图数据库项目开发说明

Graph Database Project Specification

# 1. 开发目的

1). 设计出可以供桌面应用开发使用的轻量级数据库；

2). 在设计过程中完善图数据库理论和实践经验；

3). 拓展图数据库的应用领域。

# 2. 项目划分

## 2.1 内核开发

目标：参照Neo4j数据库的基本架构，利用C#平台实现图数据库。

功能（任务）列表：

1、图为节点和连边的宿主元素，承担主调任务，节点类和连边类提供底层辅助功能；

2、数据库手工建立：

1）、图的建立与成员初始化；

2）、节点的插入、删除，与属性的插入、删除和修改；

3）、连边的插入、删除，与取值修改；

3、数据批量导入：

1）、XML文件的读取；

2）、XML文件的保存；

产出：最后完成一个可以存储数据的图数据库类库，能够实现数据逐条或批量的增、删、改操作。

## 2.2 可应用最小系统

目标：在前一阶段基础上，实现图数据库的查找功能，设计高效的查找引擎。

功能（任务）列表：

1、分析Neo4j查询语言的基本语法，并分解查找工作流程；

2、在GraphDB中实现查找算法；

3、能够通过类似Neo4j的语言（简化后）对图数据库进行查找。

产出：在类库中增加查找功能接口。

## 2.3 数据库管理系统

目标：在之前完成的数据库类库的基础上，实现一个数据库管理系统，提供可视化的操作界面。

功能（任务）列表：

1、单一操作，能够建立数据库，并逐条插入、修改和删除数据；

2、批量操作，对已建立的数据库，实现批量插入、修改和删除操作；

3、提供文字化的查询功能，并通过文字或列表返回查询结果；

4、实现可视化效果，图数据库的结构显示和图形化查询功能。

产出：一个可运行软件，可以图形化或文字化地对图数据库进行逐条或批量数据的增删改查操作。

# 3. 模块设计

## 3.1 内核结构

内核结构说明，如图3.1所示：

1、图包含一个节点列表和一个连边列表；

2、节点包含一个名称，一个类型和若干个（0-n）个属性三个主体部分，另外包含一个出边列表和一个入边列表。

3、连边包含一个类型、一个取值构成的主体部分、一个起始节点指针和一个终止节点指针。

4、节点的属性列表是由若干个key和value的键值对组成，可以由用户自主添加。

5、节点入边和出边列表中内容是对应边的指针数组或引用数组。

6、连边的起始和终止节点指针式对应节点的指针或引用。



图3.1 内核元素结构示意图

## 3.2 数据库创建与维护

### 3.2.1 数据库创建与数据插入

数据库创建流程说明：

1、主调函数调用Database类提供的操作接口CreateDatabase（）创建数据库对象；

2、Database类调用Graph类的构造函数，创建一个图对象；

3、创建成功则返回数据库对象给调用者，否则返回错误原因。

数据节点插入流程：

1、主调函数调用Database类接口AddNodeData()；

2、Database类调用Graph类接口AddNode()；

3、Graph类调用Node构造函数创建一个新数据节点；

4、将Node对象加入Graoh中节点列表，并逐层返回OK；否则逐层返回失败原因。

关系连边插入流程：

1、主调函数调用Database类接口AddEdgeData()；

2、Database类调用Graph类接口AddEdge()；

3、Graph类调用Edge构造函数创建一个新数据节点，创建成功则返回该Edge对象；

4、Graph类调用起始节点Node对象的AddEdge函数和终止节点Node对象的RegisterInbound函数完成连边在节点间的注册；

5、将Edge对象加入Graoh中连边列表，并逐层返回OK；否则逐层返回失败原因。



图3.2 数据库创建、数据插入、关系插入操作序列图

### 3.2.2 节点和连边内部数据修改



图3.2节点和连边内部信息修改操作序列图

如图3.2所示，节点和连边的内部数据相互独立，不存在依赖关系，所以依次调用接口函数进行修改即可，详细流程在此不做赘述。

*注意：数据修改是指修改节点和连边内部的附加信息，但节点内部的核心信息不能改动。如果需要修改节点和连边的和信息，则需要对节点或连边进行删除，随后重新创建后加入。*

### 3.2.3 节点和连边的删除

关系连边删除流程，如图3.3：

1、主调函数传入待删除连边起始节点和终止节点的名称、类型以及连边类型数据；

2、Graph类首先查找连边的起始节点和终止节点，以及节点之间的同类型边；

3、随后从起始节点的outbound列表和终止节点的inbound列表中分别注销目标连边；

4、最后从Graph类的EdgeList中将目标连边移除，完成后返回OK；否则返回失败原因。



图3.3连边删除操作序列图

数据节点删除流程，如图3.4：

1、主调函数传入待删除节点的名称、类型；

2、Graph类在节点列表中查找匹配的节点；

3、Graph类调用目标节点的成员函数ClearEdge，清除该节点有关的所有连边；

4、节点内部将inbound中所有连边从其起始节点的outbound中注销，将outbound中所有连边从其目标节点的inbound中注销，并将这些连边都加入待清除列表，返回给Graph类。

5、Graph类从EdgeList中删除列表所有记录下来的连边，从NodeList中删除目标节点；

6、Graph类删除完成后返回OK；否则返回出错原因。



图3.4节点删除操作序列图

## 3.3 文件读取与存储

### 3.3.1读取XML文件

XML数据导入流程，如图3.5所示：

1、主调函数调用InitData接口启动导入流程；

2、DataBase对象清除内部数据，并创建新的Graph对象；

3、Graph对象从文件中取出xml数据并调用Node构造函数，生成节点加入NodeList；

4、Graph对象从文件中取出xml数据并调用Edge构造函数；

5、对于每条连边，Graph根据文件中编号找到其起始和终止节点；

6、将起始节点和终止节点地址写入连边内部的两个指针，最后在两个节点中注册该条边；

7、将当前边加入EdgeList，并继续，直到完成所有节点和连边的生成。



图3.5 读取XML文件导入数据序列图

### 3.3.2 输出XML文件

数据导出到XML文件流程，如图3.6所示：

1、由上层主调函数提供待导出文件路径；

2、Database调用Graph的ToXML函数构造XML文档；

3、Graph类调用AdjustNodeIndex重新给节点依次赋予编号（之前如果有删除操作，会导致原有编号之间存在空值）；

4、Graph类分别遍历节点列表和连边列表，并调用各个节点和连边的成员函数ToXML生成各自的XML节点；

5、Graph类根据顺序将返回的XML节点组织成XML文档，并返回DataBase；

6、Database类将XML文档保存到指定路径，并返回OK；否则返回出错原因。



图3.6 保存数据到XML文件序列图

## 3.3 数据查询

### 3.3.1 查询语言解析器

The query language is comprised of several distinct clauses.

* START: Starting points in the graph, obtained via index lookups or by element IDs.
* MATCH: The graph pattern to match, bound to the starting points in START.
* WHERE: Filtering criteria.
* RETURN: What to return.
* CREATE: Creates nodes and relationships.
* DELETE: Removes nodes, relationships and properties.
* SET: Set values to properties.
* FOREACH: Performs updating actions once per element in a list.
* WITH: Divides a query into multiple, distinct parts.

其中查询相关的是START,MATCH, WHERE,RETURN

新建相关的是CREATE, SET

删除相关的是 START, MATCH, WHERE, DELETE

修改相关的是START, MATCH, WHERE, SET, FOREACH

目前增删改操作都通过代码直接完成，只将查询相关谓词加入支持的关键词列表。即 START, MATCH, WHERE, RETURN。

整个查询流程借鉴Neo4j的Cypher查询语言，使用START, MATCH, WHERE, RETURN四个谓词组织语句，执行查询操作。

START负责设置搜索起点，通过Index和Name+Type两种方法定位到目标节点。取值可以为\*，即全图搜索。

MATCH为设置搜索规则链。包括目标节点和目标连边两部分组成，数量不限。但要求是一条完整的关联路径。

WHERE为结果过滤条件，格式为“变量名.属性+操作符(=<>!=)+取值”

RETURN为显示内容，格式为“变量名.属性” 最终结果以列表形式返回。

如图3.7所示的一个Cypher查询语句。

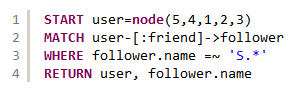


图3.7 Cypher查询语句示例

查询语句解析算法如下：

Step1. 查询语句输入后，解析器将语句拆分为功能段，START字句为起始节点选择，MATCH语句为查询规则表， WHERE为结果过滤， RETURN为返回结果列表。

Step2. 首先根据START语句查找节点，填充起始节点列表List<Node> Start；

Step3. 将MATCH子句拆分为n个变量和n-1条规则，规则由连边类型，连边取值和连边方向三部分组成。规则存入List<Rule> match中， 同时初始化结果列表List<Result>；

Step4. WHERE同样也拆分为规则列表，放入List<Rule> filter中；

Step5. 根据RETURN子句，在结果列表的对应行处加入标记，标示需要输出的内容；

Step6. 从起始列表Start中遍历节点，执行深度优先搜索，搜索成功的结果记录到结果列表中。

Step7. 搜索完毕后根据List<Rule> filter中规则执行结果过滤；

Step8. 根据之前RETURN语句做的标记输出最终结果。

### 3.3.2 有关类和数据结构

#### 1.起始节点规则类StartRule

START子句分析完成后提供一个StartRule对象，存储用户输入的起始节点筛选规则。

public class StartRule

{

string strMode;//"Index", "NameType"

bool isAll;

List<string> subRule;//子规则

public StartRule(string strSub);

public bool Match(Node tarNode);

bool MatchNumber(string strRule, int index);

bool MatchNameType(string strRule, string strNameType);

}

#### 2.中间变量类MidResult

MATCH子句分析完成后提供一组MatchRule规则对象和一个中间变量列表List<MidResult>。表示查找路径深度从0到n的各个层次。每个层次都由MATCH语句赋予一个临时变量名，给后续的WHERE和RETURN语句使用。后续的WHERE语句通过该变量名，找到对应的层次深度。而RETURN语句则在MidResult对象中标示出最终输出的元素和标签名。

public class MidResult

{

string strName;//名称，由MATCH语句指定

int level;//层次，处于树状图的深度，由MATCH语句指定

bool IsFinalResult;//由RETURN语句指定，如果为true，则strProperty被初始化

List<string> strProperty;//由RETURN语句指定，待输出的内容标签

}

#### 3.连边关系匹配规则类MatchRule

MATCH子句分析完成后提供一组MatchRule规则对象和一个中间变量列表。其中MatchRule对象是每条边的筛选规则，包括方向，类型和数量限制三部分。

public class MatchRule

{

string direction; //"IN","OUT"

bool isAll;

List<string> labels;//:label1|:label2

bool NumLimit;//

int uplimit;

int downlimit;

public MatchRule(string );

bool MatchType(string sType);

bool MatchCount(int iNodes);

}

#### 4.节点过滤规则类FilterRule

节点过滤规则用于对筛选完成后的结果进行过滤，对于搜索结果树中不满足要求的节点，不予输出。

public class FilterRule

{

string strName；//临时变量名

int level；//深度

string strProperty;//目标标签

string op;//比较操作符

string strValue;//比较操作数

}

#### 5.搜索树类SearchTree

使用C#内置的TreeNode类型，用于在MATCH匹配阶段存放深度优先搜索返回的结果。WHERE阶段会根据指定的深度对节点进行筛选，并返回RETURN子句指定的节点信息。

### 3.3.3 深度优先搜索算法

TreeNode Search(MatchRule ruleArray[],int level,int len)

{

List<Node> SearchList;

TreeNode CurrentTN, ChildTN;

CurrentTN = new TreeNode(this.intNodeNum.toString());

if(level == len)

{//到底

return CurrentTN;

}

if(Rule[level].direction == "IN")

{

SearchList = this.Inbound;

}

else

{

SearchList = this.OutBound;

}

foreach(Edge edge in SearchList)

{

if(rullArray[level].MatchType(Edge.Type) == true)

{

ChildTN = edge.End.Search(ruleArray[], level+1, len);

if(ChildTN != null)

{

CurrentTN.Nodes.Add(newTN);

}

}

}

if(Rule[level].MatchCount(CurrentTN.Nodes.Count) == true)

{

return CurrentTN;

}

return null;

}

## 3.4 查询语言解析器

### 3.4.1 START子句规范

START子句用于指定搜索的起始节点。可以通过两种方式指定，节点索引，节点“名称+类型”。

详细的方式举例如下：

START node(0) 通过索引指定一个节点

START node(0,11,34) 通过索引指定多个节点

START node(0, 2..5, 14..20) 通过索引指定连续多个节点

START node(‘A-B’)通过名称+类型指定一个节点

START node(‘A1-B1; A2-B2’) 通过名称+类型指定多个节点

START node(‘A1-\*; A2-B2, \*-B3’) 通过名称+类型和通配符指定多个节点

### 3.4.2 MATCH 子句规范

MATCH子句用于指定搜索规则，即搜索路径上的各种约束条件。可以看作是连续的节点和连边的关联路径。类似于“节点-连边-节点-连边-节点”。

MATCH子句可以分为三部分，节点、连边和箭头。节点是用于存储数据的中间变量，连边用于描述查找关系的具体要求。箭头则是选取出边或入边的依据。

节点用()包裹，可以是用户指定的任何字母+数字组合，该名称存放于结果列表中，用于标示用户期望的输出。

连边具有两种方向，<-[\*]- 和-[\*]-> 表示节点的出边或入边，方括号[]表示边的筛选条件，如果不设置条件则为\*。

以下为MATCH子句的几种可以接受的书写样例：

MATCH (n)-[\*]->(m) 表示n指向m的任意类型的边

MATCH (n)-[:r]->(m) 表示n指向m的类型为r的边

MATCH (n)-[:r1 | :r2]->(m) 表示n指向m的类型为r1或r2的边

MATCH (n)-[\*1..5]->(m) 表示n指向m的任意类型的边,数量要求在1-5之间。

MATCH (n)-[ :r1 | :r2 | 1..5]->(m) 表示n指向m的任意类型的边,数量要求在1-5之间。

### 3.4.3 WHERE子句规范

WHERE子句用于对MATCH搜索完成后返回的结果树进行过滤，对于不符合条件节点，将删除与该节点的整条树枝（branch）。即如果当前节点不符合要求，则删除当前节点及其所有子树，如果删除之后其父节点的子树为空，则删除父节点，递归向上直到root节点。

WHERE子句是由关键字加若干条规则组合而成的。包括临时变量名，标签名，比较运算符，比较运算数四部分。

以下为WHERE子句的几种可以接受的书写样例：

WHERE \* 表示不加以过滤，直接返回RETURN需要的结果

WHERE A.bb == “ccc” 单条过滤规则，临时变量A类的节点的属性bb的取值一定要是ccc。

WHERE A.bb == “ccc” AND C.dd > 5 单条过滤规则，临时变量A类的节点的属性bb的取值一定要是ccc。

### 3.4.4 RETURN子句规范

RETURN子句用于指定最终返回给用户的数据，由关键字RETURN和若干条逗号分隔的标签组成。标签的格式是“临时变量名.节点标签”。

RETURN子句解析之后，会在MidResult列表中需要输出的变量中记录标记位，并将需要输出的标签加入输出标签列表。

以下为RETURN子句的几种可以接受的书写样例：

RETURN \* 表示输出所有临时变量的标签信息。

RETURN A.b 表示输出临时变量A节点的b标签的信息

RETURN A.b, C.d 表示输出临时变量A节点的b标签的信息和C节点的d标签信息。

RETURN A.b, C.\* 表示输出临时变量A节点的b标签的信息和C节点的所有信息。

# 4. UI设计

## 4.1 UI概览

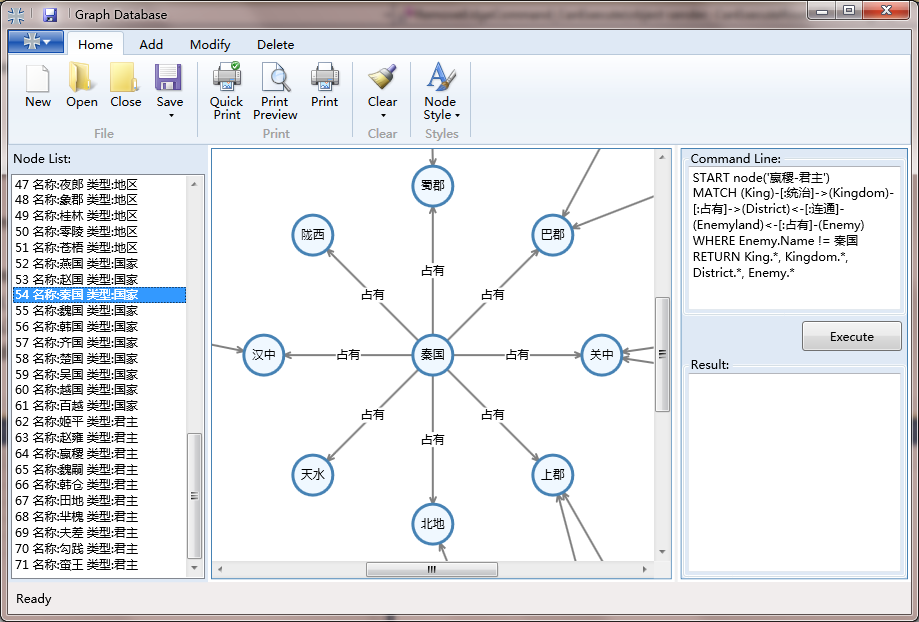


图4.1图数据库UI

### 4.1.1 UI元素

1. 功能区，采用Ribbon风格，默认位于顶部，不可移动。提供菜单工具操作按钮和其他控件，用于提供用户输入输出和数据库增删改操作。

2. 文档区，位于中心位置，用于显示以当前选中节点为中心的三层关系网络。

3. 元素列表区，位于左侧边栏，用于显示节点数据列表图。

4. 查询命令行，位于右侧边栏上部，用于输入查询语句，并执行。

5.文字结果区，位于右侧边栏下部，用于显示查询语句返回的结果。

6. 状态栏，位于底部，用于显示软件运行状态和报错信息。

## 4.2 UI功能描述

### 4.2.2文件操作

与Microsoft Access相同，GraphDB并不支持多文档操作，无法在同一个程序实例中打开多个数据库操作。但是用户可以通过打开多个GraphDB进行多数据库并行处理，相互之间不会干扰。

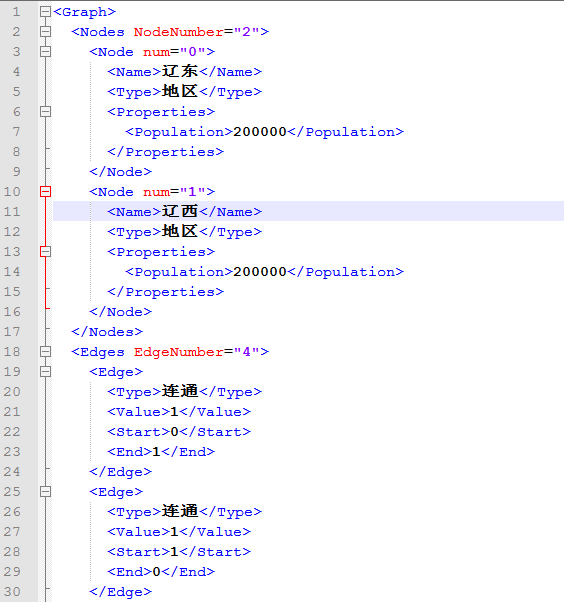


图4.2图数据库文件格式

GraphDB提供对XML格式数据库文件的新建，读取和保存。XML文件的格式如下所示，分为三层：第一层图，第二层节点集合与连边集合，第三层为节点和连边数据。

其中Node节点的Name和Type是必须不为空的字段，他们二者作为组合键唯一地标识一个节点（例如人名，国名会出现重名，需要加入类型来进一步区分，数据库中不允许出现两个Name和Type完全一样的节点）。Node内部的Properties属性集合中可以存放任意用户定义的属性标签和取值，这在节点创建的时候完全由用户决定。

Edge中存放的是连边的信息，四个标签（Type, Value, Start, End）都是必须存在的。但是Type和Value的内容完全由用户基于业务定义。 Start和End存放的是连边起始和终止节点在文件中的编号（注意该编号在进入数据库系统后可能发生变化，这依赖于数据库程序自身运行的状态）。

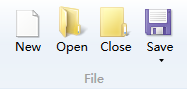


图4.3文件操作按钮组

文件控制按钮组中的新建，打开，关闭功能在当前存在数据库的情况下会提示是否保存当前数据库或者取消操作，用户可以依据需要自行选择。保存按钮分为保存和另存为两种功能，前者会保存到文件原始路径，后者则会弹出对话框提示用户选择新文件路径。

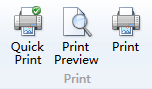


图4.4打印操作按钮组

打印功能暂时不可用，这里仅为示例。由于数据库文件本身也是文本文件，用户可以使用任何文本编辑器的打印功能来打印文件。



图4.5清除操作按钮

清除操作主要是清除右侧边栏中两个文本框的内容，其下拉箭头中有两个功能，第一个可以清除Command Line中的查询命令，第二个可以清除ResultBox中的查询结果。



图4.6绘图样式选择按钮

绘图样式选择按钮用来提供用户选择若干种内置绘图风格的功能。其下拉箭头中存放着现有集中网络图绘制风格，包括节点的底色，边框色和字符前景色。未来会提供资源配置文件提供皮肤切换功能等。

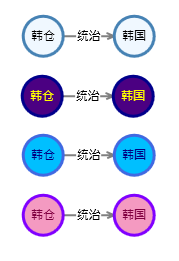


图4.7四种内置样式的效果对比

### 4.2.3 节点切换

界面的左侧边栏为节点列表，存放节点的编号，名称和类型。用户可以通过点击节点列表中的任意一项来选择对应节点为当前节点。一旦选择成功，数据库操作区中的节点信息会变为当前选中节点，而中央的绘图区会绘制以选中节点为中心的三层关系网络，如图4.9。

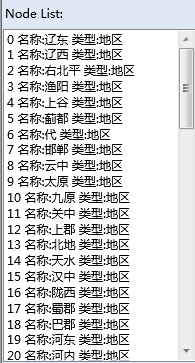


图4.8节点列表

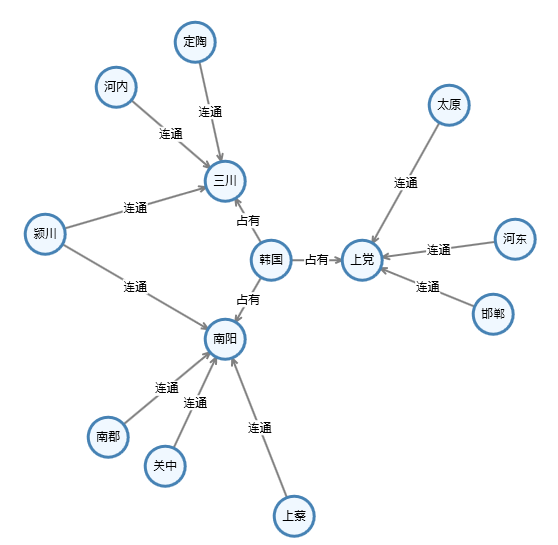


图4.9以选中节点为核心的三层关系网络

另外还有一种方法，可以直接点击关系网络中的节点，从而在节点列表中跳转到指定节点。数据库操作区和关系图也会做相应更新。

### 4.2.4 节点信息查看

首选需要选中一个节点在网络图中绘制一个关系图，随后将鼠标移动到节点处，停顿1-2s后，鼠标指针下方会弹出节点信息列表。移开鼠标三秒后信息列表自动消失。

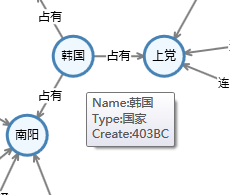


图4.10“韩国”节点的全部信息

### 4.2.5加入节点和连边

可以在功能区Add页面找到新增节点和连边的操作区，如图4.10和4.12。

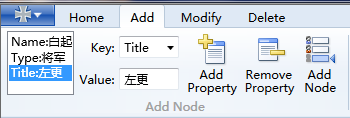


图4.10 新增节点

新增节点首先需要将节点必要的信息插入左侧列表，其中Name和Type是必须的字段，如前所述，Name和Type作为组合键唯一确定一个节点。

用户也可以在组合框Key中任意输入自己业务相关的属性标签和取值，并通过Add Property按钮加入左侧列表。如果输入错误，也可以在左侧列表中选取该属性，并在Value框中修改。注意，如果插入的属性Key相同则会被替换。如果Key输入错误，可以选中错误属性并点击Remove Property移除错误属性。

当所有信息都输入以后，单击Add Node按钮，将节点插入数据库。与此同时，UI左侧的节点列表也会一并刷新。如果插入过程中出现任何错误，都可以在底部状态栏看到提示，如图4.11。

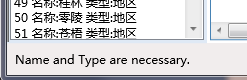


图4.11状态栏错误提示

新增连边需要首先指定起始节点，有两种方法。可以直接在Start Name和Start Type框中输入，也可以通过点击节点列表来选择节点。但是终止节点只能通过手工输入数据来指定。最后输入连边关系描述的Key和Value即可，这二者都由用户的业务来定。最后点击Add Edge按钮就可以插入连边。插入完成后，中央的网络关系图会自动更新，加入这条新边。与加入节点操作相同，插入状态和错误信息会在底部状态栏显示，后述操作均有这种功能，不再赘述。

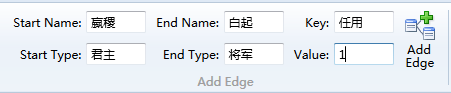


图4.12 新增连边

### 4.2.6 修改节点和连边

网络中任意节点和连边的数据都是可以修改的，但是不包括节点的Name和Type，连边的Start和End，他们属于节点和连边的键，如需修改，必须将原对象删除，重新加入数据库。

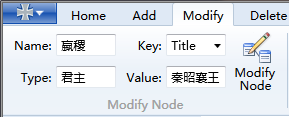


图4.13 修改节点数据

节点修改主要是针对节点数据中的用户业务数据。用户可以通过节点选择列表，单击关系图节点或手工输入Name和Type三种方法确定需要修改的节点。如果找到目标节点，右侧的Key列表中会列出节点中用户数据的几个属性，选择不同的属性，其对应属性值会在下方的Value框中显示。

修改已有属性，只需要选中一个属性，改动Value框中的值（不为空），随后单击Modify Node按钮保存数据即可，新的属性值会取代旧的值。

加入新属性，可以删除Key框中现有数据，并输入新的属性名，并在下面Value中加入属性的值，并单击Modify Node，新的属性值就会加入到节点数据中。

删除已有属性，只需要选中一个属性，删除Value框中的值，将其清空，随后单击Modify Node按钮，系统检测到属性值为空，则会从节点中删除该属性。

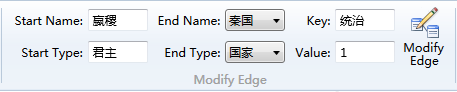


图4.14 修改连边数据

连边修改操作区中的Start节点可以使用和节点修改中相同的方法指定，一旦起始节点确定，终止节点的组合框也会自动加载起始节点指向的所有节点名称和类型，后方的Key和Value框也会读取对应连边的数据。用户可以通过下来组合框选择需要修改连边的终止节点。一旦确定，只要修改右侧的Key和Value框中的取值，并单击Modify Edge即可。由于一条边只能有一个关系和取值，新的Key和Value会取代以前的值。

### 4.2.7 删除节点和连边

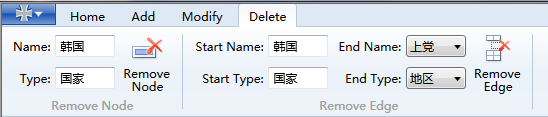


图4.15 删除节点操作

删除节点操作，只需要按照前述三种方法（节点选择列表，单击关系图节点或手工输入Name和Type）指定目标节点即可。随后点击Remove Node按钮删除节点，删除完成后左侧节点列表也会随之更新。

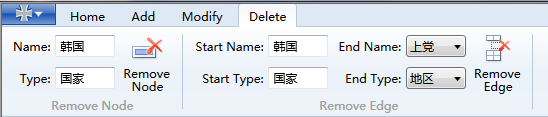


图4.16 删除连边操作

当用户指定了起始节点之后，终止节点组合下拉框中将出现所有该节点指向的目标节点。用户可以选择一个待删除的连边指向的节点，最后点击Remove Edge完成删除操作。界面中央的节点关系图将随之更新。

### 4.2.8 查询功能

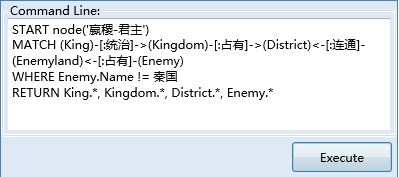


图4.17 查询命令行

在查询命令框中输入Cypher查询语句（语法规则参见3.4节），点击Execute按钮，返回的结果将在Result框中显示，如图4.18所示。



图4.18 查询结果

# 5. 项目后期任务

## 5.1 测试用例

### 5.1.1 白盒覆盖用例

|  |  |  |  |
| --- | --- | --- | --- |
| 测试组 | 测试点描述 | 语句 | 预期结果 |
| START语句 | 1. 起始节点精确选取 2.二层条件查询 3.指定名称输出全部数据 | START node('嬴稷-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.\*, Kingdom.\*, District.\* | 1.第一组秦王所有信息 2.第二组秦国所有信息 3.第三组秦国占有的所有地区的所有信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家各自占有地区的所有信息 |
| Match语句 | 查找秦国所有邻国 1.起始节点精确选取 2.四层条件查询 3.指定过滤条件Enemy字段非本国 4.返回Enemy字段的名字 | START node('嬴稷-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District)<-[:连通]-(Land)<-[:占有]-(Enemy) WHERE Enemy.Name != 秦国 RETURN Enemy.Name | 1.第一组秦国所有邻国的名称 |
| 查找秦国所有邻国，及其君主 1.起始节点精确选取 2.五层条件查询 3.指定过滤条件Enemy字段非本国 4.返回Enemy字段的名字 | START node('嬴稷-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District)<-[:连通]-(Land)<-[:占有]-(Enemy)<-[:统治]-(Enemy\_king) WHERE Enemy.Name != 秦国 RETURN Enemy.Name, Enemy\_king.\* | 1.第一组秦国所有邻国的名称 2.第二组秦国所有邻国君主的所有信息 |
| 查找拥有四个属地的国家和地区 1.起始节点批量选取 2.二层条件查询， 3.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有 4..4]->(District) WHERE \* RETURN Kingdom.\*, District.\* | 1.第一组拥有四个属地的国家信息 2.第二组为相关地区信息 |
| 查找拥有4到10个属地的国家和地区 1.起始节点批量选取 2.二层条件查询， 3.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有 4..10]->(District) WHERE \* RETURN Kingdom.\*, District.\* | 1.第一组拥有4到10个属地的国家信息 2.第二组为相关地区信息 |
| WHERE语句 | 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家占有地区的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE District.Population > 400000 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家占有地区中人口大于400000的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE District.Population < 200000 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家占有地区中人口小于200000的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE District.Population >= 400000 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家占有地区中人口大于等于400000的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE District.Population <= 200000 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家占有地区中人口小于等于200000的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE King.Name == 姬平 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组姬平的信息 2.第二组燕国的信息 3.第三组燕国占有地区的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定条件过滤结果 4.指定名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE King.Name != 芈槐 RETURN King.\*, Kingdom.\*, District.\* | 1.第一组除芈槐以外所有君主的信息 2.第二组除楚国以外所有国家的信息 3.第三组除楚国领土所有国家占有地区的信息 |
| Return语句 | 1.起始节点批量选取 2.二层条件查询 3.指定一个对象名称输出全部数据 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.\* | 1.第一组所有君主的信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定一个对象名称输出全部数据（错误用例） | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King. | 1.空 |
| 1.起始节点批量选取 2.二层条件查询 3.指定一个对象名称的一个字段 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.Name | 1.第一组所有君主的姓名信息 |
| 1.起始节点批量选取 2.二层条件查询 3.指定一个对象名称的多个字段 | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN King.Name, King.Born | 1.第一组所有君主的姓名和出生年份信息 |
| 1.起始节点批量选取 2.二层条件查询 3.设为通配符\* | START node('\*-君主')  MATCH (King)-[:统治]->(Kingdom)-[:占有]->(District) WHERE \* RETURN \* | 1.第一组所有君主的信息 2.第二组所有国家的信息 3.第三组所有国家各自占有地区的所有信息 |

## 5.2 功能缺陷与改进点

### 5.2.1 内核设计

1. GraphDB的节点内容和连边内容存放强制采用string方式，这样不仅耗费空间，解析也占用资源。以后应当改为<T>泛型，这样节点可以作为任何数据对象的容器。

2. Cypher解析器暂时不支持增删改三种操作的语句，后续版本应当继续扩充支持。

### 5.2.2 UI设计

1. WPF设计主界面，一些元素使用不熟练，存在遗留问题。

2. 由于界面空间有限，导致无法支持两节点间多连边，绘图功能中也无法支持多连边。

3. 绘图自动布局算法存在缺陷，部分节点存在相互重叠现象，需要继续改进。

# 6.项目总结

GraphDB图数据库项目分为两个主要部分，GraphDB内核和WPF的UI界面。前者单独编译为dll文件，可以作为第三方组件，被任何项目引用。后者主要是为了学习WPF富媒体界面技术进行的实践。



图6.1 GraphDB三层结构

GraphDB内核采用了三层结构，模块化构建，每一层都单独依赖于前一层，不存在反向依赖，这样有利于解耦合和模块裁剪。

WPF富媒体技术，作为一种基于XML的呈现艺术，具有非常高的自由度，几乎可以在其中按照自己的想象任意组织一个界面。其内置的命令系统，数据绑定系统和资源系统也为界面代码简化做出了巨大的贡献。本项目的完成，也代表作者本人的WPF学习实践阶段顺利结束。以后就进入了理论深化和自由使用技术的阶段。

图数据库作为一种新型的NoSQL数据库，彻底颠覆了以往基于表格的关系型数据库。由于图数据库直观地反映了自然界事务的事实关系，使得相同的数据库可以在完全不同的业务领域使用。唯一的差别就是观察节点的角度不同而已。商人关注消费关系，政客关注政治和外交关系，军事家关注军队部署，八卦娱记关注人际联系… 但是他们可以完全共用一个数据库，只需要利用查询语句，提取出他们感兴趣的节点和关系即可。

相信在未来，基于图的数据库能够取得更大的成功，利用复杂网络技术开发的图数据库GraphDB初代模型，也将继续扩充成长，拓展自己发挥作用的舞台。