# 自己动手写Impala UDF

## 概述

出于对可扩展性和性能的考虑，UDF已变成大数据生态圈查询引擎的必备功能之一，无论是Calcite、Hive、Impala都对其进行支持，但是UDF的支持有利也有弊，好处在于它提供了对某些用户独有需求的支持，例如某些产品需要将表中的某字段使用自定义的方式解析成可读字段，例如需要实现特殊的聚合函数；它的弊端在于它对用户开发，这样对于恶意的用户可能执行非正常的逻辑，例如在函数中删除或者拷贝其它文件内容，从而对非授权数据造成破坏，因此对于一个SQL引擎来说，我们需要UDF的集中管理，所有用户自定义的UDF都需要管理员审查源代码，不允许普通用户自己上传UDF，从而避免意外的发生。

对于通常UDF的需求，个人觉得有两方面的需求：1、系统提供的函数完成不了的需求，或者需要使用系统函数进行拼凑才能完成的需求。2、使用当前系统提供的函数性能太差，需要做一些特别的优化。另外，对于UDF还分为两类：自定义函数（UDF）和自定义聚合函数（UDAF），前者会处理每一条输入的记录，转换成处理后的结果，类似于map的功能，后者对于多条记录进行聚合，输出聚合之后的值，类似于reduce的功能。

众所周知Impala使用了Java和C++实现（虽然大多数时候我们都说Impala是C++实现的，所以性能更好，但是它的SQL解析部分的确是Java实现的），Impala同样也支持两种语言的UDF，但是UDAF目前只能支持C++实现，本文分别介绍这些方法如何在Impala中使用的。

## 目标

我们这里的example实现一个UDF和一个UDAF，分别实现如下的需求： 我们需要实现的UDF为int level(int)，功能为根据value的值计算出距离该值最近的2的幂数的幂值，如果有多个值则取最大的。例如15，距离它最近的2的幂数为16，则level(15)=4，level(9)=3, level(12)=3或4则level(12)=4. 我们需要实现的UDAF为int sum_str(string)，功能为计算name中出现的第一个整数的和值，如果该字符串不出现整数则为0，例如"abcd123ef"和"efdg23sd24"的和值为123+23=146. 这两个需求算是比较奇葩了吧，看看如何在impala中利用UDF和UDAF实现它们。

## 实现

### Java版UDF

了解Impala的都知道，Impala可以直接使用Hive的metestore作为元数据库，同样Impala也可以直接使用Hive的UDF，所以对于之前奋斗在Hive第一线的同学们使用Impala有了不少亲切感，那么在这里就顺便温习一遍Hive的UDF使用流程吧。

使用Java实现的UDF必须继承org.apache.hadoop.hive.ql.exec.UDF类，然后在该类中实现名为evaluate的函数，猜测Hive/Impala是根据函数名来找到具体的实现的，当然一个类里面也可以重载多个evaluate方法。该方法实现如下：package com.netease.hive.udf;

```
import org.apache.hadoop.hive.ql.exec.UDF;

public class LevelUDF extends UDF {
    public Integer evaluate(Integer value) {
        if(value == null)
            return null;

        double temp = value;
        int cnt = 0;
        int max = 1;
        while(temp > 1) {
            cnt ++;
            temp /= 2;
            max *= 2;
        }
        if(max - value > (value - max / 2))
            cnt --;

        return cnt;
    }

    public static void main(String[] args) {
          System.out.println(new LevelUDF().evaluate(9));
    }
}
```

然后编译成jar上传到HDFS上，Impala的自定义函数，无论是Java实现还是C++实现的都需要手动放到HDFS上，而非像Hive那样直接可以add jar命令，然后在impala shell中执行create function函数。

```
> hadoop fs -put ./udf.jar hdfs://namenode-or-nameservice/tmp/nrpt/
> create function level(int) returns int location 'hdfs://namenode-or-nameservice/tmp/nrpt/udf.jar' symbol='com.netease.hive.udf.L
```

在impala中创建UDF必须指定参数和返回值，并且执行symbol为类名。根据参数和返回值在运行时查找该函数，如果参数和返回值和类中实现有差别则会出现运行时错误（create function还是能够成功的）。

```
> select level(15);
+-------------------+
| default.level(15) |
+-------------------+
| 4                 |
+-------------------+
> select level(9);
+------------------+
| default.level(9) |
+------------------+
| 3                |
+------------------+
```

使用Java实现UDF是非常简单的，那么为什么还需要使用C++的实现呢，主要是性能的考虑，毕竟相同的逻辑使用C++实现无论是性能还是资源消耗都会比JAVA好得多，所以Impala官方是非常推崇使用C++实现UDF，并且声称性能会有10倍的提升。

## C++实现UDF

使用C++实现UDF就不那么轻松了，首先要面临的就是系统库的差别，所以一般要求UDF开发机和Impala运行的机器使用相同的Linux发行版，最好在其中的一台Impalad机器上开发，避免出现不可预测的问题，C++开发UDF需要首先下载impala udf devel开发包和一下其他依赖的工具：

```
sudo yum install gcc-c++ cmake boost-devel
sudo yum install impala-udf-devel
```

g++、cmake工具一般开发机上都是有的，boost开发包需要手动安装，当然如果你的函数实现不需要boost也不需要安装这个，不过Impala本身都是使用boost开发的，不过使用boost库可以提高开发效率和性能，impala-udf-devel这个包在安装的时候发现ubuntu和debain上根本找不到，不过不用担心，其实安装这些包主要就是把这个包下面的.h文件放到系统的include目录下，把.so文件放到系统的lib下，只要有源码自己也可以编译。可以在https://github.com/laserson/impala-udf-devel下载它们的源码（就两个.h文件和一个.cc文件）。对于UDF开发更推荐这种方式，省的自己写CMakeList.txt文件了，clone到本地之后可以直接运行cmake生成Makefile文件。

```
> git clone https://github.com/laserson/impala-udf-devel.git
> cd impala-udf-devel/
> cmake .
```

此时会发现出现了错误：

```
CMake Error at CMakeLists.txt:46 (add_library):
  Cannot find source file:

  my-udf-file-1.cc
```

打开CMakeList.txt文件会发现add_library(myudf SHARED my-udf-file-1.cc my-udf-file-2.cc)这一行，它表示根据my-udf-file-1.cc和my-udf-file-2.cc文件生成一个动态链接库myudf，我们可以通过该一下这个配置然后编写自己的udf文件，我们把CMakeList.txt文件的最后几行改为如下：

```
# Build the UDA/UDFs into a shared library.  You can have multiple UDFs per
# file, and/or specify multiple files here.
add_library(level-udf SHARED level-udf.cc)

# The resulting LLVM IR module will have the same name as the .cc file
if (CLANG_EXECUTABLE)
  COMPILE_TO_IR(level-udf.cc)
endif(CLANG_EXECUTABLE)
```

然后编辑代码level-udf.h(创建）：

```
#ifndef LEVEL_UDF_H
#define LEVEL_UDF_H

#include "udf/udf.h"

using namespace impala_udf;

IntVal level(FunctionContext* context, const IntVal& value);

#endif
```

编辑level-udf.cc文件（创建）：

```
#include "level-udf.h"

using namespace std;

IntVal level(FunctionContext* context, const IntVal& value) {
    if(value.is_null)
        return IntVal::null();

    int original = value.val;
    double temp = (double) original;
    int cnt = 0;
    int max = 1;
    while(temp > 1) {
        cnt ++;
        temp /= 2;
        max *= 2;
    }

    if((max - original) > (original - max / 2))
        cnt --;
    return IntVal(cnt);
}
```

这里需要说明一下的是FunctionContext对象，这个对象是调用UDF函数之前由Impala自己创建的，它包含当前查询的id、查询执行的用户名、使用该对象分配内存，并且可以向Impala日志系统输出warning日志，或者直接输出一条error日志以结束该查询。而每一个函数的输入和输出是IntVal类型，这个是在udf.h中定义的一个类型，它实际上就是包含null的int类型，除了该类型之外还有AnyVal; BooleanVal; TinyIntVal; SmallIntVal; IntVal; BigIntVal; StringVal; TimestampVal这几种类型，和Impala中的类型一一对应，其中AnyVal是其他类型的父类，它可以标识该对象是否null，其他的基本类型可以通过val成员获取原始值（在不是null的情况下），StringVal可以通过ptr和len获取首地址和长度，TimestampVal则可以获取date和time*of*date分别获取日期和纳秒数（这样的话TimestampVal就可以包含Date了，况且Impala还不支持Date类型）。

这里面没有使用boost库（大多数情况下是需要使用的），然后编译，链接生成动态链接库：

```
> cmake .
> make
```

执行完成之后在当前目录下产生了build目录（类似于maven构建之后的target目录），该目录下存在链接完成的liblevel-udf.so文件，库名为之前在CMakeList.txt文件中修改之后的名字。

然后按照之前的方式首先上传动态链接库到HDFS，然后create function。

```
> hadoop fs -put ./liblevel-udf.so hdfs://namenode-or-nameservice/tmp/nrpt/
> create function level_c(INT) returns int location 'hdfs://namenode-or-nameservice/tmp/nrpt/liblevel-udf.so' symbol='level';
```

使用C++函数时symbol执行函数名，通过该函数名可以在liblevel-udf.so中找到该函数。

```
> select level_c(12);
+---------------------+
| default.level_c(12) |
+---------------------+
```

```
| 4                  |
+--------------------+
> select level_c(9);
+--------------------+
| default.level_c(9) |
+--------------------+
| 3                  |
+--------------------+
```

至于C++实现和Java实现的性能对比，可以在实际的场景下使用不同的语言实现相同的逻辑，只要你的C++代码写的不是太挫，那么至少会有两倍以上的性能差别的，这里也推荐使用C++实现UDF，不过一定要注意内存泄漏、段错误等情况，如果实现不当可能到impalad挂掉。对于C++实现UDF的详细说明可以参考Impala官方文档：http://www.cloudera.com/documentation/archive/impala/2-x/2-1-x/topics/impala_udf.html，更多的C++实现的UDF实例可以参考https://github.com/cloudera/impala-udf-samples。

## C++实现UDAF

实现一个聚合函数需要不像简简单单的实现一个UDF一样，毕竟需要初始化环境，对于每一个输入记录进行聚合，这就要求它在整个执行过程中保持一个状态，例如计算SUM时需要保持一个sum变量记录已经处理的记录的和，这样的方式和reducer有所差别，它是通过对输入的值循环调用aggr函数，而reducer则是将这个列表作为输入处理。在Impala中实现一个UDAF需要实现下面这些函数：

- INIT_FN：初始化操作，在查询执行时执行一次，例如清理计数器，分配缓存等。例如SUM时sum=0。
- UPDATE_FN：在单机上执行的merge操作，对于每一个节点的每一条记录调用一次该函数，例如SUM时执行sum+= value。
- MERGE*FN：节点之间的merge，通常逻辑上和UPDATE*FN操作类似，例如SUM时执行sum+=value。
- SERIALIZE_FN：对于INIT、UPDATE、MERGE阶段的结果进行序列化，例如需要对包含指针的值使用它所执行的内容序列化，然后释放该指针。一般情况下不需要设置该函数。
- FINALIZE_FN：最终获取结果的 函数，只调用一次。

我们这里需要实现的是统计某一个string列中每一个成员第一次出现的整数的和，我们定义为int sum_first_int(string)，这里不牵扯到指针的序列化，所以不需要实现SERIALIZE_FN函数。

该需求的实现sum-udaf.h(新建）：

```
#ifndef _SUM_UDAF_H_
#define _SUM_UDAF_H_

#include "udf/udf.h"

using namespace impala_udf;

void init_func(FunctionContext* context, BigIntVal* result);

void update_func(FunctionContext* context, const StringVal& input, BigIntVal* result);

void merge_func(FunctionContext* context, const StringVal& input, BigIntVal* result);

BigIntVal finalize_func(FunctionContext* context, const BigIntVal& val);

#endif
```

实现了四个函数的定义，其中init*func函数传递了一个BigIntVal函数，它是由函数的返回值决定的，result参数作为保存聚合结果，update*func函数对于每一个输入的记录进行处理，input参数为该记录中该列的值，merge*func对于不同节点的聚合结果进行再聚合，finalize*func则是返回最终结果。可以看出每一个聚合组的状态信息保存在result参数中。

实现文件sum-udaf.cc（新建）：

```
#include "sum-udaf.h"
#include <ctype.h>

void init_func(FunctionContext* context, BigIntVal* result) {
    result->is_null = false;
    result->val = 0;
}

void update_func(FunctionContext* context, const StringVal& input, BigIntVal* result) {
```

```
    if(input.is_null) return;
    int cur = 0;
    uint8_t *ptr = input.ptr;
    int len = input.len;
    int i = 0;
    for(i = 0 ; i < len && !isdigit(ptr[i]); ++ i);
    while(i < len && isdigit(ptr[i])) {
        cur = cur * 10 + (ptr[i] - '0');
        ++ i;
    }
    result->val += cur;
}

void merge_func(FunctionContext* context, const BigIntVal& input, BigIntVal* result) {
    result->val += input.val;
}

BigIntVal finalize_func(FunctionContext* context, const BigIntVal& val) {
    return val;
}
```

完成之后再修改CMakeList.txt文件，增加对sum-udaf.cc的编译和链接，然后执行cmake和make（同上），就可以生成新的.so链接库了。使用相同的方式将该库上传到HDFS然后再impala shell上创建聚合函数。

```
> create aggregate function sum_first_int(string) returns bigint
> location 'hdfs://namenode-or-nameservice/tmp/nrpt/liblevel-udf.so' init_fn='init_func' update_fn='update_func' merge_fn='merge_f
```

此时可以使用该聚合函数了，如下：

```
> CREATE TABLE default.test (id string);
> insert into test values("abc1234edf"), ("12shd45"), ("dhhf"), ("qwe3"), ("2016-09-01");
> select sum_first_int(id) from test;
+---------------------------+
| default.sum_first_int(id) |
+---------------------------+
| 3265                      |
+---------------------------+
```

# 总结

本文详细讲述了在Impala中实现自定义函数和自定义聚合函数的方式并给出实现实例以供参考，使用UDF和UDAF我们可以很轻易的实现特定需求的计算逻辑，也可以用性能更好的方式实现一些相对固定的需求，例如我们可以使用UDAF函数实现更好性能的distinct count计算（利用hyoerloglog实现有误差的去重计数），但是开放的接口也可能存在一些安全性上的问题，给系统运维的也带来了一定的负担，所以对于自定义函数的引入也需要谨慎。