rpc系列4-处理超时场景.及提供hook

作者 TopGun_Viper (/u/aee2b91e3398) + 关注
2016.09.16 23:45 字数 397 阅读 30 评论 0 喜欢 0
(/u/aee2b91e3398)

问题：客户端发起远程调用，如果服务端长时间不返回怎么办？

这就涉及到一个调用超时的问题，平时我们应用中很多场景都会规定超时时间，比如：sql查询超时，http请求超时等。那么如果服务端方法执行的时间超过规定的timeout时间，那么客户端就需要调出当前调用，抛出TimeoutException。

好了，下面开始对RpcBuidler进行改造了，让其支持超时情况的处理。同样，先给出预期的测试方案和结果：

```
// 业务类UserService在之前的基础上增加超时调用的方法：
public interface UserService {

    // other method

    /**
     * 超时测试
     */
    public boolean timeoutTest();

}
//实现类
public class UserServiceImpl implements UserService {

     // other method

    @Override
    public boolean timeoutTe
        try {
            //模拟长时间执行
            Thread.sleep(10
        } catch (Interrupted          n e) {}
        return true;
    }
}
```

ClientTest中测试代码:

```
    @Test
    public void timeoutTest(){
        long beginTime = System.currentTimeMillis();
        try {
            boolean result = userService.timeoutTest();
        } catch (Exception e) {
            long period = System.currentTimeMillis() - beginTime;
            System.out.println("period:" + period);
            Assert.assertTrue(period < 3100);
        }
    }
```

有了异步方法的实现经验，其实这个超时处理过程和异步非常类似，都是利用Future机制来实现的，下面对doInvoke方法进行重构,返回一个异步任务：

```
    private Future<RpcResponse> doInvoke(final RpcRequest request) throws IOException, Cl
assNotFoundException{

        //构造并提交FutureTask异步任务
        Future<RpcResponse> retVal = (Future<RpcResponse>) handlerPool.submit(new Callabl
e<RpcResponse>(){
            @Override
            public RpcResponse call() throws Exception {
                Object res = null;
                try{
                    //创建连接,获取输入输出流
                    Socket socket = new Socket(host,port);
                    try{
                        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputS
tream());
                        ObjectInputStream in = new ObjectInputStream(socket.getInputStrea
m());
                        try{
                            //发送
                            out.writeObject(request);
                            //接受server端的返回信息---阻塞
                            res = in.readObject();
                        }finally{
                            out.close();
                            in.close();
                        }
                    }finally{
                        socket.close();
                    }
                }catch(Exception e){
                    throw e;
                }
                return (RpcResponse)res;
            }
        });
        return retVal;
    }
```

回调方法invoke修改如下：

```
    @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;
        Object retVal = null;

        RpcRequest request = new RpcRequest(method.getName(), method.getParameterTypes(),
args,RpcContext.getAttributes());
        RpcResponse rpcResp  = null;
        try{
            Future<RpcResponse> response = doInvoke(request);
            //获取异步结果
            rpcResp  = (RpcResponse)response.get(TIMEOUT,TimeUnit.MILLISECONDS);
        }catch(TimeoutException e){
            throw e;
        }catch(Exception e){}

        if(!rpcResp.isError()){
            retVal = rpcResp.getResponseBody();
        }else{
            throw new RpcException(rpcResp.getErrorMsg());
        }
        return retVal;
    }
```

可见，经过这样改造后，所有的方法调用都是通过Future获取结果。

提供Hook，让开发人员进行RPC层面的AOP。

首先看下题目提供的Hook接口：

```
public interface ConsumerHook {
    public void before(RpcRequest request);
    public void after(RpcRequest request);
}
//实现类
public class UserConsumerHook implements ConsumerHook{
    @Override
    public void before(RpcRequest request) {
        RpcContext.addAttribute("hook key","this is pass by hook");
    }

    @Override
    public void after(RpcRequest request) {
        System.out.println("I have finished Rpc calling.");
    }
}
```

hook实现的功能很简单，即在客户端进行远程调用的前后执行before和after方法。

```
public final class RpcConsumer implements InvocationHandler{

    //。。。

    //钩子
    private ConsumerHook hook;

    public RpcConsumer hook(ConsumerHook hook){
        this.hook = hook;
        return this;
    }

static{
        userService = (UserService)consumer.targetHostPort(host, port)
                            .interfaceClass(UserService.class)
                            .timeout(TIMEOUT)
                            .hook(new UserConsumerHook())//新增钩子
                            .newProxy();
    }
//。。。
}

//UserServiceImpl中的测试方法
public Map<String, Object> getMap() {
        Map<String,Object> newMap = new HashMap<String,Object>();
        newMap.put("name","getMap");
        newMap.putAll(RpcContext.getAttributes());
        return newMap;
}
```

我们只需要在doInvoke方法开始出添加钩子函数的执行逻辑即可。如下：

```
    private Future<RpcResponse> doInvoke(final RpcRequest request) throws IOException, Cl
assNotFoundException{
        //插入钩子
        hook.before(request);
        //。。。
}
```

同时在asyncCall和invoke方法的结束添加after的执行逻辑。具体实现可以看源码。

github附上源码 (https://github.com/TopGunViper/rpc-race/tree/feature_timeout)

只是一个简单的rpc demo (/nb/6229705)

TopGun_Viper (/u/aee2b91e3398)
写了 23557 字，被 15 人关注，获得了 29 个喜欢
(/u/aee2b91e3398)

＋关注

吾日三省吾code，可以为师矣。。。

赞赏支持

♡ 喜欢 (/sign_in) | 0

更多分享

(http://cwb.assets.jianshu.io/notes/images/582972

登录 (/sign_in)后发表评论

评论

智慧如你，不想发表一点想法 (/sign_in)咩~