Distributed System Project

Hou, Jue

014695647

* Solution
  + General

The whole implementation is based on Java. Therefore, the Client will be in JSP, Html and JavaScript, while the Server will be in vanilla Java. Technically, JSP is a special type of servlet and there is a little part of the JSP of my implementation which should be considered as Server. Besides that, in my solution, there is a clear line between Client and Server.

In addition to Java Html and JavaScript, I choose R to generate all the plots.

The general structure of my implementation is demonstrated as followed:

JSP

Html

JavaScript

Servletlet

Calculator

Call

GET

Response

(JSON string)

R

System Call

Client

Server

JSP

Plotting

* + Client Storage

“sessionStorage” has been used to store all my history and computation result. And it is one of HTML5 features. Since “sessionStorage” can only accept string or number, most of the items will eventually be parsed into JSON string and then stored.

All the storage items will be initiated in the Client, once any of the sites in my implementation has been accessed. For the history, the record will be kept throughout the whole session.

There are three things which are stored in “sessionStorage”: cache, history and the size of the cache. The explanation of each of them is as followed:

* + - cache: The table of operation and result. It will be managed in FIFO and **will not get in use until Step 3**. The format is an array of operation and result. For example, [[“1+2”, “3”], [“2+2”, “4”]]
    - history: The list of all the equations with computation result. It will be stored as an array of string: [“1+2=3”, “2+2=4”]
    - cacheSize: The size of the cache. By default, it is 10. Setting a new cache size will not empty the cache completely but will adjust the cache to the new size in the following manner:
      * For a smaller new size, the cache will push item until the size of cache match the new size.
      * For a larger size, nothing will happen.
  + Algorithms and Functions
    - Equation Resolution- function resolve

This algorithm is implemented to resolve numbers and operators. To be more precise, this algorithm will return the first number, the first operator and the remaining of expression. By using this algorithm recursively with other functions, I can parse the input from the left to right and get tasks solved. Detailed usage will be introduced in the section of Execution Process.

* + - * Input: equation(string)
      * Output: first number (string), first operator (string), remaining of the equation (string)
      * Pseudo Code:

offset = 0;

if(the first charater is “-”) offset = 1;

op = search for operater from the position of 0+offset;

if(op!=null) return [string before op, op, remaining];

else return [input,"",""];

* + - Simplification- function replace

This function will scan the input and replace any cached item with results. It is also one example of the usage of the previous algorithm.

* Input: equation(string)
* Output: simplified equation(string)
* Pseudo Code:

main = resolve(equation); arg1 = main[0]; op = main[1]; simplified = "";

if(main[2]=="") return equation; //a single number

while(main[2]!=""){

secondary = resolve(main[2]);

arg2 = secondary[0];

rep = look for cached item with arg1, arg2 and op;

if(rep!=null){

//replace if get hit, continue to resolve from further string

simplified += rep+secondary[1];

main = resolve(secondary[2]);

}

else{

//continue if not hit, continue to resolve from the next number

simplified += arg1+op;

main = resolve(main[2]);

}

arg1 = main[0];

op = main[1];

}

simplified += arg1;

return simplified;.

* Server Interaction- function submitSimCal

This function is responsible to the actual submission of the atomic operation by JavaScript. It is embedded with auto caching in and auto checking out. But these two functions **will not be activated until Step 3**. After the interaction with Server, it will handle the JSON message and parse it into the result.

* + Input: arg1(string), arg2(string), op(string) and caching(boolean).
  + Output: result(float)
  + Pseudo Code:

Initialize result=null;

if(caching is activated)

result = cached item; // check if item is cached

if(result!=null)

return result // If indeed cached, return the result

Form the message and send it to Server.

result = parsed Server response;

if(caching is activated){

formalize the input data;

store them into the cache and push an item out if needed

}

return result;

* Taylor Series-function getSin

This function is implemented to get a set of approximations of sin(x) according to the Taylor Series. All the calculation is completed by a large number of atomic operations. To improve the accuracy, and the loop will go for 7 times. The output is a table of sin(x) approximations with their entries x. Both entries x are stored as number, while sin(x) approximations are stored as string. The formula of Taylor Series for sine is as followed:

* + Output: a two dimensional array ([float, float])
  + Pseudo Code:

Initialize result as 0

Loop n in 0-7: {

//all computation here will be also in atomic operation

a = (-1)^n;

b = 2\*n-1

c = b!;

d = x^b;

result += a/c+d;

}

return result.

* Execution Process
  + Simple Calculator

Client sends three parameters (arg1, arg2 and op) to Server as required. Server calculates the result and then sends the first arg1, arg2, op and result to Client by using redirect. Client shows the result and keeps the history. As mentioned before, there is a little part of the JSP which should be regarded as Server. The Server part of JSP is only used to handle the coming messages (result, arg1, arg2 and op) from Server and pass them to Client JavaScript function. The Client will store the calculation results in the history and display it in the browser.

* + Step 1

After clicking the button, Client will first extract the input equation. Pseudo code of parsing the input equation is as followed:

main = resolve(equation);

Initialize result as empty string;

while(main[2]!=""){

arg1 = main[0];

op = main[1];

secondary = resolve(main[2]);

arg2 = secondary[0];

//submit arg1, arg2 and op to Server

//deactivate cache function with the fourth parameter

result = submitSimCal(arg1,arg2,op,false);

Record the atomic operation and its result;

if(result == "Infinity") break; //break the loop if result is infinity

main = resolve(result+secondary[1]+secondary[2]);

}

After parsing the text, Client will display the history and each of the atomic operation

* + Step2

The step size I choose is , which is approximately 0.078539816.

The reason for such a size is that it can reach to approximations which are closest to steps: . In this way, some more precise approximations of maximum, minimum and 0 can be achieved. On the other hand, it is smaller than 0.1, which meets the requirement perfectly.

* + - Variant1

After clicking the button, Client will send the whole line to the Server. Server will parse the line in the same way as the algorithm mentioned above in the Step 1 so that the line in form of “a+b-c\*d/f\*sin(x)” can be resolved. The goal is to get the coefficient of the sin(x). The coefficient and equation as strings will be passed to R for plotting.

The actual command in R for plotting is as followed:

plot( seq(-pi,pi,pi/40), $coefficient\*sin(seq(-pi,pi,pi/40)),

main="y=$equation", xlab="X", ylab="Y", type="l");

In Windows, Server will call “ProcessBuilder” to execute the command directly.

In Linux, besides “ProcessBuilder”, an external bash script is used to execute the command due to the problem of command interaction in the department web server. Therefore, an appropriate permission setting for the script is a must.

After the plot is generated, Server will return the image to Client. The image file will then get deleted.

* + - Variant2

After clicking the button, the input will be parsed in the same way as in Step 1. Client will in the end get the coefficient of the sin(x). Loop and multiply the coefficient with each of the steps value of sin(x) and then plot on the canvas. Data of x and a\*sin(x) will be stored in a two dimensional array and passed to “plotInCanvas” function.

“plotInCanvas” function will first set the size of canvas and plot dynamically to achieve a better visual effect. The size of canvas depends on the size of container, which is the “div” that wraps the canvas. The size of plot depends on the size of canvas and the maximum point, minimum point, starting point and ending points. After scanning all the data, the ratio of scaling can be decided.

Then, all the components such as scaler for both directions will be showed. Some components such as the labels of scalers depend on the data of the input array.

Eventually, it will take the input array and plot all the point in the canvas.

* + - Variant3

After clicking the button, the coefficient can be achieved in the same manner mentioned in the previous step.

Client will first get the table of sine according to the Taylor Series.

Loop and multiply the coefficient with each of the step value of sin(x). And all the computations are in atomic operation.

After getting all the data points, plot them in the same way as what is introduced in the previous variant.

* + Step 3
    - Simplify

After clicking this button, the Client will scan over the whole input and replace all the items which has been cached before. After simplification, the input will be replaced with the simplified equation.

* + - Set Cache Size

Input the new number of size in the input field and click the button.

If the new size is larger than the current size, nothing will happen. If not, after parsing the JSON string into an array, Client will push items out until the size of the array match the new size.

The new size will be recorded in the “cacheSize” in the end.

* + - Calculate

After clicking this button, Client will execute a recursive simplification until there is nothing can be simplified.

Resolve the simplified input equation.

Loop until getting the result of the whole equation:

Send one atomic operation and cache its returning result.

Execute a recursive simplification until there is nothing can be simplified

Form the new equation and resolve it again.

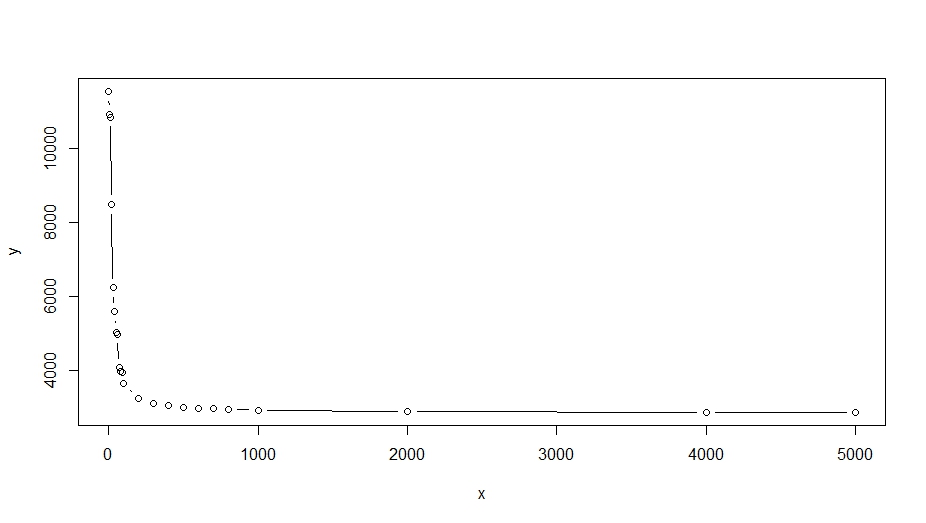
Display history and actual submitted operation.

If original equation is ended with “sin(x)”, treat the previous result as coefficient. Client will get the sine table like Step 2 Variant 3. Loop, multiply and eventually plot. Except that cache will be activated during multiplying, Client will do the exact same thing as Step 2 Variant 3.

* Message **VS** Cache Size

The equation used for experiment is y=sin(x). Cache will be cleaned after each of the experiment. The table and plot of cache size and actual number of message is as followed:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cache Size | 0 | 5 | 10 | 20 | 30 | 40 |
| Number of Messages | 11548 | 10930 | 10849 | 8488 | 6234 | 5586 |
| Cache Size | 50 | 60 | 70 | 80 | 90 | 100 |
| Number of Messages | 5033 | 4979 | 4058 | 3954 | 3939 | 3627 |
| Cache Size | 200 | 300 | 400 | 500 | 600 | 700 |
| Number of Messages | 3230 | 3086 | 3048 | 2994 | 2967 | 2967 |
| Cache Size | 800 | 1000 | 2000 | 4000 | 5000 |  |
| Number of Messages | 2940 | 2913 | 2886 | 2859 | 2859 |  |



It can be noticed that the amount of message between Client and Server is massive with no cache or a very small size of the cache. Though the number of the messages dropped dramatically with the increasing cache size, it will eventually be stabilized in a relatively large number (2859). So, it can be inferred that implementation as Step 2 Variant 3 is not a good choice.

* Optimization

As what I point out above, the design principle of Variant 3 is not good. However, there are still some ways to optimize Variant 3 and also meet the requirement in the same time. Though I did not implement all of them, I believe they can make Variant 3 a little better.

* + Law of Commutation

For addition and multiplication, the law of commutation can be used to raise the hit rate of caching.

* + Angle sum and difference identities

In my current implementation, all the sine approximation is computed according to Taylor Series. And that is a part of the reason for the massive amount of messages. Alternatively, I can implement the following formula to get approximation of sine:

The formulas will need four sine and cosine approximations to start the computations as seeds: . denotes for the step size. These four approximations will still need to be obtained through Taylor Series, while others can be obtained much quicker and simpler than the full Taylor Series formula.

* + Asynchronous Connection

At the cost of computation power for both Client and Server, asynchronous interaction between Client and Server might get a series computation done much more quickly. Another main issue might be certain computation orders will still need to be kept even though responses might be in random.

* Instruction
  + Browser: Any browser that supports HTML5 and its features, especially “sessionStorage” should be activated. Up-to-date Chrome is strongly recommended.
  + Input: Please follow the examples in each of the input field. Unexpected input will lead to a failure.
  + Access:
    - Department Web Server: <http://t-jxhou.users.cs.helsinki.fi/DSP/Index.jsp>
    - Deploy on your own computer and get access from there.
  + Deployment
    - Requirement:
      * Java 8, R
      * Tomcat 8
    - Deploy
      * Start the tomcat server.
      * Move the “DSP.war” to $CATALINA\_HOME/webapps.
        + If the OS is Linux, execute the following command to set execution permission for “r.sh”.

chmod a+x $CATALINA\_HOME/webapps/DSP/r.sh