关注人服务文档整理

系统名称	关注人服务
项目负责人	
作者	袁立舟
文档提交日期	2015-7-15

目录

一、DA 框架

关注人服务以插件的形式被 DA 框架调用来为客户端提供服务。插件通过集成 BaseResource 基类，通过调用 Resource::getResource<RealtimeRWIndex>("rtindex") 来使用插件。

DA 框架在搜索架构中起着重要的作用，多个插件都通过该框架来完成服务，包括 da_eva 的所有插件以及社会化实时序的部分插件，例如本文档的关注人服务插件。

在调试插件时，对 DA 框架的各个模块的熟悉熟知很关键，因此文档在 DA 框架的视野下分析关注人服务，从 DA 的 Run 函数开始分析整个服务的生命周期。

1.1 DA 框架的执行流程

入口函数是 dashell.cpp 的 Run()，并在其中顺序调用以下几个函数在完成具体插件的功能：

ParseParam()
GlobalInit()
DARun()
GlobalDestroy()

ParseParam()：参数解析，包括对工作路径、配置文件、日志文件的解析；
GlobalDestroy()完成资源、socket 服务、锁等的释放；
GlobalInit()和 DARun()两个函数分析如下。

1.2 全局初始化 |GlobalInit()

全局初始化函数，包括以下功能：

1.2.1 日志初始化 | InitLog()

初始化日志 level(0 表示 debug,warning,fatal,notice,trace 类型的日志都会生成)；根据配置文件生成日志文件名的前缀，例如本例中是 gs_r.log.* 其中*是上面五种日志类型。

1.2.2 DA 配置文件初始化 | DAStaticConfig ::Init()

DAStaticConfig ::Instance() |daconfig.h：生成 DA 配置文件的实例；

DAStaticConfig ::Init()|daconfig.cpp: 根据命令参数的 workpath 和 confName 指定的配置文件来初始化上步的 DA 实例，包括以下参数（可选）：workPort：工作端口；controlPort：控制端口； socketNum：sokect 服务的个数；queueLen：命令队列的长度；timeOut：超时时间；readTimeOut：读超时时间；writeTimeout：写超时时间；stayTimeOut：存活时间；taskTimeOut：任务超时时间；

下面三个文件的路径初始化比较重要：

1） plugins.xml，插件配置文件

2） algorithms.xml，算法配置文件

3） dict.conf，字典配置文件，这个有些服务可能不需要。

这些配置都写在配置文件里，可灵活配置，通过上面步骤完成了配置的初始化工作。

1.2.3 插件初始化 | PluginsManager::Init()

PluginsManager::Instance() | plugins_manager.h：单例模式，生成插件管理器的实例对象；PluginsManager::Init()|plugins_manager.cpp：初始化插件配置，经典的 plugins.xml 内容如下所示：

```
<plugins>
    <plugin so="plugins/libinputplug.so" switch="on" />
    <plugin so="plugins/plug_dupcont.so" switch="on" />
    <plugin so="plugins/libqi.so" switch="on" />
    <plugin so="plugins/libad.so" switch="off" />
    <plugin so="plugins/libtopic.so" switch="off" />
    <plugin so="plugins/libuidfilter.so" switch="off" />
    <plugin so="plugins/libindex_plug.so" switch="off" />
    <plugin so="plugins/liblang.so" switch="off" />
    <plugin so="plugins/libkwfilter.so" switch="off" />
    <plugin so="plugins/librabbish.so" switch="off" />
    <plugin so="plugins/libduplicate.so" switch="off" />
    <plugin so="plugins/plug_domain.so" switch="off" />
    <plugin so="plugins/plug_content.so" switch="off" />
    <plugin so="plugins/liburldup.so" switch="off" />
    <plugin so="plugins/plug_user.so" switch="off" />
    <plugin so="plugins/plug_hdqi.so" switch="off" />
    <plugin so="plugins/libplug_cover.so" switch="off" />
    <plugin so="plugins/liboutputplug.so" switch="on" />
</plugins>
```

图-1 plugins.xml 实例

本函数的功能是把配置文件中的 switch 选项为 on 的 so 加载进来，利用了如下数据结构其中 std::map<std::string, void *> ，其中 string 为 so 的路径，void* 存储该对应的共享文件加载到系统后的句柄。过程：根据该路径（string 内容）调用系统函数 dlopen(dlfcn.h)，该函数的作用是打开共享对象文件，将其映射到系统中，并返回一个句柄，然后 dlsym 函数根据该句柄加载该共享文件的运行时地址。

1.2.4 算法初始化 | AlgorithmsManager::Init()

实现在 algorithms_manager.cpp 中，单例模式，典型的 algorithms.xml 如下图所示：首先解析出 clusters 字段，然后再解析出 strategies 字段，其中配置策略族的名称，线程数目，算法的运行引擎（主要包括 cmdpipe 和 sequence 两种），不同驱动引擎加载不同的 socket 实例，然后遍历所有的 strategy 所有标签；调用

InitStrategy()函数来初始化策略读取其中的每个策略的配置文件、xml 文件、开关等信息，并通过插件管理器调用对应 so，并把对应的策略加载到 vector<Strategies>中。

```
<clusters>
    <strategies engine="sequence" threadNum = "1" name="WeiboQuality" switch="on">
        <strategy name="inputplug_t" switch="on" config="queue.ini"/>
        <strategy name="dupcontplug_t" switch="on" config="plug_dupcont.ini"/>
        <strategy name="qiplug_t"  switch="on" config="qi.ini" qibits="0,5"/>
        <strategy name="adplug_t"  switch="off" config="ad.ini" adbits="0,1"/>
        <strategy name="topicplug_t"  switch="off" config="topicWords.ini"/>
        <strategy name="uidplug_t" switch="off" config="uidfilter.ini" qibits="7"/>
        <strategy name="indexplug_t" switch="off" config="index.ini" qibits="16"/>
        <strategy name="langplug_t"  switch="off" config="language.ini" qibits="-1"/>
        <strategy name="kwfilterplug_t"  switch="off" config="kwfilter.ini" qibits="11"/>
        <strategy name="duplicate_t" switch="off" config="duplicate.ini" qibits="14"/>
        <strategy name="domainplug_t" switch="off" config="plug_domain.ini"/>
        <strategy name="contentplug_t" switch="off" config="plug_content.ini"/>
        <strategy name="urldup_t" switch="off" config="urldup.ini" qibits="17"/>
        <strategy name="userplug_t" switch="off" config="plug_user.ini" qibits="7"/>
        <strategy name="rabplug_t" switch="off" config="rabplug.ini" qibits="13"/>
        <strategy name="coverplug_t" switch="off" config="cover.ini" dict="/data/superwhite/"/>
        <strategy name="hdqiplug_t" switch="off" config="plug_hdqi.ini"/>
        <strategy name="outputplug_t" switch="on" config="queueout.ini"/>

    </strategies>
</clusters>
```

图-2 algorithms.xml 实例

1.3 DA 的执行函数 | DARun()

首先根据 AlgorithmsManager::Instance()获取静态的算法管理器实例，根据管理器类的静态对象中的线程数目项 n，每个策略都启动 n 个线程。

通过 DAWork()|work_thread.cpp 调用引擎驱动算法来执行每个策略算法。首先读取策略配置的 engine 项，加载对应的驱动引擎 engine = CMDPipe()，通过 Init 函数和 Run 函数来完成插件算法的调用。

1.3.1 算法驱动器初始化 Init()| engine.cpp

该函数来初始化算法引擎，主要包括以下动作：首先由算法名称调用其工厂函数生成对应的实例并赋值给抽象父类 BaseAlgorithms 的对象，然后调用 AddAlgorithm() –> Init()→DynLoad()。

AddAlgorithm()：添加算法到算法实例数组 BaseAlgorithms ** m_algorithms;

Init()：为虚函数，调用 BaseAlgorithms 子类对象的 Init 函数来完成相应的初始化工作；

DynLoad():为虚函数，调用子对象的函数，动态加载，一般情况没有具体动作则返回 true 即可。

1.3.2 算法执行 engine->Run()

最重要的部分：engine->Run()，通过 engine 父类运行时加载真正的算法驱动引擎的 Run 函数，例如 CMDPipe 引擎，MiniHttpd 引擎等。

下面通过 CMDPipe::Run()来观察下其算法流程：

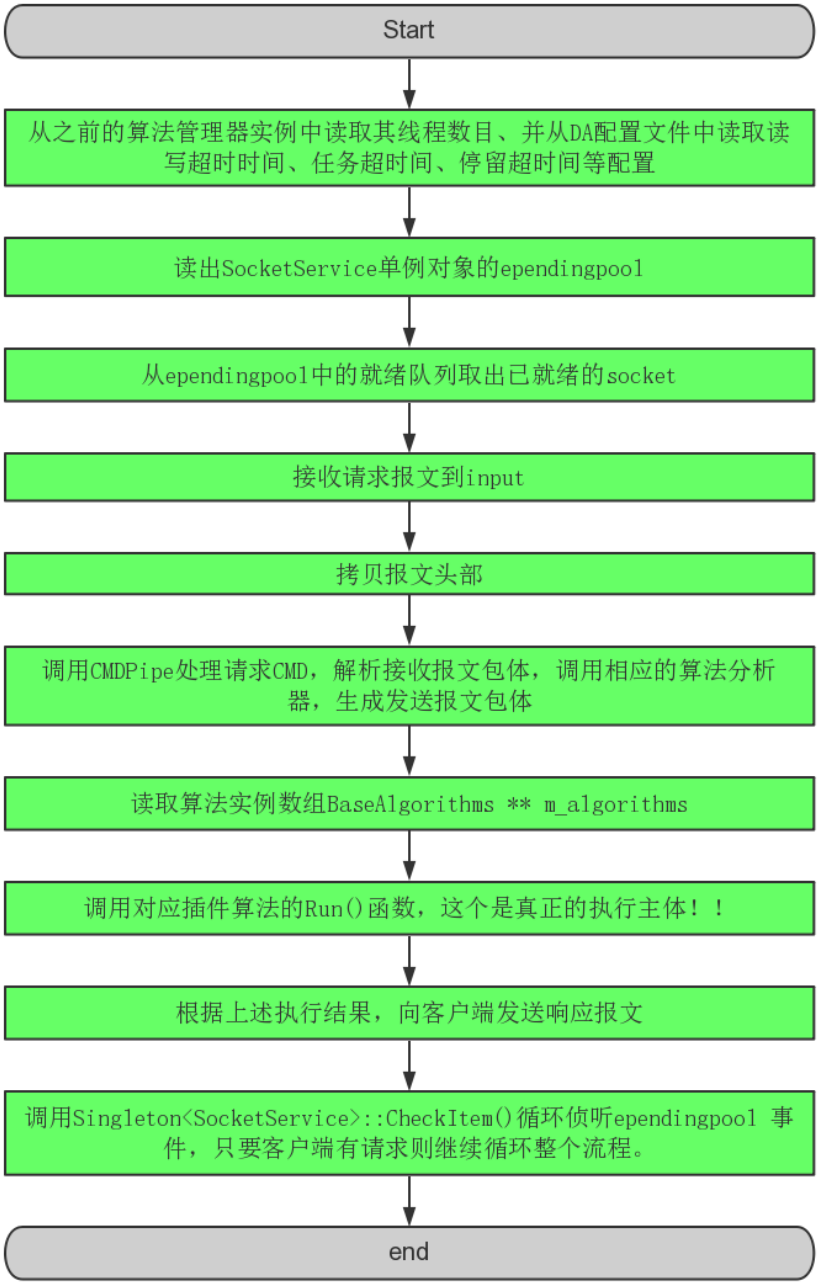

图-3CMDPipe::Run()执行流程

二、物理设计

关注人服务 SVN 路径：
https://svn1.intra.sina.com.cn/realtime_search/search_65/trunk/graphsearch/socialrealtime/

线上机器：（主用 146-147 两台机器，189 已下线）

```
<bcgroup group_id="7" level="1" priority="5" count="5000" name="social_follows">
        <bc ip="10.77.104.146" port="3022" disabled="0" priority="2" />
        <bc ip="10.77.104.147" port="3022" disabled="0" priority="2" />
        <bc ip="10.75.30.189" port="3022" disabled="0" priority="1" />
 </bcgroup>
```

物理设计研究系统中的物理实体以及他们的相互联系。系统每个逻辑实体都驻留在一个物理实体中，在服务中实时索引检索逻辑驻留在共享内存中。下面是其物理设计的简单描述：

2.1 内存结构

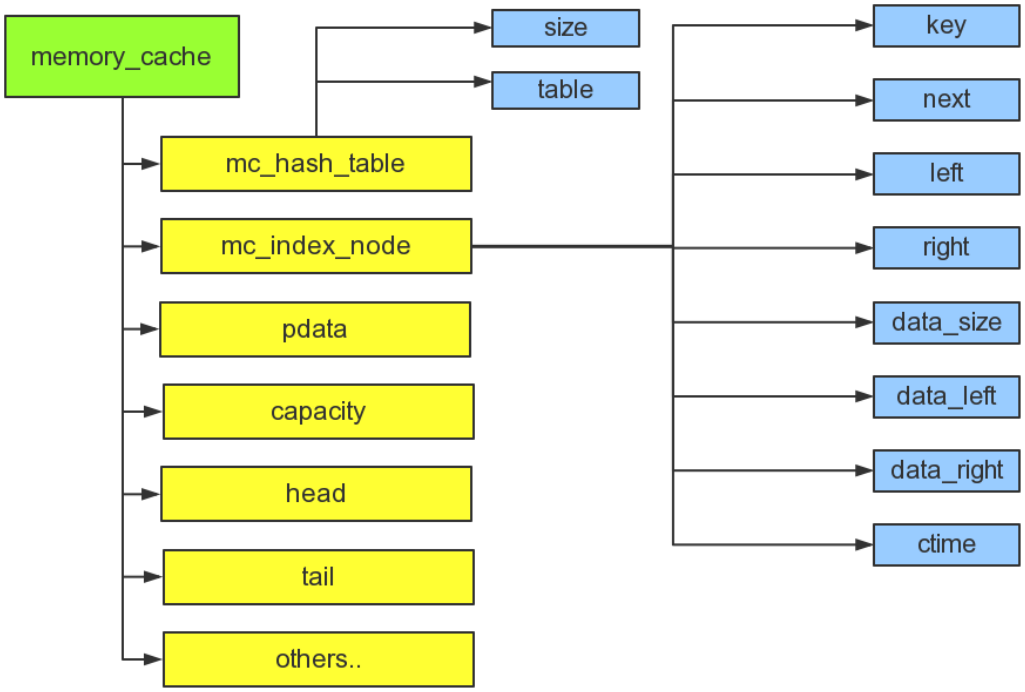

图-4 memory_cache 内存结构

2.2 地址空间分布

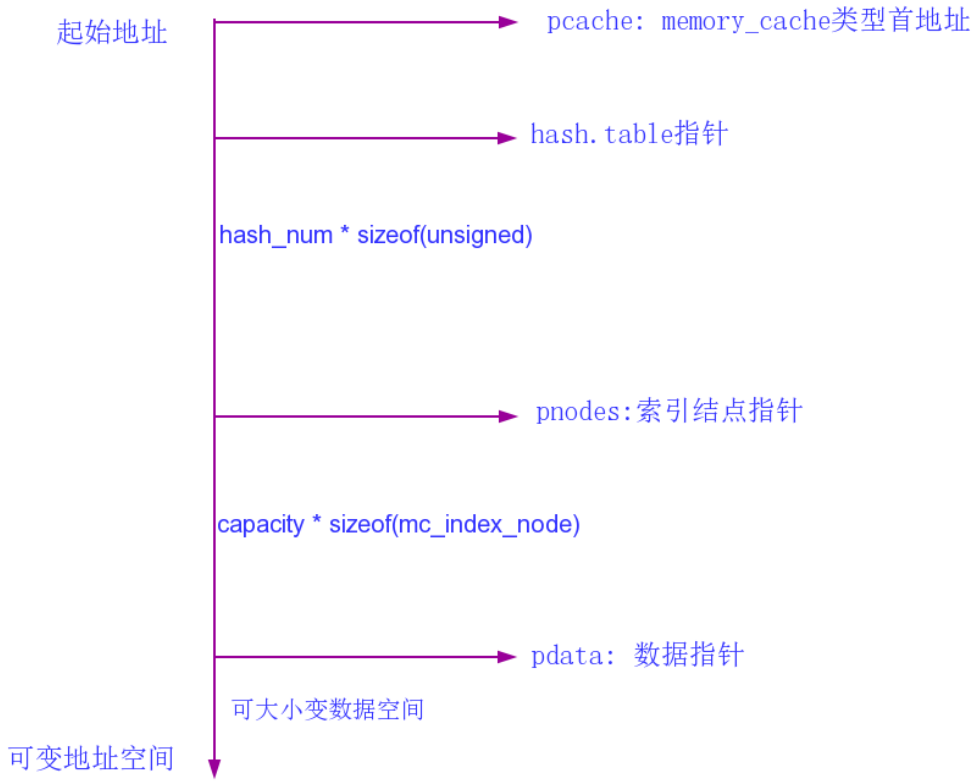

图-5 memory_cache 地址空间

三、逻辑设计

3.1 关注人服务的主要流程

实现类为 SocialRealtimeSearch，主要流程如下图：

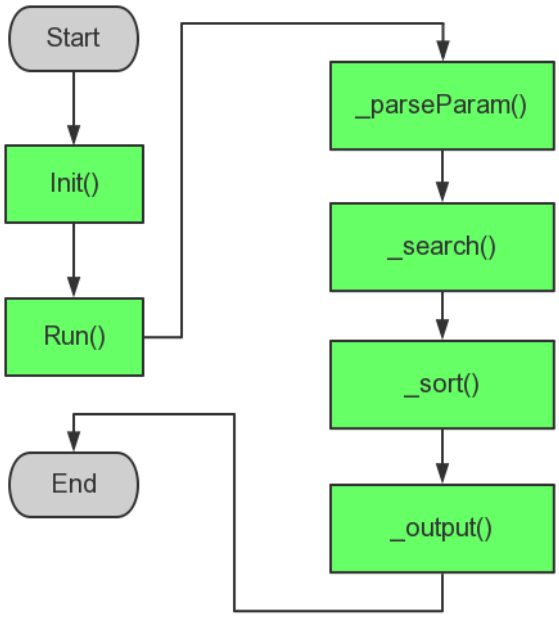

图-6 关注人服务主体流程

3.2 SocialRealtimeSearch 类

该类主要实现了关注人服务的整个控制逻辑，包括实时索引的空间初始化，索引的加载、查询参数解析、关注人转赞数据检索、结果排序、返回客户端查询等。

这个类的方法的驱动是由第一部分的 DA 框架的 Engine 类来驱动，分别调用实现了抽象类 BaseAlgorithms 的纯虚函数 Init()，DYN_ALGORITHMS，Run()函数来完成插件的工作。

3.2.1 插件初始化 | Init()

从 DA 加载的配置列表中加载实时索引插件的配置文件 rtindex.conf，根据该配置文件**调用 RealTimeRWIndex 类的 Load()方法**（见 3.3.1），申请对应的共享内存，获取 hashsize，超时时间等参数，，获取队列配置，关注人数组的初始化并加载社会化服务器的配置文件，获取相关的服务器路径。

3.2.2 Run 函数

在该函数中调用了_parseParam(), _search(), _sort(), _output() ，分别完成输入参数解析、关注人转赞数据检索、结果排序、返回结果给客户端。下面分开介绍各个函数的主要功能。

3.2.3 参数解析函数 | _parseParam()

分别完成客户端查询请求的参数解析。

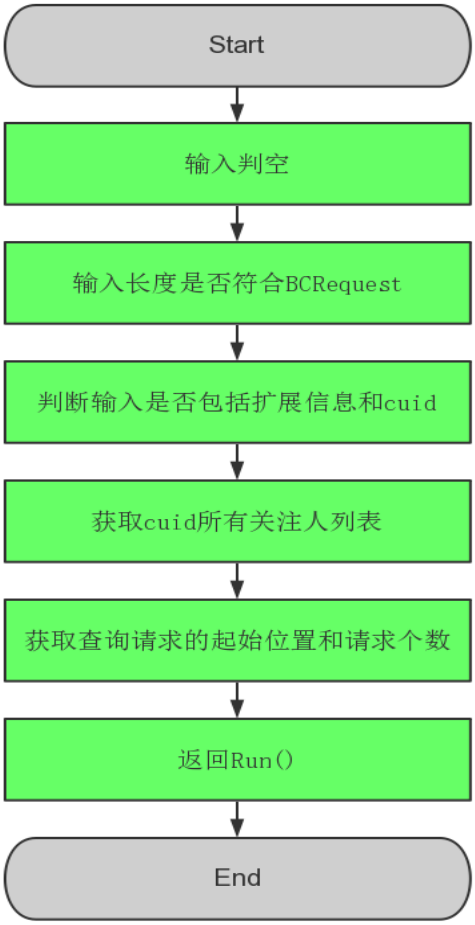

图-7 参数解析流程

3.2.4 实时索引内存检索 | _search()

对查询请求中 uid 的所有关注人的 uid **调用 RealTimeRWIndex 类的 Search() 方法**（见 3.3.3）去检索共享内存的转赞数据。对得到转赞数据进行解析，如果该转赞时间比结果集合中都要新，则以得到的 mid 为 key，用户动作为 value 添加到结果 Map 中。用到了如下数据结构：

一条 kv 结果记录：(mid, (user_action,uid,time))；

结果集 map 的定义为：map<uint64_t, user_action>；

存放检索结果最新的时间的队列为优先级队列：priority_queue<time_t, vector<time_t>, greater<time_t> >。

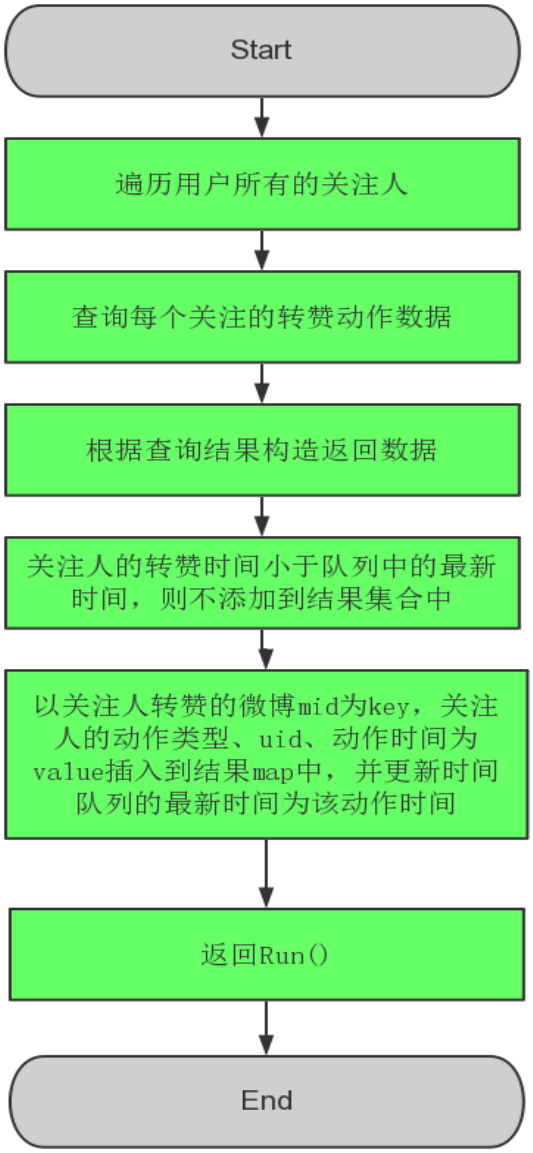

图-8 检索流程

3.2.5 结果排序 | _sort()

首先对检索结果 map 进行类型转换，将每条 Kv 数据转换为 SocialDocValue 类型，并将结果保存到 vector<SocialDocValue>中，然后调用 vector 的排序算法，按时间的新旧进行排序。

3.2.6 返回检索结果集 |_output()

将上步得到的已有序的结果复制到返回指针指向的缓存中。

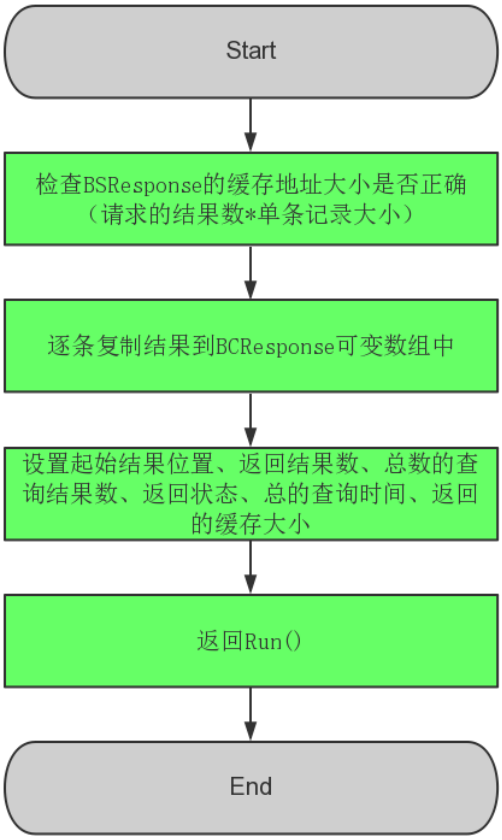

图-9 结果处理、返回流程

3.3 RealTimeRWIndex 类

该类具体实现了共享空间的申请、初始化、索引的生成，添加，删除、根据用户的关注人列表到共享内存的 hash 表里查询相应的实时索引，并返回最新的转赞数据。主要实现的功能如下：

3.3.1 实时索引空间加载 | Load()

在 SocialRealtimeSearch 类的 Init 函数中通过享元 Resource 的 getResource 方法调用 BaseResource 抽象类的 Load()方法，该 Load 方法是纯虚函数运行时加载子类的也就是 RealTimeRWIndex 的 Load 方法。这么设计的目的是统一了算法接口，使不同插件或模块遵循着相同的加载模式。
另外，最终资源加载到单例享元 Hashmap 中，即只加载一次，如果已加载则直接返回。该动作需要加锁来保证同步。
Load()的调用路径：

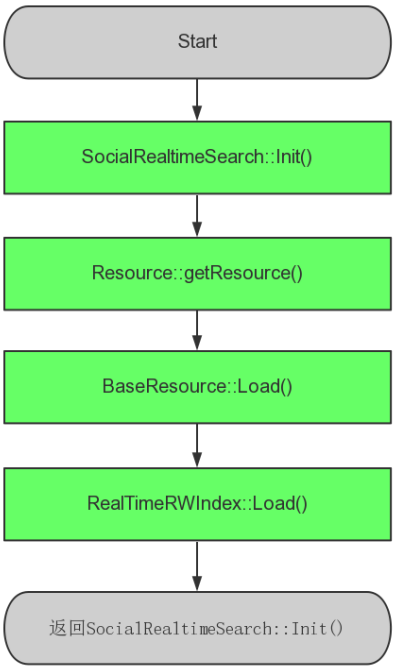

图-10 加载调用路径

Load()的执行逻辑：

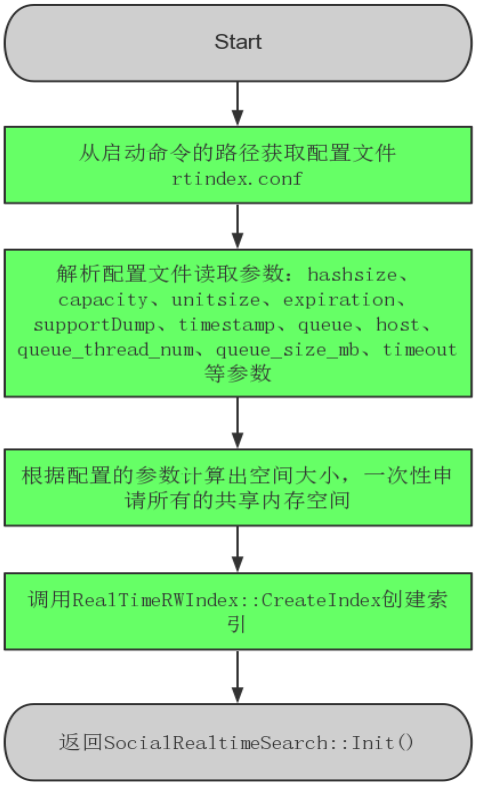

图-11 Load 执行逻辑

3.3.2 实时索引创建 | CreateIndex()

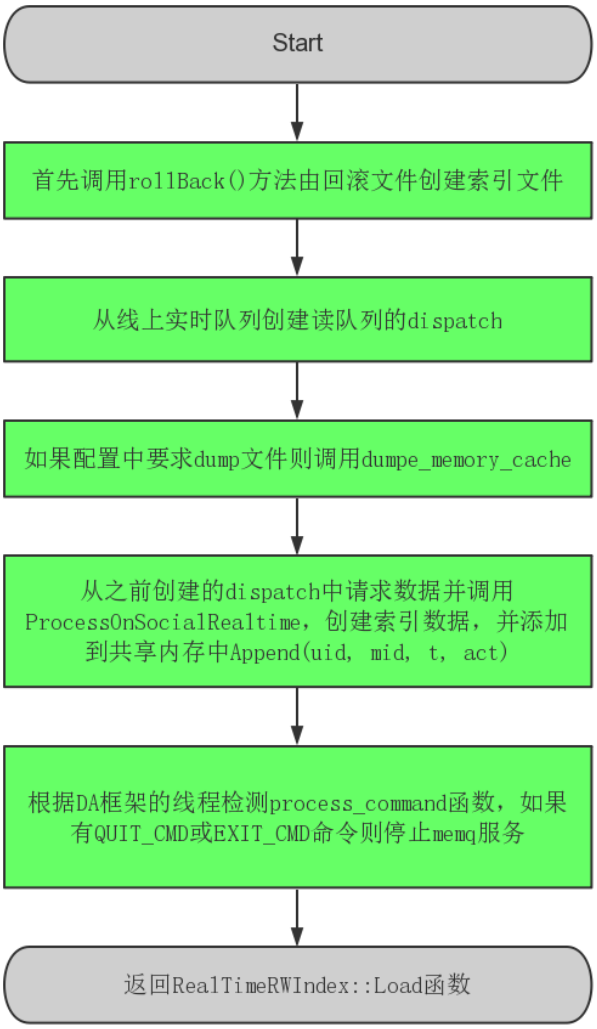

图-12 索引创建流程

3.3.3 实时索引检索 | Search()

以 uid 为 key 来检索共享内存中的转赞数据，结果保存在 pdata 中。可参考图 2。以下为关键函数：

seek_memory_cache_item(memory_cache *pcache, uint64_t key, void * pdata, size_t size, int * len)

其主要流程如下：

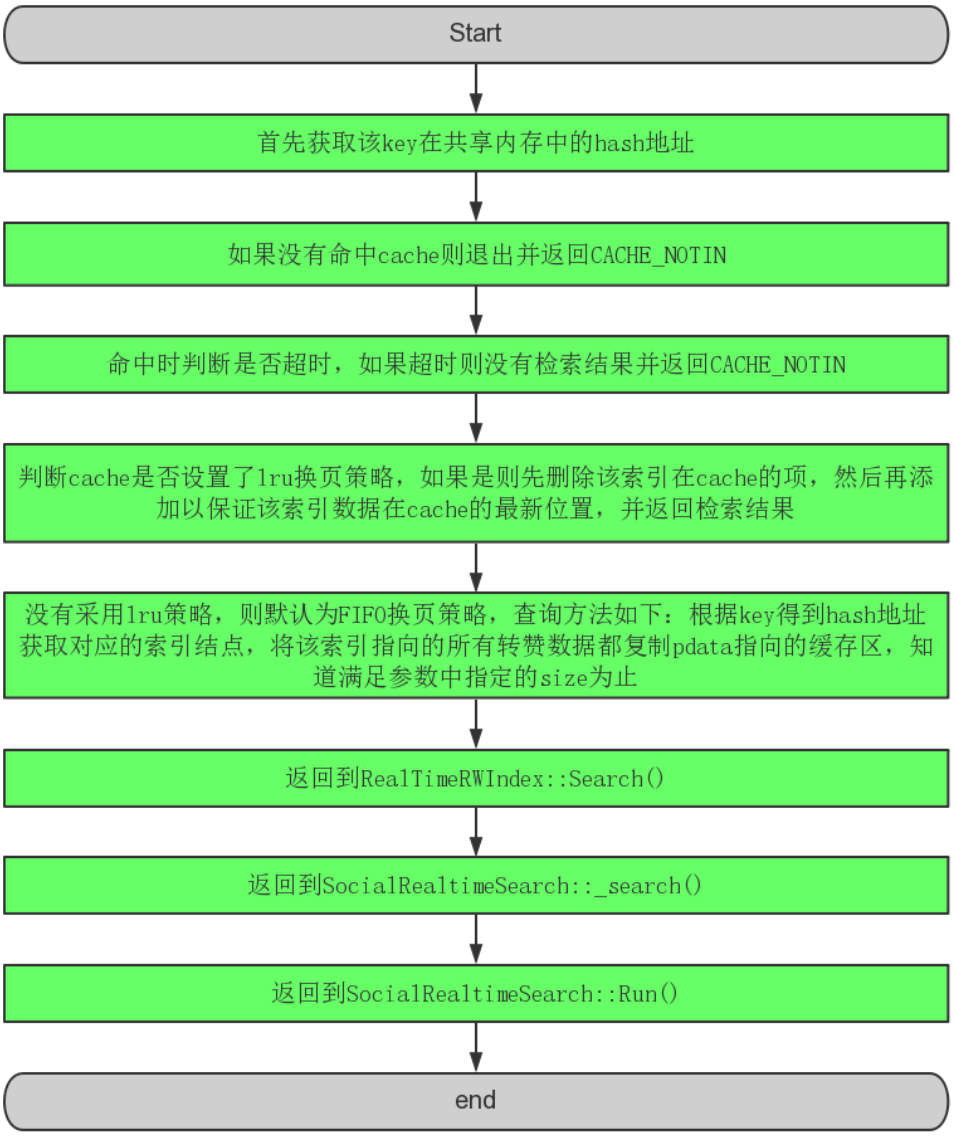
Start
首先获取该key在共享内存中的hash地址
如果没有命中cache则退出并返回CACHE_NOTIN
命中时判断是否超时，如果超时则没有检索结果并返回CACHE_NOTIN
判断cache是否设置了lru换页策略，如果是则先删除该索引在cache的项，然后再添加以保证该索引数据在cache的最新位置，并返回检索结果
没有采用lru策略，则默认为FIFO换页策略，查询方法如下：根据key得到hash地址获取对应的索引结点，将该索引指向的所有转赞数据都复制pdata指向的缓存区，知道满足参数中指定的size为止
返回到RealTimeRWIndex::Search()
返回到SocialRealtimeSearch::_search()
返回到SocialRealtimeSearch::Run()
end

图-13 检索流程