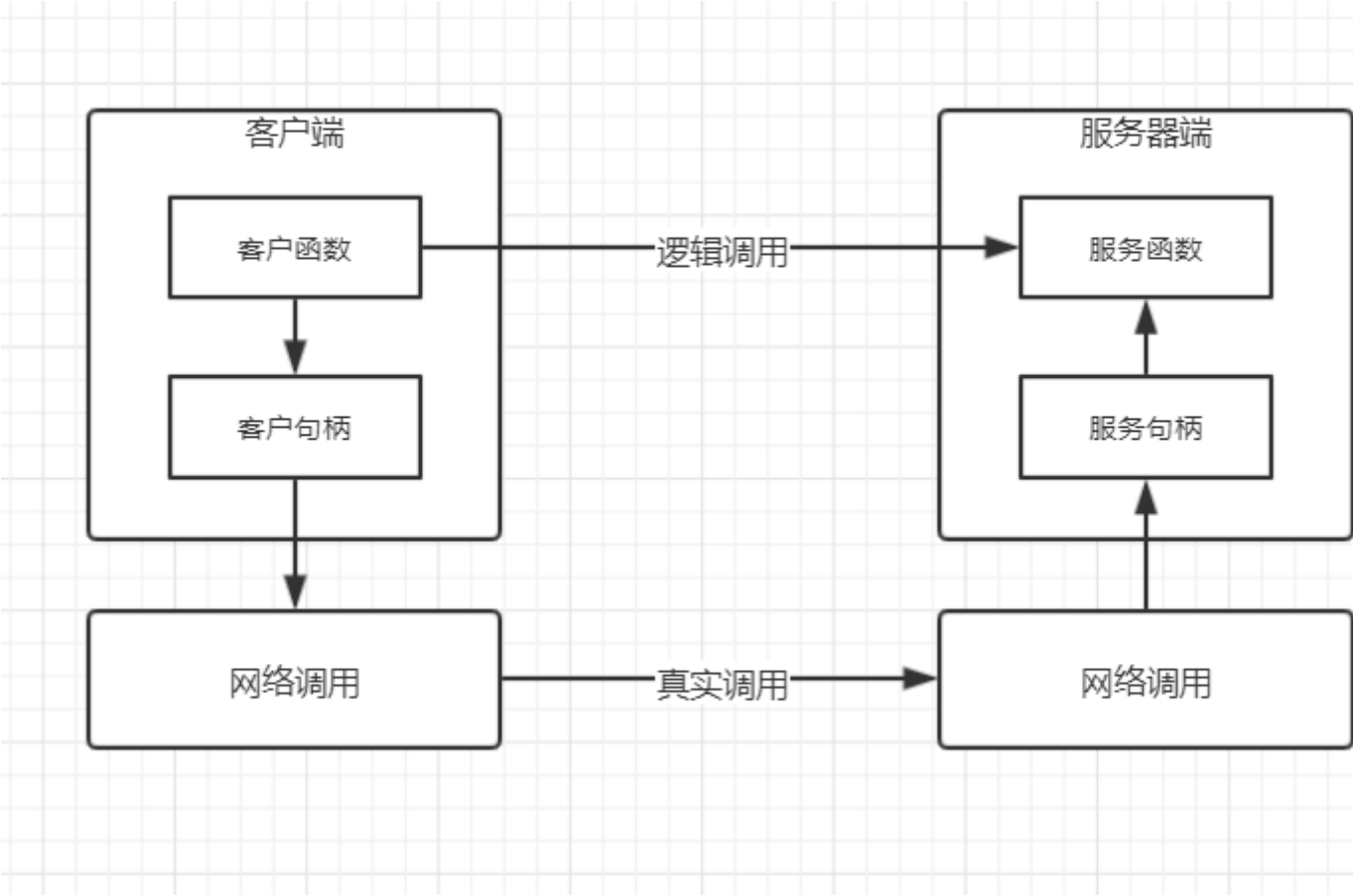
1. RPC

RPC (Remote Procedure Call) remote procedure call, that is, to call the service on the remote computer program through network communication, and this call process is as simple and transparent as calling a local method, and does not need to understand the underlying network technology protocol. RPC adopts C/S architecture, the requesting program is Client, and the service provider is Server, similar to Http request and response. The simple summary is: the method called is actually remote, and it should be as simple as calling the local method.

1) For the client: calling a local method (stub) can get the service. This stub is a proxy for remote services, and how its underlying implementation is implemented is transparent to me.

2) For remote servers: monitor whether there is a connection, call the corresponding method and return when it comes (the server is easier to understand)

The structure diagram is as follows:



1. The user calls a "local" function, which calls the client handle (the local stub of the remote service)
2. Client handle calls network communication to access remote program
3. The remote program calls the service handle after receiving the network communication and related information
4. The service handle calls the service function, the function ends and returns the result in reverse order to complete a remote call

2. Why RPC is needed

When our business volume is getting larger and larger, the effect of increasing the number of servers vertically will be less effective in improving performance. At this time, it is inevitable that we will adopt a distributed architecture to better improve performance. Each service of the distributed architecture is an independent part. When a certain business needs to be completed and depends on different service styles, these services need to call each other. At this time, the call between services requires an efficient application. Means of communication between the two, this is the reason why PRC appeared

3. RPC implementation requirements

3.1 Service provider

Provide service: realize the service provided

Service exposure: It is not enough to just realize the service. It is also necessary to expose the provided service to the outside world so that the outside world knows what and how to use the service

3.2 Service caller

Remote proxy object: when the local method is called, the remote method is actually called, so a remote proxy object is bound to be needed locally

to sum up: In order to realize RPC, there are: communication model (BIO, NIO), service location (IP, PORT), remote proxy object (local proxy of remote service), serialization (network transmission is converted into binary)

4. Simple implementation

The main objects are: server interface, server interface implementation, service exposure, client interface (share the same interface with the server), service reference

4.1 Server interface

```
public interface Service {

         // Provide two services, say hello and integer addition
        public String hello();
        public int sum(int a, int b);

}
```

4.2 Server interface implementation

```
public class ServiceImpl implements Service {

        @Override
        public String hello() {
                return "Hello World";
        }

        @Override
        public int sum(int a, int b) {
                return  a + b;
        }
}
```

4.3 Service exposure

```
public static void export(Object service, int port) {

        if (service == null || port <= 0 || port > 65535) {
        throw new RuntimeException("Arguments error");
    }

         System.out.println(service.getClass().getName() + ": "+ port + "Service Exp
osure");

        new Thread( () -> {

                try (ServerSocket server = new ServerSocket(port);) {

                        while(true){
                                try (
                                                Socket socket = server.accept();
                                                ObjectInputStream in = new ObjectIn
putStream(socket.getInputStream());
                                                ObjectOutputStream out = new Object
OutputStream(socket.getOutputStream());
                                        ) {

                                         // read method name
                                        String methodName = in.readUTF();

                                         // read parameter type
                                        Class<?>[] parameterTypes = (Class<?>[])in.
readObject();
```

```
                                        // read parameter value
                                        Object[] arguments = (Object[])in.readObjec
t();

                                        // Get method
                                        Method method = service.getClass().getMetho
d(methodName, parameterTypes);

                                                                                /
/ process result
                                        Object result = method.invoke(service, argu
ments);

                                        // Write the result
                                        out.writeObject(result);

                                } catch (Exception e) {
                                        e.printStackTrace();
                                }
                        }
                } catch (IOException e1) {
                        e1.printStackTrace();
                }
        }).start();
}
```

The logic of this exposure is that the server listens to a specific port, waits for the client to initiate a request, and then connects, then obtains the method name, parameters and other related information through the Java IO stream, and finally implements the method call through reflection and responds to the client with the result

4.4 Client interface

```
public interface ClientService {

         // Provide two services, say hello and integer addition
        public String hello();
        public int sum(int a, int b);

}
```

4.5 Service Reference

```
public static <T>T refer(Class<T> interfaceClass, String host, int port){

        if(interfaceClass == null || !interfaceClass.isInterface() || host == null |
| port <= 0 || port > 65535){
                throw new RuntimeException("Arguments error");
        }

        System.out.println("Calling remote service");

        @SuppressWarnings("unchecked")
        T proxy = (T)Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Cla
ss<?>[] {interfaceClass}, new InvocationHandler() {

                @Override
                public Object invoke(Object proxy, Method method, Object[] args) thr
ows Throwable {
```

```
                        Object result = null;
                        try (
                                        Socket socket = new Socket(host, port);
                                        ObjectOutputStream out = new ObjectOutputSt
ream(socket.getOutputStream());
                                        ObjectInputStream in = new ObjectInputStrea
m(socket.getInputStream());
                                ) {

                                out.writeUTF(method.getName());

                                out.writeObject(method.getParameterTypes());

                                out.writeObject(args);

                                result = in.readObject();

                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                        return result;
                }
        });
        return proxy;
}
```

The logic of the reference service is: create a Socket connection, serialize the related request information and send it to the server, and then wait for the response result. The transparent call uses a dynamic proxy

4.6 Testing

```
public class Test {

        public static void main(String[] args) {

                 // expose service
                ServiceImpl service = new ServiceImpl();
                RPCFramework.export(service, 8080);

                 // call service
                Client client = RPCFramework.refer(Client.class, "127.0.0.1", 8080);
                int sum = client.sum(1, 2);
                String rs = client.hello();
                 System.out.println("Remote response:" + sum);
                 System.out.println("Remote response:" + rs);
        }
}
RPC.ServiceImpl:8080----- service exposure

 Calling remote service
 Remote response: 3
 Remote response: Hello World
```

5. Thinking

5.1 Why not use Http

RPC has nothing to do with specific protocols and can be based on Http and TCP, but because TCP has relatively better performance. Http belongs to the application layer protocol, and TCP belongs to the transport layer protocol. Compared with the bottom layer, there is less encapsulation, and for reliable transmission, choose TCP instead of UDP

5.2 Commonly used RPC framework

Dubbo (Alibaba), SpringCloud, RMI (built-in JDK)

5.3 Why use dynamic proxy

Because it needs to be transparent to the user like a local call.

```
Object result = XXX(String method, String host, int port)
```

The above is actually okay, but it does not feel like calling a local method, and if an interface has multiple methods, host/port needs to be sent every time a method is called

```
// Dynamic proxy can be used like this
 ProxyObject. Method 1
 ProxyObject. Method 2

 // It is not humane if no dynamic proxy is used
XXX(String method1, String host, int port)
XXX(String method2, String host, int port)
```

5.4 Why do parameters and parameter types need to be passed separately

In order to facilitate the identification of method overloads, the method name and parameter type are required to obtain the method below

```
service.getClass().getMethod(methodName, parameterTypes)
```

6. Optimization

6.1 Network communication

In the above example, the BIO form is used, and the concurrency is not high due to blocking access. NIO can be used instead

6.2 Serialization

The JDK native method used here can only serialize classes that implement the Serializable interface, and you can use third-party class libraries to improve performance

6.3 Service load

The automatic discovery of services, the client can dynamically perceive the changes of the server, from the realization of hot deployment, the method of regular polling can be used, eg: ZooKeeper

6.4 Cluster

Clustering, which can provide load balancing

6.5 Request and Response

Requests and responses can be encoded and encapsulated, instead of sending them one by one

referenceYanyan.He

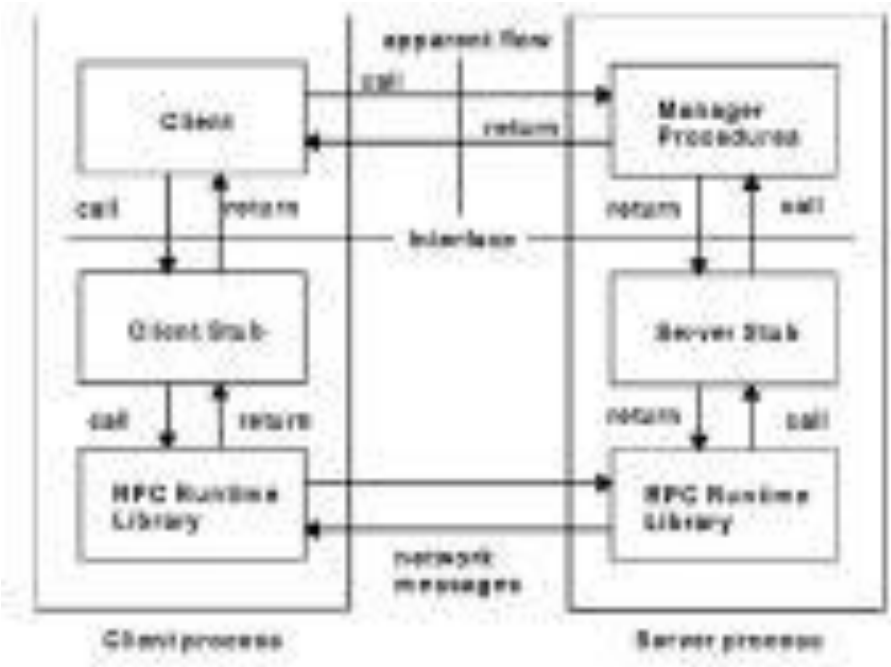
1.
2.
3.
4.
5.

HTTP file, error message prompt page, MongoDB file, user file condition restraint

Intelligent Recommendation

RabbitMQ-based on RPC implementation

In the case of using RabbitMQ to execute Command, sometimes return value information is required . At this time, it is equivalent to that after the client issues a command and listens to the queue that...



Java implementation of RPC application

One, RPC introduction what isRPC？Remote Procedure Call, Remote procedure call. In other words , the calling process code does not run locally on the caller, but to realize the connection and commu nicat...