1. **需求分析**

功能要求： 用户输入一个复合表达式，这个表达式符合数学语法规范，如果用户输入错误要能够提醒用户并且把错误的输入删除。要能够计算基本的加减乘除，求相反数，sin正弦函数,cos余弦函数,sqrt开根号函数，以及复杂表达式的左括号、右括号，清零。

非功能要求：界面按钮交互友好，意图清晰，使用户能够一目了然界面如何使用以及软件的功能。程序具有一定健壮性，要能够处理各种潜在的异常。

设计约束：界面约束，使用相对约束，以此来适配不同型号和屏幕大小的设备。

1. **软件设计**

说明：请在这里根据需求分析的内容，进行软件设计。需要阐明App采用的架构和实现方法，如所使用的主要安卓组件，功能模块划分，数据库设计等。

1. 用户输入一个长表达式，最后点击“=”后进行计算。长表达式以String类型存储，每一次输入都是字符串的拼接。
2. 计算的时候把表达式从中缀表达式转成后缀表达式(逆波兰表达式)，利用Dijkstra的调度场(Shunting Yard)算法实现。计算的时候区分单目运算符(sin/cos/sqrt函数，求相反数)和双目运算符(加减乘除)。这两类运算符的逻辑是一样的。
3. 括号匹配：用最简单的括号计数器，输入了左括号+1，输入了右括号-1。计数器初始化为0，如果表达式正确，计算时候计数器数字应该为0。
4. 检错机制：用户每次输入都检查该输入是否合法。根据枚举情况，每个位置可以输入的类型是可以定的。
   1. 符号表示，数字：#，加减乘除：+，单目运算符：F，左括号：(，右括号：)，并列情况：/ (例如a/b表示a或b)以下为合法输入情况的列举。

|  |  |  |  |
| --- | --- | --- | --- |
| 输入类别🔘 | 上一次输入 | 当前输入 | 下一次输入 |
| 数字0～9 | #/+/( | # | #/+/) |
| 加减乘除 | #/) | + | #/( |
| 左括号 | (/+ | ( | (/# |
| 右括号 | )/# | ) | )/+ |
| 单目运算符 | +/F | F | (/F |

* 1. 列举出了输入的所有情况后，如果用户输入的情况有误，就把错误的输入从输入缓冲中删去，并且用Toast消息提示用户为什么有错。
  2. 为了方便，每次用户点击sqrt/sin/cos函数后自动添加左括号。并且在没有补齐右括号但是其他地方没有语法错误的时候，用户点击等号“=”自动补齐右括号并且计算。把交互做到最人性化。

1. 字符串匹配使用正则表达式。

注：代码中所有和操作符有关的函数都命名为响应体*operator\_name*Reactor()，以此来增加代码的可读性。

1. **核心程序代码及运行结果（30分）**

本次实验的Github项目地址：https://github.com/ShaoxuanYin/Android/tree/Calculator

(文件夹是以submodule形式挂在github上，所以只有克隆后才能看到)

主要的文件MainActivity.java和activity\_main.xml单独列出来以供在线查看。

部分代码如下(省略布局和控件):

首先声明一些必要的变量：

用于计算的栈和队列、表达式存储、上一次输入、当下输入

private final Stack<String> operator = new Stack<>();

private final Queue<String> RPN = new LinkedList<>();//RPN of reverse polish expression

ArrayList<String> originalExpression = new ArrayList<>();//could be called by iterators

String lastInput = "";//null

String nowText = "";//null

Flag变量，用来表明有效性，分别是当下输入小数点是否有效、上一个输入的是否是数字、上一个输入的是否是操作符、当前是不是刚进行过计算

private boolean dotValid = true, lastIsNumber = false, lastIsOperator = false, computed = false;

用于字符串匹配的正则表达式常量，声明为全局变量

final String NUMBER\_PATTERN = "^([-+])?\\d+(\\.\\d+)?$";

final String ADDSUBMULTIDIV\_PATTERN = "^([+\\-\*/])$";

final String LEFTPARENTHESIS\_PATTERN = "^(\\()$";

final String RIGHTPARENTHESIS\_PATERN="^(\\))$";

final String ILLEGAL\_ENDING\_PATTERN = "^([+\\-\*/.])$";

final String FUNCTION\_ENDING\_PATTERN = "^(sin|cos|sqrt)$";

final String LEGAL\_ENDING= "[0-9\\)]$";

标明运算符优先级，用<key,value>形式存储，

public Map<String,Integer> map;

{

map = new HashMap<>();

map.put("sqrt", 3);

map.put("cos", 3);

map.put("sin", 3);

map.put("(", 0);

map.put("+", 1);

map.put("-", 1);

map.put("\*", 2);

map.put("/", 2);

}

根据按下按键对应的字符判断类型，按键的onClickListener中如下，使用switch-case比用if-else速度快：

switch (view.getId()) {

case R.id.btnZero:

nowText = "0";

break;

case R.id.btnOne:

nowText = "1";

break;

case R.id.btnTwo:

nowText = "2";

break;

……………

*由于代码占据篇幅较大，本switch其余省略，其余nowText的获取同理*

}

在nowText里就是读到当下按键上按钮对应的字符串，在取到以后首先判断computed量，是否上一步点击过“=”，如果点击过与否的响应不同。我称这一段代码为热操作符响应

/\*=======Hot Operator===========\*/

if (computed) {

判断当前输入的是否是数字或者小数点，如果是的话，表明用户有重新计算并不需要上一步计算结果的意图，所以就把所有和上一步有关的变量全部清零

if (typeof(nowText) == belonging.number || typeof(nowText) == belonging.Dot) {

clear();//

}

如果输入的是属于函数(sin, cos,sqrt)则先保存上一步的结果，上一步的结果在operator的顶部，将其取出暂存后清零上一步所有变量，先调用函数响应体functionReactor()来进行函数的输入，此时表达式为fun。本程序的逻辑是，每次输入函数以后，自动添加一个“(”来保证括号的匹配此时表达式为fun(。更新标志位，最后把刚才暂存的数字输入进去，此时表达式为fun(nowText。

else if(typeof(nowText) == belonging.Function){

String tempStr = operator.peek();

clear();

functionReactor();

lastInput = "(";

lastIsNumber = false;

lastIsOperator = true;

nowText = tempStr;

numberReactor();

return;

}

如果输入的不是以上情况，输入的是双目运算符，则逻辑类似，输入上一次的结果🡪输入运算符

else{

String tempStr = operator.peek();//the “operator” is guaranteed not null, "tempStr" only use here for 3 lines

clear();

lastIsNumber = true;//correct

lastIsOperator = false;

lastInput = tempStr;

为了表达式的易读，本程序中显示的表达式所有负数都用假括号(只有用户看得见，实际不存储)围起来。

if(Double.parseDouble(tempStr) < 0){

txtResults.setText("("+tempStr+")");

}else{

txtResults.setText(tempStr);//todo: should concern about (),sin(),sqrt()

}

}

如果输入的和上次一样是等号，就直接返回不响应

if(typeof(nowText)==belonging.Equals){

return;}

}

以上是刚计算完后的热操作符部分，下面是上一步不是“=”的响应，根据输入的不同有不同Reactor()，为了篇幅省略swtich-case判断。

* 数字响应体numberReactor

private void numberReactor(){

首先判断当前输入是不是表达式的第一次输入，boolean量的判断比容器ArrayList判断空执行的快。如果不是第一次输入，那么肯定有上一次输入，如果上一次输入的是“+-\*/(.”那么就把上一次输入先存入表达式。

然后保证用户在空表达式中一直输入0，因为在数字的最高位左侧加0是没有意义的。把可能出现的情况排除以后，就可以执行平凡操作，即往上一次输入lastInput里添加这一次输入的数字，为了后续的数字也可以继续输入，所以等下一步输入操作符或者“=”前不存入表达式。

if (!lastInput.isEmpty()) {

if (lastInput.matches(ADDSUBMULTIDIV\_PATTERN) || typeof(lastInput) == belonging.LeftParenthesis

|| typeof(lastInput) == belonging.Dot) {

originalExpression.add(lastInput);//add the Operator

lastInput = "";//只是让变量“干净”，不给污染后面变量的机会

}

else if ( 0 == Double.parseDouble(lastInput) && dotValid) { // cannot let lastInput == "" step to this line

if (nowText.equals("0")) {

return;

}

backSpace(TEXTVIEW\_ONLY);// delete 0

lastInput="";

}

//above

}

//last is number

lastIsNumber = true;

lastIsOperator = false;

lastInput += nowText;

txtResults.append(nowText);

}

* 加减乘除响应体addSubMultiDivisionReactor

同样逻辑，先判断上一步输入的是什么。第一步判断是否是操作符，如果是空表达式这一步也是false。如果输入的是操作符，需判断是当前的操作符在当前位置与上一步操作符搭配是否合法，只有上一步是“)”才合法，不然给Toast信息说你输入的不对，为什么不对，然后不执行任何写入操作直接返回。如果表达式还为空的时候输入也是直接返回。如果lastInput是“)”或者数字，则把lastInput存入表达式，更新标志位和TextView，把nowText赋值给新的lastInput。

private void addSubMultiDivisionReactor(){

if (lastIsOperator) {

if (typeof(lastInput) != belonging.RightParenthesis) {//other inputs won't wright into the original expression

Log.e("error", "You cannot input " + nowText + " here!");

Toast.makeText(getApplicationContext(), "You cannot input " + nowText + " here!", Toast.LENGTH\_SHORT).show();

return;

}

}

//lastIsNumber==true

if (originalExpression.isEmpty() && lastInput.isEmpty()) {

return;

}

originalExpression.add(lastInput);//add number or a different sort of operator

lastIsNumber = false;

lastIsOperator = true;

dotValid = true;

lastInput = nowText;

txtResults.append(lastInput);

}

* 小数点响应体 dotReactor

首先判断小数点标志位是否有效，无效给Toast信息说不能在此输入小数点。有效则更新标志位，并把小数点加入到lastInput中。

private void dotReactor(){

if (dotValid) {

// ".0", .+.+

if (lastIsNumber) {

lastInput += nowText;

dotValid = false;

txtResults.append(nowText);

return;

}

}

Toast.makeText(getApplicationContext(), "You cannot put a " + nowText + " here!", Toast.LENGTH\_SHORT).show();

Log.e("error", "You cannot put a " + nowText + " here!");

}

* 相反数响应体 signReactor

当用户点击“±”的时候，只有上一次输入的数字才有效。逻辑是把TextView中上一次输入的数字删去，然后把lastInput里存的数值取相反数后重新添加到TextView中。

private void signReactor(){

if(lastIsNumber){

if(lastInput.isEmpty()){

Toast.makeText(getApplicationContext(), "sineReactor() Empty!", Toast.LENGTH\_SHORT).show();

return;

}

double tempNum = Double.parseDouble(lastInput);

int len = lastInput.length();

if(tempNum<0){

backSpace(TEXTVIEW\_ONLY);// for )

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

backSpace(TEXTVIEW\_ONLY);

}

backSpace(TEXTVIEW\_ONLY);// for (

tempNum = -tempNum;

lastInput = Double.toString(tempNum);

txtResults.append(lastInput);

}

else if(tempNum>0){

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

backSpace(TEXTVIEW\_ONLY);

}

tempNum = -tempNum;

lastInput = Double.toString(tempNum);

txtResults.append("("+lastInput+")");

}

lastIsNumber = true;

lastIsOperator = false;

}

}

* 左括号响应体 leftReactor

用一个括号计数器parenthesisCounter，每输入一次左括号+1，每输入一次右括号-1，时时刻刻只有计数器≥0才合法，因为表达式末尾缺的右括号在等好响应体内有补齐操作。左括号左边只能放函数或者双目运算符、左括号。

private void leftReactor() {

// Not support omitting operators between operands and parenthesis 2(3)(4) for now

// if(lastIsOperator || typeof(lastInput) == belonging.LeftParenthesis){//obsolete

String nowStr = txtResults.getText().toString();

if(lastInput.matches(ADDSUBMULTIDIV\_PATTERN) || typeof(lastInput) == belonging.LeftParenthesis || nowStr.isEmpty()){

if(!nowStr.isEmpty()){//PANIC error: if add a "" empty strin into the original expression

originalExpression.add(lastInput);//add the last input operator

}

lastInput=nowText;

parenthesisCounter++;

txtResults.append(lastInput);

lastIsNumber=false;

lastIsOperator=true;

}

}

* 右括号响应体 rightReactor

同理，右括号左边只能有数字或者右括号，其余皆是不合法，

private void rightReactor() {

if(lastIsNumber || typeof(lastInput) == belonging.RightParenthesis){

if(parenthesisCounter > 0){

originalExpression.add(lastInput);

lastInput=nowText;

txtResults.append(lastInput);

parenthesisCounter--;

lastIsOperator=true;

lastIsNumber=false;

lastInput=nowText;

}else{//parenthesisCounter <= 0

Log.e("error","You cannot put a \""+nowText+"\" here!");

Toast.makeText(getApplicationContext(), "You cannot put a \""+nowText+"\" here!", Toast.LENGTH\_SHORT).show();

}

}

}

* 函数响应体

函数的左边不能放右括号和数字、小数点，其余都合法，在输入函数以后帮助用户输入一个左括号。

private void functionReactor() {

String nowStr = txtResults.getText().toString();

if( nowStr.isEmpty() || (!lastIsNumber && typeof(lastInput)!= belonging.RightParenthesis)){

if(!nowStr.isEmpty() && !lastInput.isEmpty()){//redundancy?

originalExpression.add(lastInput);

}

lastInput=nowText;

txtResults.append(lastInput);//be dangling first

originalExpression.add(lastInput);

lastInput="(";//add a (

txtResults.append(lastInput);

parenthesisCounter++;

lastIsOperator=true;

lastIsNumber=false;

}

}

* 等号响应体

首先判断当前表达式是否合法，如果是操作符结尾，就把操作符删掉。点击一次进行一次检错，当合法的时候就会进行调用计算函数compute()，为空的时候不操作，如果缺右括号则补一个右括号，方便用户。

private void equalsReactor(){

if(txtResults.getText().toString().isEmpty()){

Toast.makeText(getApplicationContext(), "Input somethin' Plz for God's sake!", Toast.LENGTH\_SHORT).show();

Log.e("error","Null Expression \"Equals\" operation Exception!");

return;

}

String ch = Character.toString(lastInput.charAt(lastInput.length()-1));

if (typeof(lastInput) != belonging.RightParenthesis && typeof(lastInput) != belonging.number) {

//end with +-\*./

if (ch.matches(ILLEGAL\_ENDING\_PATTERN)) {

Log.e("error", "Your expression cannot end with \"" + lastInput + "\"");

Toast.makeText(getApplicationContext(), "Your expression cannot end with \"" + lastInput + "\"\nWe have corrected it for you~", Toast.LENGTH\_SHORT).show();

backSpace(TEXTVIEW\_AND\_ORIGINALEXPRESSION);

return;

}

//end with \*(, +(, +sin(

else if (lastInput.matches(LEFTPARENTHESIS\_PATTERN) || lastInput.matches(FUNCTION\_ENDING\_PATTERN)) {

if(lastInput.matches(LEFTPARENTHESIS\_PATTERN)){parenthesisCounter--;}

Toast.makeText(getApplicationContext(), "Your expression cannot end with \"" + lastInput + "\"\nWe have corrected it for you~", Toast.LENGTH\_SHORT).show();

backSpace(TEXTVIEW\_AND\_ORIGINALEXPRESSION);

return;

}

}

if(parenthesisCounter == 0){//correctness

if (!lastInput.isEmpty()) {

originalExpression.add(lastInput);

lastInput = "";

}

String answer = compute();

if(answer.equals("NaN")){

Toast.makeText(getApplicationContext(), "Inputs invalid", Toast.LENGTH\_SHORT).show();

clear();

}else{

txtResults.setText(MessageFormat.format("Answer: {0}", answer));

computed = true;

}

}else{//parenthesis < 0

Toast.makeText(getApplicationContext(), "You need to input a ) to make the expression complete", Toast.LENGTH\_SHORT).show();

nowText = ")";//autocomplete

rightReactor();

}

}

* 计算表达式

调用本函数的前提是表达式合法，所以本函数内不考虑合法性检查。首先先利用Dijkstra发明的Shunting Yard算法实现表达式中缀转后缀，运算符优先级即前文Map里存储的，然后根据后缀表达式的计算法则计算结果，结果放在栈“operator”顶，以供后面操作调用。代码注释描述的较为清晰。

private String compute()

for (String x : originalExpression) {//scan the entrie expression

//distinguish the sign with numbers

if (x.matches(NUMBER\_PATTERN)){// error in this line, for x.charAt-->error fixed

RPN.add(x);

}else if(x.matches(LEFTPARENTHESIS\_PATTERN)){//push a ( into the stack

operator.push(x);

}else if(x.matches(RIGHTPARENTHESIS\_PATERN)){//ensured that its appearance is valid

while( !operator.peek().matches(LEFTPARENTHESIS\_PATTERN) ){

RPN.add(operator.peek());//maybe the operator is empty sometimes?

operator.pop();

}

//Now the peek() of the stack matches (

operator.pop();//discard the (

}else if(typeof(x) == belonging.Function){

operator.push(x);//push Function in the stack

}

else{//+-\*/

//meets the greater or same precedence

//noinspection ConstantConditions

while (!operator.empty() && map.get(x) <= map.get(operator.peek())){

// while ( typeof(operator.peek().charAt(0)).compareTo(typeof(x.charAt(0))) >= 0 ) {

RPN.add(operator.peek());

operator.pop();

if (operator.empty()) {

break;/\*while\*/

}

}

operator.push(x);

}

}

while (!operator.empty()) {

RPN.add(operator.peek());

operator.pop();

}

//Now the stack(operator) is empty now, could be used as calculation buffer

//Calculate the RPN expression

String iter;

double leftOperand, rightOperand;

while (!RPN.isEmpty()) {

iter = RPN.peek();

//proved that iter is a number in the first two runs

assert iter != null;

if (!iter.matches(NUMBER\_PATTERN)) {

if(typeof(iter)==belonging.Function){

//operator definitely is not null

leftOperand = Double.parseDouble(operator.peek());

operator.pop();

switch (iter){

case "sin":

leftOperand = Math.sin(leftOperand);

break;

case "cos":

leftOperand = Math.cos(leftOperand);

break;

case "sqrt":

leftOperand = Math.sqrt(leftOperand);

break;

}

}

else{

// +-\*/ below

rightOperand = Double.parseDouble(operator.peek());

operator.pop();

leftOperand = Double.parseDouble(operator.peek());

operator.pop();

switch (iter) {

case "+":

leftOperand = leftOperand + rightOperand;

break;

case "-":

leftOperand = leftOperand - rightOperand;

break;

case "\*":

leftOperand = leftOperand \* rightOperand;

break;

case "/":

leftOperand = leftOperand / rightOperand;

break;

}

}

operator.push(Double.toString(leftOperand));

}

else{

operator.push(RPN.peek());

}

RPN.poll();

}

if(Double.parseDouble(operator.peek())==0.){

return "0";

}

return operator.peek();

}

* 清除操作 clear

private void clear () {

operator.clear();

RPN.clear();

originalExpression.clear();

txtResults.setText("");

lastIsNumber = false;

lastIsOperator = false;

dotValid = true;

computed = false;

lastInput="";//null

parenthesisCounter=0;

}

* 退格操作 backspace

分两种，一种是只删除TextView，一种是TextView和表达式一起删。前者用于取相反数的时候，后者用于在等号响应体equalsReactor的自动纠错中，把表达式末尾不合法的内容删去。

private void backSpace (boolean textViewOnlySwitcher) {

String temp = txtResults.getText().toString();

if(textViewOnlySwitcher){

if(lastInput.isEmpty()){return;}//maybe redundant here

for(int i = 0; i < lastInput.length(); i++){

temp = temp.substring(0, temp.length() - 1);

}

if(!originalExpression.isEmpty()){

lastInput = originalExpression.get(originalExpression.size()-1);

originalExpression.remove(originalExpression.size()-1);

}

}

else{//only backSpace 1 character on the screen

temp = temp.substring(0,temp.length()-1);

}

txtResults.setText(temp);

if (temp.isEmpty()) {

lastInput = "";

lastIsNumber = false;

lastIsOperator = false;

} else {

if ( lastInput.matches(NUMBER\_PATTERN)) {

lastIsNumber = true;

lastIsOperator = false;

} else {

lastIsNumber = false;

lastIsOperator = true;

}

}

}

执行结果如下(照片顺序就是输入顺序)：

Calendar

Description automatically generated Calendar

Description automatically generated Calendar

Description automatically generated with medium confidence Calendar

Description automatically generated

Calendar

Description automatically generated Calendar

Description automatically generated with medium confidence

**参考文献**

[1] 安卓官网文档Documentation for app developers <https://developer.android.com/docs>

[2] 课本《第一行代码》郭霖

[3] 算法导论