Multi-Thread Dictionary Server

Hongyu Chen 1062897

03/09/2019

This is a project based on Server-Client structure to implement multi-thread server to handle concurrent requests from several clients to query or operate a dictionary file on server. Concisely speaking, this project is one simulation of distributed system, the basic components are sockets (to transfer the information between server and client) and threads (to ensure the response of different requests). Apart from this, there are also failure models to tolerate the failure and synchronized operation to ensure there will not be conflict between requests to modify the common sources.

Project Structure

The structure of this project will be demonstrated in 5 parts: class structure, socket/thread achievement, synchronization, failure tolerance, tokenization. Then 2 following parts about the execution/GUI and overall inaction relationship diagram. The last part is conclusion.

Class Structure

This part is about the construction of source code. In this project, there are two main classes – DictionaryServer and DictionaryClient, they are responsible with constructing service at server side and client side, also, to establish communication through socket. Additionally, one class WordOpeartion in package WordIO is used to handle the "query/add/delete" operations from clients.

Socket/Thread

Socket is an internal endpoint for sending or receiving data within a node on a computer network. There are two main protocol using the socket to send and receive data between two nodes in network – UDP and TCP. The most obvious difference between UDP and TCP is that TCP needs established connection but UDP not. Which means UDP is more convenient and lighter, but TCP is more reliable. Hence, considering the reliability, this project adopts TCP protocol to handle the data relevant to a dictionary.

As for thread, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. In this project, the server supports multi-thread execution, which means it can concurrently execute several tasks so the process is not one concatenation process. This design can improve the responsiveness of the program, decrease the resource consumption and accelerate execution process.

In this project, the multi-thread deployment is thread-per-request, which means for each request comes in server, it will be distributed a thread. The advantage of this design is it can avoid competition between threads in the queue and elevate the throughput to the maximum level.

Synchronization

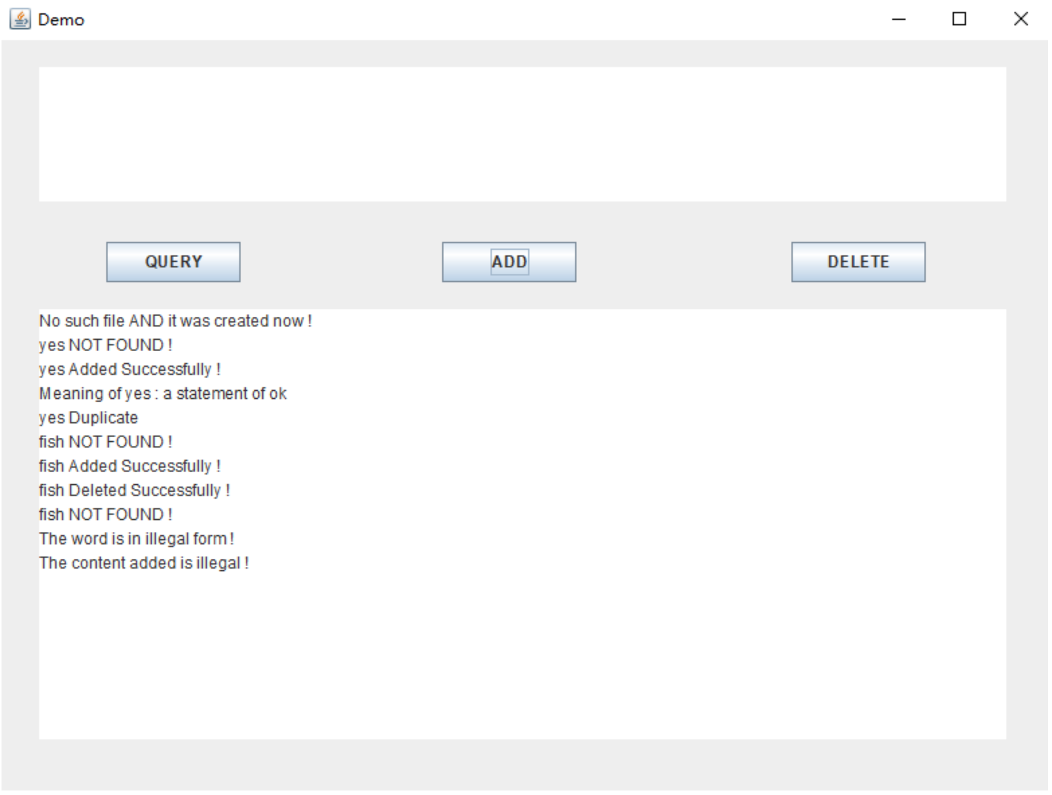
Threads can share the same resources in a process, this character can decrease the cost but throw out one another problem: What if two or more threads want to modify ('write') one same file? This may lead to conflict between threads and deteriorate the results of process. In this situation, synchronization is a very important concept because it makes the threads ordered. The concrete implement is attributing a lock to one thread, for those threads without a lock, they need to wait until the previous thread give out the lock. Specifically, locking smaller block as far as possible is conducive to saving more time. Hence, in this project, the keyword 'synchronized' only exert on class WordOperation (synchronized(WordOperation.class)).

For each operation request, it will be distributed a thread and have an object (class WordOperation), and all the requests/threads will share the same object (because this class is synchronized and this effect is global). Thus operations on dictionary from different requests/threads will not be executed concurrently, and the process will be very safe.

Failure tolerance

For a good distributed system, failure transparency is of vital importance. There will be lots of unexpected failures in a system, and the best solution is eliminating the negative effect on process, which means do not let them make the program crush. In this project, the failure defensive mechanism mainly considers following aspects.

The first is the error about opening or closing a socket (SocketException), the second is about the inputstream and outputstream in interface (IOException), the third is about checking the validation of the input from client (query/add/delete), the forth is about timeout handle, and the last is about some illegal request from clients. The first two are general inter exception throwed out by system. For example, command line from the console with illegal parameters will be handled properly, also, wrong address encapsulated in the socket from client will return host-related error. The third is using regular-expression to check whether the input from client is legal, and for the forth, I set a short waiting period (about 200ms) before the client quit a socket (This is because quitting too fast will lead no data to be received), also, I didn't choose 'while true' clause since this may be trapped in an infinite loop if the response from server was lost. (Attention, this period should change according to the size of the dictionary – longer period for larger dictionary). The last is about the result of some unsuccessful operations – (1. If query a word not in dictionary, the server will respond 'NOT FOUND' (2. If add a word already in the dictionary, the server will respond 'Duplicate'. (3. If delete a word not in dictionary, the server will respond 'NOT FOUND'. (4. If the dictionary file doesn't exist, the server will automatically create one. Above situations are illustrated as following:



```
No such file AND it was created now !
yes NOT FOUND !
yes Added Successfully !
Meaning of yes : a statement of ok
yes Duplicate
fish NOT FOUND !
fish Added Successfully !
fish Deleted Successfully !
fish NOT FOUND !
The word is in illegal form !
The content added is illegal !
```

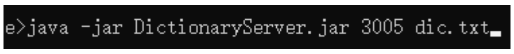
(considering the convenience, each input will be eliminated after query/add/delete operation.)

Tokenization

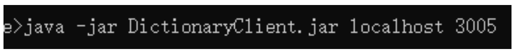
This part is some details about processing the text which to be queried, added or deleted. Token '::' represents the operations, and ':' separates the word and its meaning in add operation. For example, for query operation, the text sent to server will be like this 'query::word' and for add operation, the text will be like 'add::wo-rd:meaing'. These texts are not case-sensitive and arbitrary space is allowed since these issues have already been considered in regular-expression in failure tolerance model.

Execution and GUI

Execution command demo:

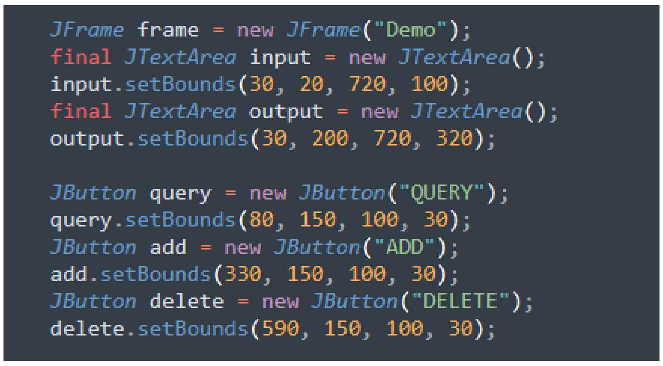


```
e>java -jar DictionaryServer.jar 3005 dic.txt_
```



```
e>java -jar DictionaryClient.jar localhost 3005
```

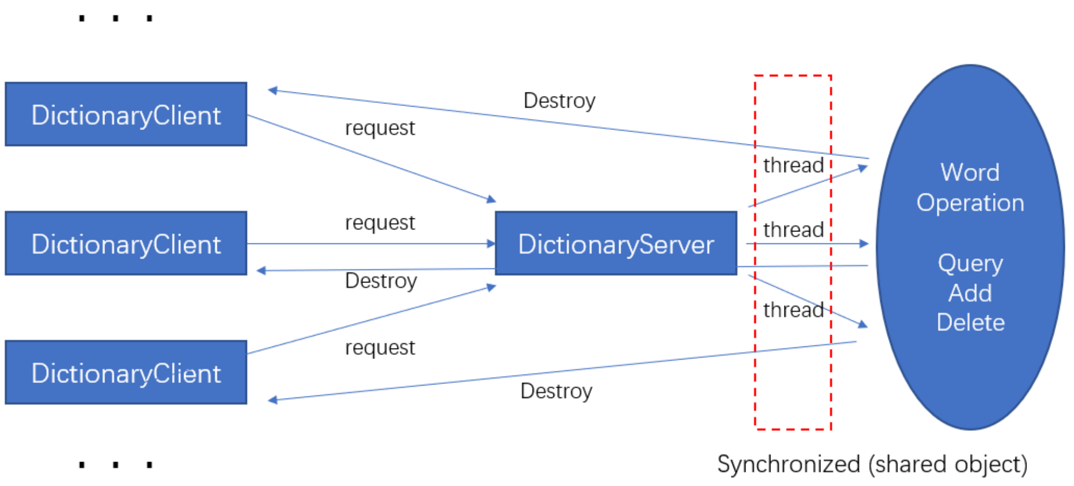
GUI of this project is supported by JFrame with BorderLayout, which is absolute layout, so it needs be set manually, a snippet of the code is illustrated as follows:



```
JFrame frame = new JFrame("Demo");
final JTextArea input = new JTextArea();
input.setBounds(30, 20, 720, 100);
final JTextArea output = new JTextArea();
output.setBounds(30, 200, 720, 320);

JButton query = new JButton("QUERY");
query.setBounds(80, 150, 100, 30);
JButton add = new JButton("ADD");
add.setBounds(330, 150, 100, 30);
JButton delete = new JButton("DELETE");
delete.setBounds(590, 150, 100, 30);
```

Overall class design and interaction diagram



Conclusion

This project achieves modifying one dictionary file (without conflict) on a server from several different clients. The code successfully simulates multi-thread server-client structure and the main achievements are as follows:

1. Implements the socket communication (TCP): benefited from TCP protocol, the data transportation will be safer and the connection between clients and server will be more reliable.
2. Thread distribution (thread-per-request): abandoning worker-pool architecture because to improve throughput, however, the design of thread-per-request may consume more resources on creating and destroying a thread.
3. Synchronizes relevant object (locking a small code block to improve performance) to avoid conflict caused by concurrency.
4. Failure tolerance designment: contains socket error, IO error, timeout error and text content error.
5. Establishes the basic GUI to process operations such as query/add/delete and demonstrates the response from server.