Prompt

源码文本

DataQL 文档表示建议使用 Unicode字符序列。但这并不是强制的，您可以通过 java.io.Reader 来读取您的字符流数据。故本文不会强调您的编写查询语言所使用的字符集。

DataQL 支持单行注释和多行注释两种注释方式，被注释的代码会被语法解析器忽略。

单行注释：以 // 开头后面的当前换行符内的所有内容均为注释内容。

多行注释：以 \\* 开始直到遇到 \*/ 为止，中间的所有内容均为注释。

空白字符

空白字符用于提高源文本的可读性，并作为标记之间的分隔，任何数量的空白都可能出现在任何标记之前或之后。 标记之间的空白对于 DataQL 文档的语义意义并不重要。

DataQL 会把这些字符识别为空白字符：(空格)、\t、\n、\r、\f

关键字

关键字 含义

if 条件语句的引导词

else 用在条件语句中，表明当条件不成立时的分支

return 三大退出指令之一，终止当前过程的执行并正常退出到上一个执行过程中

throw 三大退出指令之一，终止所有后续指令的执行并抛出异常

exit 三大退出指令之一，终止所有后续指令的执行并正常退出

var 执行一个查询动作，并把查询结果保存到临时变量中

run 仅仅执行查询动作，不保留查询的结果

hint 写在 DataQL 查询语句的最前面，或者通过局部 hint，用于设置一些执行选项参数

import 将另外一个 DataQL 查询导入并作为一个 Udf 形式存在、或直接导入一个 Udf 到当前查询中

as 与 import 关键字配合使用，用作将导入的 Udf 命名为一个本地变量名

import 将另外一个 DataQL 查询导入并作为一个 Udf 形式存在、或直接导入一个 Udf 到当前查询中

true 基础类型之一，表示 Boolean 的：真值

false 基础类型之一，表示 Boolean 的：假值

null 基础类型之一，表示 NULL 值

标识符

在编写 DataQL 中用来表示查询中的一些实体。例如：变量名、参数名

标识符必须满足正则表达式：[\_a-zA-Z][\_0-9a-zA-Z]\*

在一些接口中数据对象的 key 可能会超出标识符标示的范围，这时候可以使用反引号 ´xxx´ 的形式来标示标识符。

分割符

主要用于分割语义，例如：表达式计算中的提权；语句块的包裹；函数入参的圈定等等。DataQL 的分割符有下面几种：

分割符 含义

() 函数入参的圈定，表达式中的计算项提取权

{} 用来定义复合语句

[] 对数据通过下标方式取值操作

, 不同参数的分割；对象键值对或数组元素之间的分割

: 对象键值对，键和值之间的分割

; 语句的结束，DataQL 会自动推断语句结束，因此语句结束分割符并不是必须的

运算符

数学运算 位运算 比较运算 逻辑运算

+ 加法 & 按位于运算 > 大于 || 逻辑或

- 减法 | 按位或运算 >= 大于等于 && 逻辑与

\* 乘法 ! 按位取反 < 小于

/ 除法 ^ 异或 <= 小于等于

\ 整除 << 左位移 == 等于

% 取摸 >> 有符号右位移 != 不等于

>>> 无符号右位移

DataQL 是弱类型定义的查询语言，在DataQL 中所有数据都会被归结到有限的几种类型上。无需定义数据类型结构，在弱类型系统中编写查询会非常方便，它去掉了繁杂的类型定义。

提示

弱类型定义会带来 字段类型不确定 的问题。即：在没有搭配类型转换函数情况下，字段类型可能是 boolean 也有可能是 string，这具体要看原始数据是什么类型。

数据类型

数据类型 表示方式 详情

布尔 true 或 false 表示真假值

数值 负无穷大 或 0 或 正无穷大 浮点数、整数、科学计数法表示的数

字符串 '...' 或 "..." 字符串 或 单个字符

空值 null 或 NULL 空值

集合 [...] 数组 或 多维数组 带有顺序的多组数据的集合

对象 {'key':...} 具有键值对的数据体 DataQL 的对象不支持方法，但是可以具备 UDF 类型的属性

UDF lambda 函数 或 一个外部的 UDF 一个外部的 net.hasor.dataql.Udf 接口函数定义

DataQL 中书写的 lambda 函数也被称作为 UDF。

一个扩展代码片段的定义，也属于 UDF 的范畴

数据的表示

布尔类型

举例：var bool = true;

数值类型

二进制表示法：0b01010101100 或 0B01010101100

十进制表示法：-0000234 或 123

八进制表示法：0o1234567 或 0O1234567

十六进制表示法：0x12345 或 0X12345

科学计数法：a \* 10的n 次幂的形式，其中 1 < a ＜ 10

关于负数：目前只有十进制表示法中提供了负数的表示能力。

字符串

举例：var str = 'abcdefg...'; 或 var str = "abcdefg...";

空字符串：var empty = ''; 或 var empty = "";

Unicode字符表示：var unicode = '\u0041'; （拉丁字母 A）

空值

举例：null

集合

格式：[...]

空集合：[]

多维集合：[[...],[...]]

对象

格式：{"k1": ... ,"k2": ...}

空对象：{}

多层结构：{"k1": { ... }, "k2": { ... } }

UDF

外部UDF：外部的 UDF 被引入之后，通常以标识符形式表示它。

lambda： 中书写的 lambda 表达方式为：var foo = () -> { /\* 代码块 \*/ }

外部代码片段：var a = @@xxx() <% /\* 外部代码块 \*/ %>

JSON

DataQL 可以直接表达 Json 数据（Json 的 Key 必须通过双引号或单引号形式包裹起来）

自动类型提升

当两个操作数之间做运算时，其中一个数值类型和另外一个数值的类型不一致情况下。将两个数值的类型统一转换成同一种类型的行为。

类型自动提升行为表：

• bool byte sort int long float double BigInt Decimal string

bool • byte sort int long float double BigInt Decimal string

byte byte • sort int long float double BigInt Decimal string

sort sort sort • int long float double BigInt Decimal string

int int int int • long double double BigInt Decimal string

long long long long long • double double BigInt Decimal string

float float float float double double • double Decimal Decimal string

double double double double double double double • Decimal Decimal string

BigInt BigInt BigInt BigInt BigInt BigInt Decimal Decimal • Decimal string

Decimal Decimal Decimal Decimal Decimal Decimal Decimal Decimal Decimal • string

string string string string string string string string string string •

数值宽度

名称 缩写 长度(Bit) 取之范围

字节型 byte 8 -128 ~ 127

短整型 sort 16 -32768 ~ 32768

整型 int 32 -2147483648 ~ 2147483648

长整型 long 64 -9233372036854477808 ~ 9233372036854477808

浮点型 float 32 -3.40292347E+38 ~ 3.40292347E+38

双精度 double 64 -1.79769313486231570E+308 ~ 1.79769313486231570E+308

大整数 BigInt 整型，正无穷 ~ 负无穷

大浮点数 Decimal 浮点数，正无穷 ~ 负无穷

默认数值宽度

数值表示的默认宽度，举例：var = 235 这个数即可是 byte 的也可能是 int 类型。 默认宽度是指在 DataQL 查询语句中，定义的数字类型数据所使用的：默认类型。

整数：byte、short、int(默认)、long、big

浮点数：float、double(默认)、big

修改默认数值计算宽度

hint MIN\_DECIMAL\_WIDTH = 'double'

hint MIN\_INTEGER\_WIDTH = 'int'

提示

float 或者 double，由于其自身存储特性导致在做运算时出现精度丢失。这时建议将 MIN\_DECIMAL\_WIDTH 宽度提升为 big。 这样可以将浮点数就会以 Decimal 方式进行计算以解决精度丢失问题，但要注意性能损耗。

舍入问题

DataQL 两个浮点数计算时会默认保留 20 位小数。多余的小数部分将会以 四舍五入 方式进行进位。

修改修改浮点数计算舍入精度使用的 DataQL 语句

hint MAX\_DECIMAL\_DIGITS = 20

更换舍入规则使用的 DataQL 语句

hint NUMBER\_ROUNDING = 'HALF\_EVEN'

可选的舍入选项为（参见Hint：NUMBER\_ROUNDING）：

选项值 默认 含义

UP 向远离零的方向舍入。舍弃非零部分，并将非零舍弃部分相邻的一位数字加一

DOWN 向接近零的方向舍入。舍弃非零部分，同时不会非零舍弃部分相邻的一位数字加一，采取截取行为

CEILING 向正无穷的方向舍入。如果为正数，舍入结果同 ROUND\_UP 一致；如果为负数，舍入结果同 ROUND\_DOWN 一致

注意：此模式不会减少数值大小

FLOOR 向负无穷的方向舍入。如果为正数，舍入结果同 ROUND\_DOWN 一致；如果为负数，舍入结果同 ROUND\_UP 一致

注意：此模式不会增加数值大小

HALF\_UP 是 四舍五入 向 最接近 的数字舍入，如果与两个相邻数字的距离相等，则为向上舍入的舍入模式

注意：如果舍弃部分 >= 0.5，则舍入行为与 ROUND\_UP 相同；否则舍入行为与 ROUND\_DOWN 相同

HALF\_DOWN 五舍六入 向 最接近 的数字舍入，如果与两个相邻数字的距离相等，则为向下舍入的舍入模式

注意：如果舍弃部分 > 0.5，则舍入行为与 ROUND\_UP 相同；否则舍入行为与 ROUND\_DOWN 相同

HALF\_EVEN 向 最接近 的数字舍入，如果与两个相邻数字的距离相等，则相邻的偶数舍入

如果舍弃部分左边的数字奇数，则舍入行为与 ROUND\_HALF\_UP 相同；

如果为偶数，则舍入行为与 ROUND\_HALF\_DOWN 相同。

注意：在重复进行一系列计算时，此舍入模式可以将累加错误减到最小。此舍入模式也称为 银行家舍入法，主要在美国使用。

四舍六入，五分两种情况，如果前一位为奇数，则入位，否则舍去。

UNNECESSARY 断言请求的操作具有精确的结果，因此不需要舍入

如果对获得精确结果的操作指定此舍入模式，则抛出 ArithmeticException

语句

DataQL 一共有 8 种

var - 执行语句并将结果存入变量

run - 仅执语句，但忽略结果

return、throw、exit - 三大退出语句

if、else - 分支选择语句

hint - 选项语句

import - 资源导入

var - 执行语句并将结果存入变量

语法格式

var <变量名> = <表达式 or 值对象 or 函数定义>

提示

var 语句是用的最广泛的语句，它有两个特性：

定义变量

执行并存储表达式的值

变量定义的作用是可以保存临时查询结果，以及保存查询过程中需要用到的函数定义。例如：

var a = 1

执行表达式

var a = 1 + 1

run - 仅执语句，但忽略结果

语法格式

run <表达式 or 值对象 or 函数定义>

run 语句相比 var 语句，只是不能保存查询结果。因此 run 语句也就不具备定义变量的能力。例如：下面这个语法是正确的，但是没有实际意义：

run 1 + 1

在一些特殊场合中，查询的中间结果并不重要。因此也就无需专门为这些临时查询开辟数据保存空间，例如：把查询到的每一条数据，都调用一次 UDF。

var dataSet = ...

run dataSet => [ callUdf(#) ] // 遍历 dataSet 集合并调用 callUdf 函数

return、throw、exit - 三大退出语句

DataQL 有三大退出指令，分别是：return、throw、exit。它们的区别如下：

关键字 含义

return 终止当前过程的执行并正常退出到上一个执行过程中。

throw 终止所有后续指令的执行并抛出异常。

exit 终止所有后续指令的执行并正常退出。

它们的一般语法格式：

return <状态码>, <表达式 or 值对象 or 函数定义>

throw <状态码>, <表达式 or 值对象 or 函数定义>

exit <状态码>, <表达式 or 值对象 or 函数定义>

还可以不指定状态码（其它退出语句同理）

return <表达式 or 值对象 or 函数定义>

返回结果

return ...

抛出异常，同时带上一个查询结果

throw ...

返回 200 状态，查询结果是一个对象

return 200, { ... }

if - 分支选择语句

让 DataQL 变得灵活的正式由于 if 语句的存在，它的用法和 Java 或 JavaScript 相同。其语法格式为：

if (boolean\_expression) {

/\* 如果布尔表达式为真将执行的语句 \*/

} else {

/\* 如果布尔表达式为假将执行的语句 \*/

}

用法

if (testExpr) {

return ...

} else {

return ...

}

DataQL 也支持多重分支

if (testExpr1) {

return ...

} else if (testExpr2) {

return ...

} else if (testExpr3) {

return ...

} else {

return ...

}

hint - 选项语句

语法格式

hint <选项名称> = <选项值>

选项名称：是满足 标识符 特征的

选项值：可以定义 数字、字符串、布尔、null 四种基本类型数据。

计算百分比，精确到小数点后2位

hint MAX\_DECIMAL\_DIGITS = 4;

hint NUMBER\_ROUNDING = 'HALF\_UP'; // 默认值也是 HALF\_UP

var num = 1.0

var sumNum = 3.0

return num / sumNum \* 100 + "%"

...

执行查询结果为：33.33%

hint 语句要放在 import 语句的前面，其作用是设置一些执行查询使用的选项参数（更多 HINT 可以参考这里）

import - 资源导入

语法格式

import "<函数类或函数包类>" as <别名>

import @"<资源地址>" as <别名>

别名必须满足 标识符

import 语句必须要放在整个查询过程最开始的地方，它的作用是导入外部 函数 或者 函数包。被导入的资源会以 var 的方式定义。

例如：导入 net.hasor.dataql.sdk.CollectionUdfSource 函数包，并通过函数包中的函数判断一个集合是否为空。

import "net.hasor.dataql.sdk.CollectionUdfSource" as collect

var dataSet = []

var test = collect.isEmpty(dataSet)

import 语句还有另外一个功效就是，可以将另外一个 DataQL 查询当作 函数 的形式导入到当前查询中。

例如：有一个已经保存在文件中的 DataQL 查询语句，这个查询保存在 classpath 路径中，/net/hasor/demo.ql

return { "name": "马三", "msg": "Hello DataQL." }

有另外一个 DataQL 查询引用了 demo.ql 这个查询并拿到返回的结果。

import @"/net/hasor/demo.ql" as demo

return demo()

表达式

DataQL 具备完整的表达式计算能力，这使得数据在转换过程中在需要数值计算的情况上变得非常好用。这会节省大量精力很多的精力来编写 UDF 的操作。

一个典型的场景就是是货币和汇率的转换

return orderByUser({'id': 4}) => {

'orderID',

'price\_rmb' : '¥' + (price\_rmb \* 6.9) + '元' ,

'buyTime'

}

一元运算

对一个 bool 值进行取反

var bool = true;

return !bool;

可以被一元运算的类型有：boolean、number

一元运算 针对的类型 语意

! boolean 对Boolean 取非

- number 数值的相反数

二元运算

例如 1 + 2 等于 3

var a = 1;

var b = 2;

return a + b;

数学运算 位运算 比较运算 逻辑运算

+ 加法 & 按位于运算 > 大于 || 逻辑或

- 减法 | 按位或运算 >= 大于等于 && 逻辑与

\* 乘法 ! 按位取反 < 小于

/ 除法 ^ 异或 <= 小于等于

\ 整除 << 左位移 == 等于

% 取摸 >> 有符号右位移 != 不等于

>>> 无符号右位移

提示

二元运算主要还是面向 number。但加法运算是比较特殊的，它可以对任意类型做加法。执行结果就是两个数据字符串拼接。

三元运算

语法为：testExpr ? expr1 : expr2

var bool = true;

return (bool ? 1 : 2) ;

值域

值域是一种隔离机制。例如：同样名称的参数可以出现在不同的 域 里，在 DataQL 中一共有三个可以用的值域符号：

$ 值域符

# 值域符

@ 值域符

提示

DataQL 的值域，在没有明确分别它们的时候。三个访问符的值域内容是没有任何区别的。

值域符的应用场景只有两个

值域参数隔离

表达式访问符

值域参数隔离

获取程序传入的参数必须使用：<访问符>{<参数名>} 方式来获取。这个特性类似带参的SQL，例如：

用不同值域隔离同名参数的例子

// 创建一个 Map 为每个值域中都放入不同变量，但是变量名都是 'a'

Map<String, Map<String, Object>> objectMap = new HashMap<>();

objectMap.put("#", new HashMap<String, Object>() {{

put("a", 1);

}});

objectMap.put("$", new HashMap<String, Object>() {{

put("a", 2);

}});

objectMap.put("@", new HashMap<String, Object>() {{

put("a", 3);

}});

然后通过 Query 接口创建查询并执行查询（通过 CustomizeScope 接口来返回不同访问符的数据 Map）

${abc}、@{abc}、#{abc} 三个值域

DataQL dataQL = ...

Query query = dataQL.createQuery("return [#{a},${a},@{a}];");

DataModel dataModel = query.execute(new CustomizeScope() {

public Map<String, ?> findCustomizeEnvironment(String symbol) {

return objectMap.get(symbol);

}

}).getData();

查询结果

[1, 2, 3]

表达式访问符

表达式访问符同样使用了 @、#、$ 三个符号，在表达式中的访问符是指类似如下的取值表达式：

$ss.sss.sss

#ss[abc].sss.sss

@abc.abc(true).sss

表达式中的访问符不同含义如下：

$ 表示环境栈根

# 表示环境栈顶

@ 表示整个环境栈(数组形态)

要理解这种带有访问符的含义需要理解 DataQL 的运行模式，DataQL 的运行时模型和 JVM 有些类似。

不同的是 DataQL 采用的是两栈一堆。比 JVM 堆栈模型多了一个 环境栈。

环境栈

运行栈(作用和 JVM 相似)

数据堆(作用和 JVM 相似)

查询过程中一般情况下环境栈始终是空的，当遇到 => 操作时。 DataQL 会把 => 符左边的表达式值放入环境栈，当转换结束时 DataQL 会把表达式值从环境栈中删掉。 如果在转换过程中遇到第二次 => 操作，那么会在环境栈顶中放入新的数据。例如：下面这个查询就会出现双层环境栈

一个DataQL查询

var data = {

"userInfo" : {

"username" : "xxxxx",

"password" : "pass"

},

"basicInfo" : {

"name" : "马三",

"sex" : "F"

},

"id" : 12345667

}

return data => { // 第一次出现

"userInfo" : userInfo => { // 第二次出现

"username",

"password",

"userId" : $.id

},

"name" : basicInfo.name // 这种形式不会出现

}

执行结果为

{

"userInfo":{

"username":"xxxxx",

"password":"pass",

"userId":12345667

},

"name":"马三"

}

提示

所有表达式在编译的时都会有一个访问符，如果用户没有指定那么将会使用 # 作为默认访问符。

即便所有表达式在编译之后都具有访问符，但这并不代表数据的源头都来自环境栈。编译器会优先在本地变量表中查找。具体逻辑在 NameRouteVariableInstCompiler 类中。

例如：如下例子，在对一颗 Tree 进行结构变换时。希望每一层都能带上 parentID。

样本数据

[

{

"id": 1,

"label": "t1",

"children": [

{

"id": 2,

"label": "t2",

"children": [

{

"id": 4,

"label": "t4",

"children": []

}

]

},

{

"id": 3,

"label": "t3",

"children": []

}

]

},

{

"id": 5,

"label": "t5",

"children": []

}

]

DataQL 查询

var treeData = ..// 样本数据

var treeFmt = (dat) -> {

return {

"id" : dat.id,

"parent\_id": ((@[-3] !=null)? @[-3].id : null), // 获取整个环境栈然后在倒数第三层上获取

"label" : dat.label,

"children" : dat.children => [ treeFmt(#) ]

}

}

return treeData => [ treeFmt(#) ]

取值

对象取值

DataQL 支持面向对象中的取值方式，例如下面几个例子：

var userInfo = {

'id': 4,

'username': '马A'

}

// 返回用户ID为4的用户名

return userInfo.username;

函数结果取值

// userInfo 返回和上述例子相同结构的数据

var userInfo = userByID({'id': 4});

// 在函数返回值后直接取值

return userByID({'id': 4}).username;

数组中取值

var userInfo = [

{

'id': 4,

'username': '马A'

},{

'id': 2,

'username': '马B'

}

]

// 返回用户列表中第一条数据的用户名

return userList()[0].username;

下标取值

DataQL 支持类似 JavaScript 中类似的下标方式取值。在 DataQL 中有结构的数据只有两种：集合 和 Map。

因此带有结构的对象类型数据其就相当于一个 Map，可以通过下面的方式来获取属性值。

var userInfo = {

'id': 4,

'username': '马A'

}

return userInfo['username'];

连续下标取值

var userList = [

{

'id': 4,

'username': '马A'

},{

'id': 2,

'username': '马B'

}

]

// 返回用户列表中第一条数据的用户名

return userList[0]['username'];

数组下标取值

对于数字形式的下标分为：正向索引、反向索引 两种。例如：有一个数据集：

var list = [0,1,2,3,4,5,6,7,8,9]

正向索引：正数，从前向后数，从 0 开始。

list[3] = 3

list[5] = 5

反向索引：负数，从后向前数，从 1 开始。

list[-3] = 7

list[-5] = 5

索引溢出：无论正向还是反向，都会涉及到下标越界问题，这种情况称之为索引溢出。DataQL 有三种可以选择的处理行为：

行为 DataQL 语句 功效

throw hint INDEX\_OVERFLOW = 'throw' 当遇到索引溢出情况时严格的抛出，ArrayIndexOutOfBoundsException 异常

null hint INDEX\_OVERFLOW = 'null' 当遇到索引溢出情况时返回 null

near(默认) hint INDEX\_OVERFLOW = 'near' 正向索引溢出 list[100] 取最后一个，反向索引溢出 list[-100] 取第一个

下标变量

DataQL 允许使用变量替代下标中的值，这使得下标取值可以更加灵活。

var userInfo = {

'id': 4,

'username': '马A'

}

// 定义一个变量，变量表示要取值的字段名。

var columnKey = 'username';

// 通过下标变量方式来取值

return userInfo[columnKey];

重定义：在任何时候都可以通过 var 来重新定义一个已存在的变量

var data = [0,1,2,3,4,5,6,7,8,9]

// 此时 data 是一个数组。

var data = {}

// 此时 data 是一个对象

var data = 123

// 此时 data 是一个数字

在重定义变量的时候，可以访问自身

var tmpSql = "select \* from user"

var tmpSql = tmpSql + " where uid = ?"

结果转换

提示

结果转换是 DataQL 核心能力，它的作用是让原始数据的格式进行变换。这个变换可以大体归纳为如下几种：

组装：凭空构造一个全新的结构，每个数据元素可以从任何变量中获取。

变换：将已有数据的结构进行变换

变换语法为：

expr => fmt，fmt 有两种表达方式别为：对象 和 数组。

组装

Case1：返回一个对象数据，对象中包含一个 value 字段。value 字段的来源是 data 变量：

var data = 123;

return {

"value" : data

};

Case2：如果返回一个更加复杂的结构或者是将多个值类型组合到一个数据结构中：

var userName = "马三"; // 姓名

var userAge = 23; // 年龄

// 返回一个对象数据，将用户名称和年龄组装到一个对象中

return {

"name" : userName,

"age" : userAge

};

Case3：组装数组数据：

var data = 123; // 值

return [

data

];

Case4：组装多个元素的数组：

var data1 = 123; // 值1

var data2 = 456; // 值2

// 返回2个元素的数组

return [

data1, data2

];

Case5：下面这个组装稍微复杂一点，它组装了两个字段。但是两个字段分别来自于同一个数组数据的不同元素：

var data = [123, 456];

return {

"element\_0" : data[0], // 123

"element\_1" : data[1] // 456

};

数组的变换

比如我有一个如下数据

var data = [

{

"name" : "马三",

"age" : 23,

"type" : 1

},{

"name" : "马四",

"age" : 23,

"type" : 2

}

];

Case1：希望得到一个新的数据集，数据集中只包含 name 和 age 字段：

return data => [

{

"name", "age"

}

];

Case2：或者新的数据集中只包含 name 和 age 字段，但同时改一下字段名。

return data => [

{

"userName" : name, // 取 name 字段的值作为 userName

"userAge" : age

}

];

Case3：下面这个也比较常用，从数据集中取得所有用户名的列表。最终的数据是一个字符串数组：

return data => [ name ];

Case4：将一组值类型变换成一组对象 DataQL 也可以做到：

var data = ["马三", "马四"];

return data => [

{

"name" : # // 符号 "#" 表示在对每个元素进行转换的过程中的那个元素本身。

}

];

上面的代码用 JavaScript 来表示如下：

var data = ["马三", "马四"];

var arrayList = [];

for (i = 0; i < data.length; i++) {

arrayList.push({"name": data[i]}); // 这里的 data[i] 就相当于 # 号

}

return arrayList;

Case5：如果是只想取元素的第一个值，并且将第一个元素的结构进行变换。还可以这样简写：

return data => {

"userName" : name, // 取 name 字段的值作为 userName

"userAge" : age

};

Case6：有的时候可能会遇到多维数组，比如我们要为下列数字矩阵的每一个值都加上一个字符串前缀：

var data = [

[1,2,3],

[4,5,6],

[7,8,9]

]

return data => [

# => [ // 在结果转换中对当前元素进行二次转换

"值：" + #

]

]

查询结果

[

["值：1","值：2","值：3"],

["值：4","值：5","值：6"],

["值：7","值：8","值：9"]

]

对象的变换

首先我们有一个对象

var data = {

"userID" : 1234567890,

"age" : 31,

"name" : "this is name.",

"nick" : "my name is nick.",

"sex" : "F",

"status" : true

}

Case1：通过变换而非组装的方式将其转换为一个元素的数组

return data => [ # ];

上述这种写法通常用于，需要结果必须是数组的情况。但是又不能确定返回的值一定是数组。

比如：默认情况下 @@sql 查询结果中如果只含有一个元素，那么它会返回对象而非 List/Map。（当然也可以通过 FRAGMENT\_SQL\_OPEN\_PACKAGE hint 来控制 @@sql 返回值形态）

Case2：对象的变换通常都是对对象结构上的变化，例如：查询结果中将年龄和性别放入一个新的结构中返回：

return data => {

"name",

"info" : {

"age",

"sex"

}

}

查询结果

{

"name": "this is name.",

"info": {

"age": 31,

"sex": "F"

}

}

使用表达式

在对结果变换过程中 DataQL 还允许通过表达式来对字段进行重新计算，例如：

将年龄表示的数值转换为 xx岁。将性别表示的 F/M,转换为：男/女

return data => {

"name",

"age" : age + "岁",

"sex" : (sex == 'F') ? '男' : '女'

}

查询结果

{

"name": "this is name.",

"age": "31岁",

"sex": "男"

}

函数

定义函数

在 DataQL 查询中可以直接通过 DataQL 语言本身定义一个函数，然后在后续查询中使用它。一个典型的场景就是对性别字段的转换：

var convertSex = (sex) -> {

return (sex == 'F') ? '男' : '女'

};

var data = {

"userID" : 1234567890,

"age" : 31,

"name" : "this is name.",

"nick" : "my name is nick.",

"sex" : "F",

"status" : true

};

return data => {

"name",

"age" : age + "岁",

"sex" : convertSex(sex)

}

函数的 Lambda写法

提示

Lambda 写法相当于一个匿名的函数。

例如在使用集合函数的过滤功能时，如果没有 Lambda 写法可能整个查询写出来会比较臃肿。比如我们有如下数据：

// 原始数据

var dataList = [

{"name" : "马一" , "age" : 18 },

{"name" : "马二" , "age" : 28 },

{"name" : "马三" , "age" : 30 },

{"name" : "马四" , "age" : 25 }

]

// 过滤后的数据

// [

// {"name" : "马二" , "age" : 28 },

// {"name" : "马三" , "age" : 30 },

// {"name" : "马四" , "age" : 25 }

// ]

只保留年龄大于20岁的数据：

import 'net.hasor.dataql.fx.basic.CollectionUdfSource' as collect;

// 数据

var dataList = ...

// 年龄过滤逻辑

var filterAge = (dat) -> {

return return dat.age > 20;

};

// 调用 filter 函数

return collect.filter(dataList, filterAge);

换成 Lambda 写法可以省掉一个函数的定义：

import 'net.hasor.dataql.fx.basic.CollectionUdfSource' as collect;

// 数据

var dataList = ...

var result = collect.filter(dataList, (dat) -> { // lambda 写法

return dat.age > 20;// 年龄过滤条件

});

通过Lambda 模拟 for 循环

import 'net.hasor.dataql.fx.basic.CollectionUdfSource' as collect;

var map = {

"a" : 123,

"b" : 321

}

var data = [

{

"name" : "马三",

"type" : "a"

},

{

"name" : "n2",

"type" : "b"

}

]

var appendData = (data) -> {

var newMap = collect.newMap(data);

run newMap.put('type',map[data.type])

return newMap.data()

};

return data => [

appendData(#)

]

查询结果:

[

{

"name": "马三",

"type": 123

},

{

"name": "n2",

"type": 321

}

]

使用外部函数库

DataQL 携带了一个官方标准函数库。里面提供了大量不同功能的函数，可以通过 import 语句导入然后来使用它们（FunctionX库函数）比如：通过时间函数库获取系统时间：

import 'net.hasor.dataql.fx.basic.DateTimeUdfSource' as time;

return time.now();

或者使用 json 函数库来生成 JSON 数据：

import 'net.hasor.dataql.fx.basic.JsonUdfSource' as json;

return json.toJson([0,1,2]);// "[0,1,2]"

解析 Json

import 'net.hasor.dataql.fx.basic.JsonUdfSource' as json;

return json.fromJson("{\n\t\"key\":123\n}");// {'key':123}

代码片段

在查询中将其它语言片段直接写在 DataQL 中并作为函数来使用，这个能力在 DataQL 被称作：代码片段

注：在 DataQL 中混合其它语言一起协同处理 DataQL 查询，需要定义一个片段执行器

一个典型的场景是把 SQL 语句混合在 DataQL 查询中，利用 DataQL 对 SQL 的查询结果进行二次加工。

例如

var dataSet = @@sql(item\_code) <%

select \* from category where co\_code = #{item\_code}

%>

return dataSet() => [

{ "id","name","code","body" }

]

@@sql 是 FunctionX 扩展包中提供的一组片段执行器，这个片段执行器相当于让 DataQL 有能力执行数据库的 SQL 语句。

定义

定义一个片段执行器需要，实现 net.hasor.dataql.FragmentProcess 接口（更多信息请参考开发手册）并且将其注册到 DataQL 环境中：

方式一：通过 DataQL 接口

FragmentProcess process = ...

AppContext = appContext = ...

DataQL dataQL = appContext.getInstance(DataQL.class);//获取 DataQL 接口

dataQL.addFragmentProcess("sql", process); //注册片段执行器

方式二：通过 QueryModule

FragmentProcess process = ...

public class MyQueryModule implements QueryModule {

public void loadModule(QueryApiBinder apiBinder) {

dataQL.addFragmentProcess("sql", process); //注册片段执行器

}

}

使用

定义一个片段执行器需要使用 @@xxxx(arg1,arg2,arg3,...)<% ..... %> 语法，其中：

xxxx 为片段执行器注册到名称。

(arg1,arg2,arg3,...) 为执行这个代码时传入的参数列表。如果不需要定义任何参数可以是 ()

在 <% 和 %> 之间编写的是 目标语言的代码片段。

比如在 MySQL 中插入一条数据，并返回自增的ID：

var saveData = @@sql(data) <%

insert into my\_option (

`key`,

`value`,

`desc`

) values (

#{data.key},

#{data.value},

#{data.desc}

);

select LAST\_INSERT\_ID();

%>

return saveData(${root});