# WISE AC串讲

## 线程模型

1. 主线程接收US请求，放入ependingpool
2. 多个工作线程worker并发处理查询请求
3. detect线程定期检测不可用后端，发现可用即标识出来，servive\_detect检查心跳机制及恢复后端，service\_detect\_ext检测慢后端
4. Reload线程负责配置和词典动态更新

## 工作线程

创建 ac.cpp 5123

关键代码：

ret = pthread\_create(&tid, NULL, ACCore, (void \*)thread\_id);

ACCore 3846

主循环： 4307

一些初始化工作

### 获取查询请求 4598

gettimeofday(&ts, NULL);

ret = ac\_get\_ui\_request\_data(&fdui, prequest\_data, pdyn);

gettimeofday(&te, NULL);

pTimeData->readquery\_time += (te.tv\_sec - ts.tv\_sec)\*USECS\_PER\_SEC

+ (te.tv\_usec - ts.tv\_usec);

### 插入点： US\_TO\_AC

4646

//需求管理框架 在解析us请求包时的干涉点

//注意：对us请求解析的干预，没有放在原始解析处理函数中的原因是，要等到queryInit了QueryStrategyManager才能调用需求管理器的对应接口

us\_req\_obj.pUsData = pUsData;

sub\_state.interact\_state = US\_TO\_AC;

g\_queryStrategyManager->process(ModuleInteract, sub\_state, &us\_req\_obj);

4662

### ACSearch() 2925

4726

//返回查询结果 , 在此加入资源爆发的附加数据 linkfoundtime

ret = ac\_send\_result\_to\_ui(fdui, search\_result\_num, presult\_data\_out, prequest\_data, pdyn);

#### 3016 多query查询 multiple\_search 1924

##### 2405 注册挖掘队列

if (pUsData->local\_db\_flag > 0 && pUsData->city\_code > 0 && pUsData->midpage\_local\_flag == 0) {

//注册wise-diyu队列

g\_DataManager->reg\_queue(STR(InfernoStrategy), NULL);

}

##### 插入点： AC\_TO\_DA

也是moduleInteract的形式。

setIsReq内部先判断是req 还是 res

然后循环调用事先注册好的所有策略的相关方法。

Req就调用add\_module\_data

Res就调用parse\_module\_data

bool is\_req = false;

//判断是request还是response

setIsReq(sub\_state.interact\_state, is\_req);

//循环调用所有策略的对应接口

for(int si=0; si<\_sorted\_str\_num; si++)

{

if(true == is\_req)

{

\_sorted\_str\_list[si].pStrategy->add\_module\_data(pObj, sub\_state.interact\_state);

continue;

}

\_sorted\_str\_list[si].pStrategy->parse\_module\_data(pObj, sub\_state.interact\_state);

}

这个插入点被判断为req，调用add\_module\_data

void LocalSearchStrategy::add\_module\_data(ModuleObj\* pModuleObj, InteractState iState) {

if (pModuleObj == NULL) {

ul\_writelog(UL\_LOG\_WARNING, "[LS\_dy] pModuleObj is error in %s", \_\_PRETTY\_FUNCTION\_\_);

return;

}

switch (iState) {

case AC\_TO\_DA:

get\_query\_info\_from\_da(pModuleObj);

break;

case AC\_TO\_US:

add\_dyjh\_result\_to\_us(pModuleObj);

break;

case AC\_TO\_DISP:

add\_extra\_dyjh\_url\_to\_disp(pModuleObj);

break;

default:

break;

}

return;

}

get\_query\_info\_from\_da这个函数就是通过获取US请求DA的结果，获得DA对query的需求识别以及需求强度。在这个函数中置了几个标志位。

##### 2415 第一次统一调权

Frame/QueryManager/QueryStrategyManager.cpp

//调用需求管理框架，隐藏不需要LTR统一调权的队列

sub\_state.uniadjust\_state = UNIADJUST\_1ST;

g\_queryStrategyManager->process(Uniadjust, sub\_state);

##### 2429 第一次统一调权结束

//调用需求管理框架，恢复被隐藏的队列

sub\_state.uniadjust\_state = UNIADJUST\_1ST\_END;

g\_queryStrategyManager->process(Uniadjust, sub\_state);

##### 2443 从normal队列中挖掘出inferno队列

if (pUsData->local\_db\_flag > 0 && pUsData->city\_code > 0 && pUsData->midpage\_local\_flag == 0) {

g\_StatManager->process(STR(InfernoStrategy));

}

StatManager

第一步，通过数据总控以qid拿到inferno队列

ResultQueue\_t \*queue;

if(\_data\_manager->get\_queue(qid, &queue) != DM\_SUCC)

{

WARNING\_LOG( "get dm queue fail");

return ac::FAIL;

}

第二步，先从\_strategy\_table中取出strategy和然后取出inferno queue <null>，统计多队列。

<\_strategy\_table注册策略是在factory\_onlinestatistic.cpp <OnlineStatistic> 的构造函数中 build\_statstrategies函数 160行注册了InfernoStrategy这个策略，先放到数组，后面调用register\_strategies函数完成注册 StatManager::register\_strategies>

for(int sidx = 0; sidx < (int)STAT\_MAXSTRATEGY; ++sidx)

{

StatStrategy \*strategy = \_strategy\_table[qindex][sidx];

if (strategy == NULL)

{

// \_strategy\_table[(int)qindex]是紧凑排列，

// 一旦出现null，则后续肯定全都是null

break;

}

//这里的sid都是合法的

int sid = \_find\_sinnerid(strategy->get\_sid());

//由依赖此策略的其他策略自己去判断依赖的数据是否存在,框架简化处理

if(\_isopen[sid] != S\_OPEN)

{

continue;

}

int qnum\_start = multi\_queue->GetQueueNum();

//返回值暂时不管,出错信息由策略自己去记录

if(queue != NULL) //统计单队列

{

ResultQueue\_t \*result\_queue = multi\_queue->GetNextQueue();

//如果分配失败，系统已经异常，不用再统计了

if(result\_queue == NULL)

{

WARNING\_LOG("multi\_queue->GetNextQueue() return NULL");

return AS\_FAIL;

}

result\_queue->strategys = queue->strategys;

result\_queue->query\_type = queue->query\_type;

strategy->statistic(queue,result\_queue, qid);

}

else//统计多队列

{

strategy->statistic(multi\_queue, qid);

}

int qnum\_end = multi\_queue->GetQueueNum();

//1. 策略很可能不会自己去recycle队列，而队列完全可能没有结果

//这里保证回收空队列，节省系统资源

//2. 如果策略自己回收了队列， 这里不会重复回收

for(int qindex = qnum\_end - 1; qindex >= qnum\_start; --qindex)

{

ResultQueue\_t \*queue = multi\_queue->GetQueue(qindex);

if(queue->merged\_data.real\_res <=0)

{

multi\_queue->RecycleQueue(queue);

}

}

}

所以，这里会调用InfernoStrategy <OnlineStatistic/default/>的statistic函数，进而调用了getResultFromNormal函数，完成了从normal队列中挖掘Inferno队列。

int InfernoStrategy::getResultFromNormal(MultiQueue\_t\* multi\_queue) {

ResultQueue\_t\* owner\_queue = multi\_queue->GetNextQueue();

if (owner\_queue == NULL) {

ul\_writelog(UL\_LOG\_WARNING, "%s get empty queue", \_\_FUNCTION\_\_);

return ac::FAIL;

}

ResultQueue\_t \* normal\_queue = multi\_queue->GetQueue(Query\_Normal);

if (normal\_queue == NULL) {

ul\_writelog(UL\_LOG\_WARNING, "%s get normal queue empty", \_\_FUNCTION\_\_);

return ac::FAIL;

}

ac\_merged\_data\_t\* merged\_data = &normal\_queue->merged\_data;

ac\_merged\_data\_t\* owner\_data = &owner\_queue->merged\_data;

UsData \* us\_data = NULL;

GET\_DATA(UsData, us\_data);

if (us\_data == NULL) {

WARNING\_LOG("[InfernoStrategy get usdata failed]");

return ac::SUCC;

}

for (int i = 0; i < merged\_data->real\_res; ++i) {

//infernoµÄ½á¹ûdb\_nameÊÇwise-inferno

if (merged\_data->bs\_out[i].city\_code == us\_data->city\_code && us\_data->city\_code > 0) {

owner\_data->bs\_out[owner\_data->real\_res] = merged\_data->bs\_out[i];

++owner\_data->real\_res;

}

}

SetQueueType(owner\_queue, Query\_Inferno, Search\_Normal, NULL);

merge\_info\_t merge\_info;

memset(&merge\_info, 0, sizeof(merge\_info\_t));

SetInsertMergeInfo(&merge\_info, NULL, NO\_FIX\_MERGE\_POS, 5,

1000, MERGE\_NO\_REMIT, KEEP\_ALL\_CLUSTER, 1, NO\_CONTROL\_THRESHOLD, NULL,

NULL, Priority\_Normal, 1);

merge\_info.is\_comppk = 1;

SetQueueMergeInfo(owner\_queue, &merge\_info);

owner\_queue->pos\_state = BEFORE\_RANKING;

owner\_queue->queue\_id.SetQueueName(STR(InfernoStrategy));

return ac::SUCC;

}

##### 2449 QS\_1ST\_RANKING

sub\_state.q\_state = QS\_1ST\_RANKING;

g\_queryStrategyManager->process(OnlineStatistic, sub\_state);

对挖掘队列通过过dict的方式过滤掉一些url。

##### 2460 get\_dx

g\_Ac2DxFrameAttrStat->get\_dx(g\_DataManager, pfd\_dx, prequest\_data->multiple\_queue, hit\_result\_cache, result\_cache\_item, result\_cache\_key);

##### 2474 QS\_1ST\_DX

sub\_state.q\_state = QS\_1ST\_DX;

g\_queryStrategyManager->process(OnlineStatistic, sub\_state);

##### 2548 第二次点击调权

##### 2559 QS\_2ND\_RANKING

地域策略这里主要进行rerank和设置了参与pk的参数信息。

##### 2693 插入点MULTI\_SEARCH

地域策略主要先从final queue中过滤掉地域结果，同时记录了被过滤掉结果在final queue中的位置。

根据结果的在final queue中的位置分布，比如，记录了前5条结果个数，前3条结果个数等，外加之前获取的US请求的DA的需求识别强度，来决定最后地域聚合结果的插入位置。

##### 取disp

###### AC\_TO\_DISP

Add\_extra\_dyjh\_url\_to\_disp函数中关键代码：

g\_pDealManager->Process("Ls\_Find\_Dx", &p\_Obj);

之前在threadInit的时候，调用了

LocalResultDxFindDeal\* p\_ls\_dx\_find\_deal = new(std::nothrow) LocalResultDxFindDeal(this);

。。。

g\_pDealManager->AddDeal("Ls\_Find\_Dx", p\_ls\_dx\_find\_deal);

再分析一下DealManager的process的逻辑，发现，调用的其实就是LocalResultDxFindDeal的process\_deal

\_m\_ls\_res = \_m\_ls\_strategy->ls\_get\_res\_for\_dx(&\_m\_ls\_num);

DispDxFindDealBase::process\_deal(pO);

其中又调用了子类实现的select\_url\_to\_di\_queue，把相关的地域挖掘结果加入到了取disp的url中。

纯虚函数

###### DISP\_TO\_AC

ModuleInteract

Parse\_local\_result\_after\_disp 获取到disp结果，然后做了自己解包获取了一些用到的信息，然后重新打包，最后保存到要返回给us的结构中。

##### 最后AC\_TO\_US