西安电子科技大学

软件方向综合工程设计

题　　目　关系代数运算系统

学　　院　计算机科学与技术

专　　业　计算机科学与技术

学号、姓名　薛晴　19030100069

学号、姓名　王禹轩　19030400025

学号、姓名　陈曼琪　19030100272

导师姓名　赵亮

摘要

主要从开题报告、概要设计、详细设计、软件测试等方面分章节书写软件工程设计文档。

关键词：关系代数 词法分析 语法分析 逆波兰表达式 表达式计算

目录

第一章 选题报告

1.1 选题内容

1.1.1 选题名称

关系代数运算系统

1.1.2 总体需求

a. 对关系代数表达式进行运算，给出结果
b. 有一个对用户友好的界面

1.2 开发思路

1.2.1 主要功能

a. 后端：
    1. 接收前端输入框输入的数据，判断合理性后进行运算，返回结果。
    2. 完成关系代数的运算逻辑，包括并、交、差、笛卡尔积、选择、投影、连接、除。
b. 前端：
    1. 将输入框数据传送至后端，接收并显示返回的结果。
    2. 完成一个简洁美观、用户友好的界面。

1.2.2 开发平台

Windows 10/11

IntelliJ IDEA x64

第二章 概要设计

2.1 概述

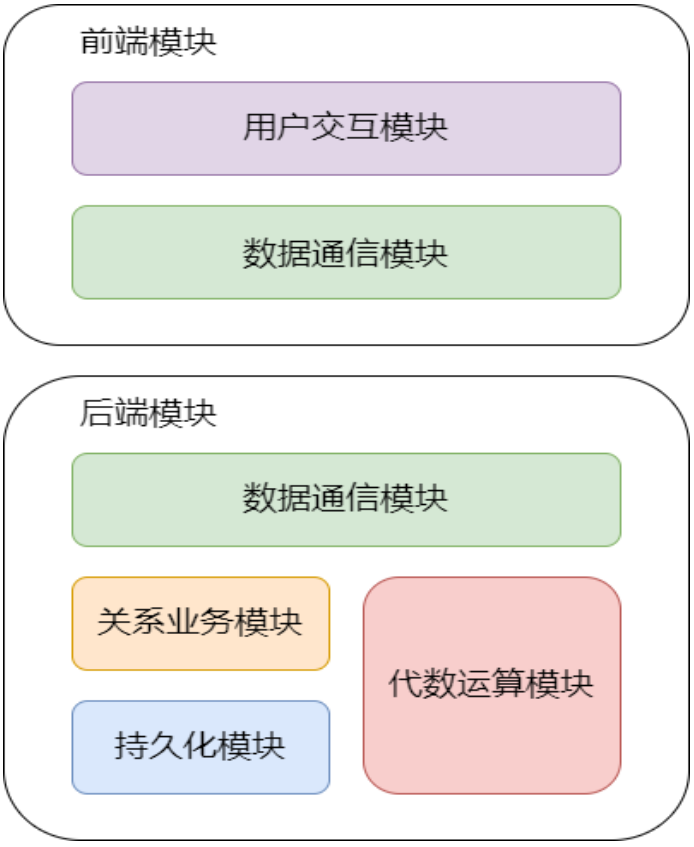
设计并实现一个具有良好交互界面的**关系代数运算系统**。系统能完成关系代数的运算逻辑，包括并、交、差、笛卡尔积、选择、投影、连接、除这 8 种运算。

经商议，为了给予用户良好的交互界面，该系统采取前后端分离的设计模式。

2.2 系统结构设计

2.2.1 软件总体结构和功能模块设计

软件总体结构宏观上可以分为前端模块与后端模块两个，两个模块下又各有若干个功能模块。



2.2.2 各项功能与程序结构的关系

程序结构由前后端两个模块构成，各个模块以及子模块的作用如下。

a. 前端模块

1. 用户交互模块：负责与用户的直接交互，用户通过该模块向系统输入，同时系统将输入处理成结果，通过该模块向用户呈现。

2. 数据通信模块：负责前后端数据的通信，功能包括关系信息的传输、计算表达式的传输等。将用户输入的数据打包交由后端，将后端的响应处理后交由交互模块展现。

b. 后端模块

1. 数据通信模块：负责前后端数据的通信，接收前端传来的数据，并将数据交由业务模块或是运算模块，同时将下层模块的处理结果打包返回前端。

2. 关系业务模式：处理关系的存储。用户在前端进行关系的新增与删除时，通过该模块与持久化模块进行通信，增加或删除对应的关系。

3. 持久化模块：负责已有关系的持久化。

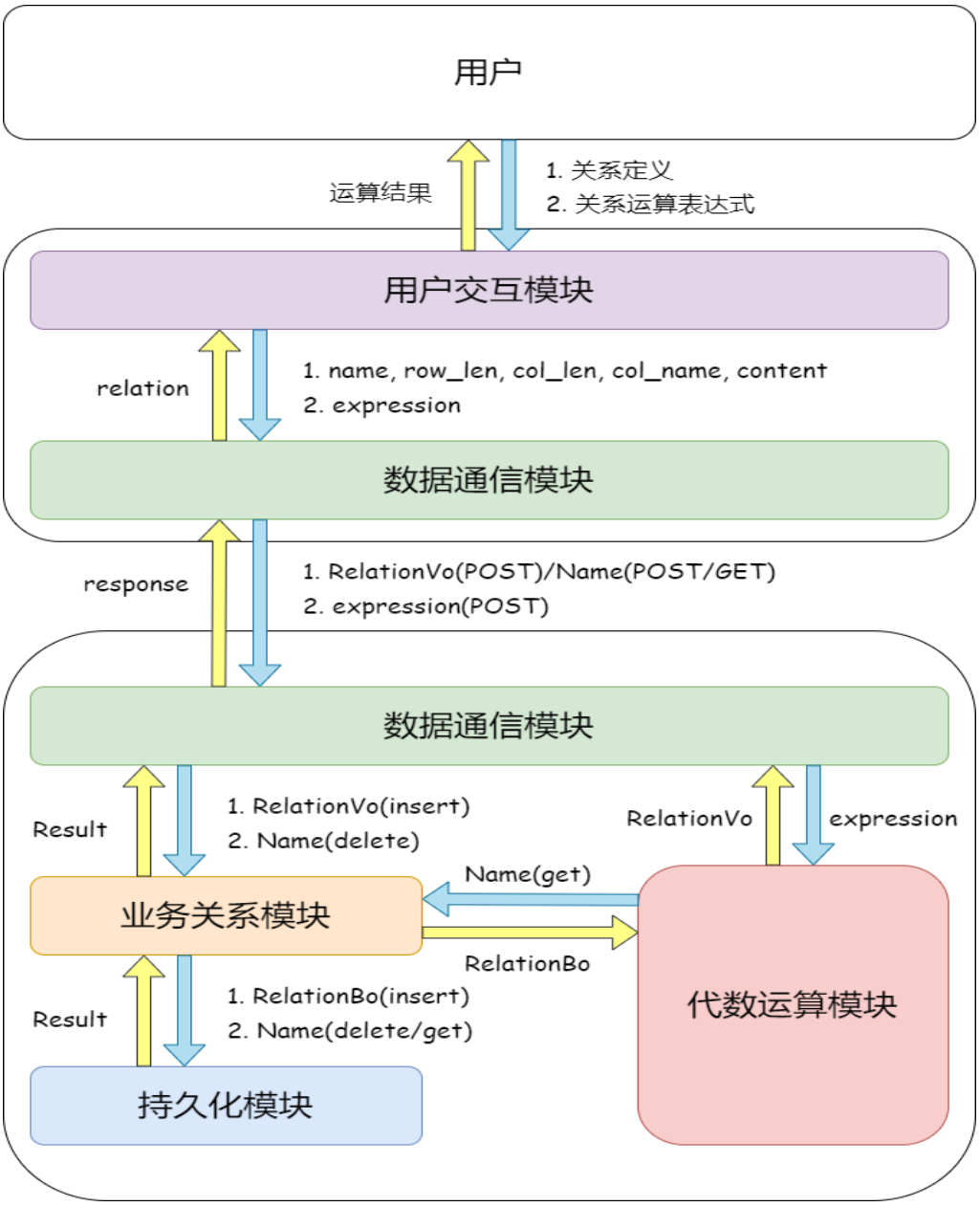
4. 代数运算模块：接收用户输入的运算表达式并进行计算，并向数据通信模块返回结果。

2.2.3 模块之间的层次结构以及模块间的调用关系

用户与前端模块中的用户交互子模块直接交互，包含关系的定义、关系运算表达式输入、结果输出等。前后端之间通过数据通信模块进行交互，负责将用户的输入结果按规定格式传递至后端、将计算结果按规定格式传递至前端。

业务关系模块负责处理关系的新增/删除等操作，与数据通信模块交互以获取数据，与代数运算模块交互以提供关系信息。持久化模块则为保存用户事先定义的

关系，与业务关系模块交互。代数运算模块则借助业务关系模块与数据通信模块完成表达式的计算以及返回结构。



2.3 接口设计

2.3.1 外部接口

外部接口主要为前端通过交互界面为用户提供的操作本系统的接口。

a. 关系管理接口

关系管理接口包含关系的新增、删除以及重置。用户可以通过交互界面进行关系的新增、关系的删除以及重置已保存的所有关系。

b. 表达式接口

表达式接口为用户呈现一个输入框，用户在此输入待运算的表达式，通过点击提交来获取系统的计算结果。系统通过该接口将运算结果向用户呈现。

2.3.2 内部接口

a. 后端程序内部模块间接口

后端程序中主要涉及到几个主要接口。ComputingService 用于表达式的计算，对外提供了以表达式字符串为入参计算表达式的接口。RelationService 用于关系业务的处理，对外暴露的接口可以执行关系的新增、删除、重置等操作。JudgementOfLegalityService 用于对输入表达式进行合法性判断。

1. RelationService 接口伪代码定义

```
interface RelationService {
    /**
     * @param relationVo 前端传来的Relations
     * @throws ParamLenException 参数不对时抛出此异常
     */
    void insertRelation(RelationVo relationVo);
    /**
     * 删除关系
     */
    void deleteRelation(String name);
    /**
     * 删除所有已建立关系
     */
```

```
    void deleteAll();
    /**
     * 是否存在名为 key 的关系
     * @param key
     * @return
     */
    boolean contains(String key);
    /**
     * 获取 name 对应的 Bo
     * @param name
     * @return
     */
    RelationBo get(String name);
}
```

2. ComputingService 接口定义

```
public interface ComputingService {
    /**
     * @param expression 关系代数表达式
     * @return 计算后的结果
     */
    RelationVo compute(String expression);
}
```

3. JudgementOfLegalityService 接口定义

```
public interface JudgmentOfLegalityService {
    /**
     * @param expression 待计算的表达式
     * @return 表达式是否合法
     */
    boolean judgeLegality(String expression);
}
```

b. 前后端接口定义

1. 新增关系

(1) 调用方式：POST

(2) URL：/api/insert/

(3) 入参：relation

(4) 出参：data 为空的 response

(5) 说明：新增关系，保存至后端。支持传入一个 Json 格式的关系。

例子如下。

入参

```
{
"relation_name":"student",
"row_len":3,
"col_len":2,
"col_name":"name,age,gender",
"content":"Johnny,18,male,Jack,20,male"
}
```

出参

```
{
"code":200,
"msg":"ok",
"data":{}
}
```

2. 删除关系

(1) 调用方式：POST

(2)URL：/api/delete/

(3)入参：关系名

(4)出参：data 为空的 response

(5)说明：删除关系。

例子如下。

入参

```
{
"name":"student"
}
```

出参

```
{
"code":200,
"msg":"ok",
"data":{}
}
```

3. 重置

(1)调用方式：GET

(2)URL：/api/delete_all/

(3)入参：无

(4)出参：data 为空的 response

(5)说明：删除已有的所有关系。

例子如下。

出参

```
{
"code":200,
"msg":"ok",
"data":{}
}
```

4. 计算关系代数

(1)调用方式：POST

(2)URL：/api/compute/

(3)入参：表达式字符串

(4)出参：data 为计算结果（一个 relation）的 response

(5)说明：计算关系代数表达式，内部出现的关系必须事先定义过。

关系运算符在表达式字符串中的表示对应关系如下：

运算符	含义	在字符串的表示	举例	说明
∪	并	#or	A #or B	
∩	交	#and	A #and B	
−	差	#diff	A #diff B	
×	笛卡尔积	#prod	A #prod B	
σ	选择	#select	#select[A,条件表达式,1]	参数为关系名、条件、固定参数
π	投影	#project	#project[A,name,age,2]	参数为关系名、投影列名、投影列数
⋈	连接	#join	A #join B	连接运算仅支持自然连接
÷	除	#div	A #div B	
(	左括号	(	-	-
)	右括号	)	-	-

共计 8 个运算类型：并、交、差、笛卡尔积、选择、投影、连接、除。

优先级定义：括号 > 选择 = 投影 > 连接 > 差 > 积 = 除 > 交 > 并

条件表达式中支持的运算符：>, <, =, >=, <=, !=。以及使用$and 与$or 来连接多个条件。

```
sage>=3|\$and|ssex=女|\$or|sage>=5|\$and|ssex=男
```

```
(|sage>=3|\$or|ssex=女|)|\$and|(|sage>=5|$or|ssex=男|)
```

例子如下。

入参

注意：任意两个符号（运算符、关系名或是括号）之间请添加一个英文空格

```
{
"expression":"( student #or class ) #and teacher",
"original_expression":"(student∪class)∩teacher"

}
```


出参


```
{
"code":200,
"msg":"ok",
"data":{
    "relation_name":"student",
    "row_len":3,
    "col_len":2,
    "col_name":"name,age,gender",
    "content":"Johnny,18,male,Jack,20,male"
    }

}
```


2.4 数据结构和数据库设计

本系统不涉及数据库设计，在此，主要展现一些计算过程中用到的数据结构。

2.4.1 后端

a. 关系

后端存储关系主要涉及两个类，RelationVo 以及 RelationBo，前者主要用于与前端交互时存储从前端送来的关系/准备送至前端的关系信息，后者则主要用于后端业务处理，包括关系的存储/表达式的运算等。

鉴于二者相似性，仅详细介绍 RelationBo 的成员，伪代码如下。

```
RelationBo {
    // 关系行长
```

```
    int rowLen;
    // 关系列长
    int colLen;
    // 关系列名
    String[] colName;
    // 关系内容，通过二维数组存储
    String[][] content;
    // bo 与 vo 相互转换的方法
    toRelationVo(RelationBo bo, String name);
    toRelationBo(RelationVo vo, String name);
}
```

b. 常量

此外，常量类 Constant，用于统一管理后端程序中用到的常量。

```
public class Constant {
    public static final String OR = "#or";
    public static final String AND = "#and";
    ...
    public static final Set<String> UNARY_OPERATOR;
    public static final Map<String, Integer> PRIORITY;
    public static final String TEMP_RELATION_PREFIX = "TEMP";
    ...
}
```

2.4.2 前后端交互

前后端交互主要用到两个数据结构。

a. relation 表示关系的数据结构（表）

包含五个字段：关系名、行长、列长、列名以及表的内容。

表的内容（content）使用一个字符串表示，每个元素间使用英文逗号分隔，应当满足字符串内元素的数量等于行长乘以列长。

举例，以下关系以及对应的 Json 结构：

name	age	gender
Johnny	18	male
Jack	20	male

```
{
"relation_name":"student",
"row_len":3,
"col_len":2,
"col_name":"name,age,gender",
"content":"Johnny,18,male,Jack,20,male"
}
```

b. response 响应

前端调用接口后返回的是一个 response 对象，包含三个字段：状态码、信息、数据体。如果是需要返回结果的请求调用，结果对象会存储在数据体中。

举例，一个请求成功，且返回数据存放在 data 中的 response 对象：

```
{
"code":200,
"msg":"ok",
"data":{
    "relation_name":"student",
    "row_len":3,
    "col_len":2,
    "col_name":"name,age,gender",
```

```
        "content":"Johnny,18,male,Jack,20,male"
        }
    }
```

2.4.3 前端

前端主要涉及两个数据结构，分别用于存储关系表及运算结果表。

a. 关系表

关系表用于存储用户输入的关系名，行数，列数，列名，关系表内容，便于后续发送至后端。同时使用 dis 标识该表的提交状态，避免多次提交同一关系表。

```
domains: [{
    name: '',
    row: '',
    col: '',
    col_name: '',
    text: '',
    dis: false
  }]
```

b. 运算结果表

运算结果表中，每行用 id 标识行号，id 依次递加。行中每项值使用 num 和 data 两个变量标识，其中 num 为列名，data 为该项值。

```
list: [{
    id: '',
    dataList: [{
      num: '',
      data: ''
    }]
```

2.5 编码规范

后端的编写语言为 Java，编码规范复用了阿里巴巴的规范，详见 Java 开发手册（嵩山版）.pdf。在此仅列举几个较为重要的编码规范：

a. 类名以驼峰式命名，变量以及方法名以首字母小写的驼峰式命名。

b. 后端内部接口命名统一以 Service 结尾，采用 Interface 结构，接口实现类统一以 Impl 结尾。

c. 后端数据通信层类名统一以 Controller 结尾。

前后端接口编码规范：

a. 接口 URL 地址统一接在/api/下，具体的接口名定义应该使用下划线式命名。如/api/insert_relation/。

b. 依据具体的业务场景，当使用 http 协议实现接口时，涉及到 Json 数据传输的应该使用 POST 方法，不涉及到数据传输的应该使用 GET 方法。

第三章 详细设计

3.1 全局变量和数据结构设计

3.1.1 全局变量

本程序中定义的全局变量有多个，包含程序执行过程中所需的各种步骤。

a. 关系表

关系表为后端程序中对关系进行持久化的数据结构。当涉及到关系的新增/删除，以及表达式的运算时，均需要与该表进行交互，进行关系的读与写。

该变量定义于接口实现类 RelationServiceImpl 下。呈现形式为 Hash 表的形式，其中 Key 为关系名，字符串类型，Value 为关系数据结构，RelationBo 类型。

```
private static final ConcurrentHashMap<String, RelationBo> relations;
```

b. 关系运算工具类

运算工具类 ComputingUtil 用于计算关系运算符的结果。其包含了 8 个对外暴露的方法，分别对应着不同的 8 种运算符。除此之外有诸多辅助计算方法，用于类内调用。

```
public class ComputingUtil {
    public static RelationBo and(RelationBo r1, RelationBo r2);
    public static RelationBo or(RelationBo r1, RelationBo r2);
    public static RelationBo diff(RelationBo r1, RelationBo r2);
    public static RelationBo prod(RelationBo r1, RelationBo r2);
    public static RelationBo div(RelationBo r1, RelationBo r2);
    public static RelationBo select(RelationBo r, String condition);
    public static RelationBo project(RelationBo r, int[] cols);
```

```
    public static RelationBo join(RelationBo r1, RelationBo r2);
}
```

c. 常量工具类

涉及到的常量主要定义为各运算符在表达式中的表示，符号表达的规则如下。通过#号开头以区分当前字符串为关系名还是运算符。

运算符	含义	在字符串的表示	举例	说明
∪	并	#or	A #or B	
∩	交	#and	A #and B	
–	差	#diff	A #diff B	
×	笛卡尔积	#prod	A #prod B	
σ	选择	#select	#select[A,条件表达式,1]	参数为关系名、条件、固定参数
π	投影	#project	#project[A,name,age,2]	参数为关系名、投影列名、投影列数
⋈	连接	#join	A #join B	连接运算仅支持自然连接
÷	除	#div	A #div B	
(	左括号	(	-	-
)	右括号	)	-	-

常量工具类 Constant 对上述运算符进行记录，以及记录一些其它会用到的常量，如运算符优先级、预测分析表等。

```
public class Constant {
    // 定义表达式的结束符
    public static final String END_SIGN = "#";
    // OR 运算符
    public static final String OR = "#or";
```

```
    ...
    // 优先级定义
    public static final Map<String, Integer> PRIORITY;
    ...
    // 预测分析表
    public static final Map<String, Map<String, String>>
PREDICTIVE_ANALYSIS_TABLE;
}
```

d. 表达式合法性判断模块

表达性合法性判断模块用于对输入的关系代数表达式进行词法分析以及语法分析，以判断输入表达式的合法性，保障后续计算过程中的正确性。该模块中含两个全局变量，为词法分析模块以及语法分析模块。

```
    @Resource
    LexicalAnalysisService lexicalAnalysisService;
    @Resource
    ParsingService parsingService;
```

3.1.2 数据结构

a. 后端

1. 关系表

后端存储关系主要设计两个类，RelationVo 以及 RelationBo，前者主要用于与前端交互时存储从前端送来的关系/准备送至前端的关系信息，后者则主要用于后端业务处理，包括关系的存储/表达式的运算等。

鉴于二者相似性，仅详细介绍 RelationBo 的成员，伪代码如下。

```
RelationBo {
// 关系行长
int rowLen;
```

```
    // 关系列长
    int colLen;
    // 关系列名
    String[] colName;
    // 关系内容，通过二维数组存储
    String[][] content;
    // bo 与 vo 相互转换的方法
    toRelationVo(RelationBo bo, String name);
    toRelationBo(RelationVo vo, String name);
}
```

2. 常量类

此外，常量类 Constant，用于统一管理后端程序中用到的常量。

```
public class Constant {
public static final String OR = "#or";
public static final String AND = "#and";
...
public static final Set<String> UNARY_OPERATOR;
public static final Map<String, Integer> PRIORITY;
public static final String TEMP_RELATION_PREFIX = "TEMP";
...
}
```

3. 预测分析表

预测分析表用于表达式合法性判断过程中的语法分析部分。常见的预测分析表为如下形式，行头和列头都为字符串形式。

	id	num	+	-	*	/	mod	(	)	;	#
L	E;L	E;L						E;L			ε
E	TE'	TE'						TE'			
E'			+TE'	+TE'					ε	ε	
T	FT'	FT'						FT'			
T'			ε	ε	*FT'	/FT'	mod FT'		ε	ε	
F	id	num						(E)			

考虑到预测分析表的特殊结构，在程序中，我们使用哈希表的形式来表示该数据结构。

```
public static final Map<String, Map<String, String>>
PREDICTIVE_ANALYSIS_TABLE;
```

4. 符号集合

此外，在一些场合需要对运算符进行分类判断，此时会用到符号集合。根据具体的应用场景，定义了不同的多种符号集合。为便于管理，其统一定义于常量类内。

```
// 终结符，在语法分析中使用到
public static final Set<String> TERMINATOR;
// 非终结符
public static final Set<String> NON_TERMINATOR;
```

b. 前后端交互

前后端交互主要用到两个数据结构。

1. relation 表示关系的数据结构（表）

包含五个字段：关系名、行长、列长、列名以及表的内容。

表的内容（content）使用一个字符串表示，每个元素间使用英文逗号分隔，应当满足字符串内元素的数量等于行长乘以列长。

举例，以下关系以及对应的 Json 结构：

name	age	gender
Johnny	18	male
Jack	20	male


```
{
"relation_name":"student",
"row_len":3,
"col_len":2,
"col_name":"name,age,gender",
"content":"Johnny,18,male,Jack,20,male"

}
```


2. response 响应

前端调用接口后返回的是一个 response 对象，包含三个字段：状态码、信息、数据体。如果是需要返回结果的请求调用，结果对象会存储在数据体中。

举例，一个请求成功，且返回数据存放在 data 中的 response 对象：


```
{
"code":200,
"msg":"ok",
"data":{
    "relation_name":"student",
    "row_len":3,
    "col_len":2,
    "col_name":"name,age,gender",
    "content":"Johnny,18,male,Jack,20,male"
    }

}
```


c. 前端

前端主要涉及两个数据结构，分别用于存储关系表及运算结果表。

1. 关系表

关系表用于存储用户输入的关系名，行数，列数，列名，关系表内容，便于后续发送至后端。同时使用 dis 标识该表的提交状态，避免多次提交同一关系表。

```
domains: [{
    name: '',
    row: '',
    col: '',
    col_name: '',
    text: '',
    dis: false
  }]
```

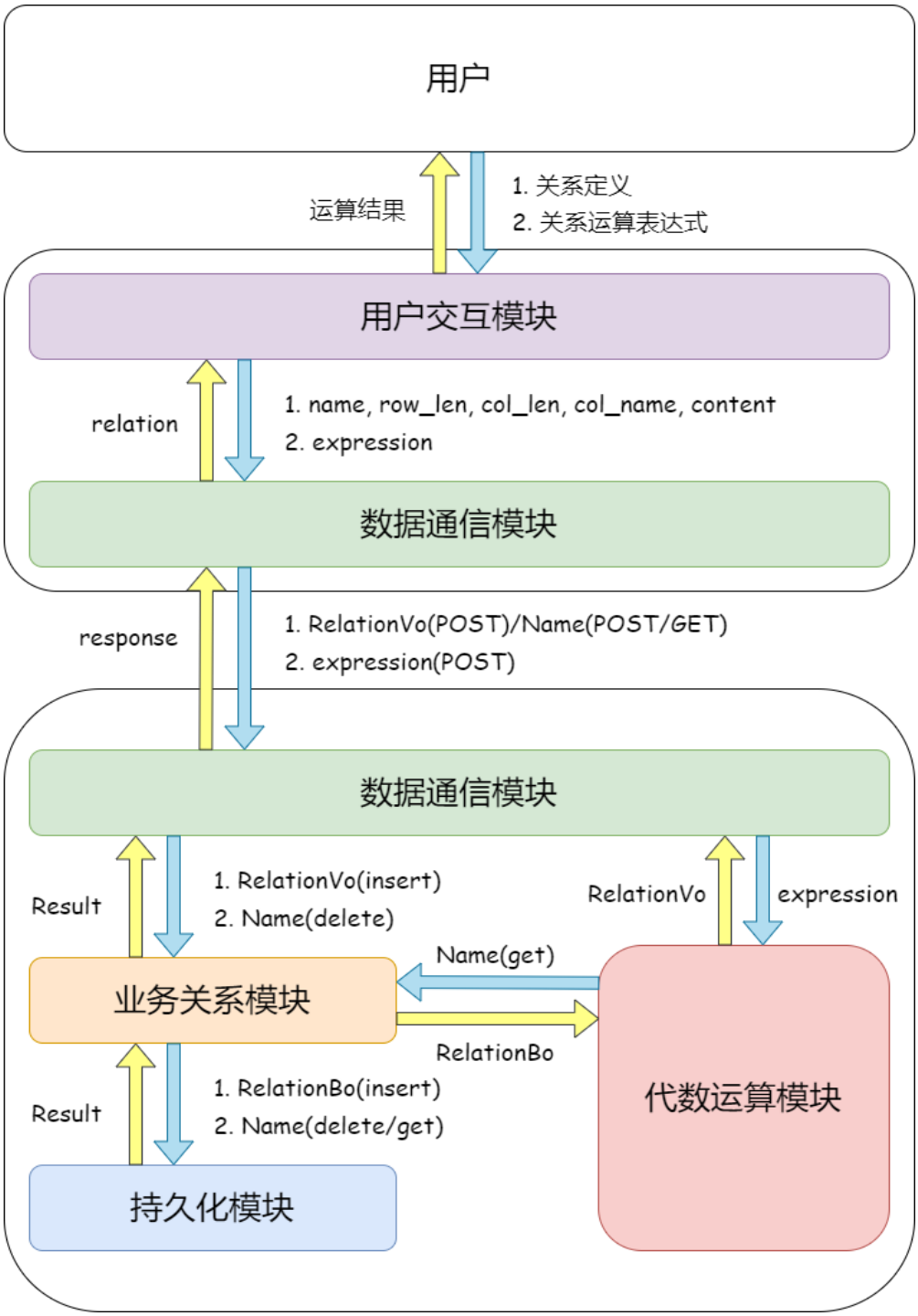
2. 运算结果表

运算结果表中，每行用 id 标识行号，id 依次递加。行中每项值使用 num 和 data 两个变量标识，其中 num 为列名，data 为该项值。

```
list: [{
    id: '',
    dataList: [{
      num: '',
      data: ''
    }]
```

3.2 各个功能模块设计

程序结构的各模块关系以及数据流关系如图所示。以下为每个模块单独进行详细的介绍。



3.2.1 用户交互模块

a. 功能描述

从属于前端模块，负责向用户提供基本的交互界面，负责提供整个应用程序的

对外用户接口。用户通过该模块可以执行关系的新增/删除以及表达式计算等操作，该模块也负责将计算结果为用户呈现。

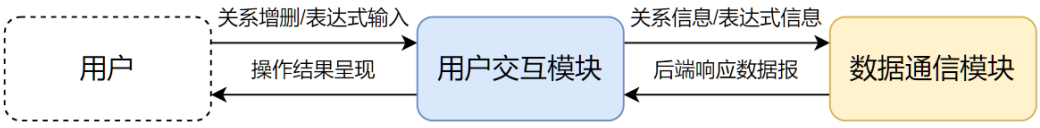
b. 输入数据

对于用户端而言，用户通过该模块将关系的信息（如关系名、行长、列长、内容等）输入系统；对于数据通信模块，数据通信模块将响应（包含请求结果、状态码等）交由用户交互模块处理。

输入数据格式是否符合规范要求不在本层做检验。

c. 输出数据

该模块负责将用户的输入数据按特定格式处理，交由数据通信模块；同时，处理数据通信模块传来的响应数据报，给用户输出、展现对应的系统处理结果。



d. 算法和流程

该模块不涉及特定算法。

e. 数据设计

前端主要涉及两个数据结构，分别用于存储关系表及运算结果表。

关系表用于存储用户输入的关系名，行数，列数，列名，关系表内容，便于后续发送至后端。同时使用 dis 标识该表的提交状态，避免多次提交同一关系表。

运算结果表中，每行用 id 标识行号，id 依次递加。行中每项值使用 num 和 data 两个变量标识，其中 num 为列名，data 为该项值。

f. 源程序文件说明

该模块主要包含三个部分，分别是关系管理部分、表达式输入部分和结果呈现部分。

1. 关系管理部分

用户可以在此对关系进行增删以及提交重置，定义所需的关系表。

2. 表达式输入部分

用户可在此输入关系表达式，其中关系名使用用户在前一部分自定义的名称，关系运算符使用按钮填入表达式。

3. 结果呈现部分

用户提交关系式成功后在此处获取结果表。

g. 主要函数说明

该模块主要包括关系管理部分中的 addDomain 函数和表达式输入部分中的 handleInputBlur、labClick 函数。

```
// 新增表单项
    addDomain () {
      this.domains.push({
        name: '',
        row: '',
        col: '',
        col_name: '',
        text: '',
        dis: false,
        key: Date.now()// 获取当前时间作为 key，以确保表单的唯一 key 值
      })
    }
//  获取光标位置
    handleInputBlur (e) {
      this.cursorIndex = e.srcElement.selectionStart
    }
// 点击标签按钮，处理文本框文本内容
    labClick (lab) {
      let s1 = ''
      let s2 = ''
      if (this.expression.length < this.cursorIndex) {
        this.expression = this.expression + lab
      } else {
        s1 = this.expression.toString()
```

```
        s2 = this.expression.toString()
        this.expression = s1.substring(0, this.cursorIndex) +
          lab +
          s2.substring(this.cursorIndex, this.expression.length)
      }
      this.$refs.inputArea.focus()
    }
```

3.2.2 数据通信模块

a. 功能描述

负责前后端程序的通信，实现上采用了 http 接口的形式。

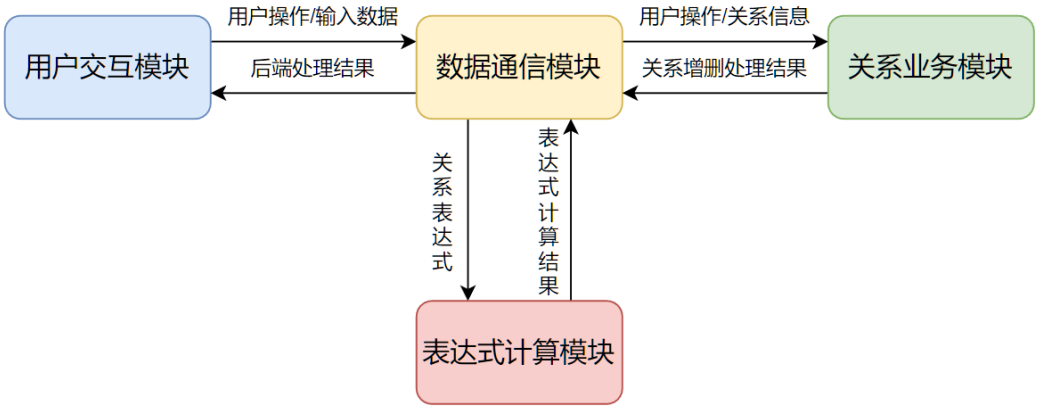
b. 输入数据

对于前端，输入数据为用户定义的关系、用户执行的关系操作、表达式以及计算表达式操作等；对于后端，输入数据为关系业务模块以及表达式计算模块两个模块所返回的处理结果。

本层会对不合规的输入数据做初步检验，包含列长列名长度一致检查、元素非空检查等。

c. 输出数据

用户输入的数据会按预先规定的格式包装成 http 数据报，输出交由后端模块进行处理;后端处理后的数据会交由该模块,进行响应数据报的组装并返还给前端。



d. 算法和流程

该模块不涉及特定算法。

e. 数据设计

数据通信模块主要用到两个数据结构。

1. relation 表示关系的数据结构（表）

包含五个字段：关系名、行长、列长、列名以及表的内容。

表的内容（content）使用一个字符串表示，每个元素间使用英文逗号分隔，应当满足字符串内元素的数量等于行长乘以列长。

举例，以下关系以及对应的 Json 结构：

name	age	gender
Johnny	18	male
Jack	20	male

```
{
"relation_name":"student",
"row_len":3,
"col_len":2,
"col_name":"name,age,gender",
"content":"Johnny,18,male,Jack,20,male"
}
```

2. response 响应

前端调用接口后返回的是一个 response 对象，包含三个字段：状态码、信息、数据体。如果是需要返回结果的请求调用，结果对象会存储在数据体中。

举例，一个请求成功，且返回数据存放在 data 中的 response 对象：

```
{
"code":200,
"msg":"ok",
"data":{
    "relation_name":"student",
    "row_len":3,
```

```
        "col_len":2,
        "col_name":"name,age,gender",
        "content":"Johnny,18,male,Jack,20,male"
       }
    }
```

f. 源程序文件说明

1. 前端部分

该模块主要完成前端向后端提交、删除、重置表单以及提交表达式函数。

```
// InputTable.vue 向后端传表单
    submit (item) {
      this.$axios
        .post('/insert/', {...})
        .then(successResponse => {
          if (successResponse.data.code === 200) {...}
          else if (successResponse.data.code === 400) {...}})
        .catch(failResponse => {...})
    }
// InputTable.vue 删除表单项
    removeDomain (item) {
      ...
      this.$axios
        .post('/delete/', {...})
        .then(successResponse => {
          if (successResponse.data.code === 200) {...}})
        .catch(failResponse => {...})
    }
// InputTable.vue 重置
    reset () {
      this.$axios
        .get('/delete_all/', {})
        .then(successResponse => {
          if (successResponse.data.code === 200) {...}
```

```
          this.reload() // 调用刷新
        })
        .catch(failResponse => {...})
    }
// InputExpression.vue 提交表达式
    submit () {
      ...
      this.$axios
        .post('/compute/', {...})
        .then(successResponse => {
          if (successResponse.data.code === 200) {...}
          else if (successResponse.data.code === 400) {...}
        })
    }
```

2. 后端部分

该模块后端由 WebController 类构成，负责向前端提供各种功能的 http 接口。篇幅原因，以下源程序省略了详细实现，仅保留主要的接口方法。

```
public class WebController {

    /**
     * 计算表达式结果并返回
     * @return
     */
    @RequestMapping(value = "/api/compute/", method =
RequestMethod.POST)
    public Result getCalculationResult(@RequestBody String expression);

    /**
     * 新建关系
     * @return
     */
    @RequestMapping(value = "/api/insert/", method =
```

```
RequestMethod.POST)
    public Result insertRelation(@RequestBody RelationVo vo);

    /**
     * 删除关系
     * @return
     */
    @RequestMapping(value = "/api/delete/", method =
RequestMethod.POST)
    public Result deleteRelation(@RequestBody String name);

    /**
     * 删除所有关系
     * @return
     */
    @RequestMapping(value = "/api/delete_all/", method =
RequestMethod.GET)
    public Result deleteAll();
}
```

g. 主要函数说明

1. 前端部分

关系管理部分中主要提供了 3 个接口方法：

(1) submit：POST 方法，向后端传表单，入参为关系表的表名，行数，列数，列名，及表单具体内容，出参为 http 响应报文。

(2) removeDomain：POST 方法，删除后端表单项，入参为关系表的表名，出参为 http 响应报文。

(3) reset：GET 方法，重置后端所有表单项，入参为空，出差为 http 响应报文。

表达式输入部分中主要提供了 1 个接口方法：

(4) submit：POST 方法，像后端传关系表达式，获取表达式，并将关系符转换为相应的#形式，入参为表达式，出参为 json 形式存储在数据体的 http 响应报文。

2. 后端部分

WebController 主要提供了 4 个接口方法。

(1) getCalculationResult：POST 方法，获取计算结果，入参为字符串形式的计算表达式，出参为结果以 json 形式存储在数据体的 http 响应报文。

(2) insertRelation：POST 方法，新增关系，入参为 json 格式的关系数据结构，出参为 http 响应报文，根据插入的结果是否成功来决定其状态码以及附注信息。

(3) deteleRelation：POST 方法，删除关系，入参为字符串格式的关系名，出参同上，根据删除结果是否成功来回复成功/失败的响应报文。

(4) deteleAll：GET 方法，删除所有关系（重置），不需要入参，调用该方法时删除所有已有的关系，根据删除结果是否成功来回复成功/失败的响应报文。

3.2.3 关系业务模块

a. 功能描述

处理关系的增/删/取，与持久化模块直接交互，涉及到关系的一系列操作都需要经过此模块。

b. 输入数据

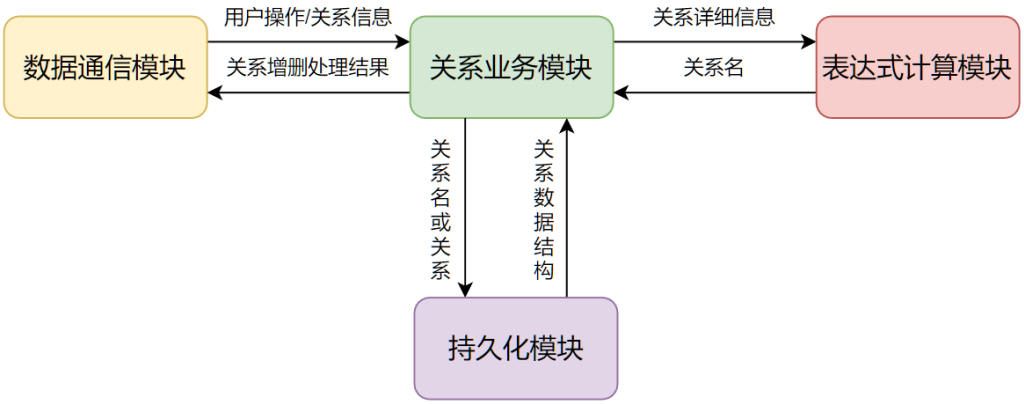
可以分为三个输入源，一个是数据通信模块，输入为用户执行的操作以及关系信息，根据用户的操作，该模块执行对应的功能；另一个是关系计算

模块，输入为关系名，根据关系名获取详细的关系信息；最后是持久化模块，根据关系名获取关系详细数据。

这一层会对关系的数据合法性进行全面校验，包括关系的内容是否符合关系的行列长等。

c. 输出数据

针对三个输入源，分别有对应的输出。对于数据通信模块，输出为关系增删是否成功的结果；针对关系计算模块，输出为名称对应的关系数据结构；持久化模块则在用户新增/删除或是计算模块获取关系详细数据时由业务模块代为处理，向持久化模块输出对应参数进行处理。



d. 算法和流程

该模块不涉及特定算法。

e. 数据设计

存储关系主要涉及两个类，RelationVo 以及 RelationBo，前者主要用于与前端交互时存储从前端送来的关系/准备送至前端的关系信息，后者则主要用于后端业务处理，包括关系的存储/表达式的运算等。

鉴于二者相似性，仅详细介绍 RelationBo 的成员，伪代码如下。

```
RelationBo {
    // 关系行长
    int rowLen;
    // 关系列长
    int colLen;
    // 关系列名
    String[] colName;
    // 关系内容，通过二维数组存储
    String[][] content;
    // bo 与 vo 相互转换的方法
    toRelationVo(RelationBo bo, String name);
    toRelationBo(RelationVo vo, String name);
}
```

f. 源程序文件说明

RelationService 接口为关系业务模块的主要体现，以下展示其接口源码，展示了其有哪些对外方法。具体的实现类请查看源码中的 RelationServiceImpl。

```
public interface RelationService {

    /**
     * @param relationVo 前端传来的Relations
     * @throws ParamLenException 参数不对时抛出此异常
     */
    void insertRelation(RelationVo relationVo) throws
ParamLenException;

    /**
     * 删除关系
```

```
     */
    void deleteRelation(String name);

    /**
     * 删除所有已建立关系
     */
    void deleteAll();

    /**
     * 是否存在名为 key 的关系
     * @param key
     * @return
     */
    boolean contains(String key);

    /**
     * 获取 name 对应的 Bo
     * @param name
     * @return
     */
    RelationBo get(String name);
}
```

g. 主要函数说明

RelationService 提供了下列对外的方法。

(1) insertRelation：新增关系，入参为 vo 形式的关系数据，无出参，添加成功时不返回参数，添加失败时会抛出参数错误异常。

(2) deleteRelation：删除特定关系，入参为字符串形式的关系名，无出参。

(3) deleteAll：删除所有关系，重置，入参与出参皆为空。

(4) contains：是否包含对应名字的关系，入参为字符串形式的关系名，出参为布尔值。

(5) get：获取对应名称的关系信息，入参为字符串形式的关系名，出参为对应关系的数据结构，未找到时为 null。

3.2.4 持久化模块

a. 功能描述

负责存储用户定义的关系，以及通过关系业务模块向计算模块提供关系信息。

b. 输入数据

只与关系业务模块直接交互，输入为关系信息/用户操作等。

c. 输出数据

只与关系业务模块直接交互，输出为关系的详细信息。

d. 算法和流程

不涉及特定算法，关系存储使用 java.util.Map.CurrentHashMap 类。

e. 数据设计

参见关系业务模块。为 RelationBo。

f. 源程序文件说明

参见关系业务模块。

g. 主要函数说明

参见业务关系模块。

3.2.5 代数运算模块

a. 功能描述

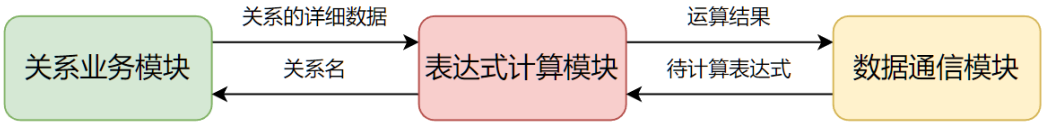
通过关系业务模块以及各种工具类的辅助，完成数据通信模块传来的表达式的计算求值，并向数据通信模块返回求得的结果。同时，该模块也具备判断表达式合法性判断的能力。

b. 输入数据

由数据通信模块输入待计算的表达式，由关系业务模块输入需要用到的关系数据，共同完成表达式的合法性判断以及最后的结果计算。

c. 输出数据

计算结果完成后，结果也是一个关系，将该关系输出传递给数据传输模块进行包装返回前端。



d. 算法和流程

1. 表达式合法性分析

在计算表达式之前，需要对表达式的合法性进行判断。程序中采用了对原始输入表达式进行词法分析、语法分析的方式来判断表达式的合法性。对此，需要先定义文法。

```
<表达式> -> <表达式>∩B | B
B -> B∪C | C
C -> C×D | C÷D | D
D -> D−E | E
E -> E⋈F | F
F -> π[G][<表达式>] | σ[I][<表达式>] | (<表达式>) | <标识符>
G -> <标识符>H
H -> ,<标识符>H | ε
I -> I∧J | J
J -> J∨K | K
K -> (I) | <标识符>L
L -> =M | >M | ≥M | <M | ≤M | ≠M
M -> <标识符> | <数字>
```

（1）词法分析

使用正规式定义标识符记号 id，数字记号 num，运算符记号 operator:

$$id = char(char|digit)^*$$

$$num = digitsoptional\_fraction$$

$$operator = \cap |\cup| - |\times| \bowtie |\div|\pi|\sigma| \wedge |\vee|(|)|[|]|.|\leq| \geq |\neq| > |<| = |,$$

其中辅助定义式：

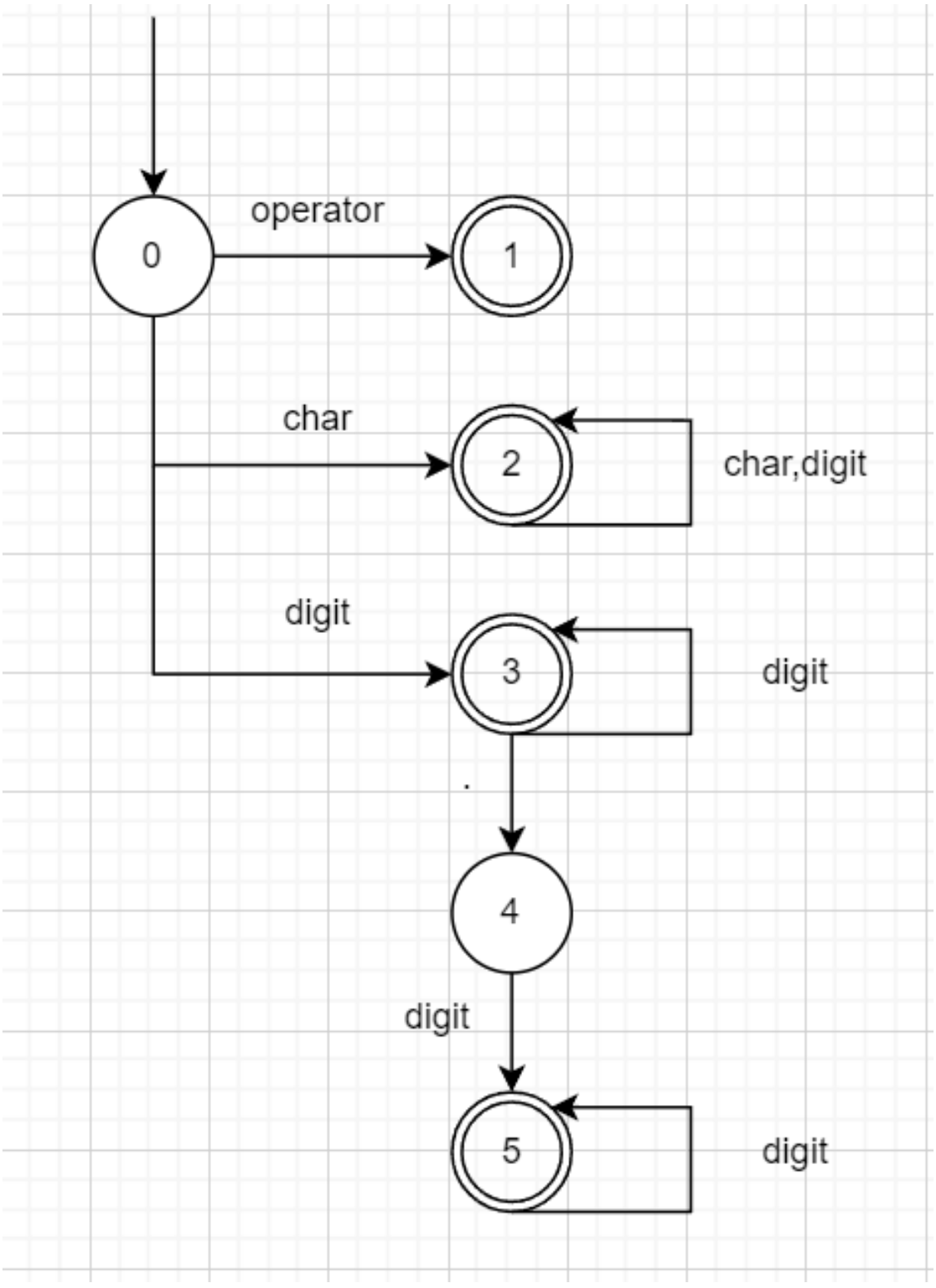
$$char = [a - zA - Z]$$

$$digit = [0 - 9]$$

$$digits = digit^+$$

$$optional_fraction = (.digits)?$$

通过正规式构建 DFA 如下图所示。


operator
0
1
char
2
char,digit
digit
3
digit
.
4
digit
5
digit

DFA 即 deterministic finite automaton 有限状态自动机，其特点是可以实现状态的自动转移，可以用于解决字符匹配问题。使用 DFA 实现词法分析器的步骤如下。

1. 先定义不同的状态 ：如操作符状态、数字状态、标识符状态等。
2. 再定义状态转移条件 ：如当前状态是初始状态，遇到数字则转移至数字状态，遇到符号则转移至符号状态。
3. 如果字符读取完成，则整个转移过程结束。

基于以上步骤，编写程序，即可通过 DFA 来完成对字符串的词法分析过程。

（2）语法分析

语法分析（英语：syntactic analysis，也叫 parsing）是根据某种给定的形式文法对由单词序列（如英语单词序列）构成的输入文本进行分析并确定其语法结构的一种过程。语法分析器通常使用一个独立的词法分析器从输入字符流中分离出一个个的“单词”，并将单词流作为其输入。本次开发中，语法分析器采用手工编写。

进行自上而下的语法分析，需要构建预测分析表，对于关系代数表达式的分析表推导过程如下。

用产生式表示文法，其中 A：表达式，id：标识符，num：数字

```
A -> A∩B | B
B -> B∪C | C
C -> C×D | C÷D | D
D -> D−E | E
E -> E⋈F | F
F -> π[G][A] | σ[I][A] | (A) | id
G -> idH
H -> ,idH | ε
I -> I∧J | J
J -> J∨K | K
K -> (I) | idL
L -> =M | >M | ≥M | <M | ≤M | ≠M
M -> id | num
```

消除左递归和公共左因子

```
A -> CB
B -> ∩CB | ε
C -> ED
D -> ∪ED | ε
E -> GF
F -> ×GF | ÷GF | ε
G -> IH
H -> −IH | ε
I -> KJ
J -> ⋈KJ | ε
K -> π[L][A] | σ[N][A] | (A) | id
L -> idM
M -> ,idM | ε
N -> PO
O -> ∧PO | ε
P -> RQ
Q -> ∨RQ | ε
R -> (N) | idS
S -> =T | >T | ≥T | <T | ≤T | ≠T
T -> id | num
```

求解 First 集和 Follow 集

First(A)={π,σ,(,id}　　Follow(A)={],),#}
First(B)={∩,Ɛ}　　Follow(B)={],),#}
First(C)={π,σ,(,id}　　Follow(C)={∩,],),#}
First(D)={∪,Ɛ}　　Follow(D)={∩,],),#}
First(E)={π,σ,(,id}　　Follow(E)={∪,∩,],),#}
First(F)={×,÷,Ɛ}　　Follow(F)={∪,∩,],),#}
First(G)={π,σ,(,id}　　Follow(G)={×,÷,∪,∩,],),#}
First(H)={-,Ɛ}　　Follow(H)={×,÷,∪,∩,],),#}
First(I)={π,σ,(,id}　　Follow(I)={-,×,÷,∪,∩,],),#}
First(J)={⋈,Ɛ}　　Follow(J)={-,×,÷,∪,∩,],),#}
First(K)={π,σ,(,id}　　Follow(K)={⋈,-,×,÷,∪,∩,],),#}
First(L)={id}　　Follow(L)={]}
First(M)={,,Ɛ}　　Follow(M)={]}
First(N)={(,id}　　Follow(N)={]}
First(O)={∧,Ɛ}　　Follow(O)={]}
First(P)={(,id}　　Follow(P)={∧,]}
First(Q)={∨,Ɛ}　　Follow(Q)={∧,]}
First(R)={(,id}　　Follow(R)={∨,∧,]}
First(S)={=,>,≥,<,≤,≠}　　Follow(S)={∨,∧,]}
First(T)={id,num}　　Follow(T)={∨,∧,]}

得到最终的预测分析表如下。

	id	num	∪	∩	-	×	⋈	+	σ	π	(	)	[	]	=	>	≥	<	≤	≠	∧	∨	,	#
A	CB	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	CB	CB	CB	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
B	ERROR	ERROR	ERROR	∩CB	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'
C	ED	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ED	ED	ED	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
D	ERROR	ERROR	∪ED	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'
E	GF	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	GF	GF	GF	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	'	'	ERROR	×GF	ERROR	÷GF	ERROR	ERROR	ERROR	'	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'
G	IH	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	IH	IH	IH	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
H	ERROR	ERROR	'	'	-IH	'	ERROR	'	ERROR	ERROR	ERROR	'	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'
I	KJ	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	KJ	KJ	KJ	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
J	ERROR	ERROR	'	'	'	'	⋈KJ	'	ERROR	ERROR	ERROR	'	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'
K	id	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	σ[N][A]	π[L][A]	(A)	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
L	idM	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
M	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	,LM	ERROR
N	PO	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	PO	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
O	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	∧PO	ERROR	ERROR	ERROR
P	RQ	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	RQ	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
Q	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	'	∨RQ	ERROR	ERROR
R	idS	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	(N)	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
S	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	=T	>T	≥T	<T	≤T	≠T	ERROR	ERROR	ERROR	ERROR
T	id	num	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

通过预测分析表以及辅助栈的帮助，可以使程序按如下步骤完成语法分析的过程。先入栈终止符，再入栈起始符。循环执行以下步骤之一。

- 栈顶符号 X = 输入符号 a = 终止符，则分析成功，停止分析。
- 栈顶符号 X = 输入符号 a 不等于 终止符，将 X 从栈顶出栈，a 指向下一个输入符号。
- 栈顶符号 X 是非终结符，查分析表 table。
    - 若 table[X, a]中存放着关于 X 的一个产生式，将 X 出栈，将产生式右部符号串按反序一一入栈。若符号为空，则不如栈。
    - 若 table[X, a]中存放着出错标志 ERROR，则判断表达式不合法。

通过以上步骤即可对输入字符串完成语法分析。

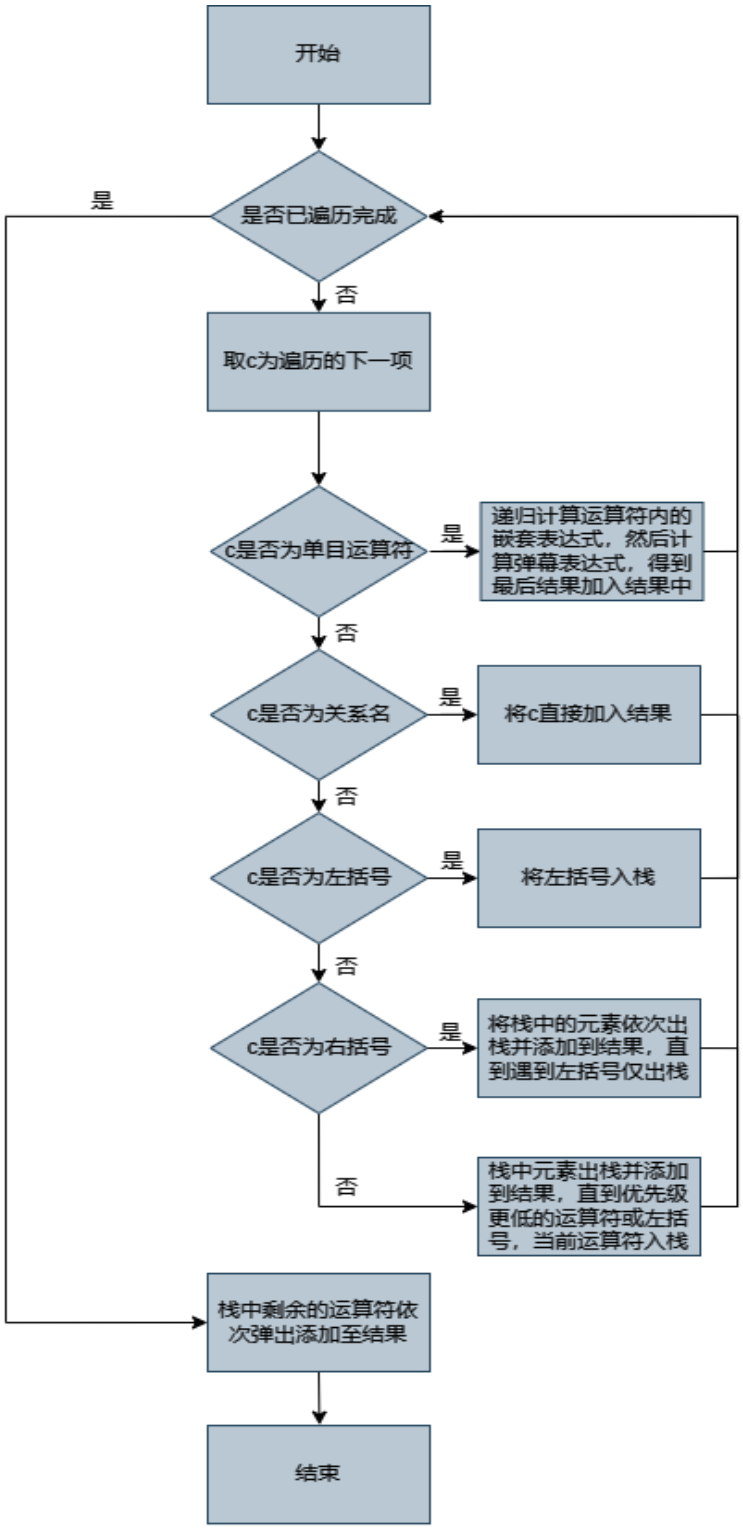
综合语法分析以及词法分析的结果，可以判断表达式是否合法。

2. 表达式解析与运算

通过前置的合法性分析以及前后端约定的传输格式，现在已经可以得到一个格式合法的、运算符与标识符等元素被分隔的以字符串数组形式表示的关系代数表达式。接下来针对该表达式，进行如下的运算逻辑。

(1)关系表达式转逆波兰表达式

为将关系表达式转为易于计算机处理的类型，需要将原来的中缀类型的表达式转为逆波兰表达式。而这种转换需要通过使用栈来辅助完成。以下简述该算法步骤，res 来表示转换后的结果数组，算法的流程图如下：


开始
是否已遍历完成
是
否
取c为遍历的下一项
c是否为单目运算符
是
递归计算运算符内的嵌套表达式，然后计算弹幕表达式，得到最后结果加入结果中
否
c是否为关系名
是
将c直接加入结果
否
c是否为左括号
是
将左括号入栈
否
c是否为右括号
是
将栈中的元素依次出栈并添加到结果，直到遇到左括号仅出栈
否
栈中元素出栈并添加到结果，直到优先级更低的运算符或左括号，当前运算符入栈
栈中剩余的运算符依次弹出添加至结果
结束

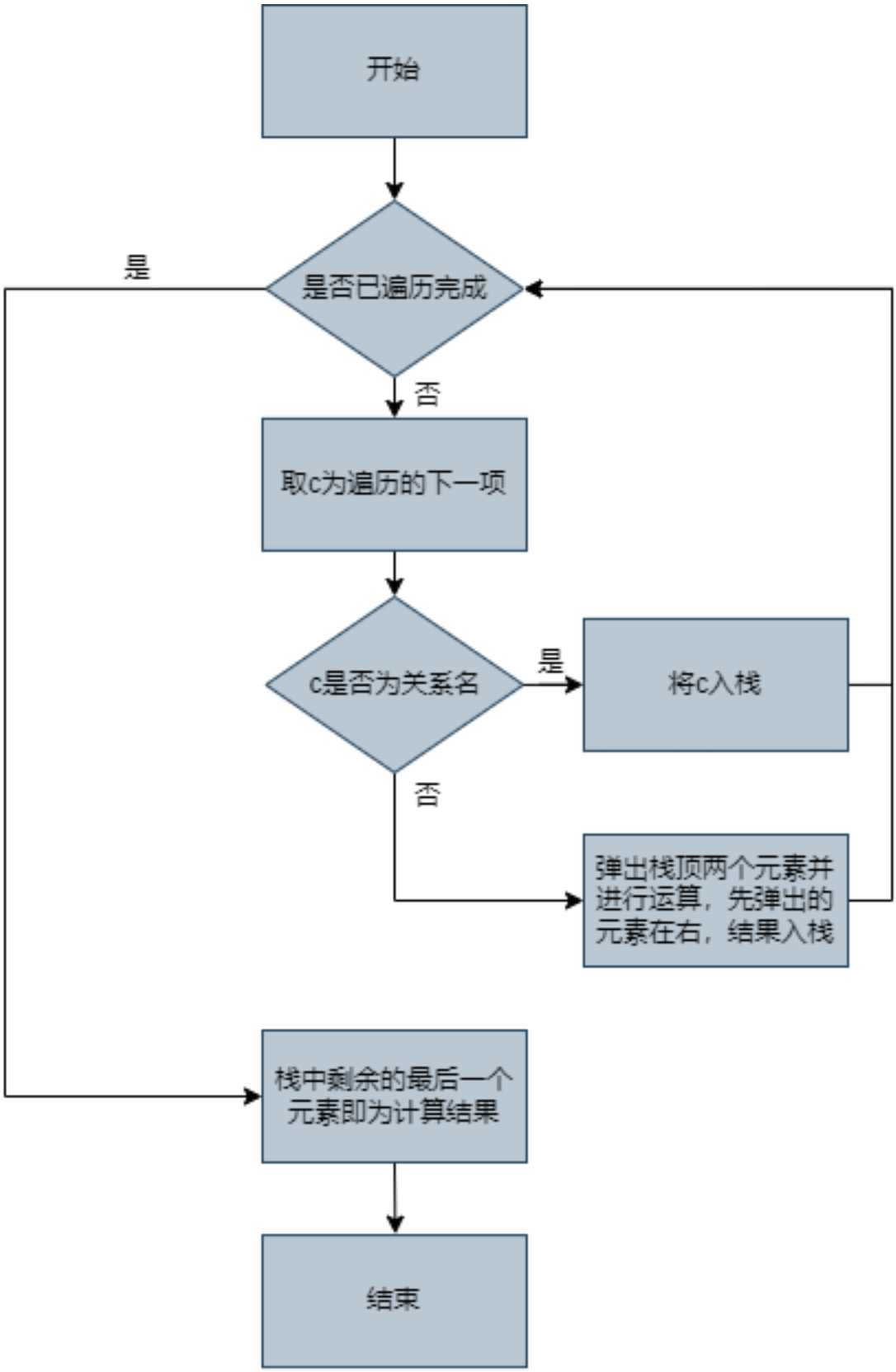
因为算法在后续计算过程中对中缀表达式的处理较为复杂，因此，对于选择与投影两个单目运算符，在转逆波兰表达式时就进行计算，得到最后结果即一张关系表后加入结果集合中。

文字表述如下：

- 由左到右开始，逐项扫描关系表达式数组 list。
- 遇到单目运算符，将单目运算符内的表达式进行递归计算处理，得到结果后计算单目运算符，将最后的结果加入 res。
- 遇到关系名，直接添加到结果数组 res。
- 遇到关系运算符或是括号，根据以下几种情况来处理。
    1. 若为左括号(，则将左括号入栈；
    2. 若为右括号)，则将栈中的元素依次出栈并添加到 res，直到遇到(，(仅出栈且不添加到 res。
    3. 若为其他关系运算符，将栈中的元素依次出栈并添加到 res，直到遇到比当前运算符优先级更低的运算符或者(，然后将当前运算符入栈。
- 扫描完成后，将栈中剩余的运算符依次弹出，添加至 res。

(2)计算逆波兰表达式

将关系代数表达式转化为逆波兰表达式后，可以继续进行结果的计算了。同样，需要用到栈这一数据结构的辅助。


开始
是否已遍历完成
是
否
取c为遍历的下一项
c是否为关系名
是
将c入栈
否
弹出栈顶两个元素并进行运算，先弹出的元素在右，结果入栈
栈中剩余的最后一个元素即为计算结果
结束

文字表述如下：

- 由左到右开始，逐项扫描逆波兰表达式数组 res。
- 遇到关系名则压栈。
- 遇到关系运算符，则弹出栈顶的两个元素，先弹出的在右边，后弹出的在左边，进行计算，将结果关系压栈。
- 扫描完成后，弹出栈顶元素，即为最终结果。

以上步骤即为计算关系表达式的步骤。

1. 各算符运算处理方式

(1)交

得到两表相同的元组。具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，果存在表 1 内的元组表 2 内没有的情况，说明两表结构不同，抛出异常；如果不存在，则找到两表列名的对应关系（相同列一一对应）。再用一个嵌套 for 循环对两表元组进行两两比较，相同的加入结果表。

(2)并

合并两表元组（去重）。具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，如果存在表 1 内的元组表 2 内没有的情况，说明两表结构不同，抛出异常；如果不存在，则找到两表列名的对应关系（相同列一一对应）。用一个 for 循环将表 1 的元组全部加入结果表，再用一个嵌套 for 循环对两表元组进行两两比较，将表 2 中与表 1 不重复的元组加入结果表。

(3)差

将表 1 中与表 2 不重复的元组加入结果表。具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，如果存在表 1 内的元组表 2 内没有的情况，说明两表结构不同，抛出异常；如果不存在，则找到两表列名的对应关系（相同列一一对应）。用一个嵌套 for 循环对两表元组进行两两比较，将表 1 中与表 2 不重复的元组加入结果表。

(4)除

笛卡尔积的逆运算，得到满足下列条件的最大的表：其中每行与表 2 中的每行组合成的新行都在表 1 中。

具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，用一个数组记录两表的相同列，形如 temp[i]=j 表示表 1 中的第 i 列与表 2 中的第 j 列相同。如果无相同列，则直接抛出异常，返回空表；如果有，则先求表 2 对相同列的投影，然后使用一个嵌套 for 循环对两表元组进行两两比较，如果表 1 中某一行的相同列与表 2 投影的某行相同，则将该行去除相同列加入结果表中。

(5)笛卡尔积

将表 1 的每一行与表 2 的每一行分别组合在一起，作为结果表的一行。

具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，用一个数组记录两表的相同列，形如 temp[i]=j 表示表 1 中的第 i 列与表 2 中的第 j 列相同。计算结果表的新列名，即非相同列保留原来的列名，相同列以形如"Relation"+count+"."+列名命名，count 为一个全局变量，意为当前表为参与运算的第 count 个关系表。使用一个嵌套 for 循环将两表元组进行两两组合加入结果表内。结果表列数=表 1 列数+表 2 列数，结果表行数=表 1 行数*表 2 行数。

(6)自然连接

将表 1 中与表 2 中相同列数值相同的行组合在一起形成结果表的新行(去重复列)。

具体的程序实现上，首先使用一个嵌套 for 循环对两表的列名进行两两对比，用一个数组记录两表的相同列，形如 temp[i]=j 表示表 1 中的第 i 列与表 2 中的第 j 列相同。如果无相同列，则直接返回两表的笛卡尔积；如果有相同列,使用一个嵌套 for 循环将相同列数值完全相同的两表中的两行按”表 1 非相同列+相同列+表 2 非相同列“的顺序组合成一个新元组加入结果表内。

(7)投影

筛选特定列。

具体的程序实现上，首先判断输入的列名是否合法，不合法则抛出异常。然后使用一个 for 循环，将每行的指定列元素重新组成为新元组加入结果表内。

(8)选择

筛选符合条件的行。具体的程序实现上，选择运算符中包含一个参数为“条件式”，条件式的结构为[列名][运算符][值]，如 Sno >= 95001，同时，支持多个条件式之间通过与或来连接。因此，选择的运算符运算需要涉及到一个条件式解析的过程，条件式解析的算法参考了四则运算中的后缀表达式处理法，我们将每一个最小的条件单元（只包含一个列名，一个运算符以及一个值的条件式）作为最小分隔单元，涉及到多个条件式通过与或连接组成的复合条件式运算时，将其转为后缀表达式后再处理，每轮循环中只需要处理单个条件单元，最后将得到的结果同与或算符相作运算，即可得到关系中某一行对该表达式的真伪。

e. 数据设计

计算过程中用到的数据结构为 RelationBo，具体见关系业务模块；同时计算过程中使用到栈这一辅助数据结构，在表达式合法性判断过程中用到了预测分析表以及符号集等这类数据结构。

f. 源程序文件说明

1. 接口 JudgementOfLegalityService 的实现类 JudgementOfLegalityService 为表达式合法性的判断类，包含词法分析以及语法分析两个算法。

2. 接口 ComputingService 的实现类 ComputingServiceImpl 为该表达式解析计算主要的类，包含上述两个算法。

3. 工具类 ComputingUtil 包含了 8 个关系运算的具体实现。

词法分析

由接口 LexicalAnalysisService 定义，由类 LexicalAnalysisServiceImpl 实现。

语法分析

由接口 ParsingService 定义，由类 ParsingServiceImpl 实现。

```
public boolean parsing(List<String> expression, String startSign) {
    Deque<String> stack = new ArrayDeque<>();
    stack.push(Constant.END_SIGN);
    stack.push(startSign);
    boolean flag = true;
    int i = 0;
    while (flag) {
        String top = stack.poll();
        String cur = expression.get(i);
        if (Constant.TERMINATOR.contains(top)) {
            if (cur.equals(top)) {
                i++;
            } else {
```

```
                return false;
            }
        } else if ("#".equals(top)) {
            if (top.equals(cur)) {
                flag = false;
            } else {
                return false;
            }
        } else if (Constant.NON_TERMINATOR.contains(top)) {
            boolean swapRes = swap(top, cur, stack);
            if (!swapRes) {
                return false;
            }
        } else {
            return false;
        }
    }
    return true;
}
```

计算逆波兰表达式

主要的计算方法如下。

```
/**
 * 计算后缀表达式
 * @param expression
 * @return
 */
private RelationBo calculate(String expression) throws
ComputingException, WrongColumnNameException,
IllegalOperationException, RelationNotExistsException {
    List<String> post = null;
    try {
```

```
        post = parse(expression);
    } catch (ParamLenException e) {
        log.error("临时关系生成错误", e);
        throw new ComputingException();
    }
    Deque<RelationBo> stack = new ArrayDeque<>();
    for (String elem : post) {
        if (relationService.contains(elem)) {
            stack.push(relationService.get(elem));
            continue;
        }
        if (elem.charAt(0) != '#') {
            throw new RelationNotExistsException();
        }
        RelationBo bo1 = stack.poll(), bo2 = stack.poll();
        if (bo1 == null || bo2 == null) {
            throw new ComputingException();
        }
        RelationBo res = null;
        switch (elem) {
            case Constant.JOIN:
                res = ComputingUtil.join(bo1, bo2);
                break;
            case Constant.DIV:
                res = ComputingUtil.div(bo2, bo1);
                break;
            case Constant.AND:
                res = ComputingUtil.and(bo1, bo2);
                break;
            case Constant.OR:
                res = ComputingUtil.or(bo1, bo2);
                break;
            case Constant.PROD:
```

```
                res = ComputingUtil.prod(bo2, bo1, relationCount);
                relationCount += 2;
                break;
            case Constant.DIFF:
                res = ComputingUtil.diff(bo2, bo1);
                break;
            default:
                throw new ComputingException();
        }
        if (res == null) {
            res = RelationBo.EMPTY_RELATION;
        }
        stack.push(res);
    }
    return stack.peek();
}
```

中缀转逆波兰表达式

```
/**
 * 中缀转后缀，并处理单目运算符
 * @param expression
 * @return
 */
private List<String> parse(String expression) throws
ComputingException, ParamLenException, WrongColumnNameException,
IllegalOperationException, RelationNotExistsException {
    String[] elems = splitExpression(expression);
    List<String> res = new ArrayList<>();
    Deque<String> stack = new ArrayDeque<>();
    // 临时表计数
    int tempCount = 0;
    for (String elem : elems) {
        // elem 为已定义的关系
```

```
            if (relationService.contains(elem)) {
                res.add(elem);
                continue;
            }
            // 处理单目运算符
            if (elem.charAt(0) == '#' && elem.length() > 6) {
                tempCount = preprocessingUnaryOperator(res, tempCount,
elem);
                continue;
            }
            // elem 为括号或是栈空且 elem 为多目运算符时，直接入栈
            if (elem.equals("(") || (elem.charAt(0) == '#') &&
stack.isEmpty()) {
                stack.push(elem);
                continue;
            }
            if (elem.equals(")")) {
                while (!stack.isEmpty() && !stack.peek().equals("(")) {
                    res.add(stack.poll());
                }
                stack.poll();
                continue;
            }
            while (!stack.isEmpty()
                    && !stack.peek().equals("(")
                    && Constant.PRIORITY.get(elem) >=
Constant.PRIORITY.get(stack.peek())) {
                res.add(stack.poll());
            }
            stack.push(elem);
        }
        while (!stack.isEmpty()) {
            res.add(stack.poll());
```

```
        }
        return res;
    }
```

处理单目运算符

```
    private int preprocessingUnaryOperator(List<String> res, int
tempCount, String elem) throws ComputingException, ParamLenException,
WrongColumnNameException, IllegalOperationException,
RelationNotExistsException {

        // 第0个参数为关系（或是嵌套表达式），其余参数为运算的参数
        String[] params = getParam(elem);
        if (params.length <= 1) {
            log.error("getParam 出错");
            throw new ComputingException();
        }
        RelationBo tmp = null;
        // params[0]本身已经是已定义的关系则直接取用，否则递归计算表达式
        if (relationService.contains(params[0])) {
            tmp = relationService.get(params[0]);
        } else {
            tmp = calculate(params[0]);
        }
        String tempRelationName = Constant.TEMP_RELATION_PREFIX +
tempCount;
        if (Constant.SELECT.equals(elem.substring(0,7))) {     // 选择运
算处理
            RelationBo selectResult = ComputingUtil.select(tmp,
params[1]);
            relationService.insertRelation(selectResult,
tempRelationName);
            res.add(tempRelationName);
```

```
            tempCount++;
        }
        if (Constant.PROJECT.equals(elem.substring(0,8))) {    // 投影运
算处理
            String[] projectColName = new String[params.length - 1];
            System.arraycopy(params, 1, projectColName, 0, params.length
- 1);
            int[] projectColNo = new int[projectColName.length];
            for (int i = 0; i < projectColName.length; i++) {
                projectColNo[i] =
tmp.getColIndexByName(projectColName[i]);
            }
            RelationBo projectResult = ComputingUtil.project(tmp,
projectColNo);
            relationService.insertRelation(projectResult,
tempRelationName);
            res.add(tempRelationName);
            tempCount++;
        }
        return tempCount;
    }
```

运算符运算

运算工具类 ComputingUtil 则负责各种运算符运算的具体实现，定义的接口如下，实现因篇幅原因，仅展示几个重要的运算符实现，其余的请见代码附件。

除法

```
public static RelationBo div(RelationBo r1, RelationBo r2) {
        RelationBo r3 = null;
        //除数为无列空表,返回 r1 的有列空表
        if(r2.getColLen()==0){
            try {
```

```
                r3 = new RelationBo(0,r1.getColLen(),r1.getColName(),
"");
            }catch (ParamLenException e){
                System.out.print("参数长度错误");
            }
            return r3;
        }
        //1 先求相同列
        String temp1 = "";        //相同列在 r1 中的索引
        String temp2 = "";        //相同列在 r2 中的索引
        String temp3 = "";        //不相同列的名字
        for(int i=0;i<r1.getColLen();i++){
            Boolean isColSame = false;
            for(int j=0;j<r2.getColLen();j++){

if(Objects.equals(r1.getColName()[i],r2.getColName()[j])){
                    temp1+=i;
                    temp1+=",";
                    temp2+=j;
                    temp2+=",";
                    isColSame = true;
                    break;
                }
            }
            if(!isColSame){
                temp3+=r1.getColName()[i];
                temp3+=",";
            }
        }
        //处理无相同列的情况
        if(temp1==""&&temp2==""){
            return RelationBo.EMPTY_RELATION;
```

```
        }
        String[] r1ColName = temp1.split(",");
        String[] r2ColName = temp2.split(",");
        String[] newColName = temp3.split(",");
        //2 求 r2 对相同列的投影
        int[] r1Temp = new int[r1ColName.length];
        int[] r2Temp = new int[r2ColName.length];
        for(int i=0;i<r1Temp.length;i++){
            r1Temp[i] = Integer.parseInt(r1ColName[i]);
            r2Temp[i] = Integer.parseInt(r2ColName[i]);
        }
        try{
            RelationBo r2New = project(r2,r2Temp);
            //3 如果 r1 中某一行的相同列与 r2 投影的某行相同，则将该行去除相同
列加入字符串
            String str = "";
            int rowLen = 0;
            //对于 r1 的每一行的特定列元素都需要与 r2New 的每一行对比
            for(int i=0;i<r1.getRowLen();i++){
                if(isHasSpecial(r1,r2New,i,r1Temp)){
                    str = deleteSpecialAdd(r1,i,r1Temp,str);
                    rowLen++;
                }
            }
            //4 赋给新表
            try {
                r3 = new RelationBo(rowLen, r1.getColLen()-
r1Temp.length, newColName, str);
                int a = r3.getColLen();
                int b = r3.getRowLen();
                String[] c = r3.getColName();
            }catch (ParamLenException e){
```

```
                System.out.print("参数长度错误");
            }
        }catch (WrongColumnNameException e) {
            e.printStackTrace();
        }
        return r3;
    }
//辅助方法：检查表 r2New 中是否含有表 r1 的第 x 行的特定列
    public static Boolean isHasSpecial(RelationBo r1,RelationBo
r2New,int x,int[] r1Temp){
        //依次检查 r2New 的每一行是否含有 r1 指定行的特定列
        Boolean isHas = false;
        for(int i=0;i<r2New.getRowLen();i++){
            isHas = true;
            for(int j=0;j<r1Temp.length;j++){

if(!Objects.equals(r2New.getContent()[i][j],r1.getContent()[x][r1Temp[j
]])){
                    isHas = false;
                    break;
                }
            }
            if(isHas) return true;
        }
        return false;
}
    //辅助方法：对于表 r 的第 x 行，去除特定列 int[]，加入字符串 str
private static String deleteSpecialAdd(RelationBo r,int x,int[]
colName,String str){
        //检查列 i 是否为特定列
        for(int i=0;i<r.getColLen();i++){
            Boolean isSpecial = false;
            for(int j=0;j<colName.length;j++){
```

```
                if(i==colName[j]){
                    isSpecial = true;
                    break;
                }
            }
            if(!isSpecial){
                str+=r.getContent()[x][i];
                str+=",";
            }
        }
        return str;
    }
```

连接

```
public static RelationBo join(RelationBo r1, RelationBo r2) {
    //1 求相同列
    String temp1 = "";        //相同列在r1中的索引
    String temp2 = "";        //相同列在r2中的索引
    for(int i=0;i<r1.getColLen();i++){
        for(int j=0;j<r2.getColLen();j++){
            if(Objects.equals(r1.getColName()[i],r2.getColName()[j])){
                temp1+=i;
                temp1+=",";
                temp2+=j;
                temp2+=",";
                break;
            }
        }
    }
    //处理无相同列的情况
    if(temp1==""&&temp2=="")  return prod(r1,r2,0);
    String[] temp1Col = temp1.split(",");
    String[] temp2Col = temp2.split(",");
```

```
    int[] r1Col = new int[temp1Col.length];
    int[] r2Col = new int[temp2Col.length];
    for(int i=0;i<temp1Col.length;i++){
        r1Col[i] = Integer.parseInt(temp1Col[i]);
        r2Col[i] = Integer.parseInt(temp2Col[i]);
    }
    //2 将特定列完全相同的两表中的两行按 A 表非特定列+特定列+B 表非特定列 的顺序加入字符串
    String str = "";
    int rowLen = 0;
    for(int i=0;i<r1.getRowLen();i++){
        for(int j=0;j<r2.getRowLen();j++){
            if(isSpecialSame(r1,r2,i,j,r1Col,r2Col)){
                str = deleteSpecialAdd(r1,i,r1Col,str);
                str = AddSpecial(r1,i,r1Col,str);
                str = deleteSpecialAdd(r2,j,r2Col,str);
                rowLen++;
            }
        }
     }
    //7 计算新列名
    int colLen = r1.getColLen()+r2.getColLen()-r1Col.length;
    String[] colName = new String[colLen];
    int n = 0;
    //先赋表 r1 中非特殊的部分
    for(int i=0;i<r1.getColLen();i++){
        Boolean isSpecial = false;
        for(int j=0;j<r1Col.length;j++){
            if(i==r1Col[j]){
                isSpecial = true;
                break;
            }
         }
```

```
            if(!isSpecial) colName[n++] = r1.getColName()[i];
        }
        for(int i=0;i<r1Col.length;i++) colName[n++] =
r1.getColName()[r1Col[i]];
        for(int i=0;i<r2.getColLen();i++){
            Boolean isSpecial = false;
            for(int j=0;j<r2Col.length;j++){
                if(i==r2Col[j]){
                    isSpecial = true;
                    break;
                }
            }
            if(!isSpecial) colName[n++] = r2.getColName()[i];
        //3 赋给新表
        RelationBo r3 = null;
        try {
            r3 = new RelationBo(rowLen,colLen,colName,str);
        }catch (ParamLenException e){
            System.out.print("参数长度错误");
        }
        return r3;
    }
    //辅助方法：对于表 r 的第 x 行，去除特定列 int[]，加入字符串 str
    private static String deleteSpecialAdd(RelationBo r,int x,int[]
colName,String str){
        //检查列 i 是否为特定列
        for(int i=0;i<r.getColLen();i++){
            Boolean isSpecial = false;
            for(int j=0;j<colName.length;j++){
                if(i==colName[j]){
                    isSpecial = true;
                    break;
                }
```

```
        }
        if(!isSpecial){
            str+=r.getContent()[x][i];
            str+=",";
        }
    }
    return str;
}
//辅助方法：对于表 r 的第 x 行，将其特定列 int[]，加入字符串 str
private static String AddSpecial(RelationBo r,int x,int[]
colName,String str){
    for(int i=0;i<colName.length;i++){
        str+=r.getContent()[x][colName[i]];
        str+=",";
    }
    return str;
}
```

选择

```
public static RelationBo select(RelationBo r, String conditions) throws
WrongColumnNameException {
    // 条件为空，返回原表
    if (conditions.equals("")) return r;
    // 空表直接返回原表
    if (r.getRowLen() == 0) {
        if(r.getColLen()==0){
            if(!Objects.equals(conditions,"")) throw new
WrongColumnNameException();
            else return r;
        }
        return r;
    }
    List<String> post = parsePostExpression(conditions);
```

```
    String str = "";
    int rowLen = 0;
    for(int i=0;i<r.getRowLen();i++){
        if(judgeMultipleCondition(post,r,i)) {
            str = addStr(r, i, str);
            rowLen++;
        }
    }
    RelationBo r3 = null;
    try {
        r3 = new RelationBo(rowLen,r.getColLen(),r.getColName(),str);
    }catch (ParamLenException e){
        System.out.print("参数长度错误");
    }
    return r3;
}
/**
    * 辅助方法7：条件表达式中缀转后缀
    * @param expression 中缀表达式
    * @return 后缀表达式
    */
   private static List<String> parsePostExpression(String expression)
{
       String[] elems = expression.split("\\|");
       List<String> res = new ArrayList<>();
       Deque<String> stack = new ArrayDeque<>();
       for (String elem : elems) {
           // elem 为非运算符
           if (elem.charAt(0) != '$' && !elem.equals("(")
&& !elem.equals(")")) {
               res.add(elem);
               continue;
           }
```

```
            // elem 为括号或是栈空且 elem 为 And 或 Or 时，直接入栈
            if (elem.equals("(") || (elem.charAt(0) == '$') &&
stack.isEmpty()) {
                stack.push(elem);
                continue;
            }
            // elem 为右括号，出栈直到遇到左括号
            if (elem.equals(")")) {
                while (!stack.isEmpty() && !stack.peek().equals("(")) {
                    res.add(stack.poll());
                }
                stack.poll();
                continue;
            }
            // 其余情况，将优先级比 elem 较大的弹栈，最后入栈 elem
            while (!stack.isEmpty()
                    && !stack.peek().equals("(")
                    && elem.equals("$or") && (stack.peek().equals("$and")
|| stack.peek().equals("$or"))) {
                res.add(stack.poll());
            }
            stack.push(elem);
        }
        // 清空栈
        while (!stack.isEmpty()) {
            res.add(stack.poll());
        }
        return res;
    }

    /**
     * 辅助方法：当前行是否满足复合条件表达式
```

```
     * @param conditions 包含and与or与括号的复合条件表达式
     * @param bo 当前所计算的关系
     * @param curRow 当前判断行
     * @return 当前行是否符合结果
     */
    private static boolean judgeMultipleCondition(List<String>
conditions, RelationBo bo, int curRow) throws WrongColumnNameException
{
        Deque<Boolean> stack = new ArrayDeque<>();
        for (String elem : conditions) {
            if (elem.charAt(0) != '$') {
                stack.push(judgeSingleCondition(elem, bo, curRow));
                continue;
            }
            Boolean b1 = stack.poll(), b2 = stack.poll();
            if (b1 == null || b2 == null) {
                try {
                    throw new ComputingException();
                } catch (ComputingException e) {
                    log.error("条件表达式判断错误", e);
                }
            }
            Boolean res = null;
            if (elem.equals("$and")) {
                res = Boolean.TRUE.equals(b1) &&
Boolean.TRUE.equals(b2);
            } else {
                res = Boolean.TRUE.equals(b1) ||
Boolean.TRUE.equals(b2);
            }
            stack.push(res);
        }
        return Boolean.TRUE.equals(stack.peek());
```

```
    }

    /**
     * 辅助方法：单个条件判断
     * @param condition 单个条件，含大于(>)、小于(<)、等于(=)、小于等于(<=)、大于等于(>=)、不等于(!=)
     * @param bo 当前关系
     * @param curRow 需要判断的行
     * @return 当前行是否符合条件
     */
    private static boolean judgeSingleCondition(String condition, RelationBo bo, int curRow) throws WrongColumnNameException {
        //1 处理condition: 列名 符号 内容
        String sysmbol = isWhat(condition);
        //2 判断
        if(Objects.equals(sysmbol,">")){
            String[] temp = condition.split(">");
            int colNum = bo.getColIndexByName(temp[0]);
            double num = Double.parseDouble(temp[1]);
            if(Double.parseDouble(bo.getContent()[curRow][colNum])>num){
                return true;
            }
            return false;
        }
        else if(Objects.equals(sysmbol,"<")){
            String[] temp = condition.split("<");
            int colNum = bo.getColIndexByName(temp[0]);
            double num = Double.parseDouble(temp[1]);
            if(Double.parseDouble(bo.getContent()[curRow][colNum])<num){
                return true;
            }
            return false;
```

```
        }
        else if(Objects.equals(sysmbol,"<=")){
            String[] temp = condition.split("<=");
            int colNum = bo.getColIndexByName(temp[0]);
            double num = Double.parseDouble(temp[1]);

if(Double.parseDouble(bo.getContent()[curRow][colNum])<=num){
                return true;
            }
            return false;
        }
        else if(Objects.equals(sysmbol,">=")){
            String[] temp = condition.split(">=");
            int colNum = bo.getColIndexByName(temp[0]);
            double num = Double.parseDouble(temp[1]);

if(Double.parseDouble(bo.getContent()[curRow][colNum])>=num){
                return true;
            }
            return false;
        }
        else if(Objects.equals(sysmbol,"=")){
            String[] temp = condition.split("=");
            int colNum = bo.getColIndexByName(temp[0]);
            if(Objects.equals(bo.getContent()[curRow][colNum],temp[1])){
                return true;
            }
            return false;
        }
        else if(Objects.equals(sysmbol,"!=")){
            String[] temp = condition.split("!=");
            int colNum = bo.getColIndexByName(temp[0]);
```

```
if(!Objects.equals(bo.getContent()[curRow][colNum],temp[1])){
                return true;
            }
            return false;
        }
        return false;
    }

    /**
     * 辅助方法：判断单个条件符号是什么
     * 单个条件，含大于(>)、小于(<)、等于(=)、小于等于(<=)、大于等于(>=)、不等于(!=)
     * @param condition 条件
     * @return 符号
     */
    private static String isWhat(String condition){
        String[] temp1 = condition.split(">=");
        if(temp1.length!=1) return ">=";
        String[] temp2 = condition.split("<=");
        if(temp2.length!=1) return "<=";
        String[] temp3 = condition.split("!");
        if(temp3.length!=1) return "!=";
        String[] temp4 = condition.split(">");
        if(temp4.length!=1) return ">";
        String[] temp5 = condition.split("<");
        if(temp5.length!=1) return "<";
        return "=";
    }
```

g. 主要函数说明

1. analysis：对字符串进行词法分析。入参为原始表达式字符串，出参为是否合法以及切割后的字符串数组。

2. startState：词法分析中的初态。

3. parsing：对字符串进行语法分析。入参为切割后的字符串数组，出参为是否合法。

4. parse：将关系表达式转化为逆波兰表达式的形式，入参为字符串形式的表达式，出参为一个字符串数组，每一项都为逆波兰表达式中的一项。

5. calculate：计算，将逆波兰表达式计算除最后的结果，入参为字符串数组，出参为一个关系实例（RelationBo 类的实例）。

3.3 模块间接口详细设计

3.3.1 后端模块内部接口

后端程序中主要涉及到几个主要接口。ComputingService 用于表达式的计算，对外提供了以表达式字符串为入参计算表达式的接口。RelationService 用于关系业务的处理，对外暴露的接口可以执行关系的新增、删除、重置等操作。JudgementOfLegalityService 用于对输入表达式进行合法性判断。

a. RelationService 接口定义（关系业务模块）

```
/**
 * @program: operation_system
 * @description: 管理（新建/删除）Relation 接口
 * @author: Xuan
 * @create: 2022-10-18 20:39
 **/
public interface RelationService {

    /**
     * @param relationVo 前端传来的Relations
     * @throws ParamLenException 参数不对时抛出此异常
```

```
     */
    void insertRelation(RelationVo relationVo) throws
ParamLenException;

    /**
     * 删除关系
     */
    void deleteRelation(String name);

    /**
     * 删除所有已建立关系
     */
    void deleteAll();

    /**
     * 是否存在名为 key 的关系
     * @param key
     * @return
     */
    boolean contains(String key);

    /**
     * 获取 name 对应的 Bo
     * @param name
     * @return
     */
    RelationBo get(String name);
}
```

需要调用时，在调用类内部通过 bean 注入接口实例至成员变量中，调用实例对应的方法传入正确的参数以使用。

b. ComputingService 接口定义（表达式计算模块）

```
/**
 * @program: operation_system
 * @description: 计算表达式接口
 * @author: Xuan
 * @create: 2022-10-17 15:12
 **/
public interface ComputingService {

    /**
     * @param expression 关系代数表达式
     * @return 计算后的结果
     */
    RelationVo compute(String expression) throws ComputingException;

}
```

同样，在需要调用时，在调用类内部通过 bean 注入接口实例至成员变量中，调用实例对应的方法传入正确的参数以使用。

c. JudgementOfLegalityService 接口定义

```
public interface JudgmentOfLegalityService {

    /**
     * @param expression 待计算的表达式
     * @return 表达式是否合法
     */
    boolean judgeLegality(String expression);
}
```

3.3.2 前后端模块间接口

前后端接口交互采用 http 的形式，需要访问接口时通过访问对应的 URL 并使用约定的 http 方法来访问。如需要新增关系时，前端访问的接口地址为{后端 ip 地

址}:8081/api/insert/，使用 POST 方法，将关系数据以 Json 格式放入数据体中。

a. 新增关系

1. 调用方式：POST
2. URL：/api/insert/
3. 入参：relation
4. 出参：data 为空的 response
5. 说明：新增关系，保存至后端。支持传入一个 Json 格式的关系。

例子如下。

入参

```
{
    "relation_name":"student",
    "row_len":3,
    "col_len":2,
    "col_name":"name,age,gender",
    "content":"Johnny,18,male,Jack,20,male"
}
```

出参

```
{
    "code":200,
    "msg":"ok",
    "data":{}
}
```

b. 删除关系

1. 调用方式：POST
2. URL：/api/delete/
3. 入参：关系名
4. 出参：data 为空的 response

5. 说明：删除关系。

例子如下。

入参

```
{
    "name":"student"
}
```

出参

```
{
    "code":200,
    "msg":"ok",
    "data":{}
}
```

c. 重置

1. 调用方式：GET
2. URL：/api/delete_all/
3. 入参：无
4. 出参：data 为空的 response
5. 说明：删除已有的所有关系。

例子如下。

出参

```
{
    "code":200,
    "msg":"ok",
    "data":{}
}
```

d. 计算关系代数

1. 调用方式：POST
2. URL：/api/compute/
3. 入参：表达式字符串
4. 出参：data 为计算结果（一个 relation）的 response
5. 说明：计算关系代数表达式，内部出现的关系必须事先定义过。

关系运算符在表达式字符串中的表示对应关系如下：

运算符	含义	在字符串的表示	举例	说明
∪	并	#or	A #or B	
∩	交	#and	A #and B	
−	差	#diff	A #diff B	
×	笛卡尔积	#prod	A #prod B	
σ	选择	#select	#select[A,条件表达式,1]	参数为关系名、条件、固定参数
π	投影	#project	#project[A,name,age,2]	参数为关系名、投影列名、投影列数
⋈	连接	#join	A #join B	连接运算仅支持自然连接
÷	除	#div	A #div B	
(	左括号	(	-	-
)	右括号	)	-	-

共计 8 个运算类型：并、交、差、笛卡尔积、选择、投影、连接、除。

优先级定义：括号 > 选择 = 投影 > 连接 > 差 > 积 = 除 > 交 > 并

条件表达式中支持的运算符：>, <, =, >=, <=, !=。以及使用$and 与$or 来连接多个条件。

如：

sage>=3|$and|ssex=女|$or|sage>=5|$and|ssex=男

(|sage>=3|$or|ssex=女|)|$and|(|sage>=5|$or|ssex=男|)

例子如下。

入参

注意：任意两个符号（运算符、关系名或是括号）之间请添加一个英文空格。

```
{
"expression":"( student #or class ) #and teacher",
"original_expression":"(student∪class)∩teacher"

}
```

出参

```
{
"code":200,
"msg":"ok",
"data":{
    "relation_name":"student",
    "row_len":3,
    "col_len":2,
    "col_name":"name,age,gender",
    "content":"Johnny,18,male,Jack,20,male"
}
}
```

第四章 软件测试

4.1 简介

4.1.1 目的

这一测试文档有助于实现以下目标：

- 明确测试对象、设计测试流程并以文档的形式总结测试过程。
- 对已经编码完成的软件各个功能模块进行测试。
- 分析测试结果，描述系统是否符合需求。

4.1.2 背景

要求设计并实现一个具有良好交互界面的**关系代数运算系统**，主要实现关系代数的运算，包括并、交、差、笛卡尔积、选择、投影、连接、除这 8 种运算。

系统测试围绕其主要功能进行，需要测试该系统的八种运算是否能够正常进行并得到正确答案。

4.1.3 范围

包含的测试类型有：功能测试、用户界面测试、性能评价、安全和访问控制测试、配置和安装测试。

4.2 测试类型

4.2.1 功能测试

测试对象的功能测试应该侧重于可以被直接追踪到用例或业务功能和业务规则的所有测试需求。这些测试的目标在于核实能否正确地接受、处理和检索数据以及业务规则是否正确实施。这种类型的测试基于黑盒方法，即通过图形用户界面 (GUI) 与应用程序交互并分析输出结果来验证应用程序及其内部进程。以下列出的是每个应用程序推荐的测试方法概要：

测试目标：	确保测试对象的功能正常，其中包括关系表添加、关系代数运算处理、结果打印等。
方法：	1. 利用有效的数据来执行各个用例、用例流或功能，测试在使用有效数据时是否能够得到预期结果。 2. 利用无效的数据来执行各个用例、用例流或功能，测试在使用无效数据时是否能够显示相应的错误或警告消息。
完成标准：	1. 所计划的测试已全部执行。 2. 所发现的缺陷已全部解决。
需考虑的特殊事项：	测试过程中若后端重启，需要刷新网页并对关系表进行重新输入。

4.2.2 用户界面测试

通过用户界面 (UI) 测试来核实用户与软件的交互。UI 测试的目标在于确保用户界面向用户提供了适当的访问和浏览测试对象功能的操作。除此之外，UI 测试还要确保 UI 功能内部的对象符合预期要求，并遵循公司或行业的标准。

测试目标：	1. 核实访问方法（鼠标移动、鼠标点击、方向键移动光标等）的使用。 2. 窗口的对象和特征（例如：大小、位置、状态等都符合标准）。
方法：	为窗口创建或修改测试，以核实窗口可正确地进行浏览，并处于正常的对象状态。
完成标准：	证实窗口与基准版本保持一致，或符合可接受标准。
需考虑的特殊事项：	无。

4.2.3 性能评价

性能评价是一种性能测试，它对响应时间、事务处理速率和其他与时间相关的需求进行评测和评估。性能评价的目标是核实性能需求是否都已满足。实施和执行性能评价的目的是将测试对象的性能行为当作条件（例如工作量或硬件配置）的一种函数来进行评价和微调。

注：以下事务均指“逻辑业务事务”。这种事务被定义为将由系统的某个主角通过使用测试对象来执行的特定用例，例如，添加或修改某个合同。

测试目标：	核实所指定的事务或业务功能在以下情况下的性能行为： 1. 正常的预期工作量 2. 预期的最繁重工作量
方法：	1. 使用为功能或业务周期测试制定的测试过程。 2. 通过增加单次输入的运算式复杂度来增加工作量。 3. 脚本应该在一台计算机上运行（最好是以单个用户、单个事务为基准），并在多台客户机（虚拟的或实际的客户机，请参见下面的“需考虑的特殊事项”）上重复。
完成标准：	1. 单个事务或单个用户：在每个事务所预期或要求的时间范围内成功地完成测试脚本，没有发生任何故障。 2. 多个事务或多个用户：在可接受的时间范围内成功地完成测试脚本，没有发生任何故障。
需考虑的特殊事项：	性能测试应该在专用的计算机上或在专用的机时内执行，以便实现完全的控制和精确的评测。

4.2.4 配置测试

配置测试核实测试对象在不同的软件和硬件配置中的运行情况。在大多数生产环境中，客户机工作站、网络连接和数据库服务器的具体硬件规格会有所不同。

客户机工作站可能会安装不同的软件,例如，应用程序、驱动程序等。而且在任何时候，都可能运行许多不同的软件组合，从而占用不同的资源。

测试目标：	核实测试对象可在要求的硬件和软件配置中正常运行。
方法：	1. 在测试过程中或在测试开始之前，打开各种与非测试对象相关的软件（例如 Microsoft 应用程序：Excel 和 Word），然后将其关闭。 2. 执行所选的事务，以模拟用户与测试对象软件和非测试对象软件之间的交互。 3. 重复上述步骤，尽量减少客户机工作站上的常规可用内存。
完成标准：	对于测试对象软件和非测试对象软件的各种组合，所有事务都成功完成，没有出现任何故障。
需考虑的特殊事项：	需要考虑系统的目标用户通常使用哪些应用程序，再进行模拟。

4.2.5 安装测试

安装测试有两个目的。第一个目的是确保该软件能够在所有可能的配置下进行安装，例如，进行首次安装、升级、完整的或自定义的安装，以及在正常和异常情况下安装。异常情况包括磁盘空间不足、缺少目录创建权限等。第二个目的是核实软件在安装后可立即正常运行。

测试目标：	核实测试对象可正确地安装到各种所需的硬件配置中。
方法：	使用多台不同硬件配置的电脑进行系统安装测试。
完成标准：	系统成功运行，功能使用正常，没有出现任何故障。

需考虑的特殊事项：	应该选择事务才能准确地测试出系统已经成功安装，而且没有遗漏主要的软件构件？

4.3 工具

此项目将使用以下工具：

	工具	厂商/自行研制	版本
JavaScript 运行环境	Node.js	NODE.JS FOUNDATION	较新版本
Java 集成开发环境	IntelliJ IDEA	JetBrains	较新版本
网络浏览器	Google Chrome	Google	较新版本

4.4 测试过程

明确测试类型并安装好相关测试工具后，根据测试类型设计测试用例，依次对软件进行测试，记录测试结果，并对结果进行评估。

4.4.1 功能测试

明确测试事项和具体事项内容：

测试事项	具体事项内容
添加自定义表	利用有效的和无效的数据来测试自定义表的添加功能。
表达式输入	利用有效的和无效的数据来测试表达式的输入功能。
关系代数运算	利用有效的和无效的数据对八种运算进行功能测试。
结果表打印	检查结果表在八种运算中是否都能够正确打印。

a. 添加自定义表

该功能要求用户自行输入表名、行数、列数、列名（用英文逗号分隔）、关系

表内容（列间以英文逗号分隔，行间以回车分隔）用以自定义表格。

用户输入完毕后按下“确认关系”，系统会判断该表是否合法，若合法则添加入数据库，不合法则报告错误信息。

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估
{{A},{1},{2},{a,b},{1,2}}	报告“输入关系表成功”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
{{A},{2},{2},{a,b},{1,2}}	报告“关系表的内容与行列不匹配”。	结果正确。
{{A},{2},{2},{a，b},{1,2}}	报告“列名个数与列数不匹配”。	结果正确。
{{A},{1},{2},{a,b},{1，2}}	报告“关系表的内容与行列不匹配”。	结果正确。

该功能正常。

b. 表达式输入

该功能要求用户输入运算表达式。

用户输入完毕后按下“提交”，系统会判断表达式是否合法，若合法打印结果表，不合法则报告错误信息。

前置关系表：

A	{{a,b},{1,2}}
B	{{a,b},{3,4}}

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估

A∩B	打印{{a,b}}，报告“返回空表”。	结果正确。
A∩(A∩B)	打印{{a,b}}，报告“返回空表”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
AB	报告“计算异常”。	结果正确。
(A∩B	报告“计算异常”。	结果正确。
A∩∩A	报告“异常”。	结果正确。

该功能正常。

c. 关系代数运算

前置关系表：

A	{{a,b},{1,2}}
B	{{a,b},{3,4}}
C	{{c},{5}}
D	{{a},{1}}

1. 交

该运算保留两表相同的元组，要求两表结构相同。

使用有效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A∩A	打印{{a,b},{1,2}}。	结果正确。
A∩B	打印{{a,b}}，报告“返回空表”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**

A∩C	报告“存在不合法的运算，如：结构不同的两个表进行交并差运算等”。	结果正确。

该功能正常。

2. 并

该运算合并两表的元组，要求两表结构相同。

使用有效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A∪A	打印{{a,b},{1,2}}。	结果正确。
A∪B	打印{{a,b},{1,2,3,4}}。	结果正确。

使用无效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A∪C	报告“存在不合法的运算，如：结构不同的两个表进行交并差运算等”。	结果正确。

该功能正常。

3. 差

该运算保留表 1 中与表 2 不重复的部分，要求两表结构相同。

使用有效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A-A	打印{{a,b}}，报告“返回空表”。	结果正确。
A-B	打印{{a,b},{1,2}}。	结果正确。
(A-A)-B	打印{{a,b}}，报告“返回空表”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
A-C	报告“存在不合法的运算，如：结构不同的两个表进行交并差运算等”。	结果正确。

该功能正常。

4. 笛卡尔积

该运算返回表 1 和表 2 的直积，列名相同则重新命名。

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估
A×A	打印{{Relation0.a,Relation0.b,Relation1.a,Relation1.b},{1,2,1,2}}。	结果正确。
A×B	打印{{Relation0.a,Relation0.b,Relation1.a,Relation1.b},{1,2,3,4 }}。	结果正确。

(A-A)×B	打印{{Relation0.a,Relation0.b,Relation1.a,Relation1.b}}，报告“返回空表”。	结果正确。
(A÷A)-B	打印{{a,b},{3,4}}，报告“返回空表”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
A×E	报告“异常”。	结果正确。

该功能正常。

5. 选择

该运算保留表中符合条件的元组。

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估
σ [A=1][A]	打印{{a,b},{1,2}}。	结果正确。
σ [A=2][A]	打印{{a,b}}，报告“返回空表”。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
σ [C=2][A]	报告“表达式中有不存在的列名”。	结果正确。

σ [A=1][A÷A]	报告“表达式中有不存在的列名”。	结果正确。

该功能正常。

6. 投影

该运算保留表中特定列。

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估
π [A,B][A]	打印{{a,b},{1,2}}。	结果正确。
π [A][A]	打印{{a},{1}。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
π [C][A]	报告“表达式中有不存在的列名”。	结果正确。
π [A][A÷A]	报告“表达式中有不存在的列名”。	结果正确。

该功能正常。

7. 连接

该运算对输入两表进行自然连接，即等值连接。

使用有效数据对该功能进行测试：

测试用例	测试结果	结果评估
A⋈B	打印{{a,b}}，报告“返回空表”。	结果正确。
A⋈C	打印{{a,b,c},{1,2,5}}。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估

A⋈E	报告“异常”。	结果正确。

该功能正常。

8. 除

该运算对输入两表进行除操作。

使用有效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A÷D	打印{{b},{2}}。	结果正确。
B÷D	打印{{b}}，报告“返回空表”。	结果正确。
(A÷A)÷D	报告“返回空表”。	

使用无效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A÷E	报告“异常”。	结果正确。

该功能正常。

d. 结果表打印

前置关系表：

A	{{a,b},{1,2}}
B	{{a,b},{3,4}}

使用有效数据对该功能进行测试：

测试用例	**测试结果**	**结果评估**
A∩B	打印{{a,b}}，报告“返回空表”。	结果正确。
A∪B	打印{{a,b},{1,2,3,4}}。	结果正确。

使用无效数据对该功能进行测试：

测试用例	测试结果	结果评估
A÷A	报告“返回空表”。	结果正确。

该功能正常。

4.4.2 用户界面测试

明确测试事项和具体事项内容：

测试事项	具体事项内容
核实访问方法的使用。	测试鼠标移动、鼠标点击、方向键移动光标等是否正常。
核实窗口的对象和特征符合标准。	大小、位置、状态等都符合标准。

a. 核实访问方法的使用

测试用例	测试结果	结果评估
鼠标移动	鼠标可在网页内随意移动。	结果正确。
鼠标点击	鼠标可在网页内指定位置进行点击，网页接收到点击信息并返回相应结果。	结果正确。
方向键移动光标	输入栏中方向键可控制光标左右移动。	结果正确。

该功能正常。

b. 核实窗口的对象和特征符合标准

呈现为一个浏览器网页，符合标准。

4.4.3 性能评价

明确测试事项和具体事项内容：

测试事项	具体事项内容

核实所指定的事务或业务功能在预期正常的工作量情况下的性能。	进行简单表达式计算。
核实所指定的事务或业务功能在预期最繁重的工作量情况下的性能。	进行较复杂表达式计算。

a. 简单表达式计算

测试用例	测试结果	结果评估
A∩A	结果在点击瞬间计算完毕。	性能良好。
A×B	结果在点击瞬间计算完毕。	性能良好。

b. 复杂表达式计算

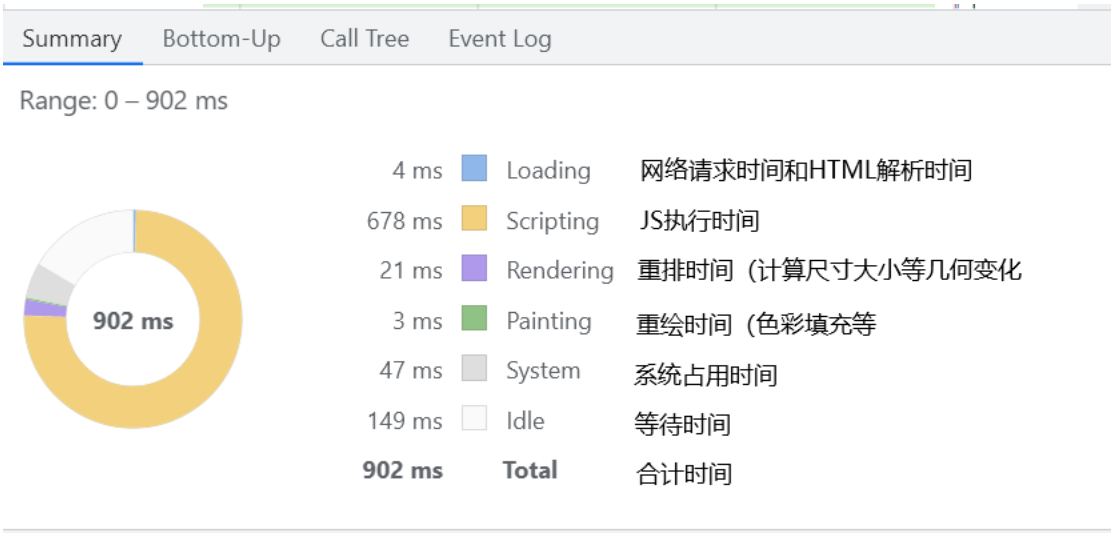
测试用例	测试结果	结果评估
(A×C)×(A×C)×(B×C)×(B×C)×(D×C)×(D×C)	结果在点击瞬间计算完毕。	性能良好。

4.4.4 配置测试

明确测试事项和具体事项内容：

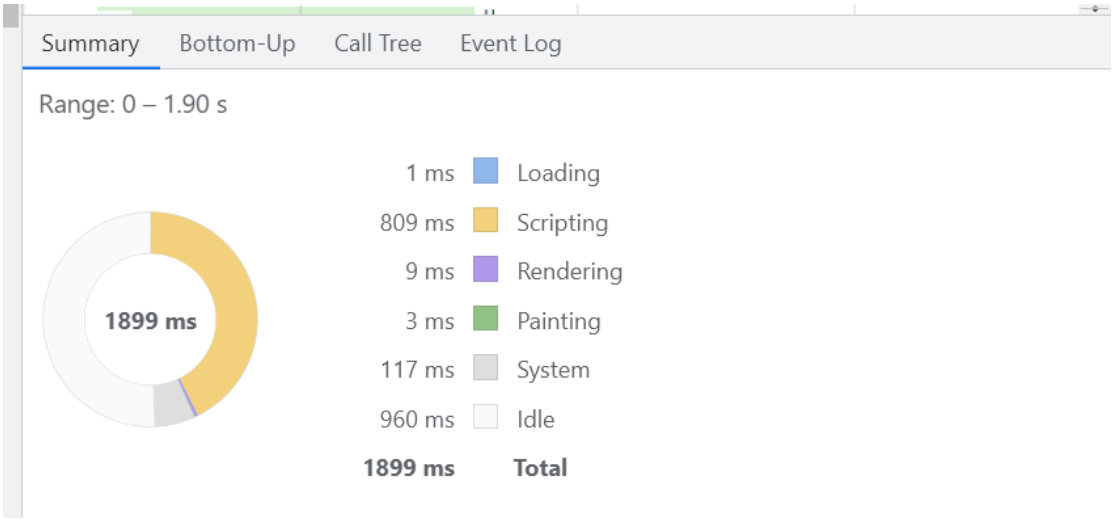
测试事项	具体事项内容
核实系统简单软件配置环境下的执行情况。	测试单运行该系统的情况下该系统的性能。
核实系统复杂软件配置环境下的执行情况。	测试多软件运行的情况下该系统的性能。

a. 测试单运行该系统的情况下该系统的性能



b. 测试多软件运行的情况下该系统的性能

在运行该系统的同时，打开了 Word 文档，QQ、微信聊天软件，力扣、BiliBili、网易云等多个网页。



4.4.5 安装测试

测试事项	具体事项内容
核实测试对象可正确地安装到各种所需的硬件配置中。	使用多台不同硬件配置的电脑进行系统安装测试。

测试用例	测试结果	结果评估
处理器：Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz，1.80 GHz **机带 RAM：**8.00 GB **系统类型：**64 位操作系统，基于 x64 的处理器	系统运行成功。	运行良好。
处理器：AMD Ryzen 7 4800H with Radeon Graphics，2.90 GHz **机带 RAM：**16.0 GB (15.4 GB 可用) **系统类型：**64 位操作系统, 基于 x64 的处理器	系统运行成功。	运行良好。
处理器：AMD Ryzen 5 5600U with Radeon Graphics，2.30 GHz **机带 RAM：**16.0 GB (15.3 GB 可用) **系统类型：**64 位操作系统, 基于 x64 的处理器	系统运行成功。	运行良好。