# 法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：大数据分析挖掘
  - 新浪微博：ChinaHadoop

# 分布式爬虫

# 大纲

- 分布式系统概述
- 主从服务设计
- Page Rank
- Url 优先级动态排序

# 分布式系统

# Deduce of Distributed System - I

- A *program*
  - is the code you write.
- A *process*
  - is what you get when you run it.
- A *message*
  - is used to communicate between processes.
- A *packet*
  - is a fragment of a message that might travel on a wire.
- A *protocol*
  - is a formal description of message formats and the rules that two processes must follow in order to exchange those messages.

# Distributed System - II

- A *network*
  - is the infrastructure that links computers, workstations, terminals, servers, etc. It consists of routers which are connected by communication links.
- A *component*
  - can be a process or any piece of hardware required to run a process, support communications between processes, store data, etc.
- A *distributed system*
  - is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.

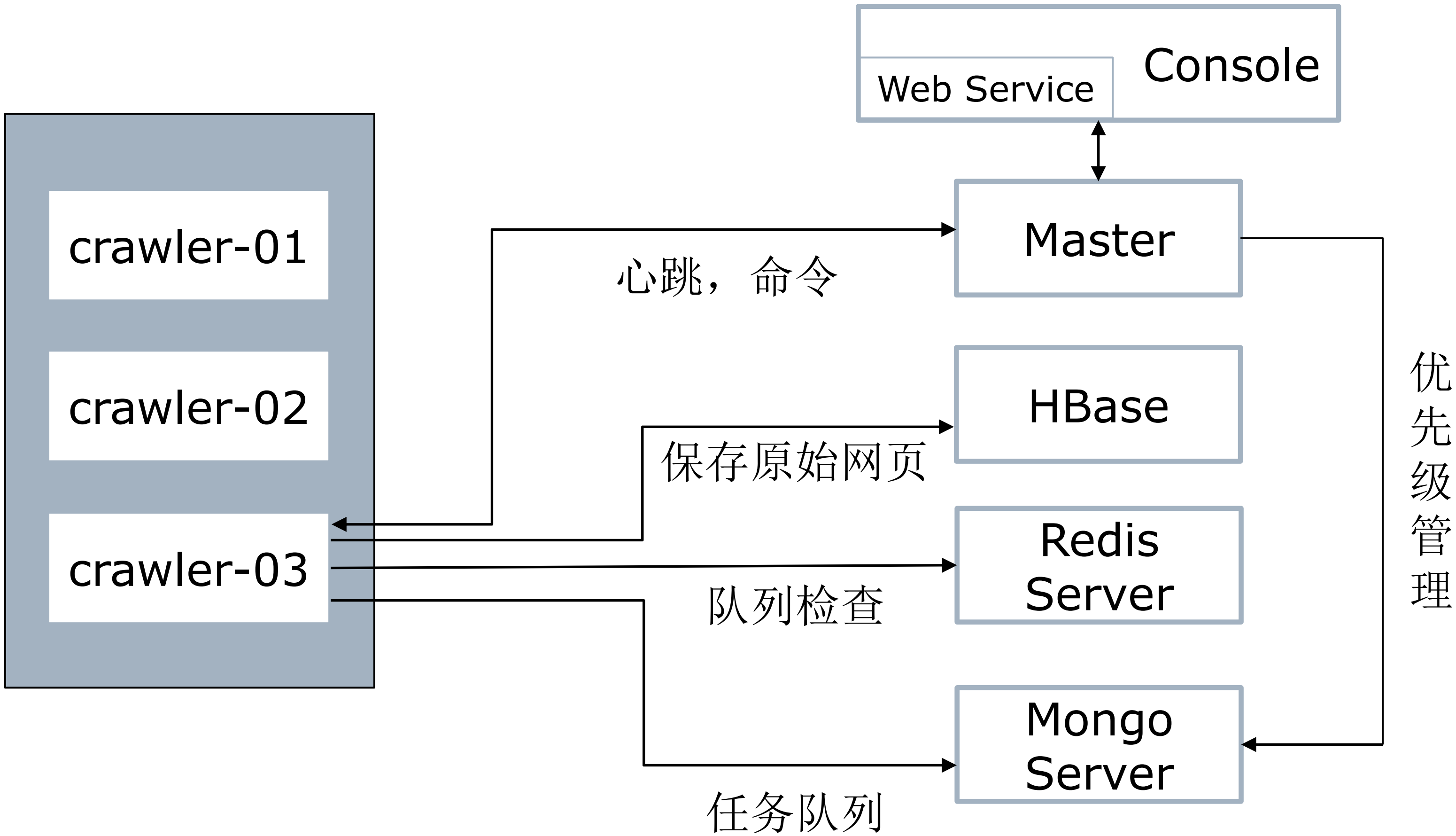
# Advantage

- Fault-Tolerant: It can recover from component failures without performing incorrect actions.
- Highly Available: It can restore operations, permitting it to resume providing services even when some components have failed.
- Recoverable: Failed components can restart themselves and rejoin the system, after the cause of failure has been repaired.
- Consistent: The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.
- Scalable: It can operate correctly even as some aspect of the system is scaled to a larger size.
- Predictable Performance: The ability to provide desired responsiveness in a timely manner.
- Secure: The system authenticates access to data and services

# Challenge

- Replications and migration cause need for ensuring consistency and distributed decision-making
- Failure modes: Not assuming data received is same as sent
- Concurrency: Update/Replication/Cache/Failure …
- Heterogeneity: Network, hardware, OS, languages, developers
- Scalability: Architecture must be able to handle increase of users, resources, etc. Considering cost of physical resources, performance loss, bottleneck
- Security:

# 分布式爬虫系统


Console
Web Service
Master
HBase
Redis Server
Mongo Server
crawler-01
crawler-02
crawler-03
心跳，命令
保存原始网页
队列检查
任务队列
优先级管理

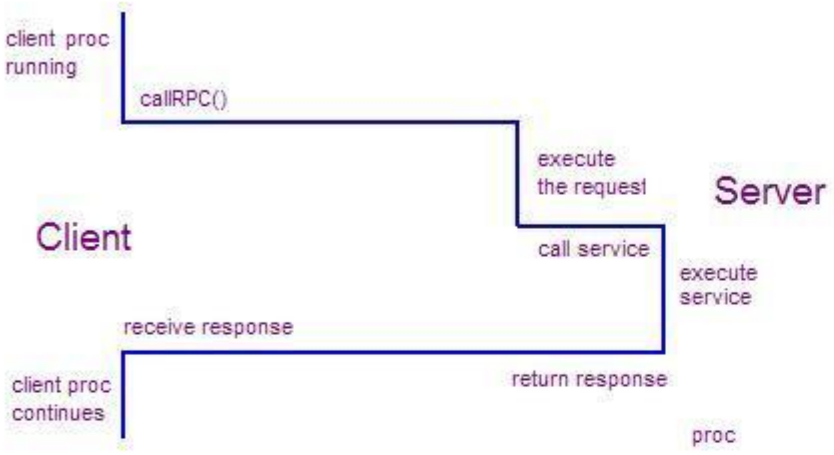
# Master-Slave 结构

# Master-Slave 结构

- 有一个主机，对所有的服务器进行管理。绝大多数分布式系统，都是 Master-Slave 的主从模式。而之前我们的爬虫，是完全独立的，依次从 url队列里获取url，进行抓取
- 当爬虫服务器多的时候，必须能通过一个中心节点对从节点进行管理
- 能对整体的爬取进行控制
- 爬虫之间信息共享的桥梁
- 负载控制

# Remote Procedure Calls

- Specifies the protocol for client-server communication
- Develops the client program
- Develops the server program



# Protocol – Message Type

```
# message type, REGISTER, UNREGISTER and HEARTBEAT
MSG_TYPE        = 'TYPE'
# send register
REGISTER        = 'REGISTER'
# unregister client with id assigned by master
UNREGISTER      = 'UNREGISTER'
# send heart beat to server with id
HEARTBEAT       = 'HEARTBEAT'
# notify master paused with id
PAUSED          = 'PAUSED'
# notify master resumed with id
RESUMED         = 'RESUMED'
# notify master resumed with id
SHUTDOWN        = 'SHUTDOWN'
```

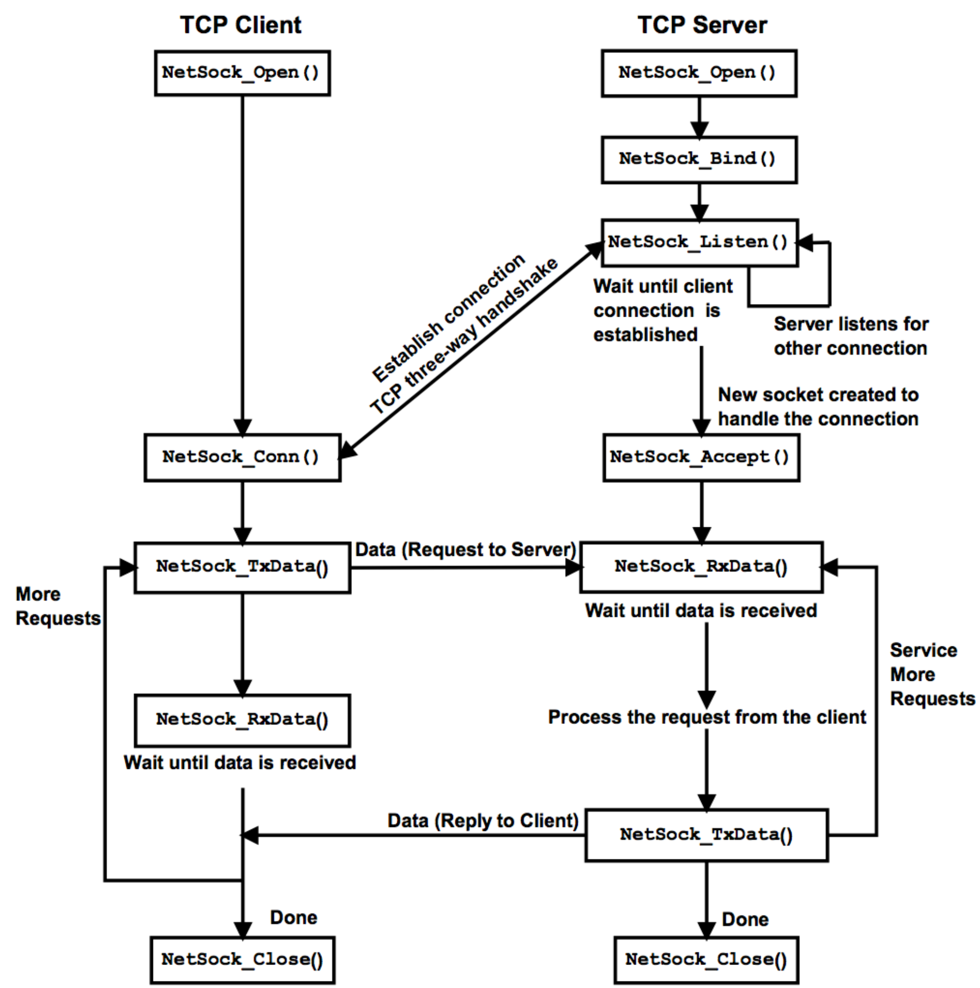
# Protocol - Actions

```
# server status key word
ACTION_REQUIRED      = 'ACTION_REQUIRED’
# server require pause
PAUSE_REQUIRED       = ‘PAUSE_REQUIRED’
# server require pause
RESUME_REQUIRED      = 'RESUME_REQUIRED’
# server require shutdown
SHUTDOWN_REQUIRED          = 'SHUTDOWN_REQUIRED'
```

# Protocol – Key Definition

```
# server status key word
SERVER_STATUS        = SERVER_STATUS
# client id key word
CLIENT_ID            = ‘CLIENT_ID’
# error key work
ERROR                = ERROR’
```

# Socket



TCP Client
TCP Server
NetSock_Open()
NetSock_Open()
NetSock_Bind()
NetSock_Listen()
Wait until client connection is established
Server listens for other connection
Establish connection
TCP three-way handshake
New socket created to handle the connection
NetSock_Conn()
NetSock_Accept()
NetSock_TxData()
Data (Request to Server)
NetSock_RxData()
More Requests
Wait until data is received
Service More Requests
NetSock_RxData()
Process the request from the client
Wait until data is received
Data (Reply to Client)
NetSock_TxData()
Done
Done
NetSock_Close()
NetSock_Close()

# Create Client Socket

```
#create an INET, STREAMing socket
s = socket.create_connection( socket.AF_INET,
                              socket.SOCK_STREAM)
```

AF_INET -- IPv4 Internet protocols

SOCK_STREAM, SOCK_DGRAM, SOCK_RAW -- socket types (SOCK_STREAM TCP, SOCK_DGRAM UDP)

# Create Server Socket

```
#create an INET, STREAMing socket
serversocket = socket.socket( socket.AF_INET,
                              socket.SOCK_STREAM)
#bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 20010))
#become a server socket
serversocket.listen(5)
```

listen(backlog) -- number of unaccepted connections that the system will allow before refusing new connections, at least 0

# Create Server Socket

```
while True:
        #accept connections from outside
         (clientsocket, address) = serversocket.accept()
        #now do something with the clientsocket
        #in this case, we'll pretend this is a threaded
        server ct = client_thread(clientsocket)
        ct.run()
```



小象学院 ChinaHadoop.cn

# Ways to listening

- a new thread to handle clientsocket
- a new process
- use non-blocking socket

# Non-blocking mode listening

- connection.setblocking(False),

  send, recv, connect and accept returns immediately

  connection.setblocking(False) is equivalent to settimeout(0.0)

- asyncore

  Provides the basic infrastructure for writing asynchronous

  socket service clients and servers.

| Event | Description |
|---|---|
| handle_connect() | Implied by the first read or write event |
| handle_close() | Implied by a read event with no data available |
| handle_accept() | Implied by a read event on a listening socket |

# Ways to end communication

- fixed length message: **while** totalsent < MSGLEN:
- delimited: some message**\0**
- indicates message length in beginning: LEN: 50;
- shutdown connection: server call close(), clietn recv() returns 0

# Page Rank

# 背景 – 搜索引擎

最早的搜索引擎采用的是 **分类目录**[^ref_1] 的方法，即通过人工进行网页分类并整理出高质量的网站。那时 Yahoo 和国内的 hao123 就是使用的这种方法。

后来网页越来越多，人工分类已经不现实了。搜索引擎进入了 **文本检索** 的时代，即计算用户查询关键词与网页内容的相关程度来返回搜索结果。这种方法突破了数量的限制，但是搜索结果不是很好。因为总有某些网页来回地倒腾某些关键词使自己的搜索排名靠前。

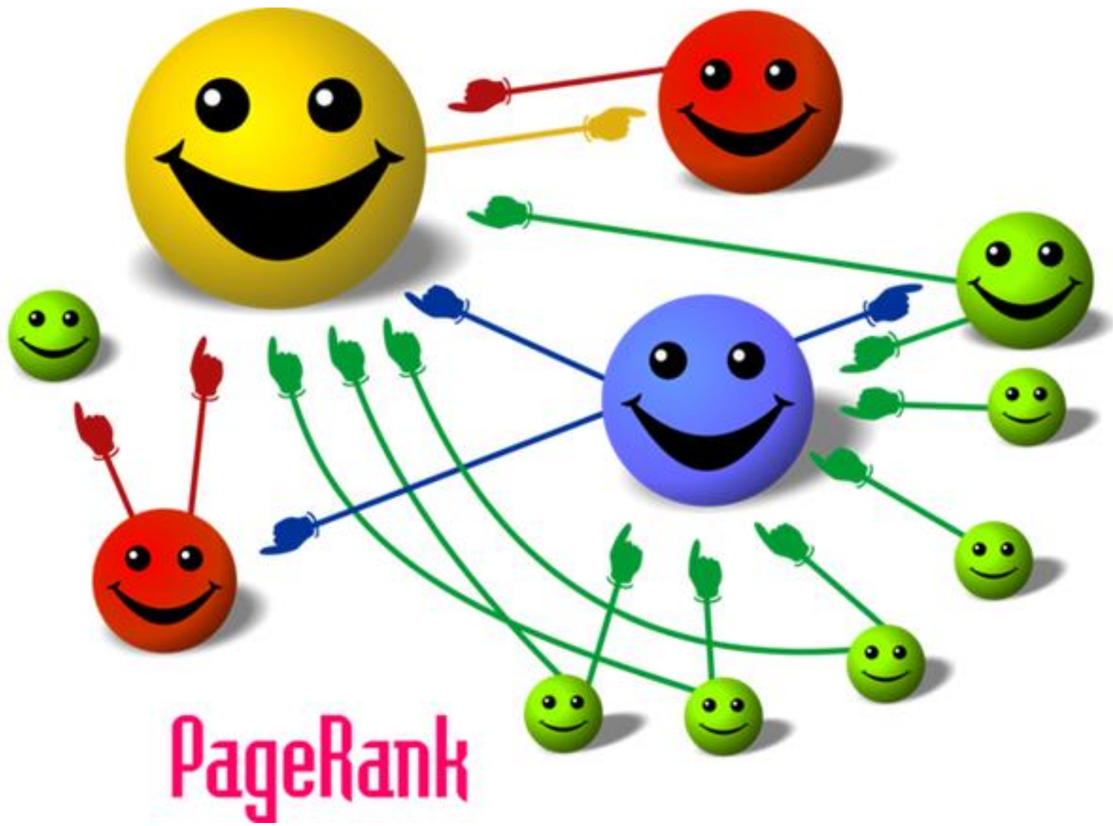
# 背景

- 网页数量急剧膨胀，用户需要有效搜索出有用的信息
- Google CEO 拉里佩奇提出的一种算法，来计算互联网里的网站的重要性，以对搜索进行排名
- PageRank 的计算量很大，因此诞生了 Map-Reduce，来分布式计算 PageRank
- PageRank 和 BigTable 是Google早期的核心

# 基本思想

- **数量假设：**在Web图模型中，如果一个页面节点接收到的其他网页指向的入链数量越多，那么这个页面越重要。
- **质量假设：**指向页面A的入链质量不同，质量高的页面会通过链接向其他页面传递更多的权重。所以越是质量高的页面指向页面A，则页面A越重要。

# 模型



# 模型

PageRank算法计算每一个网页的PageRank值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank就是估计这个悠闲的上网者分布在各个网页上的概率。

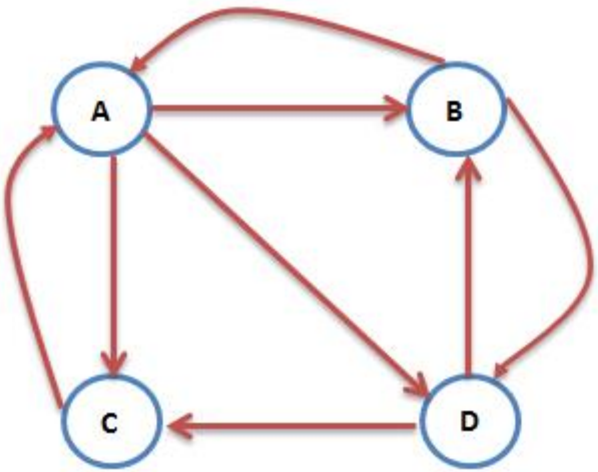
# 简单推导

A的PR值就可以表示为

PR(A)=PR(B)+PR(C)

B和D都不止有一条出链，例如从B网页打开A和C是同概率：

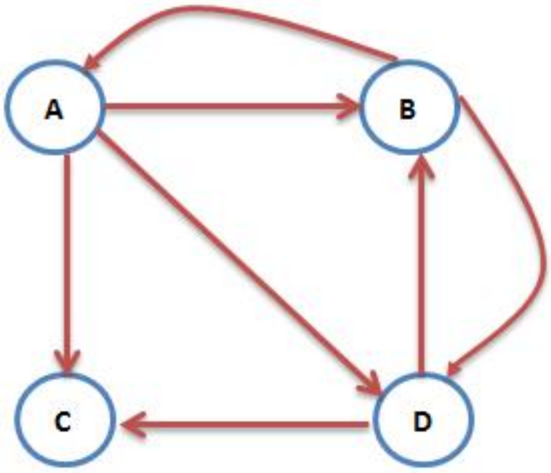
$$PR(A)=\frac{PR(B)}{2}+\frac{PR(C)}{1}$$



# 简单推导

对于没有出链的网页（例如C），设定它对所有网页都有出链

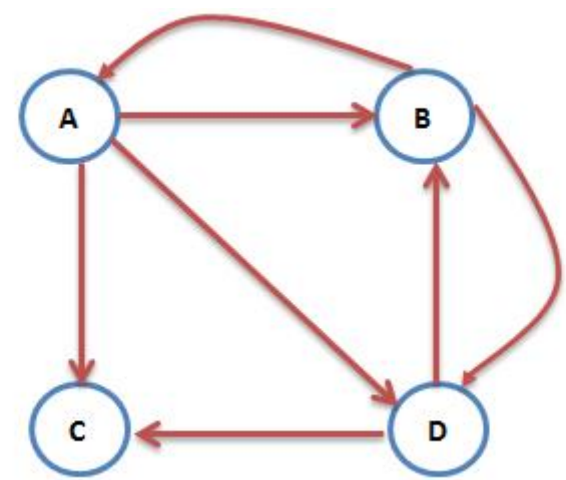
$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{4}$$



# 简单推导

用户在使用的时候，存在一种可能性，就是停止内链跳转，而是直接地址栏输入新的地址跳转，假设整体的概率是$a$，那么所有的PR值需要乘以$a$；同时，任意网页也存在随机调到这个网页的可能性。这主要解决了某些网页没有出链的情况

$$PR(A) = a\left(\frac{PR(B)}{2}\right) + \frac{(1-a)}{4}$$



# 简单推导

$$PR(A) = \alpha\left(\frac{PR(B)}{2}\right) + \frac{(1-\alpha)}{4}$$

把上面的公式推广出来，就成为了PageRank的计算方法

$$PR(p_i) = \alpha \sum_{p_j \in M_{p_i}} \frac{PR(p_j)}{L(p_j)} + \frac{(1-\alpha)}{N}$$



小象学院 ChinaHadoop.cn

# 迭代计算PR

用矩阵来表示，就是这样的一个线性代数的等式：

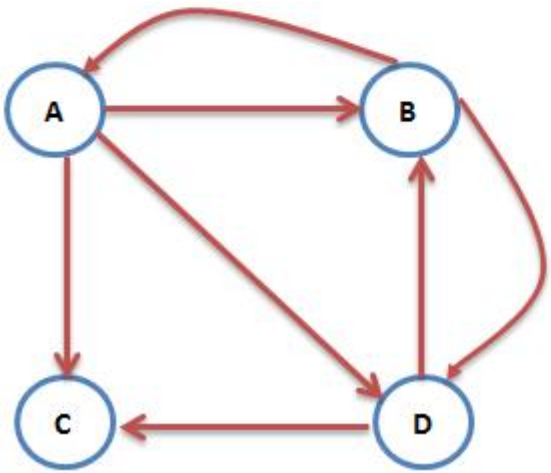
$$\mathbf{R} = \begin{bmatrix} (1-q)/N \\ (1-q)/N \\ \vdots \\ (1-q)/N \end{bmatrix} + q \begin{bmatrix} \ell(p_1,p_1) & \ell(p_1,p_2) & \cdots & \ell(p_1,p_N) \\ \ell(p_2,p_1) & \ddots & & \\ \vdots & & \ell(p_i,p_j) & \\ \ell(p_N,p_1) & & & \ell(p_N,p_N) \end{bmatrix} \mathbf{R}$$

$$\mathbf{R} = \begin{bmatrix} \mathrm{PageRank}(p_1) \\ \mathrm{PageRank}(p_2) \\ \vdots \\ \mathrm{PageRank}(p_N) \end{bmatrix}$$

其中R是PR值的特征向量，*而* $\varepsilon(p_1, p_2)$*的定义为：*

$$\begin{cases} \sum_{j=1}^{n} \varepsilon(p_i, p_j) = 1 \\ \varepsilon(p_1, p_2) = 0, 1 \nrightarrow 2 \end{cases}$$

# 迭代计算PR



# PageRank 算法优缺点

- 优点：
  - 一个与查询无关的静态算法，所有网页的PageRank值通过离线计算获得；有效减少在线查询时的计算量，极大降低了查询响应时间
- 缺点：
  - 人们的查询具有主题特征，PageRank忽略了主题相关性，导致结果的相关性和主题性降低
  - 旧的页面等级会比新页面高。因为即使是非常好的新页面也不会有很多上游链接，除非它是某个站点的子站点。

# Python 的 PageRank - NetworkX

```
pip install networkx

g = nx.DiGraph() # 构造有向图

g.add_node(url)
g.add_edge(src, dest) # 添加边
nx.pagerank(g, 0.9) # 计算pagerank，g为有向图，0.9是PR的
                    # 随机跳转概率（也称为阻尼系数）
```

# 动态排序过程

# 数据库架构

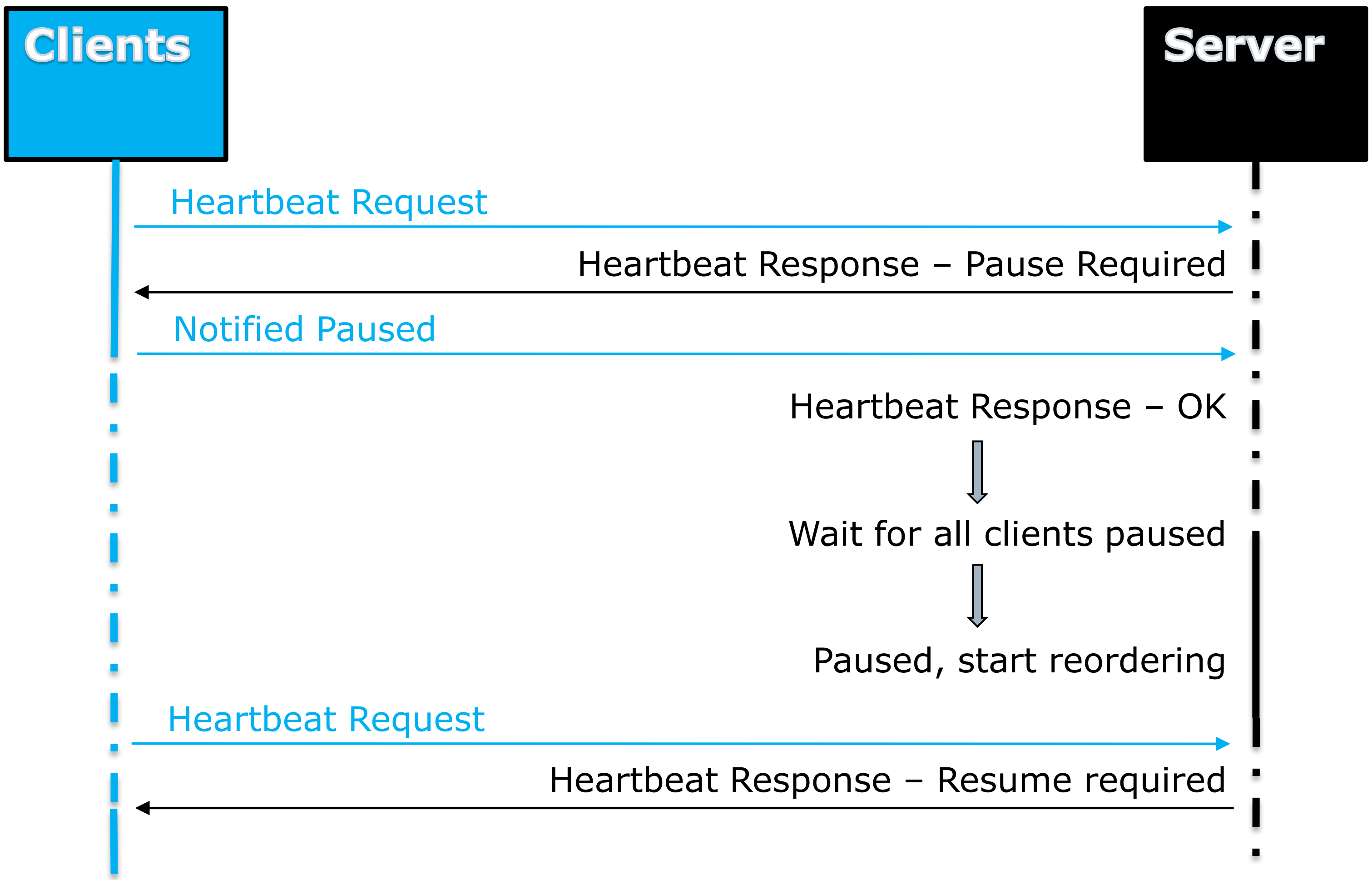
collection urlqueue

| _id | url | pr | status | done_time | queue_time |
|---|---|---|---|---|---|
| md5(url) | url | pangerank | new<br>downloading<br>done | time finished downloading | time added to the queue |

collection urlpr

| _id | url | links |
|---|---|---|
| md5(url) | url | all external links |

# 流程


Clients
Server
Heartbeat Request
Heartbeat Response – Pause Required
Notified Paused
Heartbeat Response – OK
Wait for all clients paused
Paused, start reordering
Heartbeat Request
Heartbeat Response – Resume required

# 流程

1. 主从服务器始终维持心跳
2. 根据重拍条件（queue的size、定时等），启动重拍流程
3. 通知爬虫，暂停爬取
4. 爬虫在心跳回复中收到暂停通知，暂停爬取并通知主机
5. 主机等待所有爬虫暂停
6. 主机开始重排网页
7. 重排结束，设置标志位
8. 心跳回复收到恢复指令，继续爬取

# 主要通信协议

1. 注册
2. 暂停、恢复爬取
3. 终止爬虫程序
4. 错误通知
5. 状态同步

# Master 及 Slave 工作

Master

1. 管理爬虫
2. 动态重拍
3. 间隔地状态检查

Slave

1. 注册
2. 获取并执行命令
3. 同步状态
4. 爬取网页，保存到各个分布式数据库

# 疑问

- 问题答疑：　　http://www.xxwenda.com/
  - 可邀请老师或者其他人回答问题

# 联系我们

## 小象学院：互联网新技术在线教育领航者

- 微信公众号：大数据分析挖掘
- 新浪微博：ChinaHadoop