TMDB 总体设计文档 v1.0

2022.4.3

• 界面设计

1. 定义：

activity_main.xml //主窗体布局
activity_show_class.xml //测试结果展示布局
print_result.xml //运行结果布局

2. 主要类

```
public class PrintResult extends AppCompatActivity {

    private final int W = ViewGroup.LayoutParams.WRAP_CONTENT;
    private final int M = ViewGroup.LayoutParams.MATCH_PARENT;
    private TableLayout rst_tab;

    @Override
    protected void onCreate(Bundle savedInstanceState){
        ...
        }

     public void Print(TupleList tpl,String[] attrname,int[] attrid,String[] type){
    ...
 }                                          //执行测试语句，显示测试结果


 public class ShowTable extends AppCompatActivity implements Serializable { //列出各表
    private String[] tables = new String[5];
    Context context;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
            ...
            public void onItemClick(AdapterView<?> adapterView, View view, int i, long l)
{
        ...

    }
    private void showTable(){
```

```
            ...
        }

private void printObj(ObjectTable topt){                    //传入 Obj 表参数
            ...
        }

        private void printSwi(SwitchingTable switchingT){          .//  传入 Swi 表参数
            ...
        }

        private void printDep(DeputyTable deputyt){               //传入 Dep 表参数
        ...
        }

        private void printBi(BiPointerTable biPointerT){           //传入 Bi 表参数
            ...
        }

        private void printCla(ClassTable classTable){              //传入 Cla 表参数
            ...
            }

}

        Show_xxx.java                                              //展示 xxx 表详细信息
```

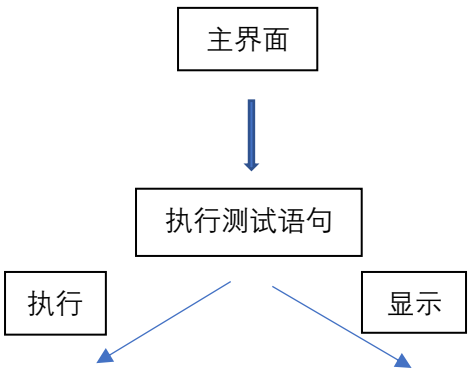
3. 扩展功能

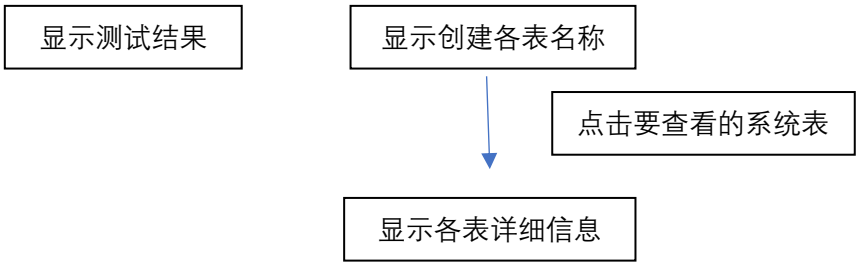
MusicSever.java　　//实现主界面背景音乐

定制主界面，将主界面各按钮设计的色彩丰富，更有艺术感并具有按压效果

定制展示界面，删除标题栏以使展示界面更加整洁，并使输出界面带有方框和分割线，并使展示栏与背景有颜色区分，看起来更加整洁明了

4. 功能实现





编译设计

1、 定义

对于 SQL 操作类别的宏定义

```
public static final int OPT_CREATE_ORIGINCLASS    = 1;
public static final int OPT_CREATE_SELECTDEPUTY   = 2;
public static final int OPT_DROP                  = 3;
public static final int OPT_INSERT                = 4;
public static final int OPT_DELETE                = 5;
public static final int OPT_SELECT_DERECTSELECT   = 6;
public static final int OPT_SELECT_INDERECTSELECT = 7;
public static final int OPT_CREATE_UPDATE         = 8;
```

2、 数据结构

以下是 javacc 词法定义

```
SKIP :
{
  " "
| "/t"
| "/n"
| "/r"
}

TOKEN:
{
<SEMICOLON:";">
|<CREATE:"CREATE">
|<DROP:"DROP">
|<CLASS:"CLASS">
|<INSERT:"INSERT">
|<INTO:"INTO">
|<VALUES:"VALUES">
|<LEFT_BRACKET:"(">
|<COMMA:",">
|<RIGHT_BRACKET:")">
|<DELETE:"DELETE">
|<FROM:"FROM">
```

```
|<WHERE:"WHERE">
|<SELECT:"SELECT">
|<SELECTDEPUTY:"SELECTDEPUTY">
|<ID: ["a"-"z"](["a"-"z","A"-"Z","0"-"9"])* >
|<EQUAL:"=">
|<INT: "0"|(["1"-"9"](["0"-"9"])*) >
|<STRING: "\""(["a"-"z","A"-"Z","1"-"9"])*"\"" >
|<CROSS:"->">
|<DOT:".">
|<AS:"AS">
|<PLUS:"+">
|<UPDATE:"UPDATE">
|<SET:"SET">
}
```

3、 模块

3.1 Run()为编译器开始语法符号 Run 识别时，程序进行的动作，程序最后返回将 SQL 转化成的 String 串

```
String[] Run():
{
    String[] s;//返回的字符串数组
}
{
  s = sql()
  {
    //System.out.println(s+"\n");
    return s ;
  }
}
```

3.2 sql()为编译器识别语法符号 sql 时，程序进行的动作，程序根据 SQL 语句的类别分成不同的处理方法

```
String[] sql() :
{
    String sql_s;
    String create_s;
    String drop_s;
    String select_s;
    String insert_s;
    String delete_s;
    String update_s;
}
{
   create_s = create() {sql_s =
```

```
create_s;System.out.println(sql_s+"\n");return sql_s.split(","); }
   |drop_s = drop() {sql_s = drop_s;System.out.println(sql_s+"\n");return
sql_s.split(","); }
   |select_s = select(){sql_s =
select_s;System.out.println(sql_s+"\n");return sql_s.split(",");  }
   |insert_s = insert2(){sql_s =
insert_s;System.out.println(sql_s+"\n");return sql_s.split(",");  }
   |delete_s = delete(){sql_s =
delete_s;System.out.println(sql_s+"\n");return sql_s.split(",");  }
   |update_s = update() {sql_s =
update_s;System.out.println(sql_s+"\n");return sql_s.split(","); }
}
```

3.3 编译器对语法符号 create 的操作，取出在队列中的 String 串，组合进返回的字符串 create_s 中，并返回

```
String create() :
{
    String create_s;//组合的字符串
    int count;//源类的属性数
}
{
    (<CREATE> count = originclass() <SEMICOLON>)
    {
      create_s = OPT_CREATE_ORIGINCLASS+",";
      create_s += count;
      while(!st.isEmpty())
      {
        create_s +=",";
        create_s += st.poll();

      }
      return create_s;
    }
    |
    (<CREATE> count = selectdeputy() <SEMICOLON>)
    {
      create_s = OPT_CREATE_SELECTDEPUTY+",";
      create_s += count;
      while(!st.isEmpty())
      {
        create_s +=",";
        create_s += st.poll();

      }
      return create_s;
```

```
    }

}
```

3.4 编译器对语法符号 drop 的操作，识别出类名 classname，组合出返回的字符串 drop_s 中，并返回

```
String drop() :
{
    String cln;//类名
    String drop_s;//返回的字符串
}
{
    <DROP> <CLASS> cln = classname() <SEMICOLON> {drop_s =
OPT_DROP+","+cln;return drop_s; }
}
```

3.5 insert()为编译器在处理 insertSQL 语句时的操作，将全局队列 st 中的 String 串组合进一个 String insert_s 中，并返回之。

```
String insert2():
{
    String insert_s;//返回的字符串
    int count;//数量
}
{
    count = insert()
    {
      insert_s = OPT_INSERT+",";
      insert_s += count;
      while(!st.isEmpty())
      {
        insert_s +=",";
        insert_s += st.poll();

      }
      return insert_s;
    }
}
```

3.6 insert()为编译器在处理 insertSQL 语句时的操作，识别出类名 classname 和值 value，依次进入全局队列 st 中，并返回 classname 与 value 的对数。

```
int insert() :
{
    String cln;//类名
    String vl;//值
```

```
    int count = 0;//值的数量
}
{
    <INSERT>
    <INTO>
    cln = classname(){st.add(cln);}
    <VALUES>
    <LEFT_BRACKET>
     vl = value() {count++;st.add(vl); }
     (<COMMA> vl = value() {count++;st.add(vl); })*
    <RIGHT_BRACKET>
    <SEMICOLON> {return count; }
}
```

3.7 delete()为编译器在处理 delete SQL 语句时的操作，识别出删除的类名 classname 和删除需要满足的条件。

```
String delete() :
{
    String cln;//类名
    String delete_s;//返回的字符串
    String cond;//条件字符串
}
{
    <DELETE> {delete_s = OPT_DELETE+","; }
    <FROM>
    cln = classname() {delete_s += cln+","; }
    <WHERE>
    cond = condition() {delete_s += cond;}
    <SEMICOLON> {return delete_s; }
}
```

3.7 select()为编译器在处理 select SQL 语句时的操作，从全局栈 st 中取出的类名 classname 和需要满足的条件，并组合进一个 String select_s 并返回之。

```
String select() :
{
    String select_s;//返回的字符串
    int count;//属性数量
}
{
    (<SELECT> count = directselect() <SEMICOLON>)
    {
      select_s = OPT_SELECT_DERECTSELECT+",";
      select_s += count;
      while(!st.isEmpty())
```

```
      {
        select_s +=",";
        select_s += st.poll();

      }
      return select_s;
    }
    |
    (<SELECT> count = indirectselect() <SEMICOLON>)
    {
      select_s = OPT_SELECT_INDERECTSELECT+",";
      select_s += count;
      while(!st.isEmpty())
      {
        select_s +=",";
        select_s += st.poll();
      }
      return select_s;
    }
}
```

3.8 originclass()是编译器处理源类语法符号的操作，它会把该源类的类名 classname，属性名 attr，属性类 attrtype，加入全局栈 st，并返回他们一共多少对的整数 count，便于后续使用

```
int originclass() :
{
    String cln;//类名
    String attr_s;//属性
    String attrtype_s;//属性类
    int count = 0;//属性数量
}
{
    <CLASS>
    cln = classname() {st.add(cln); }
    <LEFT_BRACKET>
    attr_s = attr()  {st.add(attr_s); }
    attrtype_s = attrtype() {st.add(attrtype_s); count++; }
    (<COMMA> attr_s = attr() {st.add(attr_s); } attrtype_s = attrtype()
{st.add(attrtype_s); count++; })*
    <RIGHT_BRACKET>
    {return count;}

}
```

3.9 选择代理类操作，识别类名 cln，添加进全局栈 st

```
int selectdeputy() :
```

```
{
    String cln;//类名
    int count;//直接选择的属性数
}
{
    <SELECTDEPUTY> cln = classname() {st.add(cln); }  <SELECT> count =
directselect() { return count;}
}
```

3.10 classname()为类名的处理，获得词法分析器对应的 String

```
String classname() :
{  Token s;}
{
   s=<ID>
   { return s.image;}
}
```

3.11 attrtype()为属性类别的处理，获得词法分析器对应的 String

```
String attrtype() :
{  Token s;}
{
   s=<ID>
   { return s.image;}
}
```

3.12 attr()为属性名的处理，获得词法分析器对应的 String

```
String attr() :
{  Token s;}
{
   s=<ID>
   { return s.image;}
}
```

3.12 dattr()为属性名的处理，获得词法分析器对应的 String

```
String dattr():
{  Token s;}
{
   s=<ID>
   { return s.image;}
}
```

3.13 condition()为对条件的分析，返回代表条件的字符串

```
String condition() :
{
    String attr_s;
    String cond;
```

```
    Token s;
}
{
    (attr_s = attr() <EQUAL> s = <INT>) {cond = attr_s+",=,"+s.image;
return cond;}
    |
    (attr_s = attr() <EQUAL> s = <STRING>) { cond = attr_s+",=,"+s.image;
return cond;}
}
```

3.14 value()为对值（整数或者字符串）的分析，返回代表值的字符串

```
String value() :
{
    Token s;
}
{
    s = <INT> {return s.image; }
    |
    s = <STRING> { return s.image; }
}
```

3.15 directselect()处理的是“直接选择语句”，它将对应属性 attr,值 value，重命名的属性 dattr 还有类名 cln 和条件都添加进入全局栈 st 中，并返回属性数 count。

```
int directselect() :
{
    String dattr_s;//重命名属性
    String attr_s;//属性名
    String value_s = "";//值
    boolean plus = false;//是否在取出值后增加值
    String cln;//类名
    String cond;//条件
    int count = 0;//属性数
}
{
    attr_s = attr() {st.add(attr_s);count++;}
    (< PLUS >  value_s = value() {plus = true; })? {if(plus) { st.add("1");
st.add(value_s);}else {st.add("0"); st.add("0"); } plus = false; }
    <AS>
    dattr_s = dattr() {st.add(dattr_s);}
    (<COMMA>  attr_s = attr() {st.add(attr_s);count++; } (< PLUS >  value_s
= value() {plus = true; })?{if(plus) { st.add("1"); st.add(value_s);}else
{st.add("0"); st.add("0"); } plus = false; } <AS> dattr_s =
dattr(){st.add(dattr_s);})*
    <FROM>
```

```
    cln = classname() {st.add(cln); }
    <WHERE>
    cond = condition() {st.add(cond);return count; }
}
```

3.16 indirectselect()处理的是“间接选择语句”，它将类名 cln，对应属性 attr,选择的类名还有和条件 condition 都添加进入全局栈 st 中，并返回类数 count。

```
int indirectselect() :
{
    String cln;//类名
    int count = 0;//类数
    String attr_s;//属性
    String cond;//条件
}
{
    cln = classname() {st.add(cln);count++; }
    (<CROSS> cln = classname() {st.add(cln);count++; })*
    <DOT>
     attr_s = attr() {st.add(attr_s); }
     <FROM>
     cln = classname() {st.add(cln); }
     <WHERE>
     cond = condition() {st.add(cond); return count;}
}
```

3.17 update()处理的是“更新语句”，它将对应类名 cln，值 value_s，属性 attr_s，条件 cond 直接合成返回的字符串 update_s。

```
String update():
{
    String cln;//类名
    String value_s;//值
    String attr_s;//属性
    String cond;//条件
    String update_s;//返回字符串
}
{
  <UPDATE> { update_s = OPT_CREATE_UPDATE+",";}
  cln = classname() {update_s += cln + ","; }
  <SET>
  attr_s = attr() { update_s += attr_s+","; }
  <EQUAL>
  value_s = value() { update_s += value_s+","; }
  <WHERE>
  cond = condition() { update_s += cond;}
  <SEMICOLON> { return update_s;}
}
```

• SQL 编译执行

1．数据结构

1. Class 系统表

classname	类名
classid	类 id
attrnum	类属性个数
attrid	属性 id
attrname	属性名
attrtype	属性类型
classtype	类类型

2. Deputy 表

originid	类 id
deputyid	代理类 id
deputyrule	代理规则

3. Object 表

classid	类 id
tupleid	元组 id
blockid	块 id
offset	块内偏移

4. Swiching 表

attr	源属性 id
depuattr	代理属性
rule	Swich 规则

5. BiPointer 表

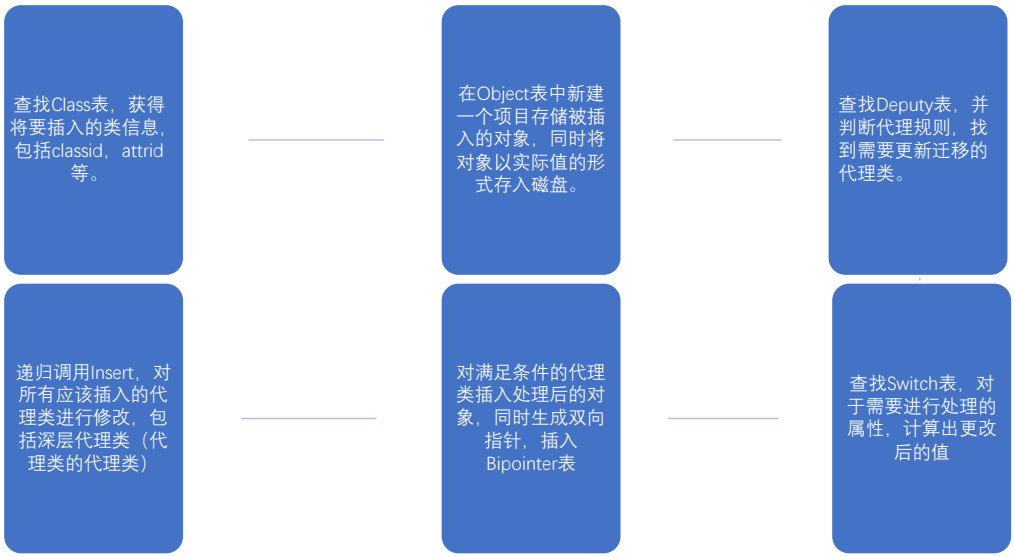
classid	源类 id
objectid	源对象号
deputyclassid	代理类 id
deputyobjectid	代理对象号

2. 主体逻辑

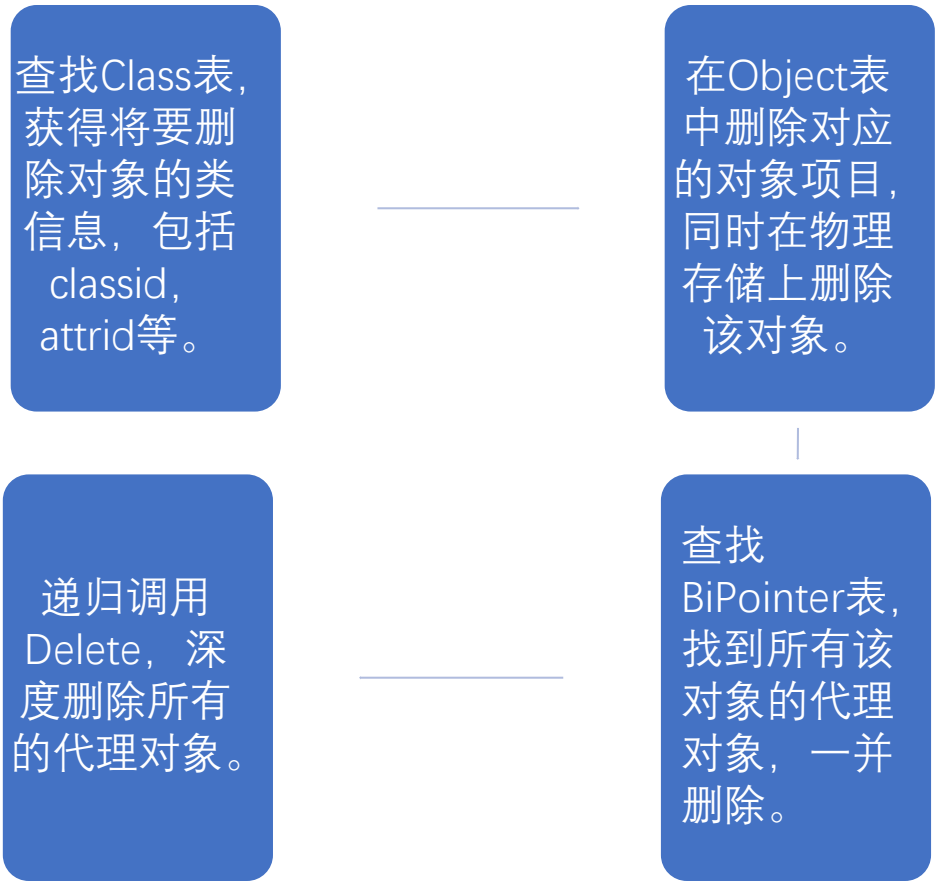
(1). 创建源类

在class表中加入新项目，存储类信息，分配classid。其中类类型置为“ori”。

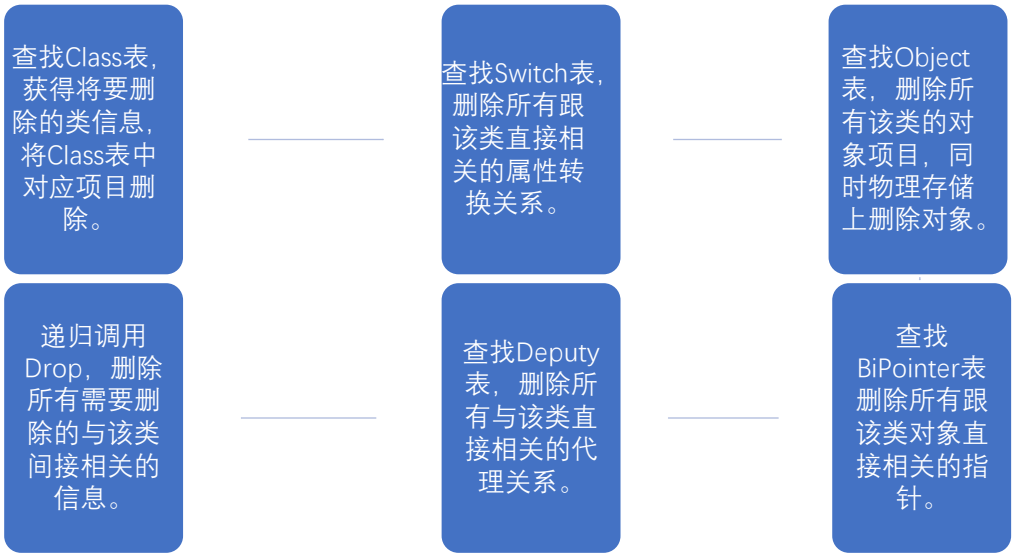
(2). 插入



(3). 删除对象

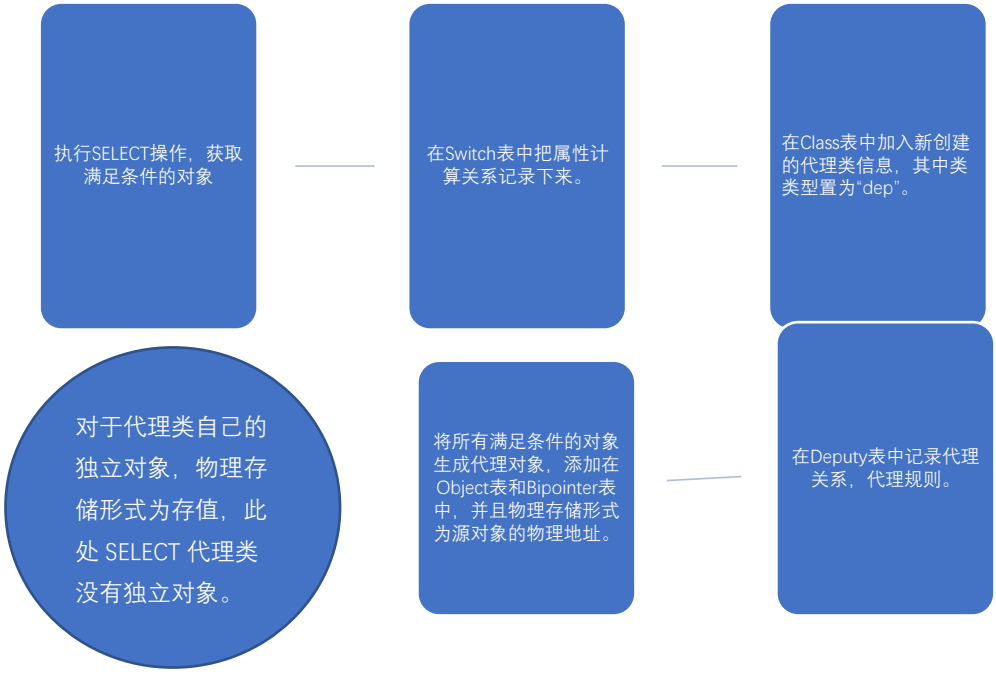

查找Class表，获得将要删除对象的类信息，包括classid，attrid等。
在Object表中删除对应的对象项目，同时在物理存储上删除该对象。
递归调用Delete，深度删除所有的代理对象。
查找BiPointer表，找到所有该对象的代理对象，一并删除。

(4). 删除类

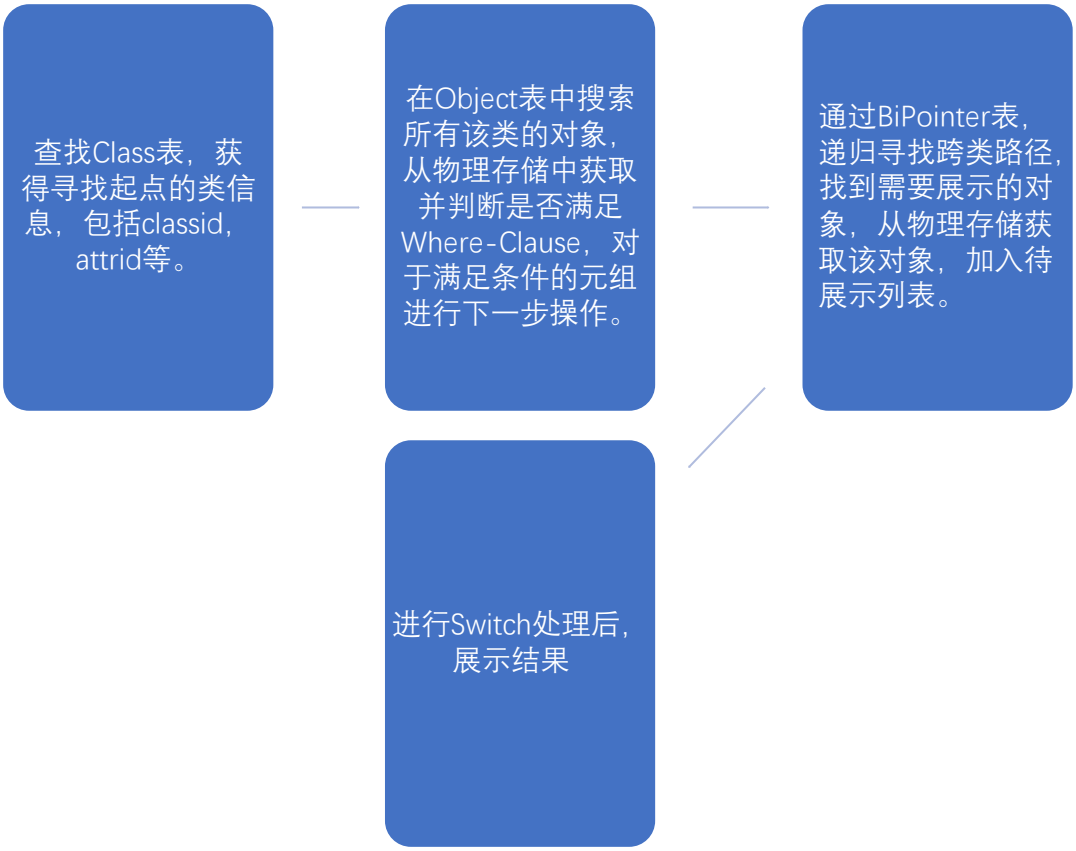

查找Class表，获得将要删除的类信息，将Class表中对应项目删除。
查找Switch表，删除所有跟该类直接相关的属性转换关系。
查找Object表，删除所有该类的对象项目，同时物理存储上删除对象。
递归调用Drop，删除所有需要删除的与该类间接相关的信息。
查找Deputy表，删除所有与该类直接相关的代理关系。
查找BiPointer表删除所有跟该类对象直接相关的指针。

(5). 查找

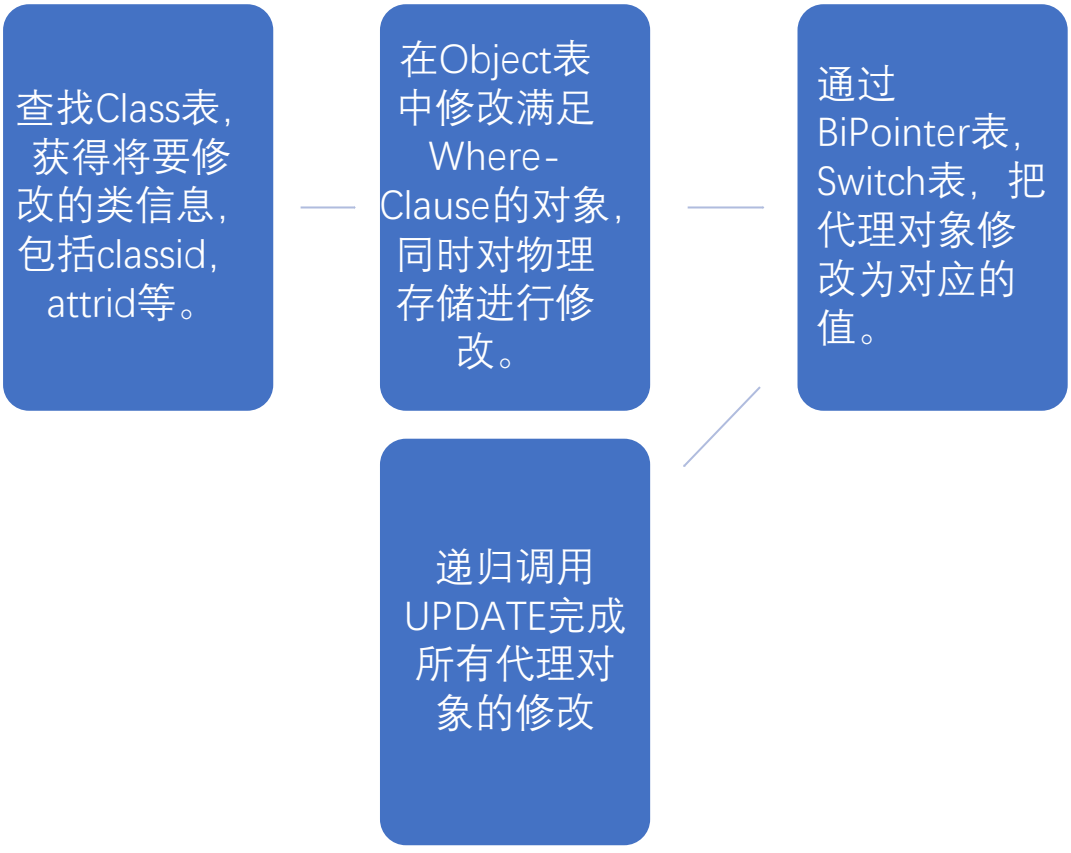
6. 创建 SELECT 代理类



7. 跨类查找


查找Class表，获得寻找起点的类信息，包括classid，attrid等。
在Object表中搜索所有该类的对象，从物理存储中获取并判断是否满足Where-Clause，对于满足条件的元组进行下一步操作。
通过BiPointer表，递归寻找跨类路径，找到需要展示的对象，从物理存储获取该对象，加入待展示列表。
进行Switch处理后，展示结果

8. 修改属性


查找Class表，获得将要修改的类信息，包括classid，attrid等。
在Object表中修改满足Where-Clause的对象，同时对物理存储进行修改。
通过BiPointer表，Switch表，把代理对象修改为对应的值。
递归调用UPDATE完成所有代理对象的修改

3.具体实现

3.1 主要函数

```
//创建源类
private void CreateOriginClass(String[] p)
//插入
private int Insert(String[] p)
//判断 WHERE 语句是否成立
private boolean Condition(String attrtype,Tuple tuple,int
attrid,String value1)
//删除
private void Delete(String[] p)
//递归实现代理删除
private OandB DeletebyID(int id)
//删除表
private void Drop(String[]p)
//递归代理删除表
private List<DeputyTableItem> Drop1(String[] p)
//查找
private TupleList DirectSelect(String[] p)
//创建代理类
private void CreateSelectDeputy(String[] p)
//跨类查询
private TupleList InDirectSelect(String[] p)
//修改
private void Update(String[] p)
//递归实现代理修改
private void UpdatebyID(int tupleid,int attrid,String value)
```

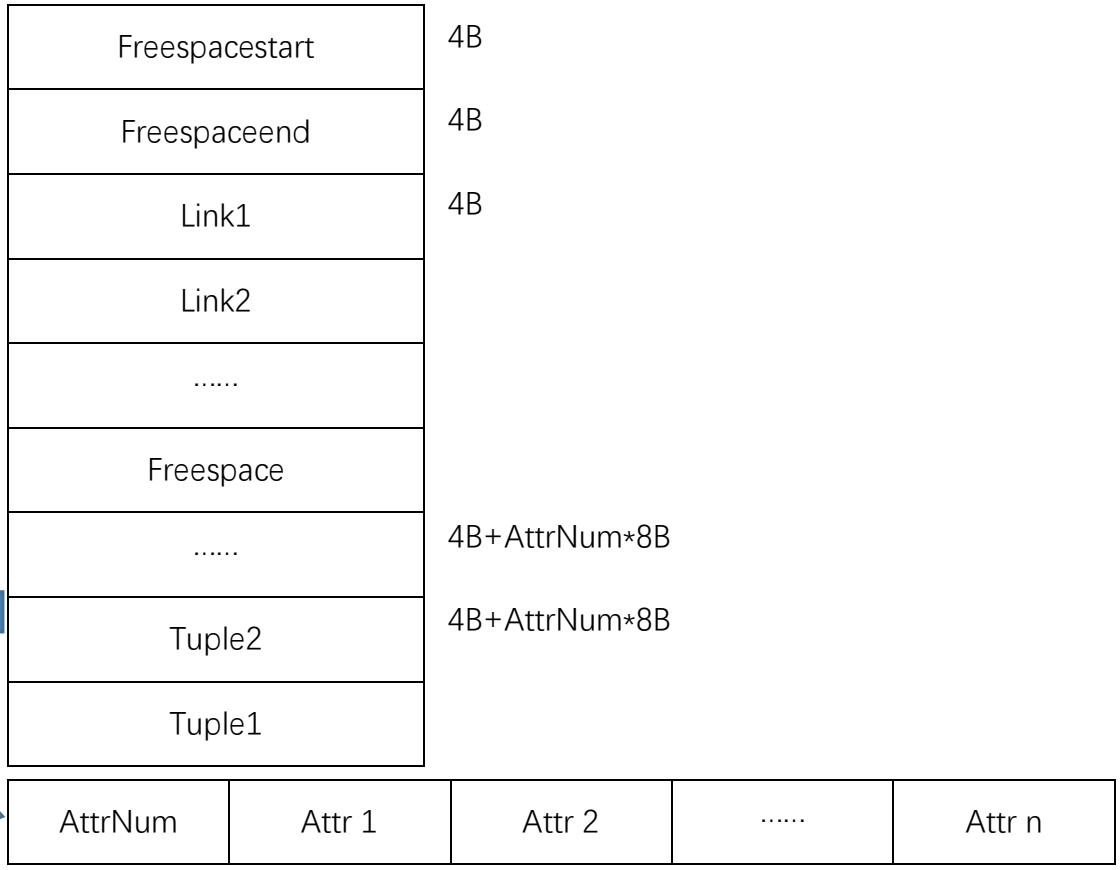
- 存储管理

1.文件结构

1.1 磁盘数据存放

8KB

4B



1.2 缓冲区

8KB
总大小 8KB*1000=8000KB

缓冲区 0	缓冲区 1	缓冲区 2	……	缓冲区 999

2.定义

```
//缓冲区指针
class buffPointer {
    int blockNum;      //块号
    Boolean flag;      //标记改块是否为脏（true 为脏）
    int buf_id;        //缓冲区索引号
}
final private int attrstringlen=8; //属性最大字符串长度为 8Byte
final private int bufflength=1000;//缓冲区大小为 1000 个块
final private int blocklength=8*1024;//块大小为 8KB

private List<buffPointer> BuffPointerList = new ArrayList<>();          //构建缓冲区指针表
private ByteBuffer MemBuff=ByteBuffer.allocateDirect(blocklength*bufflength);// 分配 blocklength*bufflength 大小的缓冲区
private boolean[] buffuse=new boolean[bufflength];//缓冲区可用状态表，true 为可用
private int blockmaxnum=-1;//最大的块号
private int[] blockspace=new int[10];//块空闲空间信息
```

3.模块

3.1 数据编码模块

```
//编码字符串为 byte
private byte[] str2Bytes(String s)
//解码 byte 为字符串
private String byte2str(byte[] b,int off,int len)
//编码 int 为 byte
private byte[] int2Bytes(int value, int len)
//解码 byte 为 int
private int bytes2Int(byte[] b, int start, int len)
```

3.2 磁盘文件操作模块

```
//将缓冲区指针所指的块存入磁盘
```

private boolean save(buffPointer blockpointer)
//根据块号加载块到缓冲区
private buffPointer load(int block)
//将每一个块的空闲空间大小存入磁盘
private boolean saveBlockSpace()
//从磁盘将没一个块的空闲空间大小加载到内存
private void loadBlockSpace()

3.3 缓冲区管理

//新建块
private buffPointer creatBlock()
//从缓冲区中根据块号寻找块，找不到返回 null，找到返回该缓冲区块的指针
private buffPointer findBlock(int x)
//初始化缓冲区使用位图，程序运行时初始化全为 true 表示可用
private void initbuffues()
//更新缓冲区指针序列：将 p 置为缓冲区列表首位（为实现缓冲区块 LRU 置换算法）
private void updateBufferPointerSequence(buffPointer p)
//将执行层传入的元组写入缓冲区，并返回块号和偏移
public int[] writeTuple(Tuple t)
//根据执行层传入的块号和偏移，读取相应元组并返回
public Tuple readTuple(int blocknum,int offset)
//根据执行层传入的旧元组块号、偏移以及新的元组，对元组进行更新修改
public void UpateTuple(Tuple tuple,int blockid,int offset)
//将缓冲区刷新到磁盘
public boolean flush()

4.其他数据存储接口

4.1 执行层系统表存储接口

4.1.1 ObjectTable

public boolean saveObjectTable(ObjectTable tab)　//将执行层传入的 ObjectTable 存入磁盘
public ObjectTable loadObjectTable()//从磁盘加载 ObjectTable 传给执行层

4.1.2 ClassTable

public boolean saveClassTable(ClassTable tab) //将执行层传入的 ClassTable 存入磁盘
public ClassTable loadClassTable()　//从磁盘加载 ClassTable 传给执行层

4.1.3 DeputyTable

public boolean saveDeputyTable(DeputyTable tab) //将执行层存入的 DeputyTable 存入磁盘
public DeputyTable loadDeputyTable()　　//从磁盘加载 DeputyTable 传给执行层

4.1.4 BiPointerTable

public boolean saveBiPointerTable(BiPointerTable tab)　//将执行层传入的 BiPointerTable 存入磁盘
public BiPointerTable loadBiPointerTable()//从磁盘加载 BiPointerTable 传给执行层

4.1.5 SwitchingTable

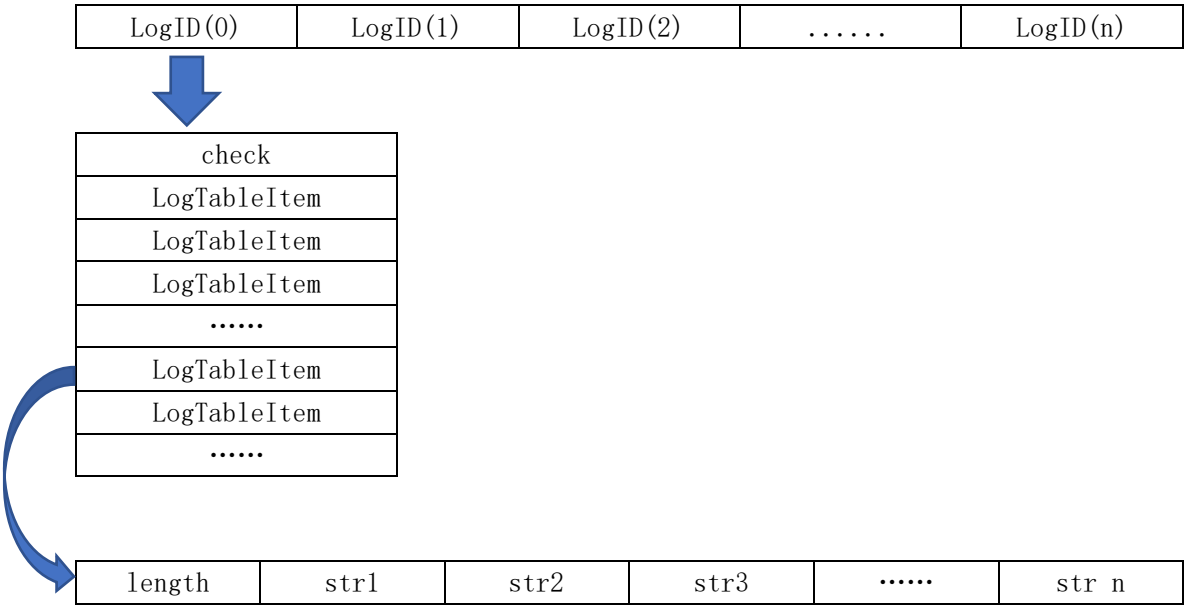
public boolean saveSwitchingTable(SwitchingTable tab) //将执行层传入的 SwitchingTable 存入磁盘
public SwitchingTable loadSwitchingTable()　　//从磁盘加载 SwitchingTable 传给执行层

4.2 日志层相关文件存储接口

//将日志层传入的日志块存入磁盘
public boolean saveLog(LogTable log)
//从磁盘加载日志块传给日志层
public　LogTable loadLog(int logid)
//设置日志块检查点为 1
public boolean setLogCheck(int logid)
//设置检查点号
public boolean setCheckPoint(int logid)
//加载日志检查检查点号给日志层
public int loadCheck()

• 事务管理

1,数据结构




```
public class LogTable {
    public int check=0;  //检查点(最大已确认块号)
    public int logID=0;  //日志块号
    public List<LogTableItem> logTable=new ArrayList<>(); //列表
}

public class LogTableItem {
    public int length=0;    //记录长度
    public String str=null; //需要记录的操作
}
```

2,定义

```
final private int MAXSIZE=5;         //设定的阈值，实现定期同步
private int checkpoint=-1;       //初始化检查点为-1
private int logid=0;             //LogTable 块 id
private TransAction trans;       //需要获得执行层内容
public LogTable LogT = new LogTable();   //存放执行层创建 LogManage 时写入的日志
```

3，模块

3.1 主要函数

```
//若达到阈值，需要调用该方法，初始化 LogT 为空
private boolean init(){
    new 一个 LogTable 对象;
}

//得到检查点号
private int GetCheck(){
    调用存储层函数;
}

//load 日志块，找出需要 redo 的命令
public LogTable GetReDo(){
    LogTable ret = null;
    得到检查点:checkpoint;
    ret = checkpoint+1 块中加载可能 redo 的日志;
    If(该块非空){
        把该块中的语句存入 ret；
        ret.logID=checkpoint+1;
    }
    return ret;
}

//写一条日志
public boolean WriteLog(String s){
    lognum=获得当前对象的 logtable 中有几条语句;
    把语句传入 logItem;
    if(lognum<MAXSIZE){
        把 logItem 添加到 logtable;
        If(lognum=MAXSIZE-1){
            保存当前日志表和五张系统表;
            当数据缓冲区刷入磁盘和该块检查点设置为 1 后;
            把该块块号设为当前最大检查点号;
        }else{
            保存当前日志表和五张系统表;
            当数据缓冲区刷入磁盘和该块检查点设置为 1 后;
            把该块块号设为当前最大检查点号;
            init();
            把语句加入新块;
            设置该块块号;
        }
    }

}
```

3.2 扩展功能

```
//删除日志文件
public void DeleteLog()

//分配事务 ID，用于多事务系统使用
private void AllocateTID()
```

4，功能

从日志中恢复数据，redo 未完成的操作，定期与数据表同步，保证一致性。
扩展功能：虽然 app 运行时只有一个事务，但是可以扩展到多事务执行，日志作为中间层，还应具有维护操作原子性的功能，所以还可以加入事务表，记录事务号和对应的日志块号，并且在每一个事务开始时为事务分配事务 id。当日志文件过多后，用户也可以选择手动删除.