**手机Totem开发详细设计**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：数据库系统实现

指 导 教 师 ：彭智勇 教授

学 生 姓 名 ：李蕴哲、李沁遥、

谢宇涛、郑晖、朱赫

# 二○二〇年二月

# 目 录

[目 录 I](#_Toc30434221)

[前言 2](#_Toc30434222)

[第一章 系统需求分析 3](#_Toc30434223)

[1.1系统概述 3](#_Toc30434224)

[1.2 功能性需求分析 3](#_Toc30434225)

[第二章 系统设计 4](#_Toc30434226)

[2.1 系统整体设计 4](#_Toc30434227)

[2.2 数据结构设计 4](#_Toc30434228)

[2.3 算法流程设计 4](#_Toc30434229)

[第三章 系统实现与测试 5](#_Toc30434230)

[3.1 系统实现 5](#_Toc30434231)

[3.2 系统测试 5](#_Toc30434232)

# 前言

本系统实现过程如下：十二月中旬确定了设计需求，并初步明确了设计方案和团队分工；十二月下旬团队成员分别学习；一月上旬进行了开发设计，在这期间，我们首先设计了APP框架，并同时编写编译、执行、存储三部分的代码，最后进行了测试和UI的改进；一月中旬进行了文档整理和系统展示；二月上旬写作了此实验报告。

团队分工如下：李沁遥复杂APP的总体设计和实现；朱赫负责编译部分的实现；李蕴哲负责执行部分的实现；郑晖负责存储部分的实现；谢宇涛负责测试和调试。

# 第一章 系统需求分析

## 1.1系统概述

本系统是一个对象代理数据库系统，它将客观实体描述为对象，每个对象都包含自身的属性和方法，而类又用来描述具有相同属性和方法的对象。为了满足不同上层应用的要求，我们又在对象上定义了代理对象。代理对象继承对象的属性和方法，也可以新增源对象没有的属性和方法。代理对象的模式就由代理类来描述。由于代理类种类比较多，而我们又时间有限，故只实现了SelectDeputy这一种代理类。

用户只需要输入对象代理数据库相应的SQL语句，系统会自动进行编译、执行、存储相关内容，并反馈出结果来，具体反馈界面会在后述内容中详细介绍。

## 1.2 功能性需求分析

本系统的功能性需求主要是对象代理数据库的基本操作，包括创建源类、创建代理类、删除类、删除对象、插入数据、查询、跨类查询、更新等操作。

创建源类：源类是最基本的类，所有属性均为实属性，是各种代理类的基础。基本类的定义过程包括:定义类名、属性名和属性类型;建立约束，如属性是否允许为空，属性的缺省值等;创建方法、定义方法的名称、参数类型、 返回类型以及函数体。

创建代理类：代理类的定义包括定义代理类名、代理类型、实属性、方法以及代理规则。 创建代理类后系统会根据代理规则将源类中的对象放入代理类中去。

删除类：通过删除类的命令，用户可以将任何层次上的类从数据库中删除。

删除对象：用户可以删除源类中的任一对象。

插入数据：通过输入数据来新建对象，一般包括对象名、类名、属性等。

查询：对于用户而言，无论要查询什么，只需输入SQL语句即可。对于系统而言，需要甄别查询对象是虚属性还是实属性，若是实属性，从内存中取出相应内容即可。如果是虚属性，则需要根据代理规则找到对应的实属性，计算后返还给用户。

跨类查询：跨类查询通过用户输入的路径表达式，从一个类出发，经过若干相关类达到目标类，并对目标类中的属性进行计算，最后返回结果给用户。

更新：在类或对象发生更新时，系统会产生更性迁移，以维护代理规则。在我们实现的这个系统中，更新迁移是通过扫描源类，根据代理规则修改代理类来实现的。

# 第二章 系统设计

## 2.1 系统整体设计

系统需要完成的工作包括编译、执行、存储。编译需要获取SQL语句中提供的信息，执行计算出结果，而存储则将执行的结果保存下来。

在**编译部分**，我们使用了JavaCC工具生成编译的Java代码。首先我们对JavaCC文件内的options参数进行设置。其他参数使用默认值，需要修改的部分有以下三个：

STATIC = false。默认值为True，此时生成的解析器中将所有方法和类变量都指定为static，此时只允许存在一个解析器对象。因为在执行过程中要new多次解析器，所以将此项设为false；

LOOKAHEAD = 2。因为在判断创建源类还是代理类、判断是否是跨类查询时可能出现语法冲突，所以将前看的词数设为2；

DEBUG\_PARSER = true。此选项用于从生成的解析器获取调试信息。设置为true后解析器会对它的动作进行追踪。

在调用完所需要的库和封装完给执行部分的接口后，我们对词法进行解析，分类为运算符和标点、逻辑语句、数据类型、SQL语句关键词等部分。之后，我们定义了一些需要用到的通用函数，并根据输入的具体的SQL语句进行解析，将解析后的参数存储到数据结构的相应位置，供下一步执行调用。

**执行部分**首先从编译部分得到数据，存入自己的数据结构中，然后对不同的语句，执行对应的操作，并将操作的结果及时交给存储部分，由存储部分存到硬盘。整个系统的各个部分都是在执行部分调用的。执行部分是整个系统最核心的部分。

在执行的过程中，用户输入一条SQL语句，点击执行按钮，系统才会执行。因此我们重载了onClick函数，在这个函数里面进行处理。用户输入的SQL语句，对于系统而言是一个string串。讲这个string串放到编译部分提供的分析器里面去分析，即可得到编译的结果。在后续的过程中，我们将多次调用这个编译的结果result。

接下来，我们调用了一个分析函数Analyze，在这里完成了执行部分的处理。函数Analyze维护两个全局变量：tip、tf。其中tip存放不同语句情况下给用户的提示信息，tf则是标志位，用以判断是否执行成功。Analyze函数中将调用不同的函数去完成具体的执行操作，具体算法在2.3节中将予以展示。

执行成功则提示用户成功，否则提示用户失败。如果是select语句，还要显示select的结果，因此需要跳转到select的界面，并将select的结果显示出来。用户的输入可能不符合语法，编译不通过，此时编译部分将无法匹配，也即无法得到结果。因此，返回值为空，此时直接退出，提示语法错误即可。

**存储部分**主要负责为本数据库系统提供底层存储接口。该部分主要维护了用于存储数据的虚拟磁盘和7张系统表，分别是CLASS\_TABLE、ATTRIBUTE\_TABLE、DEPUTY\_TABLE、DEPUTY\_RULE\_TABLE、SWITCHING\_TABLE、OBJECT\_TABLE和BIPOINTER\_TABLE。我们实现了存储格式和内存中数据结构的独立，向执行部分隐藏了存储的具体细节。比如，我们通过Next()函数，即可得到指定类的下一个tuple；执行部分在操作的时候也只能看到存储部分为其提供的虚拟blockNum等属性，而不会看到真正的虚拟磁盘blockNum等属性。

存储部分为执行部分提供了一些诸如getClassStruct的函数。该函数只需要从执行部分接收一个ClassName，便可以从CLASS\_TABLE中查找相应的classId，然后依据classId查找其它的系统表，最终为执行部分返回一个完整的ClassStruct数据结构。同理，saveClassStruct函数则是通过执行部分提供的ClassStruct数据结构，将其中的各个属性存储到不同的系统表中，所有的中间转换都是由存储部分实现的。然后，比如insert函数则是依据虚拟磁盘的slub-like表查找到该类的一个空闲位置，然后将ArrayList<String>格式的tuple以字节串的形式存储到虚拟磁盘上。其中，由于tuple的第0项为该tuple的双向指针，在插入的时候会依据其内容更新BiPointer系统表。update、delete等函数皆是如此，执行部分只需提供相应的数据格式，剩下的部分皆是由虚拟磁盘实现，向上进行了一层虚拟。

此外，存储部分还实现了缓存的设计。由于我们实现的虚拟磁盘是一个以512B为一块的32MB磁盘，所以它的缓存设计十分类似硬件中的cache，每一块都是一个512B的byte[]数组，我们总共设计了8块，提供4KB的缓存。缓存中每一个bufferLine都带有一个tag位，表示自己是对应于哪一个blockNum，还有一个dirty位，表明这个bufferLine是否为脏，主要在写回的时候要使用。替换策略很简单，就是如果还有空余的bufferLine，便直接占用，否则写回blockNum % 8的buffer Line，然后载入。

## 2.2 数据结构设计

编译部分数据结构：

public class ParseResult {

public int Type;

public String className;

public String selectClassName;

public ArrayList<Attribute> attrList;

public ArrayList<AttrNameTuple> attrNameList;

public ArrayList<String> valueList;

public WhereClause where;

}

其中Type存放语句的类型，className存放类名，selectClassName存放代理类的源类名。

class Attribute {

public String attrName;

public int attrType;

public int attrSize;

public String defaultVal;

public Attribute(String attrName, int attrType, int attrSize,String defaultVal) {

this.attrName = attrName;

this.attrType = attrType;

this.attrSize = attrSize;

this.defaultVal = defaultVal==null?"":defaultVal;

}

}

attrList存放实属性的名称、类型、大小和数值。

class AttrNameTuple {

public String attrName;

public String attrRename;

public AttrNameTuple(String \_attrName,String \_attrRename){

this.attrName = \_attrName;

this.attrRename = \_attrRename;

}

public AttrNameTuple(){}

}

attrNameList：存放虚属性名称和代理规则，代理规则在编译阶段存放的是字符串类型的表达式，后续会继续进行计算。

class WhereClause{

int operationType;

int valueInt;

String valueString;

WhereClause left;

WhereClause right;

ArrayList<String> tuple;

ArrayList<Attribute> attrs;

int is\_int;

private String getValue(String name){

int len = tuple.size();

for(int i=0;i<len;i++){

if(name.equals(attrs.get(i).attrName)){

is\_int = attrs.get(i).attrType==0?1:0;

return tuple.get(i);

}

}

return "";

}

private boolean node(WhereClause cond){

WhereClause \_left = cond.left,\_right=cond.right;

int l;

String sl;

switch (cond.operationType){

case 0:

sl = getValue(\_left.valueString);

if(\_right.operationType==9){

return Integer.parseInt(sl)>\_right.valueInt;

}else{

return sl.compareTo(\_right.valueString)>0?true:false;

}

case 1:

sl = getValue(\_left.valueString);

if(\_right.operationType==9){

return Integer.parseInt(sl)<\_right.valueInt;

}else{

return sl.compareTo(\_right.valueString)<0?true:false;

}

case 2:

sl = getValue(\_left.valueString);

if(is\_int==1)return Integer.parseInt(sl)==\_right.valueInt;

return sl.equals(\_right.valueString);

case 3:

sl = getValue(\_left.valueString);

if(is\_int==1)return Integer.parseInt(sl) != \_right.valueInt;

return !sl.equals(\_right.valueString);

case 4:

sl = getValue(\_left.valueString);

if(\_right.operationType==9){

return Integer.parseInt(sl)>=\_right.valueInt;

}else{

return sl.compareTo(\_right.valueString)>=0?true:false;

}

case 5:

sl = getValue(\_left.valueString);

if(\_right.operationType==9){

return Integer.parseInt(sl)<=\_right.valueInt;

}else{

return sl.compareTo(\_right.valueString)<=0?true:false;

}

case 6:

return !node(\_left);

case 7:

return node(\_left)&&node(\_right);

case 8:

return node(\_left)||node(\_right);

case 9:

if(cond.valueInt==0)return false;

return true;

case 10:

if(cond.valueString.equals(""))return false;

return true;

default:

return true;

}

}

public boolean judge(ArrayList<String> \_tuple,ArrayList<Attribute> \_attrs){

tuple = \_tuple;

attrs = \_attrs;

return node(this);

}

}

WhereClause存放where后面的语句，并提供了判断是否满足条件的函数。因为where语句存在计算的问题并且计算时where内各个符号的优先级不同，我们设计了一个语法树来存储where的表达式。树的每个节点都有左孩子、右孩子和符号(可以为空)。通过where节点的深度来区分优先级。优先级越高的深度越深。这样在执行部分只需要根据中间符号匹配左右两边是否满足要求即可。

执行部分数据结构：

class ClassStruct{

public String className;

public String selectClassName;

public ArrayList<String> children;

public ArrayList<Attribute> attrList;

public String condition; public ArrayList<AttrNameTuple> virtualAttr;

}

其中，className存放的是类名；selectClassName存放的是该类代理的源类，这一字段仅代理类有用；children存放的是指向源类的代理类的指针，仅对源类有用；attrList则是类的实属性列表；condition存放的是代理条件，仅代理类有用，实质存储的是创建该类的时候的sql语句；virtualAttr存放的是虚属性，只有代理类能够使用。

存储部分数据结构：

public class attributeTable {

public final static int N\_ATTR = 7;

public final static int notVirtual = 0, isVirtual = 1;

public int classId;

public int attrId;

public String attrName;

public int attrType;

public int isDeputy;

// appendix

public int attrSize;

public String defaultValue;

}

其中，classId表示attr所属class的id编号，attrId表示attr的id编号，attrName表示attr的名字，attrType表示attr的类型，isDeputy表示attr是否为一个虚属性，attrSize存储attr的大小，defaultValue存储attr的默认值。

public class biPointerTable {

public final static int N\_ATTR = 4;

public int classId;

public int objectId;

public int deputyClassId;

public int deputyObjectId;

}

其中，classId表示源类的id编号，objectId表示源类tuple的顺序编号，deputyClassId表示代理类的id编号，deputyObjectId表示代理类tuple的顺序编号。

public class bufferLine {

final static int MAX\_INTEGER = 0x7fffffff;

public int n\_block;

public int isDirty;

public byte[] buffer;

}

其中，n\_block表示缓存块的index，isDirty表示该bufferLine是否为脏，buffer用于存储数据。

public class classTable {

public final static int N\_ATTR = 5;

public final static int originClass = 0, deputyClass = 1;

public String className;

public int classId;

public int attrNum;

public int classType;

public int tupleNum;

}

其中，className表示class的名字，classId表示class的id编号，attrNum存储class所含attr的数目，classType表示class的类型，tupleNum表示class有多少数据。

public class deputyRuleTable {

public final static int N\_ATTR = 2;

public int deputyRuleId;

public String deputyRule;

}

其中，deputyId表示代理规则的id编号，deputyRule是代理规则。

public class deputyTable {

public final static int N\_ATTR = 3;

public int originId;

public int deputyId;

public int deputyRuleId;

}

其中，originId是源类的id编号，deputyId是代理类的id编号，deputyRuleId是代理规则的id编号。

public class objectTable {

public final static int N\_ATTR = 4;

public int classId;

public int tupleId;

public int blockId;

public int offset;

}

其中，classId是class的id编号，tupleId是tuple的顺序编号，blockId是该tuple所在的blockNum，offset是该tuple所在的blockOffset。

public class switchingTable {

public final static int N\_ATTR = 2;

public int attrId;

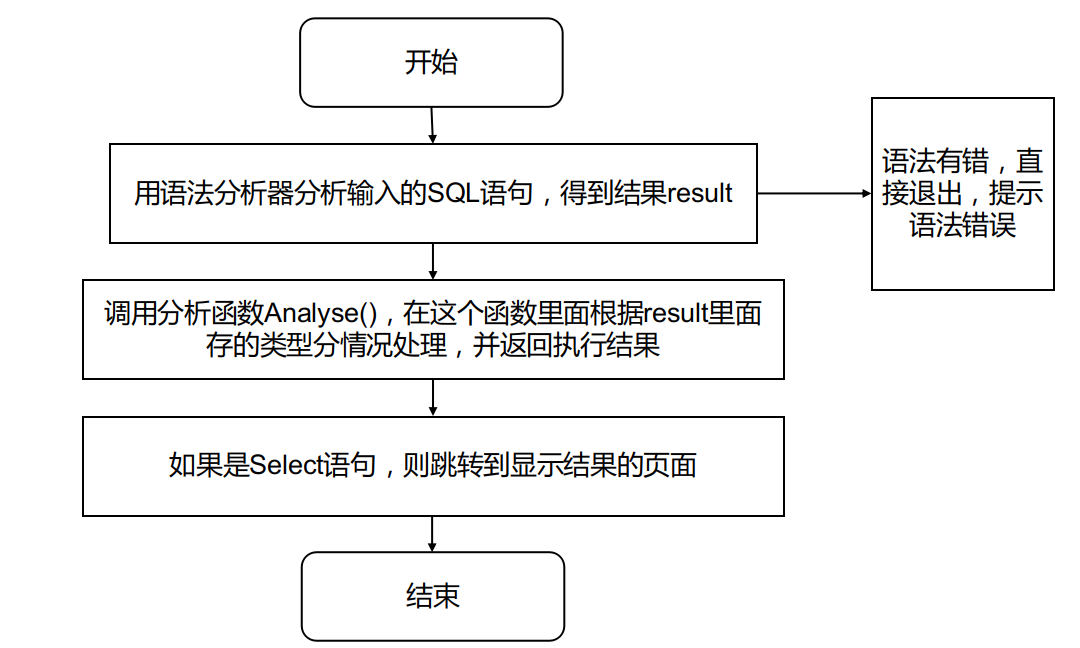
public String rule;

}

其中，attrId是属性的id编号，rule是属性的switching规则。

## 2.3 算法流程设计

系统整体算法框架如下所示：



### 创建源类

源类是最基础的类，只需要将数据按照数据结构设计的内容分别存储即可。

//创建源类

static boolean Create(){

ClassStruct classStruct = new ClassStruct();

classStruct.className = result.className;//className存放的是类名

classStruct.selectClassName = null;

classStruct.attrList = result.attrList;//实属性列表

classStruct.virtualAttr = null;

classStruct.condition = sql;//condition先不管他，先存起来，后面再分析

classStruct.tupleNum = 0;

classStruct.children = new ArrayList<>();

return storeAPI.createClass(result.className,classStruct);

}

值得一提的是，在存储表的时候，在执行部分我们只是调用了存储部分给的函数，并没有去维护这些表。创建表的工作由存储来完成。

### 创建代理类

源类在创建之初一定是空的，而代理类并不是这样。根据代理规则，代理类在创建之初，源类中符合代理规则的记录就将自动进入代理类中。因此，在创建代理类的时候，就需要找到源类，并根据代理规则进行第一次的更新。我们的更新操作是直接调用isSatisfy函数遍历所有源类中的记录，将符合条件的记录添加到代理类中。代理类中有这些记录，但是其虚属性是不保存的，而是由代理规则在查询的过程中实时计算出来。此时，我们不需要查询，因此代理规则并没有用到。由于代理规则的处理较为复杂，并且并不是全局需要的，我们在创建代理类的时候，并不直接存储代理规则。我们在代理规则部分依旧存储原语句，而在后续需要使用代理规则的时候，我们再重新调用编译部分的函数将其解析一次，以得到真正的代理规则。

//创建代理类

static boolean CreateSelectDeputy(){

ClassStruct classStruct = new ClassStruct();

classStruct.className = result.className;//类名

classStruct.selectClassName = result.selectClassName;//源类名

classStruct.children = new ArrayList<>();

classStruct.condition = sql;//将sql放到condition中间，留作后面继续解析获得where属性

classStruct.virtualAttr = result.attrNameList;//虚属性

classStruct.attrList = result.attrList;//实属性

if(storeAPI.existClass(classStruct.selectClassName)==false)return false;//如果不存在源类，就报错

if(!storeAPI.createClass(result.className,classStruct))return false;//创建失败，则返回错误

ClassStruct fa\_classStruct = storeAPI.getClassStruct(result.selectClassName);//找到他们的父节点

fa\_classStruct.children.add(classStruct.className);//父节点的孩子里面加上类名

Log.d("Hello", "CreateSelectDeputy: "+fa\_classStruct.children.size());//记录日志，用于debug

storeAPI.setClassStruct(fa\_classStruct.className,fa\_classStruct);

//更新迁移

storeAPI.initial(fa\_classStruct.className);

MEM.initial(classStruct.className);

ArrayList<String > tuple = null;

ArrayList<String> son\_tuple = new ArrayList<>();

son\_tuple.add("-1");//pointers

if(classStruct.attrList!=null)

for(int i=0;i<classStruct.attrList.size();i++){

son\_tuple.add(classStruct.attrList.get(i).defaultVal);

}//实属性如果有的话，直接就加上

while((tuple=storeAPI.Next())!=null){

int son\_offset = -1;

if(isSatisfy(result.where,fa\_classStruct,tuple)){

int offset = storeAPI.getOffset();

son\_tuple.set(0,Integer.toString(offset));

MEM.insert(classStruct.className,son\_tuple);

son\_offset = MEM.getOffset();//将满足条件的虚属性加上

}

ArrayList<String> tmp = storeAPI.decode(tuple.get(0));

tmp.add(Integer.toString(son\_offset));

tuple.set(0,storeAPI.encode(tmp));

storeAPI.update(tuple);

}

MEM.flushToDisk();

return true;

}

### 删除记录

删除操作是将记录删除的操作。但是在对象代理数据库中，如果一个类有自己的代理类，那么很可能这条记录被它的代理类代理了。删除这条记录的同时，也应当将代理类中的记录删除。

在执行部分，我们只考虑将块删除，对于系统表一致性的维护，由存储部分完成。

//从源类中删除一个元组，并把孩子的该元组都删除 Return true Anyways!

static boolean Delete(){

ClassStruct classStruct = storeAPI.getClassStruct(result.className);

//获取该类的所有结构，用于后面找孩子

ArrayList<String> tuple = null;

storeAPI.initial(result.className);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,storeAPI.getClassStruct(result.className),tuple)){

//找孩子

ArrayList<String> nexts = storeAPI.decode(tuple.get(0));

for(int i=0;i<nexts.size();i++){

if(!nexts.get(i).equals("-1")) {

int offset = Integer.parseInt(nexts.get(i));

//获取对应的块

MEM.initial(classStruct.children.get(i),

offset/StoreAPI.PAGESIZE,offset%StoreAPI.PAGESIZE);

ArrayList<String> \_t = MEM.Next();

MEM.delete();//将孩子也都删除

}

}

storeAPI.delete();//将记录删除

}

}

MEM.flushToDisk();//将删完的刷到磁盘上去，也即将这块删除

return true;

}

### 删除类

删除类分为两种，一是删除源类，一是删除代理类。我们首先判断要删除的类是源类还是代理类，接着分别进行处理。

#### 删除源类

删除一个源类，就也要将其代理类都删除，其实就是删除了整棵树。

我们递归删除了整个由源类指向代理类的指针生成的树，实现了对一个类的删除。

//删除源类,实质上会把整棵树删除

static boolean Drop(String className){

if(storeAPI.existClass(className)==false){return false;}//不存在，则返回错误

ClassStruct classStruct = storeAPI.getClassStruct(className);

storeAPI.dropClass(className);

if(classStruct!=null)

for(int i=0;i<classStruct.children.size();i++){

Drop(classStruct.children.get(i));//依次递归删除各个节点

}

return true;

}

#### 删除代理类

删除代理类和删除源类的区别在于，删除代理类除了要将它自己的孩子都删除外，还要将它的父节点指向它的指针也都删除。我们的做法是便利它父节点的所有孩子，找到属于他的信息，然后将之删除。接着就把它当做源类，调用上面的删除源类的函数，完成接下来的删除处理。

//删除代理类

static boolean DropSelectDeputy(String className){

if(!storeAPI.existClass(className)){

return false;

}

//剔除父亲的关于它的指针

ClassStruct fa\_classStruct = storeAPI.getClassStruct(

storeAPI.getClassStruct(className).selectClassName);

int index = 0;

if(fa\_classStruct.children!=null)

for(;index<fa\_classStruct.children.size();index++){

if(fa\_classStruct.children.get(index).equals(className)){

fa\_classStruct.children.remove(index);//找到对应的index，就将他删除

storeAPI.setClassStruct(fa\_classStruct.className,fa\_classStruct);

storeAPI.initial(fa\_classStruct.className);

ArrayList<String> tuple = null;

while((tuple = storeAPI.Next())!=null){

ArrayList<String> pointer = storeAPI.decode(tuple.get(0));

pointer.remove(index);

tuple.set(0,storeAPI.encode(pointer));

storeAPI.update(tuple);//将删完的刷到磁盘上去

}

break;

}

}

Drop(className);

return true;

}

### 插入

插入操作是一个较为复杂的操作，因为它存在这些问题：

* 编译部分获得的信息只有插入的类名，需要我们自己去获得类的属性表
* 插入的类型可能是非法的，我们需要进行类型检测
* 插入后，如果该类有代理类，而插入的语句符合代理规则，那么就需要将该记录同样插入到代理类中

因此，我们首先获得了要插入的类现有的类结构，同时将编译获取到的参数列表转换正确的形式。比如，对于所有的整型数据，我们在编译部分都是按照字符串存储的，现在需要将其转换成整型。此过程也检验插入数据是否合法，因为非法的数据是无法转换成功的。接着将合法的类型插入。

如果存在子类，我们则调用isSatisfy函数，判断其是否符合代理规则。如果符合代理规则，就将记录也插入子类中。在插入子类的时候，还要加上子类的实属性，由于此时没有给子类的实属性赋值，我们便插入了默认值。最后将执行结果刷进磁盘。

//insert

static boolean Insert()

{

//获取要插入的源类的类结构

ClassStruct classStructOfSourceClass = storeAPI.getClassStruct(result.className);

//类型检查

try

{

//获取要插入的源类的属性类型等信息列表

ArrayList<Attribute> attrListOfSourceClass = classStructOfSourceClass.attrList;

Attribute attrTemp;

if(attrListOfSourceClass!=null)

for(int i = 0; i < attrListOfSourceClass.size(); i++)

{

attrTemp = attrListOfSourceClass.get(i);

if(attrTemp.attrType == 0)

{

Integer.parseInt(result.valueList.get(i));

}

}

}

catch(NumberFormatException e)

{

System.out.println("Type Error!");

return false;

}

/\*建立新tuple的0号位,没有就是空串\*/

ArrayList<String>sourcePointer = new ArrayList<String>();

result.valueList.add(0,storeAPI.encode(sourcePointer));

storeAPI.insert(result.className, result.valueList);

//获取新插入的tuple的offset

int sourceOffset = storeAPI.getOffset();

//若存在子类，要判断是否插入子类，并且更新父亲的指针

if(classStructOfSourceClass.children != null && classStructOfSourceClass.children.size() > 0)

{

for(int i = 0; i < classStructOfSourceClass.children.size(); i++)

{

//用MEM块来处理子块

ClassStruct classStructOfChildrenClass = MEM.getClassStruct(classStructOfSourceClass.children.get(i));

//对创建子类时的SQL语句重新解析，目的是获得Where的结构

ParseResult conditionResult = SQLParser.sqlParse(classStructOfChildrenClass.condition);

//判断的是父类的实属性和要插入的tuple值

if(isSatisfy(conditionResult.where, classStructOfSourceClass, result.valueList))

{

//如果符合条件，则也插入子类，在子类新增一个tuple，该tuple的第0位是父类中新增tuple的偏移量

ArrayList<String>newTuple = new ArrayList<String>();

newTuple.add(String.valueOf(sourceOffset));

//然后增加实属性的值

if(classStructOfChildrenClass.attrList != null && classStructOfChildrenClass.attrList.size()>0)

{

for(int j = 0; j < classStructOfChildrenClass.attrList.size(); j++)

{

//插入实属性的默认值

newTuple.add(classStructOfChildrenClass.attrList.get(j).defaultVal);

}

}

//将newTuple插入到子类中

MEM.insert(conditionResult.className,newTuple);

//修改源类指向子类的指针

int childrenOffset = MEM.getOffset();

sourcePointer.add(String.valueOf(childrenOffset));

}

else

{

//不符合条件，不插入子类，修改源类指向子类的指针为-1

sourcePointer.add("-1");

}

}

//对源类tuple的第0位编码

String pointer = storeAPI.encode(sourcePointer);

//更新源类刚插入的tuple值

result.valueList.set(0, pointer);

storeAPI.update(result.valueList);

}

MEM.flushToDisk();

return true;

}

### 更新

更新同样分为源类的更新和代理类的更新。

由于代理类的虚属性是实时计算出来的，其值依赖于源类的值。因此对代理类的虚属性更新是没有意义的。对于代理类，仅考虑对其实属性的更新。遍历代理类的属性列表和和要更新的属性的属性列表，如果属性名相同，就将更新的属性的计算结果赋值给代理类的属性列表，完成更新。

对于源类而言，除了要根据规则更新属性的值，还要考虑是否更新之后，代理类中的元组是否还满足代理规则，对于不满足代理规则的，应该删去，而满足代理规则的，则应该加入。我们的做法就是将子类中的所有记录都遍历一遍，分别进行判断。

//return true Anyways

static boolean Update()

{

//获取要插入的源类的类结构

ClassStruct classStructOfSourceClass = storeAPI.getClassStruct(result.className);

//start 代理类更新 （支持源类和代理类的更新）

if(classStructOfSourceClass.selectClassName!=null&&!classStructOfSourceClass.selectClassName.equals("")){

//deputy class update 只考虑对代理类实属性的更新

storeAPI.initial(classStructOfSourceClass.className);

ArrayList<String> tuple = null;

ArrayList<String> son\_title = getTitle(classStructOfSourceClass);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,classStructOfSourceClass,tuple)){

ArrayList<String> l\_tuple = compute(classStructOfSourceClass,tuple);//获取这个类的完整的结构

if(classStructOfSourceClass.attrList!=null)

for(int i=0;i<classStructOfSourceClass.attrList.size();i++){

for(int j=0;j<result.attrNameList.size();j++){

if(result.attrNameList.get(j).attrName.equals(classStructOfSourceClass.attrList.get(i).attrName)){

tuple.set(i+1,tinyCompute(son\_title,l\_tuple,result.valueList.get(j)));//将计算的结果存进去

}

}

}

storeAPI.update(tuple);

//代理类更新就不在乎会不会有源类跟着变化了，tinyCompute里面有处理

}

}

return true;

}

//END 代理类更新

//要拿到该类的所有tuple，然后根据where的限制选择更新

ArrayList<String> oldTuple = null;

storeAPI.initial(result.className);

ArrayList<String> fa\_title = getTitle(classStructOfSourceClass);

while((oldTuple = storeAPI.Next()) != null)

{

//如果当前tuple符合where的限制则更新

if(isSatisfy(result.where, classStructOfSourceClass, oldTuple))

{

if(classStructOfSourceClass.attrList != null && classStructOfSourceClass.attrList.size() > 0)

{

//遍历原属性列表和要更改的属性列表，确定要更改的属性

for(int i = 0; i < classStructOfSourceClass.attrList.size(); i++)

{

for(int j = 0; j < result.attrNameList.size(); j++)

{

//找到要更改的属性,是原属性列表的第i+1个属性值（第0位是指针位），要变成更改属性列表的第j个属性的值

if(classStructOfSourceClass.attrList.get(i).attrName.equals(result.attrNameList.get(j).attrName))

{

oldTuple.set(i+1,tinyCompute(fa\_title,oldTuple,result.valueList.get(j)));

}

}

}

}

//解析指向子类的指针

ArrayList<String> sourcePointer = storeAPI.decode(oldTuple.get(0));

//接下来判断更新之后的元组是否还满足创建代理时的要求，不符合的话，要把对应的指针记录删掉,新满足的将会增加

if(classStructOfSourceClass.children != null && classStructOfSourceClass.children.size() > 0)

{

for(int i = 0; i < sourcePointer.size(); i++)//在这里循环

{

int childrenClassOffset = Integer.parseInt(sourcePointer.get(i));

//对创建子类时的SQL语句重新解析，目的是获得Where的结构

ClassStruct classStructOfChildrenClass = MEM.getClassStruct(classStructOfSourceClass.children.get(i));

ParseResult conditionResult = SQLParser.sqlParse(classStructOfChildrenClass.condition);

if(!sourcePointer.get(i).equals("-1"))

{

//子类符合，不用变动；子类不符合，要删除

if(!isSatisfy(conditionResult.where, classStructOfSourceClass, oldTuple))

{

MEM.initial(classStructOfChildrenClass.className,

childrenClassOffset/StoreAPI.PAGESIZE,childrenClassOffset%StoreAPI.PAGESIZE);

ArrayList<String> temp = MEM.Next();

MEM.delete();

sourcePointer.set(i,"-1");

}

}else{

//子类符合应该添加

if(isSatisfy(conditionResult.where, classStructOfSourceClass, oldTuple))

{

ArrayList<String> l\_tuple = new ArrayList<>();

l\_tuple.add(Integer.toString(storeAPI.getOffset()));

for(int j=0;j<classStructOfChildrenClass.attrList.size();j++){

l\_tuple.add(classStructOfChildrenClass.attrList.get(j).defaultVal);

}

MEM.insert(classStructOfChildrenClass.className,l\_tuple);

sourcePointer.set(i,Integer.toString(MEM.getOffset()));

}

}

}

}

//更新源类的元组

oldTuple.set(0,storeAPI.encode(sourcePointer));

storeAPI.update(oldTuple);

}

}

MEM.flushToDisk();

return true;

}

### 查询

查询分为两种：

* 类间查询
* 跨类查询

我们分别对这两种情况分别进行了处理。

#### 类间查询

对类间的查询，我们首先需要获得要查询的属性的列表，然后遍历存储的各个记录，如果符合要求，就将其加入结果返回。

在这里需要区分一些源类和代理类的情况。对于源类而言，我们直接获取其所有记录即可。而对于代理类而言，我们需要先进行一次计算，以恢复其完整结构。因为虚属性的值是不直接存储的。

static ArrayList<ArrayList<String>> select(){

ArrayList<ArrayList<String>> res = new ArrayList<>();

if(!storeAPI.existClass(result.selectClassName))return res;

ClassStruct classStruct = storeAPI.getClassStruct(result.selectClassName);

ArrayList<String> title = new ArrayList<>();

if(result.attrNameList!=null)

for(int i=0;i<result.attrNameList.size();i++){

if(result.attrNameList.get(i).attrRename!=null&&

!result.attrNameList.get(i).attrRename.equals(""))

title.add(result.attrNameList.get(i).attrRename);//代理类

else

title.add(result.attrNameList.get(i).attrName);

}

res.add(title);

//源类查询

if(classStruct.selectClassName==null || classStruct.selectClassName.equals("")){

storeAPI.initial(classStruct.className);

ArrayList<String> tuple = null;

ArrayList<String> fa\_title = getTitle(classStruct);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,classStruct,tuple)){

tuple.remove(0);

ArrayList<String> tar = new ArrayList<>();

for(int i=0;i<result.attrNameList.size();i++){

String tmp = tinyCompute(fa\_title,tuple,result.attrNameList.get(i).attrName);

tar.add(tmp);

}

res.add(tar);

}

}

return res;

}

//代理类查询

storeAPI.initial(classStruct.className);

ArrayList<String> tuple = null;

ArrayList<String> son\_title = getTitle(classStruct);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,classStruct,tuple)){

ArrayList<String> l\_t = compute(classStruct,tuple);

ArrayList<String> tar = new ArrayList<>();

for(int i=0;i<result.attrNameList.size();i++){

String tmp = tinyCompute(son\_title,l\_t,result.attrNameList.get(i).attrName);

tar.add(tmp);

}

res.add(tar);

}

}

return res;

}

#### 跨类查询

跨类查询通过起点的条件判断，来查询终点的记录。跨类查询较类间查询要复杂得多，因为我们需要通过指针去找结果，而不能直接通过计算来得到。

在我们的编译部分，我们只存储了起点和终点。我们也没有对起点和终点的类型进行区分。因此，我们首将跨类查询分为了三类：

* 源类->代理类
* 代理类->源类
* 代理类->代理类

##### 源类->代理类

对于源类->代理类的情况，我们首先找到查询的代理类是源类的第几个孩子，通过孩子索引到查找的内容在终点中的位置，然后将结果从磁盘中读出返回。

int childIndex = 0;

for(int i=0;i<start\_point.children.size();i++){

if(start\_point.children.get(i).equals(end\_point.className)){

childIndex = i;//找到代理类的索引号

break;

}

}

storeAPI.initial(start\_point.className);

ArrayList<String> stuple = null;

ArrayList<String> title = getTitle(end\_point);

while((stuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,start\_point,stuple)){

ArrayList<String> pointer = storeAPI.decode(stuple.get(0));

int childOffset = Integer.parseInt(pointer.get(childIndex));

if(childOffset<0)continue;

MEM.initial(end\_point.className,

childOffset/StoreAPI.PAGESIZE,childOffset%StoreAPI.PAGESIZE);

ArrayList<String> tmp = compute(end\_point,MEM.Next());

res.add(getCalc(tmp,title));

}

}

return res;

##### 代理类->源类

从代理类向源类的跨类查询比从源类向代理类的查询简单一些，因为一个代理类只有一个源类，而一个源类可以有很多的代理类。所以判断完条件之后直接从块的开始处开始读取即可，其他操作与源类查询代理类一致。

//deputy\_1 -> source

storeAPI.initial(start\_point.className);

ArrayList<String> tuple = null;

ArrayList<String> title = getTitle(end\_point);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,start\_point,tuple)){

int fatherOffset = Integer.parseInt(tuple.get(0));

MEM.initial(end\_point.className,

fatherOffset/StoreAPI.PAGESIZE,fatherOffset%StoreAPI.PAGESIZE);

ArrayList<String> tmp = MEM.Next();

tmp.remove(0);

res.add(getCalc(tmp,title));

}

}

return res;

##### 代理类->代理类

从代理类到代理类的跨类查询是前两个查询的综合。我们首先根据他们之间的联系——他们共同的父节点找到他们的索引关系，然后再依次索引得到查询结果。

//deputy\_1 -> deputy\_2

storeAPI.initial(start\_point.className);

ArrayList<String> tuple = null;;

ClassStruct father = storeAPI.getClassStruct(start\_point.selectClassName);

int index = 0;

for(int i=0;i<father.children.size();i++){

if(father.children.get(i).equals(end\_point.className)){

index = i;

break;

}

}

ArrayList<String> son\_title = getTitle(end\_point);

while((tuple=storeAPI.Next())!=null){

if(isSatisfy(result.where,start\_point,tuple)){

int fatherOffset = Integer.parseInt(tuple.get(0));

MEM.initial(father.className,

fatherOffset/StoreAPI.PAGESIZE,fatherOffset%StoreAPI.PAGESIZE);

ArrayList<String > fa\_tuple = MEM.Next();

int sonOffset = Integer.parseInt(storeAPI.decode(fa\_tuple.get(0)).get(index));

if(sonOffset<0)continue;

MEM.initial(end\_point.className,

sonOffset/StoreAPI.PAGESIZE,sonOffset%StoreAPI.PAGESIZE);

ArrayList<String > son\_tuple = MEM.Next();

son\_tuple = compute(end\_point,son\_tuple);

res.add(getCalc(son\_tuple,son\_title));

}

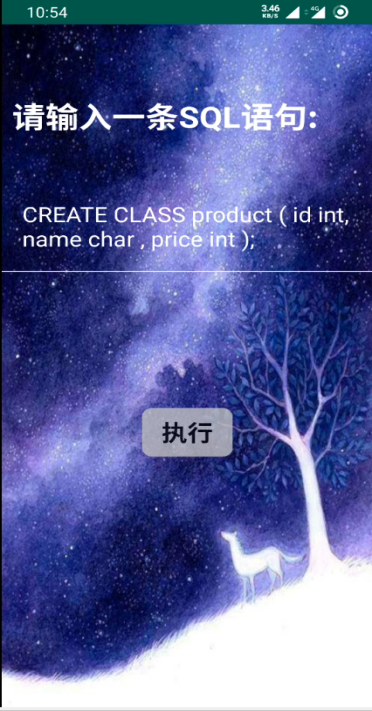
}

return res;

# 第三章 系统实现与测试

## 3.1 系统实现

软件界面如下图。



在输入框中输入sql语句后点击“执行”按钮。在界面下方会出现提示信息（语句执行成功或失败，如语法错误则报invalid syntax）。



如果输入的sql语句是select语句且语法正确，则点击“执行”按钮后会出现如下界面，显示select语句和所求表项。



## 3.2 系统测试

测试样例语句如下

第1行：创建product类，属性有id，name和price

第2-6行：向product类添加5条记录

第7行：查询此时product表项



第8行：删除product中name为“mi”的记录

第9行：将product中name为“iphone”记录的price更新为4900

第10行：查询此时product表项

第11行：创建product的代理类usproduct，属性有sales，name和usprice（price/7）；筛选条件为product中price>5000的记录

第12行：查询此时usproduct表项



第13行：将usproduct中name为“ipad”记录的sales更新为3000

第14行：将usproduct中name为“mac”记录的sales更新为2000

第15行：查询此时usproduct表项

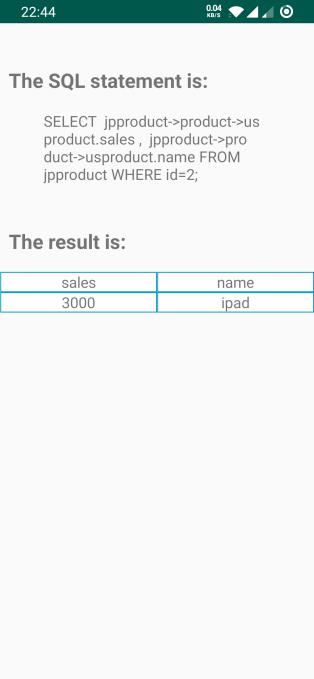


第16行：创建product的代理类jpproduct，属性有id和jpprice（price\*15）；筛选条件为product中price<=7000的记录

第17行：查询此时jpproduct表项



第18行：跨类查询 在jpproduct中id为2的记录在usproduct的sales值



第19行：向product添加一条新记录

第20行：查看此时usproduct表项



第21行：查看此时jpproduct表项



第22行：将product中name为“huawei”记录的price更新为4900

第23行：查看此时usproduct表项

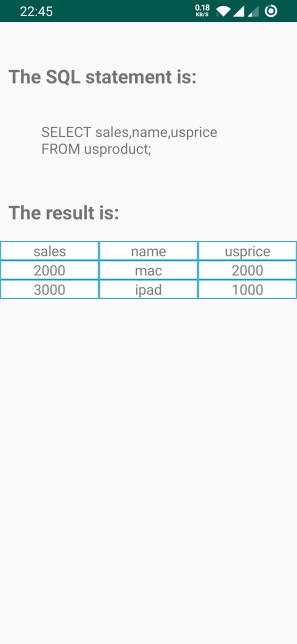


第24行：查看此时jpproduct表项



第25行：将produc中name为“huawei”记录的price更新为4500

第26行：查看此时usproduct表项



第27行：查看此时jpproduct表项

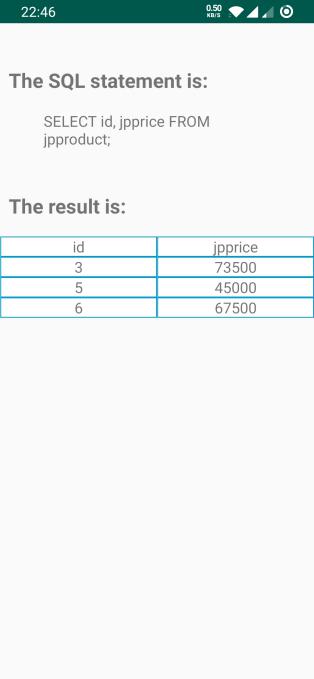


第28行：删除product中name为“ipad”的记录

第29行：查看此时usproduct表项



第30行：查看此时jpproduct表项



第31行：删除代理类usproduct

第32行：删除类product

**参考文献**

[1]《一个简单的Android端对象代理数据库系统的实现 - 浮生思潮》. [在线]. 载于: http://www.liyunzhe.cn/index.php/archives/89/. [见于: 10-2月-2020].

[2]J. D. Ullman, A first course in database systems. Pearson Education India, 1997.

[3]H. Garcia-Molina, J. D. Ullman和J. Widom, Database system implementation, 卷 672. Prentice Hall Upper Saddle River, NJ:, 2000.

[4]guanhuankang, guanhuankang/TOTEM. 2019.

[5]彭智勇和彭煜玮, PostgreSQL 数据库内核分析. Ji xie gong ye chu ban she, 2012.