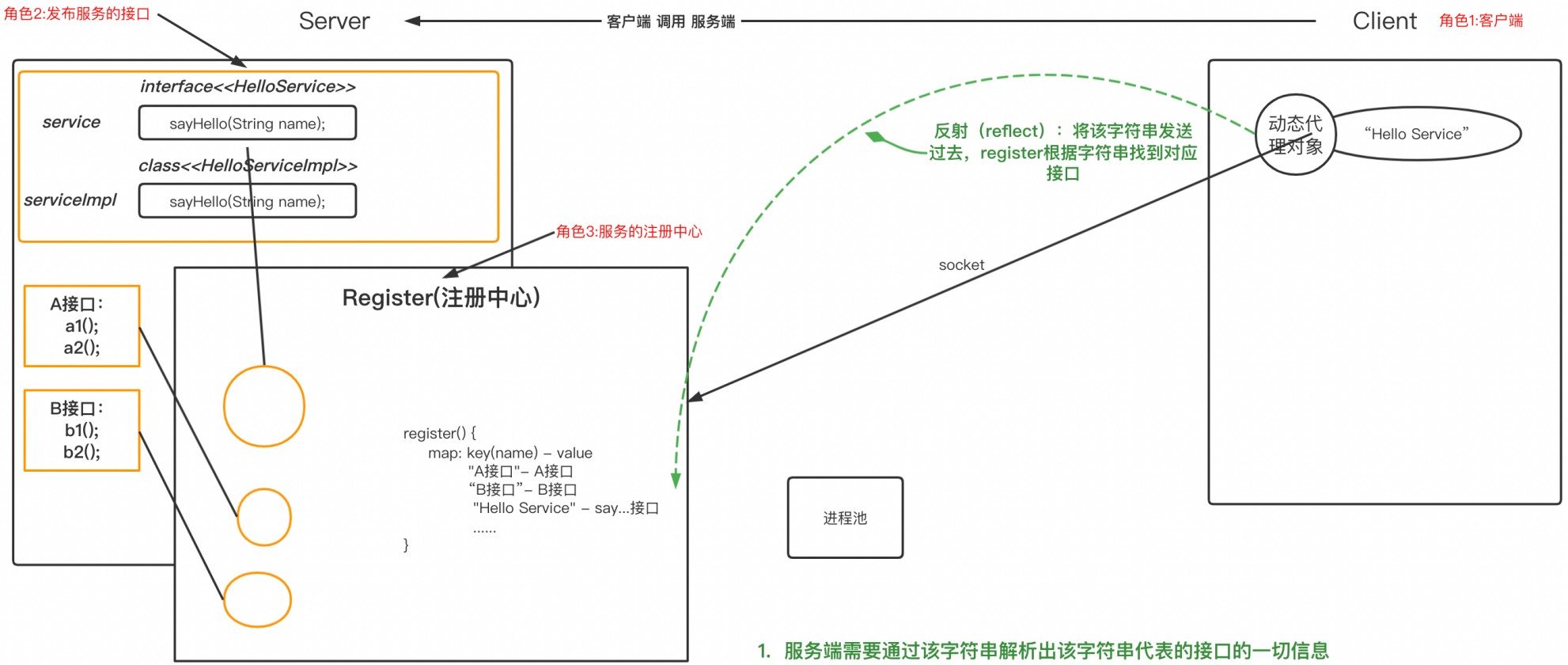
RPC: Remote Procudere Call
RMI: Remote Method Invoke



1. 服务端需要通过该字符串解析出该字符串代表的接口的一切信息
2. 服务端 – 客户端 ： socket
3. 服务端 根据 客户端 的不同请求，返回不同的接口类型；
   客户端就要根据收到的不同接口类型 -> 通过动态代理

RPC 框架 ：

<interface> Helloservice :

```
public interface HelloService {
    public String sayHi(String name);
}
```

HelloServiceImpl:

```
public class HelloServiceImpl implements HelloService{
    @Override
    public String sayHi(String name) {
        return "hi! "+ name;
    }
}
```

<interface> ServiceRegister:

```
public interface ServiceRegister {
    public void start() throws IOException, ClassNotFoundException,
NoSuchMethodException, InstantiationException, IllegalAccessException,
InvocationTargetException;
    public void close();
    public void register(String serviceName, Class serveicImpl);
}
```

ServiceRegisterImpl:

```
public class ServiceRigsterImpl implements ServiceRegister{
    // map: 服务端所有可供客户端访问的接口，都注册到map中
    // key: 接口的名字;
    // value: 具体实现
    private static HashMap<String,Class> serviceRegister = new HashMap();
    private static int port;
    // 连接池：连接池有多个连接对象，每个连接对象都可以处理一个客户请求
    private static ExecutorService executorService =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

    private static boolean isRunning = false; // 表示当前服务是否已经开启

    public ServiceRigsterImpl( int port) {
        this.port = port;
    }

    // 开启服务端的服务
    @Override
    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket();
        serverSocket.bind(new InetSocketAddress(port)); // 绑定端口
        isRunning = true;
        while (true){ // 并发 -> 多线程
            // 具体的服务内容：接受客户端请求，处理请求，并返回结果
            System.out.println("服务器已启动...");
            try {
                // 客户端每发出一次请求，则服务端从连接池中获取一个线程对象去处理
                Socket server = serverSocket.accept();
                // 每执行一次，就会从连接池里取一个线程来执行
                executorService.execute(new ServiceTask(server));
            } catch (Exception e){
                e.printStackTrace();
            }
        }
    }
    // 关闭服务
    @Override
    public void close() {
        isRunning = false;
```

```
        executorService.shutdown();
    }

    @Override
    public void register(String serviceName, Class serveicImpl) {
        serviceRegister.put(serviceName, serveicImpl);
    }
}
```

多线程：ServiceTask

```
public class ServiceTask implements Runnable{
    // map: 服务端所有可供客户端访问的接口，都注册到map中
    // key: 接口的名字;
    // value: 具体实现
    private static HashMap<String,Class> serviceRegister = new HashMap();
    private Socket server;
    public ServiceTask(Socket server) {
        this.server = server;
    }
    public ServiceTask() {
    }

    @Override
    public void run() {
        ObjectInputStream in = null;
        ObjectOutputStream out = null;
        try{
            // 接受到客户端连接 及 请求
            in = new ObjectInputStream(server.getInputStream()); // 客户端发出的所有请求
            // 因为序列化流ObjectInputStream对发送数据的顺序严格要求，因此需要参照发送顺序逐个接
收
            String serviceName = in.readUTF();
            String methodName = in.readUTF();
            Class[] parameterTypes = (Class[]) in.readObject(); // 方法的参数类型
            Object[] arguments = (Object[]) in.readObject();// 方法的参数
            // 根据客户请求，在map中找到对应的具体接口
            Class serviceClass = serviceRegister.get(serviceName); // 接口的对象
            Method serviceMethod = serviceClass.getMethod(methodName,parameterTypes);
            // 执行该方法
            Object result = serviceMethod.invoke(serviceClass.newInstance(),arguments);
// 返回任意类型（Object）的result
            // ->客户端 把方法执行完毕后的返回值 传回客户端
            out = new ObjectOutputStream(server.getOutputStream());
            out.writeObject(result);

            // 返回客户端业务
        }catch(Exception e){
            e.printStackTrace();
        }
        finally {
            if (in != null){
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (out != null){
                try {
                    out.close();
                } catch (IOException e) {
                    e.printStackTrace();
                } } } } }
```

<interface> Client:

```
public interface Client {}
```

Client:

```
public class ClientImpl {
    // 获取代表服务端接口的动态代理对象
    // serviceName: 请求的接口名
    // addr: 带请求服务端的 ip: 端口
    public static <T> T getRemoteProxyObj(Class serviceInterface, InetSocketAddress
addr) {
        /*
            Proxy.newProxyInstance(a,b,c)
            a: 类加载器: 要代理哪个类, 就将那个类的类加载器传入第一个参数
            b: 需要代理的对象, 具有哪些方法 --接口
         */
            return (T) Proxy.newProxyInstance(serviceInterface.getClassLoader(), new
Class<?>[]{serviceInterface}, new InvocationHandler() {
                @Override
                // proxy: 代理的对象
                // method: 代理的方法
                // args: 代理方法的参数列表
                public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                    // 客户端向服务端发送请求: 请求具体某一个接口
                    Socket client = new Socket();
                    ObjectOutputStream out = null;
                    ObjectInputStream in = null;

                    try{
                        // socketAddress : IP + Port
                        client.connect(addr);
                        // 发送: 序列化流 (对象流)
                        out = new ObjectOutputStream(client.getOutputStream());
                        // 需要发送的内容:
                        // 1、接口的名字
                        out.writeUTF(serviceInterface.getName());
                        // 2、方法名及其参数、参数类型
                        out.writeUTF(method.getName());
                        out.writeObject(method.getParameterTypes());
                        out.writeObject(args);
                        // 等待服务端处理...

                        // 接收服务端处理后的返回值
                        in = new ObjectInputStream(client.getInputStream());
                        // 客户端 -> 服务端 -> 返回值
                        return in.readObject();
                    } catch (Exception e){
                        e.printStackTrace();
                        return null;
                    }
                    finally {
                        if (in != null){
                            in.close();
                        }
                        if (out != null){
                            out.close();
                        }
                    }
                }
            });
    }
}
```