Report

Protocol Design

The protocol of the underlying data communication is just a request–reply model. Both clients and servers have the class `Message`. When the client want to call a method of a proxy, the underlying invocation handler will generate an object of `Message` given the called method and arguments, then the object of `Message` contains all needed information and is sent via a TCP socket after it is serialized by JVM. When the skeleton receive a TCP connection, it accepts the connection, handle it to `SkeletonReqHandler`. Inside the `SkeletonReqHandler`, it reads the object of request into convert it to a `Message` object. Based on the information provided by the `Mesage` object, it executes methods and constructs a new `Message` object as the reply. The reply `Message` object is then serialized and transmitted to the client via the TCP socket.

Problems and Solutions

In `SkeletonReqHandler`

Two Big Problems:

The argument types of a called method often *seems* to be unmatched, which leads to many `NoSuchMethodException`. For example, class `A` implements `B`, `C`, `D` and there is a method of class `E` with one argument of type `B` and we have an instance `a` of `A`, then if we write `B.class.getMethod(methodName, a.getClass())`, a `NoSuchMethodException` will be thrown because there is no such method with the argument of type `A`.

Another related problem is that when a method has primitive arguments, finding it with the wrapper classes of primitives is not acceptable, which will also lead to `NoSuchMethodException`.

Solution:

A long method has been written to solve these two problem. Basically, the polymorphism of `Java` supported by JVM is used to solve the first problem via calling `Class.isInstance(obj)`, and as for the second problem, I exchanged the primitives' classes for their wrapper classes that are the ones de–serialized arguments will have.

```
1  private Method getMethod(Class<?> claz, String methodName, Object[] args) throws RemoteException,
   NoSuchMethodException
2  {
3      if (args == null || args.length == 0)// if the desired method has no arguments.
4          return claz.getMethod(methodName, null);
5
6      //else, filter out methods that have different names and different numbers of arguments
7      List<Method> candidates = Arrays.stream(claz.getMethods())
8          .filter(method -> methodName.equals(method.getName()))
9          .filter(method -> method.getParameterCount() == args.length)
10         .collect(Collectors.toList());
11
12     if (candidates.size() == 0)
13         throw new NoSuchMethodException();
14     else
15     {
16         ArrayList<Method> matchedMethods = new ArrayList<>();
17         for (Method m : candidates)
18         {
19             Class<?>[] types = m.getParameterTypes();
20             boolean match = true;
21             //iterate over all parameter types and check whether argument match each type
22             for (int i = 0; i < types.length; i++)
23             {
24                 Class<?> argITypeOfMethod = types[i];
25                 //specially handle the cases of primitives, solved the second problem
26                 if(argITypeOfMethod.isPrimitive())
27                 {
```

```
                    if(argITypeOfMethod.equals(int.class))
                        argITypeOfMethod = Integer.class;
                    else if(argITypeOfMethod.equals(double.class))
                        argITypeOfMethod = Double.class;
                    else if(argITypeOfMethod.equals(boolean.class))
                        argITypeOfMethod = Boolean.class;
                    else if(argITypeOfMethod.equals(byte.class))
                        argITypeOfMethod = Byte.class;
                    else if(argITypeOfMethod.equals(float.class))
                        argITypeOfMethod = Float.class;
                    else if(argITypeOfMethod.equals(short.class))
                        argITypeOfMethod = Short.class;
                    else if(argITypeOfMethod.equals(char.class))
                        argITypeOfMethod = Character.class;
                    else if(argITypeOfMethod.equals(long.class))
                        argITypeOfMethod = Long.class;
                    else
                        srhLogger.severe("Should not enter this branch");
                }

                if (!argITypeOfMethod.isInstance(args[i])) // solved the first problem
                {
                    match = false;
                    break;
                }
            }
            if (match)
                matchedMethods.add(m);
        }
        if (matchedMethods.size() == 0)
        {
            throw new NoSuchMethodException();
        } else if (matchedMethods.size() > 1)
        {
            StringBuilder sb = new StringBuilder();
            sb.append("Ambiguity Exception: too many matched methods: \n");
            matchedMethods.forEach(m -> sb.append("   ").append(m.toString()).append("\n"));
            throw new RemoteException(sb.toString());
        } else
            return matchedMethods.get(0);
    }
}
```

Notice

To see what codes have been changed, see changelog