Writing DSL in Scala

艾彦波

什么是 DSL

- Domain Specific Language - 特定领域语言，通过使用合适的符号跟抽象来描述特定的问题，是一种编程语言或可执行规范。

示例

- UI - HTML, CSS
- Database - SQL
- Router - Akka HTTP

实战: 求变化率

上周期

一级访问来源	访问用户量
直接访问	12208
搜索引擎	6409
外部链接	1267
社交媒体	691

对比

当前周期

一级访问来源	访问用户量
直接访问	11843
搜索引擎	6095
外部链接	1296
社交媒体	515

一级访问来源	访问用户量	访问用户量-变化率
社交媒体	515	-11.5%
搜索引擎	6095	-0.360%
直接访问	11843	0.569%
外部链接	1296	7.73%

变化率

- 定义：根据某一维度计算指标在当前周期与上周期发生的变化，可用于观测指标的健康度。
- 计算：(当前周期 - 上周期) / 上周期

数据源

```
{
    "data": [
        ["直接访问", 12208],
        ["社交媒体", 6095],
        ["搜索引擎", 515],
        ["外部链接", 1296]
    ]
}
```

[[d1, d2, …, m1, m2, …]]

最简单的方式

```
val rates = data1.groupBy(_.head).map {
    case (k, values) =>
        data2...
}

val result = data1.zipWithIndex.map(entry => entry.1 :+ rates(entry.2))
```

先按照维度做分组 -> 计算每个维度的变化率 -> 再把变化率插入数据行中

出现多维度，多指标怎么解决?

一级访问来源	城市	访问用户量	访问用户量-变化率	访问量	访问量-变化率
直接访问	烟台	2	-92.3%	2	-92.3%
搜索引擎	潍坊	1	-85.7%	1	-97.0%
直接访问	嘉兴	1	-80.0%	1	-87.5%
直接访问	潍坊	4	-75.0%	4	-85.7%
外部链接	常州	1	-75.0%	2	-50.0%
直接访问	温州	1	-75.0%	1	-88.9%
直接访问	圣克拉拉	2	-71.4%	2	-77.8%
直接访问	四平	1	-66.7%	1	-66.7%
直接访问	晋中	1	-66.7%	1	-66.7%
直接访问	萍乡	1	-66.7%	1	-66.7%

周期内数据缺失怎么解决?

- 上一周期没有某个维度的数据
- 当前周期没有某个维度的数据

其他需求

- 排序怎么计算?
- Ranking 怎么计算?
- 数据过滤?

实现起来代码复杂度难以控制

它可以被描述成一个 SQL 查询过程

select d1,d2, …, m1, m1_r, m2, m2_r, … from current c join prev p
on (c.d1 == p.d1 and c.d2 == p.d2 …)

Table DSL

- insert/load
- full/inner join
- select
- sort
- ranking

Table

```
final case class Table(name: String, columns: Seq[String]) {
  private[this] var joined: Table = _
  private[this] val elements: util.List[Seq[_]] = new util.ArrayList[Seq[_]]()

}
```

joined: 记录被 join 的 table

elements: 记录当前 table 的数据集

Load / Insert

```
def load(rows: Seq[Seq[_]]): Table = {
  rows.foreach(elements.add)
  this
}

def insert(row: Seq[_]): Table = {
  elements.add(row)
  this
}
```

Join

```
def join(table: Table): Table = {
  joined = table
  this
}
```

一个记录外表引用的过程

Full - c1, c2

```
def full(keys: Seq[String]): Table = {
  // 给左右表的列加 namespace(格式 table.column) 组成成新 columns
  val newColumns = columnsWithNamespace(name, columns) ++ columnsWithNamespace(joined.name, joined.columns)
  // 生成新的表
  val newTable = Table("full_joined_temp", newColumns)
  // 对左右表的每行计算 rowkey
  val leftRows = elements.asScala.groupBy(row ⇒ rowkey(row, columns, keys))
  val rightRows = joined.collect.groupBy(row ⇒ rowkey(row, joined.columns, keys))
  // 取左右表的并集
  val data = leftRows.keySet union rightRows.keySet map { key ⇒
    // 根据 rowkey 从左右表获取数据，如果没有找到 rowkey 对应的数据，做默认值处理
    left ++ right
  }
  newTable.load(data)
}
```

Inner - c1, c2

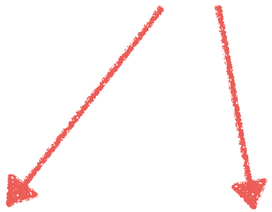
```scala
def inner(keys: Seq[String]): Table = {
  val _columns = columnsWithNamespace(name, columns) ++ columnsWithNamespace(joined.name, joined.columns)
  val newTable = Table("joined_temp", _columns)
  val leftRows = elements.asScala.groupBy(row ⇒ rowkey(row, columns, keys))
  val rightRows = joined.collect.groupBy(row ⇒ rowkey(row, joined.columns, keys))
  // 取左右表的交集
  val data = leftRows.keySet intersect rightRows.keySet map { key ⇒
    // 根据 rowkey 从左右表取数据，因为取的是交集，所以不会出现数据为 null 的情况
    left ++ right
  }
  newTable.load(data)
}
```

Select - c1/c2 as alias

```
def select(expressions: String*): Table = {
  // 解析表达式，并且获得新的别名(按照 as 做分割)
  val (names, aliases) = ...
  val nt = Table(name, aliases)
  elements.parallelStream().forEach(row ⇒ {
    val nrow = names map { name ⇒
      val index = columns.indexOf(name)
      // 如果只是查询某列(不做表达式计算)，直接在行中返回数据
      if (index > -1) {
        row(index)
      } else {
        // 将表达式中的列替换成代码形式的数据
        var expr = genrateExpr(name, columns, row)
        // 使用 pattern match 模式根据表达式的内容查找内建函数执行，否则提交给表达式引擎执行
        expr match {
          case Constants.FLUCTUATE_FUNC_REGEX(c, p) ⇒ fluctuate(c, p)
          case Constants.FORMAT_FUNC_REGEX(d, pattern, suffix) ⇒ format(d.toDouble, pattern, suffix)
          case _ ⇒ new SpelExpressionParser().parseExpression(expr).getValue
        }
      }
    }
    nt.insert(nrow)
  })
  nt
}
```

Case 1

table



```
current join prev full Seq("d1", "d2") select (
  "current.d1 as d1","current.d2 as d2", "fluctuate(current.m1, prev.m1) as m1_r", "fluctuate(current.m2, prev.m2) as m2_r"
)
```

Sort - m1 desc

```
def sort(expression: String): Table = {
  // 解析排序表达式，得到排序列与排序规则
  val (name, desc) = {
    val segments = expression.split(' ')
    (segments.head, segments.last.equalsIgnoreCase("desc"))
  }
  val data = elements.asScala.sorted((x: Seq[_], y: Seq[_]) =>{
    val index: Int = columns.indexOf(name)
    val left = x(index)
    val right = y(index)
    // 使用 pattern match 对不同类型的数据做不同的对比策略
    val compare = left match {
      case b: BigDecimal ⇒ b.doubleValue().compare(right.asInstanceOf[BigDecimal].doubleValue())
      case i: Int        ⇒ i.compare(right.asInstanceOf[Number].intValue())
      case l: Long       ⇒ l.compare(right.asInstanceOf[Number].longValue())
      case d: Double     ⇒ d.compare(right.asInstanceOf[Number].doubleValue())
      case s: String     ⇒ s.compare(right.asInstanceOf[String])
      case _             ⇒ left.toString.compare(right.toString)
    }
    if (desc) -compare else compare
  })
  Table(name,columns).load(data)
}
```

Ranking - c1, c2

```
def ranking(columns: Seq[String]): Table = {
  // 给需要 ranking 的列添加 ranking 列(c1, c1_ranking), 组成新的 columns
  val _columns = ...
  val nt = Table(name, _columns)
  // 给每列计算最大/小值
  val ranks = columns.map { ... }
  val indices = columns.map(c ⇒ this.columns.indexOf(c))
  elements.asScala.map { row ⇒
    // 计算 ranking 值, 将 ranking 值添加到新行中
    val nrow = ...
    nt.insert(nrow)
  }
  nt
}
```

Case 2

```
current join prev full Seq("d1", "d2") select (
  "current.d1 as d1","current.d2 as d2", "fluctuate(current.m1, prev.m1) as m1_r", "fluctuate(current.m2, prev.m2) as m2_r"
) sort "m1_r desc" ranking Seq("m1", "m2")
```

清除 Seq

```
def full(keys: Seq[String]): Table -> def full(keys: String*): Table

def inner(keys: Seq[String]): Table -> def inner(keys: String*) Table

def ranking(columns: Seq[String]): Table -> def ranking(columns: String*): Table
```

Case 3

```
current join prev full ("d1", "d2") select (
  "current.d1 as d1","current.d2 as d2", "fluctuate(current.m1, prev.m1) as m1_r", "fluctuate(current.m2, prev.m2) as m2_r"
) sort "m1_r desc" ranking ("m1", "m2")
```

！ ！ ！

```
current full ("d1", "d2") join prev select ("m1")
```

使用顺序错了

```scala
case class Table(name: String, columns: Seq[String]) {

  def collect: Seq[Seq[_]]

  def load(rows: Seq[Seq[_]]): Table

  def insert(row: Seq[_]): Table

  def select(expressions: String*): Table

  def sort(expression: String): Table

  def ranking(columns: String*): Table

  def join(table: Table): JoinedTable = JoinedTable(name, columns, table)

}

class JoinedTable(override val name: String, override val columns: Seq[String], joined: Table) extends Table(name, columns) {

  def inner(keys: Seq[String]): Table

  def full(keys: String*): Table
}
```

```
current full ("d1", "d2") join prev select ("m1")
```

语法错误，不能编译

问题

- 数据某列包含 null - 特定逻辑，针对性解决
- 数据类型不同 - 针对类型做不同的计算策略
- 性能问题 - SpringEL 表达式执行会使用反射，使用内建函数解决

性能测试

维度	2
指标	2
左表数据量	10万
右表数据量	10万
Join 方式	full join

测试结果

	Twitter Utils(eval)	SpringEL	内建函数	内建函数 + ParallelStream
时间(ms)	🛁	1841	1334	1270

测试结论

- 使用 SpringEL 作为主要表达式执行引擎
- 复杂的计算逻辑使用增加内建函数执行，不使用表达式引擎执行

怎么写好 DSL?

- 明确需要解决的问题 - 计算变化率
- 选择一种表达形式 - SQL
- 抽象出关键的步骤 - join, select, order, ranking, take
- 设计容易理解，形象的语义 - EventLinkRenderer ~ KeywordHighlightingRenderer ~ DesensitizationRenderer ~ UserIDRenderer ~ AttributesLocalKeyRenderer
- 完善限制 - 避免错误发生在运行时

sbt-dependency-updates

ec2-jvm-flamegraph

多谢