VsNeo4jWriter-基于datax的neo4j 写入组件使用说明

Neo4j是主流的图数据库平台，提供图结构数据的存储、查询及计算等功能。
VsNeo4jWriter是基于Datax数据抽取框架的neo4j 写入组件，实现了通过配置的方式将结构化数据的字段映射到实体、关系、属性等图元素，并写入Neo4j库。

一、示例数据集

示例数据集包括如下表：
1、t_book（图书）
表结构定义如下：

```
CREATE TABLE `t_book` (
  `id` bigint(20) NOT NULL,
  `book_name` varchar(32) NOT NULL,
  `book_desc` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据示意如下：

id	book_name	book_desc
1	天渊	弗洛。文奇著名科幻小说
2	银河帝国	阿西莫夫著名科幻小说
3	机器人系列	阿西莫夫著名科幻小说之一
4	机器人系列-裸阳	裸阳2
5	天空的孩子	深渊系列

2、t_publisher （出版商）
表结构定义如下：

```
CREATE TABLE `t_publisher` (
  `publisher_id` int(5) NOT NULL,
  `publisher_name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`publisher_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据示意如下：

publisher_id	publisher_name
1	科幻世界
2	自然

3、t_book_publisher(出版商和图书的对应关系，多对对)
表结构定义如下：

```
CREATE TABLE `t_publisher_book` (
  `id` int(11) NOT NULL,
  `book_id` int(20) NOT NULL,
  `publisher_id` int(5) NOT NULL,
  `publish_time` varchar(19) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据示意如下：

开始事务 备注 筛选 排序

id	book_id	publisher_id	publish_time
1	1	2	1970-01-01
2	1	1	1986-01-01
3	2	2	1996-01-01
4	5	1	2010-01-01

二、配置说明

VsNeo4jWriter支持多种结构化数据到图数据的映射方式，在配置文件中对应的是"create.type"配置项：

- 实体映射（NODE):将结构化数据直接映射成实体及实体属性
- 实体关系映射（RELATIONSHIP):基于已经存在的两类实体，创建实体之间的关系
- 完全映射（FULL)：这类用例的适用场景是获取的结构化数据里已经包含了两类实体和实体之间的对应关系，因此可以基于这些数据直接生成实体以及关系。
- 自定义(QL):对于以上模式无法直接映射的场景，提供了基于自定义cypher 查询语句的实现，通过自定义的cypher语句，可以灵活地处理大部分的映射关系。

由于不同模式下的配置项存在差异，因此将配置项的说明在第三节"配置开发示例"里进行说明。

三、配置开发示例

以下是针对各类映射方式的配置DEMO。

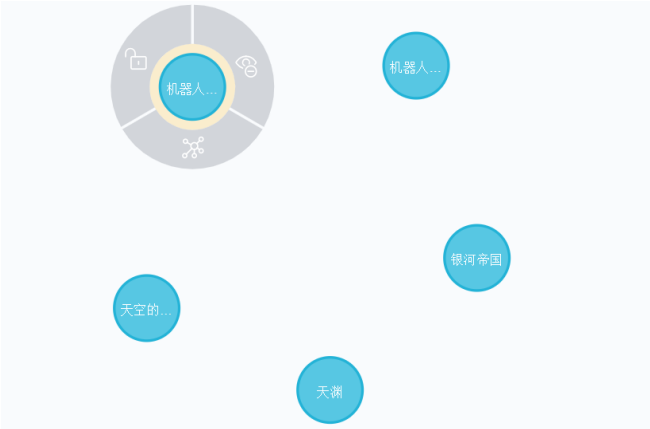
1、实体映射

本示例从t_book中生成Book实体。对应的datax job 如下：

```
{
"job": {
"setting": {
"speed": {
"channel": 5
}
},
"content": [
{
"reader": {
"name": "mysqlreader",
"parameter": {
"username": "root",
"password": "xxxx",
"where": "",
"connection": [
{
"querySql": [
"select * from t_book "
],
"jdbcUrl": [
"jdbc:mysql://mydev:3306/hyt_demo?useUnicode=true"
]
}
]
}
},
"writer": {
"name": "vsdatax-neo4jWriter",
"parameter": {
"db.uri": "bolt://localhost:7687",
```

```
"db.username": "neo4j",
"db.password": "x",
"batchSize": "100",
"column": [
"id",
"book_name",
"book_desc"
],
"create.type":"NODE",
"create.NODE.label":"BOOK",
"create.NODE.mode":"MERGE",
"create.NODE.merge.pattern":"bookId:#id",
"create.NODE.property.map":"bookName:#book_name,bookDesc:#book_desc"
}
}
}
]
}
}
```

生成结果如下：



关键配置项说明

- "create.type":"NODE"

  创建类型:单纯只创建节点

- "create.NODE.label":"BOOK"

  创建实体的Label

- "create.NODE.mode":"MERGE"

创建实体的模式，有两个值：create和merge。其中create代表创建新实体；merge 可以看成是match和create的合体：找不到实体则创建新实体，否则更新实体。

- "create.NODE.merge.pattern":"bookId:#id"

  merge的模式，即用于唯一匹配实体的关键属性

- "create.NODE.property.map":"bookName:#book_name,bookDesc:#book_desc"

字段到属性的映射关系。支持多个属性，每个属性的映射以逗号分隔；属性到字段名的映射关系以:分隔，冒号左边是实体属性，右边是字段名，字段名需要增加#作为前缀。

PUBLISHER实体创建的datax job如下：

```
{
"job": {
```

```
"setting": {
"speed": {
"channel": 5
}
},
"content": [
{
"reader": {
"name": "mysqlreader",
"parameter": {
"username": "root",
"password": "x",
"where": "",
"connection": [
{
"querySql": [
"select * from t_publisher "
],
"jdbcUrl": [
"jdbc:mysql://mydev:3306/hyt_demo?useUnicode=true"
]
}
]
}
},
"writer": {
"name": "vsdatax-neo4jWriter",
"parameter": {
"db.uri": "bolt://localhost:7687",
"db.username": "neo4j",
"db.password": "x",
"batchSize": "100",
"column": [

"publisher_id",
"publisher_name"
],
"create.type":"NODE",
"create.NODE.label":"PUBLISHER",
"create.NODE.mode":"MERGE",
"create.NODE.merge.pattern":"publisherId:#publisher_id,
"create.NODE.property.map":"publisherName:#publisher_name"
}
}
}
]
}
}
```

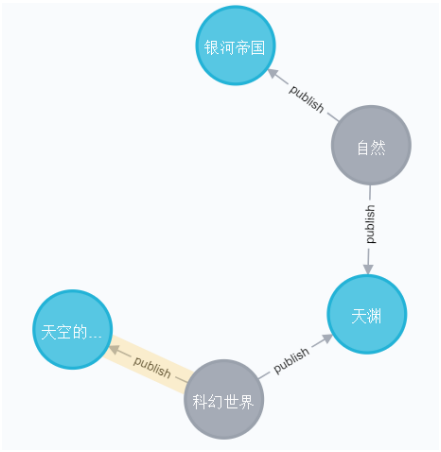
2、关系映射

这类用例的前提是已经存在两类实体数据，需要在这两类实体之间建立实体关系。从RDBMS的角度来看，对应的是用于存储关系数据的中间表。
假定已经通过前面的实体映射建立了Book和Publisher实体，本示例基于t_book_publisher表中建立实体之间的关系--publish(出版)
datax job配置如下：

```
{
"job": {
"setting": {
"speed": {
"channel": 5
}
```

```
},
"content": [
{
"reader": {
"name": "mysqlreader",
"parameter": {
"username": "root",
"password": "xx",
"where": "",
"connection": [
{
"querySql": [
"select * from t_publisher_book"
],
"jdbcUrl": [
"jdbc:mysql://172.20.240.120:3306/hyt_demo?useUnicode=true"
]
}
]
}
},
"writer": {
"name": "vsdatax-neo4jWriter",
"parameter": {
"db.uri": "bolt://localhost:7687",
"db.username": "neo4j",
"db.password": "x",
"batchSize": "100",
"column": [
"id",
"book_id",
"publisher_id",
"publish_time"
],
"create.type":"RELATIONSHIP",
"create.RELATIONSHIP.src.label":"PUBLISHER",
"create.RELATIONSHIP.src.pattern":"publisherId:#publisher_id",
"create.RELATIONSHIP.src.degree.dir":"-",
"create.RELATIONSHIP.target.label":"BOOK",
"create.RELATIONSHIP.target.pattern":"bookId:#book_id",
"create.RELATIONSHIP.target.degree.dir":"->",
"create.RELATIONSHIP.label":"publish",
"create.RELATIONSHIP.mode":"MERGE",
"create.RELATIONSHIP.create.property.map":"dbId:#id,publishTime:#publish_time",
"create.RELATIONSHIP.merge.property.map":"dbId:#id,publishTime:#publish_time"
}
}
}
]
}
}
```

publish关系结果如下:



关键配置说明

- "create.RELATIONSHIP.src.degree.dir":"-"

  源实体的关系方向

- "create.RELATIONSHIP.target.degree.dir":"->",

  目标实体的关系方向

- "create.RELATIONSHIP.label":"publish",

  关系的Label（名字）

- "create.RELATIONSHIP.mode":"MERGE"

  关系的创建方式：merge或者create

- "create.RELATIONSHIP.create.property.map":"dbId:#id,publishTime:#publish_time"

  新增的时候关系的属性与字段的映射关系

- "create.RELATIONSHIP.merge.property.map":"dbId:#id,publishTime:#publish_time"

  更新的时候关系的属性与字段的映射关系

3、完全映射

用例说明：有两张表，s2是d_line电路线路表，sd_xiaoqu2是小区表，其中sd_xiaoqu2的line_id指向sd_line2的主键。通过sql语句
`select * from sd_line2 a,sd_xiaoqu2 b where a.line_id=b.line_id` 可以获取到小区、线路、以及线路与小区的对应关系的结构化数据，
因此可以直接建立小区实体、线路实体、线路覆盖小区的实体关系。

- sd_line2的表结构定义：

```
CREATE TABLE `sd_line2` (
  `line_id` varchar(255) DEFAULT NULL,
  `line_name` varchar(255) DEFAULT NULL,
  `people_id` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- sd_xiaoqu2的表结构定义

```
CREATE TABLE `sd_xiaoqu2` (
```

```
  `xiaoqu_id` varchar(255) DEFAULT NULL,
  `xiaoqu_name` varchar(255) DEFAULT NULL,
  `line_id` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

datax job的配置如下：

```
{
"job": {
"setting": {
"speed": {
"channel": 5
}
},
"content": [
{
"reader": {
"name": "mysqlreader",
"parameter": {
"username": "root",
"password": "x",
"where": "",
"connection": [
{
"querySql": [
"select * from sd_line2 a,sd_xiaoqu2 b where a.line_id=b.line_id"
],
"jdbcUrl": [
"jdbc:mysql://mydev:3306/hyt_demo?useUnicode=true"
]
}
]
}
},
"writer": {
"name": "vsdatax-neo4jWriter",
"parameter": {
"db.uri": "bolt://localhost:7687",
"db.username": "neo4j",
"db.password": "x",
"batchSize": "100",
"column": [
"line_id",
"line_name",
"people_id",
"xiaoqu_id",
"xiaoqu_name",
"line_id1"
],
"create.type":"FULL",
"create.FULL.mode":"MERGE",

"create.FULL.src.label":"线路",
"create.FULL.src.mode":"MERGE",
"create.FULL.src.pattern":"en_line_id:#line_id",
"create.FULL.src.create.property.map":"en_line_name:#line_name",
"create.FULL.src.merge.property.map":"en_line_name:#line_name",
"create.FULL.src.degree.dir":"-",


"create.FULL.target.label":"小区",
"create.FULL.target.mode":"MERGE",
"create.FULL.target.pattern":"en_xiaoqu_id:#xiaoqu_id",
```

```
"create.FULL.target.create.property.map":"en_xiaoqu_name:#xiaoqu_name",
"create.FULL.target.merge.property.map":"en_xiaoqu_name:#xiaoqu_name",
"create.FULL.target.degree.dir":"->",

"create.FULL.label":"覆盖小区"
}
}
}
]
}
}
```

4、自定义生成语句

用例说明：以一个简化的客服工单表为例，包含电话号码、户号、工单内容、呼入时间等字段，通过该表数据可以构建出户号实体、电话实体
以及两者之间的关系：电话呼入（call）关系。
表结构定义如下：

```
CREATE TABLE `kf_worksheet` (
  `id` varchar(32) NOT NULL,
  `phone_num` varchar(20) DEFAULT NULL,
  `hh` varchar(64) DEFAULT NULL,
  `content` varchar(2000) DEFAULT NULL,
  `call_time` char(19) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

datax job配置如下：

```
{
"job": {
"setting": {
"speed": {
"channel": 5
}
},
"content": [
{
"reader": {
"name": "mysqlreader",
"parameter": {
"username": "root",
"password": "x",
"where": "",
"connection": [
{
"querySql": [
"select * from kf_worksheet "
],
"jdbcUrl": [
"jdbc:mysql://mydev:3306/hyt_demo?useUnicode=true"
]
}
]
}
},
"writer": {
"name": "vsdatax-neo4jWriter",
"parameter": {
"db.uri": "bolt://localhost:7687",
"db.username": "neo4j",
"db.password": "x",
"batchSize": "100",
```

```
"column": [
"id",
"phone_num",
"hh",
"content",
"call_time"
],
"create.type": "QL",
"create.QL.ql": "UNWIND $batches AS batch MERGE
(s:UserPhone_datax_node{phone_num:batch.phone_num}) ON CREATE SET
s.phone_num=batch.phone_num MERGE (t:HH_datax_node{hh:batch.hh}) ON CREATE SET
t.hh=batch.hh MERGE(s)-[rel:Belongs_datax]->(t) ON CREATE SET
rel.content=batch.content,rel.call_time=batch.call_time",
"create.QL.batchKey": "batches"
}
}
}
]
}
}
```

关键配置项说明

- **"create.QL.ql"**

  自定义的cypher语句

- **"create.QL.batchKey"**

对应“create.QL.ql”的cypher语句中的展开变量名。如ql里的值是$batches，则对应的值是batches.