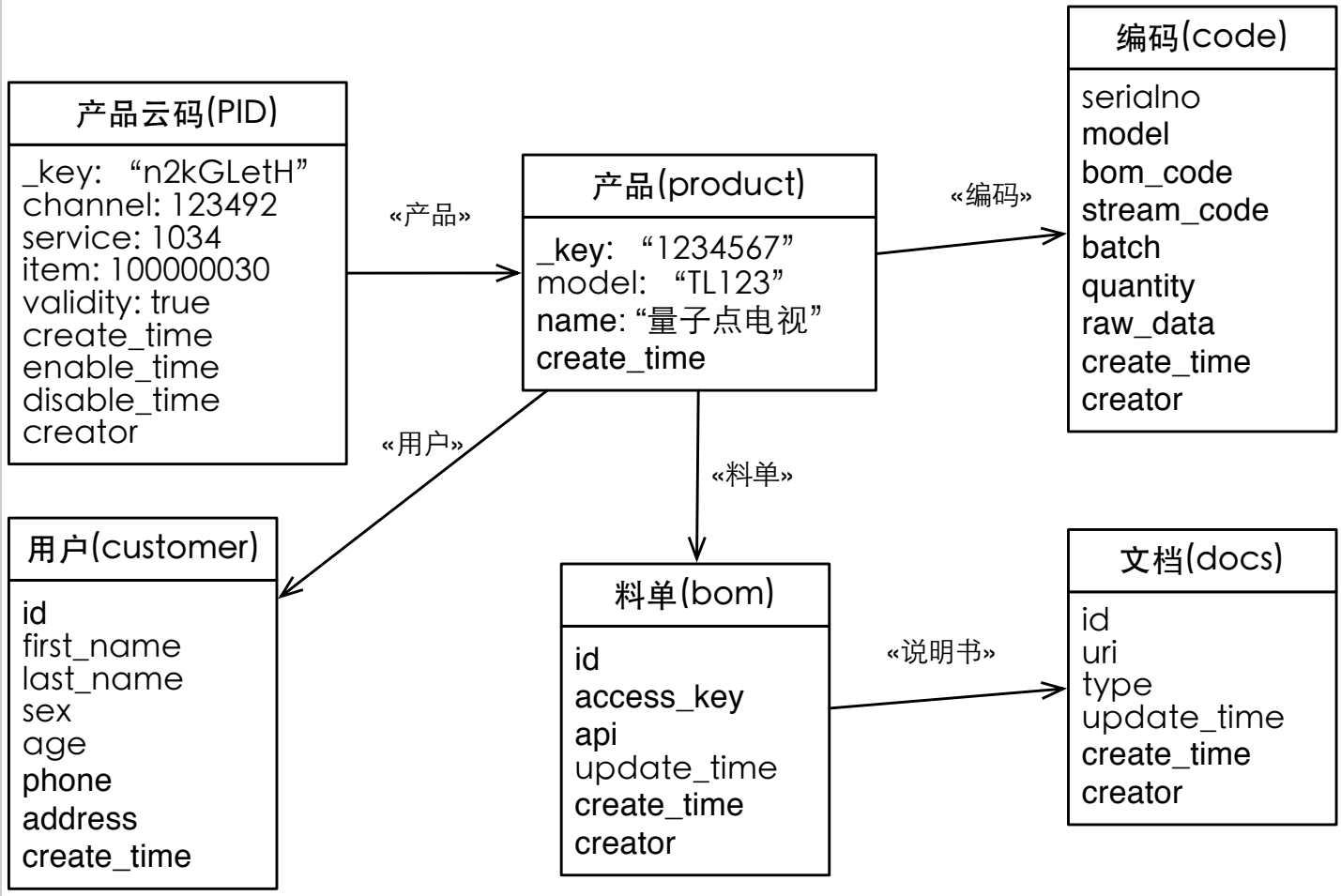
Data Engine Graph API 应用指南

数据模型



数据模型示例

上图是一个通过产品二维码检索该产品相关信息的数据模型。图中以该产品为中心，关联二维码编码信息、生产编码、零部件清单、说明书及用户信息。上图可以认为是一个复杂网状数据结构中的一个局部。

传统的数据库设计面向的作为数据集合的表进行设计，而DataEngine 采用的是Graph结构的设计，面向具体的数据实体和各个数据实体之间的关联。也就是说DataEngine数据设计强调个性，传统数据设计则强调共性。上图并不是传统数据库表及表间关联示意图，是个体之间的关系图。

DataEngine采用有向图组织数据，数据对象之间无方向的关系可以用两个方向相反、名字相同的连接等效。

应用场景是这样的，首先用户扫描产品机身上的二维码，获取该产品的二维码信息，然后以此二维码为根，采用DataEngine的寻址方式顺藤摸瓜，获取该产品各方面的信息。比如，获取该产品的说明书：`/pid/xxxxx/产品/部件/说明书`；获取购买该产品用户电话号码：`/pid/xxxxx/产品/用户.phone`

建立运行环境

安装

*arangodb*的本地安装参考 https://docs.arangodb.com/Installing/index.html 。这里不再赘述。以下假定默认安装执行文件在 `/usr/local/sbin` 。

运行

最简单的执行方式是在命令行运行：

```
unix$ /usr/local/sbin/arangod &
```

安装 DataEngine 模块

从github安装：

```
unix$ foxx-manager install git:neulinx/de /test/de
```

或者下载源码后在本地安装：

```
unix$ foxx-manager install ~/work/de /test/de
```

以后代码更新后可以用upgrade替换install命令更新DataEngine。
检查是否安装或者更新成功，运行：

```
curl -i http://localhost:8529/test/de/v1/g/_key/root
```

产生正确回应：

```
HTTP/1.1 200 OK
Server: ArangoDB
Connection: Keep-Alive
Content-Type: application/json
Content-Length: 28

"Root of data engine graph."
```

也可以在Web浏览器上输入如下地址，得出正确的回应。

```
http://localhost:8529/test/de/v1/g/_key/root
```

为了得到更丰富的错误信息，可以开启DataEngine的开发模式。在默认的生产模式下，只产生简单固定的错误回应。可以在命令行，也可以在Web图形界面中设置开发模式。

```
unix$ foxx-manager development /test/de
```

```
Activated development mode for Application de version 0.0.1 on mount point /test/de
```

建立数据集

为了兼容原有的数据库表方式，同时为了方便同类数据的组织与检索，我们并没有把所有数据统一存放在同一个key-value数据集中，而是按照上图所示分置在不同的数据集，每个数据集有一个大致相同的数据模板，即*scheme*，每个数据集可以建立各自的数据索引。

DataEngine设计中并不限定原始数据对象的来源，可以来自于传统关系型数据库表，也可以是本地文件或者来自互联网。当前示例只展示数据对象来自同一个*arangodb*数据库中不同数据集的情况。

设定数据库使用*arangodb*的默认数据库*_system*，然后可以通过*arangosh*在命令行创建上图中的各个数据集：

```
arangosh
arangosh [_system]>db._create("pid");
arangosh [_system]>db._create("product");
arangosh [_system]>db._create("code");
arangosh [_system]>db._create("customer");
arangosh [_system]>db._create("bom");
arangosh [_system]>db._create("docs");
```

也可以使用web图形界面建立。这里不做过多描述。

使用 *REST API* 实现数据模型

本示例以*arangodb*默认安装。中统一采用*curl*调用DataEngine REST API。数据对象及互相之间的关联可以在网页上通过Web界面看到。

在默认主根*root*上创建

首先建立本数据图的入口点，也就是图中的产品云码。

DE API中有一个默认的总根节点：`root`，可以以此默认根节点创建：

```
unix$ curl -d '{"_key": "n2kGLetH", "channel": "123492", "service": "1034", "item": "100000030"}' http://localhost:8529/test/de/v1/g/_key/root/n2kGLetH?s=pid
```

运行上面的命令会发起一个`POST`请求到*DataEngine*。*DE*收到请求后，首先根据`s=pid`参数，在*pid*数据集中创建一个*document*，内容是：

```
{"_key": "n2kGLetH",
```

```
    "channel": "123492",
     "service": "1034",
     "item": "100000030"
   }
```

然后*DE*在内部*Graph*中创建一个*node*，通过`{ref: "pid/n2kGLetH"}`指向新建的*document*；最后*DE*再建立一个从默认的`root`指向新建*node*的连接，这个新连接内部包含一个`{name: "n2kGLetH"}`字段。

上述命令顺利执行后，就可以通过`http://localhost:8529/test/de/v1/g/_key/root/n2kGLetH`访问到`pid`数据集里的二维码对象：

```
{
 "_id":"pid/n2kGLetH",
 "_key":"n2kGLetH",
 "_rev":"251197426126",
 "channel":"123492",
 "service":"1034",
 "item":"100000030"
 }
```

创建新的根域

为了方便后期的数据挖掘工作，*DE*底层的*Graph*数据节点需要尽量保持图的连通性，通常会以默认的总根节点`root`作为主根或者全局命名空间。不过考虑到有些企业对业务数据非常敏感，需要对其产生的数据进行严格隔离，此时可以专门为该企业建立数据孤岛。这个数据孤岛通常也需要先建立一个域根或者说该企业自己的命名空间。此时可以使用*POST*方法中的特殊用法。

例如，建立一个独立的neulinx节点作为域根：

```
unix$ curl -d '{"_key": "neulinx", "type": "_self", "data": "Realm of Neulinx"}' http://localhost:8529/test/de/v1/g/._
```

```
{"_key":"neulinx","type":"_self","data":"Realm of Neulinx","_id":"test_de_nodes/neulinx"}
```

然后可以按照上面以`root`为根创建数据对象的方法以`neulinx`为根节点开始生长(populate)后面的分支节点。

建立二维码命名空间

首先建立一个二维码的内部数据节点作为命名空间，用于指向所有的二维码编码对象。

```
unix$ curl -d '{"type": "_self", "data": "namespace of QRCode"}' http://localhost:8529/test/de/v1/g/_key/neulinx/二维码?s=.
```

```
{"_id":"test_de_links/252139637198","_rev":"252139637198","_key":"252139637198"}
```

创建**内部数据对象**其实就是在*node*节点的`data`字段直接创建数据。与创建外部数据对象不同的首先是`s=数据集`参数，此时**数据集**参数为`.`，而不是外部数据集名称。其次内部数据字段是系统定义的`node`节点数据结构，其中`type="_self"`，`data`字段则可以是任意类型数据。

想要显示刚刚建立的`二维码`节点内部数据，可以发起如下HTTP GET请求：

```
unix$ curl http://localhost:8529/test/de/v1/g/_key/neulinx/二维码?s=.
```

```
{"_id":"test_de_nodes/252139440590","_key":"252139440590","type":"_self","data":"Namespace
of QRCode"}
```

更进一步，如果想获取`二维码`节点的`link`内部数据结构，可以运行如下命令：

```
unix$ curl http://localhost:8529/test/de/v1/g/_key/neulinx/二维码?s=..
```

```
{"_id":"test_de_links/252139637198","_key":"252139637198","_rev":"252139637198","_from":"te
st_de_nodes/neulinx","_to":"test_de_nodes/252139440590","name":"二维码"}
```

链接已经存在的二维码节点

如果想把刚才已经建立的产品二维码`pid/n2kGLetH`链接到`neulinx/二维码`节点上，则可以使用如下命令：

```
unix$ curl -d '{"_ref": "pid/n2kGLetH"}' http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH?s=..
```

```
{"_id":"test_de_links/252412922318","_rev":"252412922318","_key":"252412922318"}
```

这个命令会链接`neulinx/二维码`到`pid/n2kGLetH`。需要注意的是，HTTP POST BODY中的`{"_ref": "pid/n2kGLetH"}`并不会创建新的*node*，而是把上面已经创建的指`"pid/n2kGLetH"`的数据节点链接进来。根据DE API参考手册，`curl -d '{"_ref": "pid/n2kGLetH"}'`等效`curl -d '{"_key": "test_de_nodes/251197622734"}'`，其中`test_de_nodes/251197622734`是原来已经创建的二维码n2kGLetH对象的**指向节点**。

创建产品数据对象

二维码*n2kGLetH*指向产品序列号为**1234567**，名称为**量子点电视**的具体产品。创建该产品的*curl*命令如下：

```
unix$ curl -d '{"_key": "1234567", "name":"量子点电视", "model": "TL123"}' http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品?s=product
```

```
{"_id":"test_de_links/253745269198","_rev":"253745269198","_key":"253745269198"}
```

此项命令会在前面创建的`product`数据集中新建一个以产品序列号为`_key`的新数据对象。在浏览器地址栏中输入`http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品`就可以访问到这个产品的数据。


```
{"_id":"product/1234567","_key":"1234567","_rev":"253744810446","model":"TL123","name":"量子点电视"}
```


如果嫌路径太长，也可以通过原始数据`id`来访问：
`http://localhost:8529/test/de/v1/g/product/1234567`
得到相同的数据。

创建编码数据对象

根据产品生产过程中产生的各种编码，可以用上面短路径建立该产品编码数据：

```
unix$ curl -d '{"serialno":"1234567", "model": "TL123", "bom_code": "bc12345", "batch": "7890", "quantity": 20000}' http://localhost:8529/test/de/v1/g/product/1234567/编码?s=code
```


```
{"_id":"test_de_links/253977594318","_rev":"253977594318","_key":"253977594318"}
```


用全路径访问该产品的编码：

```
unix$ curl http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品/编码
```


```
{"_id":"code/253977201102","_key":"253977201102","_rev":"253977201102","quantity":20000,"model":"TL123","batch":"7890","serialno":"1234567","bom_code":"bc12345"}
```


如果只想获取**批次**和**数量**数据，则可以在`s=数据选择`参数中设置：

```
unix$ curl http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品/编码?s=batch,quantity
```


```
{"batch":"7890","quantity":20000}
```


创建料单数据对象

假定料单是通过一个需要授权的*HTTP REST API*接口获取的，`bom`数据集中的数据对象如下创立：

```
unix$ curl -d '{"id":"bc12345", "access_key": "45a6b7d", "api": "https:/www.example.com/bom/_api"}' http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品/料单?s=bom
```


```
{"_id":"test_de_links/254121576910","_rev":"254121576910","_key":"254121576910"}
```

获取一下料单编号的访问码 `access_key` 信息：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/料单?s=access_key
```

```
{"access_key":"45a6b7d"}
```

创建说明书数据对象

说明书数据可以是一个*pdf*文件，也可以是一个*URL*指向的网站，也可以是本地目录下的一组图片文件。具体创建方式如下：

```
unix$ curl -d '{"id": "1235", "uri":"file:///var/docs/manual/1235", "type": "images"}'
http://localhost:8529/test/de/v1/g/product/1234567/料单/说明书?s=docs
```

```
{"_id":"test_de_links/254332013006","_rev":"254332013006","_key":"254332013006"}
```

获取说明书数据对象的全路径调用如下：

```
unix$ curl http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品/料单/说明书
```

```
{"_id":"docs/254331619790","_key":"254331619790","_rev":"254331619790","id":"1235","type":"
images","uri":"file:///var/docs/manual/1235"}
```

创建用户数据对象

命令行执行*HTTP POST*如下：

```
unix$ curl -d '{"id": "123456789012345", "last_name": "张", "first_name": "三丰", "sex":
"        男        ",        "age":        100,        "phone":        "12345678901"}'
http://localhost:8529/test/de/v1/g/product/1234567/用户?s=customer
```

```
{"_id":"test_de_links/254417209806","_rev":"254417209806","_key":"254417209806"}
```

获取用户的 `id` 和 `phone` 信息的请求如下：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/用户?s=id,phone
```

```
{"id":"123456789012345","phone":"12345678901"}
```

修改用户信息

可以通过*HTTP PUT*命令修改数据对象内容。如修改该产品用户的年龄 `age` 为99岁，`_key` 为用户的 `id` 值：

```
unix$ curl -X PUT -d '{"_key": "123456789012345", "age": 99}'
```

`http://localhost:8529/test/de/v1/g/product/1234567/用户`


```
{"_id":"customer/254416816590","_rev":"255978998222","_oldRev":"254416816590","_key":"25441
6816590"}
```


获取更新后的用户信息：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/用户?s=age
```


```
{"age":99}
```


修改路径信息

如果想把指向 `customer` 的路径名称从 `用户` 改为 `客户` ，则需要给定 `s` 参数为 `s=..` 。

```
unix$ curl -X PUT -d '{"name": "客户"}' http://localhost:8529/test/de/v1/g/product/1234567/用
户?s=..
```


```
{"_id":"test_de_links/254417209806","_rev":"256049187278","_oldRev":"254417209806","_key":"
254417209806"}
```


此时再用旧的路径 `用户` 获取信息时，将会报如下错误：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/用户?s=age
```


```
HTTP/1.1 404 Not Found
Server: ArangoDB
Connection: Keep-Alive
Content-Type: application/json
Content-Length: 36

{"error":"The route is not viable."}
```


用修改后的路径则能获得正确的结果：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/客户?s=age
```


```
{"age":99}
```


用相对寻址方式建立新的路径

上述数据模型图访问**说明书**时，需要这样的路径： `/product/1234567/料单/说明书` ，可以建立一个从 `产品` 直接指向 `说明书` 的快捷路径：

```
unix$ curl -d '{"_path": "./料单/说明书"}' http://localhost:8529/test/de/v1/g/_key/neulinx/二
维码/n2kGLetH/产品/说明书?s=..
```

{"_id":"test_de_links/256253397454","_rev":"256253397454","_key":"256253397454"}

用缩短后的路径访问 `说明书`：

```
unix$ curl http://localhost:8529/test/de/v1/g/product/1234567/说明书
```

{"_id":"docs/254331619790","_key":"254331619790","_rev":"254331619790","id":"1235","type":"images","uri":"file:///var/docs/manual/1235"}

删除操作

客户数据对象建立后，其 `_key` 值为只读属性，当我们想修改 `_key` 属性时，只能先把原来的数据对象删除。删除操作使用*HTTP DELETE*方法：

```
unix$ curl -X DELETE http://localhost:8529/test/de/v1/g/product/1234567/客户
```

{"success":true}

然后重新建立：

```
unix$ curl -d '{"_key": "123456789012345", "id": "123456789012345", "last_name": "张", "first_name": "三丰", "sex": "男", "age": 99, "phone": "12345678901"}' http://localhost:8529/test/de/v1/g/product/1234567/用户?s=customer
```

{"_id":"test_de_links/256692619726","_rev":"256692619726","_key":"256692619726"}

然后想再把 `link` 名称从 `用户` 改为 `客户`，不过这次先删除二者的*链接*关系，再重新建立：

```
unix$ curl -X DELETE http://localhost:8529/test/de/v1/g/product/1234567/用户?s=..
```

{"success":true}

```
unix$ curl -d '{"_ref": "customer/123456789012345"}' http://localhost:8529/test/de/v1/g/product/1234567/客户?s=..
```

{"_id":"test_de_links/256761498062","_rev":"256761498062","_key":"256761498062"}

执行成功后，产品 `product/1234567` 与客户 `customer/123456789012345` 之间的链接路径就变为 `product/1234567/客户` 。

上面的例子是删除链接关系，当然，我们也可以通过删除*node*来完成同样的事情，只不过后者系统会重新建立新的*node*指向客户原始的数据对象，前者则不会删除指向客户数据的*node*。需要注意的是，删除*node*的同时也会删除该*node*所有的*link*。与删除链接的操作相比，只有 `s` 参数不同，一个

是`.`，另一个是`..`。具体的操作命令如下：

```
unix$ curl -X DELETE http://localhost:8529/test/de/v1/g/product/1234567/客户?s=.
```

```
{"success":true}
```

这个操作并没有删除原始数据对象`customer/123456789012345`，还可以在数据集查到该客户。当然删除操作也可以直接针对原始数据。删除原始数据对象的同时，也会删除该数据对象在*Graph*中的参考节点，并删除该节点所有链接关系。

除了可以使用相对路径`/v1/g/product/1234567/客户`定位待删除的对象外，还可以直接使用原始数据对象的*document handle*来寻址。如下命令所示，*DE*系统并未直接操作`customer`数据集，而是通过其*document handle*反查得到。因此由于没有通过*node*指向，下面的命令会报错：

```
unix$ curl -X DELETE http://localhost:8529/test/de/v1/g/customer/123456789012345
```

```
{"error":"The route is not viable."}
```

重新建立与*Graph*的关联后，就可以采用上述命令删除。下面的例子是直接建立一个“孤儿”*node*指向该客户，然后执行删除。

```
unix$ curl -d '{"ref": "customer/123456789012345"}' http://localhost:8529/test/de/v1/g/._
```

```
{"ref":"customer/123456789012345","_id":"test_de_nodes/257704692174","_key":"257704692174"}
```

```
unix$ curl -X DELETE http://localhost:8529/test/de/v1/g/customer/123456789012345
```

```
{"success":true}
```

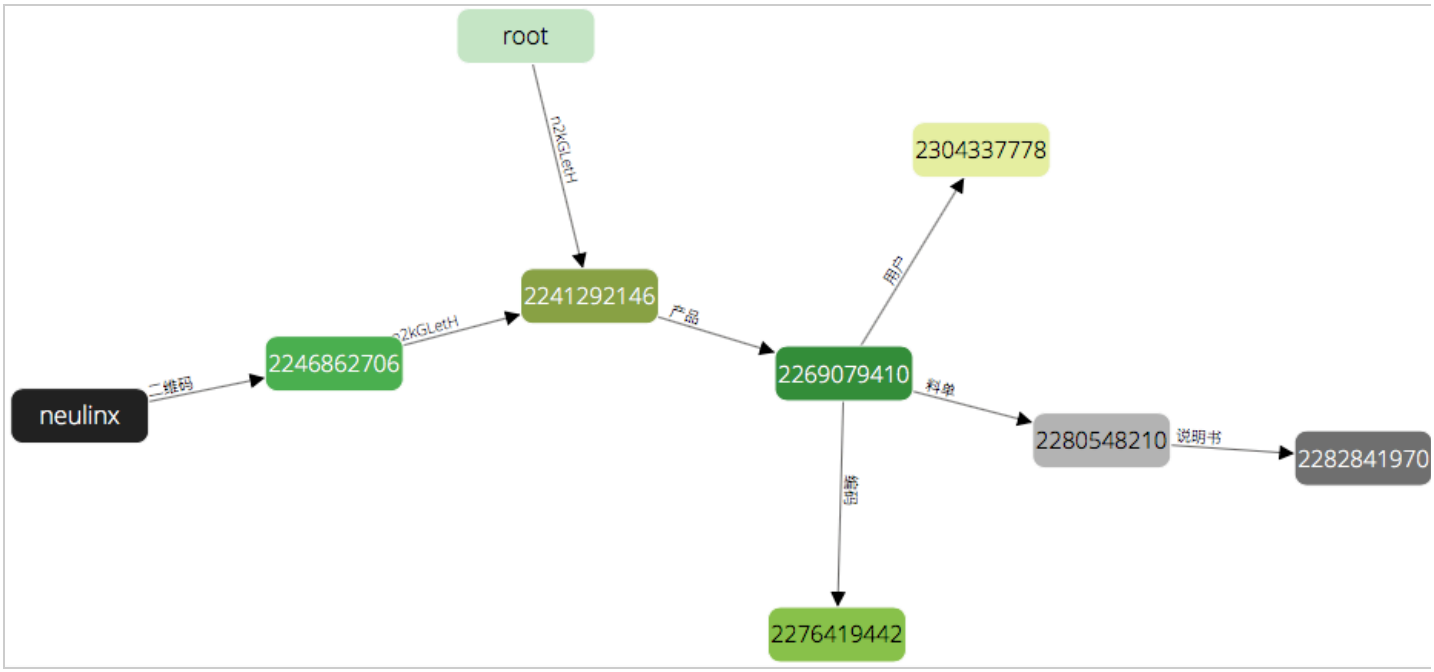
图形化当前*Graph*

重建客户数据对象：

```
unix$ curl -d '{"_key": "123456789012345", "id": "123456789012345", "last_name": "张",
"first_name": "三丰", "sex": "男", "age": 99, "phone": "12345678901"}'
http://localhost:8529/test/de/v1/g/product/1234567/用户?s=customer
```

```
{"_id":"test_de_links/257869842894","_rev":"257869842894","_key":"257869842894"}
```

然后可以在Web图形界面中得到如下的**关系图**：



示例图

图中对应的*node*对象：

参考节点对象数据	唯一标示
{“type”：“_self”，“data”：“Realm of Neulinx”}	neulinx
{“ref”：“pid/n2kGLetH”}	251197622734
{“type”：“_self”，“data”：“Namespace of QRCode”}	252139440590
{“ref”：“product/1234567”}	253745007054
{“ref”：“code/253977201102”}	253977397710
{“ref”：“bom/254121183694”}	254121380302
{“ref”：“docs/254331619790”}	254331816398
{“ref”：“customer/123456789012345”}	257869646286

网页模板的渲染测试

为了方便，使用arangodb的图形界面在 collections 中找到 docs/254331619790, 添加一个`_template`字段，填写如下内容：

```
<html>
  <head>
    <title>Hello {{model}}</title>
  </head>
  <body>
    <p>第一层：产品名称：{{name}}，产品型号：{{model}}</p>
    <p>第二层：{{#locate "./用户"}}买家是{{last_name}}{{first_name}}{{/locate}}</p>
    <p>第三层：{{#locate "./料单/说明书"}}说明书下载地址：{{uri}}{{/locate}}</p>
```

```
      <p>嵌套:
        {{#locate "./料单"}}access key:{{access_key}}
          {{#locate "./说明书"}} 说明书类型:{{type}}{{/locate}}
        {{/locate}}
      </p>
      <p>直接访问说明书类型数据: {{locate "料单/说明书" "type"}} </p>
      <p>直接访问产品用户的手机号码: {{locate "/_key/neulinx/二维码/n2kGLetH/产品/用户" "phone"}}
  </p>
    </body>
  </html>
```

通过HTTP GET访问 http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品?r=/docs/254331619790

渲染后的数据为:

```
  <html>
    <head>
      <title>Hello TL123</title>
    </head>
    <body>
      <p>第一层: 产品名称: 量子点电视, 产品型号: TL123</p>
      <p>第二层: 买家是张三丰</p>
      <p>第三层: 说明书下载地址: file:///var/docs/manual/1235</p>
      <p>嵌套:
        access key:45a6b7d
           说明书类型:images
      </p>
      <p>直接访问说明书类型数据: images </p>
      <p>直接访问产品用户的手机号码: 123 </p>
    </body>
  </html>
```

。还可以由前端推送动态模板内容到后端进行渲染，从而提高复杂数据的访问效率。例如:

`curl -d '购买该产品的用户是: {{last_name}}{{first_name}}' http://localhost:8529/test/de/v1/g/_key/neulinx/二维码/n2kGLetH/产品/用户?r`

返回: `购买该产品的用户是: 张三丰`