轻量级分布式 RPC 框架

RPC，即 Remote Procedure Call（远程过程调用），说得通俗一点就是：调用远程计算机上的服务，就像调用本地服务一样。

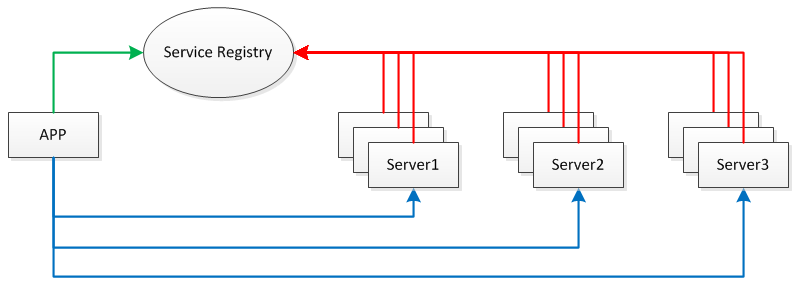
RPC 可基于 HTTP 或 TCP 协议，Web Service 就是基于 HTTP 协议的 RPC，它具有良好的跨平台性，但其性能却不如基于 TCP 协议的 RPC。会两方面会直接影响 RPC 的性能，一是传输方式，二是序列化。

众所周知，TCP 是传输层协议，HTTP 是应用层协议，而传输层较应用层更加底层，在数据传输方面，越底层越快，因此，在一般情况下，TCP 一定比 HTTP 快。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。

为了支持高并发，传统的阻塞式 IO 显然不太合适，因此我们需要异步的 IO，即 NIO。Java 提供了 NIO 的解决方案，Java 7 也提供了更优秀的 NIO.2 支持，用 Java 实现 NIO 并不是遥不可及的事情，只是需要我们熟悉 NIO 的技术细节。

我们需要将服务部署在分布式环境下的不同节点上，通过服务注册的方式，让客户端来自动发现当前可用的服务，并调用这些服务。这需要一种服务注册表（Service Registry）的组件，让它来注册分布式环境下所有的服务地址（包括：主机名与端口号）。

应用、服务、服务注册表之间的关系见下图：



每台 Server 上可发布多个 Service，这些 Service 共用一个 host 与 port，在分布式环境下会提供 Server 共同对外提供 Service。此外，为防止 Service Registry 出现单点故障，因此需要将其搭建为集群环境。

本文将为您揭晓开发轻量级分布式 RPC 框架的具体过程，该框架基于 TCP 协议，提供了 NIO 特性，提供高效的序列化方式，同时也具备服务注册与发现的能力。

根据以上技术需求，我们可使用如下技术选型：

1. Spring：它是最强大的依赖注入框架，也是业界的权威标准。
2. Netty：它使 NIO 编程更加容易，屏蔽了 Java 底层的 NIO 细节。
3. Protostuff：它基于 Protobuf 序列化框架，面向 POJO，无需编写 .proto 文件。
4. ZooKeeper：提供服务注册与发现功能，开发分布式系统的必备选择，同时它也具备天生的集群能力。

相关 Maven 依赖请见附录。

第一步：编写服务接口

<!-- lang: java -->

public interface HelloService {

String hello(String name);

}

将该接口放在独立的客户端 jar 包中，以供应用使用。

第二步：编写服务接口的实现类

<!-- lang: java -->

@RpcService(HelloService.class) // 指定远程接口

public class HelloServiceImpl implements HelloService {

@Override

public String hello(String name) {

return "Hello! " + name;

}

}

使用RpcService注解定义在服务接口的实现类上，需要对该实现类指定远程接口，因为实现类可能会实现多个接口，一定要告诉框架哪个才是远程接口。

RpcService代码如下：

<!-- lang: java -->

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Component // 表明可被 Spring 扫描

public @interface RpcService {

Class<?> value();

}

该注解具备 Spring 的Component注解的特性，可被 Spring 扫描。

该实现类放在服务端 jar 包中，该 jar 包还提供了一些服务端的配置文件与启动服务的引导程序。

第三步：配置服务端

服务端 Spring 配置文件名为spring.xml，内容如下：

<!-- lang: xml -->

<beans ...>

<context:component-scan base-package="com.xxx.rpc.sample.server"/>

<context:property-placeholder location="classpath:config.properties"/>

<!-- 配置服务注册组件 -->

<bean id="serviceRegistry" class="com.xxx.rpc.registry.ServiceRegistry">

<constructor-arg name="registryAddress" value="${registry.address}"/>

</bean>

<!-- 配置 RPC 服务器 -->

<bean id="rpcServer" class="com.xxx.rpc.server.RpcServer">

<constructor-arg name="serverAddress" value="${server.address}"/>

<constructor-arg name="serviceRegistry" ref="serviceRegistry"/>

</bean>

</beans>

具体的配置参数在config.properties文件中，内容如下：

<!-- lang: java -->

# ZooKeeper 服务器

registry.address=127.0.0.1:2181

# RPC 服务器

server.address=127.0.0.1:8000

以上配置表明：连接本地的 ZooKeeper 服务器，并在 8000 端口上发布 RPC 服务。

第四步：启动服务器并发布服务

为了加载 Spring 配置文件来发布服务，只需编写一个引导程序即可：

<!-- lang: java -->

public class RpcBootstrap {

public static void main(String[] args) {

new ClassPathXmlApplicationContext("spring.xml");

}

}

运行RpcBootstrap类的main方法即可启动服务端，但还有两个重要的组件尚未实现，它们分别是：ServiceRegistry与RpcServer，下文会给出具体实现细节。

第五步：实现服务注册

使用 ZooKeeper 客户端可轻松实现服务注册功能，ServiceRegistry代码如下：

<!-- lang: java -->

public class ServiceRegistry {

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

private CountDownLatch latch = new CountDownLatch(1);

private String registryAddress;

public ServiceRegistry(String registryAddress) {

this.registryAddress = registryAddress;

}

public void register(String data) {

if (data != null) {

ZooKeeper zk = connectServer();

if (zk != null) {

createNode(zk, data);

}

}

}

private ZooKeeper connectServer() {

ZooKeeper zk = null;

try {

zk = new ZooKeeper(registryAddress, Constant.ZK\_SESSION\_TIMEOUT, new Watcher() {

@Override

public void process(WatchedEvent event) {

if (event.getState() == Event.KeeperState.SyncConnected) {

latch.countDown();

}

}

});

latch.await();

} catch (IOException | InterruptedException e) {

LOGGER.error("", e);

}

return zk;

}

private void createNode(ZooKeeper zk, String data) {

try {

byte[] bytes = data.getBytes();

String path = zk.create(Constant.ZK\_DATA\_PATH, bytes, ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

LOGGER.debug("create zookeeper node ({} => {})", path, data);

} catch (KeeperException | InterruptedException e) {

LOGGER.error("", e);

}

}

}

其中，通过Constant配置了所有的常量：

<!-- lang: java -->

public interface Constant {

int ZK\_SESSION\_TIMEOUT = 5000;

String ZK\_REGISTRY\_PATH = "/registry";

String ZK\_DATA\_PATH = ZK\_REGISTRY\_PATH + "/data";

}

注意：首先需要使用 ZooKeeper 客户端命令行创建/registry永久节点，用于存放所有的服务临时节点。

第六步：实现 RPC 服务器

使用 Netty 可实现一个支持 NIO 的 RPC 服务器，需要使用ServiceRegistry注册服务地址，RpcServer代码如下：

<!-- lang: java -->

public class RpcServer implements ApplicationContextAware, InitializingBean {

private static final Logger LOGGER = LoggerFactory.getLogger(RpcServer.class);

private String serverAddress;

private ServiceRegistry serviceRegistry;

private Map<String, Object> handlerMap = new HashMap<>(); // 存放接口名与服务对象之间的映射关系

public RpcServer(String serverAddress) {

this.serverAddress = serverAddress;

}

public RpcServer(String serverAddress, ServiceRegistry serviceRegistry) {

this.serverAddress = serverAddress;

this.serviceRegistry = serviceRegistry;

}

@Override

public void setApplicationContext(ApplicationContext ctx) throws BeansException {

Map<String, Object> serviceBeanMap = ctx.getBeansWithAnnotation(RpcService.class); // 获取所有带有 RpcService 注解的 Spring Bean

if (MapUtils.isNotEmpty(serviceBeanMap)) {

for (Object serviceBean : serviceBeanMap.values()) {

String interfaceName = serviceBean.getClass().getAnnotation(RpcService.class).value().getName();

handlerMap.put(interfaceName, serviceBean);

}

}

}

@Override

public void afterPropertiesSet() throws Exception {

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

try {

ServerBootstrap bootstrap = new ServerBootstrap();

bootstrap.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel channel) throws Exception {

channel.pipeline()

.addLast(new RpcDecoder(RpcRequest.class)) // 将 RPC 请求进行解码（为了处理请求）

.addLast(new RpcEncoder(RpcResponse.class)) // 将 RPC 响应进行编码（为了返回响应）

.addLast(new RpcHandler(handlerMap)); // 处理 RPC 请求

}

})

.option(ChannelOption.SO\_BACKLOG, 128)

.childOption(ChannelOption.SO\_KEEPALIVE, true);

String[] array = serverAddress.split(":");

String host = array[0];

int port = Integer.parseInt(array[1]);

ChannelFuture future = bootstrap.bind(host, port).sync();

LOGGER.debug("server started on port {}", port);

if (serviceRegistry != null) {

serviceRegistry.register(serverAddress); // 注册服务地址

}

future.channel().closeFuture().sync();

} finally {

workerGroup.shutdownGracefully();

bossGroup.shutdownGracefully();

}

}

}

以上代码中，有两个重要的 POJO 需要描述一下，它们分别是RpcRequest与RpcResponse。

使用RpcRequest封装 RPC 请求，代码如下：

<!-- lang: java -->

public class RpcRequest {

private String requestId;

private String className;

private String methodName;

private Class<?>[] parameterTypes;

private Object[] parameters;

// getter/setter...

}

使用RpcResponse封装 RPC 响应，代码如下：

<!-- lang: java -->

public class RpcResponse {

private String requestId;

private Throwable error;

private Object result;

// getter/setter...

}

使用RpcDecoder提供 RPC 解码，只需扩展 Netty 的ByteToMessageDecoder抽象类的decode方法即可，代码如下：

<!-- lang: java -->

public class RpcDecoder extends ByteToMessageDecoder {

private Class<?> genericClass;

public RpcDecoder(Class<?> genericClass) {

this.genericClass = genericClass;

}

@Override

public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {

if (in.readableBytes() < 4) {

return;

}

in.markReaderIndex();

int dataLength = in.readInt();

if (dataLength < 0) {

ctx.close();

}

if (in.readableBytes() < dataLength) {

in.resetReaderIndex();

return;

}

byte[] data = new byte[dataLength];

in.readBytes(data);

Object obj = SerializationUtil.deserialize(data, genericClass);

out.add(obj);

}

}

使用RpcEncoder提供 RPC 编码，只需扩展 Netty 的MessageToByteEncoder抽象类的encode方法即可，代码如下：

<!-- lang: java -->

public class RpcEncoder extends MessageToByteEncoder {

private Class<?> genericClass;

public RpcEncoder(Class<?> genericClass) {

this.genericClass = genericClass;

}

@Override

public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out) throws Exception {

if (genericClass.isInstance(in)) {

byte[] data = SerializationUtil.serialize(in);

out.writeInt(data.length);

out.writeBytes(data);

}

}

}

编写一个SerializationUtil工具类，使用Protostuff实现序列化：

<!-- lang: java -->

public class SerializationUtil {

private static Map<Class<?>, Schema<?>> cachedSchema = new ConcurrentHashMap<>();

private static Objenesis objenesis = new ObjenesisStd(true);

private SerializationUtil() {

}

@SuppressWarnings("unchecked")

private static <T> Schema<T> getSchema(Class<T> cls) {

Schema<T> schema = (Schema<T>) cachedSchema.get(cls);

if (schema == null) {

schema = RuntimeSchema.createFrom(cls);

if (schema != null) {

cachedSchema.put(cls, schema);

}

}

return schema;

}

@SuppressWarnings("unchecked")

public static <T> byte[] serialize(T obj) {

Class<T> cls = (Class<T>) obj.getClass();

LinkedBuffer buffer = LinkedBuffer.allocate(LinkedBuffer.DEFAULT\_BUFFER\_SIZE);

try {

Schema<T> schema = getSchema(cls);

return ProtostuffIOUtil.toByteArray(obj, schema, buffer);

} catch (Exception e) {

throw new IllegalStateException(e.getMessage(), e);

} finally {

buffer.clear();

}

}

public static <T> T deserialize(byte[] data, Class<T> cls) {

try {

T message = (T) objenesis.newInstance(cls);

Schema<T> schema = getSchema(cls);

ProtostuffIOUtil.mergeFrom(data, message, schema);

return message;

} catch (Exception e) {

throw new IllegalStateException(e.getMessage(), e);

}

}

}

以上了使用 Objenesis 来实例化对象，它是比 Java 反射更加强大。

注意：如需要替换其它序列化框架，只需修改SerializationUtil即可。当然，更好的实现方式是提供配置项来决定使用哪种序列化方式。

使用RpcHandler中处理 RPC 请求，只需扩展 Netty 的SimpleChannelInboundHandler抽象类即可，代码如下：

<!-- lang: java -->

public class RpcHandler extends SimpleChannelInboundHandler<RpcRequest> {

private static final Logger LOGGER = LoggerFactory.getLogger(RpcHandler.class);

private final Map<String, Object> handlerMap;

public RpcHandler(Map<String, Object> handlerMap) {

this.handlerMap = handlerMap;

}

@Override

public void channelRead0(final ChannelHandlerContext ctx, RpcRequest request) throws Exception {

RpcResponse response = new RpcResponse();

response.setRequestId(request.getRequestId());

try {

Object result = handle(request);

response.setResult(result);

} catch (Throwable t) {

response.setError(t);

}

ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);

}

private Object handle(RpcRequest request) throws Throwable {

String className = request.getClassName();

Object serviceBean = handlerMap.get(className);

Class<?> serviceClass = serviceBean.getClass();

String methodName = request.getMethodName();

Class<?>[] parameterTypes = request.getParameterTypes();

Object[] parameters = request.getParameters();

/\*Method method = serviceClass.getMethod(methodName, parameterTypes);

method.setAccessible(true);

return method.invoke(serviceBean, parameters);\*/

FastClass serviceFastClass = FastClass.create(serviceClass);

FastMethod serviceFastMethod = serviceFastClass.getMethod(methodName, parameterTypes);

return serviceFastMethod.invoke(serviceBean, parameters);

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

LOGGER.error("server caught exception", cause);

ctx.close();

}

}

为了避免使用 Java 反射带来的性能问题，我们可以使用 CGLib 提供的反射 API，如上面用到的FastClass与FastMethod。

第七步：配置客户端

同样使用 Spring 配置文件来配置 RPC 客户端，spring.xml代码如下：

<!-- lang: java -->

<beans ...>

<context:property-placeholder location="classpath:config.properties"/>

<!-- 配置服务发现组件 -->

<bean id="serviceDiscovery" class="com.xxx.rpc.registry.ServiceDiscovery">

<constructor-arg name="registryAddress" value="${registry.address}"/>

</bean>

<!-- 配置 RPC 代理 -->

<bean id="rpcProxy" class="com.xxx.rpc.client.RpcProxy">

<constructor-arg name="serviceDiscovery" ref="serviceDiscovery"/>

</bean>

</beans>

其中config.properties提供了具体的配置：

<!-- lang: java -->

# ZooKeeper 服务器

registry.address=127.0.0.1:2181

第八步：实现服务发现

同样使用 ZooKeeper 实现服务发现功能，见如下代码：

<!-- lang: java -->

public class ServiceDiscovery {

private static final Logger LOGGER = LoggerFactory.getLogger(ServiceDiscovery.class);

private CountDownLatch latch = new CountDownLatch(1);

private volatile List<String> dataList = new ArrayList<>();

private String registryAddress;

public ServiceDiscovery(String registryAddress) {

this.registryAddress = registryAddress;

ZooKeeper zk = connectServer();

if (zk != null) {

watchNode(zk);

}

}

public String discover() {

String data = null;

int size = dataList.size();

if (size > 0) {

if (size == 1) {

data = dataList.get(0);

LOGGER.debug("using only data: {}", data);

} else {

data = dataList.get(ThreadLocalRandom.current().nextInt(size));

LOGGER.debug("using random data: {}", data);

}

}

return data;

}

private ZooKeeper connectServer() {

ZooKeeper zk = null;

try {

zk = new ZooKeeper(registryAddress, Constant.ZK\_SESSION\_TIMEOUT, new Watcher() {

@Override

public void process(WatchedEvent event) {

if (event.getState() == Event.KeeperState.SyncConnected) {

latch.countDown();

}

}

});

latch.await();

} catch (IOException | InterruptedException e) {

LOGGER.error("", e);

}

return zk;

}

private void watchNode(final ZooKeeper zk) {

try {

List<String> nodeList = zk.getChildren(Constant.ZK\_REGISTRY\_PATH, new Watcher() {

@Override

public void process(WatchedEvent event) {

if (event.getType() == Event.EventType.NodeChildrenChanged) {

watchNode(zk);

}

}

});

List<String> dataList = new ArrayList<>();

for (String node : nodeList) {

byte[] bytes = zk.getData(Constant.ZK\_REGISTRY\_PATH + "/" + node, false, null);

dataList.add(new String(bytes));

}

LOGGER.debug("node data: {}", dataList);

this.dataList = dataList;

} catch (KeeperException | InterruptedException e) {

LOGGER.error("", e);

}

}

}

第九步：实现 RPC 代理

这里使用 Java 提供的动态代理技术实现 RPC 代理（当然也可以使用 CGLib 来实现），具体代码如下：

<!-- lang: java -->

public class RpcProxy {

private String serverAddress;

private ServiceDiscovery serviceDiscovery;

public RpcProxy(String serverAddress) {

this.serverAddress = serverAddress;

}

public RpcProxy(ServiceDiscovery serviceDiscovery) {

this.serviceDiscovery = serviceDiscovery;

}

@SuppressWarnings("unchecked")

public <T> T create(Class<?> interfaceClass) {

return (T) Proxy.newProxyInstance(

interfaceClass.getClassLoader(),

new Class<?>[]{interfaceClass},

new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

RpcRequest request = new RpcRequest(); // 创建并初始化 RPC 请求

request.setRequestId(UUID.randomUUID().toString());

request.setClassName(method.getDeclaringClass().getName());

request.setMethodName(method.getName());

request.setParameterTypes(method.getParameterTypes());

request.setParameters(args);

if (serviceDiscovery != null) {

serverAddress = serviceDiscovery.discover(); // 发现服务

}

String[] array = serverAddress.split(":");

String host = array[0];

int port = Integer.parseInt(array[1]);

RpcClient client = new RpcClient(host, port); // 初始化 RPC 客户端

RpcResponse response = client.send(request); // 通过 RPC 客户端发送 RPC 请求并获取 RPC 响应

if (response.isError()) {

throw response.getError();

} else {

return response.getResult();

}

}

}

);

}

}

使用RpcClient类实现 RPC 客户端，只需扩展 Netty 提供的SimpleChannelInboundHandler抽象类即可，代码如下：

<!-- lang: java -->

public class RpcClient extends SimpleChannelInboundHandler<RpcResponse> {

private static final Logger LOGGER = LoggerFactory.getLogger(RpcClient.class);

private String host;

private int port;

private RpcResponse response;

private final Object obj = new Object();

public RpcClient(String host, int port) {

this.host = host;

this.port = port;

}

@Override

public void channelRead0(ChannelHandlerContext ctx, RpcResponse response) throws Exception {

this.response = response;

synchronized (obj) {

obj.notifyAll(); // 收到响应，唤醒线程

}

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

LOGGER.error("client caught exception", cause);

ctx.close();

}

public RpcResponse send(RpcRequest request) throws Exception {

EventLoopGroup group = new NioEventLoopGroup();

try {

Bootstrap bootstrap = new Bootstrap();

bootstrap.group(group).channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel channel) throws Exception {

channel.pipeline()

.addLast(new RpcEncoder(RpcRequest.class)) // 将 RPC 请求进行编码（为了发送请求）

.addLast(new RpcDecoder(RpcResponse.class)) // 将 RPC 响应进行解码（为了处理响应）

.addLast(RpcClient.this); // 使用 RpcClient 发送 RPC 请求

}

})

.option(ChannelOption.SO\_KEEPALIVE, true);

ChannelFuture future = bootstrap.connect(host, port).sync();

future.channel().writeAndFlush(request).sync();

synchronized (obj) {

obj.wait(); // 未收到响应，使线程等待

}

if (response != null) {

future.channel().closeFuture().sync();

}

return response;

} finally {

group.shutdownGracefully();

}

}

}

第十步：发送 RPC 请求

使用 JUnit 结合 Spring 编写一个单元测试，代码如下：

<!-- lang: java -->

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = "classpath:spring.xml")

public class HelloServiceTest {

@Autowired

private RpcProxy rpcProxy;

@Test

public void helloTest() {

HelloService helloService = rpcProxy.create(HelloService.class);

String result = helloService.hello("World");

Assert.assertEquals("Hello! World", result);

}

}

运行以上单元测试，如果不出意外的话，您应该会看到绿条。

总结

本文通过 Spring + Netty + Protostuff + ZooKeeper 实现了一个轻量级 RPC 框架，使用 Spring 提供依赖注入与参数配置，使用 Netty 实现 NIO 方式的数据传输，使用 Protostuff 实现对象序列化，使用 ZooKeeper 实现服务注册与发现。使用该框架，可将服务部署到分布式环境中的任意节点上，客户端通过远程接口来调用服务端的具体实现，让服务端与客户端的开发完全分离，为实现大规模分布式应用提供了基础支持。

附录：Maven 依赖

<!-- lang: xml -->

<!-- JUnit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

<!-- SLF4J -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.7</version>

</dependency>

<!-- Spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>3.2.12.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>3.2.12.RELEASE</version>

<scope>test</scope>

</dependency>

<!-- Netty -->

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-all</artifactId>

<version>4.0.24.Final</version>

</dependency>

<!-- Protostuff -->

<dependency>

<groupId>com.dyuproject.protostuff</groupId>

<artifactId>protostuff-core</artifactId>

<version>1.0.8</version>

</dependency>

<dependency>

<groupId>com.dyuproject.protostuff</groupId>

<artifactId>protostuff-runtime</artifactId>

<version>1.0.8</version>

</dependency>

<!-- ZooKeeper -->

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.6</version>

</dependency>

<!-- Apache Commons Collections -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-collections4</artifactId>

<version>4.0</version>

</dependency>

<!-- Objenesis -->

<dependency>

<groupId>org.objenesis</groupId>

<artifactId>objenesis</artifactId>

<version>2.1</version>

</dependency>

<!-- CGLib -->

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.1</version>

</dependency>

源码地址：<http://git.oschina.net/huangyong/rpc>