rpc系列3-支持异步调用，提供future、callback的能力。

作者 TopGun_Viper (/u/aee2b91e3398) +关注
2016.09.14 21:28 字数 965 阅读 58 评论 0 喜欢 0
(/u/aee2b91e3398)

支持异步调用，提供future、callback的能力。

在实现新功能之前，先将RpcBuilder重构下，职责分离：

- RpcConsumer：提供给客户端操作接口
- RpcProvider：提供给服务端

```
public final class RpcConsumer implements InvocationHandler{

    private String host;

    private int port;

    private Class<?> interfaceClass;

    private int timeout;

    private static int nThreads = Runtime.getRuntime().availableProcessors() * 2;

    private static ExecutorService handlerPool = Executors.newFixedThreadPool(nThreads);


    public RpcConsumer targetHostPort(String host, int port){
        this.host = host;
        this.port = port;
        return this;
    }
    public RpcConsumer interfaceClass(Class<?> interfaceClass) {
        this.interfaceClass = interfaceClass;
        return this;
    }
    public RpcConsumer timeout(int timeout){
        this.timeout = timeout;
        return this;
    }
    public Object newProxy(){
        return Proxy.newProxyInstance(RpcConsumer.class.getClassLoader(), new Class<?>[]{
this.interfaceClass}, this);
    }


    /**
     * 拦截目标方法->序列化method对象->发起socket连接
     */
    @Override
    public Object invoke(Object proxy, Method method,
            Object[] args) throws Throwable {
        Object retVal = null;

        RpcRequest request = new RpcRequest(method.getName(), method.getParameterTypes(),
args,RpcContext.getAttributes());
        Object response;
        try{
            //网络传输模块分到doInvoke中
            response = doInvoke(request);
        }catch(Exception e){
            throw e;
        }
        if(response instanceof RpcResponse){
            RpcResponse rpcResp  = (RpcResponse)response;
            if(!rpcResp.isError()){
                retVal = rpcResp.getResponseBody();
            }else{
                throw new RpcException(rpcResp.getErrorMsg());
            }
        }
        return retVal;
    }
    private Object doInvoke(RpcRequest request) throws IOException, ClassNotFoundExceptio
n{
        //创建连接,获取输入输出流
        Socket socket = new Socket(host,port);
        Object retVal = null;
        try{
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
            try{
                //发送
                out.writeObject(request);
                //接受server端的返回信息---阻塞
                retVal = in.readObject();

            }finally{
                out.close();
                in.close();
            }
        }finally{
            socket.close();
        }
        return retVal;
    }
}
```

RpcProvider：

```
public final class RpcProvider {


    private static int nThreads = Runtime.getRuntime().availableProcessors() * 2;
    private static ExecutorService handlerPool = Executors.newFixedThreadPool(nThreads);

//发布服务
    public static void publish(final Object service, final int port) throws IOException{
        if (service == null)
            throw new IllegalArgumentException("service can not be null.");

        ServerSocket server = new ServerSocket(port);
        System.out.println("server started!!!");
        while(true){
            Socket socket = server.accept();//监听请求--阻塞
            //异步处理
            handlerPool.submit(new Handler(service,socket));
        }
    }
    static class Handler implements Runnable{

        private Object service;

        private Socket socket;

        public Handler(Object service,Socket socket){
            this.service = service;
            this.socket = socket;
        }
        public void run() {
            try{
                ObjectInputStream in = null;
                ObjectOutputStream out = null;
                RpcResponse response = new RpcResponse();
                try {
                    in = new ObjectInputStream(socket.getInputStream());
                    out = new ObjectOutputStream(socket.getOutputStream());

                    Object req = in.readObject();
                    if(req instanceof RpcRequest){
                        RpcRequest rpcRequest = (RpcRequest)req;
                        //关联客户端传来的上下文
                        RpcContext.context.set(rpcRequest.getContext());
                        Method method = service.getClass().getMethod(rpcRequest.getMethod
Name(), rpcRequest.getParameterTypes());
                        Object retVal = method.invoke(service, rpcRequest.getArgs());
                        response.setResponseBody(retVal);
                        out.writeObject(response);
                    }
                } catch (InvocationTargetException e) {
                    response.setErrorMsg(e.getTargetException().getMessage());
                    response.setResponseBody(e.getTargetException());
                    out.writeObject(response);
                }finally{
                    in.close();
                    out.close();
                }
            }catch(Exception e){}
        }
    }
}
```

> 下面开始考虑如何实现future、callback功能。

谈到异步，我们首先想到了Java提供的Future机制，Future代表一个异步计算结果，提交一个任务后会立刻返回，通过future.get()方法来获取计算结果，该方法会阻塞当前线程，直到结果返回。使用形式如下：

```
//提交异步任务，立即返回
Future<Object> future = executePool.submit(new Callable<Object>(){
    @Override
    public Object call(){
        //do business
    }
});

//do othre business

Object retVal = future.get();//阻塞，直到计算出结果
```

> 思路

rpc中异步方法可以使用Future这个特性。支持异步调用效果和future类似，假设异步方法调用入口：

- asyncCall(String methodName)
  我们再asyncCall方法中构造一个异步任务，其目的就是通过socket将需要调用的方法传给server端，然后等待获取server返回的结果。这个异步任务我们可以直接实现一个FutureTask对象，如下：

```
FutureTask<RpcResponse> futureTask = new FutureTask<RpcResponse>(new Callable<RpcResponse
>(){
        public RpcResponse call() throws Exception {
            //构造RpcRequest对象，发送给server并获取返回结果
             RpcResponse retVal = sendRequest(request);
             return retVal;
        }
    });
    new Thread(futureTask).start();
```

上面是一种实现方法，不过我这里没有新建Thread，而是直接将任务提交到线程池中，实现如下：

```
//公用的线程池
private static ExecutorService handlerPool = Executors.newFixedThreadPool(nThreads);

        //构造并提交FutureTask异步任务
Future<RpcResponse> f = (Future<RpcResponse>) handlerPool.submit(new Callable<RpcResponse
>(){
        public RpcResponse call() throws Exception {
            //构造RpcRequest对象，发送给server并获取返回结果
             RpcResponse retVal = sendRequest(request);
            return retVal;
            }
        });
```

> 异步任务已经构造完毕了，那么异步结果如何获取？

最简单的方式是直接将Future实例返回给客户端即可，客户端通过获取的Future对象，调用相应方法获取异步结果。不过这样话有一个问题，我们获取的RpcResponse对象封装的是server端返回的结果，这个结果可能是我们期望的方法执行返回值，也可能是server端抛出的异常，这个获取结果的过程对用户应该是透明的，即用户进行一次方法调用，如果正常，则返回结果，不正常直接抛出对应的Exception即可，让用户自己通过RpcResponse的isError判断结果是不是异常显然是不合适的，所以这里使用了题目中提供的异步结果获取的一个工具类：ResponseFuture。ResponseFuture的作用就是将上面分析的结果获取过程进行封装，实现如下：

```
public class ResponseFuture {

    public static ThreadLocal<Future<RpcResponse>> futureThreadLocal = new ThreadLocal<Fu
ture<RpcResponse>>();

    public static Object getResponse(long timeout) throws InterruptedException {
        if (null == futureThreadLocal.get()) {
            throw new RuntimeException("Thread [" + Thread.currentThread() + "] have not
set the response future!");
        }

        try {
            RpcResponse response =futureThreadLocal.get().get(timeout, TimeUnit.MILLISECO
NDS);
            //如果是异常，直接抛出
            if (response.isError()) {
                throw new RuntimeException(response.getErrorMsg());
            }
            return response.getResponseBody();
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        } catch (TimeoutException e) {
            throw new RuntimeException("Time out", e);
        }
    }

    public static void setFuture(Future<RpcResponse> future){
        futureThreadLocal.set(future);
    }
}
```

客户端在进行异步方法调用之后，直接用ResponseFuture.get(timeout)即可获取结果。

> 异步方法能否多次调用？

考虑这么一个问题，如果客户端异步调用methodA方法，在结果返回之前，客户端能否再次调用methodA呢？显然是不可以！所以每次异步调用的时候，我们需要对异步调用方法进行记录，保证结果返回前只调用一次。保存方法的数据结构也是ThreadLocal实现，如下所示：

```
    /**
     * 存放当前线程正在执行的异步方法
     */
    private static final ThreadLocal<Set<String>> asyncMethods = new ThreadLocal<Set<Stri
ng>>(){
        @Override
        public Set<String> initialValue()
        {
            return new LinkedHashSet<String>();
        }
    };
```

异步调用的的Future能力已经完成，下面考虑下callback如何实现。

> 同时在异步调用过程中添加callback函数。

题目提供了Callback接口：

```
public interface ResponseCallbackListener {

    public void onResponse(Object response);

    public void onTimeout();

    public void onException(Exception e);
}
```

callback的实现其实很简单了,在asyncCall执行过程中在适当的位置执行callback函数，比如抛出异常了，那么执行onException函数，调用超时了，则执行onTimeout函数。

综合上述分析，下面看下asyncCall的整体实现：

```
public final class RpcConsumer implements InvocationHandler{

    //。。。

    private int timeout;

    private static int nThreads = Runtime.getRuntime().availableProcessors() * 2;

    private static ExecutorService handlerPool = Executors.newFixedThreadPool(nThreads);

    /**
     * 存放当前线程正在执行的异步方法
     */
    private static final ThreadLocal<Set<String>> asyncMethods = new ThreadLocal<Set<Stri
ng>>(){
        @Override
        public Set<String> initialValue()
        {
            return new LinkedHashSet<String>();
        }
    };

    public void asynCall(String methodName) {
        asynCall(methodName, null);
    }

    /**
     * 异步方法，支持callback
     *
     * @param methodName
     * @param callbackListener
     */
    public <T extends ResponseCallbackListener> void asynCall(final String methodName, T
callbackListener) {
        //记录异步方法调用
        asyncMethods.get().add(methodName);

        //构造并提交FutureTask异步任务
        Future<RpcResponse> f = (Future<RpcResponse>) handlerPool.submit(new Callable<Rpc
Response>(){
            @Override
            public RpcResponse call() throws Exception {
                RpcRequest request = new RpcRequest(methodName,null,null,RpcContext.getAt
tributes());
                Object response;
                try{
                    response = doInvoke(request);
                }catch(Exception e){
                    throw e;
                }
                return (RpcResponse) response;
            }
        });

        RpcResponse response;
        if(callbackListener != null){
            try {
                //阻塞
                response = (RpcResponse) f.get(timeout,TimeUnit.MILLISECONDS);
                if(response.isError()){
                    //执行回调方法
                    callbackListener.onException(new RpcException(response.getErrorMsg())
);
                }else{
                    callbackListener.onResponse(response.getResponseBody());
                }
            } catch(TimeoutException e){
                callbackListener.onTimeout();
            }catch (Exception e) {}
        }else{
            //client端将从ResponseFuture中获取结果
            ResponseFuture.setFuture(f);
        }
    }
    public void cancelAsyn(String methodName) {
        asyncMethods.get().remove(methodName);
    }
```

```
    @Override
    public Object invoke(Object proxy, Method method,
            Object[] args) throws Throwable {
        //如果是异步方法，立即返回null
        if(asyncMethods.get().contains(method.getName())) return null;
        //。。。。
    }
```

future功能测试代码：

```
@Test
public void testAsyncCall(){
    consumer.asynCall("test");//测试future能力
    //立即返回
    String nullValue = userService.test();
    System.out.println(nullValue);
    Assert.assertEquals(null, nullValue);
    try {
        String result = (String) ResponseFuture.getResponse(TIMEOUT);
        Assert.assertEquals("hello client, this is rpc server.", result);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        consumer.cancelAsyn(
    }
}
```

callback测试：

自定义ResponseCallbackListener实现类UserServiceListener：

```
public class UserServiceListener implements ResponseCallbackListener {
    private CountDownLatch latch = new CountDownLatch(1);

    private Object response;

    public Object getResponse() throws InterruptedException {
        latch.await(10, TimeUnit.SECONDS);
        if(response == null)
            throw new RuntimeException("The response doesn't come back.");
        return response;
    }
    @Override
    public void onResponse(Object response) {
        System.out.println("This method is call when response arrived");
        this.response = response;
        latch.countDown();
    }

    @Override
    public void onTimeout() {
        throw new RuntimeException("This call has taken time more than timeout value");
    }

    @Override
    public void onException(Exception e) {
        throw new RuntimeException(e);
    }
}
```

ClientTest中测试代码：

```
    @Test
    public void testCallback() {
        UserServiceListener listener = new UserServiceListener();
        consumer.asynCall("test", listener);
        String nullStr = userService.test();//立刻返回null
        Assert.assertEquals(null, nullStr);
        try {
            String str = (String)listener.getResponse();
            Assert.assertEquals("hello client, this is rpc server.", str);
        } catch (InterruptedException e) {
        }
    }
```

输出：

> This method is call when response arrived

好了，到此**支持异步调用，提供future、callback的能力**，基本实现，当然实现过程肯定还有很多改进的地方，不断学习，不断进步！！！

github上完整源码 (https://github.com/TopGunViper/rpc-race/tree/feature_callback&future)