
目录

0、摘要	1.1
1、初识RPC	1.2
2、实现简单RPC	1.3
2.1、场景模拟	1.3.1
2.2、思路分析	1.3.2
2.3、代码实现	1.3.3
3、优化RPC框架	1.4
3.1、与spring整合	1.4.1
3.2、利用zookeeper实现服务注册中心	1.4.2
3.3、使用Netty中的提供NIO网络模型	1.4.3
3.4、使用Protostuff实现序列化	1.4.4

课程介绍

通过java来实现一个简单的RPC框架，熟悉java网络编程、多线程、动态代理、自定义注解、反射等知识。

[去GitBook阅读](#)

[去GitHub点赞](#)

大纲

- 1、初识RPC
- 2、实现简单RPC
 - 2.1、场景模拟
 - 2.2、思路分析
 - 2.3、代码实现
- 3、细节优化
 - 3.1、将框架与spring整合，实现服务调用透明
 - 3.2、使用Netty中的提供NIO网络模型
 - 3.3、使用Protostuff实现序列化
 - 3.4、利用zookeeper实现服务自动注册和发现

学习目标

通过学习，达到以下目的：

- 熟悉java的BIO、NIO的网络编程。
- 会使用线程池
- 熟练使用动态代理
- 会编写自定义注解，并结合spring使用
- 使用netty编写简单的网络通信
- 掌握zookeeper的节点树的基本操作
- 使用zookeeper实现服务自动注册和发现

课程内容较多，可能需要分多个课时讲完。

1、初识RPC

1.1、什么是RPC？

RPC，即 Remote Procedure Call（远程过程调用），是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。说得通俗一点就是：A计算机提供一个服务，B计算机可以像调用本地服务那样调用A计算机的服务。

通过上面的概念，我们可以知道，实现RPC主要是做到两点：

- 实现远程调用其他计算机的服务
 - 要实现远程调用，肯定是通过网络传输数据。A程序提供服务，B程序通过网络将请求参数传递给A，A本地执行后得到结果，再将结果返回给B程序。这里需要关注的有两点：
 - 1) 采用何种网络通讯协议？
 - 现在比较流行的RPC框架，都会采用TCP作为底层传输协议
 - 2) 数据传输的格式怎样？
 - 两个程序进行通讯，必须约定好数据传输格式。就好比两个人聊天，要用同一种语言，否则无法沟通。所以，我们必须定义好请求和响应的格式。另外，数据在网络中传输需要进行序列化，所以还需要约定统一的序列化的方式。
- 像调用本地服务一样调用远程服务
 - 如果仅仅是远程调用，还不算是RPC，因为RPC强调的是过程调用，调用的过程对用户而言是应该是透明的，用户不应该关心调用的细节，可以像调用本地服务一样调用远程服务。所以RPC一定要对调用的过程进行封装

1.2、问题：Http和RPC有什么关系

Http协议：超文本传输协议，是一种应用层协议。规定了网络传输的请求格式、响应格式、资源定位和操作的方式等。但是底层采用什么网络传输协议，并没有规定，不过现在都是采用TCP协议作为底层传输协议。说到这里，大家可能觉得，Http与RPC的远程调用非常像，都是按照某种规定好的数据格式进行网络通信，有请求，有响应。没错，在这点来看，两者非常相似，但是还是有一些细微差别。

- RPC并没有规定数据传输格式，这个格式可以任意指定。
- Http中还定义了资源定位的路径，RPC中并不需要
- 最重要的一点：RPC需要满足像调用本地服务一样调用远程服务，也就是对调用过程在API层面进行封装。Http协议没有这样的要求，因此请求、响应等细节需要我们自己去实现。
 - 优点：RPC方式更加透明，对用户更方便。Http方式更灵活，没有规定API和语言，跨语言、跨平台
 - 缺点：RPC方式需要在API层面进行封装，限制了开发的语言环境。

1.3、流行的RPC框架

事实上，限制的RPC框架，不仅仅是实现透明化的远程调用，更多的侧重点放在了服务治理上。实现诸如：服务自动发现、自动注册、负载均衡、服务治理等功能。例如阿里巴巴的Dubbo就是流行RPC框架的佼佼者。

2、实现简单的RPC

通过前面介绍，我们已经知道，现在主流的RPC框架其实是比较复杂的，除了“远程过程调用”以外，更多的是服务的治理。我们在入门案例中，先侧重于对“远程过程调用”的实现。接下来就来完成一个简单的RPC入门框架

2.1、场景模拟

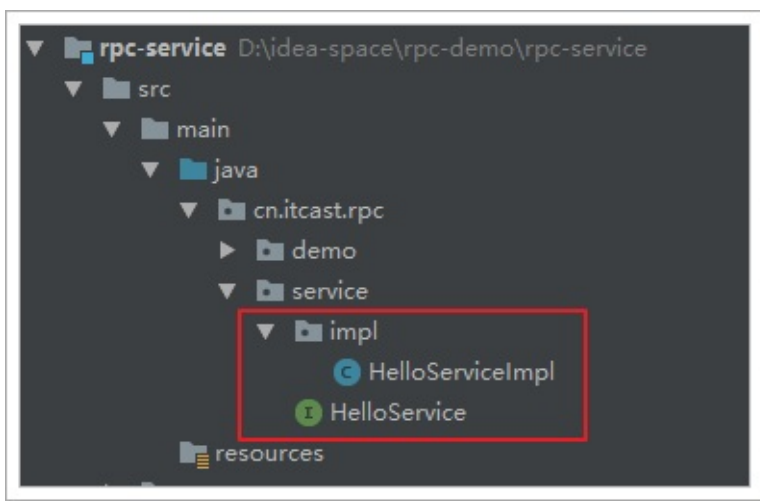
我们先模拟一个远程调用的场景：计算机A提供服务，计算机B远程调用服务。

2.1.1、服务提供方

2.1.2、服务调用方

2.1.1、服务提供方

现在我们创建一个工程rpc-service，模拟计算机A，提供一个简单的服务：



服务接口：

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

服务的具体实现：

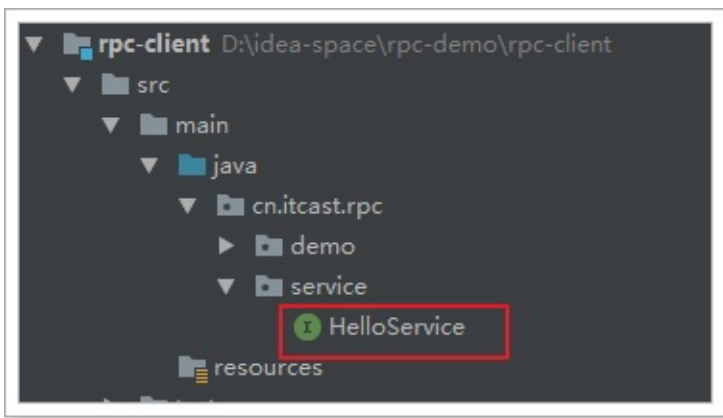
```
public class HelloServiceImpl implements HelloService{  
    public String sayHello(String name) {  
        return "hello," + name;  
    }  
}
```

调用一个本地服务，需要知道是调用哪个类的哪个方法，然后创建对象，调用方法，传递具体参数即可。例如，在计算机A中，我们直接new对象，调用方法即可：

```
public static void main(String[] args) {  
    HelloService service = new HelloServiceImpl();  
    String msg = service.sayHello("Jack");  
    System.out.println(msg); // hello, Jack  
}
```

2.1.2、服务调用方

现在，我们再创建一个工程rpc-client,模拟计算机B,这里只有HelloService接口，没有具体实现。



服务接口：

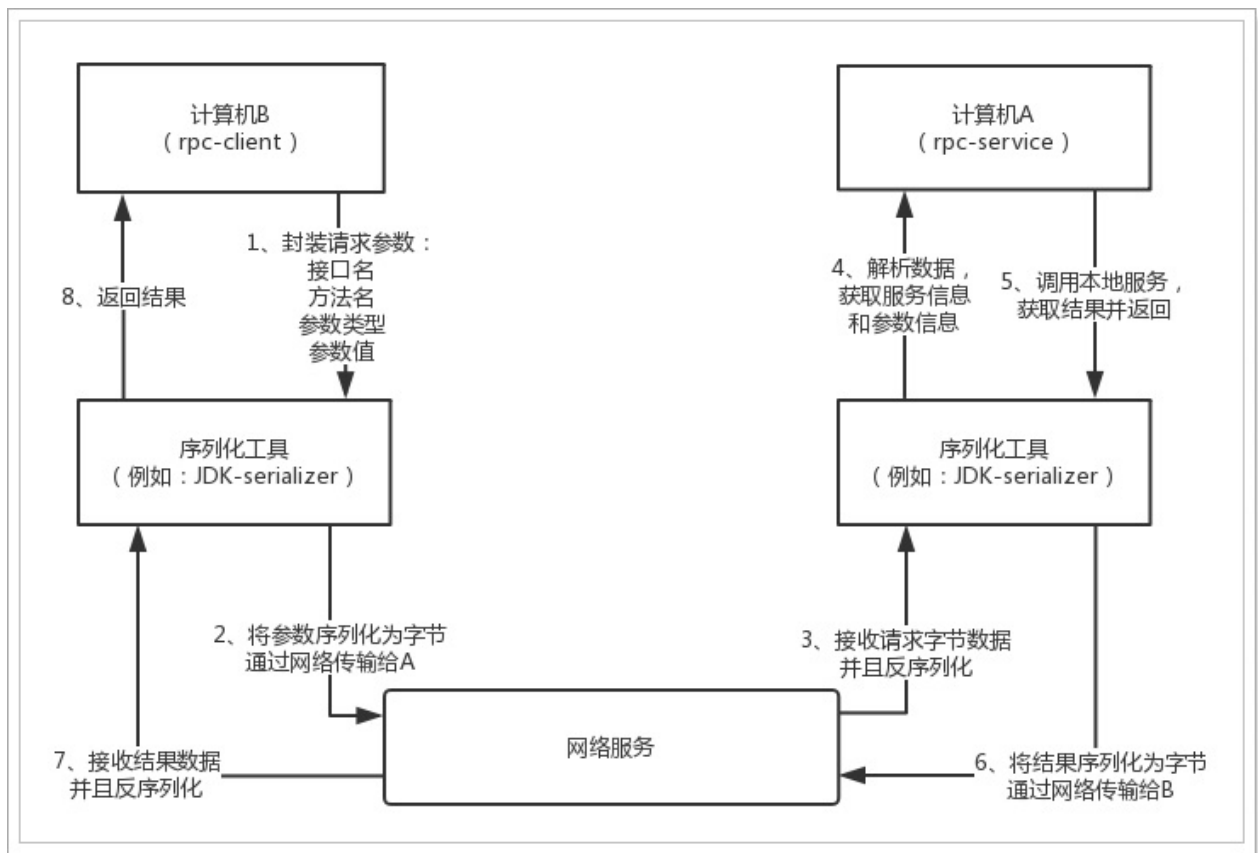
```
public interface HelloService {  
    String sayHello(String name);  
}
```

我们要在计算机B中远程调用计算机A的HelloService服务，该怎么做？

2.2、思路分析

在计算机B中，我们确切的知道要调用哪个类的哪个方法，而且知道具体的参数。但问题是：在计算机B中，只有接口，并没有方法的实现，无法直接调用。在计算机A中才有方法的具体实现。因此，现在就需要计算机B来调用计算机A中的方法，并且传参。也就是说：计算机B需要遥控指挥A做事情，把要执行的方法及参数告诉A即可。

如图所示：



整个调用过程分为以下几个步骤：

- 1) 计算机B封装请求的参数信息
- 2) 计算机B将请求参数信息序列化（网络传输可以接收的形式）
- 3) 计算机A接收请求并进行反序列化，得到请求参数
- 4) 计算机A解析请求参数，获取服务信息及参数信息
- 5) 计算机A调用本地服务，获取结果
- 6) 计算机A将执行的结果序列化
- 7) 计算机B接收数据，进行反序列化

这里有两个需要大家注意的地方：

- 序列化和反序列化的方式
 - 就序列化而言，Java 提供了默认的序列化方式，但在高并发的情况下，这种方式将会带来一些性能上的瓶颈，于是市面上出现了一系列优秀的序列化框架，比如：Protobuf、Kryo、Hessian、Jackson 等，它们可以取代 Java 默认的序列化，从而提供更高效的性能。不过在入门案例中，我们先采用Java的默认序列化方式。
- 网络传输的方式
 - 现在主流的RPC框架主要有两种网络传输方式：一种是Http协议，一种是TCP协议。事实上TCP才是底层的传输协议，Http是在TCP基础上进行了封装的应用层协议。大部分情况下，TCP协议的效率会更改。所以我们采用TCP方式传输数据。
 - 而TCP方式又有传统的 BIO（阻塞IO），性能更好的NIO（非阻塞IO）。当然我们可以选择非常热门的框架Netty来编写代码。不过因为有一定的学习成本，在入门案例中，我们将采用JDK原生的网络编程来实现。
- 请求参数和响应结果封装
 - 请求参数中需要包含的数据：
 - 服务的接口名
 - 接口中的方法名
 - 方法的参数类型（以防方法重载）
 - 方法的参数值
 - 响应结果要包含的数据：
 - 响应的状态（请求不一定会成功）
 - 异常信息
 - 结果数据

2.3、代码实现

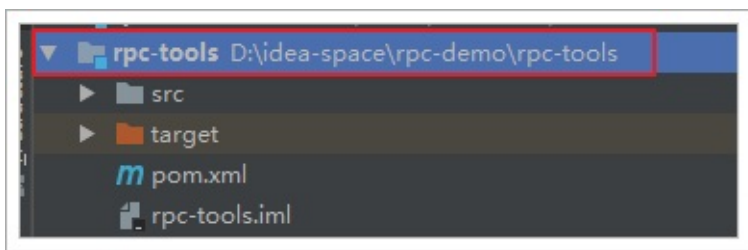
我们通过代码来实现刚才的思路：

传送门：

- 2.3.1、准备工作
- 2.3.2、请求参数RpcRequest
- 2.3.3、响应参数RpcResponse
- 2.3.4、服务提供方RpcServer
 - 2.3.4.1、服务端要做的事情
 - 2.3.4.2、服务注册器ServiceRegistry
 - 2.3.4.3、请求处理器RequestHandler
 - 2.3.4.4、服务端BioRpcServer
- 2.3.5、服务消费方
 - 2.3.5.1、客户端接口RpcClient
 - 2.3.5.2、客户端实现BioRpcClient
 - 2.3.5.3、动态代理工厂RpcProxyFactory
- 2.3.6、项目结构
- 2.3.7、测试

2.3.1、准备工作

为了便于以后的复用，我们创建一个新的maven工程，编写所有RPC的工具，以后使用时，就可以直接引入坐标即可。



列出我们需要编写的部分：

- 对外提供的服务（已完成，就是rpc-service中定义的 `HelloService`）
- 服务提供方（rpc-service，需要通过 `ServerSocket` 对外提供服务）
- 服务消费方（rpc-client，需要通过 `Socket` 连接rpc-service，实现远程访问）

- 需要注意的是，客户端只有服务接口，并没有实现类，所以我们需要利用动态代理的方式为这个接口生成一个实现类，然后在代理方法中，通过远程连接发起请求到达服务端，获取响应结果并返回。
- 序列化和反序列化：采用JDK默认的序列化
- 请求参数封装
- 响应参数封装

2.3.2、请求参数RpcRequest

注意，因为要使用JDK的序列化，因此该类需要实现 `Serializable` 接口：

```
/**
 * @author: HuYi.Zhang
 */
public class RpcRequest implements Serializable{

    private static final long serialVersionUID = 1L;
    private String className;
    private String methodName;
    private Class<?>[] parameterTypes;
    private Object[] parameters;

    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }

    public Class<?>[] getParameterTypes() {
        return parameterTypes;
    }
}
```

```
    }

    public void setParameterTypes(Class<?>[] parameterTypes) {
        this.parameterTypes = parameterTypes;
    }

    public Object[] getParameters() {
        return parameters;
    }

    public void setParameters(Object[] parameters) {
        this.parameters = parameters;
    }
}
```

2.3.3、响应参数RpcResponse

与RpcRequest一样，这里也需要实现Serializable接口：

另外为了使用方便，我们定义了几个静态方法，用来生成该类实例：

- `ok(Object data)` 表示响应成功，接收要返回的数据
- `error(String error)` 表示响应失败，接收错误信息
- `build(int stastus, String error, Object data)` 用来自定义返回状态和消息

```
/**
 * @author: HuYi.Zhang
 */
public class RpcResponse implements Serializable{

    private static final long serialVersionUID = 2L;
    private int status;// 响应状态 0失败，1成功
    private String error;// 错误信息
    private Object data;// 返回结果数据

    public static RpcResponse ok(Object data) {
        return build(1, null, data);
    }
}
```

```
public static RpcResponse error(String error) {
    return build(0, error, null);
}

public static RpcResponse build(int status, String error, Object data) {
    RpcResponse r = new RpcResponse();
    r.setStatus(status);
    r.setError(error);
    r.setData(data);
    return r;
}

public int getStatus() {
    return status;
}

public void setStatus(int status) {
    this.status = status;
}

public String getError() {
    return error;
}

public void setError(String error) {
    this.error = error;
}

public Object getData() {
    return data;
}

public void setData(Object data) {
    this.data = data;
}
}
```

2.3.4、服务提供方RpcServer

服务端的实现方式多种多样，可以用 BIO、NIO、AIO 来实现，为了便于后期扩展，这里先定义一个接口：

```
/**
 * RPC的服务端接口
 * @author HuYi.Zhang
 */
public interface RpcServer {
    /**
     * 启动服务
     */
    void start();

    /**
     * 停止服务
     */
    void stop();
}
```

2.3.4.1、服务端要做的事情

我们思考一下服务端要做的事情：

- 1) 启动服务：根据指定端口，启动一个 `ServerSocket`，等待客户端连接
- 2) 请求处理：接收客户端连接，接收请求（`RpcRequest`）并解析请求，得到要调用的接口信息
- 3) 服务发现：根据接口查找接口的具体实现
- 4) 本地执行：执行接口中的方法，获取响应结果（`RpcResponse`）
- 5) 返回结果

需要注意的是：

- 服务发现阶段需要从众多的类中，找到已知接口的实现，比较麻烦。为了方便查找，我们可以在一开始就将所有接口及其实现类关系缓存，实现简单的服务注册。
- 解析 `RpcRequest`，返回 `RpcRespon` 的流程，在以后其它的 `RpcServer` 实现类中也会用到，为了复用性，我们可以进行抽取。

2.3.4.2、服务注册器 `ServiceRegistry`

当请求到达，我们就需要根据请求的接口信息，找到对应的实现类。因此，我们最好提前把所有要对外提供的服务，提前记录在一个地方，方便以后寻找。

我们定义一个类ServiceRegistry，在服务启动前，先向其中注册服务。这里我们先手动注册服务，以后再考虑实现自动扫描并注册服务：

```
/**
 * @author: HuYi.Zhang
 */
public class ServiceRegistry{

    private static final Logger logger = LoggerFactory.getLogger(
ServiceRegistry.class);

    private static final Map<String, Object> registeredServices =
new HashMap<>();

    public static <T> T getService(String className) {
        return (T) registeredServices.get(className);
    }

    public static void registerService(Class<?> interfaceClass,
Class<?> implClass) {
        try {
            registeredServices.put(interfaceClass.getName(), imp
lClass.newInstance());
            logger.info("服务注册成功, 接口: {}, 实现{}", interfaceCla
ss.getName(), implClass.getName());
        } catch (Exception e) {
            e.printStackTrace();
            logger.error("服务" + implClass + "注册失败", e);
        }
    }
}
```

2.3.4.3、请求处理器RequestHandler

处理请求，获取响应的过程在 RpcServer 的各个实现类中都可能用到，所以我们进行抽取：


```
/**
 * @author: HuYi.Zhang
 **/
public class RequestHandler {

    private static final Logger logger = LoggerFactory.getLogger(
RequestHandler.class);

    public static RpcResponse handleRequest(RpcRequest request) {

        try {
            // 获取服务
            Object service = ServiceRegistry.getService(request.
getClassName());
            if (service != null) {
                Class<?> clazz = service.getClass();
                // 获取方法
                Method method = clazz.getMethod(request.getMetho
dName(),
                    request.getParameterTypes());
                // 执行方法
                Object result = method.invoke(service, request.g
etParameters());
                // 写回结果
                return RpcResponse.ok(result);
            } else {
                logger.error("请求的服务未找到:{}.{}({})",
                    request.getClassName(),
                    request.getMethodName(),
                    StringUtils.join(request.getParameterTyp
es(), ", "));
                return RpcResponse.error("未知服务!");
            }
        } catch (Exception e) {
            e.printStackTrace();
            logger.error("处理请求失败", e);
            return RpcResponse.error(e.getMessage());
        }
    }
}
```

2.3.4.4、服务端BioRpcServer

入门案例中，我们先通过BIO方式实现 `RpcServer` 类：

```
/**
 * BIO的RPC服务端
 *
 * @author HuYi.Zhang
 */
public class BioRpcServer implements RpcServer{

    private static final Logger logger = LoggerFactory.getLogger(
        BioRpcServer.class);
    // 用来处理请求的连接池
    private static final ExecutorService es = Executors.newCachedThreadPool();

    private int port = 9000; // 默认端口

    private volatile boolean shutdown = false; // 是否停止

    /**
     * 使用默认端口9000，构建一个BIO的RPC服务端
     */
    public BioRpcServer() {
    }

    /**
     * 使用指定端口构建一个BIO的RPC服务端
     *
     * @param port 服务端端口
     */
    public BioRpcServer(int port) {
        this.port = port;
    }

    @Override
    public void start() {
        try {
            // 启动服务
            ServerSocket server = new ServerSocket(this.port);
            logger.info("服务启动成功，端口：{}", this.port);
        } catch (IOException e) {
            logger.error("服务启动失败，端口：{}", this.port, e);
        }
    }
}
```

```

        while (!this.shutdown) {
            // 接收客户端请求
            Socket client = server.accept();
            es.execute(() -> {
                try (
                    // 使用JDK的序列化流
                    ObjectInputStream in = new ObjectInputP
utStream(client.getInputStream());
                    ObjectOutputStream out = new ObjectO
utputStream(client.getOutputStream())
                ) {
                    // 读取请求参数
                    RpcRequest request = (RpcRequest) in.rea
dObject();

                    logger.info("接收请求，{}.{}({})",
                                request.getClassName(),
                                request.getMethodName(),
                                StringUtils.join(request.getPara
meterTypes(), ", "));

                    logger.info("请求参数：{}",
                                StringUtils.join(request.getPara
meters(), ", "));

                    // 处理请求
                    out.writeObject(RequestHandler.handleReq
uest(request));
                } catch (Exception e) {
                    logger.error("客户端连接异常，客户端{}:{}",
client.getInetAddress().toString());
                    throw new RuntimeException(e);
                }
            });
        }
    } catch (IOException e) {
        e.printStackTrace();
        logger.error("服务启动失败", e);
    }
}

@Override
public void stop() {
    this.shutdown = true;
    logger.info("服务即将停止");
}

```

```
    }  
}
```

2.3.5、服务消费方

2.3.5.1、客户端接口RpcClient

同 `RpcServer` 一样，我们先定义一个接口：

客户端做的事情比较单一，发起 `RpcRequest` 请求，获取响应并解析即可。

```
/**  
 * RPC客户端  
 * @author HuYi.Zhang  
 */  
public interface RpcClient {  
  
    /**  
     * 发起请求，获取响应  
     * @param request  
     * @return  
     */  
    RpcResponse sendRequest(RpcRequest request) throws Exception;  
  
}
```

2.3.5.2、客户端实现BioRpcClient

然后实现一个BIO的客户端：

```
/**  
 * RPC客户端的BIO实现  
 *  
 * @author HuYi.Zhang  
 */  
public class BioRpcClient implements RpcClient {  
  
    private static final Logger logger = LoggerFactory.getLogger(  

```

```
BioRpcClient.class);

    private String host;

    private int port;

    public BioRpcClient(String host, int port) throws IOException {
        this.host = host;
        this.port = port;
    }

    @Override
    public RpcResponse sendRequest(RpcRequest request) throws Exception {
        try {
            Socket socket = new Socket(host, port);
            ObjectOutputStream out = new ObjectOutputStream(
                socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(
                socket.getInputStream())
        } {
            logger.info("建立连接成功: {}: {}", host, port);
            // 发起请求
            out.writeObject(request);
            logger.info("发起请求, 目标主机 {}: {}, 服务: {}. {}({})", host, port,
                request.getClassName(), request.getMethodName(),
                StringUtils.join(request.getParameterTypes(),
                    ","));
            // 获取结果
            return (RpcResponse) in.readObject();
        }
    }
}
```

2.3.5.3、动态代理工厂 RpcProxyFactory

刚刚实现的 `RpcClient` 中，我们实现了发起请求，获取响应的功能。那么问题来了：

谁来发起请求？

谁来解析响应？

根据我们前面的分析，客户端（计算机B）只有 `HelloService` 接口，并没有具体的实现。我们需要通过动态代理为 `HelloService` 生成实现类。当有人调用 `HelloService` 的 `sayHello()` 方法时，底层可以调用 `RpcClient` 的 `sendRequest()` 功能向服务端（计算机A）发起请求，获取执行结果。

不管接口是什么，生成动态代理的代码和逻辑几乎是一样的。因此我们可以抽取出一个生成代理的工厂：

```
/**
 * 一个动态代理工厂，为接口生成实现了Rpc远程调用的实现类。
 * @author: HuYi.Zhang
 */
public class RpcProxyFactory<T> implements InvocationHandler {

    private static final Logger logger = LoggerFactory.getLogger(
RpcProxyFactory.class);
    private Class<T> clazz;

    public RpcProxyFactory(Class<T> clazz) {
        this.clazz = clazz;
    }

    public T getProxyObject() {
        return (T) Proxy.newProxyInstance(clazz.getClassLoader(),
new Class[]{clazz}, this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) {
        // 处理Object中的方法
        if (Object.class == method.getDeclaringClass()) {
            String name = method.getName();
```

```

        if ("equals".equals(name)) {
            return proxy == args[0];
        } else if ("hashCode".equals(name)) {
            return System.identityHashCode(proxy);
        } else if ("toString".equals(name)) {
            return proxy.getClass().getName() + "@" +
                Integer.toHexString(System.identityHashC
ode(proxy)) +
                ", with InvocationHandler " + this;
        } else {
            throw new IllegalStateException(String.valueOf(m
ethod));
        }
    }
    // 封装请求参数
    RpcRequest request = new RpcRequest();
    request.setClassName(clazz.getName());
    request.setMethodName(method.getName());
    request.setParameters(args);
    request.setParameterTypes(method.getParameterTypes());
    try {
        // 发起网络请求, 并接收响应
        RpcClient client = new BioRpcClient("127.0.0.1", 9000
);

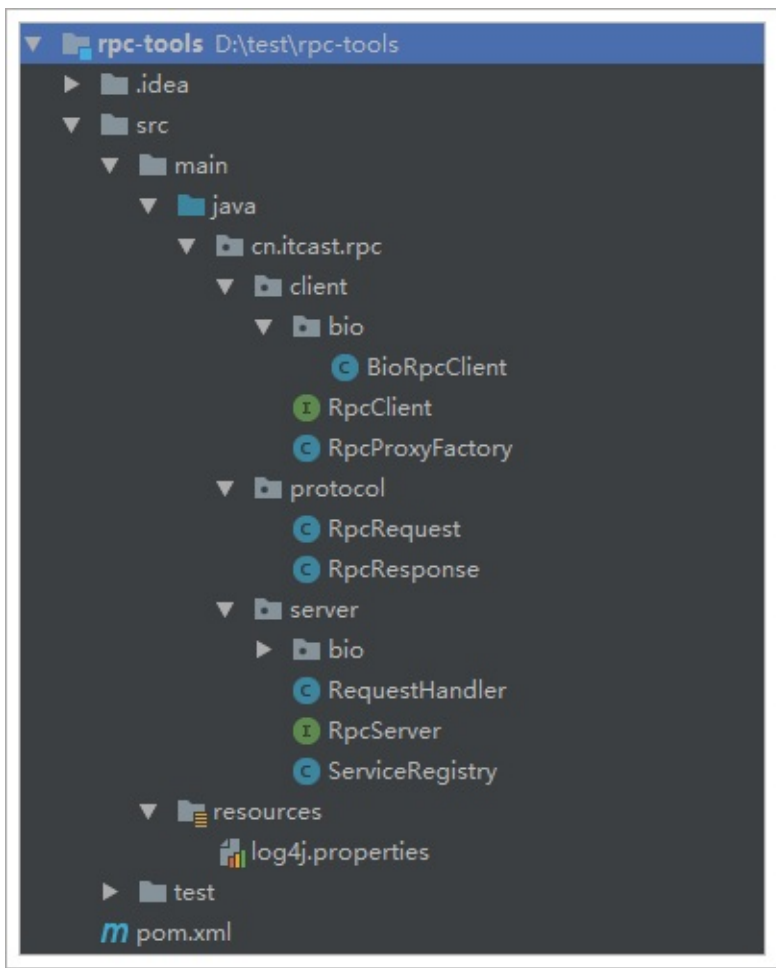
        RpcResponse response = client.sendRequest(request);
        // 解析并返回
        if (response.getStatus() == 1) {
            logger.info("调用远程服务成功!");
            return response.getData();
        }
        logger.debug("远程服务调用失败, {}。", response.getError
());

        return null;
    } catch (Exception e) {
        logger.error("远程调用异常", e);
        throw new RuntimeException(e);
    }
}

```

2.3.6、项目结构

如图：



2.3.7、测试

在rpc-service中编写测试类，启动一个服务端：

```
public class RpcServerTest {  
    @Test  
    public void test01() throws InterruptedException {  
        // 注册服务  
        ServiceRegistry.registerService>HelloService.class, HelloServiceImpl.class);  
        // 启动服务  
        new BioRpcServer(9000).start();  
    }  
}
```


运行日志：

```
2018-03-16 16:14:31 INFO BioRpcServer:69 - 注册服务cn.itcast.rpc.service.HelloService, 实现类cn.itcast.rpc.service.impl.HelloServiceImpl
2018-03-16 16:14:31 INFO BioRpcServer:78 - 服务启动成功，端口：9000
```

在rpc-client中启动一个客户端：


```
public class RpcClientTest {
    @Test
    public void test01(){
        // 通过代理工厂，获取服务
        HelloService helloService = new RpcProxyFactory<>(HelloService.class).getProxyObject();
        // 调用服务
        String result = helloService.sayHello("Jack");
        System.out.println(result);
        Assert.assertEquals("调用失败", "hello, Jack", result);
    }
}
```

运行后，服务端日志：

```
2018-03-16 16:14:37 INFO BioRpcServer:113 - 接收请求，cn.itcast.rpc.service.HelloService.sayHello(class java.lang.String)
2018-03-16 16:14:37 INFO BioRpcServer:117 - 请求参数：Jack
```

客户端日志：

```
2018-03-18 14:23:39 INFO BioRpcClient:44 - 建立连接成功：127.0.0.1 : 9000
2018-03-18 14:23:39 INFO BioRpcClient:47 - 发起请求, 目标主机127.0.0.1:9000, 服务：cn.itcast.rpc.service.HelloService.sayHello(class java.lang.String)
2018-03-18 14:23:39 INFO RpcProxyFactory:61 - 调用远程服务成功！
hello, Jack
```



到这里为止，一个简单的RPC框架就实现了！

3、优化RPC框架

在刚才实现的RPC框架中，其实隐藏着许多问题需要去解决：

- 服务的注册和发现需要手动完成
- 序列化采用的是JDK的默认方式，效率较低
- 网络模型采用的是BIO，而且是短连接，效率很差

接下来，我们就一一解决这些问题，对框架进行升级

3.1、与spring整合

Spring几乎是现在开发JavaEE的必备框架，特别是在SpringBoot出现以后，其快速搭建项目的功能一直被人们津津乐道。当然Spring的核心功能AOP和依赖注入功能，也非常强大。我们接下来就利用Spring的依赖注入功能，将服务通过注解直接扫描并注册到Spring容器，并且可以通过依赖注入功能实现自动注入。

快速通道：

- 3.1.1、服务的自动注册
 - 3.1.1.1、将接口的实现类注册到Spring容器
 - 3.1.1.2、将接口和实现类注册到ServiceRegistry
- 3.1.2、服务端RpcServer与Spring整合
- 3.1.3、服务端测试
- 3.1.4、客户端实现服务的依赖注入
 - 3.1.4.1、需求
 - 3.1.4.2、@Autowired的原理
 - 3.1.4.3、实现自动注入
- 3.1.5、客户端测试

3.1.1、服务的自动注册

在刚才的案例中，我们要在服务端注册一个服务，需要通过下面的方式手动注册：

```
// 注册服务
ServiceRegistry.registerService(HelloService.class, HelloServiceImpl.class);
```

这种方式兼职弱爆了。我们回忆一下Spring的功能，在spring中提供了以下一些注解：

- @Component
- @Service
- @Controller
- @Repository

如果想要一个Bean加入Spring容器，只需要使用上面任意一个注解即可。我们能不能通过类似的方式来实现将服务自动注册到Spring，并且注册到 `ServiceRegistry` 的功能呢？

先来看看我们要达到的目标：

- 1) 将接口的实现类注册到Spring容器
- 2) 将接口和实现类信息保存到ServiceRegistry中，方便以后查找

好了，接下来我们分别实现这两个目标

3.1.1.1、将接口的实现类注册到Spring容器

相信大家很快就能想到：我们直接在实现类上加上前面提到过的Spring提供的任意一个注解就可以实现了。

没错，这样确实能达到目的。但是大家思考一下，如果我们使用Spring提供的注解，那么我们将来如何能知道Spring容器中的哪些类是需要注册到 `ServiceRegistry` 的呢？

所以，我们不能使用Spring的注解，这样就产生了新的问题：

- 如果我们不使用Spring的注解，Spring就不会主动把类注册到Spring容器了。

这个问题其实很好解决，大家看一下spring的 `@Service` 或者 `@Controller` 源码就明白了：

```
/**
 * Indicates that an annotated class is a "Controller" (e.g. a web
 * controller).
 *
 * <p>This annotation serves as a specialization of {@link Component},
 * allowing for implementation classes to be autodetected through
 * classpath scanning.
 * It is typically used in combination with annotated handler methods
 * based on the
 * {@link org.springframework.web.bind.annotation.RequestMapping}
 * annotation.
 *
 * @author Arjen Poutsma
 * @author Juergen Hoeller
 * @since 2.5
 * @see Component
 * @see org.springframework.web.bind.annotation.RequestMapping
 * @see org.springframework.context.annotation.ClassPathBeanDefinitionScanner
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
    String value() default "";

}
```

我们可以发现，在 `@Controller` 注解上，其实有一个 `@Component` 注解，然后类上的一段注释：

```
This annotation serves as a specialization of @Component,allowing for implementation classes to be autodetected through classpath scanning.
```

此类作为`@Component`注解的一个特例，运行通过类路径自动扫描获取实现类。

也就是说，一个自定义注解，只要加上了 `@Component` 注解，就会起到跟 `@Component` 一样的作用，被标记的类会被Spring自动扫描，并加入spring容器中。

所以，我们定义一个自定义注解，用来识别需要对外提供的服务：

```
/**
 * 用来标记RPC服务，并且声明其接口
 * @author: HuYi.Zhang
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface Service {

    /**
     * 接口名称
     * @return
     */
    Class<?> value();
}
```

这个注解需要接收value属性，用来指定被标记的类所实现的接口。

接下来，我们就可以在 `HelloServiceImpl` 上使用这个自定义的注解了：

```
@Service(HelloService.class)
public class HelloServiceImpl implements HelloService{
    public String sayHello(String name) {
        return "hello, " + name;
    }
}
```

3.1.1.2、将接口和实现类注册到ServiceRegistry

我们已经将 `HelloServiceImpl` 注册到Spring容器了。下一步动作，就是将该实现类及其接口`HelloService`信息注册到 `ServiceRegistry` 中。

这一步动作，我们也希望由Spring来帮我们完成，怎么做呢？

实现的方式有很多种，我们这里介绍其中一种，使用 `ApplicationContextAware` 接口。

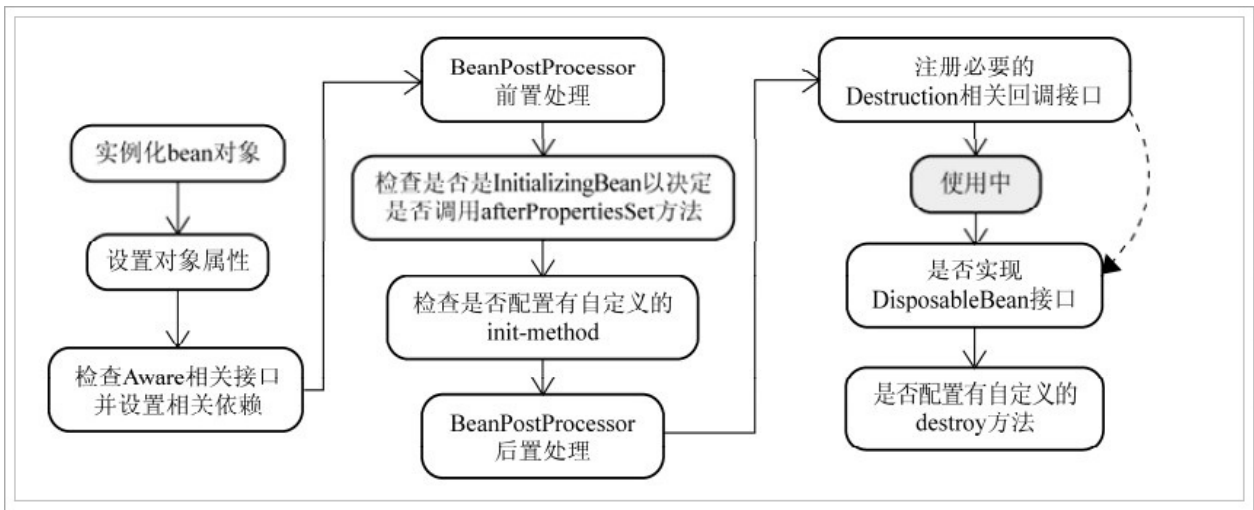

```
public interface ApplicationContextAware extends Aware {

    /**
     * Set the ApplicationContext that this object runs in.
     * Normally this call will be used to initialize the object.
     * <p>Invoked after population of normal bean properties but
     before an init callback such
     * as {@link org.springframework.beans.factory.InitializingBe
     an#afterPropertiesSet()}
     * or a custom init-method. Invoked after {@link ResourceLoad
     erAware#setResourceLoader},
     * {@link ApplicationEventPublisherAware#setApplicationEventP
     ublisher} and
     * {@link MessageSourceAware}, if applicable.
     * @param applicationContext the ApplicationContext object to
     be used by this object
     * @throws ApplicationContextException in case of context ini
     tialization errors
     * @throws BeansException if thrown by application context me
     thods
     * @see org.springframework.beans.factory.BeanInitializationE
     xception
     */
    void setApplicationContext(ApplicationContext applicationCont
    ext) throws BeansException;

}
```

这个东西是干什么的呢？

我们先看一下Spring初始化Bean的流程：



当Spring对所有Bean进行实例化后，会完成对属性的设置。然后会检查有没有类实现了与Aware相关的接口，并且处理，我们的 `ApplicationContextAware` 就是其中之一。

在这个接口上有这么一段注释：

```

/**
 * Interface to be implemented by any object that wishes to be notified
 * of the {@link ApplicationContext} that it runs in.
 *
 * <p>Implementing this interface makes sense for example when an object
 * requires access to a set of collaborating beans. Note that configuration
 * via bean references is preferable to implementing this interface just
 * for bean lookup purposes.
 *
 * <p>This interface can also be implemented if an object needs access to file
 * resources, i.e. wants to call {@code getResource}, wants to publish
 * an application event, or requires access to the MessageSource. However,
 * it is preferable to implement the more specific {@link ResourceLoaderAware},
 * {@link ApplicationEventPublisherAware} or {@link MessageSourceAware} interface
 * in such a specific scenario.
 *
 * <p>Note that file resource dependencies can also be exposed as bean properties
 * of type {@link org.springframework.core.io.Resource}, populated via Strings
 * with automatic type conversion by the bean factory. This removes the need
 * for implementing any callback interface just for the purpose of accessing
 * a specific file resource.
 */

```

翻译一下：

当一个对象需要访问容器一些bean时，可以实现这个接口，否则就没有必要。请注意，通过Bean引用进行配置优于为了bean查找目的而实现此接口。

当一个类实现 `ApplicationContextAware` 接口时，spring扫描到以后，就会调用 `setApplicationContext` 方法，将spring容器传递给这个方法中。拿到了容器，我们就可以从中寻找带有我们自定义注解 `@Service` 的类了。

我们改造ServiceRegistry，让它实现 ApplicationContextAware 接口：

```
/**
 * @author: HuYi.Zhang
 */
public class ServiceRegistry implements ApplicationContextAware {

    private static final Logger logger = LoggerFactory.getLogger(
        ServiceRegistry.class);

    private static final Map<String, Object> registeredServices =
        new HashMap<>();

    public static <T> T getService(String className) {
        return (T) registeredServices.get(className);
    }

    public static void registerService(Class<?> interfaceClass,
        Class<?> implClass) {
        try {
            registeredServices.put(interfaceClass.getName(), implClass.newInstance());
            logger.info("服务注册成功, 接口: {}, 实现{}", interfaceClass.getName(), implClass.getName());
        } catch (Exception e) {
            e.printStackTrace();
            logger.error("服务" + implClass + "注册失败", e);
        }
    }

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException {
        Map<String, Object> services = ctx.getBeansWithAnnotation(Service.class);
        if (services != null && services.size() > 0) {
            for (Object service : services.values()) {
                String interfaceName = service.getClass().getAnnotation(Service.class).value().getName();
                registeredServices.put(interfaceName, service);
            }
        }
    }
}
```

```
        logger.info("加载服务：{}", interfaceName);
    }
}
}
```

这样以来，我们就不需要手动注册服务了！

3.1.2、服务端RpcServer与Spring整合

要把RpcServer与Spring整合，就需要Bean能够在初始化完成后启动，我们可以给BioRpcServer添加一个初始化方法 `init()`。还需要添加一个注解 `@PostConstructor`，这样Spring在初始化完成后，会自动调用该方法。另外，可以给`stop`方法添加 `@PreDestroy` 注解，这样Spring会在Bean销毁前调用该方法，将服务停止。

```
@Override
@PreDestroy
public void stop() {
    this.shutdown = true;
    logger.info("服务即将停止");
}

@PostConstruct
public void init(){
    es.submit(this::start);
}
```

注意：因为 `start()` 方法是阻塞的，所以不能在 `init()` 方法中直接调用 `start()`，会导致Spring线程阻塞，所以我们在init中开启线程来异步执行 `start()` 方法；

3.1.3、服务端测试

首先编写配置类，将注册器ServiceRegistry及服务端BioRpcServer注册到Spring：

这里我们采用Java配置方式，千万不要忘了指定扫描包：

```
/**
 * @author: HuYi.Zhang
 **/
@Configuration
@ComponentScan(basePackages = "cn.itcast.rpc.service")
public class RpcServerConfig {

    /**
     * BIO的RPC服务端
     * @return
     */
    @Bean
    public RpcServer rpcServer() {
        return new BioRpcServer();
    }

    /**
     * 服务的自动注册器
     * @return
     */
    @Bean
    public ServiceRegistry serviceRegistry(){
        return new ServiceRegistry();
    }
}
```

编写测试类，我们并不需要手动注册任何服务：

```
/**
 * @author: HuYi.Zhang
 **/
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = RpcServerConfig.class)
public class RpcServerTestWithSpring {

    @Test
    public void test01() throws InterruptedException {
        // spring会自动注册服务，只要保证容器存活即可
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

启动并查看日志：

```
2018-03-18 17:31:45 INFO DefaultTestContextBootstrapper:248 - L
loaded default TestExecutionListener class names from location [M
ETA-INF/spring.factories]: [org.springframework.test.context.web.
ServletTestExecutionListener, org.springframework.test.context.s
upport.DirtiesContextBeforeModesTestExecutionListener, org.sprin
gframework.test.context.support.DependencyInjectionTestExecution
Listener, org.springframework.test.context.support.DirtiesContex
tTestExecutionListener, org.springframework.test.context.transac
tion.TransactionalTestExecutionListener, org.springframework.test
.context.jdbc.SqlScriptsTestExecutionListener]
```

```
2018-03-18 17:31:45 INFO DefaultTestContextBootstrapper:174 - U
sing TestExecutionListeners: [org.springframework.test.context.s
upport.DirtiesContextBeforeModesTestExecutionListener@39fb3ab6,
org.springframework.test.context.support.DependencyInjectionTest
ExecutionListener@6276ae34, org.springframework.test.context.sup
port.DirtiesContextTestExecutionListener@7946e1f4]2018-03-18 17:
31:45 INFO GenericApplicationContext:583 - Refreshing org.sprin
gframework.context.support.GenericApplicationContext@1e88b3c: st
artup date [Sun Mar 18 17:31:45 CST 2018]; root of context hiera
rchy
```

```
2018-03-18 17:31:46 INFO ServiceRegistry:44 - 加载服务:cn.itcast.
rpc.service.HelloService
```

```
2018-03-18 17:31:46 INFO BioRpcServer:60 - 服务启动成功，端口：9000
```

3.1.4、客户端实现服务的依赖注入

3.1.4.1、需求

在前面的案例中，客户端要想获取服务，需要手动创建 `RpcProxyFactory` 实例，并从中获取服务的代理：

```
// 通过代理工厂，获取服务
HelloService helloService = new RpcProxyFactory<>(HelloService.c
lass).getProxyObject();
```

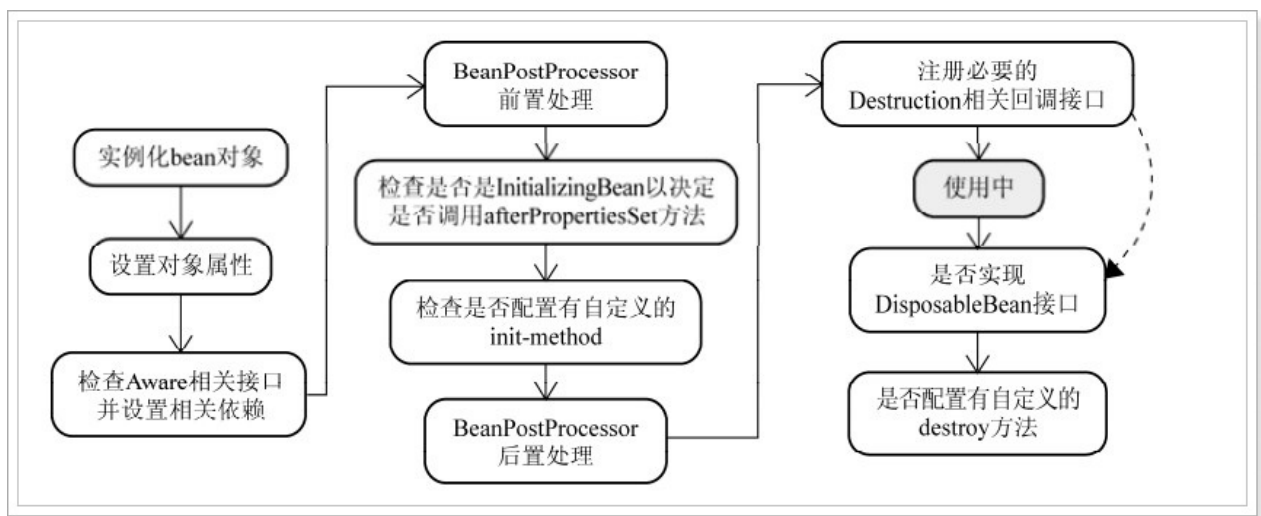
这样太麻烦了，能不能像Spring中那样，通过 `@Autowired` 实现自动的依赖注入呢？

要想通过 `@Autowired` 自动注入，就必须在Spring容器中有对应的实例对象。然而在客户端只有接口，并没有实现类，更不会有对象，所以无法通过 `@Autowired` 自动注入。

我们能不能模拟 `@Autowired` 功能，自己来实现依赖注入呢？

3.1.4.2、@Autowired的原理

先来看一下Spring的Bean初始化流程：



执行流程：

- 1) 所有的Bean实例化
- 2) 对Bean的属性进行初始化
- 3) 检查Aware相关接口
- 4) 执行BeanPostProcessor的前置方法
- 5) 处理实现了InitializingBean的类
- 6) 处理添加了自定义init-method的类
- 7) BeanPostProcessor的后置方法
- ...

我们需要注意其中的BeanPostProcessor这个东西：

BeanPostProcessor 接口中定义了两个方法：

```
/**
```



```

    * Factory hook that allows for custom modification of new bean
    instances,
    * e.g. checking for marker interfaces or wrapping them with pro
    xies.
    * ...
    * 运行客户对 bean 进行自定义修改的 Bean工厂钩子（hook），Spring通过该接
    口作为标记进行检查
    *
    * <p>ApplicationContexts can autodetect BeanPostProcessor beans
    in their
    * bean definitions and apply them to any beans subsequently cre
    ated.
    * ...
    * ApplicationContexts 可以自动检测到实现该接口的Bean，并且在任何其他Be
    an创建之后执行它。
    * 略...
    */
public interface BeanPostProcessor {

    /**
     * Apply this BeanPostProcessor to the given new bean instan
     ce <i>before</i> any bean
     * initialization callbacks (like InitializingBean's {@code
     afterPropertiesSet}
     * or a custom init-method).
     * 前置方法：在给定的这个bean的初始化方法（afterPropertiesSet或ini
     t-method）执行之前执行。
     * 略...
     */
    Object postProcessBeforeInitialization(Object bean, String b
    eanName) throws BeansException;

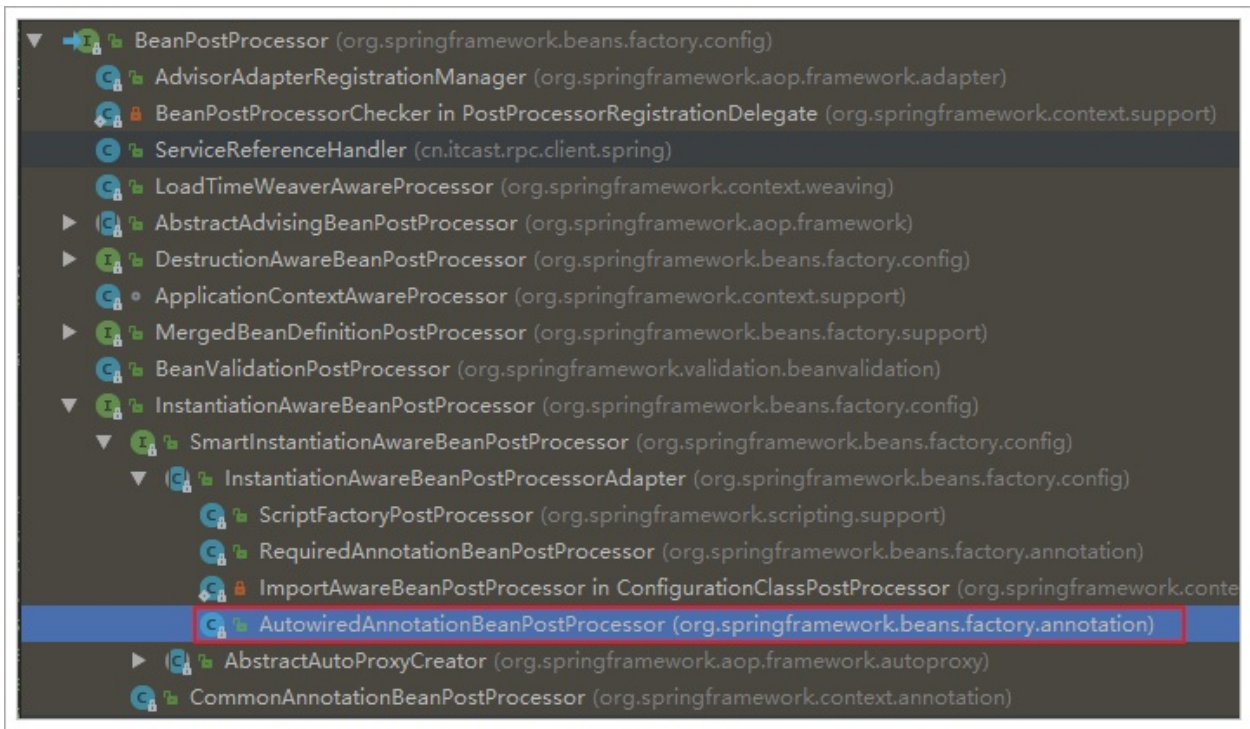
    /**
     * Apply this BeanPostProcessor to the given new bean instan
     ce <i>after</i> any bean
     * initialization callbacks (like InitializingBean's {@code
     afterPropertiesSet}
     * or a custom init-method).
     * 后置方法：在给定的这个bean的初始化方法（afterPropertiesSet或ini
     t-method）执行之后执行。
     * 略...
     */

```

```
Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;

}
```

再来了解一下 `@Autowired` 的实现原理，Spring正是通过一个`BeanPostProcessor`来实现 `@Autowired` 自动注入的：



我们看类上的一段说明：

`org.springframework.beans.factory.config.BeanPostProcessor` implementation that autowires annotated fields, setter methods and arbitrary config methods. Such members to be injected are detected through a Java 5 annotation: by default, Spring's `@Autowired` and `@Value` annotations.

`org.springframework.beans.factory.config.BeanPostProcessor` 的一个实现，用来自动注入带有`Autowired`注解的字段、setter方法和任意配置方法。默认情况下，这些成员可以通过Java 5注解检测到，包括Spring的`@Autowired`和`@Value`注释。

总结：

Spring在实例化完成所有Bean以后，会检查Bean上是否实现了 `BeanPostProcessor` 接口，如果有就会在任何普通Bean初始化时，先调用该接口的前置方法：`postProcessBeforeInitialization()`，然后进行普通Bean的初始化。然后再调用后置方法：`postProcessAfterInitialization()`。

`@Autowired` 注解正是利用了这个接口的特性，编写了一个 `AutowiredAnnotationBeanPostProcessor`，然后在其中对每一个Bean的成员进行检测，如果发现实现了 `@Autowired` 注解或者 `@Value` 注解（也支持JSR-330的注解），就会对其进行注入。

3.1.4.3、实现自动注入

思路：

我们也可以通过自定义注解的方式，来标记这些需要注入的属性。然后实现一个 `BeanPostProcessor`，在普通Bean初始化的时候进行拦截。如果发现我们的自定义标记，则通过 `RpcProxFactory` 来生成代理并且注入。

首先，编写一个自定义注解，用来标记需要注入的成员：

```
/**
 * 标记一个需要通过RPC注入的资源
 * @author: HuYi.Zhang
 * @create: 2018-03-16 22:04
 */
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Reference {
}
```

然后，再编写一个 `BeanPostProcessor`：

```
/**
 * Bean的后处理器，用来注入Rpc的动态代理对象
 * @author: HuYi.Zhang
 */
public class RpcProxyBeanPostProcessor implements BeanPostProcessor {

    private static final Logger logger = LoggerFactory.getLogger(
```

```

RpcProxyBeanPostProcessor.class);

    private final Map<Class<?>, Object> cache = new HashMap<>();

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        // 遍历所有字段
        for (Field f : bean.getClass().getDeclaredFields()) {
            // 判断是否有@Reference注解
            if (f.isAnnotationPresent(Reference.class)) {
                f.setAccessible(true);
                Class<?> clazz = f.getType();
                Object proxy = null;
                // 判断该字段类型在缓存中是否存在
                if (cache.containsKey(clazz)) {
                    proxy = cache.get(clazz);
                } else {
                    // 动态代理生成对象
                    proxy = new RpcProxyFactory<>(clazz).getProxyObject();

                    cache.put(bean.getClass(), proxy);
                }
                try {
                    f.set(bean, proxy);
                    logger.info("为{}注入{}。", f, proxy);
                } catch (Exception e) {
                    e.printStackTrace();
                    logger.error("属性" + f + "注入失败", e);
                }
            }
        }
        return bean;
    }
}

```

3.1.5、客户端测试

首先编写Spring的配置类，只需要把自定义的 `BeanPostProcessor` 注册就可以了：

```
@Configuration
public class RpcClientConfig {

    /**
     * 处理@Reference注解标记的属性自动注入
     * @return
     */
    @Bean
    public RpcProxyBeanPostProcessor serviceReferenceHandler() {
        return new RpcProxyBeanPostProcessor();
    }

}
```

测试类：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = RpcClientConfig.class)
public class RpcClientTestWithSpring {

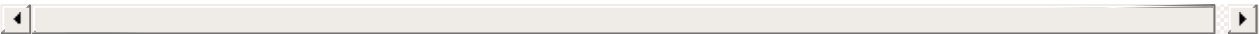
    @Reference
    private HelloService helloService;// 自动注入，无需手动获取

    @Test
    public void test01() {
        String result = this.helloService.sayHello("Jack");
        System.out.println(result);
        Assert.assertEquals("调用失败", "hello, Jack", result);
    }

}
```

启动并查看日志：

```
2018-03-18 17:38:34 INFO DefaultTestContextBootstrapper:248 - L
loaded default TestExecutionListener class names from location [M
ETA-INF/spring.factories]: [org.springframework.test.context.web.
ServletTestExecutionListener, org.springframework.test.context.s
upport.DirtiesContextBeforeModesTestExecutionListener, org.sprin
gframework.test.context.support.DependencyInjectionTestExecution
Listener, org.springframework.test.context.support.DirtiesContex
tTestExecutionListener, org.springframework.test.context.transac
tion.TransactionalTestExecutionListener, org.springframework.test
.context.jdbc.SqlScriptsTestExecutionListener]
2018-03-18 17:38:34 INFO DefaultTestContextBootstrapper:174 - U
sing TestExecutionListeners: [org.springframework.test.context.s
upport.DirtiesContextBeforeModesTestExecutionListener@5cc7c2a6,
org.springframework.test.context.support.DependencyInjectionTest
ExecutionListener@b97c004, org.springframework.test.context.supp
ort.DirtiesContextTestExecutionListener@4590c9c3]2018-03-18 17:38
:34 INFO GenericApplicationContext:583 - Refreshing org.springf
ramework.context.support.GenericApplicationContext@5e3a8624: sta
rtup date [Sun Mar 18 17:38:34 CST 2018]; root of context hierar
chy
2018-03-18 17:38:34 INFO PostProcessorRegistrationDelegate$Bean
PostProcessorChecker:327 - Bean 'rpcClientConfig' of type [cn.it
cast.rpc.config.RpcClientConfig$EnhancerBySpringCGLIB$$c5d66a2a]
is not eligible for getting processed by all BeanPostProcessors
(for example: not eligible for auto-proxying)
2018-03-18 17:38:34 INFO RpcProxyBeanPostProcessor:47 - 为priva
te cn.itcast.rpc.service.HelloService cn.itcast.rpc.RpcClientTes
tWithSpring.helloService注入com.sun.proxy.$Proxy17@e056f20, with
InvocationHandler cn.itcast.rpc.client.RpcProxyFactory@4b0b0854
。
2018-03-18 17:38:34 INFO BioRpcClient:44 - 建立连接成功:127.0.0.1
:9000
2018-03-18 17:38:34 INFO BioRpcClient:47 - 发起请求,目标主机127.0.
0.1:9000,服务:cn.itcast.rpc.service.HelloService.sayHello(class
java.lang.String)
2018-03-18 17:38:34 INFO RpcProxyFactory:61 - 调用远程服务成功!
hello, Jack
2018-03-18 17:38:34 INFO GenericApplicationContext:984 - Closin
g org.springframework.context.support.GenericApplicationContext@
5e3a8624: startup date [Sun Mar 18 17:38:34 CST 2018]; root of c
ontext hierarchy
```



3.2、利用**zookeeper**实现服务注册中心

to be continue ...

待续 ...

3.3、使用Netty提供的NIO网络模型

to be continue ...

待续 ...