UDAF(Hive)问题总结

一、 简要说明

UDAF 是 Hive 中用户自定义的聚集函数，UDAF 实现有简单（UDAF）与通用（AbstractGenericUDAFResolver）两种方式，简单 UDAF 因为使用 Java 反射导致性能损失，而且有些特性不能使用，**已被弃用。**

提示：初期开发，采用简单 UDAF 语法实现逻辑功能，出现不能成功运行、批量补数据出现错误等问题，导致无法准确、有效定位具体问题，期间自定义函数经历至少 3 个版本。后续改写为通用接口（AbstractGenericUDAFResolver），约束各阶段输入、输出数据格式，问题得以解决。

说明：如果知晓如上问题点，那么实现 UDAF 通用接口自定义函数不难实现。但如果未发现上述问题点，一直围绕简单 UDAF 接口中的数据结构等问题进行调整，将很难彻底解决异常问题。

Wiki:

Writing GenericUDAFs: A Tutorial

User-Defined Aggregation Functions (UDAFs) are an excellent way to integrate advanced data-processing into Hive. Hive allows two varieties of UDAFs: simple and generic. Simple UDAFs, as the name implies, are rather simple to write, but incur performance penalties because of the use of Java Reflection, and do not allow features such as variable-length argument lists. Generic UDAFs allow all these features, but are perhaps not quite as intuitive to write as Simple UDAFs.

This tutorial walks through the development of the `histogram()` UDAF, which computes a histogram with a fixed, user-specified number of bins, using a constant amount of memory and time linear in the input size. It demonstrates a number of features of Generic UDAFs, such as a complex return type (an array of structures), and type checking on the input. The assumption is that the reader wants to write a UDAF for eventual submission to the Hive open-source project, so steps such as modifying the function registry in Hive and writing `.q` tests are also included. If you just want to write a UDAF, debug and deploy locally, see this page.

二、 开发步骤

开发通用 UDAF 包含以下 2 个步骤：

1） 编写 resolver 类

2） 编写 evaluator 类

resolver 负责类型检查，操作符重载。evaluator 真正实现 UDAF 的逻辑。通常顶层 UDAF 类继承 org.apache.hadoop.hive.ql.udf.GenericUDAFResolver2，里面编写嵌套类 evaluator 实现 UDAF 的逻辑。

但强烈建议继承 AbstractGenericUDAFResolver，隔离将来 hive 接口的变

化。GenericUDAFResolver 和 GenericUDAFResolver2 接口的区别是，后面的允许 evaluator 实现可以访问更多的信息，例如 DISTINCT 限定符，通配符 FUNCTION(*)。

三、 UDAF 代码框架示例

```
缩进 减少缩进 注释 格式化
public class LastValueUDAF extends AbstractGenericUDAFResolver {

    static final Log LOG = LogFactory.getLog(LastValueUDAF.class.getName());

    @Override
        public GenericUDAFEvaluator getEvaluator(TypeInfo[] parameters) throws
SemanticException {
        // Type-checking goes here!
        return new LastValueEvaluator();
    }

    public static class LastValueEvaluator extends GenericUDAFEvaluator {
        // UDAF logic goes here!
    }
}
```

四、 GenericUDAFEvaluator

UDAF 逻辑处理主要发生在 Evaluator 中，需要实现该抽象类的几个方法。

1) ObjectInspector 接口

主要用于解耦数据使用与数据格式,使得数据流在输入输出端切换不同的输入输出格式，不同的 Operator 上使用不同的格式。

2) GenericUDAFEvaluator 内部类 Model

代表 UDAF 在 mapreduce 的各个阶段。

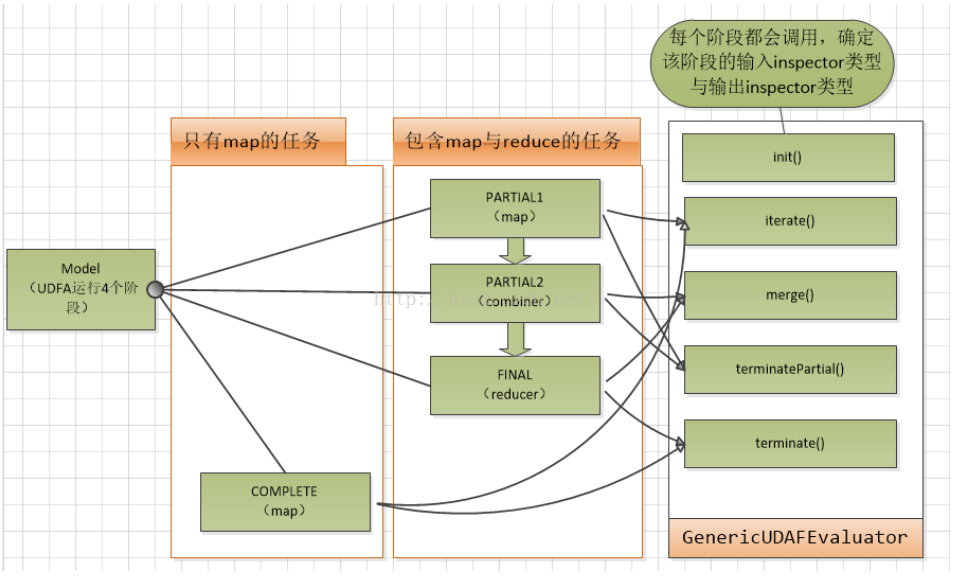
```java
public static enum Mode {
    /**
     * PARTIAL1: map阶段:从原始数据到部分数据聚合
     * 将会调用iterate()和terminatePartial()
     */
    PARTIAL1,
    /**
     * PARTIAL2: map端的Combiner阶段，负责在map端合并map的数据::从部分数据聚合到部分数据聚合:
     * 将会调用merge() 和 terminatePartial()
     */
    PARTIAL2,
    /**
     * FINAL: mapreduce的reduce阶段:从部分数据的聚合到完全聚合
     * 将会调用merge()和terminate()
     */
    FINAL,
    /**
     * COMPLETE:从原始数据直接到完全聚合,将会调用 iterate()和terminate()
     */
    COMPLETE
};
```

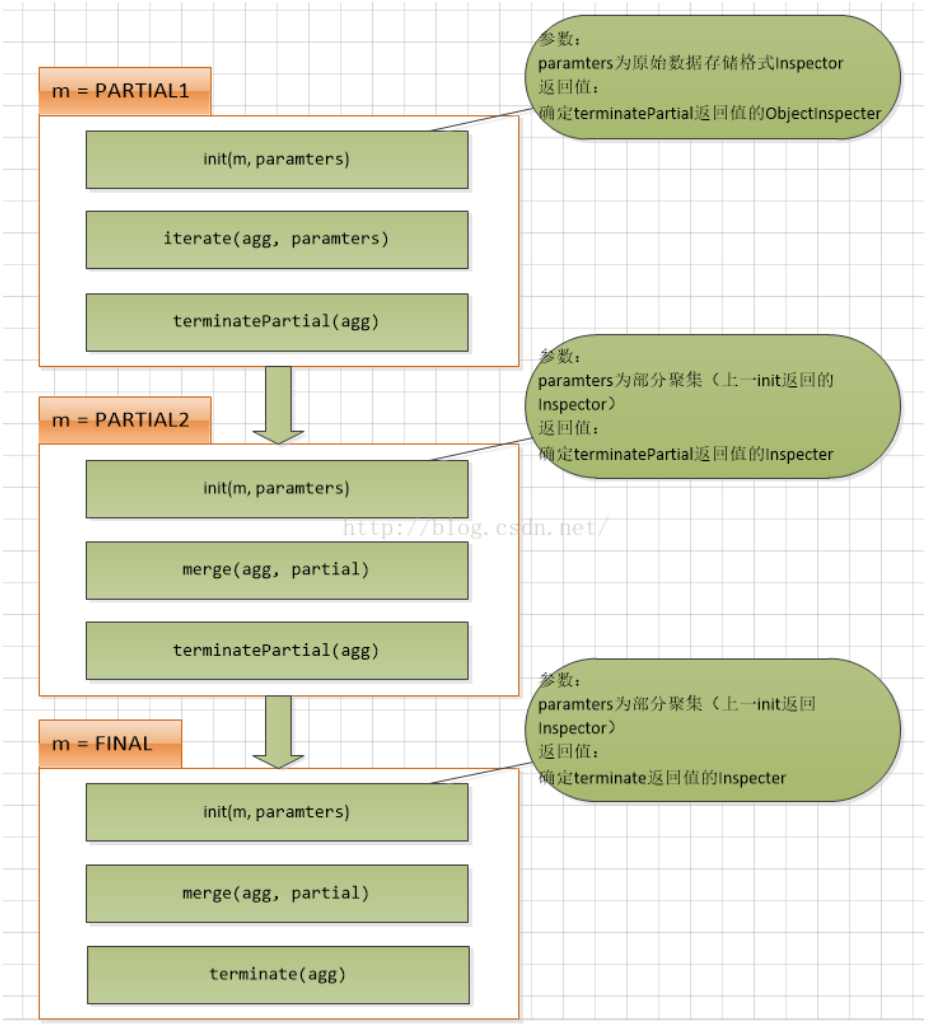
一般情况下，完整 UDAF 逻辑是一个 mapreduce 过程，如果有 mapper 和 reducer，就会经历 PARTIAL1(mapper)，FINAL(reducer)；如果还有 combiner，那就会经历 PARTIAL1(mapper)，PARTIAL2(combiner)，FINAL(reducer)。

3） GenericUDAFEvaluator 方法

```java
// 确定各个阶段输入输出参数数据格式
public  ObjectInspector init(Mode m, ObjectInspector[] parameters) throws HiveException;

// 保存数据聚集结果的类
abstract AggregationBuffer getNewAggregationBuffer() throws HiveException;

// 重置聚集结果
public void reset(AggregationBuffer agg) throws HiveException;

// map阶段，迭代处理输入sql传入列数据
public void iterate(AggregationBuffer agg, Object[] parameters) throws HiveException;

// map与combiner结束返回结果，得到部分数据聚集结果
public Object terminatePartial(AggregationBuffer agg) throws HiveException;

// combiner合并map返回的结果，还有reducer合并mapper或combiner返回的结果。
public void merge(AggregationBuffer agg, Object partial) throws HiveException;

// reducer阶段，输出最终结果
public Object terminate(AggregationBuffer agg) throws HiveException;
```

4） Model 与 Evaluator 关系





5) UDAF 示例代码

```
@Description(name = "letters", value = "_FUNC_(expr) - 返回该列中所有字符串的字符总数")
public class TotalNumOfLettersGenericUDAF extends AbstractGenericUDAFResolver {
  @Override
  public GenericUDAFEvaluator getEvaluator(TypeInfo[] parameters)
      throws SemanticException {
    if (parameters.length != 1) {
      throw new UDFArgumentTypeException(parameters.length - 1,
          "Exactly one argument is expected.");
    }
    ObjectInspector oi = TypeInfoUtils.getStandardJavaObjectInspectorFromTypeInfo(parameters[0]);
    if (oi.getCategory() != ObjectInspector.Category.PRIMITIVE){
      throw new UDFArgumentTypeException(0,
              "Argument must be PRIMITIVE, but "
              + oi.getCategory().name()
              + " was passed.");
    }
    PrimitiveObjectInspector inputOI = (PrimitiveObjectInspector) oi;
    if (inputOI.getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING){
      throw new UDFArgumentTypeException(0,
              "Argument must be String, but "
              + inputOI.getPrimitiveCategory().name()
              + " was passed.");
    }
    return new TotalNumOfLettersEvaluator();
  }
  public static class TotalNumOfLettersEvaluator extends GenericUDAFEvaluator {
    PrimitiveObjectInspector inputOI;
    ObjectInspector outputOI;
    PrimitiveObjectInspector integerOI;
    int total = 0;
    @Override
    public ObjectInspector init(Mode m, ObjectInspector[] parameters)
        throws HiveException {
      assert (parameters.length == 1);
      super.init(m, parameters);
      //map 阶段读取 sql 列, 输入为 String 基础数据格式
      if (m == Mode.PARTIAL1 || m == Mode.COMPLETE) {
```

```
        inputOI = (PrimitiveObjectInspector) parameters[0];
      } else {
      //其余阶段，输入为 Integer 基础数据格式
        integerOI = (PrimitiveObjectInspector) parameters[0];
      }
       //指定各个阶段输出数据格式都为 Integer 类型
      outputOI = ObjectInspectorFactory.getReflectionObjectInspector(
Integer.class,
          ObjectInspectorOptions.JAVA);
      return outputOI;
    }
    /**
    * 存储当前字符总数的类
    */
    static class LetterSumAgg implements AggregationBuffer {
      int sum = 0;
      void add(int num){
        sum += num;
      }
    }
    @Override
    public AggregationBuffer getNewAggregationBuffer() throws HiveExc
eption {
      LetterSumAgg result = new LetterSumAgg();
      return result;
    }
    @Override
    public void reset(AggregationBuffer agg) throws HiveException {
      LetterSumAgg myagg = new LetterSumAgg();
    }
    private boolean warned = false;
    @Override
    public void iterate(AggregationBuffer agg, Object[] parameters)
        throws HiveException {
      assert (parameters.length == 1);
      if (parameters[0] != null) {
        LetterSumAgg myagg = (LetterSumAgg) agg;
        Object p1 = ((PrimitiveObjectInspector) inputOI).getPrimitiveJ
avaObject(parameters[0]);
        myagg.add(String.valueOf(p1).length());
      }
    }
    @Override
    public Object terminatePartial(AggregationBuffer agg) throws HiveE
```

```
xception {
      LetterSumAgg myagg = (LetterSumAgg) agg;
      total += myagg.sum;
      return total;
    }
    @Override
    public void merge(AggregationBuffer agg, Object partial)
        throws HiveException {
      if (partial != null) {
        LetterSumAgg myagg1 = (LetterSumAgg) agg;
        Integer partialSum = (Integer) integerOI.getPrimitiveJavaObjec
t(partial);
        LetterSumAgg myagg2 = new LetterSumAgg();
        myagg2.add(partialSum);
        myagg1.add(myagg2.sum);
      }
    }
    @Override
    public Object terminate(AggregationBuffer agg) throws HiveExceptio
n {
      LetterSumAgg myagg = (LetterSumAgg) agg;
      total = myagg.sum;
      return myagg.sum;
    }
  }
}
```

五、 经典示例推荐

1）文章

1. Hadoop Hive UDF Tutorial - Extending Hive with Custom Functions
2. ObjectInspector

1）项目

1. hive-extension-examples

■ Siry@20190121