WebCrawler

闵卫

版本：1.0.0.1

2015年11月20日 星期五

目录

Table of contents

继承关系索引

类继承关系

此继承关系列表按字典顺序粗略的排序:

Configurator 13
Hash 33
HttpHeader 155
HttpResponse 41
Log 44
MultiIo 52
Plugin 56
    DomainLimit 29
    HeaderFilter 38
    MaxDepth 49
    SaveHTMLToFile 73
    SaveImageToFile 77
PluginMngr 59
RawUrl 65
    DnsUrl 24
Socket 84
StrKit 94
Thread 98
    DnsThread 20
    RecvThread 69
    SendThread 82
UrlFilter 102
    BloomFilter 6
UrlQueues 104
WebCrawler 119

类索引

类列表

这里列出了所有类、结构、联合以及接口定义等，并附带简要说明:

BloomFilter (布隆过滤器) 6
Configurator (配置器) 13
DnsThread (域名解析线程) 20
DnsUrl (解析统一资源定位符) 24
DomainLimit (域名限制插件) 29
Hash (哈希器) 33
HeaderFilter (超文本传输协议响应包头过滤器插件) 38
HttpResponse (超文本传输协议响应) 41
Log (日志) 44
MaxDepth (最大深度插件) 49
MultiIo (多路输入输出) 52
Plugin (插件接口) 56
PluginMngr (插件管理器) 59
RawUrl (原始统一资源定位符) 65
RecvThread (接收线程) 69
SaveHTMLToFile (超文本标记语言文件存储插件) 73
SaveImageToFile (图像文件存储插件) 77
SendThread (发送线程) 82
Socket (套接字) 84
StrKit (字符串工具包) 94
Thread (线程) 98
UrlFilter (统一资源定位符过滤器接口) 102
UrlQueues (统一资源定位符队列) 104
WebCrawler (网络爬虫) 119

文件索引

文件列表

这里列出了所有文件，并附带简要说明:

G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.cpp (实现::DomainLimit类) 136
G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.h (声明::DomainLimit类) ..138
G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.cpp (实现::HeaderFilter类) 139
G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.h (声明::HeaderFilter类) 140
G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.cpp (实现::MaxDepth类) 141
G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.h (声明::MaxDepth类) 142
G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.cpp (实现::SaveHTMLToFile类) 143
G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.h (声明::SaveHTMLToFile类) 144
G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.cpp (实现::SaveImageToFile类) 145
G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.h (声明::SaveImageToFile类) 146
G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.cpp (实现::BloomFilter类) 147
G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.h (声明::BloomFilter类) 148
G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.cpp (实现::Configurator类) 149
G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.h (声明::Configurator类) 150
G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.cpp (实现::DnsThread类) 151
G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.h (声明::DnsThread类) 152
G:/Projects/Tarena/WebCrawler/teacher/src/Hash.cpp (实现::Hash类) 153
G:/Projects/Tarena/WebCrawler/teacher/src/Hash.h (声明::Hash类) 154
G:/Projects/Tarena/WebCrawler/teacher/src/Http.h (定义::HttpHeader类和::HttpResponse类) 155
G:/Projects/Tarena/WebCrawler/teacher/src/Log.cpp (实现::Log类) 157
G:/Projects/Tarena/WebCrawler/teacher/src/Log.h (声明::Log类) 158
G:/Projects/Tarena/WebCrawler/teacher/src/Main.cpp (定义::main函数) 159
G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.cpp (实现::MultiIo类) 164
G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.h (声明::MultiIo类) 165
G:/Projects/Tarena/WebCrawler/teacher/src/Plugin.h (定义::Plugin接口类) 166
G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.cpp (实现::PluginMngr类) 167
G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.h (声明::PluginMngr类) 168
G:/Projects/Tarena/WebCrawler/teacher/src/Precompile.h (预编译头文件) 169
G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.cpp (实现::RecvThread类) 171
G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.h (声明::RecvThread类) 172
G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.cpp (实现::SendThread类) 173

G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.h (声明::SendThread类) 174
G:/Projects/Tarena/WebCrawler/teacher/src/Socket.cpp (实现::Socket类) 175
G:/Projects/Tarena/WebCrawler/teacher/src/Socket.h (声明::Socket类) 176
G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.cpp (实现::StrKit类) 178
G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.h (声明::StrKit类) 179
G:/Projects/Tarena/WebCrawler/teacher/src/Thread.cpp (实现::Thread抽象基类) 180
G:/Projects/Tarena/WebCrawler/teacher/src/Thread.h (声明::Thread抽象基类) 181
G:/Projects/Tarena/WebCrawler/teacher/src/Url.cpp (实现::RawUrl类和::DnsUrl类) 182
G:/Projects/Tarena/WebCrawler/teacher/src/Url.h (声明::RawUrl类和::DnsUrl类) 183
G:/Projects/Tarena/WebCrawler/teacher/src/UrlFilter.h (定义::UrlFilter接口类) 184
G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.cpp (实现::UrlQueues类) 185
G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.h (声明::UrlQueues类) 186
G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.cpp (实现::WebCrawler类) 187
G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.h (声明::WebCrawler类) 188

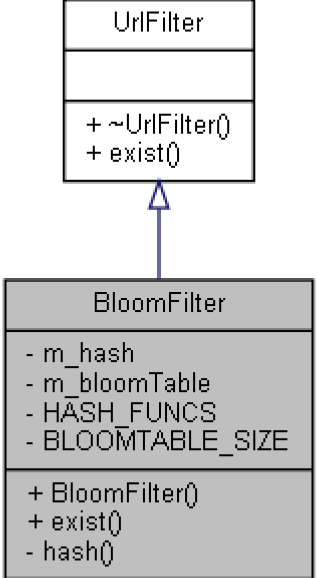
类说明

BloomFilter类 参考

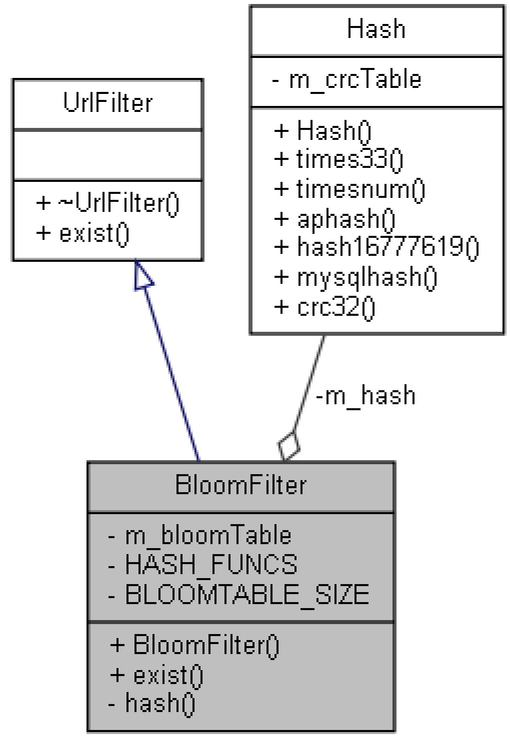
布隆过滤器

```
#include <BloomFilter.h>
```

类 BloomFilter 继承关系图:



BloomFilter 的协作图:



Public 成员函数

- BloomFilter (void)
  构造器
- bool exist (string const &strUrl)
  判断某个统一资源定位符是否已经存在

Private 成员函数

- unsigned int hash (int id, string const &strUrl) const
  用特定的哈希算法计算某个统一资源定位符的哈希值

Private 属性

- Hash m_hash
  哈希器
- unsigned int m_bloomTable [BLOOMTABLE_SIZE]
  布隆表

静态 Private 属性

- static int const HASH_FUNCS = 8
  哈希值数
- static size_t const BLOOMTABLE_SIZE = 1000000
  布隆表元素数

详细描述

布隆过滤器

构造及析构函数说明

BloomFilter::BloomFilter (void)

构造器

参考 m_bloomTable.

```
12                     {
13     // 初始化布隆表
14     bzero (m_bloomTable, sizeof (m_bloomTable));
15 }
```

成员函数说明

bool BloomFilter::exist (string const & *strUrl*)[virtual]

判断某个统一资源定位符是否已经存在

返回值:

true	存在
false	不存在，同时将其加入布隆表

注解:

根据布隆过滤器的策略实现基类中的纯虚函数

参数:

in	*strUrl*	统一资源定位符

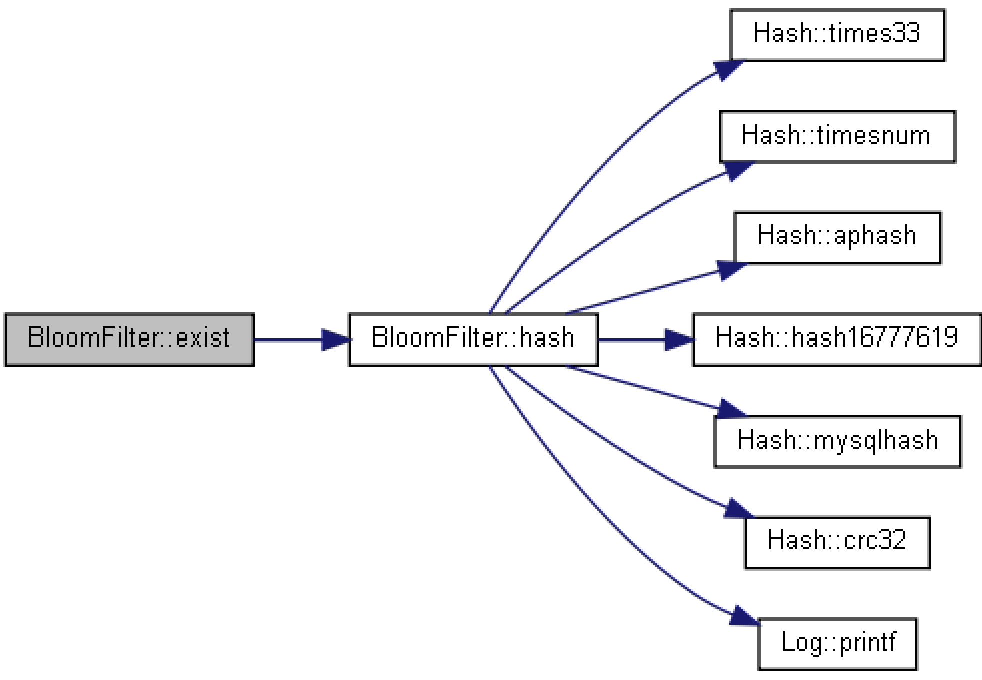
实现了 UrlFilter.

参考 hash(), HASH_FUNCS , 以及 m_bloomTable.

```
22         {
23     // 初始化1位计数器
24     int one = 0;
25
26     // 逐个计算哈希值
27     for (int i = 0; i < HASH_FUNCS; ++i) {
28         // 计算哈希值对应的布隆表位数
29         unsigned int bit = hash (i, strUrl) % (sizeof (m_bloomTable) * 8);
30         // 计算哈希值对应的布隆表元素下标
31         unsigned int idx = bit / (sizeof (m_bloomTable[0]) * 8);
32         // 计算哈希值对应的布隆表元素中的位数
33         bit %= sizeof (m_bloomTable[0]) * 8;
34
35         // 若此位已为1
36         if (m_bloomTable[idx] & 0x80000000 >> bit)
37             // 1位计数器加1
38             ++one;
```

```
39          // 否则
40          else
41              // 此位置1
42              m_bloomTable[idx] |= 0x80000000 >> bit;
43      }
44
45      // 若全部哈希值对应的布隆表位都为1，
46      // 则返回true，否则返回false
47      return one == HASH_FUNCS;
48  }
```

函数调用图:



unsigned int BloomFilter::hash (int *id*, string const & *strUrl*) const `[private]`

用特定的哈希算法计算某个统一资源定位符的哈希值

返回:

32位哈希值

参数:

in	*id*	哈希算法标识号
in	*strUrl*	统一资源定位符

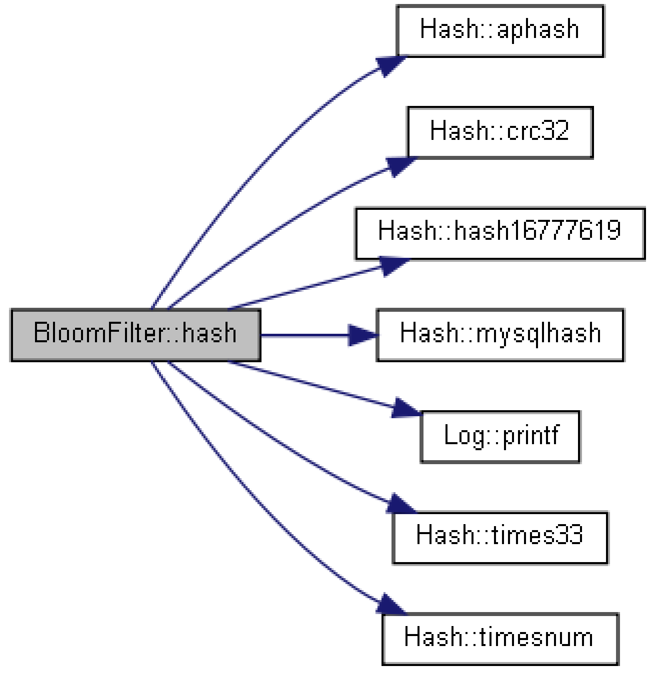
参考 Hash::aphash(), Hash::crc32(), g_app, Hash::hash16777619(), Log::LEVEL_ERR, m_hash, WebCrawler::m_log, Hash::mysqlhash(), Log::printf(), Hash::times33() , 以及 Hash::timesnum().

参考自 exist().

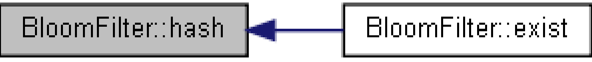
```
55          {
56      // 32位哈希值
57      unsigned int val = 0;
58
```

```
 59     // 若哈希算法标识号...
 60     switch (id) {
 61         // 为0
 62         case 0:
 63             // 用Times33算法计算参数统一资源
 64             // 定位符字符串的32位哈希值
 65             val = m_hash.times33 (strUrl);
 66             break;
 67
 68         // 为1
 69         case 1:
 70             // 用Times31算法计算参数统一资源
 71             // 定位符字符串的32位哈希值
 72             val = m_hash.timesnum (strUrl, 31);
 73             break;
 74
 75         // 为2
 76         case 2:
 77             // 用AP算法计算参数统一资源
 78             // 定位符字符串的32位哈希值
 79             val = m_hash.aphash (strUrl);
 80             break;
 81
 82         // 为3
 83         case 3:
 84             // 用FNV算法计算参数统一资源
 85             // 定位符字符串的32位哈希值
 86             val = m_hash.hash16777619 (strUrl);
 87             break;
 88
 89         // 为4
 90         case 4:
 91             // 用MySQL算法计算参数统一资源
 92             // 定位符字符串的32位哈希值
 93             val = m_hash.mysqlhash (strUrl);
 94             break;
 95
 96         // 为5
 97         case 5:
 98             // 用循环冗余校验算法计算参数统一资源
 99             // 定位符字符串的32位哈希值
100             val = m_hash.crc32 (strUrl);
101             break;
102
103         // 为6
104         case 6:
105             // 用Times131算法计算参数统一资源
106             // 定位符字符串的32位哈希值
107             val = m_hash.timesnum (strUrl, 131);
108             break;
109
110         // 为7
111         case 7:
112             // 用Times1313算法计算参数统一资源
113             // 定位符字符串的32位哈希值
114             val = m_hash.timesnum (strUrl, 1313);
115             break;
116
117         // 为其它
118         default:
119             // 记录一般错误日志
120             g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
121                 "无效哈希算法标识: %d", id);
122     }
123
124     // 返回32位哈希值
```

```
125     return val;
126 }
```
函数调用图:



这是这个函数的调用关系图:



类成员变量说明

size_t const BloomFilter::BLOOMTABLE_SIZE = 1000000 [static], [private]

布隆表元素数

int const BloomFilter::HASH_FUNCS = 8 [static], [private]

哈希值数

参考自 exist().

unsigned int BloomFilter::m_bloomTable[BLOOMTABLE_SIZE] [private]

布隆表

参考自 BloomFilter() , 以及 exist().

Hash BloomFilter::m_hash [private]

哈希器

参考自 hash().

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.h
- G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.cpp

Configurator类 参考

配置器

#include <Configurator.h>

Configurator 的协作图:

Configurator

+ m_logLevel
+ m_logFile
+ m_maxJobs
+ m_maxDepth
+ m_maxRawUrls
+ m_maxDnsUrls
+ m_statInterval
+ m_seeds
+ m_includePrefixes
+ m_excludePrefixes
+ m_pluginsPath
+ m_loadPlugins
+ m_acceptTypes

+ Configurator()
+ load()

Public 成员函数

- Configurator (void)
  构造器
- void load (string const &cfgFile)
  从指定的配置文件中加载配置信息

Public 属性

- int m_logLevel
  最低日志等级
- string m_logFile
  日志文件路径
- int m_maxJobs
  最大抓取任务数，0表示不抓取，-1表示无限抓取
- int m_maxDepth
  最大递归深度，种子深度为0，之后逐层递增，-1表示无限深度
- int m_maxRawUrls
  原始统一资源定位符队列最大容量，-1表示无限大
- int m_maxDnsUrls
  解析统一资源定位符队列最大容量，-1表示无限大
- long m_statInterval
  状态间隔，即状态定时器的周期秒数，0表示不设定时器
- string m_seeds

种子链接，多个链接以逗号隔开

- string m_includePrefixes
  包含前缀，只抓取带有这些前缀的URL，多个前缀以逗号隔开
- string m_excludePrefixes
  排除前缀，不抓取带有这些前缀的URL，多个前缀以逗号隔开
- string m_pluginsPath
  插件路径
- vector< string > m_loadPlugins
  插件列表
- vector< string > m_acceptTypes
  接受类型

详细描述

配置器

构造及析构函数说明

Configurator::Configurator (void)

构造器

```
12                               :
13     m_logLevel     (Log::LEVEL_DBG), // 最低日志等级缺省为调试
14     m_maxJobs      (-1),           // 最大抓取任务数缺省为无限
15     m_maxDepth     (-1),           // 最大递归深度缺省为无限
16     m_maxRawUrls   (-1),           // 原始统一资源定位符队
17                                   // 列最大容量缺省为无限
18     m_maxDnsUrls   (-1),           // 解析统一资源定位符队
19                                   // 列最大容量缺省为无限
20     m_statInterval (0) {}           // 状态间隔缺省为不设定时器
```

成员函数说明

void Configurator::load (string const & *cfgFile*)

从指定的配置文件中加载配置信息

参数:

in	*cfgFile*	配置文件路径

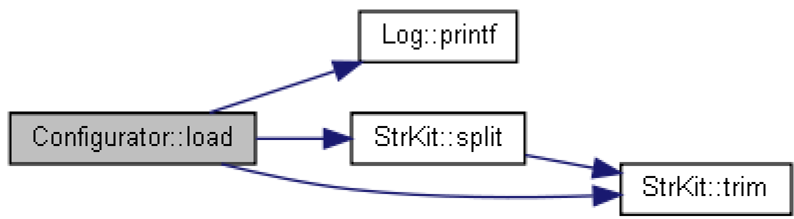
参考 g_app, Log::LEVEL_ERR, m_acceptTypes, m_excludePrefixes, m_includePrefixes, m_loadPlugins, WebCrawler::m_log, m_logFile, m_logLevel, m_maxDepth, m_maxDnsUrls, m_maxJobs, m_maxRawUrls, m_pluginsPath, m_seeds, m_statInterval, Log::printf(), StrKit::split() , 以及 StrKit::trim().

参考自 WebCrawler::init().

```
25      {
26    // 根据路径打开配置文件输入流
27    ifstream ifs (cfgFile.c_str ());
28    // 若失败
29    if (! ifs)
30        // 记录一般错误日志
31        g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
32            "加载配置文件\"%s\"失败", cfgFile.c_str ());
33
34    // 文件行字符串
35    string line;
36    // 逐行读取配置文件
37    for (int lineNo = 0; getline (ifs, line); ++lineNo) {
38        // 修剪行字符串
39        StrKit::trim (line);
40
41        // 若为注释行或空行
42        if (line[0] == '#' || line[0] == '\0')
43            // 继续下一轮循环
44            continue;
45
46        // 拆分行字符串，以等号为分隔符，最多拆分一次
47        vector<string> strv = StrKit::split (line, "=", 1);
48        // 若成功拆分出键和值两个子串
49        if (strv.size () == 2) {
50            // 若键为"LOG_LEVEL"
51            if (! strcasecmp (strv[0].c_str (), "LOG_LEVEL"))
52                // 则值为"最低日志等级"
53                m_logLevel = atoi (strv[1].c_str ());
54            // 否则
55            else
56            // 若键为"LOG_FILE"
57            if (! strcasecmp (strv[0].c_str (), "LOG_FILE"))
58                // 则值为"日志文件路径"
59                m_logFile = strv[1];
60            // 否则
61            else
62            // 若键为"MAX_JOBS"
63            if (! strcasecmp (strv[0].c_str (), "MAX_JOBS"))
64                // 则值为"最大抓取任务数"
65                m_maxJobs = atoi (strv[1].c_str ());
66            // 否则
67            else
68            // 若键为"MAX_DEPTH"
69            if (! strcasecmp (strv[0].c_str (), "MAX_DEPTH"))
70                // 则值为"最大递归深度"
71                m_maxDepth = atoi (strv[1].c_str ());
72            // 否则
73            else
74            // 若键为"MAX_RAW_URLS"
75            if (! strcasecmp (strv[0].c_str (), "MAX_RAW_URLS"))
76                // 则值为"原始统一资源定位符队列最大容量"
77                m_maxRawUrls = atoi (strv[1].c_str ());
78            // 否则
79            else
80            // 若键为"MAX_DNS_URLS"
81            if (! strcasecmp (strv[0].c_str (), "MAX_DNS_URLS"))
82                // 则值为"解析统一资源定位符队列最大容量"
83                m_maxDnsUrls = atoi (strv[1].c_str ());
84            // 否则
85            else
86            // 若键为"STAT_INTERVAL"
87            if (! strcasecmp (strv[0].c_str (), "STAT_INTERVAL"))
88                // 则值为"状态间隔"
89                m_statInterval = atoi (strv[1].c_str ());
90            // 否则
```

```
 91             else
 92             // 若键为"SEEDS"
 93             if (! strcasecmp (strv[0].c_str (), "SEEDS"))
 94                 // 则值为"种子链接"
 95                 m_seeds = strv[1];
 96             // 否则
 97             else
 98             // 若键为"INCLUDE_PREFIXES"
 99             if (! strcasecmp (strv[0].c_str (), "INCLUDE_PREFIXES"))
100                 // 则值为"包含前缀"
101                 m_includePrefixes = strv[1];
102             // 否则
103             else
104             // 若键为"EXCLUDE_PREFIXES"
105             if (! strcasecmp (strv[0].c_str (), "EXCLUDE_PREFIXES"))
106                 // 则值为"排除前缀"
107                 m_excludePrefixes = strv[1];
108             // 否则
109             else
110             // 若键为"PLUGINS_PATH"
111             if (! strcasecmp (strv[0].c_str (), "PLUGINS_PATH"))
112                 // 则值为"插件路径"
113                 m_pluginsPath = strv[1];
114             // 否则
115             else
116             // 若键为"LOAD_PLUGIN"
117             if (! strcasecmp (strv[0].c_str (), "LOAD_PLUGIN"))
118                 // 则值为"插件名"
119                 m_loadPlugins.push_back (strv[1]);
120             // 否则
121             else
122             // 若键为"ACCEPT_TYPE"
123             if (! strcasecmp (strv[0].c_str (), "ACCEPT_TYPE"))
124                 // 则值为"接受类型"
125                 m_acceptTypes.push_back (strv[1]);
126             // 否则
127             else
128                 // 记录一般错误日志
129                 g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
130                     "无效配置项: %s, %d, %s", cfgFile.c_str (),
131                     lineNo, strv[0].c_str ());
132         }
133         // 否则
134         else
135             // 记录一般错误日志
136             g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
137                 "无效配置行: %s, %d, %s", cfgFile.c_str (),
138                 lineNo, line.c_str ());
139     }
140 
141     // 关闭配置文件输入流
142     ifs.close ();
143 }
```

函数调用图:



这是这个函数的调用关系图:

类成员变量说明

vector<string> Configurator::m_acceptTypes

接受类型

参考自 HeaderFilter::handler()，以及 load().

string Configurator::m_excludePrefixes

排除前缀，不抓取带有这些前缀的URL，多个前缀以逗号隔开

参考自 DomainLimit::handler(), DomainLimit::init()，以及 load().

string Configurator::m_includePrefixes

包含前缀，只抓取带有这些前缀的URL，多个前缀以逗号隔开

参考自 DomainLimit::handler(), DomainLimit::init()，以及 load().

vector<string> Configurator::m_loadPlugins

插件列表

参考自 PluginMngr::load()，以及 load().

string Configurator::m_logFile

日志文件路径

参考自 WebCrawler::initDaemon()，以及 load().

int Configurator::m_logLevel

最低日志等级

从低到高依次为：

- 0 - 调试
- 1 - 信息
- 2 - 警告
- 3 - 错误
- 4 - 致命

系统将记录所有不低于指定等级的日志信息

参考自 load()，以及 Log::printf().

int Configurator::m_maxDepth

最大递归深度，种子深度为0，之后逐层递增，-1表示无限深度

参考自 MaxDepth::handler()，以及 load().

int Configurator::m_maxDnsUrls

解析统一资源定位符队列最大容量，-1表示无限大

参考自 UrlQueues::fullDnsUrl(), load(), UrlQueues::popDnsUrl()，以及 UrlQueues::pushDnsUrl().

int Configurator::m_maxJobs

最大抓取任务数，0表示不抓取，-1表示无限抓取

参考自 WebCrawler::exec(), load(), WebCrawler::startJob()，以及 WebCrawler::stopJob().

int Configurator::m_maxRawUrls

原始统一资源定位符队列最大容量，-1表示无限大

参考自 UrlQueues::fullRawUrl(), load(), UrlQueues::popRawUrl()，以及 UrlQueues::pushRawUrl().

string Configurator::m_pluginsPath

插件路径

参考自 PluginMngr::load()，以及 load().

string Configurator::m_seeds

种子链接，多个链接以逗号隔开

参考自 WebCrawler::initSeeds()，以及 load().

long Configurator::m_statInterval

状态间隔，即状态定时器的周期秒数，0表示不设定时器

参考自 WebCrawler::initTicker()，以及 load().

该类的文档由以下文件生成:

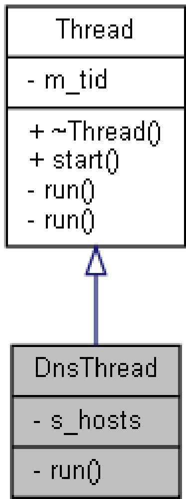
- G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.cpp

DnsThread类 参考

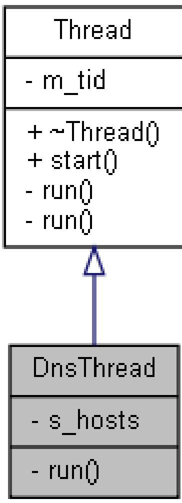
域名解析线程

`#include <DnsThread.h>`

类 DnsThread 继承关系图:



DnsThread 的协作图:



Private 成员函数

- void * run (void)
  线程处理函数

静态 Private 属性

- static map< string, string > s_hosts
  主机域名————IP地址映射表

额外继承的成员函数

详细描述

域名解析线程

成员函数说明

void * DnsThread::run (void) [private], [virtual]

线程处理函数

注解:

根据域名解析线程的任务实现基类中的纯虚函数

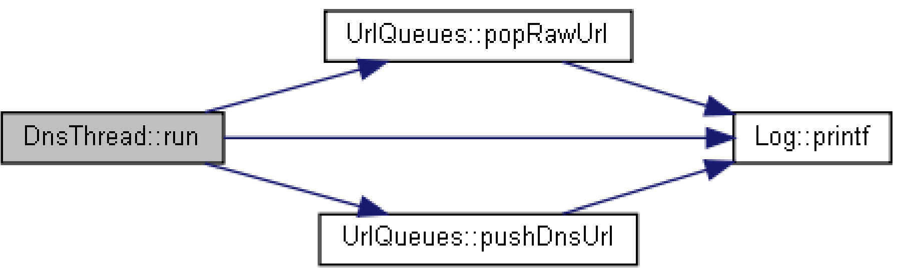
实现了 Thread.

参考 g_app, Log::LEVEL_DBG, Log::LEVEL_WAR, DnsUrl::m_domain, DnsUrl::m_ip, WebCrawler::m_log, WebCrawler::m_urlQueues, UrlQueues::popRawUrl(), Log::printf(), UrlQueues::pushDnsUrl() , 以及 s_hosts.

```
12                   {
13  // 记录调试日志
14  g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
15      "DNS线程开始");
16
17  // 无限循环
18  for (;;) {
19      // 从统一资源定位符队列，弹出原始统一资源
20      // 定位符，并显式转换为解析统一资源定位符
21      DnsUrl dnsUrl = static_cast<DnsUrl> (
22          g_app->m_urlQueues.popRawUrl ());
23
24      // 在主机域名————IP地址映射表中，
25      // 查找该统一资源定位符的主机域名
26      map<string, string>::const_iterator it =
27          s_hosts.find (dnsUrl.m_domain);
28
29      // 若找到了
30      if (it != s_hosts.end ()) {
31          // 将与该主机域名对应的IP地址，
32          // 存入解析统一资源定位符
33          dnsUrl.m_ip = it->second;
34          // 将解析统一资源定位符，压入统一资源定位符队列
35          g_app->m_urlQueues.pushDnsUrl (dnsUrl);
36
37          // 记录调试日志
38          g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
39              "域名\"%s\"曾经被解析为\"%s\"", dnsUrl.m_domain.c_str (),
40              dnsUrl.m_ip.c_str ());
41          // 继续下一轮循环
42          continue;
43      }
44
45      // 若没找到，则通过域名系统获取与该主机域名对应的IP地址
46      hostent* host = gethostbyname (dnsUrl.m_domain.c_str ());
47
```

```
 48         // 若失败
 49         if (! host) {
 50             // 记录警告日志
 51             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 52                 "gethostbyname: %s", hstrerror (h_errno));
 53             // 继续下一轮循环
 54             continue;
 55         }
 56
 57         //     hostent
 58         // +--------------+
 59         // | h_name      -> xxx\0          - 正式主机名
 60         // | h_aliases   -> * * * ... NULL - 别名表
 61         // | h_addrtype  |  AF_INET        - 地址类型
 62         // | h_length    |  4              - 地址字节数
 63         // | h_addr_list -> * * * ... NULL - 地址表
 64         // +--------------+  +-> in_addr   - IPv4地址结构
 65
 66         // 若地址类型不是IPv4
 67         if (host->h_addrtype != AF_INET) {
 68             // 记录警告日志
 69             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 70                 "无效地址类型");
 71             // 继续下一轮循环
 72             continue;
 73         }
 74
 75         // 将IPv4地址结构转换为点分十进制字符串，存入解析统
 76         // 一资源定位符，同时加入主机域名————IP地址映射表
 77         s_hosts[dnsUrl.m_domain] = dnsUrl.m_ip =
 78             inet_ntoa (**(in_addr**)host->h_addr_list);
 79         // 将解析统一资源定位符，压入统一资源定位符队列
 80         g_app->m_urlQueues.pushDnsUrl (dnsUrl);
 81
 82         // 记录调试日志
 83         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
 84             "域名\"%s\"被成功解析为\"%s\"", dnsUrl.m_domain.c_str (),
 85             dnsUrl.m_ip.c_str ());
 86         /*
 87         // 初始化libevent库
 88         event_base* base = event_init ();
 89         // 初始化dns模块
 90         evdns_init ();
 91
 92         // 一旦完成对该主机域名的解析即调用回调函数
 93         evdns_resolve_ipv4 (dnsUrl.m_domain.c_str (), 0,
 94             callback, &dnsUrl);
 95         // 进入事件循环
 96         event_dispatch ();
 97
 98         // 释放libevent库
 99         event_base_free (base);
100         */
101     }
102
103     // 记录调试日志
104     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
105         "DNS线程终止");
106     // 终止线程
107     return NULL;
108 }
```

函数调用图:



类成员变量说明

map< string, string > DnsThread::s_hosts [static], [private]

主机域名————IP地址映射表

参考自 run().

该类的文档由以下文件生成:

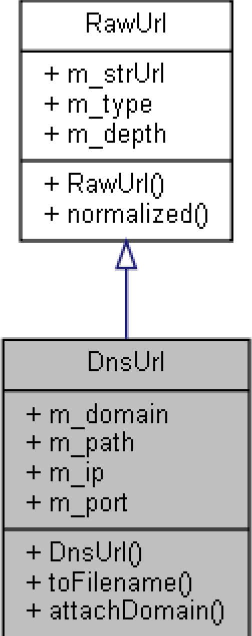
- G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.h
- G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.cpp

DnsUrl类 参考

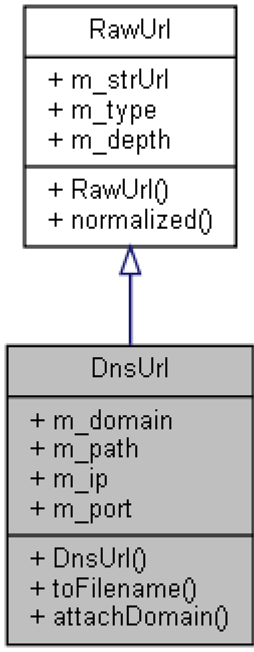
解析统一资源定位符

```
#include <Url.h>
```

类 DnsUrl 继承关系图:



DnsUrl 的协作图:



Public 成员函数

- DnsUrl (RawUrl const &rawUrl)
  构造器
- string toFilename (void) const
  转换为文件名字符串
- bool attachDomain (string &strUrl) const
  添加域名

Public 属性

- string m_domain
  服务器域名
- string m_path
  资源路径
- string m_ip
  服务器IP地址
- short m_port
  服务器通信端口

额外继承的成员函数

详细描述

解析统一资源定位符

构造及析构函数说明

DnsUrl::DnsUrl (RawUrl const & *rawUrl*) [explicit]

构造器

参数:

in	*rawUrl*	原始统一资源定位符

参考 m_domain, m_path, m_port , 以及 RawUrl::m_strUrl.

```
  68        : RawUrl (rawUrl) { // 初始化原始统一资源定位符基类子对象
  69      // 在从基类继承的统一资源定位符字符串中查找第一个'/'
  70      string::size_type pos = m_strUrl.find_first_of ('/');
  71      // 若没找到
  72      if (pos == string::npos)
  73          // 整个统一资源定位符字符串都认作服务器域名
  74          m_domain = m_strUrl;
  75      // 否则
  76      else {
  77          // 统一资源定位符字符串中第一个'/'之前为服务器域名
  78          m_domain = m_strUrl.substr (0, pos);
  79          // 之后为资源路径。注意作为服务器域名和
```

```
80        // 资源路径之间分隔符的'/'不放在路径中
81        m_path = m_strUrl.substr (pos + 1);
82    }
83
84    // 在服务器域名中查找最后一个':'。若找到了，则将其后
85    // 的子串转换为整数作为服务器通信端口；若没有找到或转
86    // 换后的端口号非法，则将服务器通信端口设置为缺省值80
87    if ((pos = m_domain.find_last_of (':')) == string::npos ||
88        ! (m_port = atoi (m_domain.substr (pos + 1).c_str ())))
89        m_port = 80;
90 }
```

成员函数说明

bool DnsUrl::attachDomain (string & *strUrl*) const

添加域名

返回值:

true	成功
false	失败

参数:

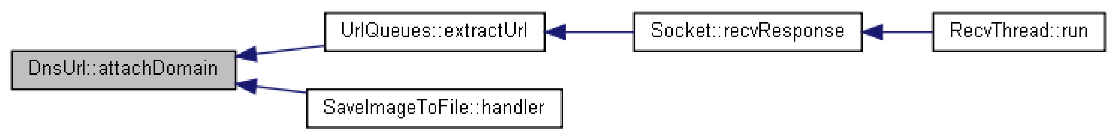
in,out	*strUrl*	待加域名统一资源定位符字符串

参考 m_domain.

参考自 UrlQueues::extractUrl() , 以及 SaveImageToFile::handler().

```
130         {
131    // 若待加域名统一资源定位符字符串以"http"开头，说明已有域名
132    if (! strUrl.find ("http", 0))
133        // 直接返回成功
134        return true;
135
136    // 若待加域名统一资源定位符字符串为空，或首字符不是'/'
137    if (strUrl.empty () || strUrl[0] != '/')
138        // 返回失败
139        return false;
140
141    // 将服务器域名插到待加域名统一资源定位符字符串首字符之前
142    strUrl.insert (0, m_domain);
143    // 返回成功
144    return true;
145 }
```

这是这个函数的调用关系图:



string DnsUrl::toFilename (void) const

转换为文件名字符串

返回:

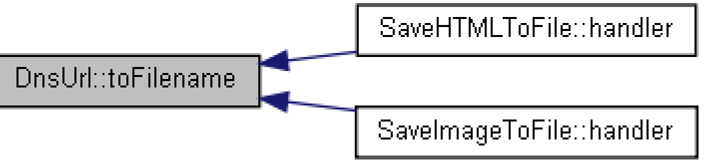
文件名字符串

参考 RawUrl::ETYPE_HTML, m_domain, m_path，以及 RawUrl::m_type.

参考自 SaveHTMLToFile::handler()，以及 SaveImageToFile::handler().

```
 94                                    {
 95     // 用服务器域名初始化文件名字符串
 96     string filename = m_domain;
 97 
 98     // 若资源路径非空
 99     if (! m_path.empty ())
100         // 在文件名字符串中追加"/"和资源路径
101         (filename += "/") += m_path;
102 
103     // 逐个查找文件名字符串中的每个'/'
104     for (string::size_type pos = 0; (pos = filename.find (
105         '/', pos)) != string::npos; ++pos)
106         // 若找到，则将其替换为"_"
107         filename.replace (pos, 1, "_");
108 
109     // 若资源类型为超文本标记语言
110     if (m_type == ETYPE_HTML) {
111         // 在文件名字符串中查找最后一个'.'
112         string::size_type pos = filename.find_last_of ('.');
113         // 若没找到，或虽然找到但连同其后
114         // 的子串既不是".htm"也不是".html"
115         if (pos == string::npos || (
116             filename.substr (pos) != ".htm" &&
117             filename.substr (pos) != ".html"))
118             // 在文件名字符串中追加扩展名".html"
119             filename += ".html";
120     }
121 
122     // 返回文件名字符串
123     return filename;
124 }
```

这是这个函数的调用关系图:



类成员变量说明

string DnsUrl::m_domain

服务器域名

参考自 attachDomain(), DnsUrl(), DnsThread::run(), Socket::sendRequest()，以及 toFilename().

string DnsUrl::m_ip

服务器IP地址

参考自 UrlQueues::popDnsUrl(), UrlQueues::pushDnsUrl(), DnsThread::run() ，以及 Socket::sendRequest().

string DnsUrl::m_path

资源路径

参考自 DnsUrl(), UrlQueues::popDnsUrl(), UrlQueues::pushDnsUrl(), Socket::sendRequest()，以及 toFilename().

short DnsUrl::m_port

服务器通信端口

参考自 DnsUrl(), UrlQueues::popDnsUrl(), UrlQueues::pushDnsUrl()，以及 Socket::sendRequest().

该类的文档由以下文件生成:

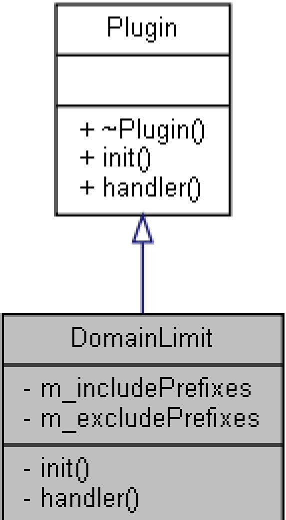
- G:/Projects/Tarena/WebCrawler/teacher/src/Url.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Url.cpp

DomainLimit类 参考

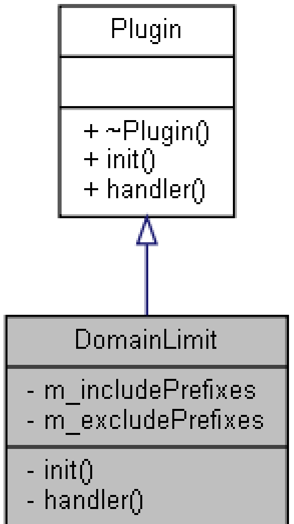
域名限制插件

`#include <DomainLimit.h>`

类 DomainLimit 继承关系图:



DomainLimit 的协作图:



Private 成员函数

- bool init (WebCrawler *app)
  插件初始化
- bool handler (void *arg)
  插件处理

Private 属性

- vector< string > m_includePrefixes
  包含前缀字符串向量
- vector< string > m_excludePrefixes
  排除前缀字符串向量

额外继承的成员函数

详细描述

域名限制插件

成员函数说明

bool DomainLimit::handler (void * *arg*)`[private]`, `[virtual]`

插件处理

返回值:

true	成功
false	失败

注解:

根据域名限制插件的功能实现基类中的虚函数

参数:

in,out	*arg*	插件参数

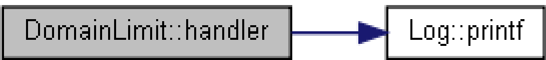
实现了 Plugin.

参考 RawUrl::ETYPE_HTML, Log::LEVEL_WAR, WebCrawler::m_cfg, RawUrl::m_depth, m_excludePrefixes, Configurator::m_excludePrefixes, m_includePrefixes, Configurator::m_includePrefixes, WebCrawler::m_log, RawUrl::m_strUrl, RawUrl::m_type ，以及 Log::printf().

```
42    {
43    // 原始统一资源定位符
44    RawUrl* rawUrl = static_cast<RawUrl*> (arg);
45
46    // 若是种子链接(链接深度为0)或非超文本标记语言
47    if (! rawUrl->m_depth || rawUrl->m_type != RawUrl::ETYPE_HTML)
48        // 返回成功，抓取该统一资源定位符
49        return true;
50
51    // 字符串向量只读迭代器
52    vector<string>::const_iterator it;
53
54    // 统一资源定位符字符串以哪个包含前缀开头
55    for (it = m_includePrefixes.begin ();
56        it != m_includePrefixes.end () &&
57        rawUrl->m_strUrl.find (*it, 0); ++it);
```

```
58
59      // 若没有与统一资源定位符字符串匹配的包含前缀
60      if (! m_includePrefixes.empty () &&
61          it == m_includePrefixes.end ()) {
62          // 记录警告日志
63          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
64              "不抓不在包含集\"%s\"中的统一资源定位符\"%s\"",
65              g_app->m_cfg.m_includePrefixes.c_str (),
66              rawUrl->m_strUrl.c_str ());
67          // 返回失败，不抓该统一资源定位符
68          return false;
69      }
70
71      // 统一资源定位符字符串以哪个排除前缀开头
72      for (it = m_excludePrefixes.begin ();
73          it != m_excludePrefixes.end () &&
74          rawUrl->m_strUrl.find (*it, 0); ++it);
75
76      // 若存在与统一资源定位符字符串匹配的排除前缀
77      if (it != m_excludePrefixes.end ()) {
78          // 记录警告日志
79          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
80              "不抓排除集\"%s\"中的统一资源定位符\"%s\"",
81              g_app->m_cfg.m_excludePrefixes.c_str (),
82              rawUrl->m_strUrl.c_str ());
83          // 返回失败，不抓该统一资源定位符
84          return false;
85      }
86
87      // 返回成功，抓取该统一资源定位符
88      return true;
89  }
```

函数调用图:



bool DomainLimit::init (WebCrawler * *app*)[private], [virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

根据域名限制插件的功能实现基类中的虚函数

参数:

in,out	*app*	应用程序对象

实现了 Plugin.

参考 WebCrawler::m_cfg, m_excludePrefixes, Configurator::m_excludePrefixes, m_includePrefixes, Configurator::m_includePrefixes , 以及 StrKit::split().

```
21      {
22      // 以统一资源定位符插件的身份
23      // 注册到应用程序对象的插件管理器中
```

```
24      (g_app = app)->m_pluginMngr.registerUrlPlugin (this);
25
26      // 拆分配置器中的包含前缀字符串，以逗号为分隔符，不限拆分次数
27      m_includePrefixes = StrKit::split (
28          g_app->m_cfg.m_includePrefixes, ",", 0);
29      // 拆分配置器中的排除前缀字符串，以逗号为分隔符，不限拆分次数
30      m_excludePrefixes = StrKit::split (
31          g_app->m_cfg.m_excludePrefixes, ",", 0);;
32
33      // 返回成功
34      return true;
35  }
```

函数调用图:

类成员变量说明

vector<string> DomainLimit::m_excludePrefixes [private]

排除前缀字符串向量

参考自 handler()，以及 init().

vector<string> DomainLimit::m_includePrefixes [private]

包含前缀字符串向量

参考自 handler()，以及 init().

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.h
- G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.cpp

Hash类 参考

哈希器

```
#include <Hash.h>
```

Hash 的协作图:

Hash
- m_crcTable
+ Hash()
+ times33()
+ timesnum()
+ aphash()
+ hash16777619()
+ mysqlhash()
+ crc32()

Public 成员函数

- Hash (void)
  构造器
- unsigned int times33 (string const &str) const
  Times33哈希算法
- unsigned int timesnum (string const &str, int num) const
  TimesN哈希算法
- unsigned int aphash (string const &str) const
  AP哈希算法
- unsigned int hash16777619 (string const &str) const
  FNV哈希算法
- unsigned int mysqlhash (string const &str) const
  MySQL哈希算法
- unsigned int crc32 (string const &str) const
  循环冗余校验算法

Private 属性

- unsigned int m_crcTable [256]
  CRC表

详细描述

哈希器

构造及析构函数说明

Hash::Hash (void)

构造器

参考 m_crcTable.

```
11              {
12    // 初始化CRC表
13    for (unsigned int i = 0; i < sizeof (m_crcTable) /
14        sizeof (m_crcTable[0]); ++i) {
15        unsigned int crc = i;
16
17        for (unsigned j = 0; j < 8; ++j)
18            if (crc & 1)
19                crc = crc >> 1 ^ 0xEDB88320;
20            else
21                crc = crc >> 1;
22
23        m_crcTable[i] = crc;
24    }
25 }
```

成员函数说明

unsigned int Hash::aphash (string const & *str*) const

AP哈希算法

返回:

参数字符串的32位哈希值

参数:

in	*str*	被哈希字符串

参考自 BloomFilter::hash().

```
62          {
63    unsigned int val = 0;
64
65    size_t len = str.size ();
66    for (size_t i = 0; i < len; ++i)
67        if (i & 1)
68            val ^= ~(val << 11 ^ (unsigned char)str[i] ^ val >> 5);
69        else
70            val ^= val << 7 ^ (unsigned char)str[i] ^ val >> 3;
71
72    return val & 0x7FFFFFFF;
73 }
```

这是这个函数的调用关系图:

unsigned int Hash::crc32 (string const & *str*) const

循环冗余校验算法

返回:

参数字符串的循环冗余校验码

参数:

in	*str*	被校验字符串

参考 m_crcTable.

参考自 BloomFilter::hash().

```
111         {
112     unsigned int val = 0xFFFFFFFF;
113 
114     size_t len = str.size ();
115     for (size_t i = 0; i < len; ++i)
116         val = m_crcTable[(val & 0xFF) ^ (unsigned char)str[i]] ^ val >> 8;
117 
118     return ~val;
119 }
```

这是这个函数的调用关系图:

unsigned int Hash::hash16777619 (string const & *str*) const

FNV哈希算法

返回:

参数字符串的32位哈希值

参数:

in	*str*	被哈希字符串

参考自 BloomFilter::hash().

```
79          {
80      unsigned int val = 0;
81 
82      size_t len = str.size ();
83      for (size_t i = 0; i < len; ++i) {
84          val *= 16777619;
85          val ^= (unsigned char)str[i];
86      }
87 
88      return val;
89 }
```

这是这个函数的调用关系图:

unsigned int Hash::mysqlhash (string const & *str*) const

MySQL哈希算法

返回:

参数字符串的32位哈希值

参数:

in	*str*	被哈希字符串

参考自 BloomFilter::hash().

```
 95         {
 96    unsigned int nr1 = 1, nr2 = 4;
 97
 98    size_t len = str.size ();
 99    for (size_t i = 0; i < len; ++i) {
100        nr1 ^= ((nr1 & 63) + nr2) * (unsigned char)str[i] + (nr1 << 8);
101        nr2 += 3;
102    }
103
104    return nr1;
105 }
```

这是这个函数的调用关系图:

unsigned int Hash::times33 (string const & *str*) const

Times33哈希算法

返回:

参数字符串的32位哈希值

备注:

hash(i) = hash(i-1)×33+str[i] (hash(-1) = 0)

参数:

in	*str*	被哈希字符串

参考自 BloomFilter::hash().

```
 32         {
 33    unsigned int val = 0;
 34
 35    size_t len = str.size ();
 36    for (size_t i = 0; i < len; ++i)
 37        val = (val << 5) + val + (unsigned char)str[i];
 38
 39    return val;
 40 }
```

这是这个函数的调用关系图:

unsigned int Hash::timesnum (string const & *str*, int *num*) const

TimesN哈希算法

返回:

参数字符串的32位哈希值

备注:

hash(i) = hash(i-1)×N+str[i] (hash(-1) = 0)

参数:

in	*str*	被哈希字符串
in	*num*	N

参考自 BloomFilter::hash().

```
48          {
49      unsigned int val = 0;
50
51      size_t len = str.size ();
52      for (size_t i = 0; i < len; ++i)
53          val = val * num + (unsigned char)str[i];
54
55      return val;
56  }
```

这是这个函数的调用关系图:

类成员变量说明

unsigned int Hash::m_crcTable[256] [private]

CRC表

参考自 crc32()，以及 Hash().

该类的文档由以下文件生成:

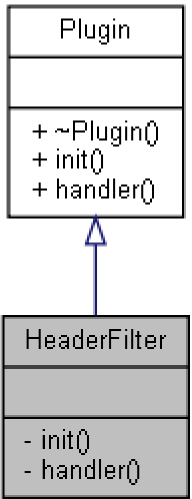
- G:/Projects/Tarena/WebCrawler/teacher/src/Hash.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Hash.cpp

HeaderFilter类 参考

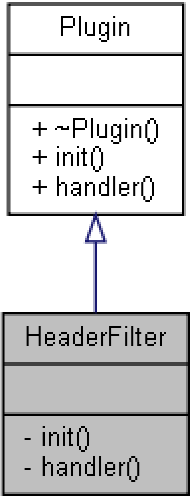
超文本传输协议响应包头过滤器插件

```
#include <HeaderFilter.h>
```

类 HeaderFilter 继承关系图:



HeaderFilter 的协作图:



Private 成员函数

- bool init (WebCrawler *app)
  插件初始化
- bool handler (void *arg)
  插件处理

额外继承的成员函数

详细描述

超文本传输协议响应包头过滤器插件

成员函数说明

bool HeaderFilter::handler (void * *arg*) [private], [virtual]

插件处理

返回值:

true	成功
false	失败

注解:

根据超文本传输协议响应包头过滤器插件的功能实现基类中的虚函数

参数:

in,out	*arg*	插件参数

实现了 Plugin.

参考 Log::LEVEL_DBG, Configurator::m_acceptTypes, WebCrawler::m_cfg, HttpHeader::m_contentType, WebCrawler::m_log, HttpHeader::m_statusCode , 以及 Log::printf().

```
35      {
36      // 超文本传输协议响应包头
37      HttpHeader* header = static_cast<HttpHeader*> (arg);
38
39      // 若超文本传输协议响应状态码不在合理区间[200,300)之内
40      if (header->m_statusCode < 200 || 300 <= header->m_statusCode) {
41          // 记录调试日志
42          g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
43              "超文本传输协议响应状态码%d不在合理区间[200,300)之内",
44              header->m_statusCode);
45          // 返回失败，不再接收该响应包包体
46          return false;
47      }
48
49      // 若超文本传输协议响应内容类型不是超文本标记语言
50      if (header->m_contentType.find ("text/html", 0) == string::npos) {
51          // 字符串向量只读迭代器
52          vector<string>::const_iterator it;
53
54          // 超文本传输协议响应内容类型与
55          // 配置器中的哪个接受类型匹配
56          for (it = g_app->m_cfg.m_acceptTypes.begin ();
57              it != g_app->m_cfg.m_acceptTypes.end () &&
58              header->m_contentType.find (*it, 0) == string::npos; ++it);
59
60          // 若超文本传输协议响应内容类型与
61          // 配置器中任何接受类型都不匹配
```

```
62         if (it == g_app->m_cfg.m_acceptTypes.end ()) {
63             // 记录调试日志
64             g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
65                 "超文本传输协议响应类型%s不在可接受范围之内",
66                 header->m_contentType.c_str ());
67             // 返回失败，不再接收该响应包包体
68             return false;
69         }
70     }
71 
72     // 返回成功，继续接收该响应包包体
73     return true;
74 }
```

函数调用图:

bool HeaderFilter::init (WebCrawler * *app*)[private], [virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

根据超文本传输协议响应包头过滤器插件的功能实现基类中的虚函数

参数:

in,out	*app*	应用程序对象

实现了 Plugin.

```
21     {
22     // 以超文本传输协议响应包头插件的身份
23     // 注册到应用程序对象的插件管理器中
24     (g_app = app)->m_pluginMngr.registerHeaderPlugin (this);
25 
26     // 返回成功
27     return true;
28 }
```

该类的文档由以下文件生成:

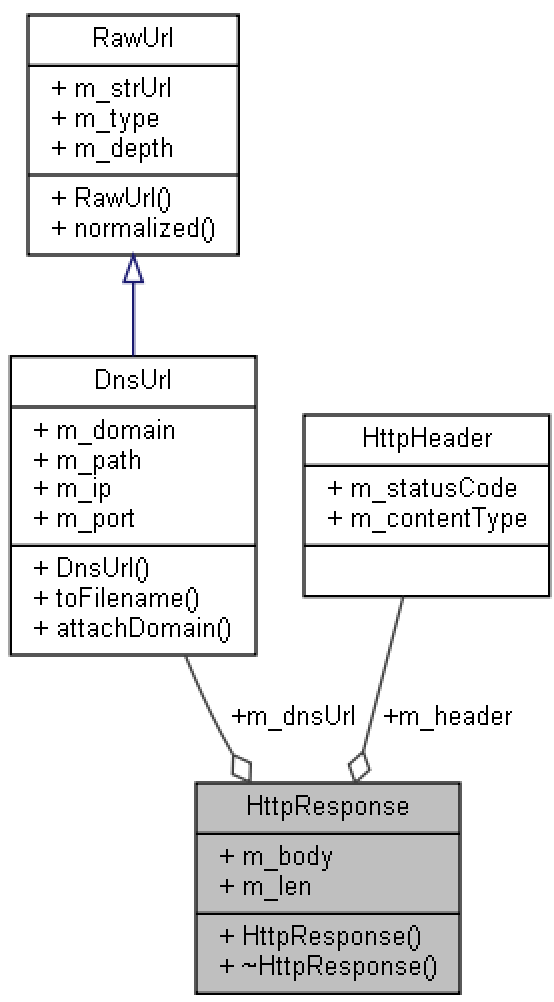
- G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.h
- G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.cpp

HttpResponse类 参考

超文本传输协议响应

`#include <Http.h>`

HttpResponse 的协作图:



Public 成员函数

- HttpResponse (DnsUrl const &dnsUrl)
  构造器
- ~HttpResponse (void)
  析构器

Public 属性

- DnsUrl m_dnsUrl
  服务器统一资源定位符
- HttpHeader m_header

超文本传输协议响应包头

- char * m_body
  超文本传输协议响应包体指针
- size_t m_len
  超文本传输协议响应包体长度

详细描述

超文本传输协议响应

构造及析构函数说明

HttpResponse::HttpResponse (DnsUrl const & *dnsUrl*) [inline]

构造器

参数:

in	*dnsUrl*	服务器统一资源定位符

```
27        : m_dnsUrl (dnsUrl), m_body (NULL), m_len (0) {}
```

HttpResponse::~HttpResponse (void) [inline]

析构器

```
30                      {
31          if (m_body) {
32              free (m_body);
33              m_body = NULL;
34          }
35          m_len = 0;
36      }
```

类成员变量说明

char* HttpResponse::m_body

超文本传输协议响应包体指针

参考自 SaveHTMLToFile::handler(), SaveImageToFile::handler() , 以及 Socket::recvResponse().

DnsUrl HttpResponse::m_dnsUrl

服务器统一资源定位符

参考自 SaveHTMLToFile::handler() , 以及 SaveImageToFile::handler().

HttpHeader HttpResponse::m_header

超文本传输协议响应包头

参考自 SaveHTMLToFile::handler(), SaveImageToFile::handler() , 以及 Socket::recvResponse().

size_t HttpResponse::m_len

超文本传输协议响应包体长度

参考自 SaveHTMLToFile::handler(), SaveImageToFile::handler() , 以及 Socket::recvResponse().

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/src/Http.h

Log类 参考

日志

`#include <Log.h>`

Log 的协作图:

Public 成员函数

- void printf (int level, char const *file, int line, char const *format,...) const
  按格式打印日志

静态 Public 属性

- static int const LEVEL_DBG = 0
  日志等级 - 调试
- static int const LEVEL_INF = 1
  日志等级 - 信息
- static int const LEVEL_WAR = 2
  日志等级 - 警告
- static int const LEVEL_ERR = 3
  日志等级 - 一般错误
- static int const LEVEL_CRT = 4
  日志等级 - 致命错误

静态 Private 属性

- static char const * s_levels [] = {"dbg", "inf", "war", "err", "crt"}
  日志等级标签数组

详细描述

日志

成员函数说明

void Log::printf (int *level*, char const * *file*, int *line*, char const * *format*, ...) const

按格式打印日志

参数:

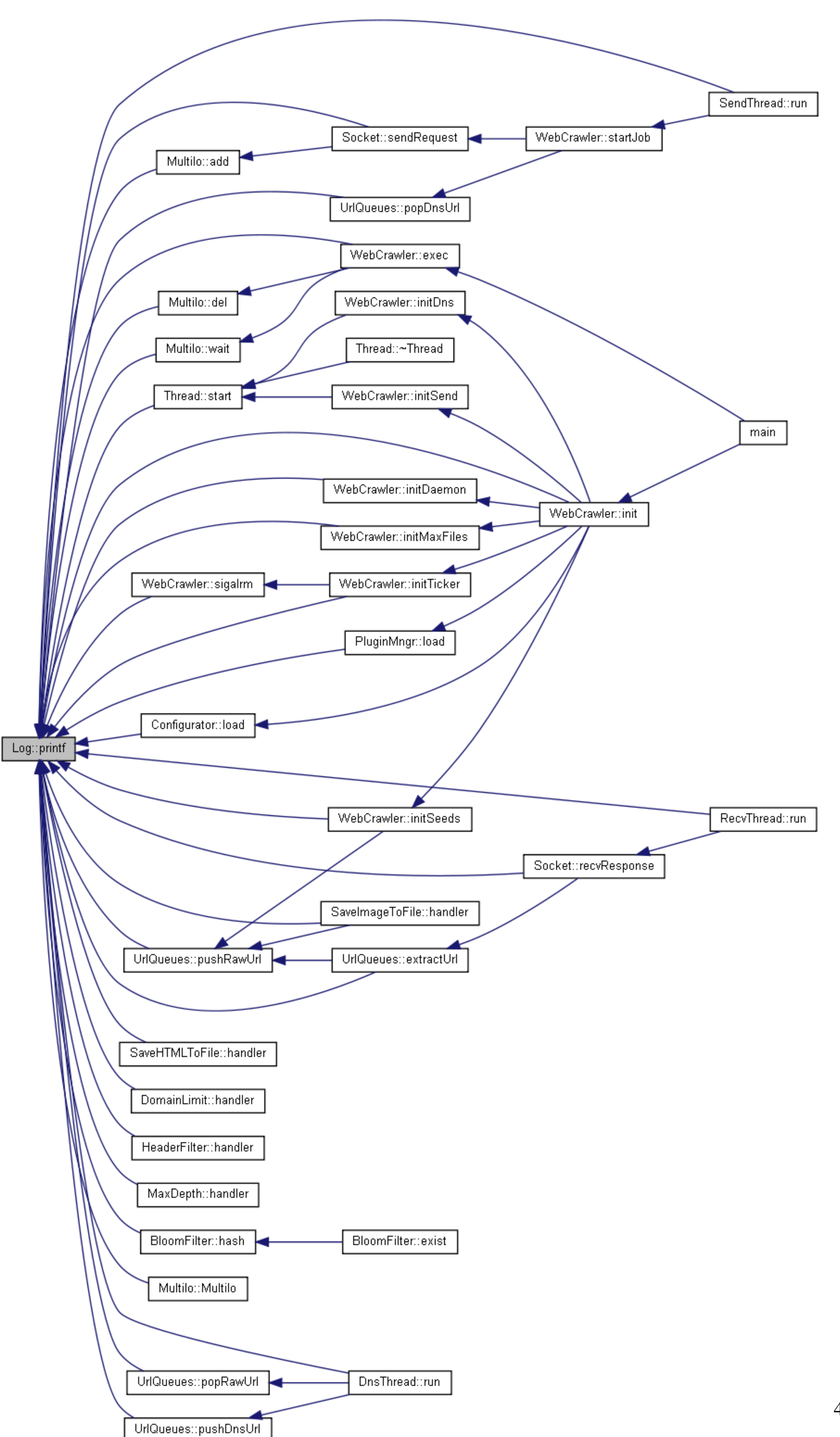
in	*level*	日志等级
in	*file*	源码文件
in	*line*	源码行号
in	*format*	格式化串

参考 g_app, LEVEL_ERR, WebCrawler::m_cfg, Configurator::m_logLevel , 以及 s_levels.

参考自 MultiIo::add(), MultiIo::del(), WebCrawler::exec(), UrlQueues::extractUrl(), SaveHTMLToFile::handler(), SaveImageToFile::handler(), DomainLimit::handler(), HeaderFilter::handler(), MaxDepth::handler(), BloomFilter::hash(), WebCrawler::init(), WebCrawler::initDaemon(), WebCrawler::initMaxFiles(), WebCrawler::initSeeds(), WebCrawler::initTicker(), PluginMngr::load(), Configurator::load(), MultiIo::MultiIo(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), Socket::recvResponse(), DnsThread::run(), SendThread::run(), RecvThread::run(), Socket::sendRequest(), WebCrawler::sigalrm(), Thread::start() , 以及 MultiIo::wait().

```
18           {
19     // 若日志等级不低于配置文件中的"最低日志等级"
20     if (level >= g_app->m_cfg.m_logLevel) {
21         // 格式化当前系统日期和时间字符串
22         char dateTime[32];
23         time_t now = time (NULL);
24         strftime (dateTime, sizeof (dateTime),
25             "%Y-%m-%d %H:%M:%S", localtime (&now));
26         // 打印日志头:
27         // [日期时间][日志等级][pid=进程标识 tid=线程标识][文件:行号]
28         fprintf (stdout, "[%s][%s][pid=%d tid=%lu][%s:%d]\n",
29             dateTime, s_levels[level], getpid (), pthread_self (),
30             file, line);
31
32         // 变长参数表
33         va_list ap;
34         // 用format以后的参数初始化变长参数表
35         va_start (ap, format);
36         // 按format格式打印变长参数表中的内容
37         vfprintf (stdout, format, ap);
38         // 销毁变长参数表
39         va_end (ap);
40
41         // 换行并打印空行
42         fprintf (stdout, "\n\n");
43     }
44
45     // 若日志等级不低于一般错误
46     if (level >= LEVEL_ERR)
47         // 提前终止进程
48         exit (EXIT_FAILURE);
49 }
```

这是这个函数的调用关系图:


SendThread::run
Socket::sendRequest
WebCrawler::startJob
MultiIo::add
UrlQueues::popDnsUrl
WebCrawler::exec
MultiIo::del
WebCrawler::initDns
MultiIo::wait
Thread::~Thread
Thread::start
WebCrawler::initSend
main
WebCrawler::initDaemon
WebCrawler::init
WebCrawler::initMaxFiles
WebCrawler::sigalrm
WebCrawler::initTicker
PluginMngr::load
Configurator::load
Log::printf
WebCrawler::initSeeds
RecvThread::run
Socket::recvResponse
SaveImageToFile::handler
UrlQueues::pushRawUrl
UrlQueues::extractUrl
SaveHTMLToFile::handler
DomainLimit::handler
HeaderFilter::handler
MaxDepth::handler
BloomFilter::hash
BloomFilter::exist
MultiIo::MultiIo
UrlQueues::popRawUrl
DnsThread::run
UrlQueues::pushDnsUrl

类成员变量说明

int const Log::LEVEL_CRT = 4 [static]

日志等级 - 致命错误

int const Log::LEVEL_DBG = 0 [static]

日志等级 - 调试

参考自 WebCrawler::exec(), UrlQueues::extractUrl(), SaveHTMLToFile::handler(), SaveImageToFile::handler(), HeaderFilter::handler(), PluginMngr::load(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), Socket::recvResponse(), DnsThread::run(), SendThread::run(), RecvThread::run() , 以及 Socket::sendRequest().

int const Log::LEVEL_ERR = 3 [static]

日志等级 - 一般错误

参考自 UrlQueues::extractUrl(), SaveImageToFile::handler(), BloomFilter::hash(), WebCrawler::init(), WebCrawler::initDaemon(), WebCrawler::initSeeds(), WebCrawler::initTicker(), PluginMngr::load(), Configurator::load(), MultiIo::MultiIo(), printf() , 以及 Thread::start().

int const Log::LEVEL_INF = 1 [static]

日志等级 - 信息

参考自 WebCrawler::sigalrm().

int const Log::LEVEL_WAR = 2 [static]

日志等级 - 警告

参考自 MultiIo::add(), MultiIo::del(), WebCrawler::exec(), UrlQueues::extractUrl(), SaveImageToFile::handler(), MaxDepth::handler(), SaveHTMLToFile::handler(), DomainLimit::handler(), WebCrawler::initMaxFiles(), Socket::recvResponse(), DnsThread::run(), Socket::sendRequest() , 以及 MultiIo::wait().

char const * Log::s_levels = {"dbg", "inf", "war", "err", "crt"} [static], [private]

日志等级标签数组

从低到高依次为:

- dbg - 调试
- inf - 信息

- war - 警告
- err - 一般错误
- crt - 致命错误

参考自 printf().

该类的文档由以下文件生成:

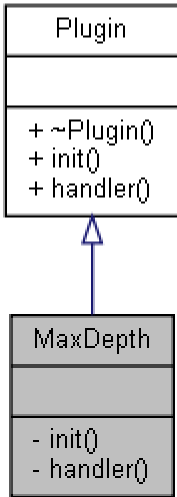
- G:/Projects/Tarena/WebCrawler/teacher/src/Log.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Log.cpp

MaxDepth类 参考

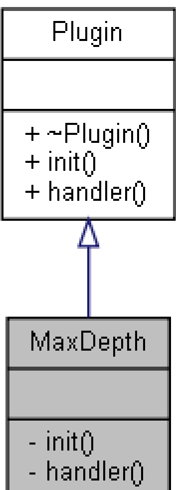
最大深度插件

```
#include <MaxDepth.h>
```

类 MaxDepth 继承关系图:



MaxDepth 的协作图:



Private 成员函数

- bool init (WebCrawler *app)
  插件初始化
- bool handler (void *arg)
  插件处理

额外继承的成员函数

详细描述

最大深度插件

成员函数说明

bool MaxDepth::handler (void * *arg*) [private], [virtual]

插件处理

返回值:

true	成功
false	失败

注解:

根据最大深度插件的功能实现基类中的虚函数

参数:

in,out	*arg*	插件参数

实现了 Plugin.

参 考 Log::LEVEL_WAR, WebCrawler::m_cfg, RawUrl::m_depth, WebCrawler::m_log, Configurator::m_maxDepth, RawUrl::m_strUrl , 以及 Log::printf().

```
34     {
35     // 原始统一资源定位符
36     RawUrl* rawUrl = static_cast<RawUrl*> (arg);
37
38     // 若配置器中的最大递归深度有效，且
39     // 该统一资源定位符的链接深度已超限
40     if (0 <= g_app->m_cfg.m_maxDepth &&
41         g_app->m_cfg.m_maxDepth < rawUrl->m_depth) {
42         // 记录警告日志
43         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
44             "不抓过深(%d>%d)的统一资源定位符\"%s\"", rawUrl->m_depth,
45             g_app->m_cfg.m_maxDepth, rawUrl->m_strUrl.c_str ());
46         // 返回失败，不抓该统一资源定位符
47         return false;
48     }
49
50     // 返回成功，抓取该统一资源定位符
51     return true;
52 }
```

函数调用图:

bool MaxDepth::init (WebCrawler * *app*)[private], [virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

根据最大深度插件的功能实现基类中的虚函数

参数:

in,out	*app*	应用程序对象

实现了 Plugin.

```
20       {
21      // 以统一资源定位符插件的身份
22      // 注册到应用程序对象的插件管理器中
23      (g_app = app)->m_pluginMngr.registerUrlPlugin (this);
24
25      // 返回成功
26      return true;
27  }
```

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.h
- G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.cpp

MultiIo类 参考

多路输入输出

`#include <MultiIo.h>`

MultiIo 的协作图:

MultiIo
- m_epoll
+ MultiIo()
+ ~MultiIo()
+ add()
+ del()
+ wait()

Public 成员函数

- MultiIo (void)
  构造器
- ~MultiIo (void)
  析构器
- bool add (int fd, epoll_event &event) const
  增加需要被关注的输入输出事件
- bool del (int fd, epoll_event &event) const
  删除需要被关注的输入输出事件
- int wait (epoll_event events[], int max, int timeout) const
  等待所关注输入输出事件的发生

Private 属性

- int m_epoll
  多路输入输出对象实例句柄(文件描述符)

详细描述

多路输入输出

构造及析构函数说明

MultiIo::MultiIo (void)

构造器

参考 g_app, Log::LEVEL_ERR, m_epoll, WebCrawler::m_log , 以及 Log::printf().

```
11                      {
```

```
12      // 创建epoll对象，若失败
13      if ((m_epoll = epoll_create1 (0)) == -1)
14          // 记录一般错误日志
15          g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
16              "epoll_create1: %s", strerror (errno));
17  }
```

函数调用图:

Multilo::Multilo → Log::printf

Multilo::~Multilo (void)

析构器

参考 m_epoll.

```
20                      {
21      // 销毁epoll对象
22      close (m_epoll);
23  }
```

成员函数说明

bool Multilo::add (int *fd*, epoll_event & *event*) const

增加需要被关注的输入输出事件

返回值:

true	成功
false	失败

参数:

in	*fd*	发生输入输出事件的文件描述符
in	*event*	事件描述结构

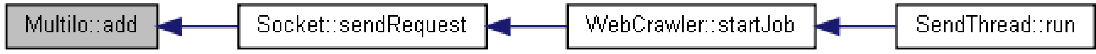
参考 g_app, Log::LEVEL_WAR, m_epoll, WebCrawler::m_log , 以及 Log::printf().

参考自 Socket::sendRequest().

```
30              {
31      // 将文件描述符及其被关注的事件加入epoll，若失败
32      if (epoll_ctl (m_epoll, EPOLL_CTL_ADD, fd, &event) == -1) {
33          // 记录警告日志
34          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
35              "epoll_ctl: %s", strerror (errno));
36          // 返回失败
37          return false;
38      }
39
40      // 返回成功
41      return true;
42  }
```

函数调用图:

这是这个函数的调用关系图:



bool MultiIo::del (int *fd*, epoll_event & *event*) const

删除需要被关注的输入输出事件

返回值:

true	成功
false	失败

参数:

in	*fd*	发生输入输出事件的文件描述符
in	*event*	事件描述结构

参考 g_app, Log::LEVEL_WAR, m_epoll, WebCrawler::m_log , 以及 Log::printf().

参考自 WebCrawler::exec().

```
49          {
50     // 将文件描述符及其被关注的事件从epoll中删除，若失败
51     if (epoll_ctl (m_epoll, EPOLL_CTL_DEL, fd, &event) == -1) {
52         // 记录警告日志
53         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
54             "epoll_ctl: %s", strerror (errno));
55         // 返回失败
56         return false;
57     }
58
59     // 返回成功
60     return true;
61 }
```

函数调用图:

这是这个函数的调用关系图:

int MultiIo::wait (epoll_event *events*[], int *max*, int *timeout*) const

等待所关注输入输出事件的发生

返回值:

>0	处于就绪状态的文件描述符数
0	超时
-1	失败

参数:

out	*events*	事件描述结构数组
in	*max*	事件描述结构数组容量
in	*timeout*	超时毫秒数，0立即超时，-1无限超时

参考 g_app, Log::LEVEL_WAR, m_epoll, WebCrawler::m_log，以及 Log::printf().

参考自 WebCrawler::exec().

```
69          {
70     // 等待epoll中的文件描述符发生所关注的事件
71     int fds = epoll_wait (m_epoll, events, max, timeout);
72     // 若发生除被信号中断以外的错误
73     if (fds == -1 && errno != EINTR)
74         // 记录警告日志
75         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
76             "epoll_wait: %s", strerror (errno));
77
78     // 返回处于就绪状态的文件描述符数，超时返回0，失败返回-1
79     return fds;
80 }
```

函数调用图:

这是这个函数的调用关系图:

类成员变量说明

int MultiIo::m_epoll [private]

多路输入输出对象实例句柄(文件描述符)

参考自 add(), del(), MultiIo(), wait()，以及 ~MultiIo().

该类的文档由以下文件生成:

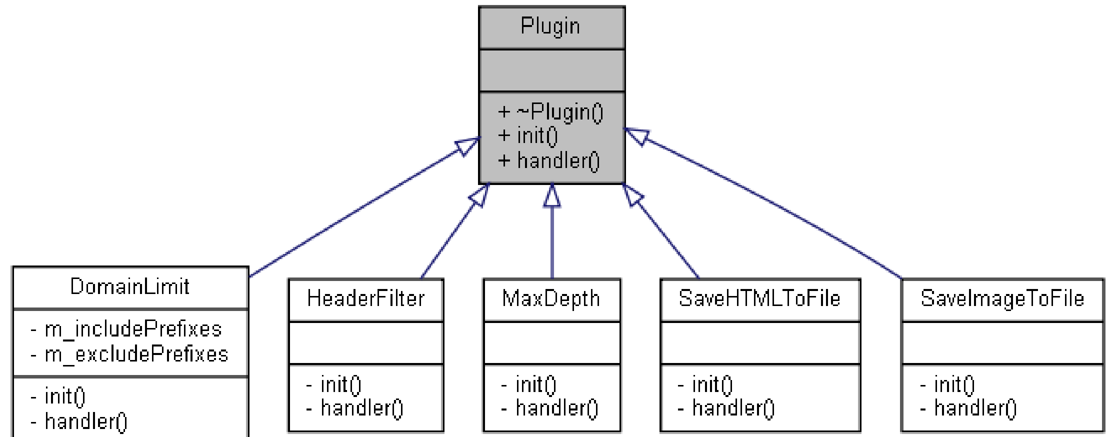
- G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.h
- G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.cpp

Plugin类 参考

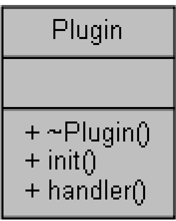
插件接口

```
#include <Plugin.h>
```

类 Plugin 继承关系图:



Plugin 的协作图:



Public 成员函数

- virtual ~Plugin (void)
  析构器
- virtual bool init (WebCrawler *app)=0
  插件初始化
- virtual bool handler (void *arg)=0
  插件处理

详细描述

插件接口

构造及析构函数说明

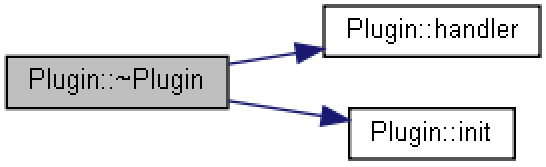
virtual Plugin::~Plugin (void) [inline], [virtual]

析构器

参考 handler() , 以及 init().

```
16 {}
```

函数调用图:



成员函数说明

virtual bool Plugin::handler (void * *arg*) [pure virtual]

插件处理

返回值:

true	成功
false	失败

注解:

纯虚函数，子类根据不同插件的具体功能给出具体实现

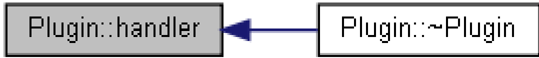
参数:

in,out	*arg*	插件参数

在 DomainLimit, HeaderFilter, MaxDepth, SaveHTMLToFile , 以及 SaveImageToFile 内被实现.

参考自 ~Plugin().

这是这个函数的调用关系图:



virtual bool Plugin::init (WebCrawler * *app*) [pure virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

纯虚函数，子类根据不同的插件功能给出具体实现

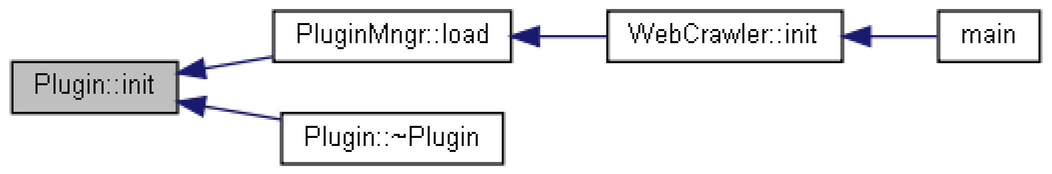
参数:

in,out	*app*	应用程序对象

在 DomainLimit, HeaderFilter, MaxDepth, SaveHTMLToFile , 以及 SaveImageToFile 内被实现.

参考自 PluginMngr::load() , 以及 ~Plugin().

这是这个函数的调用关系图:



该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/src/Plugin.h

PluginMngr类 参考

插件管理器

`#include <PluginMngr.h>`

PluginMngr 的协作图:

PluginMngr

- m_urlPlugins
- m_headerPlugins
- m_htmlPlugins

+ load()
+ registerUrlPlugin()
+ registerHeaderPlugin()
+ registerHtmlPlugin()
+ invokeUrlPlugins()
+ invokeHeaderPlugins()
+ invokeHtmlPlugins()

Public 成员函数

- void load (void)
  加载插件
- void registerUrlPlugin (Plugin *plugin)
  注册统一资源定位符插件
- void registerHeaderPlugin (Plugin *plugin)
  注册超文本传输协议响应包头插件
- void registerHtmlPlugin (Plugin *plugin)
  注册超文本标记语言插件
- bool invokeUrlPlugins (void *arg) const
  调用统一资源定位符插件处理函数
- bool invokeHeaderPlugins (void *arg) const
  调用超文本传输协议响应包头插件处理函数
- void invokeHtmlPlugins (void *arg) const
  调用超文本标记语言插件处理函数

Private 属性

- vector< Plugin * > m_urlPlugins
  统一资源定位符插件向量
- vector< Plugin * > m_headerPlugins
  超文本传输协议响应包头插件向量
- vector< Plugin * > m_htmlPlugins
  超文本标记语言插件向量

详细描述

插件管理器

成员函数说明

bool PluginMngr::invokeHeaderPlugins (void * *arg*) const

调用超文本传输协议响应包头插件处理函数

返回值:

true	成功
false	失败

备注:

依次调用每个超文本传输协议响应包头插件处理函数，只要有一个失败即返回失败

参数:

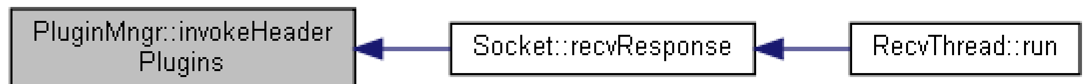
in,out	*arg*	插件参数

参考 m_headerPlugins.

参考自 Socket::recvResponse().

```
110         {
111     // 遍历超文本传输协议响应包头插件向量
112     for (vector<Plugin*>::const_iterator it = m_headerPlugins.begin ();
113         it != m_headerPlugins.end (); ++it)
114         // 依次调用每个超文本传输协议响应
115         // 包头插件的插件处理接口，若失败
116         if (! (*it)->handler (arg))
117             // 返回失败
118             return false;
119
120     // 返回成功
121     return true;
122 }
```

这是这个函数的调用关系图:



void PluginMngr::invokeHtmlPlugins (void * *arg*) const

调用超文本标记语言插件处理函数

备注:

依次调用每个超文本标记语言插件处理函数，忽略其成功失败

参数:

in,out	*arg*	插件参数

参考 m_htmlPlugins.

参考自 Socket::recvResponse().

```
128          {
129     // 遍历超文本标记语言插件向量
130     for (vector<Plugin*>::const_iterator it = m_htmlPlugins.begin ();
131         it != m_htmlPlugins.end (); ++it)
132         // 依次调用每个超文本标记语言插件
133         // 的插件处理接口，不管成功失败
134         (*it)->handler (arg);
135 }
```

这是这个函数的调用关系图:

bool PluginMngr::invokeUrlPlugins (void * *arg*) const

调用统一资源定位符插件处理函数

返回值:

true	成功
false	失败

备注:

依次调用每个统一资源定位符插件处理函数，只要有一个失败即返回失败

参数:

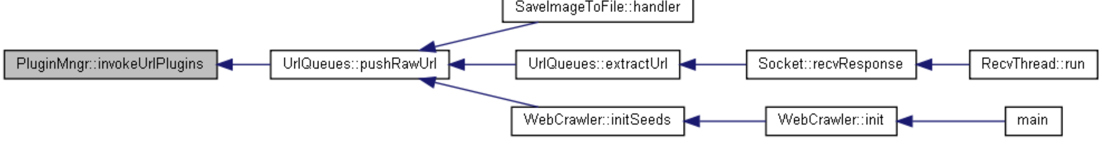
in,out	*arg*	插件参数

参考 m_urlPlugins.

参考自 UrlQueues::pushRawUrl().

```
 91          {
 92     // 遍历统一资源定位符插件向量
 93     for (vector<Plugin*>::const_iterator it = m_urlPlugins.begin ();
 94         it != m_urlPlugins.end (); ++it)
 95         // 依次调用每个统一资源定位符插件的插件处理接口，若失败
 96         if (! (*it)->handler (arg))
 97             // 返回失败
 98             return false;
 99
100     // 返回成功
101     return true;
102 }
```

这是这个函数的调用关系图:



void PluginMngr::load (void)

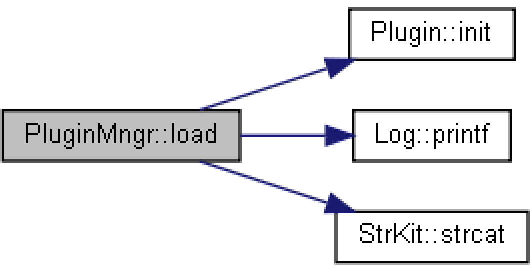
加载插件

参 考 g_app, Plugin::init(), Log::LEVEL_DBG, Log::LEVEL_ERR, WebCrawler::m_cfg, Configurator::m_loadPlugins, WebCrawler::m_log, Configurator::m_pluginsPath, Log::printf() , 以及 StrKit::strcat().

参考自 WebCrawler::init().

```
 13                  {
 14     // 遍历配置器中的插件列表向量
 15     for (vector<string>::const_iterator it =
 16         g_app->m_cfg.m_loadPlugins.begin (); it !=
 17         g_app->m_cfg.m_loadPlugins.end (); ++it) {
 18         // 利用配置器中的插件路径，构造插件共享库路径：
 19         // <插件路径>/<插件名>.so，例如：
 20         // ../plugins/MaxDepth.so
 21         // ../plugins/DomainLimit.so，等等
 22         string path = StrKit::strcat (
 23             g_app->m_cfg.m_pluginsPath.c_str (), "/",
 24             it->c_str (), ".so", NULL);
 25
 26         // 记录调试日志
 27         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
 28             "加载\"%s\"插件", path.c_str ());
 29
 30         // 打开插件共享库
 31         void* handle = dlopen (path.c_str (), RTLD_GLOBAL | RTLD_NOW);
 32         // 若失败
 33         if (! handle)
 34             // 记录一般错误日志
 35             g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
 36                 "dlopen: %s", dlerror ());
 37
 38         // 利用插件名，构造插件对象名：
 39         // g_plugin<插件名>，例如：
 40         // g_pluginMaxDepth
 41         // g_pluginDomainLimit，等等
 42         string symbol = "g_plugin";
 43         symbol += *it;
 44
 45         // 从插件共享库中获取插件对象地址，并转换为其基类类型的指针
 46         Plugin* plugin = (Plugin*)dlsym (handle, symbol.c_str ());
 47         // 若失败
 48         if (! plugin)
 49             // 记录一般错误日志
 50             g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
 51                 "dlsym: %s", dlerror ());
 52
 53         // 通过插件基类调用插件子类的初始化接口
 54         plugin->init (g_app);
 55     }
 56 }
```

函数调用图:



这是这个函数的调用关系图:

void PluginMngr::registerHeaderPlugin (Plugin * *plugin*)

注册超文本传输协议响应包头插件

备注:

超文本传输协议响应包头插件通过此接口将其自身注册到插件管理器中

参数:

in	*plugin*	超文本传输协议响应包头插件

参考 m_headerPlugins.

```
 71     {
 72     // 将超文本传输协议响应包头插件指针压
 73     // 入超文本传输协议响应包头插件向量
 74     m_headerPlugins.push_back (plugin);
 75 }
```

void PluginMngr::registerHtmlPlugin (Plugin * *plugin*)

注册超文本标记语言插件

备注:

超文本标记语言插件通过此接口将其自身注册到插件管理器中

参数:

in	*plugin*	超文本标记语言插件

参考 m_htmlPlugins.

```
 81     {
 82     // 将超文本标记语言插件指针压入超文本标记语言插件向量
 83     m_htmlPlugins.push_back (plugin);
 84 }
```

void PluginMngr::registerUrlPlugin (Plugin * *plugin*)

注册统一资源定位符插件

备注:

统一资源定位符插件通过此接口将其自身注册到插件管理器中

参数:

in	*plugin*	统一资源定位符插件

参考 m_urlPlugins.

```
62      {
63     // 将统一资源定位符插件指针压入统一资源定位符插件向量
64     m_urlPlugins.push_back (plugin);
65 }
```

类成员变量说明

vector<Plugin*> PluginMngr::m_headerPlugins [private]

超文本传输协议响应包头插件向量

参考自 invokeHeaderPlugins(), 以及 registerHeaderPlugin().

vector<Plugin*> PluginMngr::m_htmlPlugins [private]

超文本标记语言插件向量

参考自 invokeHtmlPlugins(), 以及 registerHtmlPlugin().

vector<Plugin*> PluginMngr::m_urlPlugins [private]

统一资源定位符插件向量

参考自 invokeUrlPlugins(), 以及 registerUrlPlugin().

该类的文档由以下文件生成:

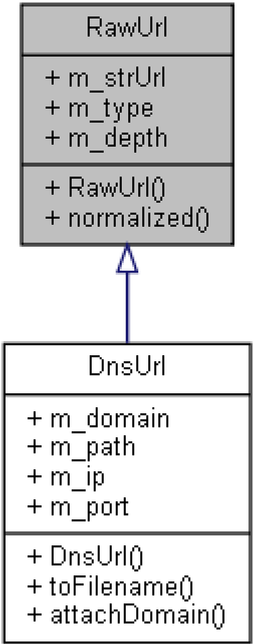
- G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.h
- G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.cpp

RawUrl类 参考

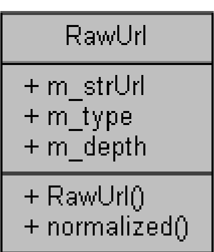
原始统一资源定位符

#include <Url.h>

类 RawUrl 继承关系图:



RawUrl 的协作图:



Public 类型

- enum ETYPE { ETYPE_HTML, ETYPE_IMAGE } *资源类型*

Public 成员函数

- RawUrl (string const &strUrl, ETYPE type=ETYPE_HTML, int depth=0)
  构造器

静态 Public 成员函数

- static bool normalized (string &strUrl)
  规格化

Public 属性

- string m_strUrl
  统一资源定位符字符串
- ETYPE m_type
  资源类型
- int m_depth
  链接深度

详细描述

原始统一资源定位符

成员枚举类型说明

enum RawUrl::ETYPE

资源类型

枚举值

ETYPE_HTML 超文本标记语言

ETYPE_IMAGE 图像

```
14                    {
15          ETYPE_HTML,
16          ETYPE_IMAGE
17      }   ETYPE;
```

构造及析构函数说明

RawUrl::RawUrl (string const & *strUrl*, ETYPE *type* = ETYPE_HTML, int *depth* = 0)

构造器

参数:

in	*strUrl*	统一资源定位符字符串
in	*type*	资源类型
in	*depth*	链接深度

```
16        : m_strUrl (strUrl),  // 初始化统一资源定位符字符串
17          m_type   (type),    // 初始化资源类型
18          m_depth  (depth) {} // 初始化链接深度
```

成员函数说明

bool RawUrl::normalized (string & *strUrl*) [static]

规格化

返回值:

true	成功
false	失败

备注:

删除协议标签(http://或https://)和结尾分隔符(/)

参数:

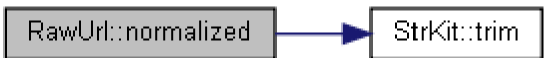
in,out	*strUrl*	待规格化统一资源定位符字符串

参考 StrKit::trim().

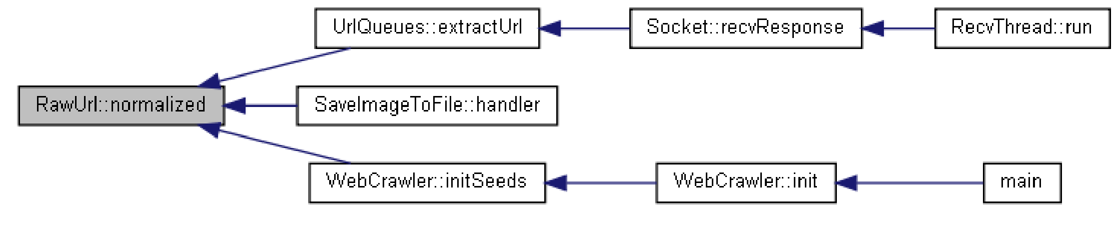
参考自 UrlQueues::extractUrl(), SaveImageToFile::handler() , 以及 WebCrawler::initSeeds().

```
25   {
26     // 将待规格化统一资源定位符字符串复制到临时变量
27     string strTmp = strUrl;
28     // 修剪临时变量中的待规格化统一资源定位符字符串
29     StrKit::trim (strTmp);
30
31     // 若待规格化统一资源定位符字符串为空串
32     if (strTmp.empty ())
33         // 返回失败
34         return false;
35
36     // 若待规格化统一资源定位符字符串以"http://"开头
37     if (! strTmp.find ("http://", 0))
38         // 剪除该协议子串
39         strTmp = strTmp.substr (7);
40     // 否则
41     else
42     // 若待规格化统一资源定位符字符串以"https://"开头
43     if (! strTmp.find ("https://", 0))
44         // 剪除该协议子串
45         strTmp = strTmp.substr (8);
46
47     // 若待规格化统一资源定位符字符串以'/'结尾
48     if (*(strTmp.end () - 1) == '/')
49         // 剪除该路径分隔符
50         strTmp.erase (strTmp.end () - 1);
51
52     // 若待规格化统一资源定位符字符串过长(大于128个字符)
53     if (strTmp.size () > 128)
54         // 返回失败
55         return false;
56
57     // 将临时变量中已规格化的统一资源定位符字符串复制到参数中
58     strUrl = strTmp;
59     // 返回成功
60     return true;
61 }
```

函数调用图:



这是这个函数的调用关系图:



类成员变量说明

int RawUrl::m_depth

链接深度

参考自 UrlQueues::extractUrl(), MaxDepth::handler(), SaveImageToFile::handler() ，以及 DomainLimit::handler().

string RawUrl::m_strUrl

统一资源定位符字符串

参考自 DnsUrl::DnsUrl(), MaxDepth::handler(), DomainLimit::handler(), UrlQueues::popRawUrl() ，以及 UrlQueues::pushRawUrl().

ETYPE RawUrl::m_type

资源类型

参考自 DomainLimit::handler() ，以及 DnsUrl::toFilename().

该类的文档由以下文件生成:

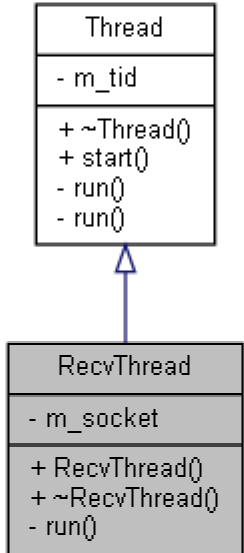
- G:/Projects/Tarena/WebCrawler/teacher/src/Url.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Url.cpp

RecvThread类 参考

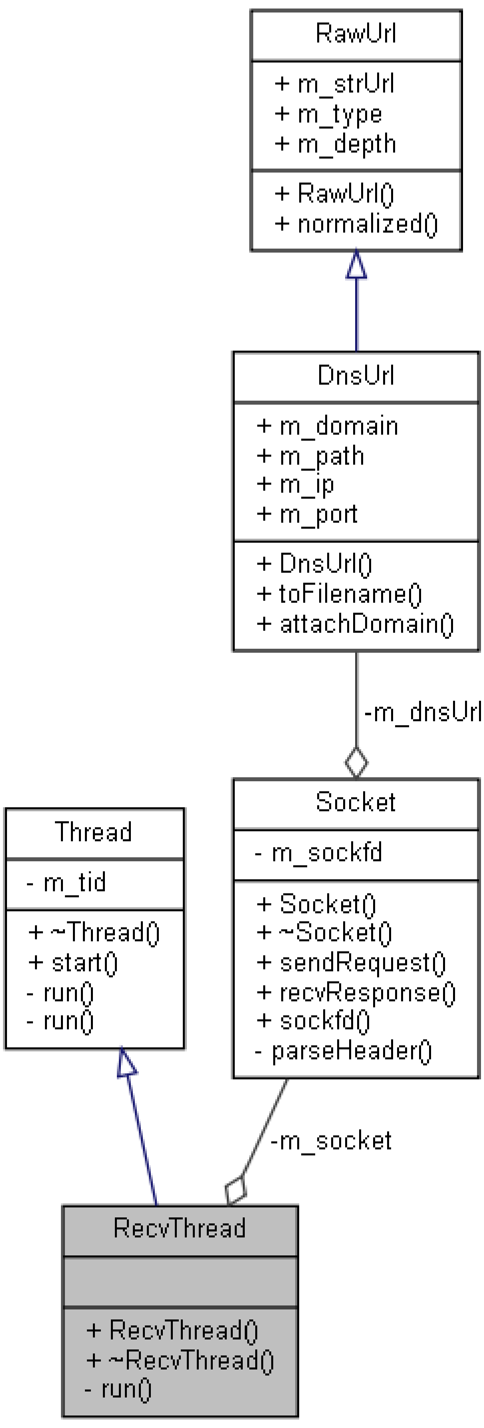
接收线程

`#include <RecvThread.h>`

类 RecvThread 继承关系图:



RecvThread 的协作图:



Public 成员函数

- RecvThread (Socket *socket)
  构造器
- ~RecvThread (void)
  析构器

Private 成员函数

- void * run (void)
  线程处理函数

Private 属性

- Socket * m_socket
  套接字

详细描述

接收线程

构造及析构函数说明

RecvThread::RecvThread (Socket * *socket*)

构造器

参数:

in	*socket*	套接字

```
15        : m_socket (socket) {} // 初始化套接字
```

RecvThread::~RecvThread (void)

析构器

参考 m_socket.

```
18                        {
19      // 销毁套接字
20      delete m_socket;
21 }
```

成员函数说明

void * RecvThread::run (void) [private], [virtual]

线程处理函数

注解:

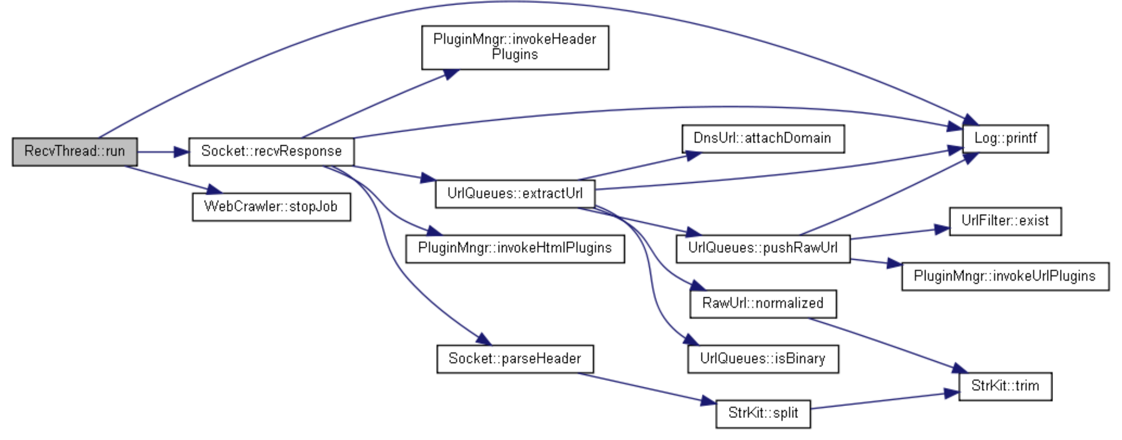
根据接收线程的任务实现基类中的纯虚函数

实现了 Thread.

参考 g_app, Log::LEVEL_DBG, WebCrawler::m_log, m_socket, Log::printf(), Socket::recvResponse(), 以及 WebCrawler::stopJob().

```
25                      {
26      // 记录调试日志
27      g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
28          "接收线程开始");
29
30      // 通过套接字接收超文本传输协议响
31      // 应，根据其执行情况停止抓取任务
32      g_app->stopJob (m_socket->recvResponse ());
33      // 对象自毁
34      delete this;
35
36      // 记录调试日志
37      g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
38          "接收线程终止");
39      // 终止线程
40      return NULL;
41  }
```

函数调用图:



类成员变量说明

Socket* RecvThread::m_socket [private]

套接字

参考自 run(), 以及 ~RecvThread().

该类的文档由以下文件生成:

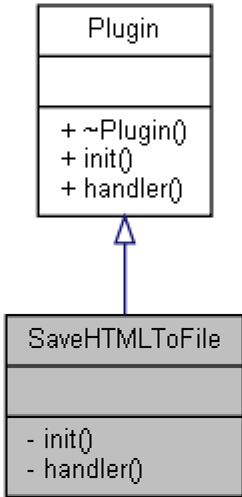
- G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.h
- G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.cpp

SaveHTMLToFile类 参考

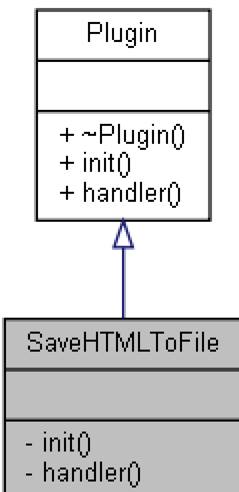
超文本标记语言文件存储插件

`#include <SaveHTMLToFile.h>`

类 SaveHTMLToFile 继承关系图:



SaveHTMLToFile 的协作图:



Private 成员函数

- bool init (WebCrawler *app)
  插件初始化
- bool handler (void *arg)
  插件处理

额外继承的成员函数

详细描述

超文本标记语言文件存储插件

成员函数说明

bool SaveHTMLToFile::handler (void * *arg*) [private], [virtual]

插件处理

返回值:

true	成功
false	失败

注解:

根据超文本标记语言文件存储插件的功能实现基类中的虚函数

参数:

in,out	*arg*	插件参数

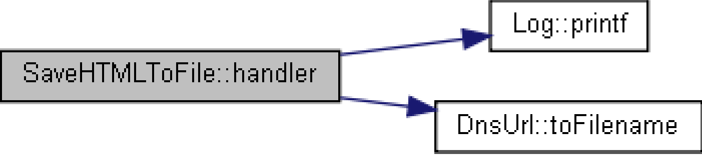
实现了 Plugin.

参考 Log::LEVEL_DBG, Log::LEVEL_WAR, HttpResponse::m_body, HttpHeader::m_contentType, HttpResponse::m_dnsUrl, HttpResponse::m_header, HttpResponse::m_len, WebCrawler::m_log, Log::printf() , 以及 DnsUrl::toFilename().

```
35      {
36      // 超文本传输协议响应
37      HttpResponse* res = static_cast<HttpResponse*> (arg);
38
39      // 若超文本传输协议响应内容类型不是超文本标记语言
40      if (res->m_header.m_contentType.find ("text/html", 0) ==
41          string::npos)
42          // 返回失败
43          return false;
44
45      // 将解析统一资源定位符转换为文件名字符串
46      string filename = res->m_dnsUrl.toFilename ();
47
48      // 根据文件名打开超文本标记语言文件输出流
49      ofstream ofs (filename.c_str (), ios::binary);
50      // 若失败
51      if (! ofs) {
52          // 记录警告日志
53          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
54              "打开文件%s失败: %s", filename.c_str (),
55              strerror (errno));
56          // 返回失败
57          return false;
58      }
59
60      // 将超文本传输协议响应包体写入超文本标记语言文件输出流，若失败
```

```
61      if (! ofs.write (res->m_body, res->m_len)) {
62          // 记录警告日志
63          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
64              "写入文件%s失败: %s", filename.c_str (), strerror (errno));
65          // 关闭超文本标记语言文件输出流
66          ofs.close ();
67          // 删除超文本标记语言文件
68          unlink (filename.c_str ());
69          // 返回失败
70          return false;
71      }
72
73      // 记录调试日志
74      g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
75          "文件%s保存成功", filename.c_str ());
76
77      // 关闭超文本标记语言文件输出流
78      ofs.close ();
79      // 返回成功
80      return true;
81  }
```

函数调用图:



bool SaveHTMLToFile::init (WebCrawler * *app*) [private], [virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

根据超文本标记语言文件存储插件的功能实现基类中的虚函数

参数:

in,out	*app*	应用程序对象

实现了 Plugin.

```
21      {
22      // 以超文本标记语言插件的身份
23      // 注册到应用程序对象的插件管理器中
24      (g_app = app)->m_pluginMngr.registerHtmlPlugin (this);
25
26      // 返回成功
27      return true;
28  }
```

该类的文档由以下文件生成:

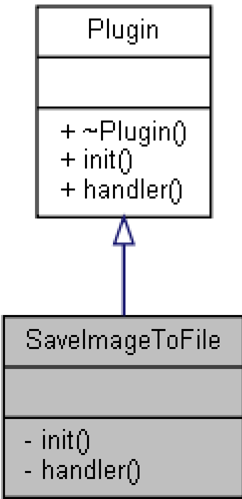
- G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.h
- G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.cpp

SaveImageToFile类 参考

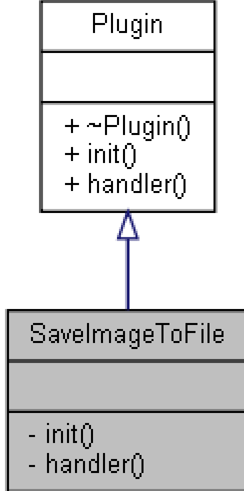
图像文件存储插件

`#include <SaveImageToFile.h>`

类 SaveImageToFile 继承关系图:



SaveImageToFile 的协作图:



Private 成员函数

- bool init (WebCrawler *app)
  插件初始化
- bool handler (void *arg)
  插件处理

额外继承的成员函数

详细描述

图像文件存储插件

成员函数说明

bool SaveImageToFile::handler (void * *arg*) [private], [virtual]

插件处理

返回值:

true	成功
false	失败

注解:

根据图像文件存储插件的功能实现基类中的虚函数

参数:

in,out	*arg*	插件参数

实现了 Plugin.

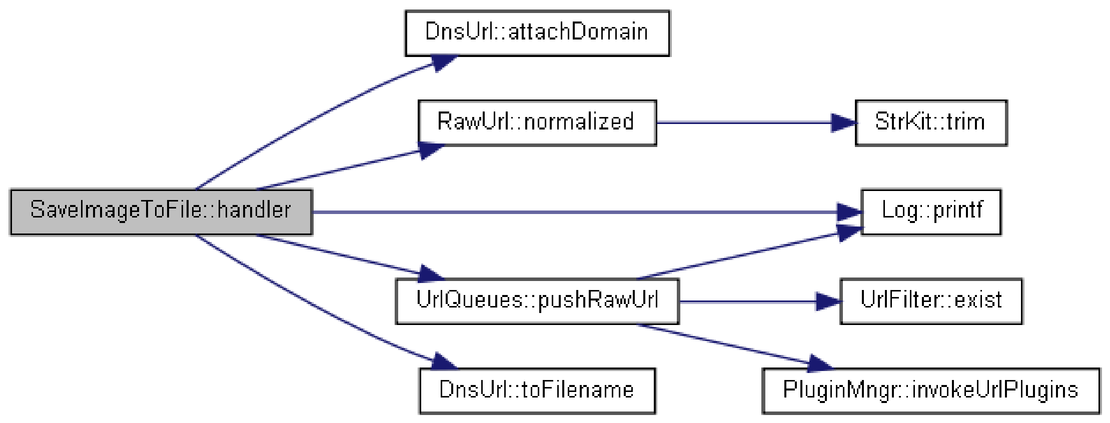
参考 DnsUrl::attachDomain(), RawUrl::ETYPE_IMAGE, Log::LEVEL_DBG, Log::LEVEL_ERR, Log::LEVEL_WAR, HttpResponse::m_body, HttpHeader::m_contentType, RawUrl::m_depth, HttpResponse::m_dnsUrl, HttpResponse::m_header, HttpResponse::m_len, WebCrawler::m_log, WebCrawler::m_urlQueues, RawUrl::normalized(), Log::printf(), UrlQueues::pushRawUrl() , 以及 DnsUrl::toFilename().

```
35      {
36  // 超文本传输协议响应
37  HttpResponse* res = static_cast<HttpResponse*> (arg);
38
39  // 若超文本传输协议响应内容类型是超文本标记语言
40  if (res->m_header.m_contentType.find ("text/html", 0) !=
41      string::npos) {
42      // 正则表达式
43      regex_t ex;
44
45      // 编译正则表达式: <img [^>]*src="\s*\([^ >"]*\)\s*"
46      //     \s - 匹配任意空白字符(空格、制表、换页等)
47      //      * - 重复前一个匹配项任意次
48      //     \( - 子表达式左边界
49      //     \) - 子表达式右边界
50      // [^ >"] - 匹配任意不是空格、大于号和双引号的字符
51      int error = regcomp (&ex,
52          "<img [^>]*src=\"\\s*\\([^ >\"]*\\)\\s*\"", 0);
53      // 若失败
54      if (error) {
55          // 错误信息缓冲区
56          char errInfo[1024];
57          // 获取正则表达式编译错误信息
```

```
 58            regerror (error, &ex, errInfo, sizeof (errInfo) /
 59                sizeof (errInfo[0]));
 60            // 记录一般错误日志
 61            g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
 62                "regcomp: %s", errInfo);
 63        }
 64
 65        // 超文本标记语言页面内容字符串
 66        char const* html = res->m_body;
 67        // 匹配集合
 68        regmatch_t match[2];
 69
 70        // 在超文本标记语言页面内容字符串中,
 71        // 查找所有与正则表达式匹配的内容
 72        while (regexec (&ex, html, sizeof (match) /
 73            sizeof (match[0]), match, 0) != REG_NOMATCH) {
 74            // regex : <img [^>]*src="\s*\([^ >"]*\)\s*"
 75            // html  : ...<img src="  /images/potalaka.jpg  "...
 76            //            |           |<----match[1]---->|  |
 77            //            |         rm_so             rm_eo|
 78            //            |<--------------match[0]--------->|
 79            //          rm_so                             rm_eo
 80
 81            // 匹配子表达式的内容首地址
 82            html += match[1].rm_so;
 83            // 匹配子表达式的内容字符数
 84            size_t len = match[1].rm_eo - match[1].rm_so;
 85            // 匹配子表达式的内容字符串,即图像源中的统一资源定位符
 86            string strUrl (html, len);
 87            // 移至匹配主表达式的内容后,以备在下一轮循环中继续查找
 88            html += len + match[0].rm_eo - match[1].rm_eo;
 89
 90            // 若添加域名失败
 91            if (! res->m_dnsUrl.attachDomain (strUrl))
 92                // 继续下一轮循环
 93                continue;
 94
 95            // 记录调试日志
 96            g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
 97                "抽取到一个深度为%d的统一资源定位符\"%s\"",
 98                res->m_dnsUrl.m_depth, strUrl.c_str ());
 99
100            // 若规格化失败
101            if (! RawUrl::normalized (strUrl)) {
102                // 记录警告日志
103                g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
104                    "规格化统一资源定位符\"%s\"失败", strUrl.c_str ());
105                // 继续下一轮循环
106                continue;
107            }
108
109            // 压入原始统一资源定位符队列
110            g_app->m_urlQueues.pushRawUrl (RawUrl (strUrl,
111                RawUrl::ETYPE_IMAGE, res->m_dnsUrl.m_depth));
112        }
113
114        // 释放正则表达式
115        regfree (&ex);
116    }
117    // 否则,若超文本传输协议响应内容类型是图像
118    else if (res->m_header.m_contentType.find (
119        "image", 0) != string::npos) {
120        // 将解析统一资源定位符转换为文件名字符串
121        string filename = res->m_dnsUrl.toFilename ();
122
123        // 根据文件名打开图像文件输出流
124        ofstream ofs (filename.c_str (), ios::binary);
```

```
125         // 若失败
126         if (! ofs) {
127             // 记录警告日志
128             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
129                 "打开文件%s失败: %s", filename.c_str (),
130                 strerror (errno));
131             // 返回失败
132             return false;
133         }
134 
135         // 将超文本传输协议响应包体写入图像文件输出流，若失败
136         if (! ofs.write (res->m_body, res->m_len)) {
137             // 记录警告日志
138             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
139                 "写入文件%s失败: %s", filename.c_str (), strerror (errno));
140             // 关闭图像文件输出流
141             ofs.close ();
142             // 删除图像文件
143             unlink (filename.c_str ());
144             // 返回失败
145             return false;
146         }
147 
148         // 记录调试日志
149         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
150             "文件%s保存成功", filename.c_str ());
151 
152         // 关闭图像文件输出流
153         ofs.close ();
154     }
155 
156     // 返回成功
157     return true;
158 }
```

函数调用图:



bool SaveImageToFile::init (WebCrawler * *app*)[private], [virtual]

插件初始化

返回值:

true	成功
false	失败

注解:

根据图像文件存储插件的功能实现基类中的虚函数

参数:

in,out	*app*	应用程序对象

实现了 Plugin.

```
21      {
22     // 以超文本标记语言插件的身份
23     // 注册到应用程序对象的插件管理器中
24     (g_app = app)->m_pluginMngr.registerHtmlPlugin (this);
25
26     // 返回成功
27     return true;
28 }
```

该类的文档由以下文件生成:

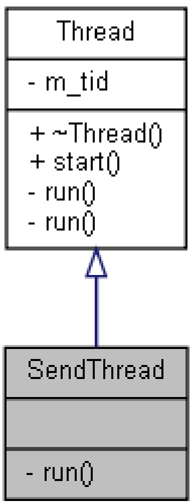
- G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.h
- G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.cpp

SendThread类 参考

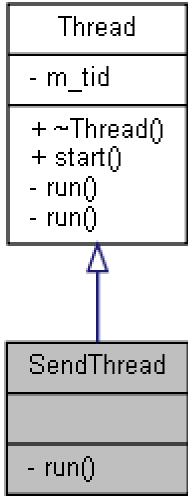
发送线程

`#include <SendThread.h>`

类 SendThread 继承关系图:



SendThread 的协作图:



Private 成员函数

- void * run (void)
  线程处理函数

额外继承的成员函数

详细描述

发送线程

成员函数说明

void * SendThread::run (void) [private], [virtual]

线程处理函数

注解:

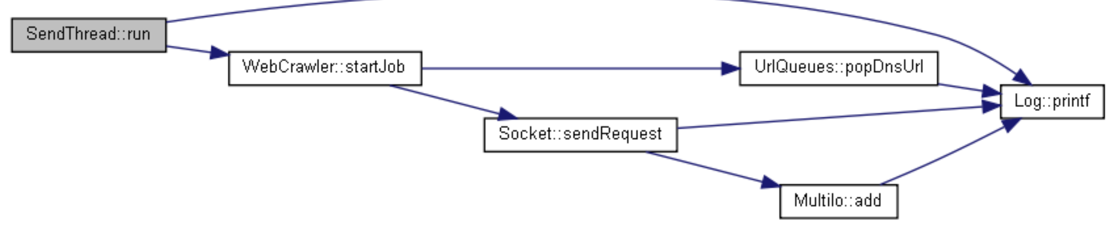
根据发送线程的任务实现基类中的纯虚函数

实现了 Thread.

参考 g_app, Log::LEVEL_DBG, WebCrawler::m_log, Log::printf() , 以及 WebCrawler::startJob().

```
12                        {
13     // 记录调试日志
14     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
15         "发送线程开始");
16
17     // 无限循环
18     for (;;)
19         // 启动一个抓取任务
20         g_app->startJob ();
21
22     // 记录调试日志
23     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
24         "接收线程终止");
25     // 终止线程
26     return NULL;
27 }
```

函数调用图:



该类的文档由以下文件生成:

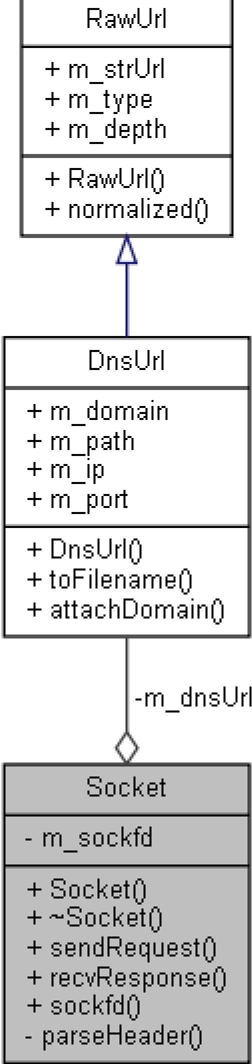
- G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.h
- G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.cpp

Socket类 参考

套接字

`#include <Socket.h>`

Socket 的协作图:



Public 成员函数

- Socket (DnsUrl const &dnsUrl)
  构造器
- ~Socket (void)
  析构器
- bool sendRequest (void)
  发送超文本传输协议请求
- bool recvResponse (void)

接收超文本传输协议响应

- int sockfd (void) const
  获取套接字描述符

Private 成员函数

- HttpHeader parseHeader (string str) const
  解析超文本传输协议响应包头

Private 属性

- int m_sockfd
  套接字描述符
- DnsUrl m_dnsUrl
  服务器统一资源定位符

详细描述

套接字

构造及析构函数说明

Socket::Socket (DnsUrl const & *dnsUrl*)

构造器

参数:

in	*dnsUrl*	服务器统一资源定位符

```
15      : m_sockfd (-1),       // 套接字描述符缺省为无效
16        m_dnsUrl (dnsUrl) {} // 初始化服务器统一资源定位符
```

Socket::~Socket (void)

析构器

参考 m_sockfd.

```
19                  {
20     // 若套接字描述符有效
21     if (m_sockfd >= 0) {
22         // 关闭套接字描述符
23         close (m_sockfd);
24         // 将套接字描述符设置为无效
25         m_sockfd = -1;
26     }
27 }
```

成员函数说明

HttpHeader Socket::parseHeader (string *str*) const [private]

解析超文本传输协议响应包头

返回:

超文本传输协议响应包头

参数:

in	*str*	超文本传输协议响应包头字符串

参考 HttpHeader::m_contentType, HttpHeader::m_statusCode , 以及 StrKit::split().

参考自 recvResponse().

```
305         {
306     // 超文本传输协议响应包头
307     HttpHeader header = {};
308
309     // 超文本传输协议响应包头实例:
310     //
311     // HTTP/1.1 200 OK
312     // Server: nginx
313     // Date: Wed, 26 Oct 2016 10:52:04 GMT
314     // Content-Type: text/html;charset=UTF-8
315     // Connection: close
316     // Vary: Accept-Encoding
317     // Server-Host: classa-study30
318     // Set-Cookie: Domain=study.163.com
319     // Cache-Control: no-cache
320     // Pragma: no-cache
321     // Expires: -1
322     // Content-Language: en-US
323
324     // 在超文本传输协议响应包头字符串中查找第一个回车换行
325     string::size_type pos = str.find ("\r\n", 0);
326     // 若找到了
327     if (pos != string::npos) {
328         // 拆分超文本传输协议响应包头的第一
329         // 行,以空格为分隔符,最多拆分两次
330         // HTTP/1.1 200 OK
331         //    0    ^ 1 ^ 2
332         vector<string> strv = StrKit::split (
333             str.substr (0, pos), " ", 2);
334         // 若成功拆分出三个子串
335         if (strv.size () ==  3)
336             // 其中的第二个子串即为状态码
337             header.m_statusCode = atoi (strv[1].c_str ());
338         // 否则
339         else
340             // 取状态码为600,即"无法解析的响应包头"
341             header.m_statusCode = 600;
342         // 截取除第一行(含回车换行)以外的其余内容,继续解析
343         str = str.substr (pos + 2);
344     }
345
346     // 在超文本传输协议响应包头字符串中逐个查找回车换行
347     while ((pos = str.find ("\r\n", 0)) != string::npos) {
348         // 拆分超文本传输协议响应包头字符串中的
349         // 每一行,以冒号为分隔符,最多拆分一次
```

```
350         vector<string> strv = StrKit::split (
351             str.substr (0, pos), ":", 1);
352         // 若成功拆分出键和值两个子串，且键为"content-type"
353         if (strv.size () == 2 && ! strcasecmp (
354             strv[0].c_str (), "content-type")) {
355             // 则值为"内容类型"
356             header.m_contentType = strv[1];
357             break;
358         }
359         // 截取除该行(含回车换行)以外的其余内容，继续解析
360         str = str.substr (pos + 2);
361     }
362 
363     // 返回超文本传输协议响应包头
364     return header;
365 }
```

函数调用图:

这是这个函数的调用关系图:

bool Socket::recvResponse (void)

接收超文本传输协议响应

返回值:

true	成功
false	失败

参　考　UrlQueues::extractUrl(), g_app, PluginMngr::invokeHeaderPlugins(), PluginMngr::invokeHtmlPlugins(), Log::LEVEL_DBG, Log::LEVEL_WAR, HttpResponse::m_body, HttpHeader::m_contentType, m_dnsUrl, HttpResponse::m_header, HttpResponse::m_len, WebCrawler::m_log, WebCrawler::m_pluginMngr, m_sockfd, WebCrawler::m_urlQueues, parseHeader() , 以及 Log::printf().

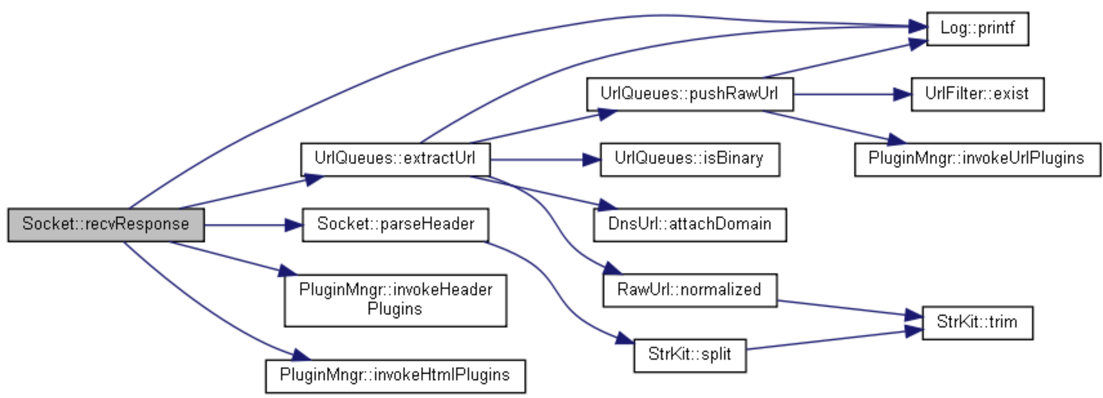
参考自 RecvThread::run().

```
157                              {
158     // 超文本传输协议响应对象
159     HttpResponse res (m_dnsUrl);
160     // 超文本传输协议响应包头尚未解析
161     bool headerParsed = false;
162 
163     // 分多次将超文本传输协议响应包收完
164     for (;;) {
165         // 接收缓冲区
166         char buf[1024] = {};
167         // 接收超文本传输协议响应，注意留一个字符放'\0'
168         ssize_t rlen = recv (m_sockfd, buf,
169             sizeof (buf) - sizeof (buf[0]), 0);
170 
171         // 若失败
172         if (rlen == -1) {
173             // 若因内核接收缓冲区空或被信号中断而导致失败
174             if (errno == EAGAIN || errno == EWOULDBLOCK ||
```

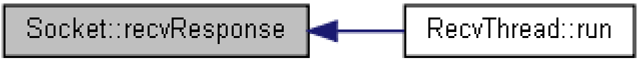
```
175                 errno == EINTR) {
176                 // 延迟重收
177                 usleep (100000);
178                 continue;
179             }
180 
181             // 其它原因导致失败，记录警告日志
182             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
183                 "recv: %s", strerror (errno));
184             // 返回失败
185             return false;
186         }
187 
188         // 若服务器已关闭连接
189         if (! rlen) {
190             // 记录调试日志
191             g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
192                 "接收超文本传输协议响应包体%u字节", res.m_len);
193             /*
194             // 若超文本传输协议响应包体中包含空字符
195             if (strlen (res.m_body) != res.m_len) {
196                 // 记录警告日志
197                 g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
198                     "超文本传输协议响应包体异常(%u!=%u)\n\n%s",
199                     strlen (res.m_body), res.m_len, res.m_body);
200                 // 返回失败
201                 return false;
202             }
203             */
204             // 若超文本传输协议响应的内容类型为超文本标记语言
205             if (res.m_header.m_contentType.find (
206                 "text/html", 0) != string::npos)
207                 // 从超文本标记语言页面内容中抽取统一资源定位符
208                 g_app->m_urlQueues.extractUrl (res.m_body, m_dnsUrl);
209 
210             // 调用超文本标记语言插件处理函数
211             g_app->m_pluginMngr.invokeHtmlPlugins (&res);
212             // 超文本传输协议响应接收并处理完毕，跳出循环
213             break;
214         }
215         /*
216         // 记录调试日志
217         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
218             "接收超文本传输协议响应%u字节", rlen);
219         // 按十六进制打印接收到的数据
220         g_app->m_log.printh (buf, rlen);
221         */
222         // 扩展超文本传输协议响应包体缓冲区，以容纳新
223         // 接收到的响应数据，注意多分配一个字符放'\0'
224         res.m_body = (char*)realloc (res.m_body,
225             res.m_len + rlen + sizeof (res.m_body[0]));
226         // 将新接收到的响应数据，连同终止空字符('\0')一起，
227         // 从接收缓冲区复制到超文本传输协议响应包体缓冲区
228         memcpy (res.m_body + res.m_len, buf,
229             rlen + sizeof (res.m_body[0]));
230         // 增加超文本传输协议响应包体长度
231         res.m_len += rlen;
232 
233         // 若超文本传输协议响应包头尚未解析
234         if (! headerParsed) {
235             // 在已收到的超文本传输协议响应中查找包
236             // 头结束标志————连续出现的两个回车换行
237             char* p = strstr (res.m_body, "\r\n\r\n");
238             // 若找到了
239             if (p) {
240                 // 截取超文本传输协议响应包头
241                 string header (res.m_body, p - res.m_body);
```

```
242
243                 // 记录调试日志
244                 g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
245                     "接收超文本传输协议响应包头%u字节\n\n%s",
246                     header.size (), header.c_str ());
247
248                 // 解析超文本传输协议响应包头
249                 res.m_header = parseHeader (header);
250                 // 调用超文本传输协议响应包头插件处理函数，若失败
251                 if (! g_app->m_pluginMngr.invokeHeaderPlugins (
252                     &res.m_header))
253                     // 返回失败
254                     return false;
255
256                 // 超文本传输协议响应包头已被解析
257                 headerParsed = true;
258
259                 // 跳过超文本传输协议响应包头结束标志————
260                 // 连续出现的两个回车换行————共四个字节，
261                 // 得到超文本传输协议响应包体在超文本传
262                 // 输协议响应包体缓冲区中的起始地址
263                 p += 4;
264                 // 计算已接收超文本传输协议响应包体长度
265                 res.m_len -= p - res.m_body;
266                 // 分配足以容纳已接收超文本传输协议响应
267                 // 包体(含终止空字符('\0'))的临时缓冲区
268                 char* tmp = new char[
269                     res.m_len + sizeof (res.m_body[0])];
270                 // 将已接收超文本传输协议响应包体，连同
271                 // 终止空字符('\0')一起，从超文本传输协
272                 // 议响应包体缓冲区复制到临时缓冲区
273                 memcpy (tmp, p,
274                     res.m_len + sizeof (res.m_body[0]));
275                 // 将已接收超文本传输协议响应包体，连同
276                 // 终止空字符('\0')一起，从临时缓冲区复
277                 // 制到超文本传输协议响应包体缓冲区
278                 memcpy (res.m_body, tmp,
279                     res.m_len + sizeof (res.m_body[0]));
280                 // 释放临时缓冲区
281                 delete[] tmp;
282                 // 借助临时缓冲区移动位于超文本传输协议
283                 // 响应包体缓冲区中的已接收超文本传输协
284                 // 议响应包体，可以有效规避源内存和目的
285                 // 内存间发生重叠覆盖的潜在风险
286             }
287         }
288     }
289
290     // 返回成功
291     return true;
292 }
```

函数调用图:



这是这个函数的调用关系图:



bool Socket::sendRequest (void)

发送超文本传输协议请求

返回值:

true	成功
false	失败

参考 MultiIo::add(), g_app, Log::LEVEL_DBG, Log::LEVEL_WAR, m_dnsUrl, DnsUrl::m_domain, DnsUrl::m_ip, WebCrawler::m_log, WebCrawler::m_multiIo, DnsUrl::m_path, DnsUrl::m_port, m_sockfd , 以及 Log::printf().

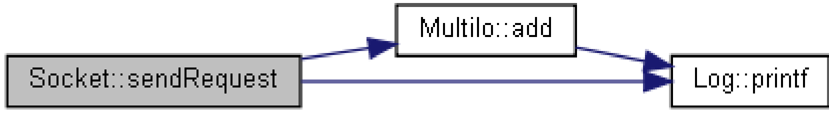
参考自 WebCrawler::startJob().

```
31                          {
32      // 创建套接字(TCP)，并获得其描述符，若失败
33      if ((m_sockfd = socket (PF_INET, SOCK_STREAM, 0)) == -1) {
34          // 记录警告日志
35          g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
36              "socket: %s", strerror (errno));
37          // 返回失败
38          return false;
39      }
40
41      // 记录调试日志
42      g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
43          "创建套接字%d成功", m_sockfd);
44
45      // 服务器地址结构
46      sockaddr_in addr;
47      // 清零
48      bzero (&addr, sizeof (addr));
49      // 地址族
50      addr.sin_family = AF_INET;
51      // 端口号
52      addr.sin_port = htons (m_dnsUrl.m_port);
53      // 将点分十进制字符串形式的IPv4地址转换为32位无符号整数，若失败
54      if (! inet_aton (m_dnsUrl.m_ip.c_str (), &(addr.sin_addr))) {
55          // 记录警告日志
```

```
 56         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 57             "inet_aton: %s", strerror (errno));
 58         // 返回失败
 59         return false;
 60     }
 61
 62     // 向服务器发起连接请求，若连接失败
 63     if (connect (m_sockfd, (sockaddr*)&addr, sizeof (addr)) == -1) {
 64         // 记录警告日志
 65         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 66             "connect: %s", strerror (errno));
 67         // 返回失败
 68         return false;
 69     }
 70
 71     // 记录调试日志
 72     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
 73         "连接服务器\"%s\"成功", m_dnsUrl.m_ip.c_str ());
 74
 75     // 获取套接字的状态标志
 76     int flags = fcntl (m_sockfd, F_GETFL);
 77     // 若失败
 78     if (flags == -1) {
 79         // 记录警告日志
 80         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 81             "fcntl: %s", strerror (errno));
 82         // 返回失败
 83         return false;
 84     }
 85
 86     // 为套接字增加非阻塞状态标志，若失败
 87     if (fcntl (m_sockfd, F_SETFL, flags | O_NONBLOCK) == -1) {
 88         // 记录警告日志
 89         g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
 90             "fcntl: %s", strerror (errno));
 91         // 返回失败
 92         return false;
 93     }
 94
 95     // 输出字符串流
 96     ostringstream oss;
 97     // 在输出字符串流中格式化超文本传输协议请求
 98     oss <<
 99         "GET /" << m_dnsUrl.m_path << " HTTP/1.0\r\n"
100         "Host: " << m_dnsUrl.m_domain << "\r\n"
101         "Accept: */*\r\n"
102         "Connection: Keep-Alive\r\n"
103         "User-Agent: Mozilla/5.0 (compatible; Qteqpidspider/1.0;)\r\n"
104         "Referer: " << m_dnsUrl.m_domain << "\r\n\r\n";
105     // 从输出字符串流中获取超文本传输协议请求字符串
106     string request = oss.str ();
107     // 从超文本传输协议请求字符串中获取其内存缓冲区指针
108     char const* buf = request.c_str ();
109     // 成功发送的字节数
110     ssize_t slen;
111
112     // 分多次将超文本传输协议请求包发完
113     for (size_t len = request.size (); len;
114         len -= slen, buf += slen)
115         // 发送超文本传输协议请求，若失败
116         if ((slen = send (m_sockfd, buf, len, 0)) == -1) {
117             // 若因内核发送缓冲区满而导致失败
118             if (errno == EAGAIN || errno == EWOULDBLOCK) {
119                 // 延迟重发
120                 usleep (1000);
121                 slen = 0;
122                 continue;
123             }
```

```
124
125             // 其它原因导致失败，记录警告日志
126             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
127                 "send: %s", strerror (errno));
128             // 返回失败
129             return false;
130         }
131
132     // 记录调试日志
133     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
134         "发送超文本传输协议请求包%u字节\n\n%s", request.size (),
135         request.c_str ());
136
137     // 关注边沿触发的输入事件，并将套接字对象指针存入事件结
138     // 构中。一旦发自服务器的超文本传输协议响应到达，即套接
139     // 字接收缓冲区由空变为非空，MultiIo::wait便会立即发现，
140     // 并在独立的接收线程中通过此套接字对象接收响应数据
141     epoll_event event = {EPOLLIN | EPOLLET, this};
142     // 增加需要被关注的输入输出事件，若失败
143     if (! g_app->m_multiIo.add (m_sockfd, event))
144         // 返回失败
145         return false;
146
147     // 记录调试日志
148     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
149         "关注套接字%d上的I/O事件", m_sockfd);
150
151     // 返回成功
152     return true;
153 }
```

函数调用图:



这是这个函数的调用关系图:

int Socket::sockfd (void) const

获取套接字描述符

返回:

套接字描述符

参考 m_sockfd.

参考自 WebCrawler::exec().

```
296                         {
297     // 返回套接字描述符
298     return m_sockfd;
299 }
```

这是这个函数的调用关系图:

类成员变量说明

DnsUrl Socket::m_dnsUrl [private]

服务器统一资源定位符

参考自 recvResponse() , 以及 sendRequest().

int Socket::m_sockfd [private]

套接字描述符

参考自 recvResponse(), sendRequest(), sockfd() , 以及 ~Socket().

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/src/Socket.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Socket.cpp

StrKit类 参考

字符串工具包

`#include <StrKit.h>`

StrKit 的协作图:

静态 Public 成员函数

- static string strcat (char const *str1, char const *str2,...)
  字符串拼接
- static string & trim (string &str)
  字符串修剪
- static vector< string > split (string const &str, string const &delim, int limit=0)
  字符串拆分

详细描述

字符串工具包

成员函数说明

vector< string > StrKit::split (string const & *str*, string const & *delim*, int *limit* = 0) `[static]`

字符串拆分

返回:

被拆分出的子串向量

备注:

以delim中的字符作为分隔符，对str字符串进行拆分，并对每个被拆分出的子串做修剪，拆分次数不超过limit，除非该参数的值为0

参数:

in	*str*	待拆分字符串

in	*delim*	分隔符字符串
in	*limit*	拆分次数限制

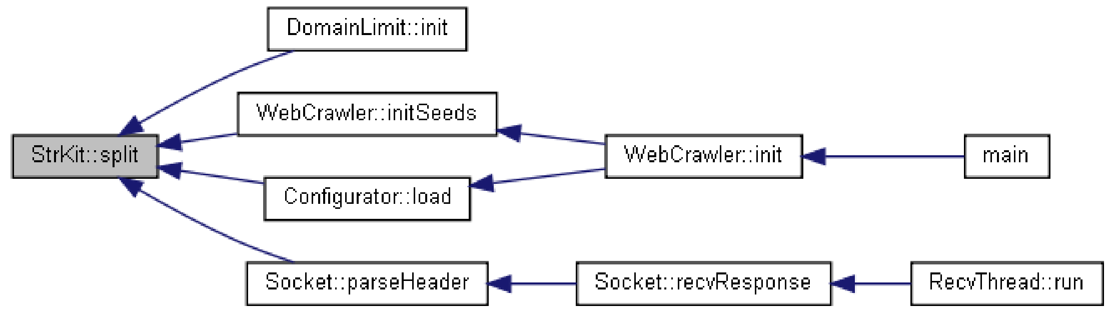
参考 trim().

参 考 自 DomainLimit::init(), WebCrawler::initSeeds(), Configurator::load() , 以 及 Socket::parseHeader().

```
 73      {
 74     // 存放拆分结果的子串向量
 75     vector<string> strv;
 76
 77     // 待拆分字符串临时缓冲区，注意多分配一个字符放'\0'
 78     char temp[str.size () + 1];
 79     // 将待拆分字符串，连同终止空字符('\0')一起，复制到临时缓冲区中
 80     strcpy (temp, str.c_str ());
 81     // strtok要求待拆分字符串所在内存必须可写
 82
 83     //   delim : " ,.:;()$"
 84     //   limit : 3
 85     //    temp : The quick,brown.fox:jumps;over(the)lazy$dog/0
 86     //          t  /t    /t    /t  /t    /t   /t  /t   /t
 87     //          o  0o    0o    0o  0o    0o   0o  0o   0o
 88     //          k   k     k     k   k     k    k   k    k
 89     //          e   e     e     e   e     e    e   e    e
 90     //          n   n     n     n   n     n    n   n    n
 91     //    strv : The
 92     //              quick
 93     //                    brown
 94     //                          fox:jumps;over(the)lazy$dog
 95     // --limit : 2    1     0
 96
 97     // 依次提取待拆分字符串中每个被分隔符字符串中的字符分隔的子串
 98     for (char* token = strtok (temp, delim.c_str ());
 99         token; token = strtok (NULL, delim.c_str ())) {
100         // 被拆分出的子串
101         string part = token;
102         // 经修剪后压入存放拆分结果的子串向量
103         strv.push_back (trim (part));
104
105         // 若拆分次数限制已到(若limit参数取缺省值0，则此条件永远不
106         // 可能满足，即不限制拆分次数)，且本次所拆并非最后一个子串
107         if (! --limit && (token += strlen (token)) - temp <
108             (int)str.size ()) {
109             // 将待拆分字符串的其余部分一次
110             // 性压入存放拆分结果的子串向量
111             strv.push_back (trim (part = ++token));
112             // 提前结束拆分循环
113             break;
114         }
115     }
116
117     // 返回存放拆分结果的子串向量
118     return strv;
119 }
```

函数调用图:

这是这个函数的调用关系图:



string StrKit::strcat (char const * *str1*, char const * *str2*, ...) `[static]`

字符串拼接

返回:

拼接后的字符串

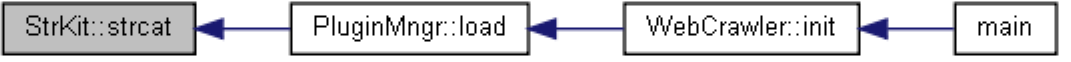
参数:

in	*str1*	字符串1
in	*str2*	字符串2

参考自 PluginMngr::load().

```
16     {
17   // 用第一个参数字符串初始化结果字符串
18   string str = str1;
19   // 将第二个参数字符串拼接到结果字符串末尾
20   str += str2;
21
22   // 变长参数表
23   va_list ap;
24   // 用str2以后的参数初始化变长参数表
25   va_start (ap, str2);
26
27   // 指向变长参数表中参数的指针
28   char const* strx = NULL;
29   // 依次获取变长参数表中的每个字符串参数
30   while (strx = va_arg (ap, char const*))
31       // 拼接到结果字符串末尾
32       str += strx;
33
34   // 销毁变长参数表
35   va_end (ap);
36   // 返回结果字符串
37   return str;
38 }
```

这是这个函数的调用关系图:



string & StrKit::trim (string & *str*) `[static]`

字符串修剪

返回:

被修剪过的参数字符串本身

备注:

截去字符串的首尾空白字符(空格、制表、回车、换行等)

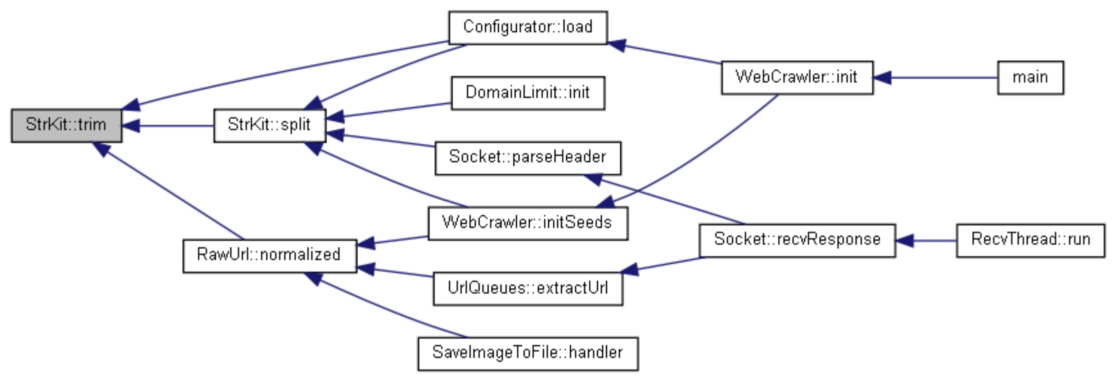
参数:

in,out	*str*	待修剪字符串

参考自 Configurator::load(), RawUrl::normalized() , 以及 split().

```
45     {
46     // 在待修剪字符串查找第一个和最后一个非空白字符
47     string::size_type first = str.find_first_not_of (" \t\r\n"),
48         last = str.find_last_not_of (" \t\r\n");
49
50     // 若没有找到任何非空白字符，说明待修剪字
51     // 符串要么是空串，要么全部由空白字符组成
52     if (first == string::npos || last == string::npos)
53         // 直接清空
54         str.clear ();
55     // 否则
56     else
57         // 截取从第一个到最后一个非空白字符之间的子串
58         str = str.substr (first, last - first + 1);
59
60     // 返回被修剪过的参数字符串本身
61     return str;
62 }
```

这是这个函数的调用关系图:



该类的文档由以下文件生成:

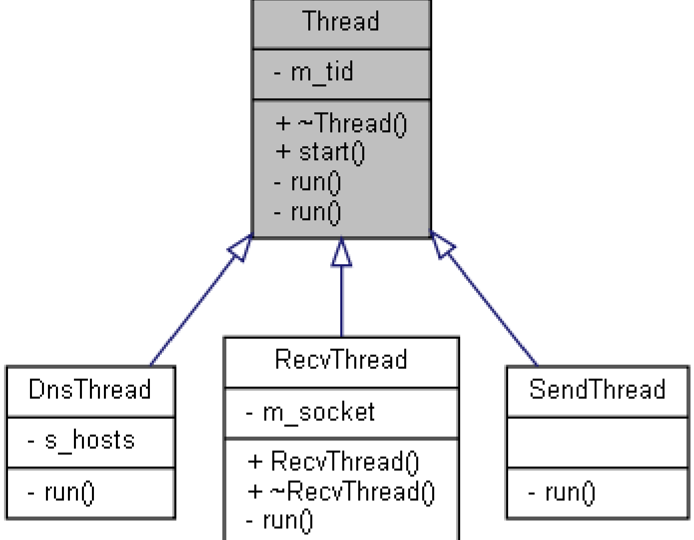
- G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.h
- G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.cpp

Thread类 参考

线程

`#include <Thread.h>`

类 Thread 继承关系图:



Thread 的协作图:

Public 成员函数

- virtual ~Thread (void)
  析构器
- void start (void)
  启动线程

Private 成员函数

- virtual void * run (void)=0
  线程处理函数

静态 Private 成员函数

- static void * run (void *arg)

线程过程函数

Private 属性

- pthread_t m_tid
  线程标识

详细描述

线程

构造及析构函数说明

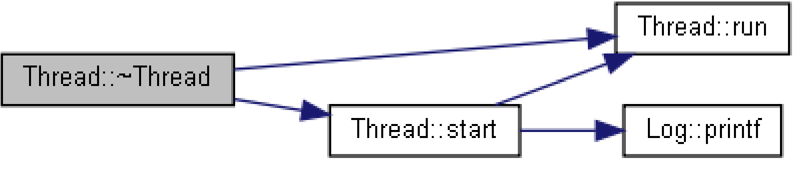
virtual Thread::~Thread (void) [inline], [virtual]

析构器

参考 run() ，以及 start().

```
14 {}
```

函数调用图:



成员函数说明

void * Thread::run (void * *arg*) [static], [private]

线程过程函数

返回:

线程返回值

参数:

in,out	*arg*	线程参数

参考 run().

```
40     {
41     // 通过指向子类对象的基类指针，即创建线程时交给系统内核并由系
42     // 统内核回传给线程过程函数的参数arg，调用在线程抽象基类中声
43     // 明并为其具体子类所覆盖的虚函数run，执行具体线程的具体任务
44     return static_cast<Thread*> (arg)->run ();
45 }
```

函数调用图:

virtual void* Thread::run (void) [private], [pure virtual]

线程处理函数

返回:

处理结果

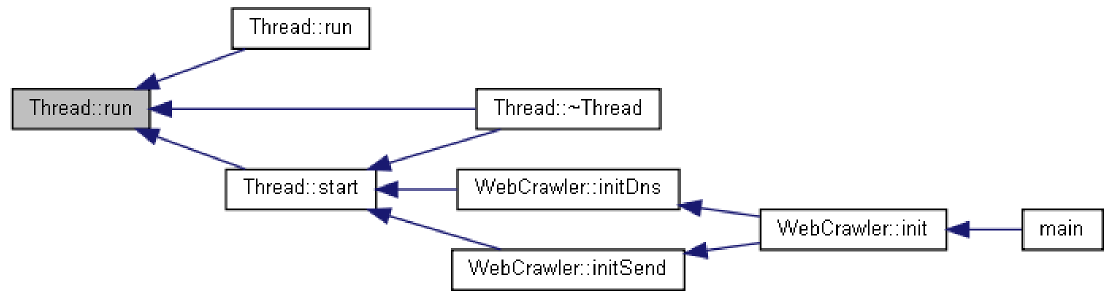
注解:

纯虚函数，子类根据不同线程的具体任务给出具体实现

在 RecvThread, SendThread , 以及 DnsThread 内被实现.

参考自 run(), start() , 以及 ~Thread().

这是这个函数的调用关系图:



void Thread::start (void)

启动线程

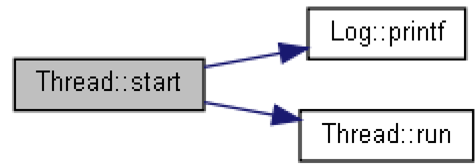
参考 g_app, Log::LEVEL_ERR, WebCrawler::m_log, m_tid, Log::printf() , 以及 run().

参考自 WebCrawler::initDns(), WebCrawler::initSend() , 以及 ~Thread().

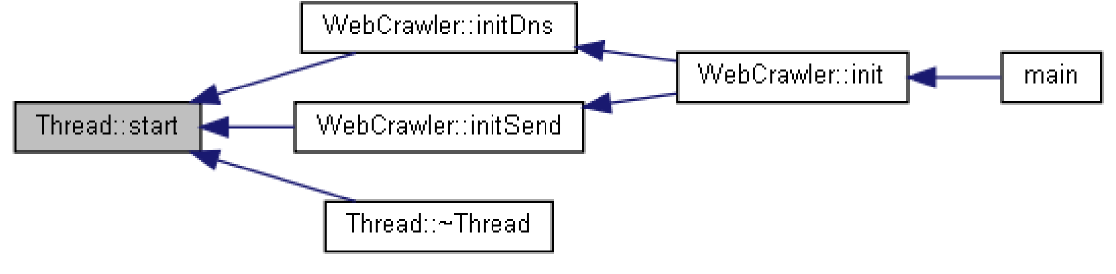
```
12                      {
13      // 线程属性结构
14      pthread_attr_t attr;
15      // 初始化线程属性结构
16      pthread_attr_init (&attr);
17      // 设置线程栈空间大小：1M字节
18      pthread_attr_setstacksize (&attr, 1024 * 1024);
19      // 设置线程分离状态：创建即分离
20      pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
21
22      // 创建线程。线程标识存入成员变量m_tid，线程属性取自先前设
23      // 置好的线程属性结构attr，线程过程函数为静态成员函数run，
24      // 传递给线程过程函数的参数为指向线程(子类)对象的this指针
25      int error = pthread_create (&m_tid, &attr, run, this);
26      // 若失败
27      if (error)
28          // 记录一般错误日志
29          g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
```

```
30              "pthread_create: %s", strerror (error));
31
32      // 销毁线程属性结构
33      pthread_attr_destroy (&attr);
34  }
```

函数调用图:



这是这个函数的调用关系图:



类成员变量说明

pthread_t Thread::m_tid [private]

线程标识

参考自 start().

该类的文档由以下文件生成:

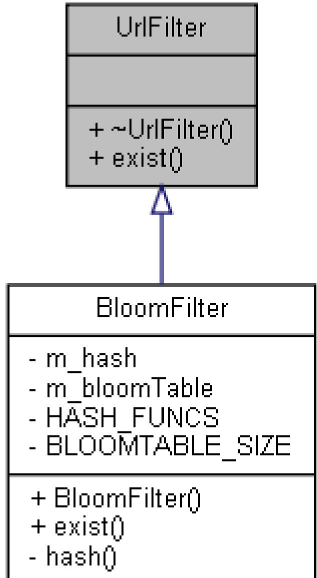
- G:/Projects/Tarena/WebCrawler/teacher/src/Thread.h
- G:/Projects/Tarena/WebCrawler/teacher/src/Thread.cpp

UrlFilter类 参考

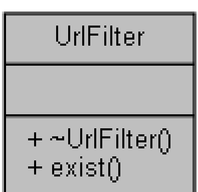
统一资源定位符过滤器接口

`#include <UrlFilter.h>`

类 UrlFilter 继承关系图:



UrlFilter 的协作图:



Public 成员函数

- virtual ~UrlFilter (void)
  析构器
- virtual bool exist (string const &strUrl)=0
  判断某个统一资源定位符是否已经存在

详细描述

统一资源定位符过滤器接口

构造及析构函数说明

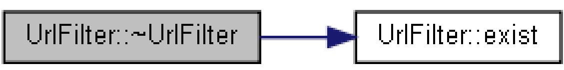
virtual UrlFilter::~UrlFilter (void) [inline], [virtual]

析构器

参考 exist().

```
14 {}
```

函数调用图:



成员函数说明

virtual bool UrlFilter::exist (string const & *strUrl*) [pure virtual]

判断某个统一资源定位符是否已经存在

返回值:

true	存在
false	不存在

注解:

纯虚函数，子类根据不同过滤器的具体策略给出具体实现

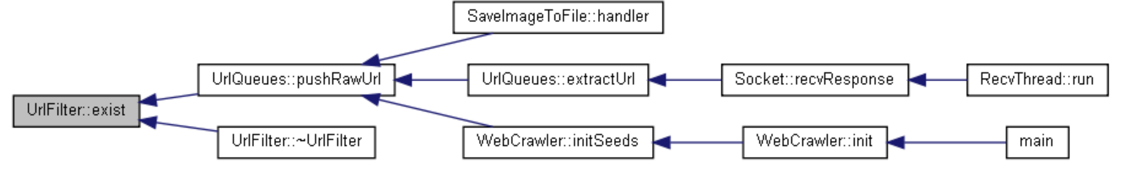
参数:

in	*strUrl*	统一资源定位符

在 BloomFilter 内被实现.

参考自 UrlQueues::pushRawUrl() , 以及 ~UrlFilter().

这是这个函数的调用关系图:



该类的文档由以下文件生成:

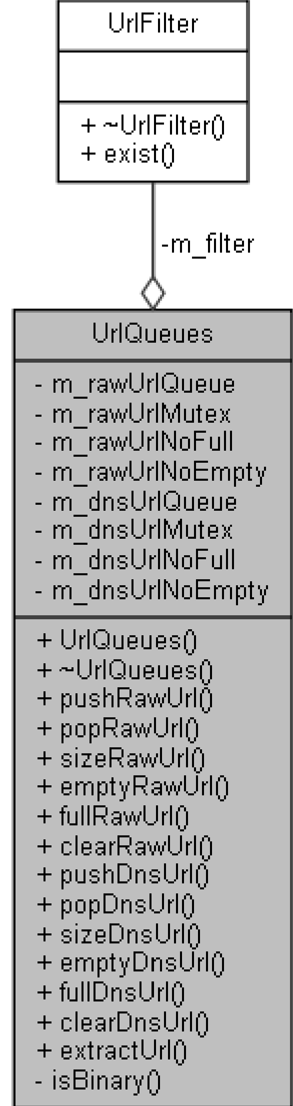
- G:/Projects/Tarena/WebCrawler/teacher/src/UrlFilter.h

UrlQueues类 参考

统一资源定位符队列

`#include <UrlQueues.h>`

UrlQueues 的协作图:



Public 成员函数

- UrlQueues (UrlFilter &filter)
  构造器
- ~UrlQueues (void)
  析构器
- void pushRawUrl (RawUrl const &rawUrl)
  压入原始统一资源定位符

- RawUrl popRawUrl (void)
  弹出原始统一资源定位符
- size_t sizeRawUrl (void) const
  获取原始统一资源定位符数
- bool emptyRawUrl (void) const
  原始统一资源定位符队列空否
- bool fullRawUrl (void) const
  原始统一资源定位符队列满否
- void clearRawUrl (void)
  清空原始统一资源定位符队列
- void pushDnsUrl (DnsUrl const &dnsUrl)
  压入解析统一资源定位符
- DnsUrl popDnsUrl (void)
  弹出解析统一资源定位符
- size_t sizeDnsUrl (void) const
  获取解析统一资源定位符数
- bool emptyDnsUrl (void) const
  解析统一资源定位符队列空否
- bool fullDnsUrl (void) const
  解析统一资源定位符队列满否
- void clearDnsUrl (void)
  清空解析统一资源定位符队列
- void extractUrl (char const *html, DnsUrl const &dnsUrl)
  从超文本标记语言页面内容中抽取统一资源定位符

静态 Private 成员函数

- static bool isBinary (string const &strUrl)
  判断某统一资源定位符所表示的资源是否是二进制资源

Private 属性

- UrlFilter & m_filter
  统一资源定位符过滤器
- list< RawUrl > m_rawUrlQueue
  原始统一资源定位符队列
- pthread_mutex_t m_rawUrlMutex
  原始统一资源定位符队列互斥锁
- pthread_cond_t m_rawUrlNoFull
  原始统一资源定位符队列非满条件变量
- pthread_cond_t m_rawUrlNoEmpty
  原始统一资源定位符队列非空条件变量
- list< DnsUrl > m_dnsUrlQueue
  解析统一资源定位符队列
- pthread_mutex_t m_dnsUrlMutex
  解析统一资源定位符队列互斥锁

- pthread_cond_t m_dnsUrlNoFull
  解析统一资源定位符队列非满条件变量
- pthread_cond_t m_dnsUrlNoEmpty
  解析统一资源定位符队列非空条件变量

详细描述

统一资源定位符队列

构造及析构函数说明

UrlQueues::UrlQueues (UrlFilter & *filter*)

构造器

参数:

in	*filter*	统一资源定位符过滤器

参考 m_dnsUrlMutex, m_dnsUrlNoEmpty, m_dnsUrlNoFull, m_rawUrlMutex, m_rawUrlNoEmpty , 以及 m_rawUrlNoFull.

```
14      : m_filter (filter) { // 初始化统一资源定位符过滤器
15    // 初始化原始统一资源定位符队列互斥锁
16    pthread_mutex_init (&m_rawUrlMutex, NULL);
17    // 初始化原始统一资源定位符队列非满条件变量
18    pthread_cond_init (&m_rawUrlNoFull, NULL);
19    // 初始化原始统一资源定位符队列非空条件变量
20    pthread_cond_init (&m_rawUrlNoEmpty, NULL);
21    // 初始化解析统一资源定位符队列互斥锁
22    pthread_mutex_init (&m_dnsUrlMutex, NULL);
23    // 初始化解析统一资源定位符队列非满条件变量
24    pthread_cond_init (&m_dnsUrlNoFull, NULL);
25    // 初始化解析统一资源定位符队列非空条件变量
26    pthread_cond_init (&m_dnsUrlNoEmpty, NULL);
27 }
```

UrlQueues::~UrlQueues (void)

析构器

参考 m_dnsUrlMutex, m_dnsUrlNoEmpty, m_dnsUrlNoFull, m_rawUrlMutex, m_rawUrlNoEmpty , 以及 m_rawUrlNoFull.

```
30                        {
31    // 销毁解析统一资源定位符队列非空条件变量
32    pthread_cond_destroy (&m_dnsUrlNoEmpty);
33    // 销毁解析统一资源定位符队列非满条件变量
34    pthread_cond_destroy (&m_dnsUrlNoFull);
35    // 销毁解析统一资源定位符队列互斥锁
36    pthread_mutex_destroy (&m_dnsUrlMutex);
37    // 销毁原始统一资源定位符队列非空条件变量
38    pthread_cond_destroy (&m_rawUrlNoEmpty);
39    // 销毁原始统一资源定位符队列非满条件变量
40    pthread_cond_destroy (&m_rawUrlNoFull);
```

```
41     // 销毁原始统一资源定位符队列互斥锁
42     pthread_mutex_destroy (&m_rawUrlMutex);
43 }
```

成员函数说明

void UrlQueues::clearDnsUrl (void)

清空解析统一资源定位符队列

参考 m_dnsUrlMutex, m_dnsUrlNoFull , 以及 m_dnsUrlQueue.

```
284                            {
285     // 加锁解析统一资源定位符队列互斥锁
286     pthread_mutex_lock (&m_dnsUrlMutex);
287
288     // 清空解析统一资源定位符队列
289     m_dnsUrlQueue.clear ();
290     // 唤醒等待解析统一资源定位符队列非满条件变量的线程
291     pthread_cond_signal (&m_dnsUrlNoFull);
292
293     // 解锁解析统一资源定位符队列互斥锁
294     pthread_mutex_unlock (&m_dnsUrlMutex);
295 }
```

void UrlQueues::clearRawUrl (void)

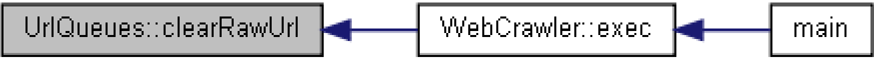
清空原始统一资源定位符队列

参考 m_rawUrlMutex, m_rawUrlNoFull , 以及 m_rawUrlQueue.

参考自 WebCrawler::exec().

```
162                            {
163     // 加锁原始统一资源定位符队列互斥锁
164     pthread_mutex_lock (&m_rawUrlMutex);
165
166     // 清空原始统一资源定位符队列
167     m_rawUrlQueue.clear ();
168     // 唤醒等待原始统一资源定位符队列非满条件变量的线程
169     pthread_cond_signal (&m_rawUrlNoFull);
170
171     // 解锁原始统一资源定位符队列互斥锁
172     pthread_mutex_unlock (&m_rawUrlMutex);
173 }
```

这是这个函数的调用关系图:



bool UrlQueues::emptyDnsUrl (void) const

解析统一资源定位符队列空否

返回值:

true	空

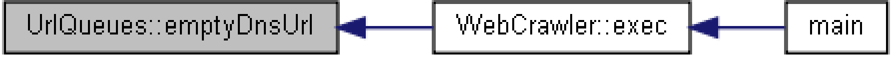
false	不空

参考 m_dnsUrlMutex , 以及 m_dnsUrlQueue.

参考自 WebCrawler::exec().

```
254                                    {
255     // 加锁解析统一资源定位符队列互斥锁
256     pthread_mutex_lock (&m_dnsUrlMutex);
257
258     // 获取解析统一资源定位符队列是否空
259     bool empty = m_dnsUrlQueue.empty ();
260
261     // 解锁解析统一资源定位符队列互斥锁
262     pthread_mutex_unlock (&m_dnsUrlMutex);
263     // 返回解析统一资源定位符队列是否空
264     return empty;
265 }
```

这是这个函数的调用关系图:



bool UrlQueues::emptyRawUrl (void) const

原始统一资源定位符队列空否

返回值:

true	空
false	不空

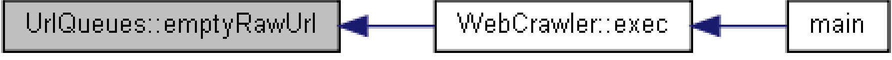
原始统一资源定位符队列空否 空返回true，不空返回false

参考 m_rawUrlMutex , 以及 m_rawUrlQueue.

参考自 WebCrawler::exec().

```
132                                    {
133     // 加锁原始统一资源定位符队列互斥锁
134     pthread_mutex_lock (&m_rawUrlMutex);
135
136     // 获取原始统一资源定位符队列是否空
137     bool empty = m_rawUrlQueue.empty ();
138
139     // 解锁原始统一资源定位符队列互斥锁
140     pthread_mutex_unlock (&m_rawUrlMutex);
141     // 返回原始统一资源定位符队列是否空
142     return empty;
143 }
```

这是这个函数的调用关系图:



void UrlQueues::extractUrl (char const * *html*, DnsUrl const & *dnsUrl*)

从超文本标记语言页面内容中抽取统一资源定位符

参数:

in	*html*	超文本标记语言页面内容字符串
in	*dnsUrl*	被抽取页面解析统一资源定位符

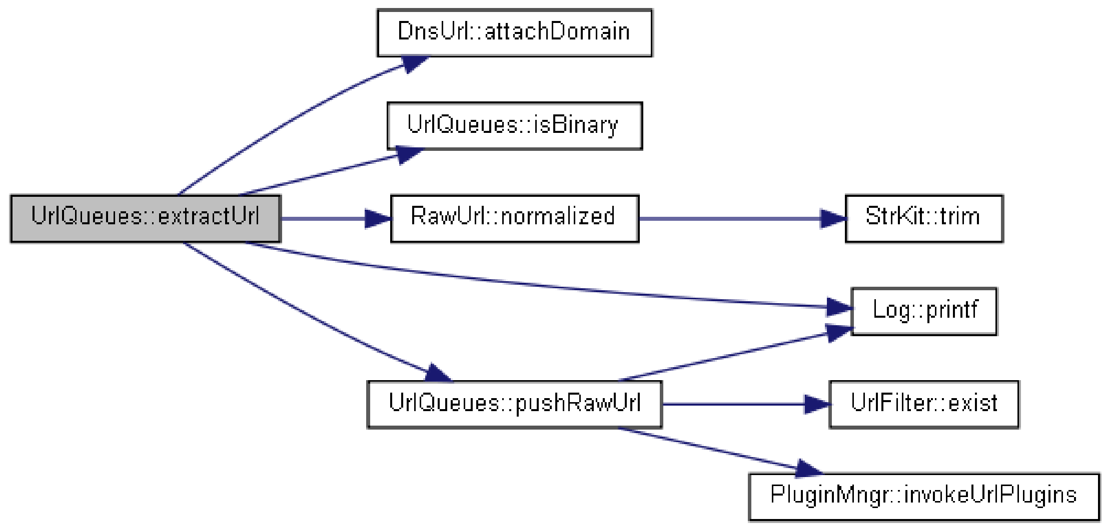
参考 DnsUrl::attachDomain(), RawUrl::ETYPE_HTML, g_app, isBinary(), Log::LEVEL_DBG, Log::LEVEL_ERR, Log::LEVEL_WAR, RawUrl::m_depth, WebCrawler::m_log, RawUrl::normalized(), Log::printf() , 以及 pushRawUrl().

参考自 Socket::recvResponse().

```
301      {
302     // 正则表达式
303     regex_t ex;
304
305     // 编译正则表达式: href="\s*\([^ >"]*\)\s*"
306     //     \s - 匹配任意空白字符(空格、制表、换页等)
307     //      * - 重复前一个匹配项任意次
308     //     \( - 子表达式左边界
309     //     \) - 子表达式右边界
310     // [^ >"] - 匹配任意不是空格、大于号和双引号的字符
311     int error = regcomp (&ex, "href=\"\\s*\\([^ >\"]*\\)\\s*\"", 0);
312     // 若失败
313     if (error) {
314         // 错误信息缓冲区
315         char errInfo[1024];
316         // 获取正则表达式编译错误信息
317         regerror (error, &ex, errInfo, sizeof (errInfo) /
318             sizeof (errInfo[0]));
319         // 记录一般错误日志
320         g_app->m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
321             "regcomp: %s", errInfo);
322     }
323
324     // 匹配集合
325     regmatch_t match[2];
326
327     // 在超文本标记语言页面内容字符串中,
328     // 查找所有与正则表达式匹配的内容
329     while (regexec (&ex, html, sizeof (match) /
330         sizeof (match[0]), match, 0) != REG_NOMATCH) {
331         // regex : href="\s*\([^ >"]*\)\s*"
332         // html  : ...href="  /software/download.html  "...
333         //            |       |<-----match[1]------>|  |
334         //            |     rm_so                rm_eo|
335         //            |<-------------match[0]--------->|
336         //          rm_so                           rm_eo
337
338         // 匹配子表达式的内容首地址
339         html += match[1].rm_so;
340         // 匹配子表达式的内容字符数
341         size_t len = match[1].rm_eo - match[1].rm_so;
342         // 匹配子表达式的内容字符串,即超链接中的统一资源定位符
343         string strUrl (html, len);
344         // 移至匹配主表达式的内容后,以备在下一轮循环中继续查找
345         html += len + match[0].rm_eo - match[1].rm_eo;
346
347         // 若是二进制资源
348         if (isBinary (strUrl))
349             // 继续下一轮循环
350             continue;
351
352         // 若添加域名失败
353         if (! dnsUrl.attachDomain (strUrl))
354             // 继续下一轮循环
355             continue;
356
```

```
357         // 记录调试日志
358         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
359             "抽取到一个深度为%d的统一资源定位符\"%s\"",
360             dnsUrl.m_depth + 1, strUrl.c_str ());
361
362         // 若规格化失败
363         if (! RawUrl::normalized (strUrl)) {
364             // 记录警告日志
365             g_app->m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
366                 "规格化统一资源定位符\"%s\"失败", strUrl.c_str ());
367             // 继续下一轮循环
368             continue;
369         }
370
371         // 压入原始统一资源定位符队列
372         pushRawUrl (RawUrl (strUrl, RawUrl::ETYPE_HTML,
373             dnsUrl.m_depth + 1));
374     }
375
376     // 释放正则表达式
377     regfree (&ex);
378 }
```

函数调用图:



这是这个函数的调用关系图:

bool UrlQueues::fullDnsUrl (void) const

解析统一资源定位符队列满否

返回值:

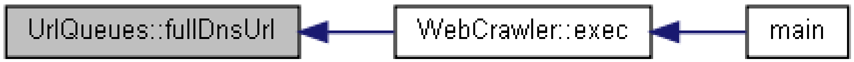
true	满
false	不满

参考 g_app, WebCrawler::m_cfg, m_dnsUrlMutex, m_dnsUrlQueue, 以及 Configurator::m_maxDnsUrls.

参考自 WebCrawler::exec().

```
269                                     {
270     // 加锁解析统一资源定位符队列互斥锁
271     pthread_mutex_lock (&m_dnsUrlMutex);
272
273     // 获取解析统一资源定位符队列是否满
274     bool full = 0 <= g_app->m_cfg.m_maxDnsUrls &&
275         (size_t)g_app->m_cfg.m_maxDnsUrls <= m_dnsUrlQueue.size ();
276
277     // 解锁解析统一资源定位符队列互斥锁
278     pthread_mutex_unlock (&m_dnsUrlMutex);
279     // 返回解析统一资源定位符队列是否满
280     return full;
281 }
```

这是这个函数的调用关系图:



bool UrlQueues::fullRawUrl (void) const

原始统一资源定位符队列满否

返回值:

true	满
false	不满

参考 g_app, WebCrawler::m_cfg, Configurator::m_maxRawUrls, m_rawUrlMutex , 以及 m_rawUrlQueue.

参考自 WebCrawler::exec().

```
147                                     {
148     // 加锁原始统一资源定位符队列互斥锁
149     pthread_mutex_lock (&m_rawUrlMutex);
150
151     // 获取原始统一资源定位符队列是否满
152     bool full = 0 <= g_app->m_cfg.m_maxRawUrls &&
153         (size_t)g_app->m_cfg.m_maxRawUrls <= m_rawUrlQueue.size ();
154
155     // 解锁原始统一资源定位符队列互斥锁
156     pthread_mutex_unlock (&m_rawUrlMutex);
157     // 返回原始统一资源定位符队列是否满
158     return full;
159 }
```

这是这个函数的调用关系图:

bool UrlQueues::isBinary (string const & *strUrl*) [static], [private]

判断某统一资源定位符所表示的资源是否是二进制资源

返回值:

true	是二进制资源
false	非二进制资源

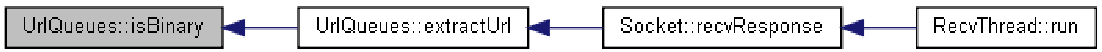
参数:

in	*strUrl*	统一资源定位符字符串

参考自 extractUrl().

```
384     {
385     // 统一资源定位符字符串中'.'字符的下标
386     string::size_type pos;
387     // 若统一资源定位符字符串中最后一个'.'字符，连同其后的子串
388     // 构成二进制资源的文件扩展名，则返回true，否则返回false
389     return (pos = strUrl.find_last_of ('.')) != string::npos &&
390         string (".jpg.jpeg.gif.png.ico.bmp.swf").find (
391             strUrl.substr (pos)) != string::npos;
392 }
```

这是这个函数的调用关系图:



DnsUrl UrlQueues::popDnsUrl (void)

弹出解析统一资源定位符

返回:

解析统一资源定位符

参考 g_app, Log::LEVEL_DBG, WebCrawler::m_cfg, m_dnsUrlMutex, m_dnsUrlNoEmpty, m_dnsUrlNoFull, m_dnsUrlQueue, DnsUrl::m_ip, WebCrawler::m_log, Configurator::m_maxDnsUrls, DnsUrl::m_path, DnsUrl::m_port , 以及 Log::printf().

参考自 WebCrawler::startJob().

```
207                         {
208     // 加锁解析统一资源定位符队列互斥锁
209     pthread_mutex_lock (&m_dnsUrlMutex);
210 
211     // 若解析统一资源定位符队列空
212     while (m_dnsUrlQueue.empty ())
213         // 等待解析统一资源定位符队列非空条件变量
214         pthread_cond_wait (&m_dnsUrlNoEmpty, &m_dnsUrlMutex);
215 
216     // 获取解析统一资源定位符队列首元素
217     DnsUrl dnsUrl = m_dnsUrlQueue.front ();
218     // 从解析统一资源定位符队列弹出解析统一资源定位符
219     m_dnsUrlQueue.pop_front ();
220 
221     // 记录调试日志
222     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
223         "解析统一资源定位符\"ip=%s, port=%d, path=%s\"出队",
224         dnsUrl.m_ip.c_str (), dnsUrl.m_port, dnsUrl.m_path.c_str ());
225 
226     // 若解析统一资源定位符队列由满变为非满
227     if (m_dnsUrlQueue.size () == (size_t)g_app->m_cfg.m_maxDnsUrls - 1)
228         // 唤醒等待解析统一资源定位符队列非满条件变量的线程
229         pthread_cond_signal (&m_dnsUrlNoFull);
230 
```

```
231      // 解锁解析统一资源定位符队列互斥锁
232      pthread_mutex_unlock (&m_dnsUrlMutex);
233      // 返回解析统一资源定位符
234      return dnsUrl;
235 }
```

函数调用图:

这是这个函数的调用关系图:

RawUrl UrlQueues::popRawUrl (void)

弹出原始统一资源定位符

返回:

原始统一资源定位符

参考 g_app, Log::LEVEL_DBG, WebCrawler::m_cfg, WebCrawler::m_log, Configurator::m_maxRawUrls, m_rawUrlMutex, m_rawUrlNoEmpty, m_rawUrlNoFull, m_rawUrlQueue, RawUrl::m_strUrl , 以及 Log::printf().

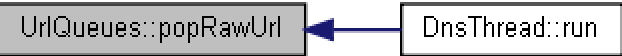
参考自 DnsThread::run().

```
 86                                    {
 87      // 加锁原始统一资源定位符队列互斥锁
 88      pthread_mutex_lock (&m_rawUrlMutex);
 89
 90      // 若原始统一资源定位符队列空
 91      while (m_rawUrlQueue.empty ())
 92          // 等待原始统一资源定位符队列非空条件变量
 93          pthread_cond_wait (&m_rawUrlNoEmpty, &m_rawUrlMutex);
 94
 95      // 获取原始统一资源定位符队列首元素
 96      RawUrl rawUrl = m_rawUrlQueue.front ();
 97      // 从原始统一资源定位符队列弹出原始统一资源定位符
 98      m_rawUrlQueue.pop_front ();
 99
100      // 记录调试日志
101      g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
102          "原始统一资源定位符\"%s\"出队", rawUrl.m_strUrl.c_str ());
103
104      // 若原始统一资源定位符队列由满变为非满
105      if (m_rawUrlQueue.size () == (size_t)g_app->m_cfg.m_maxRawUrls - 1)
106          // 唤醒等待原始统一资源定位符队列非满条件变量的线程
107          pthread_cond_signal (&m_rawUrlNoFull);
108
109      // 解锁原始统一资源定位符队列互斥锁
110      pthread_mutex_unlock (&m_rawUrlMutex);
111      // 返回原始统一资源定位符
112      return rawUrl;
113 }
```

函数调用图:

UrlQueues::popRawUrl → Log::printf

这是这个函数的调用关系图:



void UrlQueues::pushDnsUrl (DnsUrl const & *dnsUrl*)

压入解析统一资源定位符

参数:

in	*dnsUrl*	解析统一资源定位符

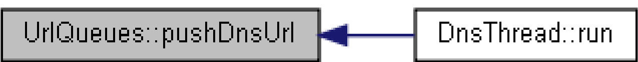
参考 g_app, Log::LEVEL_DBG, WebCrawler::m_cfg, m_dnsUrlMutex, m_dnsUrlNoEmpty, m_dnsUrlNoFull, m_dnsUrlQueue, DnsUrl::m_ip, WebCrawler::m_log, Configurator::m_maxDnsUrls, DnsUrl::m_path, DnsUrl::m_port , 以及 Log::printf().

参考自 DnsThread::run().

```
178     {
179     // 加锁解析统一资源定位符队列互斥锁
180     pthread_mutex_lock (&m_dnsUrlMutex);
181 
182     // 若配置器中的解析统一资源定位符队列最大容量有效且到限
183     while (0 <= g_app->m_cfg.m_maxDnsUrls &&
184         (size_t)g_app->m_cfg.m_maxDnsUrls <= m_dnsUrlQueue.size ())
185         // 等待解析统一资源定位符队列非满条件变量
186         pthread_cond_wait (&m_dnsUrlNoFull, &m_dnsUrlMutex);
187 
188     // 向解析统一资源定位符队列压入解析统一资源定位符
189     m_dnsUrlQueue.push_back (dnsUrl);
190 
191     // 记录调试日志
192     g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
193         "解析统一资源定位符\"ip=%s, port=%d, path=%s\"入队",
194         dnsUrl.m_ip.c_str (), dnsUrl.m_port, dnsUrl.m_path.c_str ());
195 
196     // 若解析统一资源定位符队列由空变为非空
197     if (m_dnsUrlQueue.size () == 1)
198         // 唤醒等待解析统一资源定位符队列非空条件变量的线程
199         pthread_cond_signal (&m_dnsUrlNoEmpty);
200 
201     // 解锁解析统一资源定位符队列互斥锁
202     pthread_mutex_unlock (&m_dnsUrlMutex);
203 }
```

函数调用图:

这是这个函数的调用关系图:



void UrlQueues::pushRawUrl (RawUrl const & *rawUrl*)

压入原始统一资源定位符

参数:

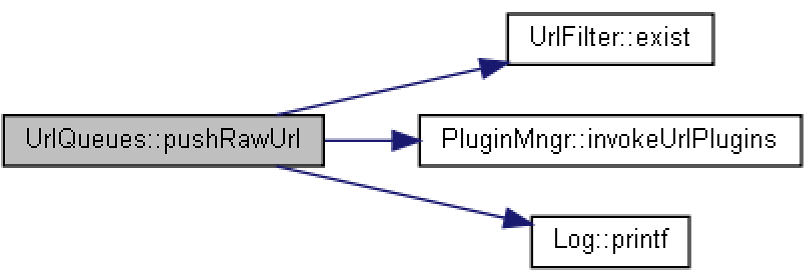
in	*rawUrl*	原始统一资源定位符

参考 UrlFilter::exist(), g_app, PluginMngr::invokeUrlPlugins(), Log::LEVEL_DBG, WebCrawler::m_cfg, m_filter, WebCrawler::m_log, Configurator::m_maxRawUrls, WebCrawler::m_pluginMngr, m_rawUrlMutex, m_rawUrlNoEmpty, m_rawUrlNoFull, m_rawUrlQueue, RawUrl::m_strUrl , 以及 Log::printf().

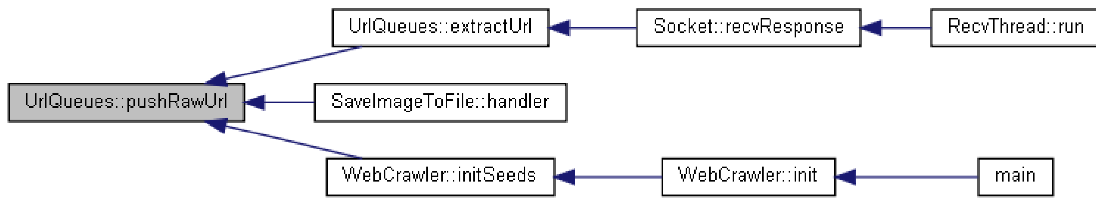
参考自 extractUrl(), SaveImageToFile::handler() , 以及 WebCrawler::initSeeds().

```
48   {
49     // 加锁原始统一资源定位符队列互斥锁
50     pthread_mutex_lock (&m_rawUrlMutex);
51
52     // 若已被处理过
53     if (m_filter.exist (rawUrl.m_strUrl))
54         // 记录调试日志
55         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
56             "不再处理已处理过的统一资源定位符\"%s\"",
57             rawUrl.m_strUrl.c_str ());
58     // 否则，调用统一资源定位符插件处理函数，若成功
59     else if (g_app->m_pluginMngr.invokeUrlPlugins (
60         const_cast<RawUrl*> (&rawUrl))) {
61         // 若配置器中的原始统一资源定位符队列最大容量有效且到限
62         while (0 <= g_app->m_cfg.m_maxRawUrls &&
63             (size_t)g_app->m_cfg.m_maxRawUrls <= m_rawUrlQueue.size ())
64             // 等待原始统一资源定位符队列非满条件变量
65             pthread_cond_wait (&m_rawUrlNoFull, &m_rawUrlMutex);
66
67         // 向原始统一资源定位符队列压入原始统一资源定位符
68         m_rawUrlQueue.push_back (rawUrl);
69
70         // 记录调试日志
71         g_app->m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
72             "原始统一资源定位符\"%s\"入队", rawUrl.m_strUrl.c_str ());
73
74         // 若原始统一资源定位符队列由空变为非空
75         if (m_rawUrlQueue.size () == 1)
76             // 唤醒等待原始统一资源定位符队列非空条件变量的线程
77             pthread_cond_signal (&m_rawUrlNoEmpty);
78     }
79
80     // 解锁原始统一资源定位符队列互斥锁
81     pthread_mutex_unlock (&m_rawUrlMutex);
82 }
```

函数调用图:



这是这个函数的调用关系图:



size_t UrlQueues::sizeDnsUrl (void) const

获取解析统一资源定位符数

返回:

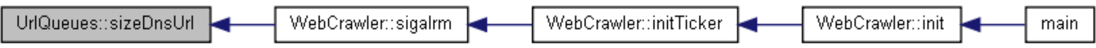
解析统一资源定位符数

参考 m_dnsUrlMutex , 以及 m_dnsUrlQueue.

参考自 WebCrawler::sigalrm().

```
239                                     {
240     // 加锁解析统一资源定位符队列互斥锁
241     pthread_mutex_lock (&m_dnsUrlMutex);
242
243     // 获取解析统一资源定位符队列大小
244     size_t size = m_dnsUrlQueue.size ();
245
246     // 解锁解析统一资源定位符队列互斥锁
247     pthread_mutex_unlock (&m_dnsUrlMutex);
248     // 返回解析统一资源定位符数
249     return size;
250 }
```

这是这个函数的调用关系图:



size_t UrlQueues::sizeRawUrl (void) const

获取原始统一资源定位符数

返回:

原始统一资源定位符数

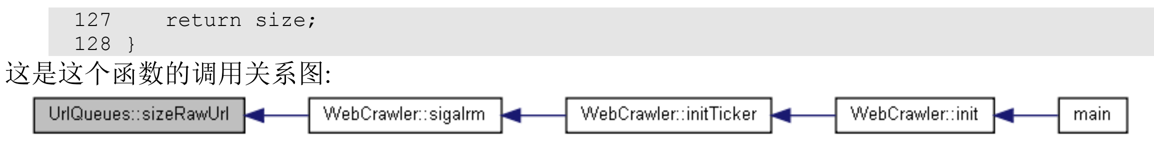
参考 m_rawUrlMutex , 以及 m_rawUrlQueue.

参考自 WebCrawler::sigalrm().

```
117                                     {
118     // 加锁原始统一资源定位符队列互斥锁
119     pthread_mutex_lock (&m_rawUrlMutex);
120
121     // 获取原始统一资源定位符队列大小
122     size_t size = m_rawUrlQueue.size ();
123
124     // 解锁原始统一资源定位符队列互斥锁
125     pthread_mutex_unlock (&m_rawUrlMutex);
126     // 返回原始统一资源定位符数
```

```
127     return size;
128 }
```

这是这个函数的调用关系图:



类成员变量说明

pthread_mutex_t UrlQueues::m_dnsUrlMutex [mutable], [private]

解析统一资源定位符队列互斥锁

参考自 clearDnsUrl(), emptyDnsUrl(), fullDnsUrl(), popDnsUrl(), pushDnsUrl(), sizeDnsUrl(), UrlQueues() , 以及 ~UrlQueues().

pthread_cond_t UrlQueues::m_dnsUrlNoEmpty [private]

解析统一资源定位符队列非空条件变量

参考自 popDnsUrl(), pushDnsUrl(), UrlQueues() , 以及 ~UrlQueues().

pthread_cond_t UrlQueues::m_dnsUrlNoFull [private]

解析统一资源定位符队列非满条件变量

参考自 clearDnsUrl(), popDnsUrl(), pushDnsUrl(), UrlQueues() , 以及 ~UrlQueues().

list<DnsUrl> UrlQueues::m_dnsUrlQueue [private]

解析统一资源定位符队列

参考自 clearDnsUrl(), emptyDnsUrl(), fullDnsUrl(), popDnsUrl(), pushDnsUrl() , 以及 sizeDnsUrl().

UrlFilter& UrlQueues::m_filter [private]

统一资源定位符过滤器

参考自 pushRawUrl().

pthread_mutex_t UrlQueues::m_rawUrlMutex [mutable], [private]

原始统一资源定位符队列互斥锁

参考自 clearRawUrl(), emptyRawUrl(), fullRawUrl(), popRawUrl(), pushRawUrl(), sizeRawUrl(), UrlQueues() , 以及 ~UrlQueues().

pthread_cond_t UrlQueues::m_rawUrlNoEmpty `[private]`

原始统一资源定位符队列非空条件变量

参考自 popRawUrl(), pushRawUrl(), UrlQueues() , 以及 ~UrlQueues().

pthread_cond_t UrlQueues::m_rawUrlNoFull `[private]`

原始统一资源定位符队列非满条件变量

参考自 clearRawUrl(), popRawUrl(), pushRawUrl(), UrlQueues() , 以及 ~UrlQueues().

list<RawUrl> UrlQueues::m_rawUrlQueue `[private]`

原始统一资源定位符队列

参考自 clearRawUrl(), emptyRawUrl(), fullRawUrl(), popRawUrl(), pushRawUrl() , 以及 sizeRawUrl().

该类的文档由以下文件生成:

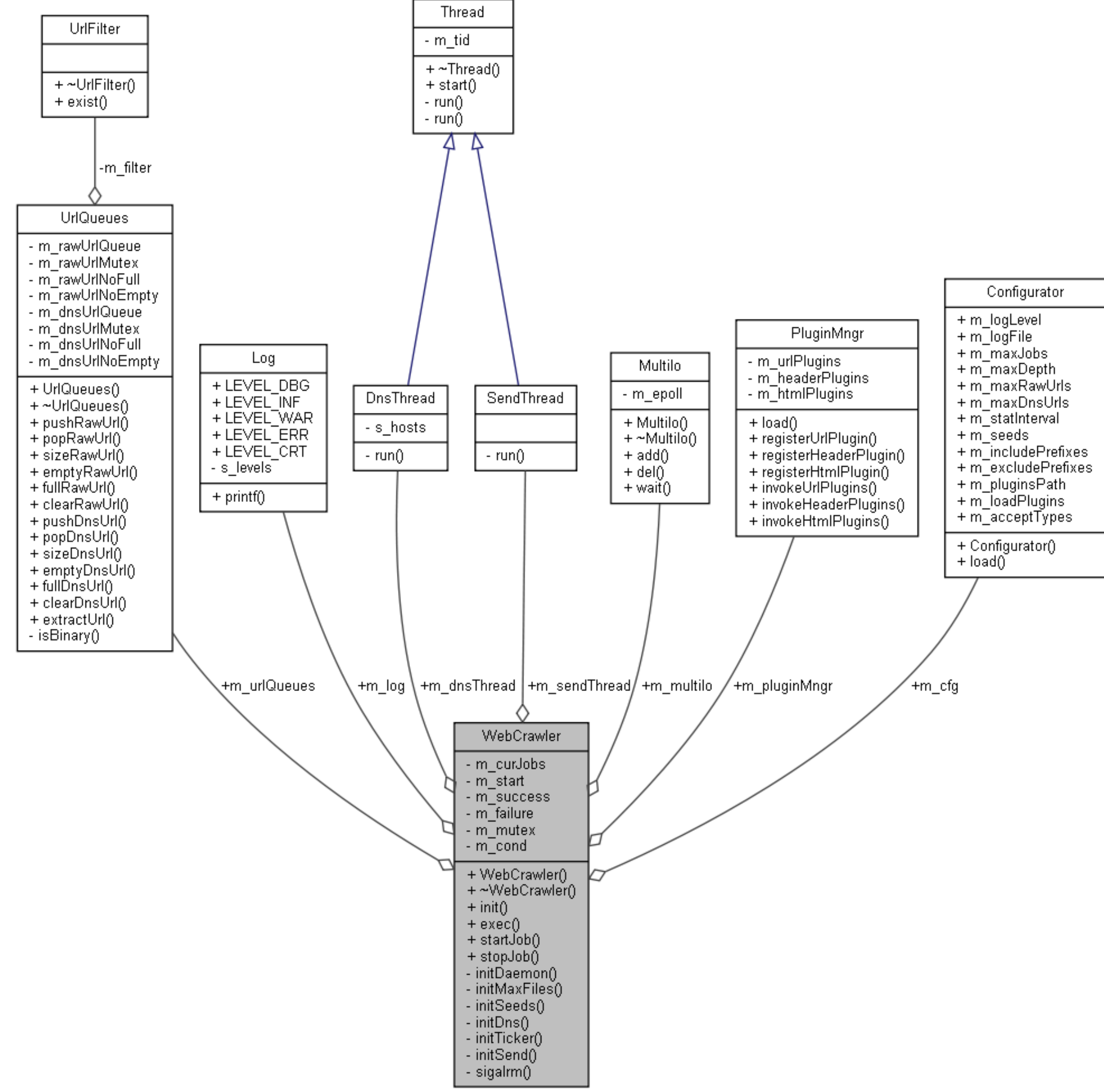
- G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.h
- G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.cpp

WebCrawler类 参考

网络爬虫

`#include <WebCrawler.h>`

WebCrawler 的协作图:



Public 成员函数

- WebCrawler (UrlFilter &filter)
  构造器
- ~WebCrawler (void)
  析构器
- void init (bool daemon=false)
  初始化
- void exec (void)

执行多路输入输出循环

- void startJob (void)
  启动一个抓取任务
- void stopJob (bool success=true)
  停止一个抓取任务

Public 属性

- Log m_log
  日志
- Configurator m_cfg
  配置器
- MultiIo m_multiIo
  多路输入输出
- PluginMngr m_pluginMngr
  插件管理器
- UrlQueues m_urlQueues
  统一资源定位符队列
- DnsThread m_dnsThread
  域名解析线程
- SendThread m_sendThread
  发送线程

Private 成员函数

- void initDaemon (void) const
  使调用进程成为精灵进程
- bool initMaxFiles (rlim_t maxFiles) const
  初始化最大文件描述符数
- void initSeeds (void)
  将种子链接压入原始统一资源定位符队列
- void initDns (void)
  启动域名解析线程
- void initTicker (void) const
  启动状态定时器
- void initSend (void)
  启动发送线程

静态 Private 成员函数

- static void sigalrm (int signum)
  SIGALRM(14)信号处理

Private 属性

- int m_curJobs
  当前抓取任务数
- time_t m_start

启动时间

- unsigned int m_success
  成功次数
- unsigned int m_failure
  失败次数
- pthread_mutex_t m_mutex
  互斥锁
- pthread_cond_t m_cond
  条件变量

详细描述

网络爬虫

构造及析构函数说明

WebCrawler::WebCrawler (UrlFilter & *filter*)

构造器

参数:

in	*filter*	统一资源定位符过滤器

参考 m_cond , 以及 m_mutex.

```
16      : m_urlQueues  (filter),      // 初始化统一资源定位符队列
17        m_curJobs    (0),          // 初始化当前抓取任务数为0
18        m_start     (time (NULL)), // 初始化启动时间为当前系统时间
19        m_success    (0),          // 初始化成功次数为0
20        m_failure    (0) {         // 初始化失败次数为0
21     // 初始化互斥锁
22     pthread_mutex_init (&m_mutex, NULL);
23     // 初始化条件变量
24     pthread_cond_init (&m_cond, NULL);
25 }
```

WebCrawler::~WebCrawler (void)

析构器

参考 m_cond , 以及 m_mutex.

```
28                          {
29     // 销毁条件变量
30     pthread_cond_destroy (&m_cond);
31     // 销毁互斥锁
32     pthread_mutex_destroy (&m_mutex);
33 }
```

成员函数说明

void WebCrawler::exec (void)

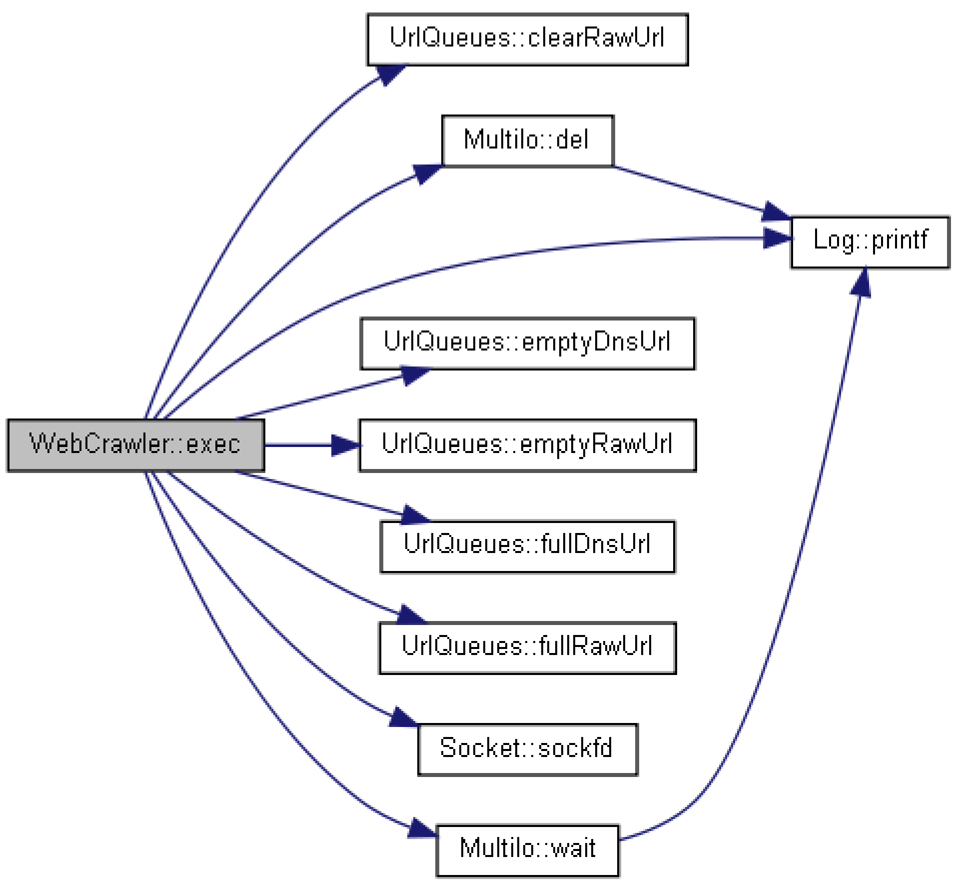
执行多路输入输出循环

参考 UrlQueues::clearRawUrl(), MultiIo::del(), UrlQueues::emptyDnsUrl(), UrlQueues::emptyRawUrl(), UrlQueues::fullDnsUrl(), UrlQueues::fullRawUrl(), Log::LEVEL_DBG, Log::LEVEL_WAR, m_cfg, m_curJobs, m_log, Configurator::m_maxJobs, m_multiIo, m_urlQueues, Log::printf(), Socket::sockfd() , 以及 MultiIo::wait().

参考自 main().

```
69                      {
70     // 无限循环
71     for (;;) {
72         // 输入输出事件结构数组
73         epoll_event events[10];
74         // 等待所关注输入输出事件的发生，两秒超时
75         int fds = m_multiIo.wait (events, sizeof (events) /
76             sizeof (events[0]), 2000);
77
78         // 若超时或被信号中断
79         if (fds <= 0) {
80             // 若没有抓取任务且原始统一资源定位符
81             // 队列和解析统一资源定位符队列都为空
82             if (! m_curJobs &&
83                 m_urlQueues.emptyRawUrl () &&
84                 m_urlQueues.emptyDnsUrl ()) {
85
86                 // 等一秒
87                 sleep (1);
88
89                 // 若没有抓取任务且原始统一资源定位符
90                 // 队列和解析统一资源定位符队列都为空
91                 if (! m_curJobs &&
92                     m_urlQueues.emptyRawUrl () &&
93                     m_urlQueues.emptyDnsUrl ())
94                     // 抓取任务完成，跳出循环
95                     break;
96             }
97             // 否则
98             else
99             // 若抓取任务到限且原始统一资源定位符
100            // 队列和解析统一资源定位符队列都为满
101            if (m_curJobs == m_cfg.m_maxJobs &&
102                m_urlQueues.fullRawUrl () &&
103                m_urlQueues.fullDnsUrl ()) {
104
105                // 等一秒
106                sleep (1);
107
108                // 若抓取任务到限且原始统一资源定位符
109                // 队列和解析统一资源定位符队列都为满
110                if (m_curJobs == m_cfg.m_maxJobs &&
111                    m_urlQueues.fullRawUrl () &&
112                    m_urlQueues.fullDnsUrl ())
113                    // 清空原始统一资源定位符队列，避免死锁
114                    m_urlQueues.clearRawUrl ();
115            }
116        }
117
118        // 依次处理每个处于就绪状态的文件描述符
119        for (int i = 0; i < fds; ++i) {
120            // 从事件结构中取出套接字对象指针
```

```
121         Socket* socket = (Socket*)events[i].data.ptr;
122
123         // 若为异常事件
124         if (events[i].events & EPOLLERR ||
125             events[i].events & EPOLLHUP ||
126             ! (events[i].events & EPOLLIN)) {
127             // 记录警告日志
128             m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
129                 "套接字异常");
130             // 删除需要被关注的输入输出事件
131             m_multiIo.del (socket->sockfd (), events[i]);
132             // 销毁套接字对象
133             delete socket;
134             // 继续下一轮循环
135             continue;
136         }
137
138         // 删除需要被关注的输入输出事件
139         m_multiIo.del (socket->sockfd (), events[i]);
140
141         // 记录调试日志
142         m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__,
143             "套接字%d上有数据可读，创建HTTP线程接收数据",
144             socket->sockfd ());
145
146         // 创建接收线程对象并启动接收线程
147         (new RecvThread (socket))->start ();
148     }
149 }
150
151 // 记录调试日志
152 m_log.printf (Log::LEVEL_DBG, __FILE__, __LINE__, "任务完成");
153 }
```
函数调用图:



这是这个函数的调用关系图:

void WebCrawler::init (bool *daemon* = false)

初始化

参数:

in	*daemon*	是否以精灵进程方式运行

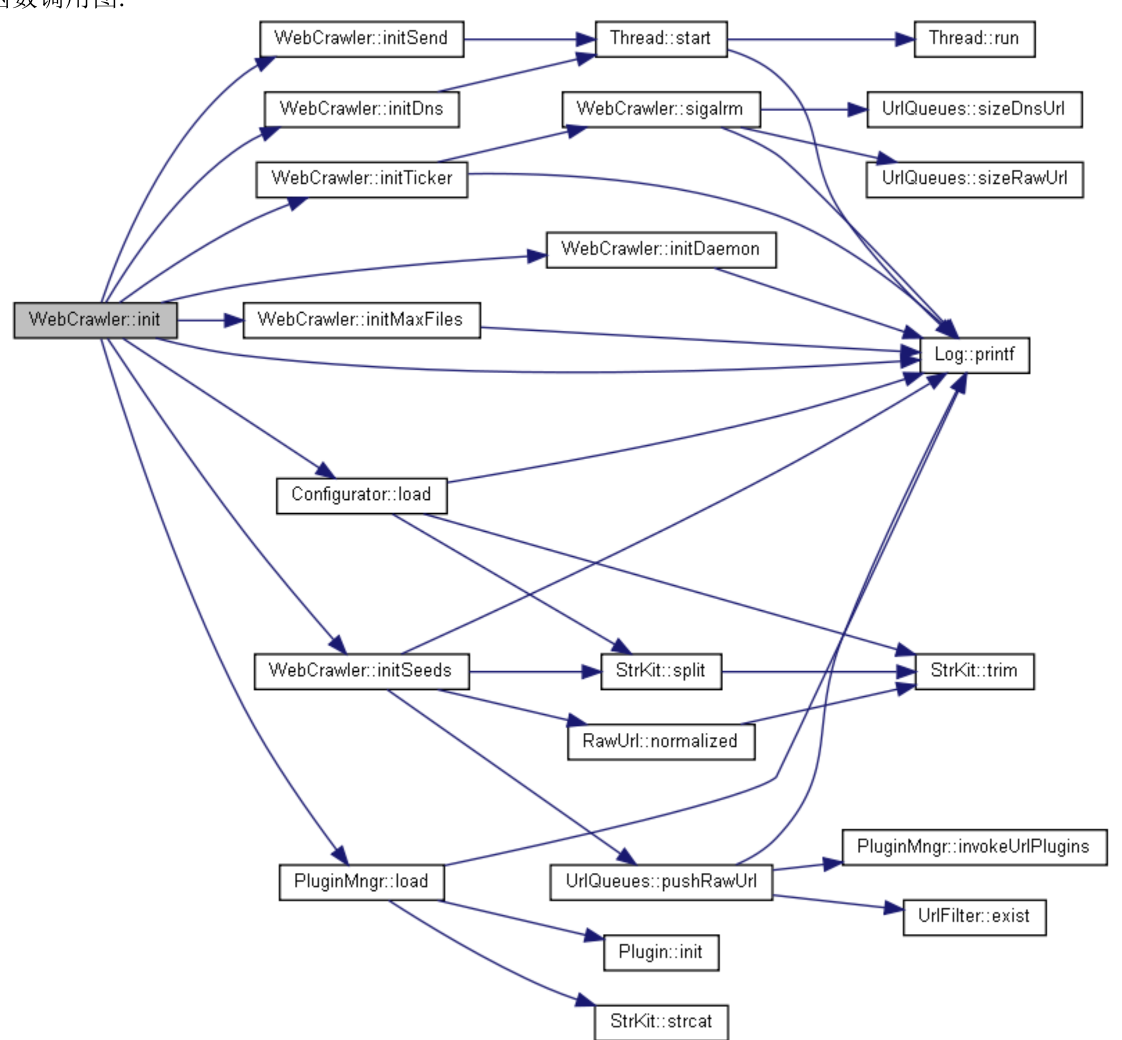
参考 initDaemon(), initDns(), initMaxFiles(), initSeeds(), initSend(), initTicker(), Log::LEVEL_ERR, PluginMngr::load(), Configurator::load(), m_cfg, m_log, m_pluginMngr , 以及 Log::printf().

参考自 main().

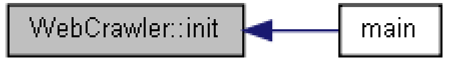
```
38      {
39      // 通过配置器从指定的配置文件中加载配置信息
40      m_cfg.load ("WebCrawler.cfg");
41
42      // 若以精灵进程方式运行
43      if (daemon)
44          // 使调用进程成为精灵进程
45          initDaemon ();
```

```
46
47     // 通过插件管理器加载插件
48     m_pluginMngr.load ();
49
50     // 若切换工作目录至下载目录失败
51     if (chdir ("../download") == -1)
52         // 记录一般错误日志
53         m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
54             "chdir: %s", strerror (errno));
55
56     // 初始化最大文件描述符数为1K
57     initMaxFiles (1024);
58     // 将种子链接压入原始统一资源定位符队列
59     initSeeds ();
60     // 启动域名解析线程
61     initDns ();
62     // 启动状态定时器
63     initTicker ();
64     // 启动发送线程
65     initSend ();
66 }
```

函数调用图:



这是这个函数的调用关系图:



void WebCrawler::initDaemon (void) const [private]

使调用进程成为精灵进程

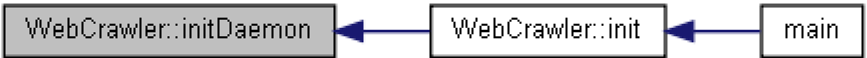
参考 Log::LEVEL_ERR, m_cfg, m_log, Configurator::m_logFile , 以及 Log::printf().

参考自 init().

```
231                               {
232     // 创建子进程
233     pid_t pid = fork ();
234
235     // 若失败
236     if (pid == -1)
237         // 记录一般错误日志
238         m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
239             "fork: %s", strerror (errno));
240
241     // 若为父进程
242     if (pid)
243         // 退出，使子进程成为孤儿进程并被init进程收养
244         exit (EXIT_SUCCESS);
245
246     // 子进程创建新会话并成为新会话中唯一进程组的组长
247     // 进程，进而与原会话、原进程组和控制终端脱离关系
248     setsid ();
249
250     // 打开空设备文件
251     int fd = open("/dev/null", O_RDWR, 0);
252     // 若成功
253     if (fd != -1) {
254         // 复制空设备文件描述符到标准输入
255         dup2 (fd, STDIN_FILENO);
256         // 复制空设备文件描述符到标准输出
257         dup2 (fd, STDOUT_FILENO);
258         // 复制空设备文件描述符到标准出错
259         dup2 (fd, STDERR_FILENO);
260         // 若空设备文件描述符大于标准出错
261         if (fd > STDERR_FILENO)
262             // 关闭空设备文件描述符
263             close (fd);
264     }
265
266     // 若配置器中的日志文件路径非空且打开(创建)日志文件成功
267     if (! m_cfg.m_logFile.empty () &&
268         (fd = open (m_cfg.m_logFile.c_str (),
269         O_WRONLY | O_APPEND | O_CREAT, 0664)) != -1) {
270         // 复制日志文件描述符到标准输出
271         dup2 (fd, STDOUT_FILENO);
272         // 若日志文件描述符大于标准出错
273         if (fd > STDERR_FILENO)
274             // 关闭日志文件描述符
275             close (fd);
276     }
277 }
```

函数调用图:

这是这个函数的调用关系图:



void WebCrawler::initDns (void) [private]

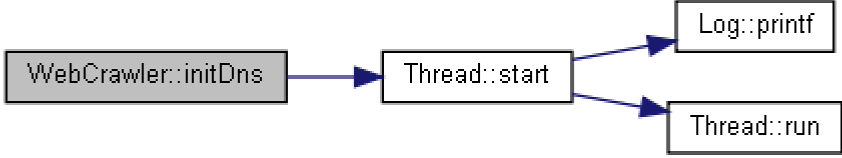
启动域名解析线程

参考 m_dnsThread , 以及 Thread::start().

参考自 init().

```
340                             {
341     // 通过域名解析线程对象启动域名解析线程
342     m_dnsThread.start ();
343     /*
344     // 若解析统一资源定位符队列为空，延时再查，最多查八次
345     for (int try = 1; m_urlQueues.emptyDnsUrl () && try < 8;
346         usleep (10000 << try++));
347 
348     // 若解析统一资源定位符队列为空
349     if (m_urlQueues.emptyDnsUrl ())
350         // 记录一般错误日志
351         m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
352             "解析URL队列空");
353     */
354 }
```

函数调用图:



这是这个函数的调用关系图:

bool WebCrawler::initMaxFiles (rlim_t *maxFiles*) const [private]

初始化最大文件描述符数

返回值:

true	成功
false	失败

参数:

in	*maxFiles*	最大文件描述符数

参考 Log::LEVEL_WAR, m_log , 以及 Log::printf().

参考自 init().

```
283         {
284     // 资源限制结构
285     rlimit rl;
286
287     // 若获取当前进程可同时打开的最大文件数失败
288     if (getrlimit (RLIMIT_NOFILE, &rl) == -1) {
289         // 记录警告日志
290         m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
291             "getrlimit: %s", strerror (errno));
292         // 返回失败
293         return false;
294     }
295
296     // 若最大文件描述符数超过系统允许的极限
297     if (maxFiles > rl.rlim_max) {
298         // 记录警告日志
299         m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
300             "文件描述符上限不能超过%lu", rl.rlim_max);
301         // 返回失败
302         return false;
303     }
304
305     // 若设置当前进程可同时打开的最大文件数失败
306     rl.rlim_cur = maxFiles;
307     if (setrlimit (RLIMIT_NOFILE, &rl) == -1) {
308         // 记录警告日志
309         m_log.printf (Log::LEVEL_WAR, __FILE__, __LINE__,
310             "setrlimit: %s", strerror (errno));
311         // 返回失败
312         return false;
313     }
314
315     // 返回成功
316     return true;
317 }
```

函数调用图:

这是这个函数的调用关系图:

void WebCrawler::initSeeds (void) [private]

将种子链接压入原始统一资源定位符队列

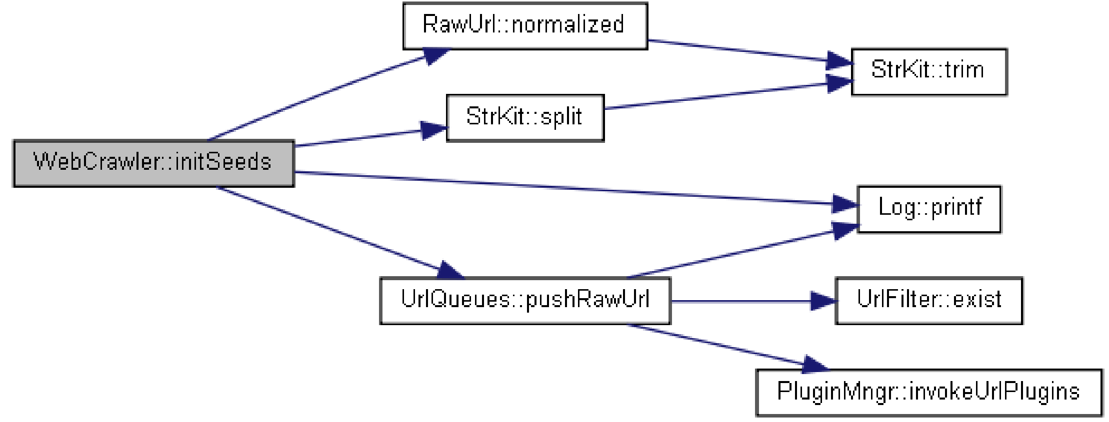
参考 Log::LEVEL_ERR, m_cfg, m_log, Configurator::m_seeds, m_urlQueues, RawUrl::normalized(), Log::printf(), UrlQueues::pushRawUrl() , 以及 StrKit::split().

参考自 init().

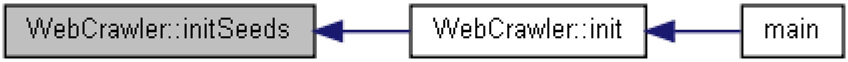
```
320                                 {
321     // 若配置器中的种子链接为空
322     if (m_cfg.m_seeds.empty ())
323         // 记录一般错误日志
324         m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
325             "没种子咋干活嘞? ");
326
```

```
327     // 拆分种子链接字符串，以逗号为分隔符，不限拆分次数
328     vector<string> seeds = StrKit::split (m_cfg.m_seeds, ",", 0);
329
330     // 依次处理每个种子链接
331     for (vector<string>::iterator it = seeds.begin ();
332         it != seeds.end (); ++it)
333         // 若规格化成功
334         if (RawUrl::normalized (*it))
335             // 压入原始统一资源定位符队列
336             m_urlQueues.pushRawUrl (RawUrl (*it));
337 }
```

函数调用图:



这是这个函数的调用关系图:



void WebCrawler::initSend (void) [private]

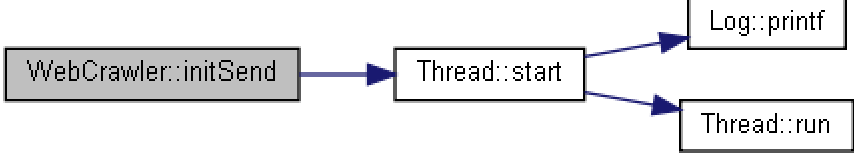
启动发送线程

参考 m_sendThread , 以及 Thread::start().

参考自 init().

```
392                                 {
393     // 通过发送线程对象启动发送线程
394     m_sendThread.start ();
395 }
```

函数调用图:



这是这个函数的调用关系图:

void WebCrawler::initTicker (void) const `[private]`

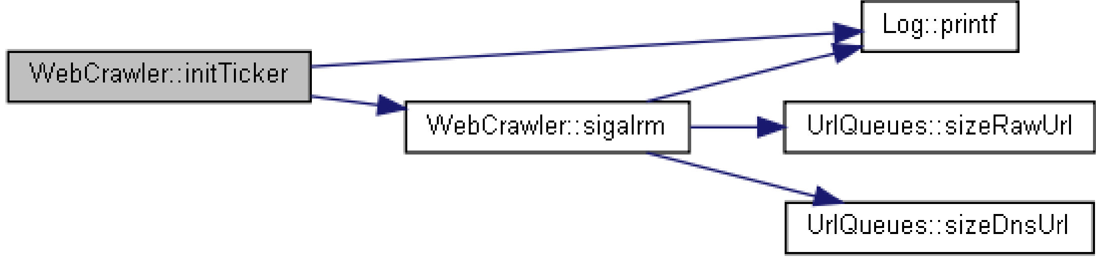
启动状态定时器

参考 Log::LEVEL_ERR, m_cfg, m_log, Configurator::m_statInterval, Log::printf() , 以及 sigalrm().

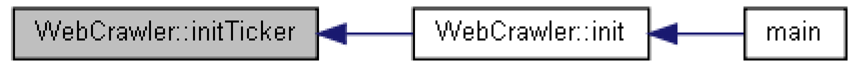
参考自 init().

```
357                                {
358     // 若配置器中的状态间隔有效
359     if (m_cfg.m_statInterval > 0) {
360         // 若设置SIGALRM(14)信号处理失败
361         if (signal (SIGALRM, sigalrm) == SIG_ERR)
362         /*
363         struct sigaction act = {};
364         act.sa_handler = sigalrm;
365         act.sa_flags = SA_RESTART;
366         if (sigaction (SIGALRM, &act, NULL) == -1)
367         */
368             // 记录一般错误日志
369             m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
370                 "signal: %s", strerror (errno));
371 
372         // 间隔时间
373         itimerval it;
374         // 初始间隔(秒)
375         it.it_value.tv_sec = m_cfg.m_statInterval;
376         // 初始间隔(微秒)
377         it.it_value.tv_usec = 0;
378         // 重复间隔(秒)
379         it.it_interval.tv_sec = m_cfg.m_statInterval;
380         // 重复间隔(微秒)
381         it.it_interval.tv_usec = 0;
382 
383         // 若设置真实间隔定时器失败
384         if (setitimer (ITIMER_REAL, &it, NULL) == -1)
385             // 记录一般错误日志
386             m_log.printf (Log::LEVEL_ERR, __FILE__, __LINE__,
387                 "setitimer: %s", strerror (errno));
388     }
389 }
```

函数调用图:



这是这个函数的调用关系图:



void WebCrawler::sigalrm (int *signum*) `[static], [private]`

SIGALRM(14)信号处理

参数:

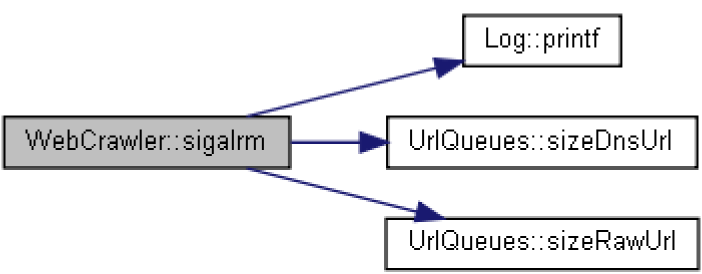
in	*signum*	信号编号

参考 g_app, Log::LEVEL_INF, m_curJobs, m_failure, m_log, m_start, m_success, m_urlQueues, Log::printf(), UrlQueues::sizeDnsUrl() , 以及 UrlQueues::sizeRawUrl().

参考自 initTicker().

```
211     {
212     // 抓取时间 = 当前时间 - 启动时间
213     time_t t = time (NULL) - g_app->m_start;
214
215     // 记录信息日志
216     g_app->m_log.printf (Log::LEVEL_INF, __FILE__, __LINE__,
217         "当前任务  原始队列  解析队列  抓取时间  成功次数  失败次数  成功率\n"
218         "------------------------------------------------------------------\n"
219         "%8d  %8u  %8u  %02d:%02d:%02d  %8u  %8u  %5u%%",
220         g_app->m_curJobs,
221         g_app->m_urlQueues.sizeRawUrl (),
222         g_app->m_urlQueues.sizeDnsUrl (),
223         t / 3600, t % 3600 / 60, t % 60,
224         g_app->m_success,
225         g_app->m_failure,
226         g_app->m_success ? g_app->m_success * 100 /
227             (g_app->m_success + g_app->m_failure) : 0);
228 }
```

函数调用图:



这是这个函数的调用关系图:

void WebCrawler::startJob (void)

启动一个抓取任务

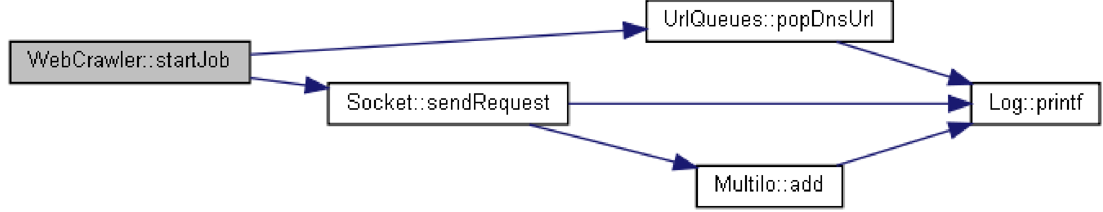
参考 m_cfg, m_cond, m_curJobs, Configurator::m_maxJobs, m_mutex, m_urlQueues, UrlQueues::popDnsUrl() , 以及 Socket::sendRequest().

参考自 SendThread::run().

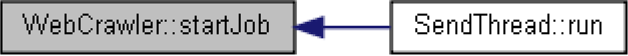
```
156                 {
157     // 加锁互斥锁
158     pthread_mutex_lock (&m_mutex);
159
160     // 若当前抓取任务数到限
161     while (m_curJobs == m_cfg.m_maxJobs)
162         // 等待条件变量
163         pthread_cond_wait (&m_cond, &m_mutex);
164
```

```
165     // 从解析统一资源定位符队列弹出解析统一资源定位符
166     DnsUrl dnsUrl = m_urlQueues.popDnsUrl ();
167
168     // 用所弹出解析统一资源定位符创建套接字对象
169     Socket* socket = new Socket (dnsUrl);
170     // 若通过套接字对象发送超文本传输协议请求成功
171     if (socket->sendRequest ())
172         // 当前抓取任务数加一
173         ++m_curJobs;
174     // 否则
175     else
176         // 销毁套接字对象
177         delete socket;
178
179     // 解锁互斥锁
180     pthread_mutex_unlock (&m_mutex);
181 }
```

函数调用图:



这是这个函数的调用关系图:



void WebCrawler::stopJob (bool *success* = `true`)

停止一个抓取任务

参数:

in	*success*	是否成功

参考 m_cfg, m_cond, m_curJobs, m_failure, Configurator::m_maxJobs, m_mutex , 以及 m_success.

参考自 RecvThread::run().

```
186     {
187     // 加锁互斥锁
188     pthread_mutex_lock (&m_mutex);
189
190     // 当前抓取任务数减一，若其由到限变为未到限
191     if (--m_curJobs == m_cfg.m_maxJobs - 1)
192         // 唤醒等待条件变量的线程
193         pthread_cond_signal (&m_cond);
194
195     // 若成功抓取
196     if (success)
197         // 成功次数加一
198         ++m_success;
199     // 否则
200     else
201         // 失败次数加一
202         ++m_failure;
```

```
203
204    // 解锁互斥锁
205    pthread_mutex_unlock (&m_mutex);
206 }
```

这是这个函数的调用关系图:

WebCrawler::stopJob ◀── RecvThread::run

类成员变量说明

Configurator WebCrawler::m_cfg

配置器

参考自 exec(), UrlQueues::fullDnsUrl(), UrlQueues::fullRawUrl(), DomainLimit::handler(), MaxDepth::handler(), HeaderFilter::handler(), DomainLimit::init(), init(), initDaemon(), initSeeds(), initTicker(), PluginMngr::load(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), Log::printf(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), startJob() , 以及 stopJob().

pthread_cond_t WebCrawler::m_cond [private]

条件变量

参考自 startJob(), stopJob(), WebCrawler() , 以及 ~WebCrawler().

int WebCrawler::m_curJobs [private]

当前抓取任务数

参考自 exec(), sigalrm(), startJob() , 以及 stopJob().

DnsThread WebCrawler::m_dnsThread

域名解析线程

参考自 initDns().

unsigned int WebCrawler::m_failure [private]

失败次数

参考自 sigalrm() , 以及 stopJob().

Log WebCrawler::m_log

日志

参考自 MultiIo::add(), MultiIo::del(), exec(), UrlQueues::extractUrl(), SaveHTMLToFile::handler(), SaveImageToFile::handler(), DomainLimit::handler(), HeaderFilter::handler(), MaxDepth::handler(), BloomFilter::hash(), init(), initDaemon(), initMaxFiles(), initSeeds(), initTicker(), PluginMngr::load(), Configurator::load(), MultiIo::MultiIo(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), Socket::recvResponse(), DnsThread::run(), SendThread::run(), RecvThread::run(), Socket::sendRequest(), sigalrm(), Thread::start() , 以及 MultiIo::wait().

MultiIo WebCrawler::m_multiIo

多路输入输出

参考自 exec() , 以及 Socket::sendRequest().

pthread_mutex_t WebCrawler::m_mutex [private]

互斥锁

参考自 startJob(), stopJob(), WebCrawler() , 以及 ~WebCrawler().

PluginMngr WebCrawler::m_pluginMngr

插件管理器

参考自 init(), UrlQueues::pushRawUrl() , 以及 Socket::recvResponse().

SendThread WebCrawler::m_sendThread

发送线程

参考自 initSend().

time_t WebCrawler::m_start [private]

启动时间

参考自 sigalrm().

unsigned int WebCrawler::m_success [private]

成功次数

参考自 sigalrm() , 以及 stopJob().

UrlQueues WebCrawler::m_urlQueues

统一资源定位符队列

参考自 exec(), SaveImageToFile::handler(), initSeeds(), Socket::recvResponse(), DnsThread::run(), sigalrm() , 以及 startJob().

该类的文档由以下文件生成:

- G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.h
- G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.cpp

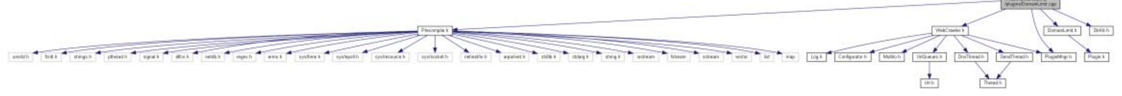
文件说明

G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.cpp 文件参考

实现::DomainLimit类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "DomainLimit.h"
#include "PluginMngr.h"
#include "StrKit.h"
```

DomainLimit.cpp 的引用(Include)关系图:



变量

- DomainLimit g_pluginDomainLimit
- WebCrawler * g_app

  应用程序对象

详细描述

实现::DomainLimit类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

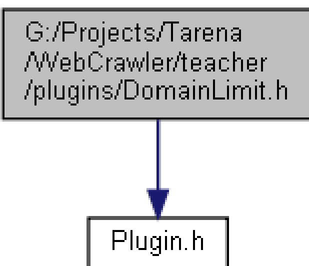
DomainLimit g_pluginDomainLimit

G:/Projects/Tarena/WebCrawler/teacher/plugins/DomainLimit.h 文件参考

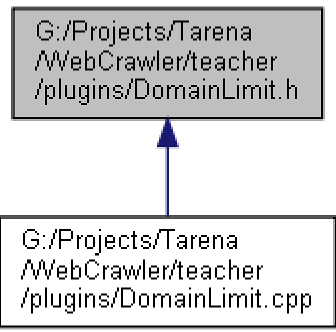
声明::DomainLimit类

```
#include "Plugin.h"
```

DomainLimit.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class DomainLimit

域名限制插件

详细描述

声明::DomainLimit类

作者:

闵卫

日期:

2015年11月20日

版本:

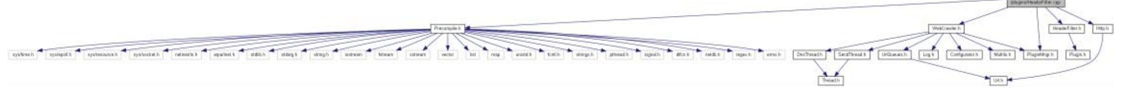
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.cpp 文件参考

实现::HeaderFilter类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "HeaderFilter.h"
#include "PluginMngr.h"
#include "Http.h"
```

HeaderFilter.cpp 的引用(Include)关系图:



变量

- HeaderFilter g_pluginHeaderFilter
- WebCrawler * g_app
  应用程序对象

详细描述

实现::HeaderFilter类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

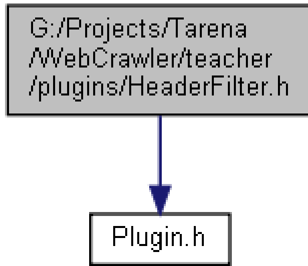
HeaderFilter g_pluginHeaderFilter

G:/Projects/Tarena/WebCrawler/teacher/plugins/HeaderFilter.h 文件参考

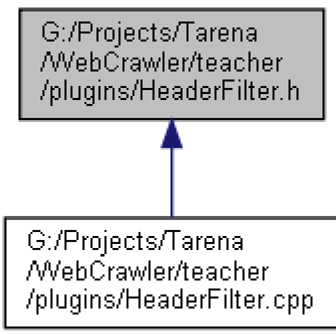
声明::HeaderFilter类

```
#include "Plugin.h"
```

HeaderFilter.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class HeaderFilter

超文本传输协议响应包头过滤器插件

详细描述

声明::HeaderFilter类

作者:

闵卫

日期:

2015年11月20日

版本:

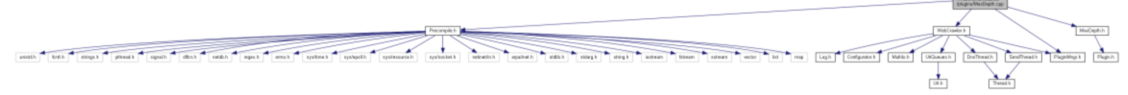
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.cpp 文件参考

实现::MaxDepth类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "MaxDepth.h"
#include "PluginMngr.h"
```

MaxDepth.cpp 的引用(Include)关系图:



变量

- MaxDepth g_pluginMaxDepth
- WebCrawler * g_app
  应用程序对象

详细描述

实现::MaxDepth类

作者:
闵卫

日期:
2015年11月20日

版本:
1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

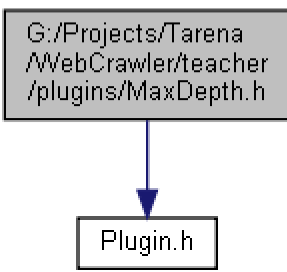
MaxDepth g_pluginMaxDepth

G:/Projects/Tarena/WebCrawler/teacher/plugins/MaxDepth.h 文件参考

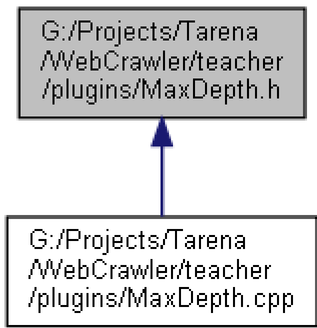
声明::MaxDepth类

```
#include "Plugin.h"
```

MaxDepth.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class MaxDepth

最大深度插件

详细描述

声明::MaxDepth类

作者:

闵卫

日期:

2015年11月20日

版本:

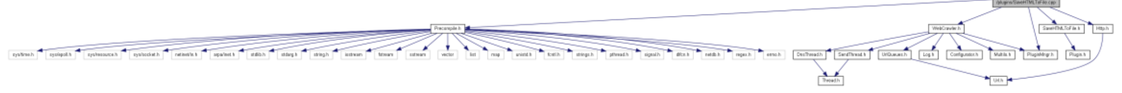
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.cpp 文件参考

实现::SaveHTMLToFile类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "SaveHTMLToFile.h"
#include "PluginMngr.h"
#include "Http.h"
```

SaveHTMLToFile.cpp 的引用(Include)关系图:



变量

- SaveHTMLToFile g_pluginSaveHTMLToFile
- WebCrawler * g_app
  应用程序对象

详细描述

实现::SaveHTMLToFile类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

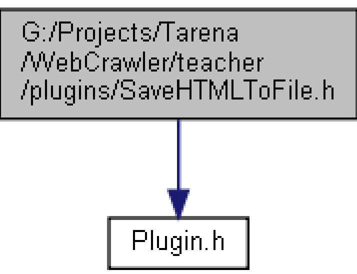
SaveHTMLToFile g_pluginSaveHTMLToFile

G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveHTMLToFile.h 文件参考

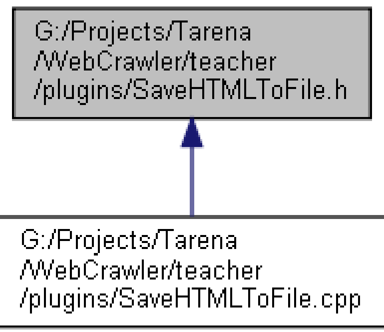
声明::SaveHTMLToFile类

```
#include "Plugin.h"
```

SaveHTMLToFile.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class SaveHTMLToFile

超文本标记语言文件存储插件

详细描述

声明::SaveHTMLToFile类

作者:

闵卫

日期:

2015年11月20日

版本:

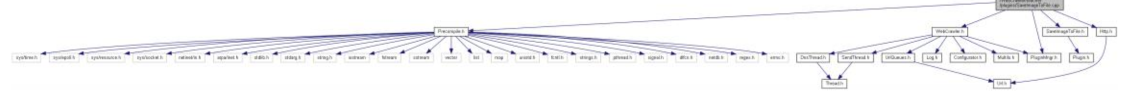
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.cpp 文件参考

实现::SaveImageToFile类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "SaveImageToFile.h"
#include "PluginMngr.h"
#include "Http.h"
```

SaveImageToFile.cpp 的引用(Include)关系图:



变量

- SaveImageToFile g_pluginSaveImageToFile
- WebCrawler * g_app
  应用程序对象

详细描述

实现::SaveImageToFile类

作者:
闵卫

日期:
2015年11月20日

版本:
1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

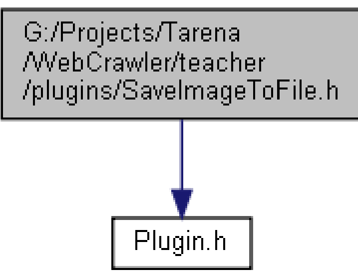
SaveImageToFile g_pluginSaveImageToFile

G:/Projects/Tarena/WebCrawler/teacher/plugins/SaveImageToFile.h 文件参考

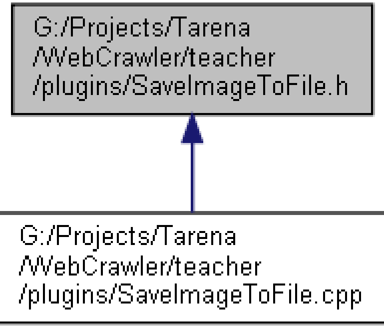
声明::SaveImageToFile类

```
#include "Plugin.h"
```

SaveImageToFile.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class SaveImageToFile

图像文件存储插件

详细描述

声明::SaveImageToFile类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.cpp 文件参考

实现::BloomFilter类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "BloomFilter.h"
```

BloomFilter.cpp 的引用(Include)关系图:

详细描述

实现::BloomFilter类

作者:

闵卫

日期:

2015年11月20日

版本:

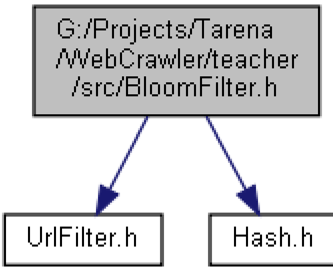
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/BloomFilter.h 文件参考

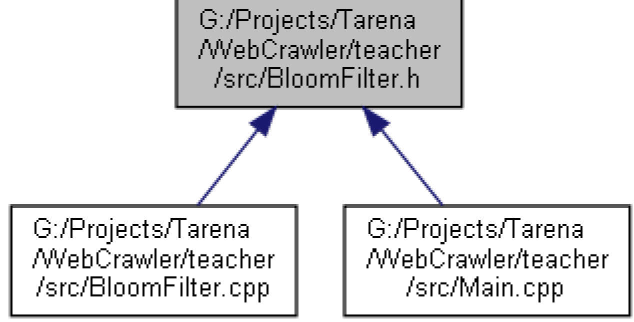
声明::BloomFilter类

```
#include "UrlFilter.h"
#include "Hash.h"
```

BloomFilter.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class BloomFilter

布隆过滤器

详细描述

声明::BloomFilter类

作者:

闵卫

日期:

2015年11月20日

版本:

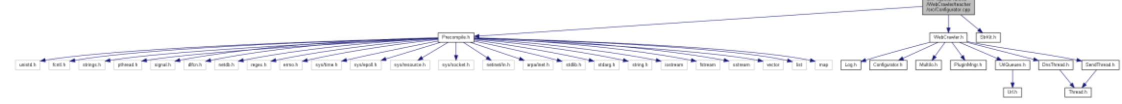
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.cpp 文件参考

实现::Configurator类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "StrKit.h"
```

Configurator.cpp 的引用(Include)关系图:



详细描述

实现::Configurator类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Configurator.h 文件参考

声明::Configurator类
此图展示该文件直接或间接的被哪些文件引用了:

类

- class Configurator

配置器

详细描述

声明::Configurator类

作者:

闵卫

日期:

2015年11月20日

版本:

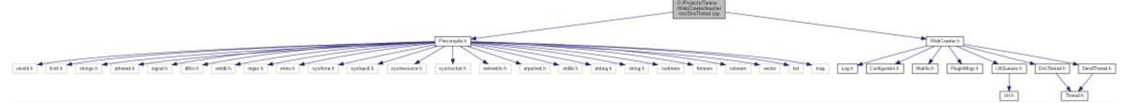
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.cpp 文件参考

实现::DnsThread类

```
#include "Precompile.h"
#include "WebCrawler.h"
```

DnsThread.cpp 的引用(Include)关系图:



详细描述

实现::DnsThread类

作者:

闵卫

日期:

2015年11月20日

版本:

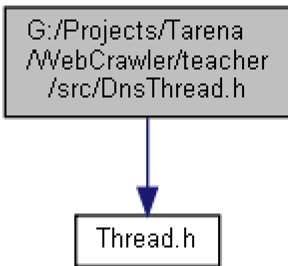
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/DnsThread.h 文件参考

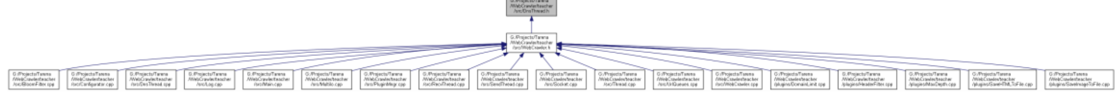
声明::DnsThread类

`#include "Thread.h"`

DnsThread.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class DnsThread

域名解析线程

详细描述

声明::DnsThread类

作者:

闵卫

日期:

2015年11月20日

版本:

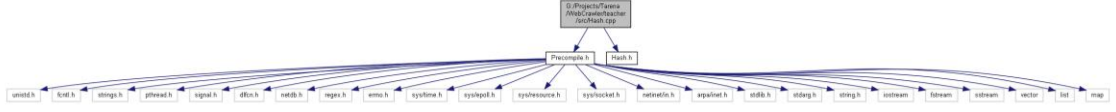
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Hash.cpp 文件参考

实现::Hash类

```
#include "Precompile.h"
#include "Hash.h"
```

Hash.cpp 的引用(Include)关系图:



详细描述

实现::Hash类

作者:

闵卫

日期:

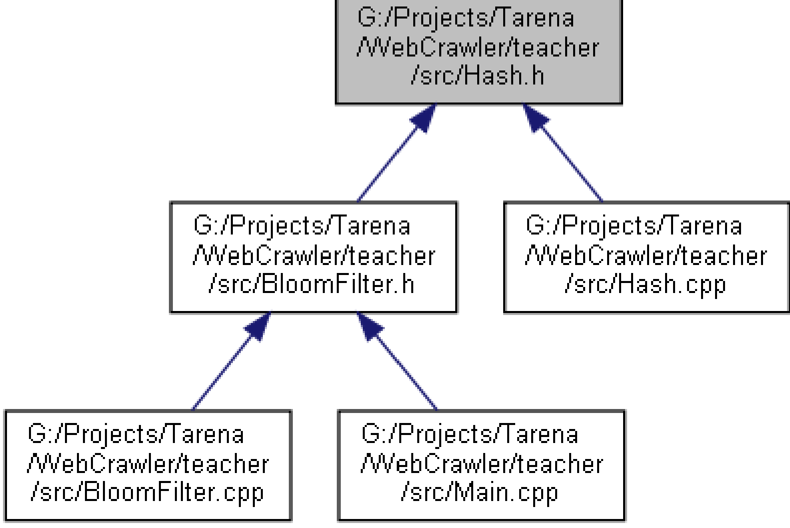
2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Hash.h 文件参考

声明::Hash类
此图展示该文件直接或间接的被哪些文件引用了:



类

- class Hash

哈希器

详细描述

声明::Hash类

作者:

闵卫

日期:

2015年11月20日

版本:

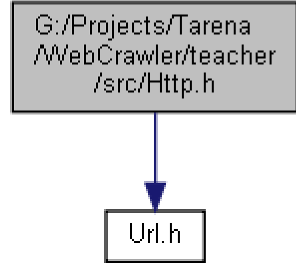
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Http.h 文件参考

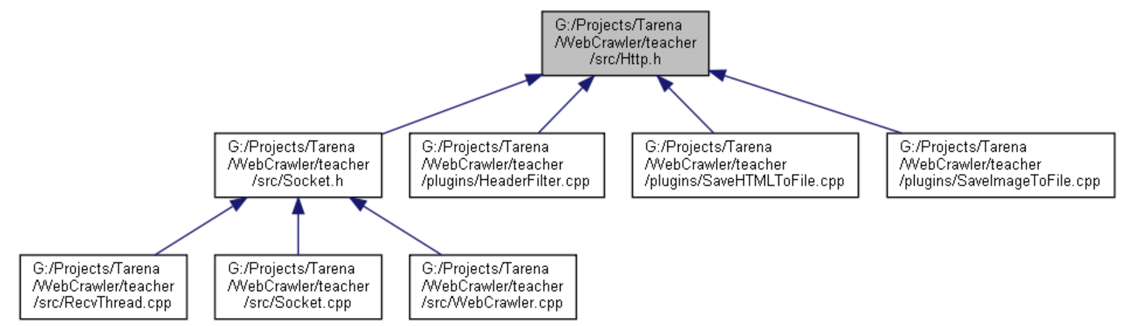
定义::HttpHeader类和::HttpResponse类

```
#include "Url.h"
```

Http.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class HttpHeader
  超文本传输协议响应包头 *更多...*
- class HttpResponse

超文本传输协议响应

详细描述

定义::HttpHeader类和::HttpResponse类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

类说明

class HttpHeader

超文本传输协议响应包头

HttpHeader 的协作图:

HttpHeader
+ m_statusCode
+ m_contentType

类成员:

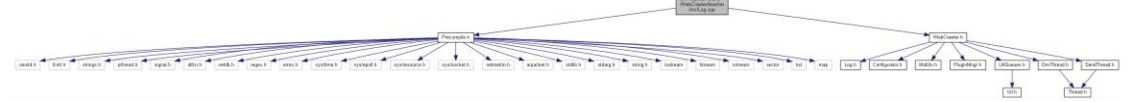
string	m_contentType	内容类型
int	m_statusCode	状态码

G:/Projects/Tarena/WebCrawler/teacher/src/Log.cpp 文件参考

实现::Log类

```
#include "Precompile.h"
#include "WebCrawler.h"
```

Log.cpp 的引用(Include)关系图:



详细描述

实现::Log类

作者:

闵卫

日期:

2015年11月20日

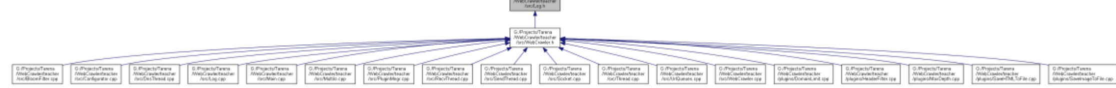
版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Log.h 文件参考

声明::Log类

此图展示该文件直接或间接的被哪些文件引用了:



类

- class Log

日志

详细描述

声明::Log类

作者:

闵卫

日期:

2015年11月20日

版本:

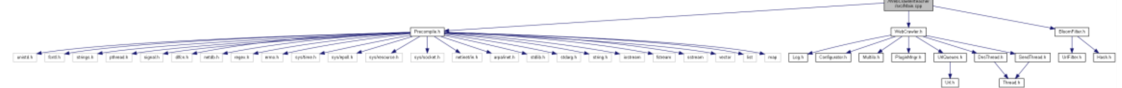
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Main.cpp 文件参考

定义::main函数

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "BloomFilter.h"
```

Main.cpp 的引用(Include)关系图:



函数

- static void showVersion (void)
  显示版本信息
- static void splashScreen (void)
  显示启动画面
- static void showUsage (string const &cmd)
  显示使用方法
- int main (int argc, char *argv[])
  进程入口函数

变量

- BloomFilter g_filter
- WebCrawler * g_app = new WebCrawler (g_filter)
  应用程序对象

详细描述

定义::main函数

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

函数说明

int main (int *argc*, char * *argv*[])

进程入口函数

返回值:

EXIT_SUCCESS	进程成功退出
EXIT_FAILURE	进程失败退出

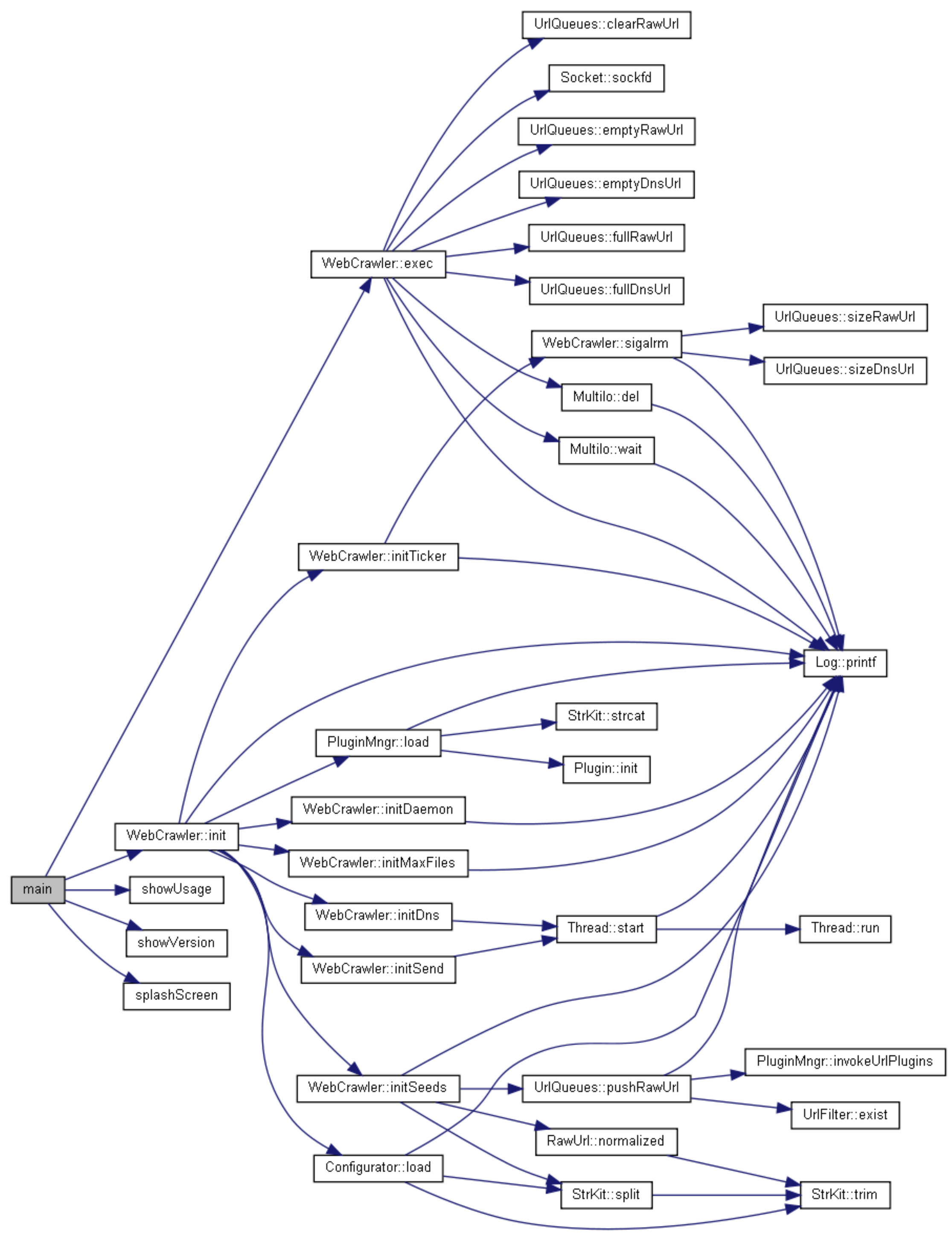
参数:

in	*argc*	命令行参数个数
in	*argv*	命令行参数列表

参考 WebCrawler::exec(), WebCrawler::init(), showUsage(), showVersion() , 以及 splashScreen().

```
 66  {
 67    // 是否以精灵模式运行，初始化为"否"
 68    bool daemon = false;
 69
 70    // 解析命令行选项
 71    char ch;
 72    while ((ch = getopt (argc, argv, "vdh")) != -1)
 73        // 若选项...
 74        switch (ch) {
 75            // 为v
 76            case 'v':
 77                // 显示版本信息
 78                showVersion ();
 79                return EXIT_SUCCESS;
 80
 81            // 为d
 82            case 'd':
 83                // 以精灵模式运行
 84                daemon = true;
 85                break;
 86
 87            // 为h或者?
 88            case 'h':
 89            case '?':
 90                // 显示使用方法
 91                showUsage (argv[0]);
 92                return EXIT_SUCCESS;
 93
 94            // 为其它
 95            default:
 96                // 显示使用方法
 97                showUsage (argv[0]);
 98                return EXIT_FAILURE;
 99        }
100
101    // 显示启动画面
102    splashScreen ();
103
104    // 初始化应用程序对象
105    g_app->init (daemon);
106    // 执行多路输入输出循环
107    g_app->exec ();
108
109    return EXIT_SUCCESS;
110  }
```

函数调用图:



static void showUsage (string const & *cmd*) `[static]`

显示使用方法

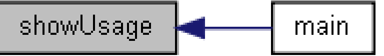
参数:

in	*cmd*	启动命令

参考自 main().

```
49   {
50     cout << endl;
51
52     cout << "用法: " << cmd << " [-v|-d|-h]" << endl;
53     cout << "选项: -v - 版权信息" << endl;
54     cout << "      -d - 启动精灵" << endl;
55     cout << "      -h - 帮助信息" << endl;
56
57     cout << endl;
58 }
```

这是这个函数的调用关系图:



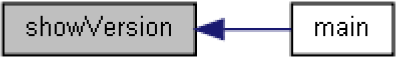
static void showVersion (void) [static]

显示版本信息

参考自 main().

```
15                              {
16     cout << endl;
17
18     cout << "网络爬虫 1.0 版" << endl;
19     cout << "版权所有 (C) 2015 达内科技" << endl;
20
21     cout << endl;
22 }
```

这是这个函数的调用关系图:



static void splashScreen (void) [static]

显示启动画面

参考自 main().

```
25                               {
26     cout << endl;
27
28     // 打开画面文件输入流
29     ifstream ifs ("WebCrawler.scr");
30     // 若成功
31     if (ifs) {
32         // 行字符串
33         string line;
34         // 逐行读取画面文件中的文本
35         while (getline (ifs, line))
36             // 打印到标准输出
37             cout << line << endl;
38         // 关闭画面文件输入流
39         ifs.close ();
40     }
```

```
41
42     // 打印空行
43     cout << endl;
44 }
```

这是这个函数的调用关系图:

splashScreen ← main

变量说明

WebCrawler* g_app = new WebCrawler (g_filter)

应用程序对象

参考自 MultiIo::add(), MultiIo::del(), UrlQueues::extractUrl(), UrlQueues::fullDnsUrl(), UrlQueues::fullRawUrl(), BloomFilter::hash(), PluginMngr::load(), Configurator::load(), MultiIo::MultiIo(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), Log::printf(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), Socket::recvResponse(), DnsThread::run(), SendThread::run(), RecvThread::run(), Socket::sendRequest(), WebCrawler::sigalrm(), Thread::start() , 以及 MultiIo::wait().

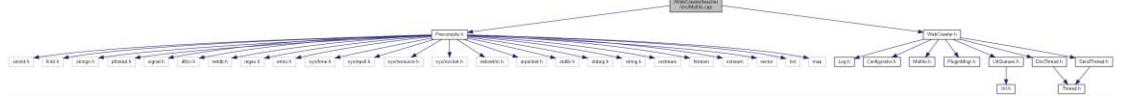
BloomFilter g_filter

G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.cpp 文件参考

实现::MultiIo类

```
#include "Precompile.h"
#include "WebCrawler.h"
```

MultiIo.cpp 的引用(Include)关系图:



详细描述

实现::MultiIo类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/MultiIo.h 文件参考

声明::MultiIo类
此图展示该文件直接或间接的被哪些文件引用了:

类

- class MultiIo

多路输入输出

详细描述

声明::MultiIo类

作者:

闵卫

日期:

2015年11月20日

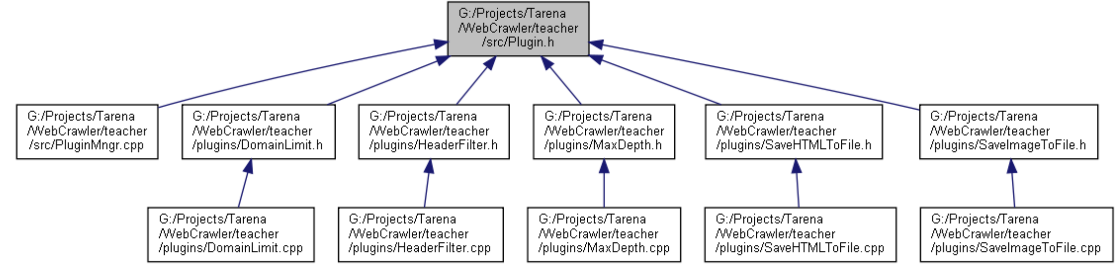
版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Plugin.h 文件参考

定义::Plugin接口类

此图展示该文件直接或间接的被哪些文件引用了:



类

- class Plugin

插件接口

详细描述

定义::Plugin接口类

作者:

闵卫

日期:

2015年11月20日

版本:

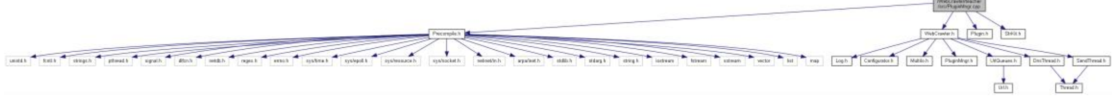
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.cpp 文件参考

实现::PluginMngr类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "Plugin.h"
#include "StrKit.h"
```

PluginMngr.cpp 的引用(Include)关系图:



详细描述

实现::PluginMngr类

作者:

闵卫

日期:

2015年11月20日

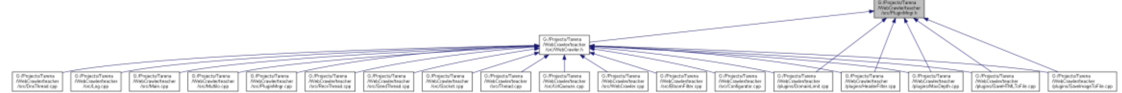
版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/PluginMngr.h 文件参考

声明::PluginMngr类

此图展示该文件直接或间接的被哪些文件引用了:



类

- class PluginMngr

插件管理器

详细描述

声明::PluginMngr类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Precompile.h 文件参考

预编译头文件

```
#include <unistd.h>
#include <fcntl.h>
#include <strings.h>
#include <pthread.h>
#include <signal.h>
#include <dlfcn.h>
#include <netdb.h>
#include <regex.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/epoll.h>
#include <sys/resource.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <list>
#include <map>
```

Precompile.h 的引用(Include)关系图:

此图展示该文件直接或间接的被哪些文件引用了:

详细描述

预编译头文件

作者:

闵卫

日期:

2015年11月20日

版本:

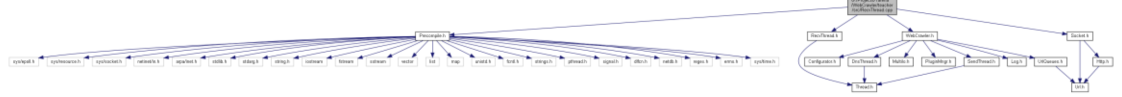
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.cpp 文件参考

实现::RecvThread类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "RecvThread.h"
#include "Socket.h"
```

RecvThread.cpp 的引用(Include)关系图:



详细描述

实现::RecvThread类

作者:

闵卫

日期:

2015年11月20日

版本:

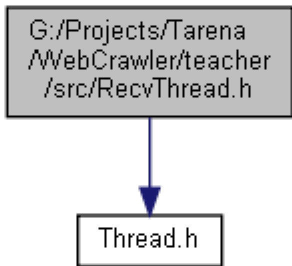
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/RecvThread.h 文件参考

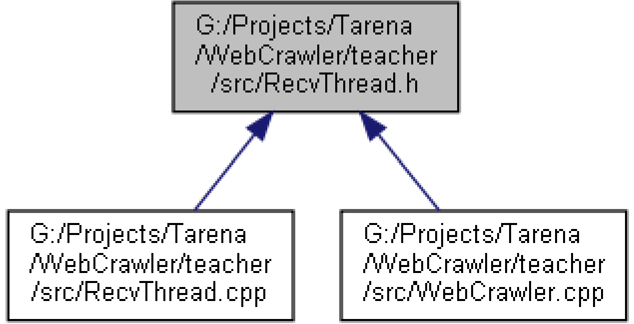
声明::RecvThread类

```
#include "Thread.h"
```

RecvThread.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class RecvThread

接收线程

详细描述

声明::RecvThread类

作者:

闵卫

日期:

2015年11月20日

版本:

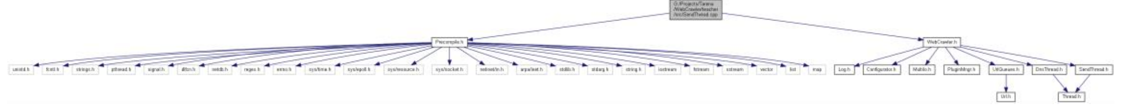
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.cpp 文件参考

实现::SendThread类

```
#include "Precompile.h"
#include "WebCrawler.h"
```

SendThread.cpp 的引用(Include)关系图:



详细描述

实现::SendThread类

作者:

闵卫

日期:

2015年11月20日

版本:

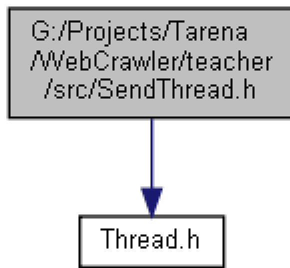
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/SendThread.h 文件参考

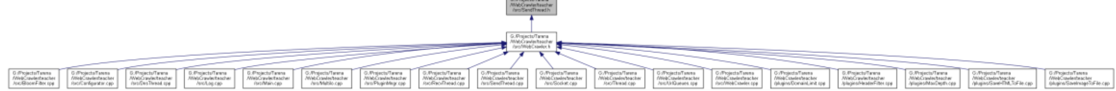
声明::SendThread类

```
#include "Thread.h"
```

SendThread.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class SendThread

发送线程

详细描述

声明::SendThread类

作者:

闵卫

日期:

2015年11月20日

版本:

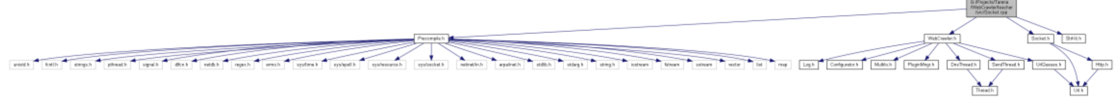
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Socket.cpp 文件参考

实现::Socket类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "Socket.h"
#include "StrKit.h"
```

Socket.cpp 的引用(Include)关系图:



详细描述

实现::Socket类

作者:

闵卫

日期:

2015年11月20日

版本:

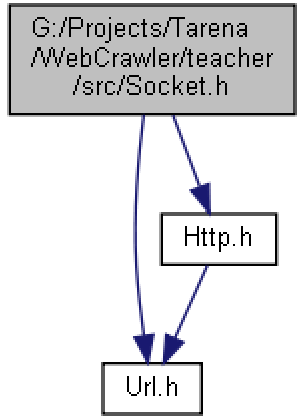
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Socket.h 文件参考

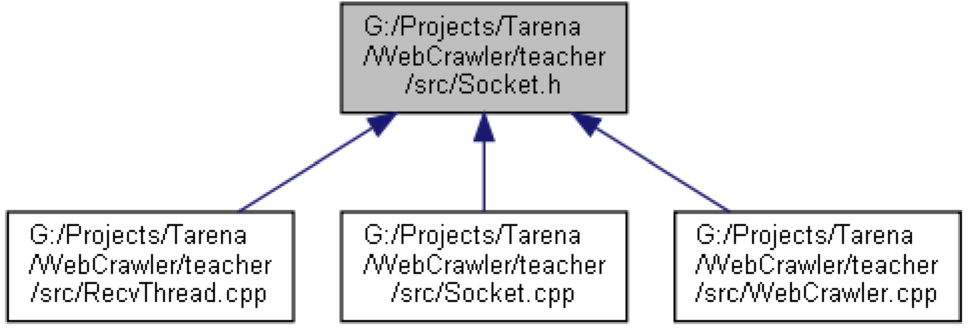
声明::Socket类

```
#include "Url.h"
#include "Http.h"
```

Socket.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class Socket

套接字

详细描述

声明::Socket类

作者:

闵卫

日期:

2015年11月20日

版本:

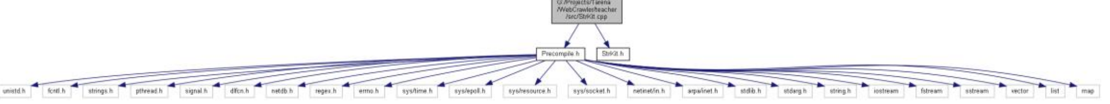
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.cpp 文件参考

实现::StrKit类

```
#include "Precompile.h"
#include "StrKit.h"
```

StrKit.cpp 的引用(Include)关系图:



详细描述

实现::StrKit类

作者:

闵卫

日期:

2015年11月20日

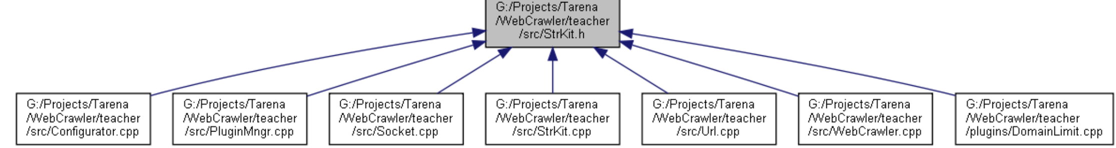
版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/StrKit.h 文件参考

声明::StrKit类

此图展示该文件直接或间接的被哪些文件引用了:



类

- class StrKit

字符串工具包

详细描述

声明::StrKit类

作者:

闵卫

日期:

2015年11月20日

版本:

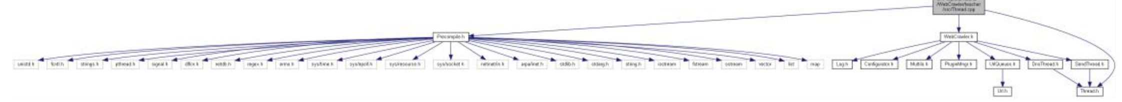
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Thread.cpp 文件参考

实现::Thread抽象基类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "Thread.h"
```

Thread.cpp 的引用(Include)关系图:



详细描述

实现::Thread抽象基类

作者:

闵卫

日期:

2015年11月20日

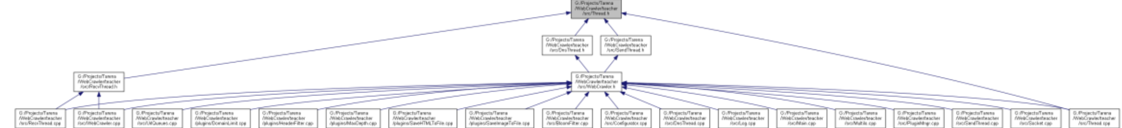
版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Thread.h 文件参考

声明::Thread抽象基类

此图展示该文件直接或间接的被哪些文件引用了:



类

- class Thread

线程

详细描述

声明::Thread抽象基类

作者:

闵卫

日期:

2015年11月20日

版本:

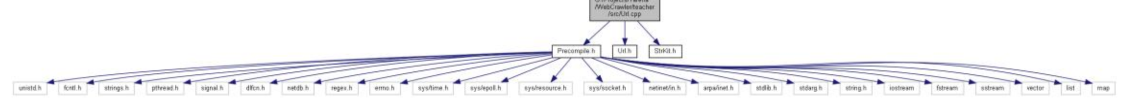
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Url.cpp 文件参考

实现::RawUrl类和::DnsUrl类

```
#include "Precompile.h"
#include "Url.h"
#include "StrKit.h"
```

Url.cpp 的引用(Include)关系图:



详细描述

实现::RawUrl类和::DnsUrl类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/Url.h 文件参考

声明::RawUrl类和::DnsUrl类

此图展示该文件直接或间接的被哪些文件引用了:

类

- class RawUrl
- *原始统一资源定位符* class DnsUrl

解析统一资源定位符

详细描述

声明::RawUrl类和::DnsUrl类

作者:

闵卫

日期:

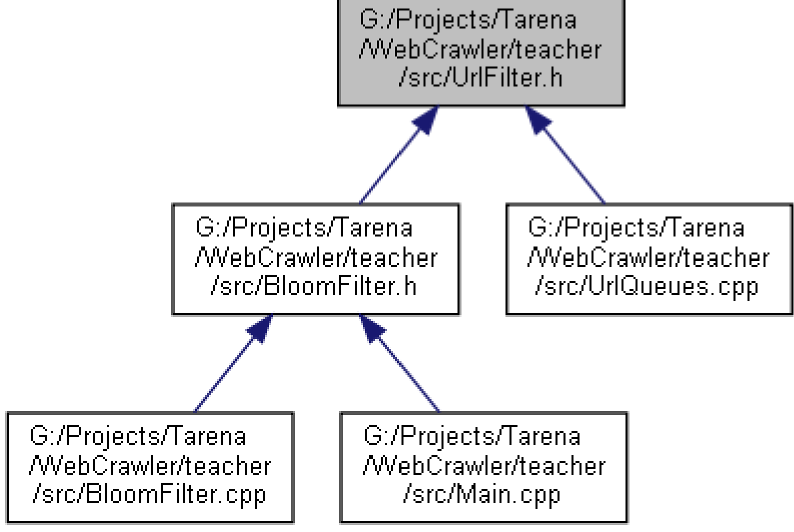
2015年11月20日

版本:

1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/UrlFilter.h 文件参考

定义::UrlFilter接口类
此图展示该文件直接或间接的被哪些文件引用了:



类

- class UrlFilter

统一资源定位符过滤器接口

详细描述

定义::UrlFilter接口类

作者:

闵卫

日期:

2015年11月20日

版本:

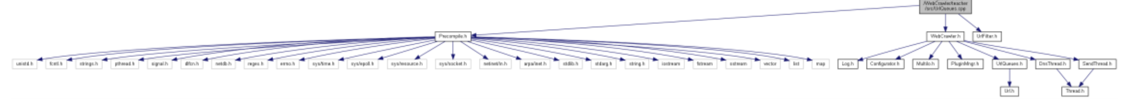
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.cpp 文件参考

实现::UrlQueues类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "UrlFilter.h"
```

UrlQueues.cpp 的引用(Include)关系图:



详细描述

实现::UrlQueues类

作者:

闵卫

日期:

2015年11月20日

版本:

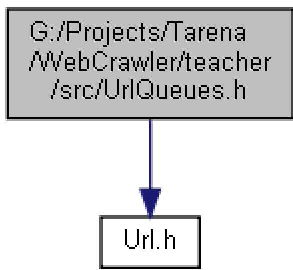
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/UrlQueues.h 文件参考

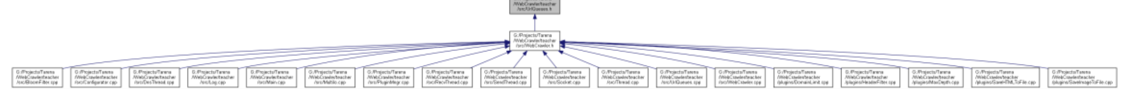
声明::UrlQueues类

```
#include "Url.h"
```

UrlQueues.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class UrlQueues

统一资源定位符队列

详细描述

声明::UrlQueues类

作者:

闵卫

日期:

2015年11月20日

版本:

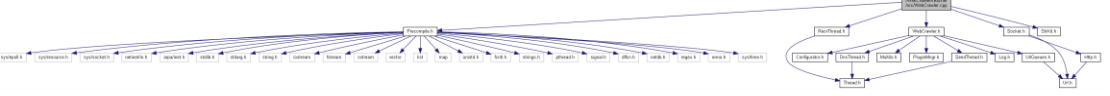
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.cpp 文件参考

实现::WebCrawler类

```
#include "Precompile.h"
#include "WebCrawler.h"
#include "RecvThread.h"
#include "Socket.h"
#include "StrKit.h"
```

WebCrawler.cpp 的引用(Include)关系图:



详细描述

实现::WebCrawler类

作者:

闵卫

日期:

2015年11月20日

版本:

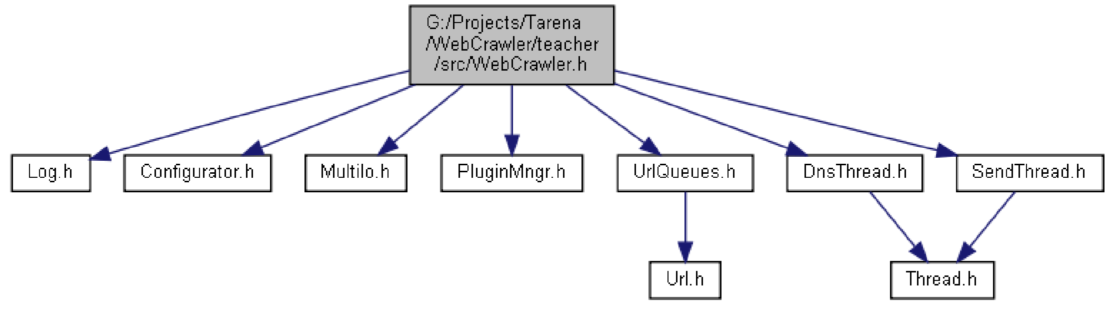
1.0.0.1

G:/Projects/Tarena/WebCrawler/teacher/src/WebCrawler.h 文件参考

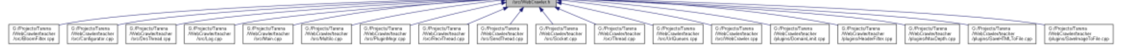
声明::WebCrawler类

```
#include "Log.h"
#include "Configurator.h"
#include "MultiIo.h"
#include "PluginMngr.h"
#include "UrlQueues.h"
#include "DnsThread.h"
#include "SendThread.h"
```

WebCrawler.h 的引用(Include)关系图:



此图展示该文件直接或间接的被哪些文件引用了:



类

- class WebCrawler

网络爬虫 变量

- WebCrawler * g_app
  应用程序对象

详细描述

声明::WebCrawler类

作者:

闵卫

日期:

2015年11月20日

版本:

1.0.0.1

变量说明

WebCrawler* g_app

应用程序对象

参考自 MultiIo::add(), MultiIo::del(), UrlQueues::extractUrl(), UrlQueues::fullDnsUrl(), UrlQueues::fullRawUrl(), BloomFilter::hash(), PluginMngr::load(), Configurator::load(), MultiIo::MultiIo(), UrlQueues::popDnsUrl(), UrlQueues::popRawUrl(), Log::printf(), UrlQueues::pushDnsUrl(), UrlQueues::pushRawUrl(), Socket::recvResponse(), DnsThread::run(), SendThread::run(), RecvThread::run(), Socket::sendRequest(), WebCrawler::sigalrm(), Thread::start() , 以及 MultiIo::wait().

索引
INDEX