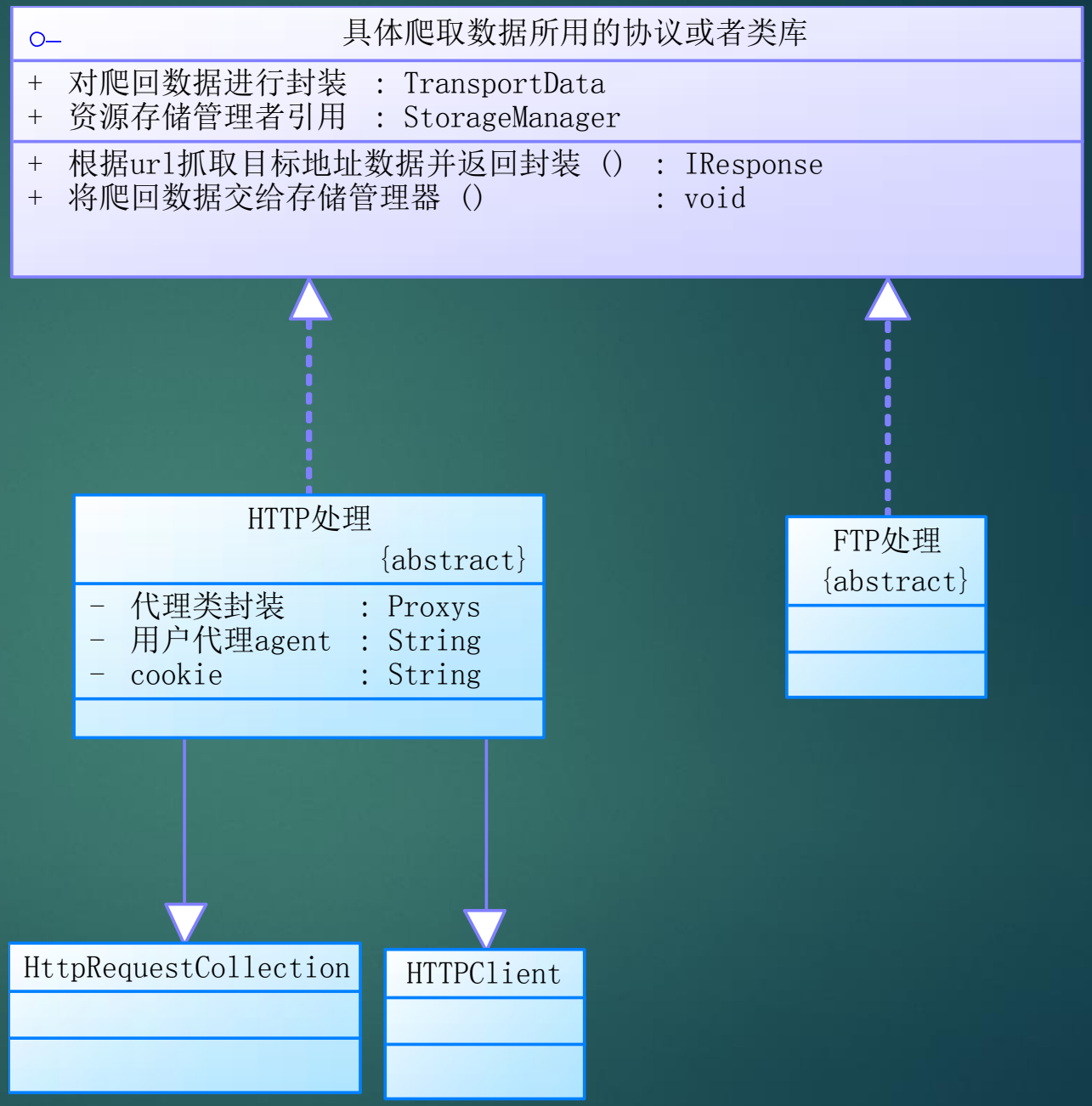
子爬虫


下载器（多线程中调用，注意线程安全）
+ 爬取 (CrawlerTaskEntity 爬虫任务实体, Downloader 下载器) : void
+ 将爬取的数据以及实体持久化 (ResObj resObj) : boolean
+ 将标识(rowkey)发布到消息队列kafka (String identity) : void
RequestFactory
+ createRequest (String url) : Request
TaskRunner 线程任务
- taskEntity : CrawlerTask
- downlaoder : Downloader
分布式爬虫阻塞队列接口
+ 放入队列(成功返回true,队列满了阻塞掉) () : boolean
+ 取出队列（成功返回ture,队列空的阻塞队列） () : boolean
ResObj
- tastEntity : CrawlerTaskEntity
- response : Response
Request
+ getURL () : URL
+ setURL (URL url) : void
+ download (CrawlerTaskEntity taskEntity) : Response
Response
+ getURL () : URL
+ getCode () : int
+ getHeader(根据key获取头信息) (String key) : List<String>
+ getHeaders(获取所有的头信息) (Map<String, List<String>> setHeaders) : Map<String,List<String>>
+ setHeaders (Map<String,List<String>> : void
: String
: byte[]
(单例，常驻内存)分布式爬虫维护阻塞队列，定时去远程队列取得任务
- ArrayBlockingQueue : BlockingQueue
HttpRequest
- url : URL
- ip : String
- port : int
- retryTimes(重试次数) : int
HttpResponse
- url : URL
- code : int
- headers : Map<String,List<String>>
- content : byte[]

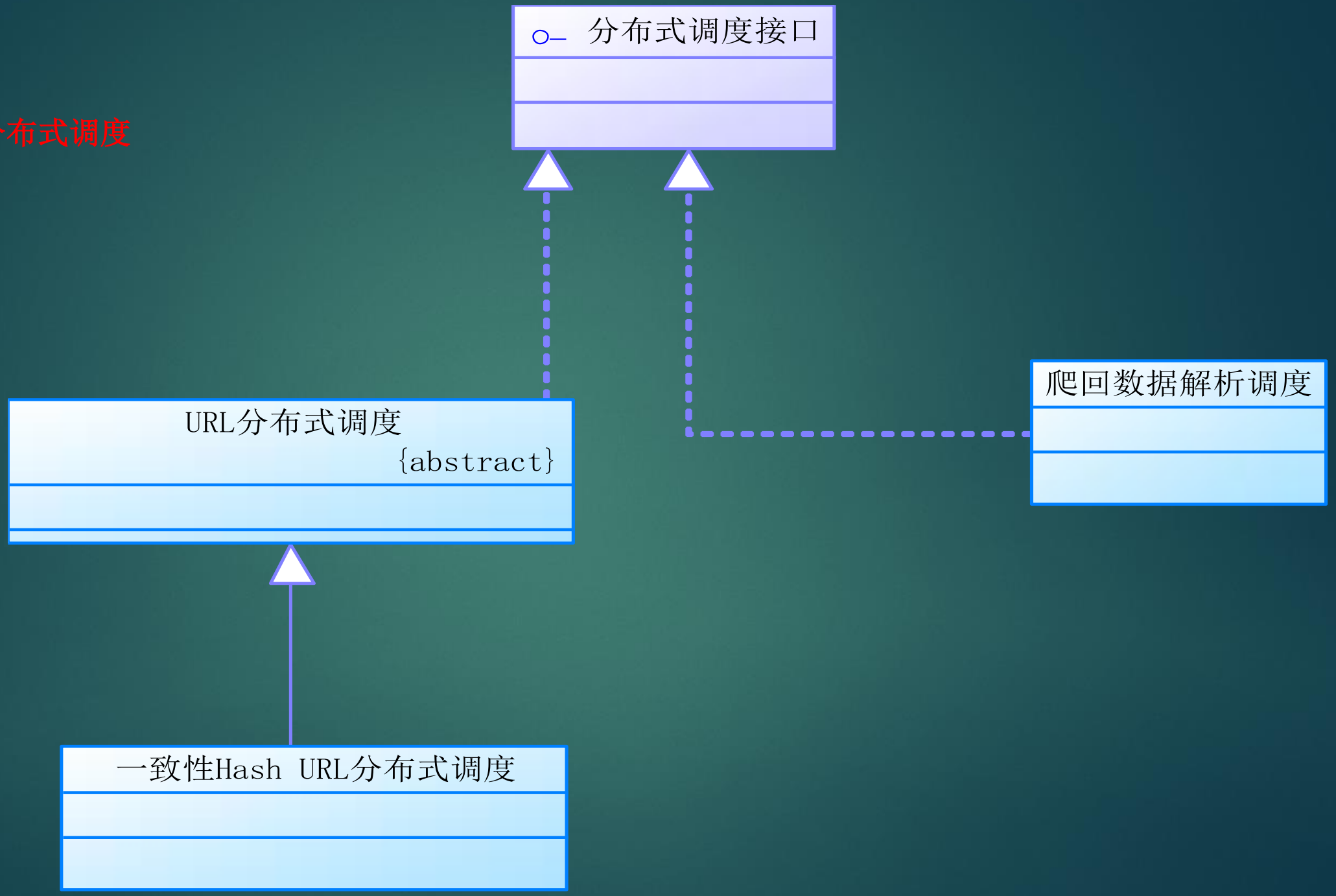
阻塞队列


分布式爬虫阻塞队列接口
+ 放入队列(成功返回true,队列满了阻塞掉) () : boolean
+ 取出队列（成功返回ture,队列空的阻塞队列） () : boolean
(单例，常驻内存)分布式爬虫维护阻塞队列，定时去远程队列取得任务
- ArrayBlockingQueue : BlockingQueue

具体爬虫协议

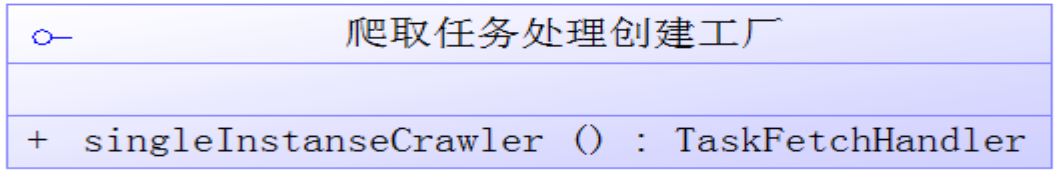

具体爬取数据所用的协议或者类库
+ 对爬回数据进行封装 : TransportData
+ 资源存储管理者引用 : StorageManager
+ 根据url抓取目标地址数据并返回封装 () : IResponse
+ 将爬回数据交给存储管理器 () : void
HTTP处理
{abstract}
- 代理类封装 : Proxys
- 用户代理agent : String
- cookie : String
FTP处理
{abstract}
HttpRequestCollection
HTTPClient

分布式调度

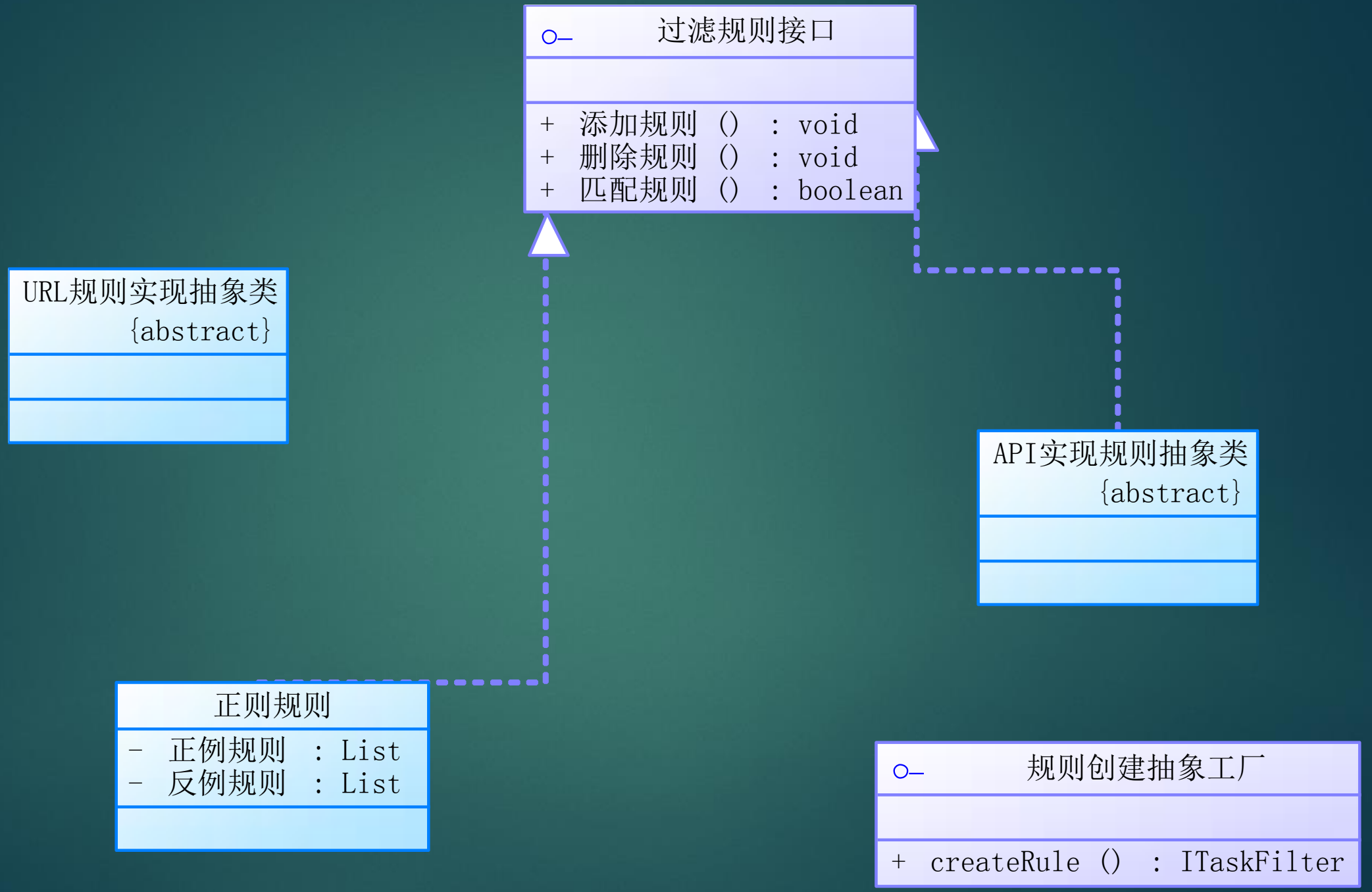


分布式爬取处理

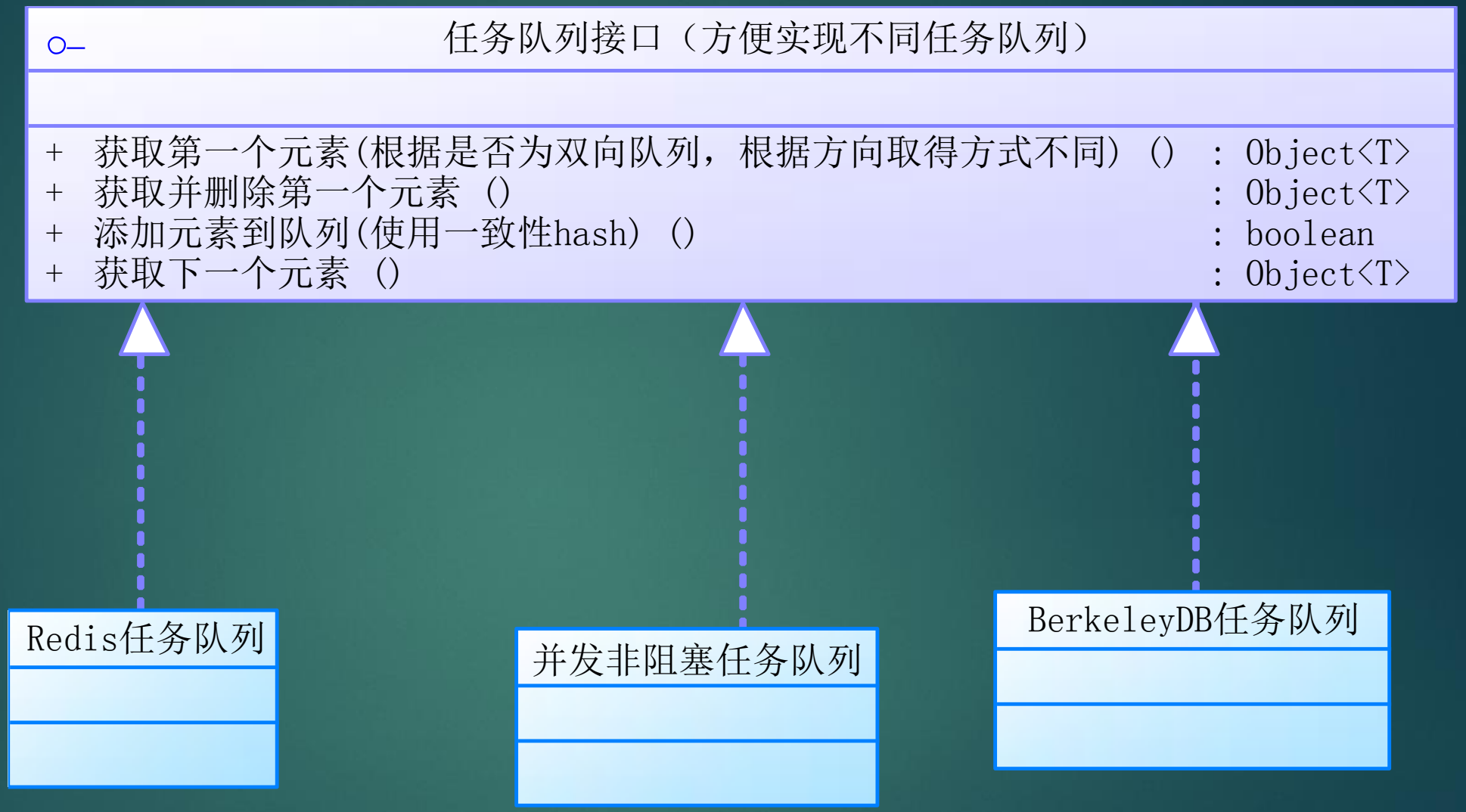

任务爬取处理抽象类
{abstract}
网页爬虫
Api爬虫


爬取任务处理创建工厂
+ singleInstanseCrawler () : TaskFetchHandler

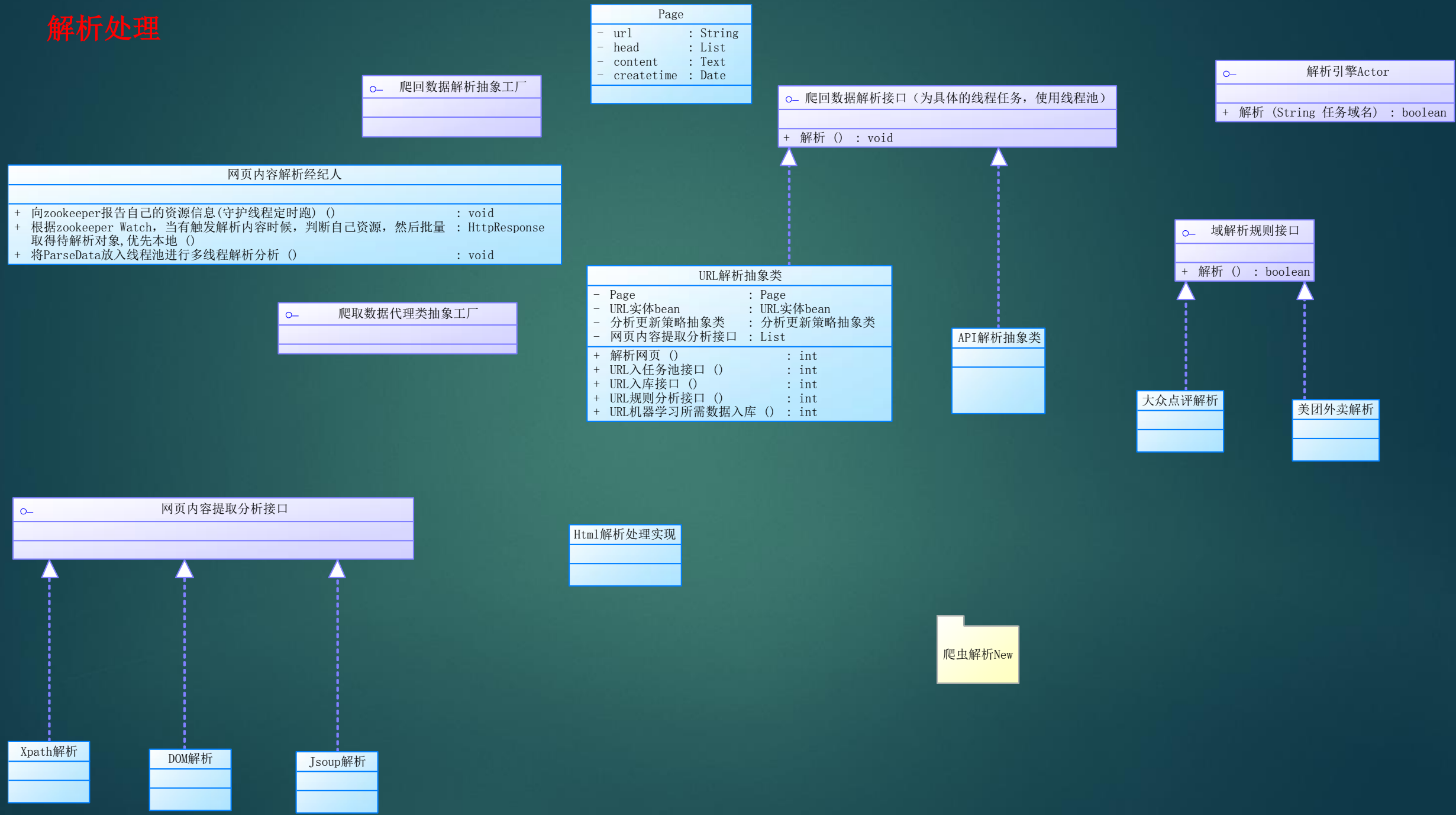
任务过滤规则

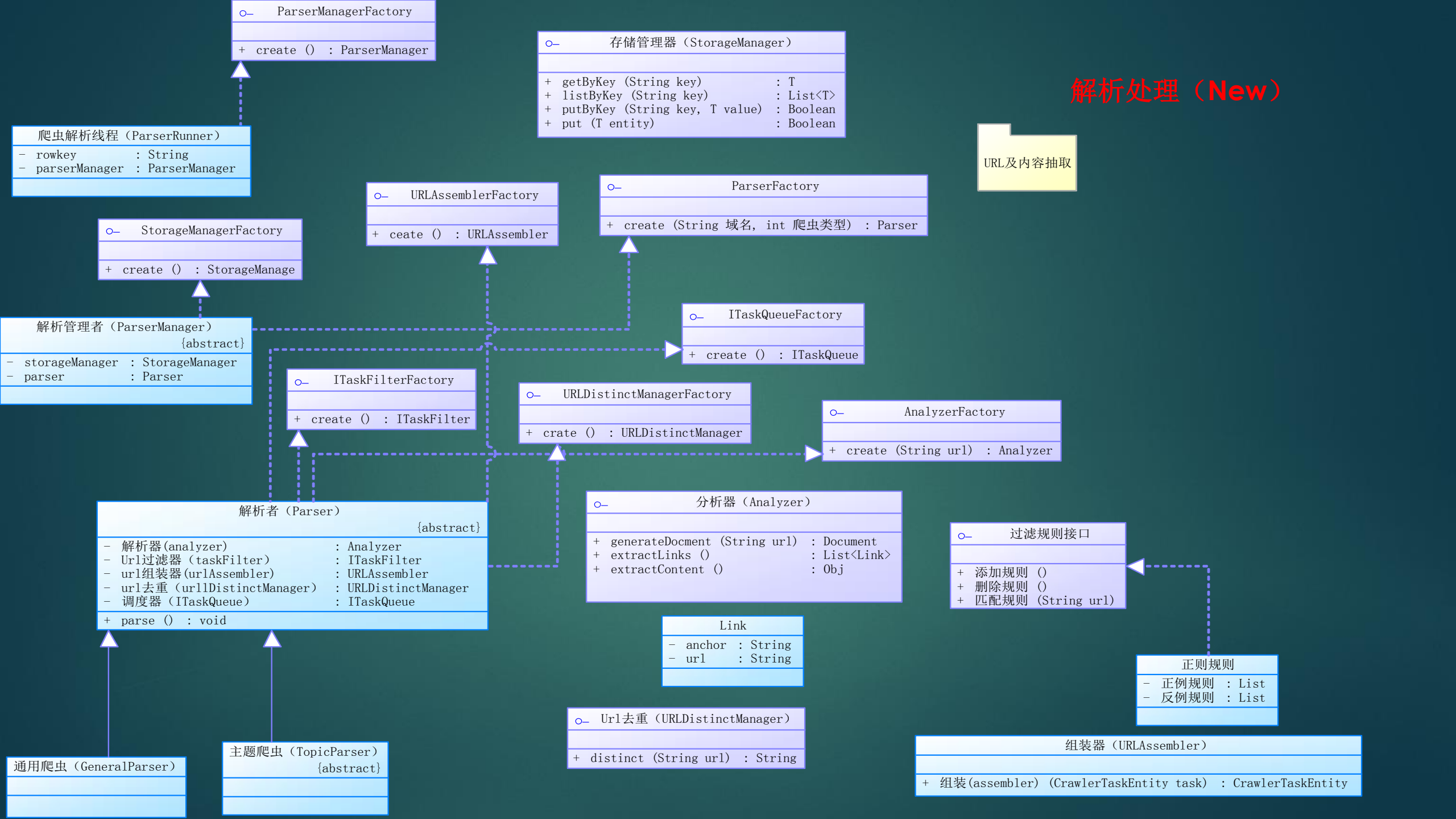

过滤规则接口
+ 添加规则 () : void
+ 删除规则 () : void
+ 匹配规则 () : boolean
URL规则实现抽象类
{abstract}
API实现规则抽象类
{abstract}
正则规则
- 正例规则 : List
- 反例规则 : List
规则创建抽象工厂
+ createRule () : ITaskFilter

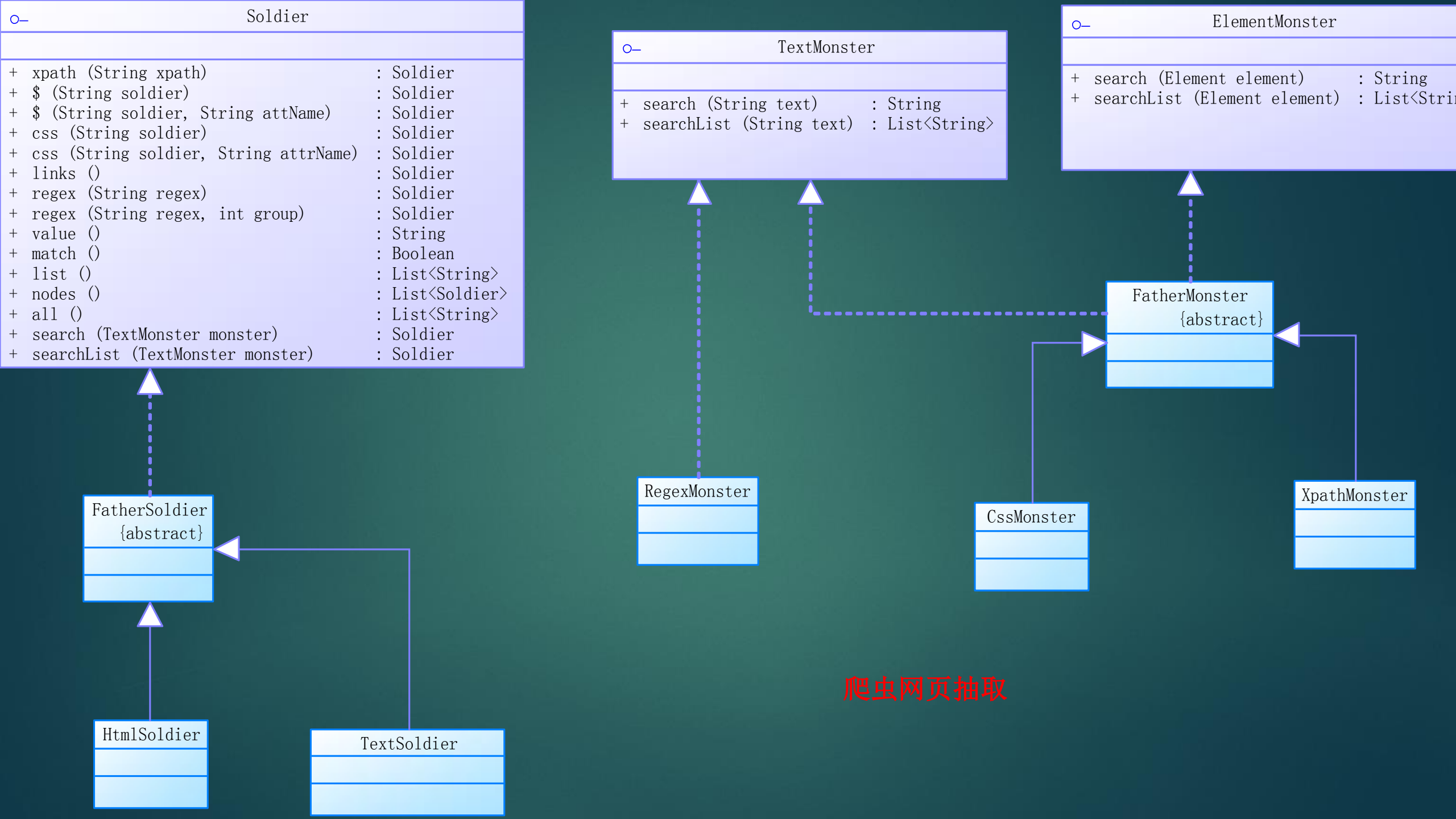
任务队列


任务队列接口（方便实现不同任务队列）
+ 获取第一个元素(根据是否为双向队列，根据方向取得方式不同) () : Object<T>
+ 获取并删除第一个元素 () : Object<T>
+ 添加元素到队列(使用一致性hash) () : boolean
+ 获取下一个元素 () : Object<T>
Redis任务队列
并发非阻塞任务队列
BerkeleyDB任务队列

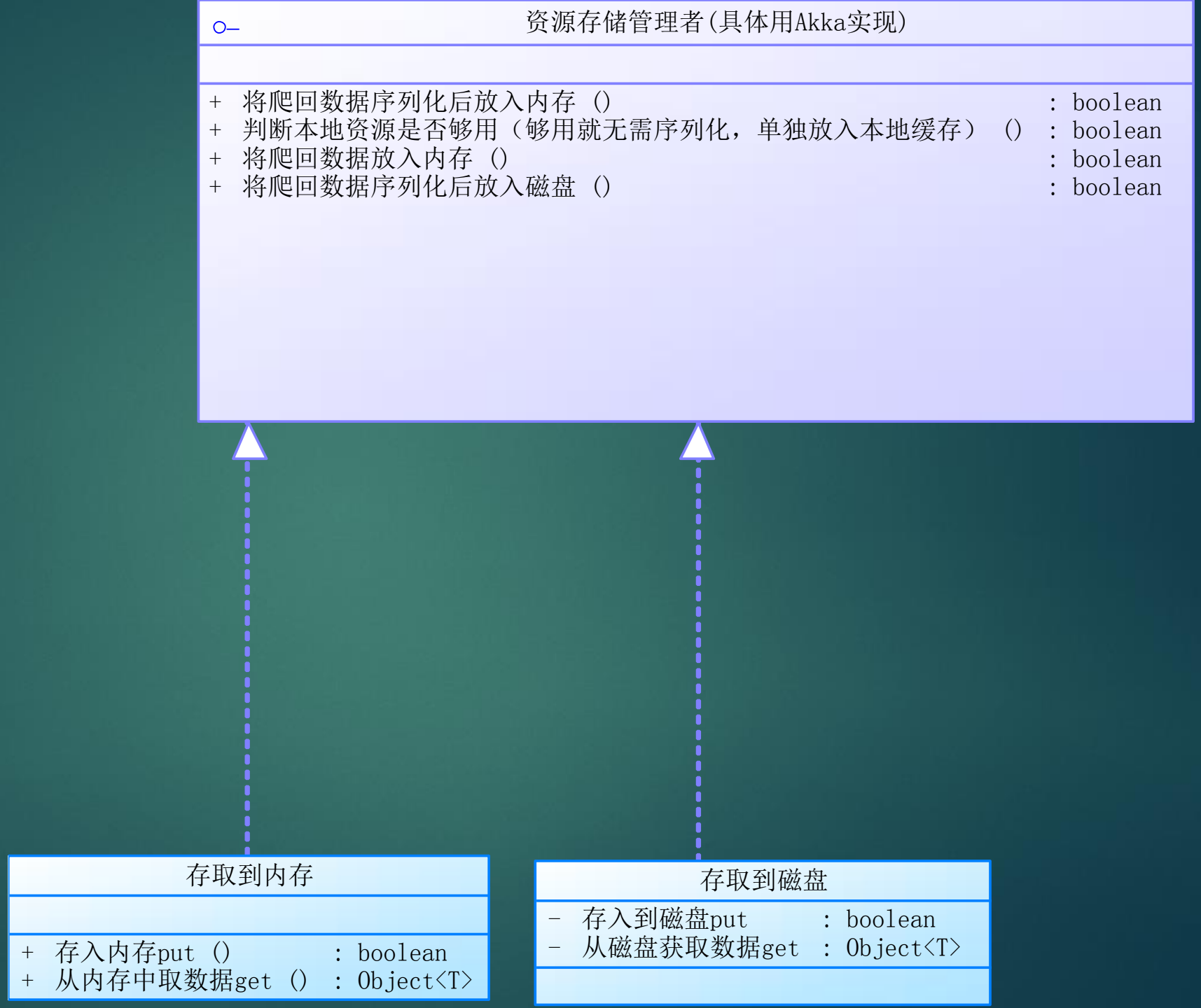
解析处理


Page
- url : String
- head : List
- content : Text
- createtime : Date
爬回数据解析抽象工厂
爬回数据解析接口（为具体的线程任务，使用线程池）
+ 解析 () : void
解析引擎Actor
+ 解析 (String 任务域名) : boolean
网页内容解析经纪人
+ 向zookeeper报告自己的资源信息(守护线程定时跑) () : void
+ 根据zookeeper Watch，当有触发解析内容时候，判断自己资源，然后批量取得待解析对象,优先本地 () : HttpResponse
+ 将ParseData放入线程池进行多线程解析分析 () : void
URL解析抽象类
- Page : Page
- URL实体bean : URL实体bean
- 分析更新策略抽象类 : 分析更新策略抽象类
- 网页内容提取分析接口 : List
+ 解析网页 () : int
+ URL入任务池接口 () : int
+ URL入库接口 () : int
+ URL规则分析接口 () : int
+ URL机器学习所需数据入库 () : int
API解析抽象类
域解析规则接口
+ 解析 () : boolean
大众点评解析
美团外卖解析
爬取数据代理类抽象工厂
网页内容提取分析接口
Html解析处理实现
爬虫解析New
Xpath解析
DOM解析
Jsoup解析


解析处理（New）
ParserManagerFactory
+ create () : ParserManager
存储管理器（StorageManager）
+ getByKey (String key) : T
+ listByKey (String key) : List<T>
+ putByKey (String key, T value) : Boolean
+ put (T entity) : Boolean
URL及内容抽取
爬虫解析线程（ParserRunner）
- rowkey : String
- parserManager : ParserManager
URLAssemblerFactory
+ ceate () : URLAssembler
ParserFactory
+ create (String 域名, int 爬虫类型) : Parser
StorageManagerFactory
+ create () : StorageManage
ITaskQueueFactory
+ create () : ITaskQueue
解析管理者（ParserManager）
{abstract}
- storageManager : StorageManager
- parser : Parser
ITaskFilterFactory
+ create () : ITaskFilter
URLDistinctManagerFactory
+ crate () : URLDistinctManager
AnalyzerFactory
+ create (String url) : Analyzer
解析者（Parser）
{abstract}
- 解析器(analyzer) : Analyzer
- Url过滤器（taskFilter） : ITaskFilter
- url组装器(urlAssembler) : URLAssembler
- url去重（urllDistinctManager） : URLDistinctManager
- 调度器（ITaskQueue） : ITaskQueue
+ parse () : void
分析器（Analyzer）
+ generateDocment (String url) : Document
+ extractLinks () : List<Link>
+ extractContent () : Obj
过滤规则接口
+ 添加规则 ()
+ 删除规则 ()
+ 匹配规则 (String url)
Link
- anchor : String
- url : String
正则规则
- 正例规则 : List
- 反例规则 : List
Url去重（URLDistinctManager）
+ distinct (String url) : String
组装器（URLAssembler）
+ 组装(assembler) (CrawlerTaskEntity task) : CrawlerTaskEntity
通用爬虫（GeneralParser）
主题爬虫（TopicParser）
{abstract}


Soldier
+ xpath (String xpath) : Soldier
+ $ (String soldier) : Soldier
+ $ (String soldier, String attName) : Soldier
+ css (String soldier) : Soldier
+ css (String soldier, String attrName) : Soldier
+ links () : Soldier
+ regex (String regex) : Soldier
+ regex (String regex, int group) : Soldier
+ value () : String
+ match () : Boolean
+ list () : List<String>
+ nodes () : List<Soldier>
+ all () : List<String>
+ search (TextMonster monster) : Soldier
+ searchList (TextMonster monster) : Soldier
TextMonster
+ search (String text) : String
+ searchList (String text) : List<String>
ElementMonster
+ search (Element element) : String
+ searchList (Element element) : List<Stri
FatherSoldier
{abstract}
HtmlSoldier
TextSoldier
RegexMonster
FatherMonster
{abstract}
CssMonster
XpathMonster
爬虫网页抽取

存储管理


资源存储管理者(具体用Akka实现)
+ 将爬回数据序列化后放入内存 () : boolean
+ 判断本地资源是否够用（够用就无需序列化，单独放入本地缓存） () : boolean
+ 将爬回数据放入内存 () : boolean
+ 将爬回数据序列化后放入磁盘 () : boolean
存取到内存
+ 存入内存put () : boolean
+ 从内存中取数据get () : Object<T>
存取到磁盘
- 存入到磁盘put : boolean
- 从磁盘获取数据get : Object<T>

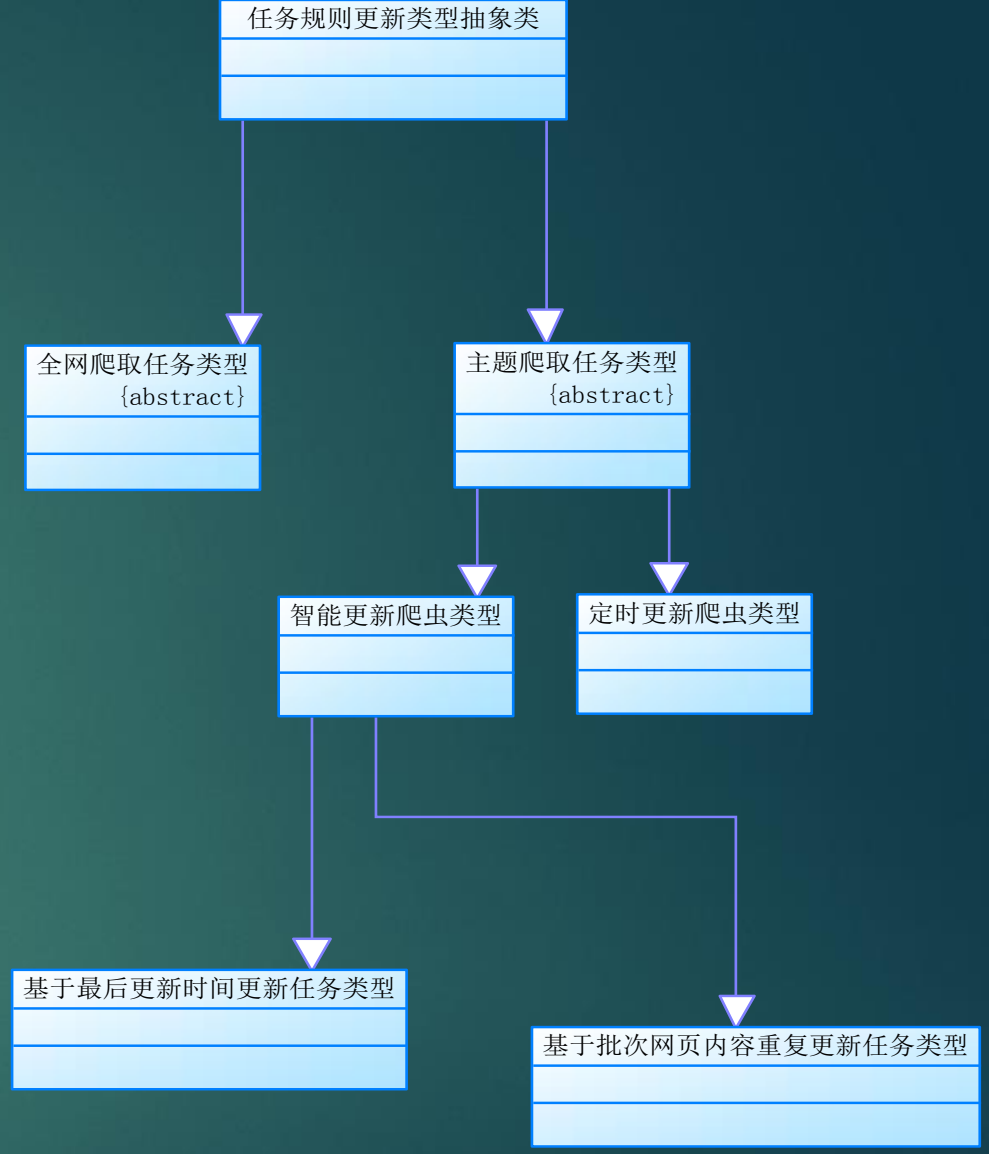
爬取对象生成器

○— 爬虫url生成器(是对爬取所需资源的一种算子抽象)
+ 获取下一个URI () : CrawlerTaskEntity

规则更新

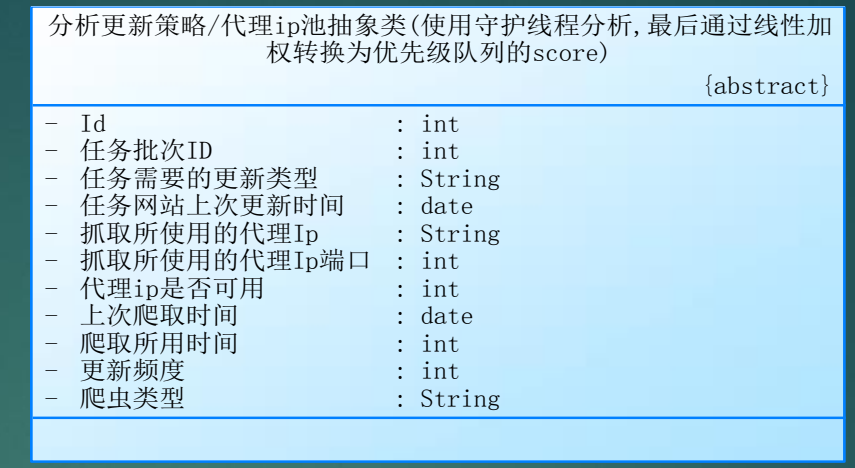
(守护线程Main)获得历史数据，通过机器学习建模更新任务队列score
- 数据池 : AnalyseEntity
+ 建模到score () : Double

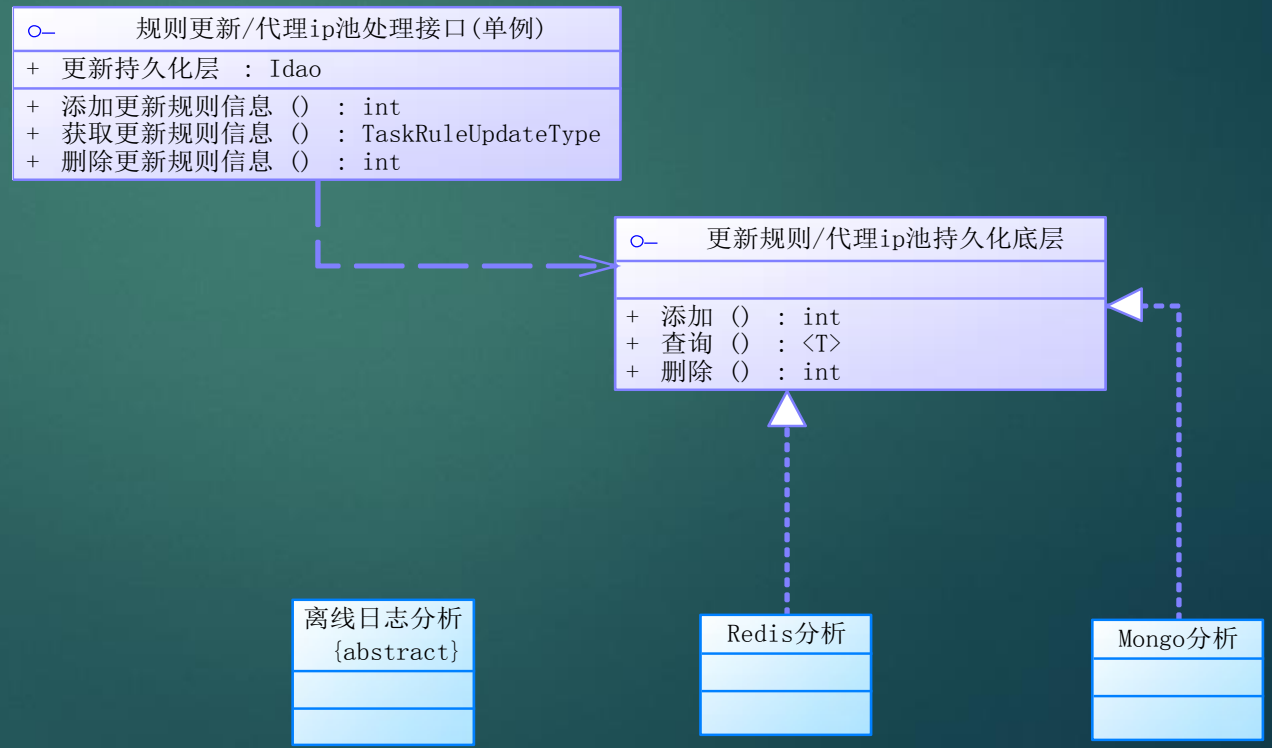
规则更新类型（枚举）	
- 无需类型	: int
- 基于最后更新时间更新任务类型（机器学习）	: int
- 基于批次网页内容重复更新任务类型（机器学习）	: int
- 定时更新爬虫类型	: int



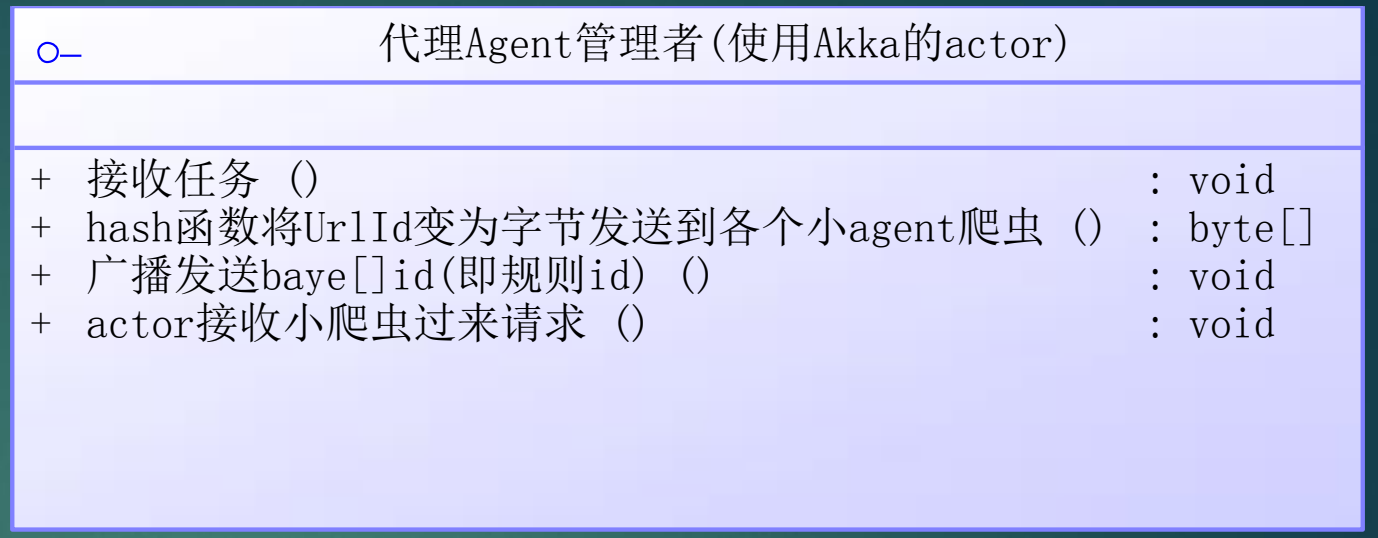
规则更新类型创建工厂
+ 创建伪任务规则更新类型 () : TaskRuleUpdateType

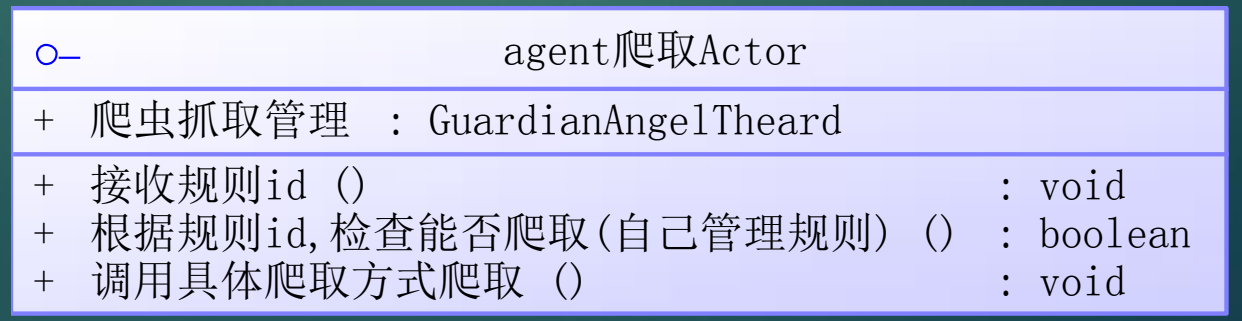
机器学习数据积累


分析更新策略/代理ip池抽象类(使用守护线程分析,最后通过线性加权转换为优先级队列的score)
{abstract}
- Id : int
- 任务批次ID : int
- 任务需要的更新类型 : String
- 任务网站上次更新时间 : date
- 抓取所使用的代理Ip : String
- 抓取所使用的代理Ip端口 : int
- 代理ip是否可用 : int
- 上次爬取时间 : date
- 爬取所用时间 : int
- 更新频度 : int
- 爬虫类型 : String

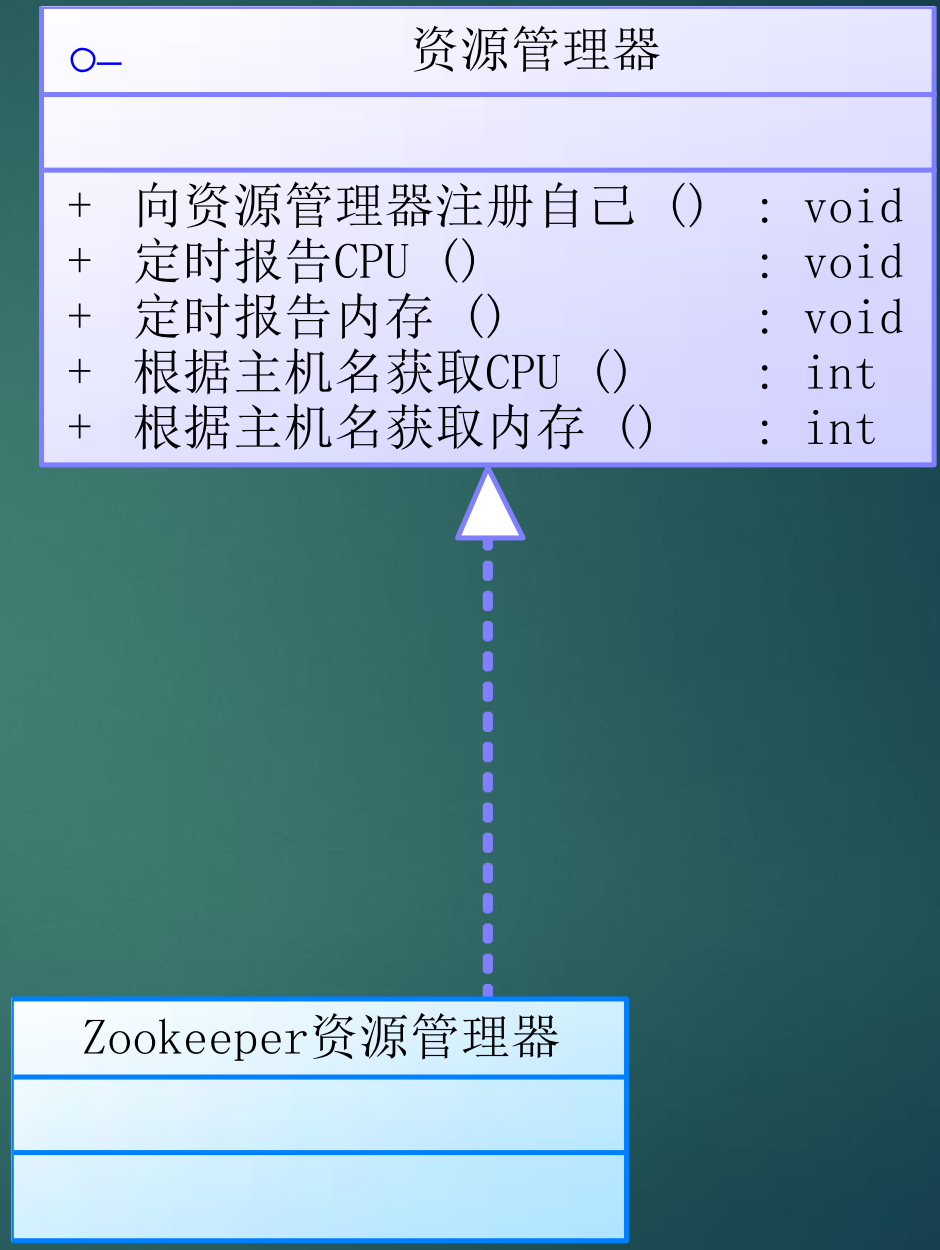

规则更新/代理ip池处理接口(单例)
+ 更新持久化层 : Idao
+ 添加更新规则信息 () : int
+ 获取更新规则信息 () : TaskRuleUpdateType
+ 删除更新规则信息 () : int
更新规则/代理ip池持久化底层
+ 添加 () : int
+ 查询 () : <T>
+ 删除 () : int
离线日志分析
{abstract}
Redis分析
Mongo分析

爬虫代理资源池

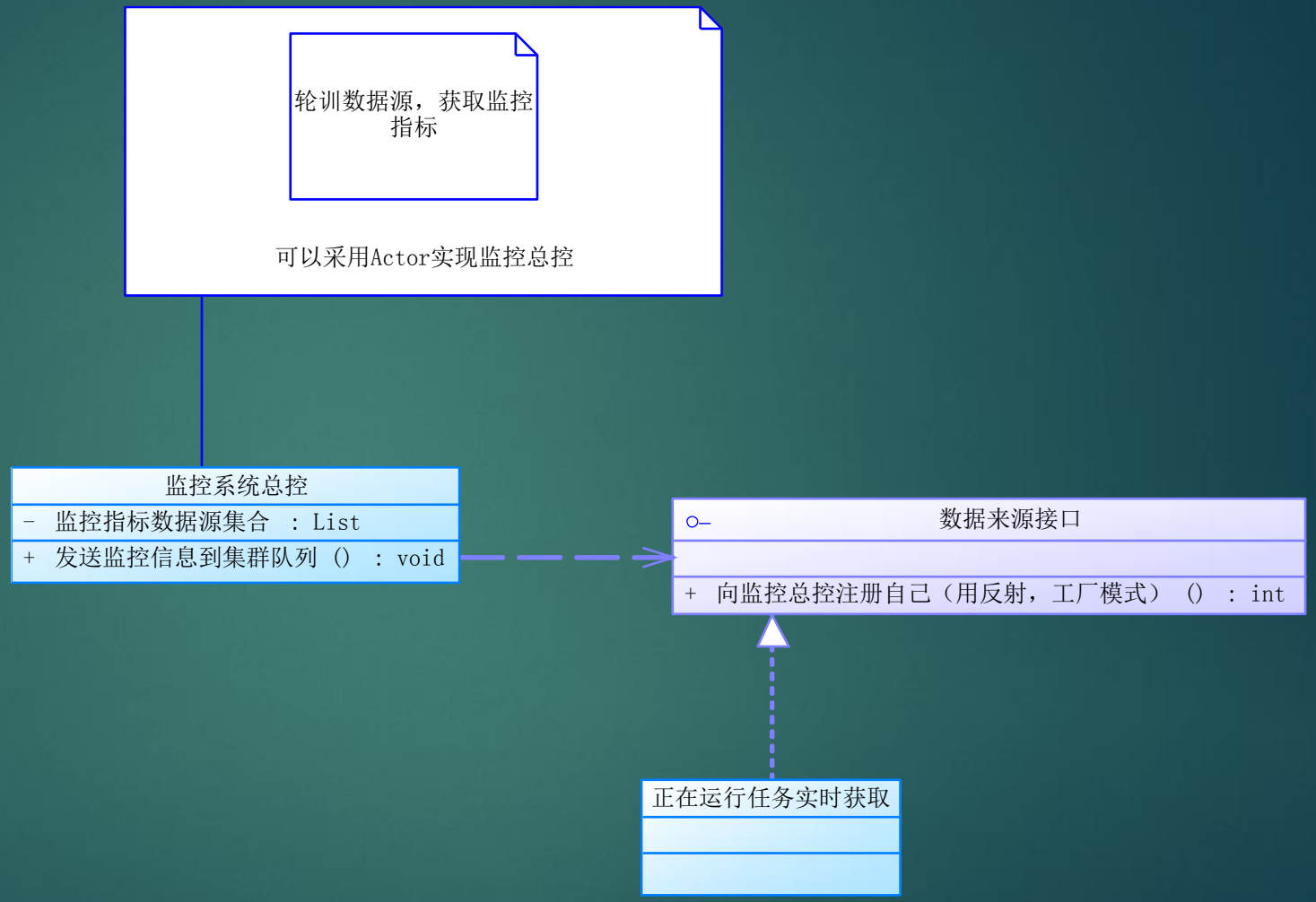

代理Agent管理者(使用Akka的actor)
+ 接收任务 () : void
+ hash函数将UrlId变为字节发送到各个小agent爬虫 () : byte[]
+ 广播发送baye[]id(即规则id) () : void
+ actor接收小爬虫过来请求 () : void


agent爬取Actor
+ 爬虫抓取管理 : GuardianAngelTheard
+ 接收规则id () : void
+ 根据规则id,检查能否爬取(自己管理规则) () : boolean
+ 调用具体爬取方式爬取 () : void

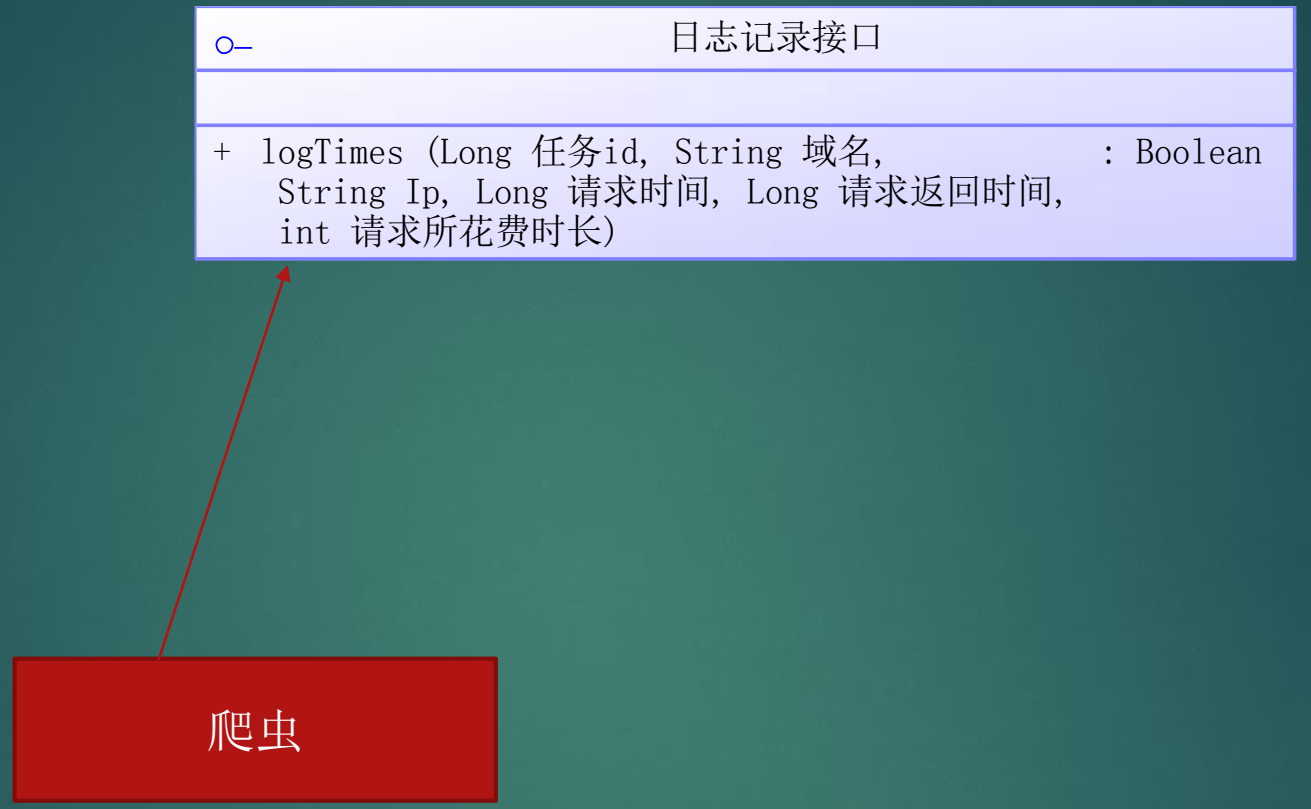
资源管理器


资源管理器
+ 向资源管理器注册自己 () : void
+ 定时报告CPU () : void
+ 定时报告内存 () : void
+ 根据主机名获取CPU () : int
+ 根据主机名获取内存 () : int
Zookeeper资源管理器

监控


轮训数据源，获取监控指标
可以采用Actor实现监控总控
监控系统总控
- 监控指标数据源集合 : List
+ 发送监控信息到集群队列 () : void
数据来源接口
+ 向监控总控注册自己（用反射，工厂模式） () : int
正在运行任务实时获取

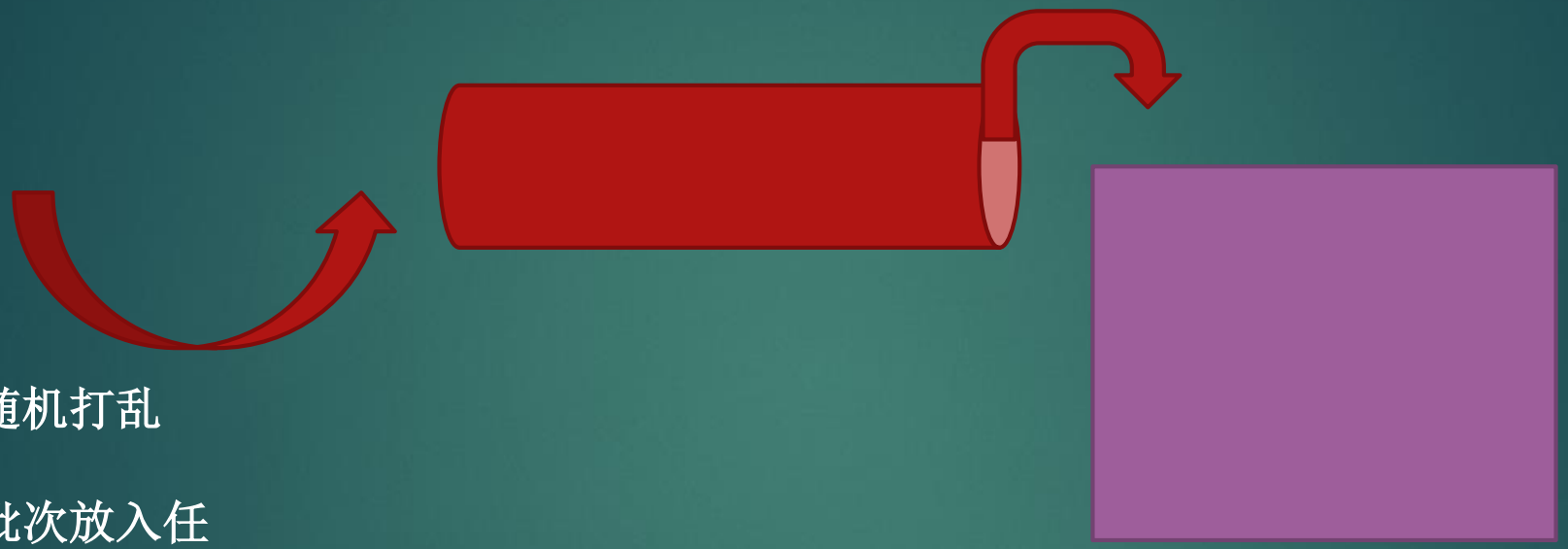
日志记录模块


日志记录接口
+ logTimes (Long 任务id, String 域名, : Boolean
String Ip, Long 请求时间, Long 请求返回时间,
int 请求所花费时长)
爬虫

美团外卖爬取（二次开发方式）

因为美团外卖url是序列递增的,可以借助这一规则穷举爬取得到餐厅信息，但是得不到电话号码
 所以得到餐厅信息和地址信息后可以再去百度外卖手机和大众点评网站去爬

当输入的序列号不存在的时候，会被美团外卖重定向，而如果ip被封了也会被重定向，怎么解决



将序列号随机打乱

并定时分批次放入任务队列

子爬虫

子爬虫在每次请求获得302重定向code后，定义一个全局线程安全的连续302记录器，如果超过某一阀值则认为为被封，否则认为为美团的正常餐厅下线

同时，记录下所有302所对应的url和子爬虫ip地址以及我们认为的被封的状态，为什么这么做？

因为我们虽然随机为正态分布，但是也有可能误判，我们重新拿被认为被封的url去为爬取过该url或者过了某一定时间后再去爬

美团外卖获取城市和经纬度策略

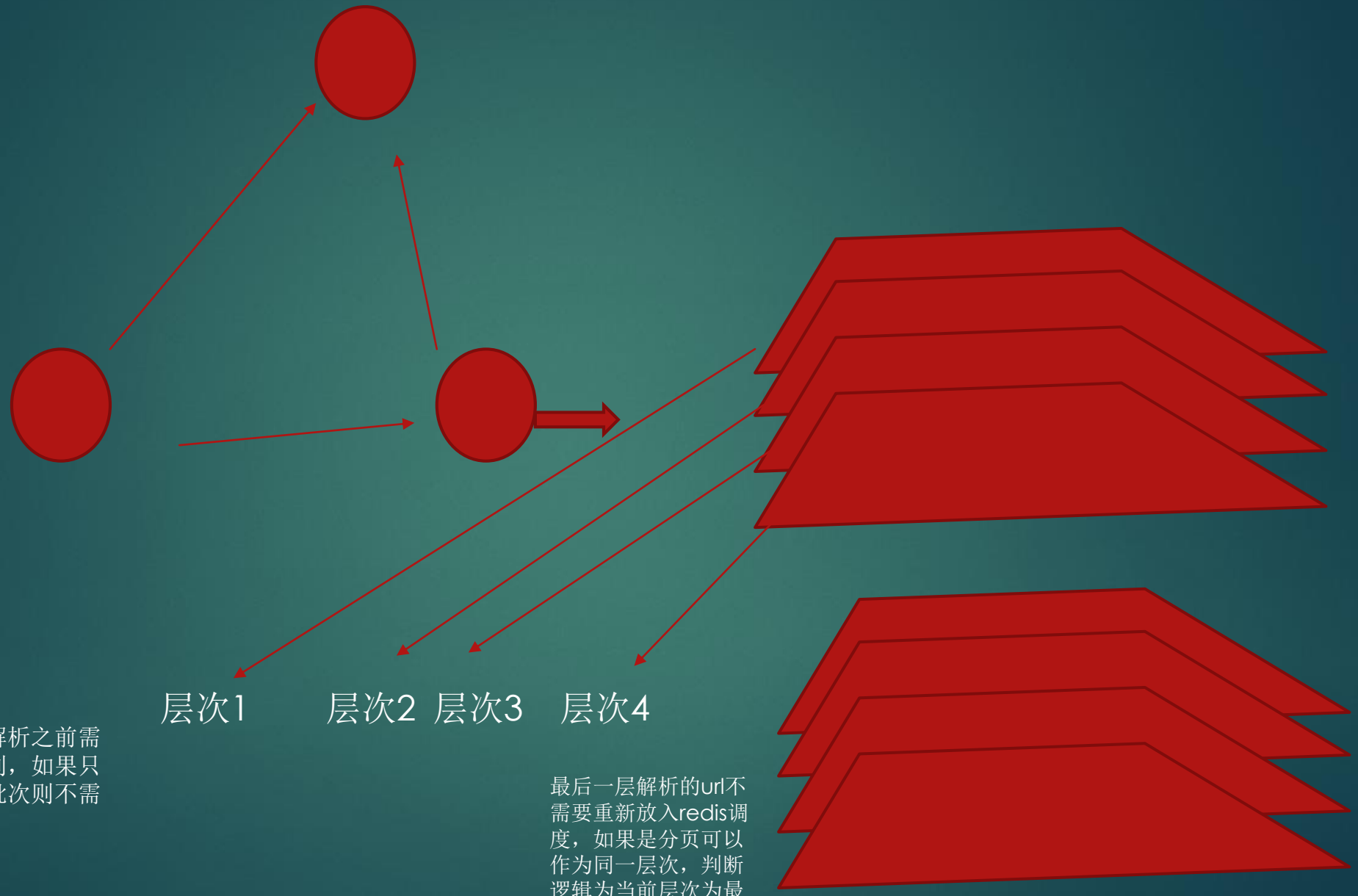
首先爬取一遍美团网，得到地址和餐厅名字

然后通过地址名字去搜狗爬取该地址所对应的经纬度
http://www.gpsspg.com/apps/maps/baidu_140802.htm

或者直接请求百度api
http://api.map.baidu.com/geocoder/v2/?address="浦东大道"&output=json&ak=UW8gadYahcUD3etpegMrsVhW
得到具体城市

然后再通过餐厅名在百度手机外卖和大众点评反搜得到餐厅的其他信息，比如电话号码

任务层次批次

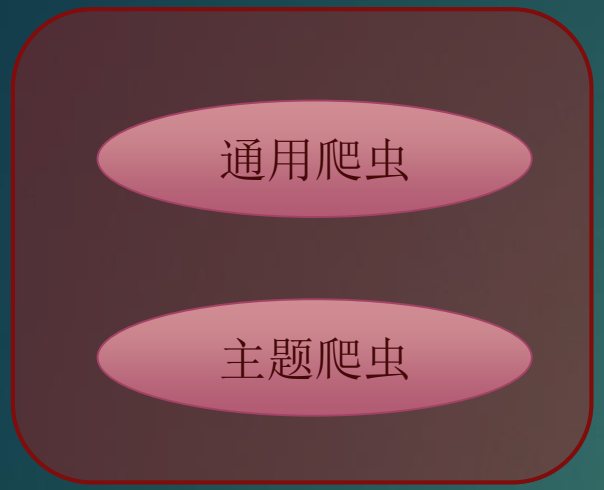


第一层在进行解析之前需要重新放入队列，如果只需要爬取一个批次则不需要

最后一层解析的url不需要重新放入redis调度，如果是分页可以作为同一层次，判断逻辑为当前层次为最后层且需要被终止层次则不放入队列

Job管理

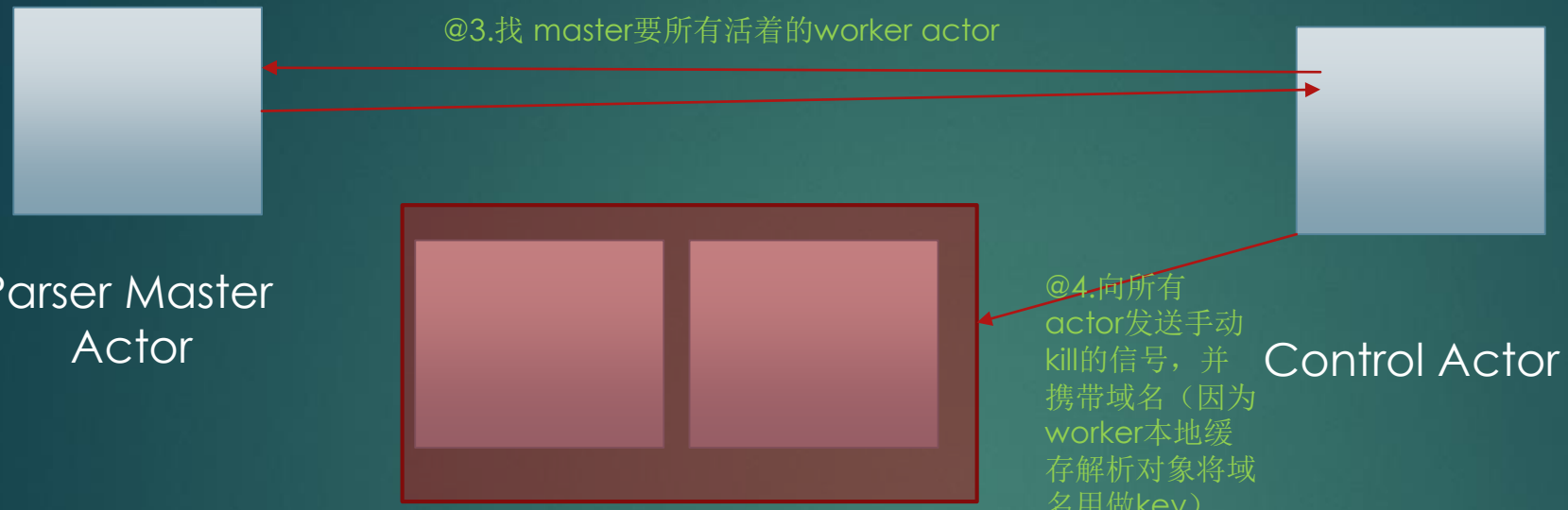
自动kill job



在ParseManager里面的start方法里面（后面）调用autoKillJob(ResOjb),根据CrawlerTaskEntity的爬虫类型判断、

1.如果是通用爬虫，则根据爬虫深度判断，为-1则不限制，否则，任务到此为止，不需要重新放入队列
2.如果是主题爬虫，则根据爬虫深度+批次进行判断，都为-1则不限制，否则任务到此为止，不需要重新放入队列

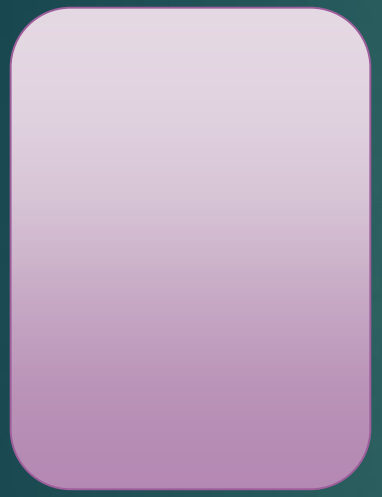
手动kill job



在ParseManager里面的start方法里面（后面）调用autoKillJob(ResOjb),根据CrawlerTaskEntity的爬虫类型判断、

1.如果是通用爬虫，则根据爬虫深度判断，为-1则不限制，否则，任务到此为止，不需要重新放入队列
2.如果是主题爬虫，则根据爬虫深度+批次进行判断，都为-1则不限制，否则任务到此为止，不需要重新放入队列

Schema抽取



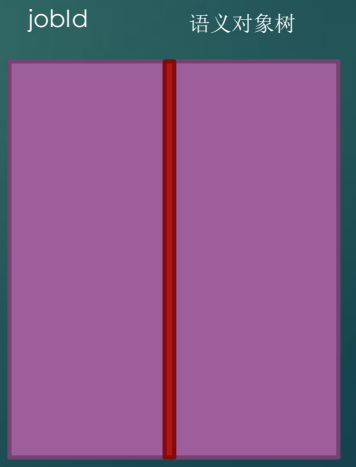
pull

Controller Actor

接收scheme并生成scheme解析语义树

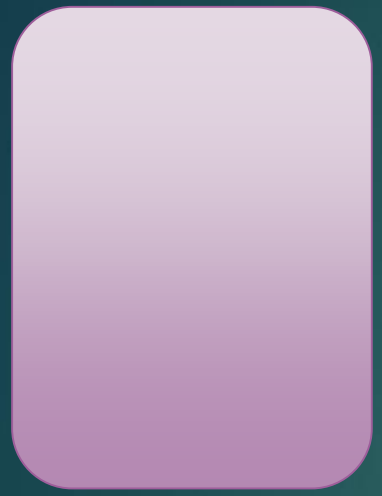
ParserWorker Actor

解析的时候先从本地缓存取，取不到到Controller取，本地缓存一天取一次



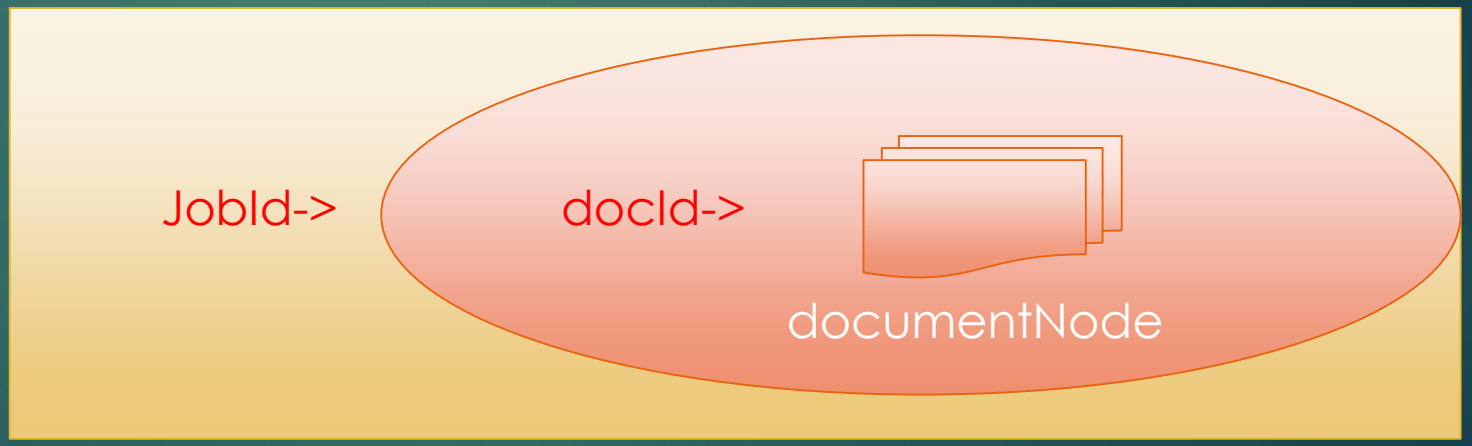
Map

Parser Worker缓存数据结构



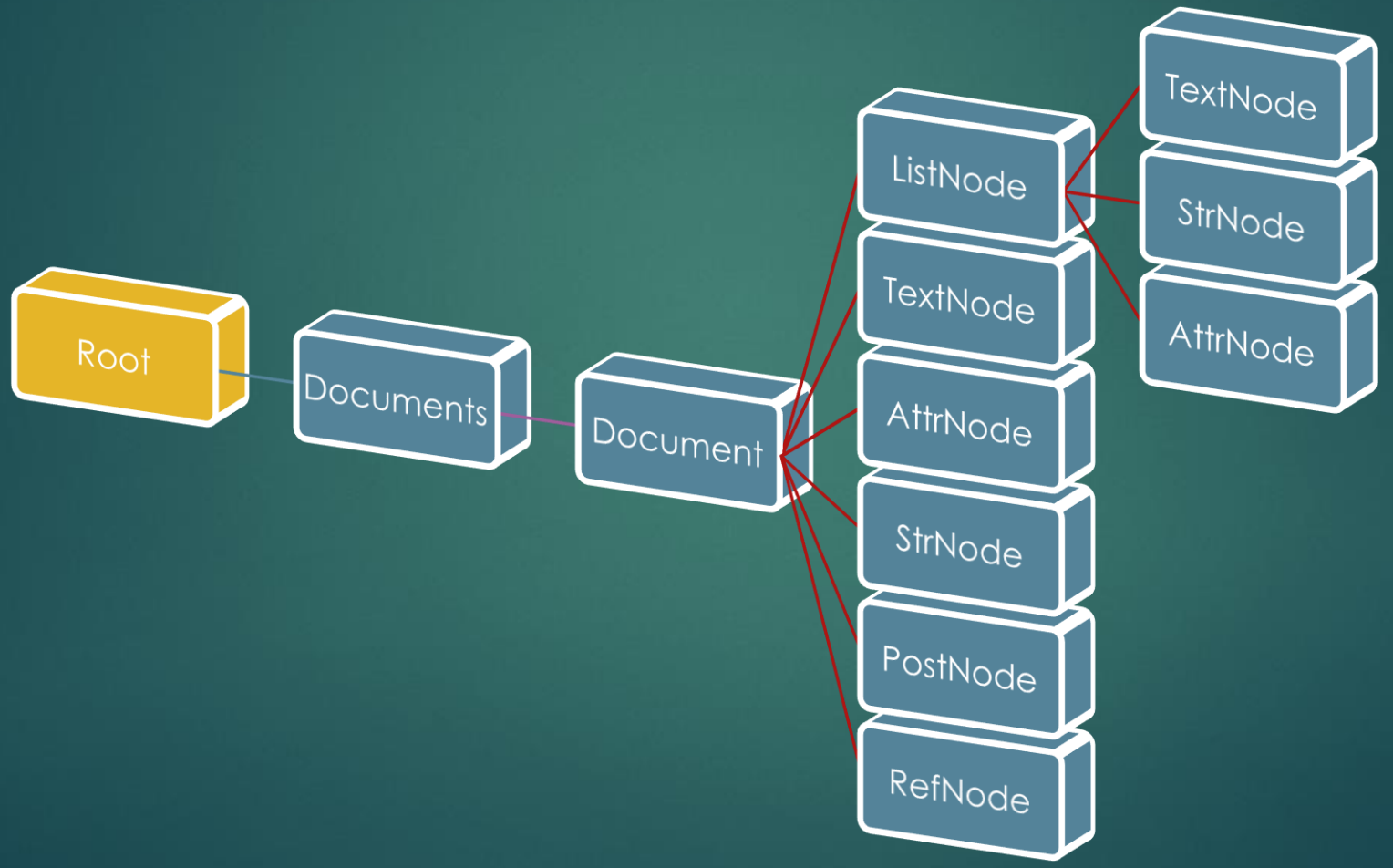
ParserWorker

Map(jobid->Map(docId->DocumentNode))



SemanticTree

pipeline



docTree逻辑

通过语义树进行内容解析后(exec())

1. 如果语义树解析完内容为空，则说明该docNode没内容解析，则直接使用引用(递归)该docNode的docNode的docId重新取得被应用的语义树

2. 根据树节点逻辑进行处理，并根据page或者pageSize判断是否需要分页，请求支持post,get，解析支持Json,html

3. Schema支持正则，xpath表达式混合编写

Sheme样例

```
<!--一个doc对应一个网页,url表示网页-->
<doc docid="dipingcity"
     url="http://www.dianping.com/ajax/json/index/citylist/getCitylist?do=allCitylist&amp;_nr_force=1439875530364"
     method="get" firstdoc="true">

    <list jsonselector="msg->html" name="result" notsave="true">
        <text name="cityid" parameter="true" seprator="|" index="3" notsave="true"/>
        <!-- jsonselector="cityid"  you can omit it ,default we choose name as jsonselector, also you can use regex el-->
    </list>

    <ref docid="dipingnavcity"/>
</doc>

<!--
 http://www.dianping.com/search/category/${cityid}/10#nav-tab|0|1
 http://www.dianping.com/search/category/${cityid}/10#nav-tab%7c0%7c1
 -->
<doc docid="dipingnavcity" url="http://www.dianping.com/search/category/${cityid}/10" method="get">
    <list selector="div.nav-category.nav-tabs div.nc-contain div#J_nt_items div#region-nav a" name="result"
          isfiled="false">
        <attr attr="href" name="href" parameter="true" notsave="true" filterregex="([\w/]+)#" filterGroup="1"/>
    </list>

    <text selector="body#top div.section.Fix div.bread.J_bread span.num" name="total" filterRegex="(\d+)"
          filterGroup="1"
          notsave="true"/>

    <ref docid="dipingsubnavcity"/>
</doc>

<doc docid="dipingsubnavcity" url="http://www.dianping.com${href}" method="get">
    <list selector="div.nav-category.nav-tabs div.nc-contain div#J_nt_items div#region-nav-sub a" name="result">
        <attr attr="href" name="href" parameter="true" notsave="true"/>
    </list>
    <text selector="body#top div.section.Fix div.bread.J_bread span.num" name="total" filterRegex="(\d+)"
          filterGroup="1"
          notsave="true"/>
    <!-- if there is no total item number, you also can use totalpagesize-->

    <ref docid="dipinglist"/>
</doc>

<doc docid="dipinglist" firsturl="http://www.dianping.com${href}"
     url="http://www.dianping.com${href}/p${page}?aid=bf3133f6cf5079c59b8c8b315bcca026dc74ae45e529cc10368c3562c4b17f2715152b6bdfb29834f8e761cbec7f47c8cc713db7df194db5557cff3892124763190b89756f9e4156a7f2016980b11045&amp;tc=2"
     method="get" page="true" maxPage="50" pagesize="15" firstpagesize="17">
    <list selector="div.content-wrap div.shop-wrap div.content div.shop-list.J_shop-list.shop-all-list ul li"
          name="result">
        <attr selector="div.pic a" name="image" attr="href" index="0"/>
        <!--index default 0 -->
        <text selector="div.txt div.tit a h4" name="title"/>
        <text selector="div.tag-addr a span" name="tag" index="0"/>
        <text selector="div.tag-addr a span" name="address" index="1"/>
        <!--default index=0 -->
        <attr attr="href" selector="div.txt div.tit a" name="href" parameter="true" notsave="true"/>
    </list>

    <text selector="body#top div.section.Fix div.bread.J_bread span.num" name="total" filterRegex="(\d+)"
          filterGroup="1"
          notsave="true"/>
    <ref docid="dipingdetail"/>
</doc>


<doc docid="dipingdetail" url="http://www.dianping.com${href}">
    <text selector="div.body-content.clearfix div.main div#basic-info p.expand-info.tel span.item" name="tel"
          type="multiple"
          seprator="|"/>
</doc>

<!--post test-->
<!--<doc docid="dipingdetail" url="http://www.dianping.com/${href}">
    <post>
        <str name="a" value="haha" notsave="true"/>   [default notsave=false ]
        <str name="title" notsave="true">${page}</str>
```

任务跟踪系统

```scala
/**
 * Created by soledede on 2015/9/17.
 */
trait TraceListener {
  def onJobStart(jobstart: JobStarted) = {}
  def onJobTaskFailed(jobTaskFailed: JobTaskFailed) = {}
  def onJobTaskCompleted(jobTaskCompleted: JobTaskCompleted) = {}
  def onJobTaskAdded(jobTaskAdded: JobTaskAdded) = {}
}
```

Listeners

Add listener

异步线程，实时执行，阻塞通过信号量通信

Add event

ListenerEvents

该job当前总任务数

该job当前已完成任务数

该job当前失败任务数

to

同步锁


```scala
/**
 * Created by soledede on 2015/9/17.
 */
sealed trait TraceListenerEvent

case class JobStarted(jobId: String, seedNum: Int) extends TraceListenerEvent

case class JobTaskFailed(jobId: String, num: Int) extends TraceListenerEvent

case class JobTaskCompleted(jobId: String, num: Int) extends TraceListenerEvent

case class JobTaskAdded(jobId: String, num: Int) extends TraceListenerEvent
```

监控系统

Metrix+Servlet+Ganglia+JMX