Dubbo的负载均衡和rpc通讯原理

1、负载均衡

1.1、RandomLoadBalance

1.1.1、使用场景

在大请求量的情况下，想要让请求落到所有节点的比率跟机器的权重比率接近那么就可以使用随机算法。

随机算法是dubbo默认的负载均衡算法，在dubbo随机的负载均衡算法有两种

1、完全随机

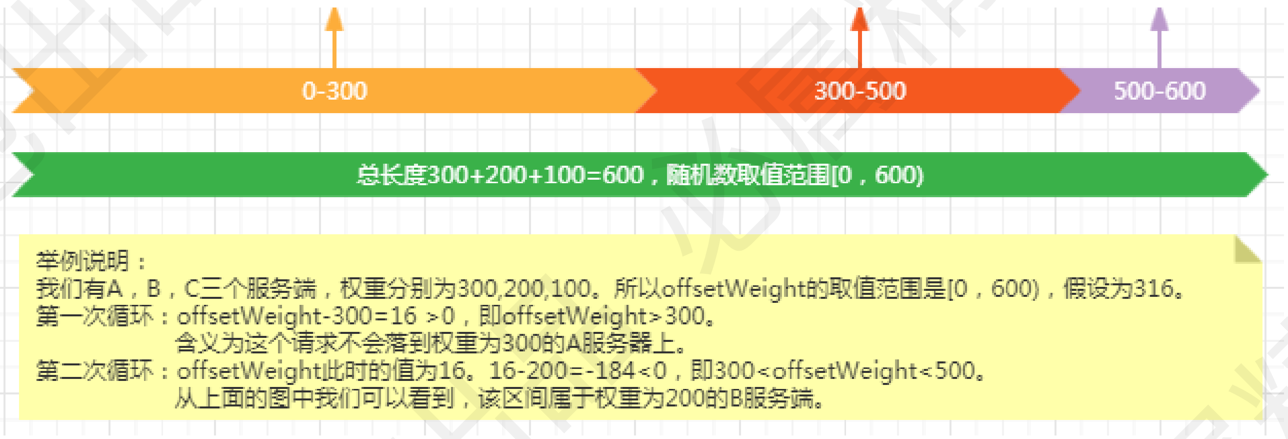
2、权重随机

完全随机就是不考虑权重从服务列表中根据服务列表的长度来随机选择一个，这样做是没考虑机器性能差异的。

权重随机，是给不同机器的机器加不通的权重，机器性能好点的权重高，性能差点的权重低，权重高的机器能随机分配更多的请求，反之亦然。

1.1.2、源码分析

```
@Override
protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation
invocation) {
    // Number of invokers
    int length = invokers.size();

    //判断是否需要权重随机
    if (!needWeightLoadBalance(invokers,invocation)){
        return invokers.get(ThreadLocalRandom.current().nextInt(length));
    }

    // Every invoker has the same weight?
    boolean sameWeight = true;
    // the maxWeight of every invokers, the minWeight = 0 or the maxWeight of the last
invoker
    int[] weights = new int[length];
    // The sum of weights
    int totalWeight = 0;
    for (int i = 0; i < length; i++) {
        //获取每一个invoker的权重
        int weight = getWeight(invokers.get(i), invocation);
        // Sum
        totalWeight += weight;
        // save for later use
        weights[i] = totalWeight;
        //如果各节点权重是不相同的
        if (sameWeight && totalWeight != weight * (i + 1)) {
            sameWeight = false;
        }
    }
    if (totalWeight > 0 && !sameWeight) {
        // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly based on totalWeight.
        int offset = ThreadLocalRandom.current().nextInt(totalWeight);
        // Return a invoker based on the random value.
        for (int i = 0; i < length; i++) {
            //判断offset落在哪一个权重区间
            if (offset < weights[i]) {
                return invokers.get(i);
            }
        }
    }
    //如果权重相同，则是真随机
    // If all invokers have the same weight value or totalWeight=0, return evenly.
    return invokers.get(ThreadLocalRandom.current().nextInt(length));
}
```



举例说明：
我们有A，B，C三个服务端，权重分别为300,200,100。所以offsetWeight的取值范围是[0，600)，假设为316。
第一次循环：offsetWeight-300=16 >0，即offsetWeight>300。
含义为这个请求不会落到权重为300的A服务器上。
第二次循环：offsetWeight此时的值为16。16-200=-184<0，即300<offsetWeight<500。
从上面的图中我们可以看到，该区间属于权重为200的B服务端。

1.2、RoundRobinLoadBalance

1.2.1、使用场景

无需记录当前所有服务器的链接状态，所以它一种无状态负载均衡算法，实现简单，适用于每台服务器性能相近的场景下。为了解决轮询算法应用场景的局限性。当遇到每台服务器的性能不一致的情况，我们需要对轮询过程进行加权，以调控每台服务器的负载。

经过加权后，每台服务器能够得到的请求数比例，**接近或等于他们的权重比。**比如服务器 A、B、C 权重比为 5:3:2。那么在10次请求中，服务器 A 将收到其中的5次请求，服务器 B 会收到其中的3次请求，服务器 C 则收到其中的2次请求。这篇博客对加权轮询分析得比较透彻：https://www.cnblogs.com/thisiswhy/p/12048507.html

1.2.2、源码分析

```
@Override
protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
    ConcurrentMap<String, WeightedRoundRobin> map = methodWeightMap.computeIfAbsent(key, k
-> new ConcurrentHashMap<>());
    int totalWeight = 0;
    long maxCurrent = Long.MIN_VALUE;
    long now = System.currentTimeMillis();
    Invoker<T> selectedInvoker = null;
    WeightedRoundRobin selectedWRR = null;
    for (Invoker<T> invoker : invokers) {
        String identifyString = invoker.getUrl().toIdentityString();
        int weight = getWeight(invoker, invocation);
        WeightedRoundRobin weightedRoundRobin = map.computeIfAbsent(identifyString, k -> {
            WeightedRoundRobin wrr = new WeightedRoundRobin();
            wrr.setWeight(weight);
            return wrr;
        });

        if (weight != weightedRoundRobin.getWeight()) {
            //weight changed
            weightedRoundRobin.setWeight(weight);
        }
        long cur = weightedRoundRobin.increaseCurrent();
        weightedRoundRobin.setLastUpdate(now);
        if (cur > maxCurrent) {
            maxCurrent = cur;
            selectedInvoker = invoker;
            selectedWRR = weightedRoundRobin;
        }
        totalWeight += weight;
    }
    if (invokers.size() != map.size()) {
        map.entrySet().removeIf(item -> now - item.getValue().getLastUpdate() >
RECYCLE_PERIOD);
    }
    if (selectedInvoker != null) {
        selectedWRR.sel(totalWeight);
        return selectedInvoker;
    }
    // should not happen here
    return invokers.get(0);
}
```

从上述代码我们可以看到，MockClusterInvoker当中就是走了三套逻辑：

1、没有配置mock的情况

2、mock="force:"的情况

3、配置了mock的其他情况

我们都知道如果配置了force:就代表要进行强制降级，就不会走后端的rpc调用了，所以这里是直接调用到了

```
result = doMockInvoke(invocation, null);
```

```java
private Result doMockInvoke(Invocation invocation, RpcException e) {
    Result result = null;
    Invoker<T> minvoker;

    //选择一个MockInvoker的实例，这里是选不到的
    List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
    if (CollectionUtils.isEmpty(mockInvokers)) {
        //所以代码会走这里，创建一个MockInvoker对象
        minvoker = (Invoker<T>) new MockInvoker(getUrl(), directory.getInterface());
    } else {
        minvoker = mockInvokers.get(0);
    }
    try {
        //调用mock的实现类方法
        result = minvoker.invoke(invocation);
    } catch (RpcException me) {
        if (me.isBiz()) {
            result = AsyncRpcResult.newDefaultAsyncResult(me.getCause(), invocation);
        } else {
            throw new RpcException(me.getCode(), getMockExceptionMessage(e, me),
me.getCause());
        }
    } catch (Throwable me) {
        throw new RpcException(getMockExceptionMessage(e, me), me.getCause());
    }
    return result;
}
```

我们再看看mockInvoker的invoke方法

```java
@Override
public Result invoke(Invocation invocation) throws RpcException {
    if (invocation instanceof RpcInvocation) {
        ((RpcInvocation) invocation).setInvoker(this);
    }
    String mock = null;
    if (getUrl().hasMethodParameter(invocation.getMethodName())) {
        mock = getUrl().getParameter(invocation.getMethodName() + "." + MOCK_KEY);
    }
    if (StringUtils.isBlank(mock)) {
        //获取配置的mock的属性值
        mock = getUrl().getParameter(MOCK_KEY);
    }

    if (StringUtils.isBlank(mock)) {
        throw new RpcException(new IllegalAccessException("mock can not be null. url :" +
url));
    }
    //把force:前缀去掉，获取后面的值
    mock = normalizeMock(URL.decode(mock));
    //如果是return开头
    if (mock.startsWith(RETURN_PREFIX)) {
        //获取return后面的值
```

```
            mock = mock.substring(RETURN_PREFIX.length()).trim();
            try {
                //获取返回值类型
                Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
                //把return后面的值包装成返回值类型
                Object value = parseMockValue(mock, returnTypes);
                //注解把结果返回没走后端rpc调用
                return AsyncRpcResult.newDefaultAsyncResult(value, invocation);
            } catch (Exception ew) {
                throw new RpcException("mock return invoke error. method :" +
  invocation.getMethodName()
                        + ", mock:" + mock + ", url: " + url, ew);
            }
            //如果是throw
        } else if (mock.startsWith(THROW_PREFIX)) {
            mock = mock.substring(THROW_PREFIX.length()).trim();
            if (StringUtils.isBlank(mock)) {
                throw new RpcException("mocked exception for service degradation.");
            } else { // user customized class
                //获取异常实例
                Throwable t = getThrowable(mock);
                //直接往上抛异常
                throw new RpcException(RpcException.BIZ_EXCEPTION, t);
            }
        } else { //impl mock
            //mock实现类的方式
            try {
                Invoker<T> invoker = getInvoker(mock);
                //调用mock实例
                return invoker.invoke(invocation);
            } catch (Throwable t) {
                throw new RpcException("Failed to create mock implementation class " + mock,
  t);
            }
        }
  }
```

我们看一下normalizeMock方法

```
  public static String normalizeMock(String mock) {
      if (mock == null) {
          return mock;
      }

      mock = mock.trim();

      if (mock.length() == 0) {
          return mock;
      }

      //如果是只有一个return 则加上一个return null
      if (RETURN_KEY.equalsIgnoreCase(mock)) {
          return RETURN_PREFIX + "null";
```

```
    }

    if (ConfigUtils.isDefault(mock) || "fail".equalsIgnoreCase(mock) ||
"force".equalsIgnoreCase(mock)) {
        return "default";
    }

    if (mock.startsWith(FAIL_PREFIX)) {
        mock = mock.substring(FAIL_PREFIX.length()).trim();
    }

    //把force:去掉
    if (mock.startsWith(FORCE_PREFIX)) {
        mock = mock.substring(FORCE_PREFIX.length()).trim();
    }

    if (mock.startsWith(RETURN_PREFIX) || mock.startsWith(THROW_PREFIX)) {
        mock = mock.replace('`', '"');
    }

    return mock;
}
```

我们获取到mock的返回值内容后需要把该返回值包装成方法返回值类型，所以这里必须要有一个返回值类型的包装，我们看一下parseMockValue方法：

```
public static Object parseMockValue(String mock, Type[] returnTypes) throws Exception {
    Object value = null;
    if ("empty".equals(mock)) {
        value = ReflectUtils.getEmptyObject(returnTypes != null && returnTypes.length > 0 ?
(Class<?>) returnTypes[0] : null);
    } else if ("null".equals(mock)) {
        value = null;
    } else if ("true".equals(mock)) {
        value = true;
    } else if ("false".equals(mock)) {
        value = false;
    } else if (mock.length() >= 2 && (mock.startsWith("\"") && mock.endsWith("\"")
            || mock.startsWith("\'") && mock.endsWith("\'"))) {
        value = mock.subSequence(1, mock.length() - 1);
    } else if (returnTypes != null && returnTypes.length > 0 && returnTypes[0] ==
String.class) {
        value = mock;
    } else if (StringUtils.isNumeric(mock, false)) {
        value = JSON.parse(mock);
    } else if (mock.startsWith("{")) {
        value = JSON.parseObject(mock, Map.class);
    } else if (mock.startsWith("[")) {
        value = JSON.parseObject(mock, List.class);
    } else {
        value = mock;
    }
    if (ArrayUtils.isNotEmpty(returnTypes)) {
        value = PojoUtils.realize(value, (Class<?>) returnTypes[0], returnTypes.length > 1
? returnTypes[1] : null);
    }
    return value;
}
```

从上面的逻辑来看，如果是force:return 则会不走rpc直接返回一个结果，然后把这个结果包装成方法的返回值类型。

2、调用流程

2.1、消费端请求流程

下面我们来从大局分析消费端的整个请求流程，前面我们分析过消费端用代理对象调用方法时，最终会来到InvokerInvocationHandler中，那么我们就从这个类开始看起。先掉到该类的invoker方法，该类持有了MigrationInvoker类的引用。

2.1.1、InvokerInvocationHandler

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if (method.getDeclaringClass() == Object.class) {
        return method.invoke(invoker, args);
    }
    String methodName = method.getName();
    Class<?>[] parameterTypes = method.getParameterTypes();
    if (parameterTypes.length == 0) {
        if ("toString".equals(methodName)) {
            return invoker.toString();
        } else if ("$destroy".equals(methodName)) {
            invoker.destroy();
            return null;
        } else if ("hashCode".equals(methodName)) {
            return invoker.hashCode();
        }
    } else if (parameterTypes.length == 1 && "equals".equals(methodName)) {
        return invoker.equals(args[0]);
    }
    RpcInvocation rpcInvocation = new RpcInvocation(method,
invoker.getInterface().getName(), protocolServiceKey, args);
    String serviceKey = url.getServiceKey();
    rpcInvocation.setTargetServiceUniqueName(serviceKey);

    // invoker.getUrl() returns consumer url.
    RpcServiceContext.setRpcContext(url);

    if (consumerModel != null) {
        rpcInvocation.put(Constants.CONSUMER_MODEL, consumerModel);
        rpcInvocation.put(Constants.METHOD_MODEL, consumerModel.getMethodModel(method));
    }
    //掉到 MigrationInvoker
    return invoker.invoke(rpcInvocation).recreate();
}
```

2.1.2、MigrationInvoker

```
@Override
public Result invoke(Invocation invocation) throws RpcException {
    if (currentAvailableInvoker != null) {
        if (step == APPLICATION_FIRST) {
            // call ratio calculation based on random value
            if (ThreadLocalRandom.current().nextDouble(100) > promotion) {
                return invoker.invoke(invocation);
            }
        }
        //代码会走这里，走到MockClusterInvoker中
        return currentAvailableInvoker.invoke(invocation);
    }

    switch (step) {
        case APPLICATION_FIRST:
            if (checkInvokerAvailable(serviceDiscoveryInvoker)) {
                currentAvailableInvoker = serviceDiscoveryInvoker;
            } else if (checkInvokerAvailable(invoker)) {
                currentAvailableInvoker = invoker;
            } else {
                currentAvailableInvoker = serviceDiscoveryInvoker;
            }
            break;
        case FORCE_APPLICATION:
            currentAvailableInvoker = serviceDiscoveryInvoker;
            break;
        case FORCE_INTERFACE:
        default:
            currentAvailableInvoker = invoker;
    }

    return currentAvailableInvoker.invoke(invocation);
}
```

2.1.3、MockClusterInvoker

在该invoker中进行了服务的降级逻辑的处理

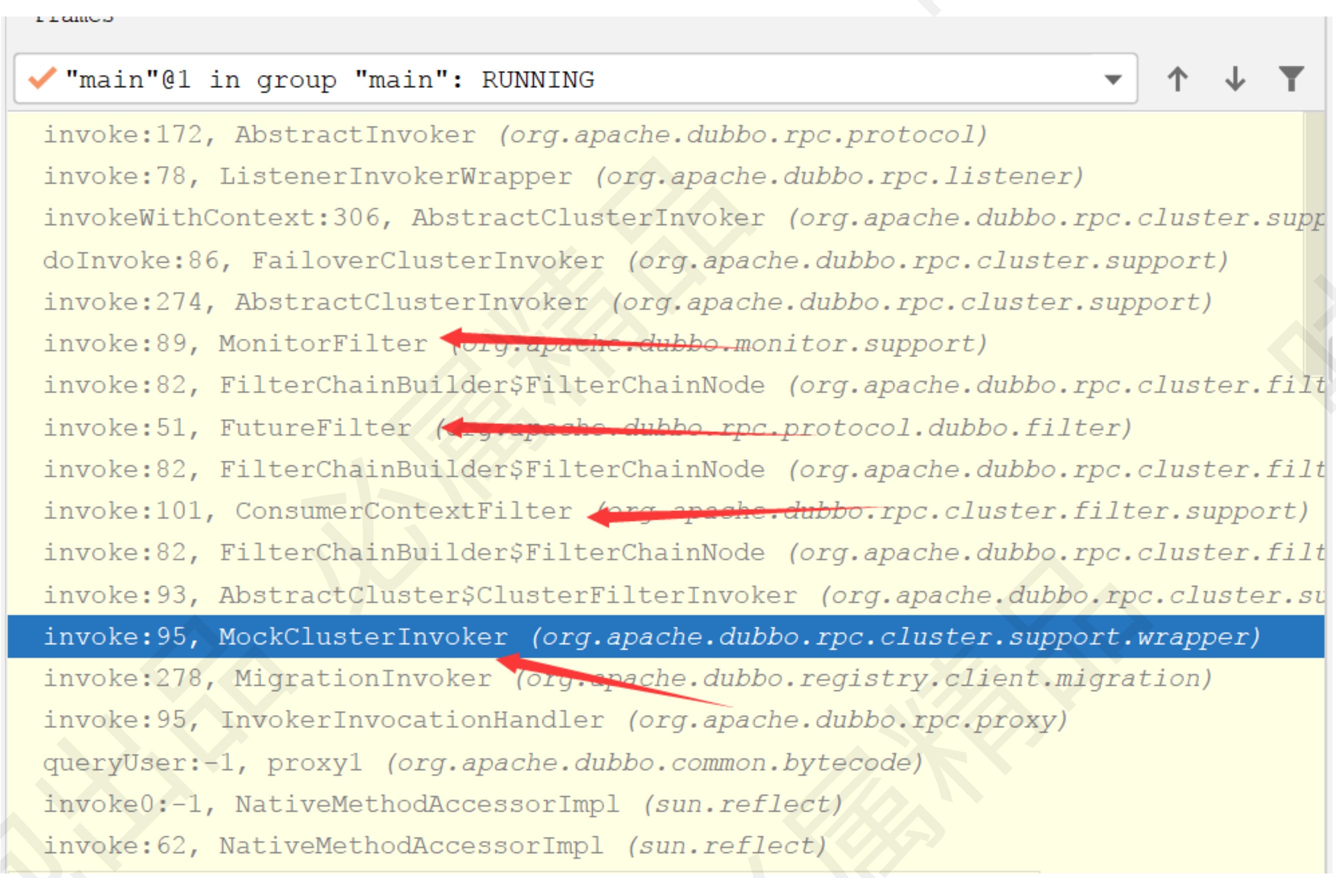
```
@Override
public Result invoke(Invocation invocation) throws RpcException {
    Result result = null;

    //获取url中的mock参数
    String value = getUrl().getMethodParameter(invocation.getMethodName(), MOCK_KEY,
Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || "false".equalsIgnoreCase(value)) {
        //no mock
        //如果没mock则直接走后面调用逻辑
        result = this.invoker.invoke(invocation);
    } else if (value.startsWith("force")) {
        //如果是force开头
        if (logger.isWarnEnabled()) {
            logger.warn("force-mock: " + invocation.getMethodName() + " force-mock enabled
, url : " + getUrl());
        }
        //强制降级，调用mock的实现类逻辑
        //force:direct mock
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock
        try {
            result = this.invoker.invoke(invocation);

            //fix:#4585
            if(result.getException() != null && result.getException() instanceof
RpcException){
                RpcException rpcException= (RpcException)result.getException();
                if(rpcException.isBiz()){
                    throw  rpcException;
                }else {
                    result = doMockInvoke(invocation, rpcException);
                }
            }

        } catch (RpcException e) {
            if (e.isBiz()) {
                throw e;
            }

            if (logger.isWarnEnabled()) {
                logger.warn("fail-mock: " + invocation.getMethodName() + " fail-mock
enabled , url : " + getUrl(), e);
            }
            result = doMockInvoke(invocation, e);
        }
    }
    return result;
}
```

2.1.4、FilterChainNode

经过消费端过滤器处理

```
class FilterChainNode<T, TYPE extends Invoker<T>, FILTER extends BaseFilter> implements
Invoker<T>{
    TYPE originalInvoker;
    Invoker<T> nextNode;
    FILTER filter;

    public FilterChainNode(TYPE originalInvoker, Invoker<T> nextNode, FILTER filter) {
        this.originalInvoker = originalInvoker;
```

```java
        this.nextNode = nextNode;
        this.filter = filter;
    }

    public TYPE getOriginalInvoker() {
        return originalInvoker;
    }

    @Override
    public Class<T> getInterface() {
        return originalInvoker.getInterface();
    }

    @Override
    public URL getUrl() {
        return originalInvoker.getUrl();
    }

    @Override
    public boolean isAvailable() {
        return originalInvoker.isAvailable();
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        Result asyncResult;
        try {
            asyncResult = filter.invoke(nextNode, invocation);
        } catch (Exception e) {
            if (filter instanceof ListenableFilter) {
                ListenableFilter listenableFilter = ((ListenableFilter) filter);
                try {
                    Filter.Listener listener = listenableFilter.listener(invocation);
                    if (listener != null) {
                        listener.onError(e, originalInvoker, invocation);
                    }
                } finally {
                    listenableFilter.removeListener(invocation);
                }
            } else if (filter instanceof FILTER.Listener) {
                FILTER.Listener listener = (FILTER.Listener) filter;
                listener.onError(e, originalInvoker, invocation);
            }
            throw e;
        } finally {

        }
        return asyncResult.whenCompleteWithContext((r, t) -> {
            if (filter instanceof ListenableFilter) {
                ListenableFilter listenableFilter = ((ListenableFilter) filter);
                Filter.Listener listener = listenableFilter.listener(invocation);
                try {
                    if (listener != null) {
                        if (t == null) {
                            listener.onResponse(r, originalInvoker, invocation);
                        } else {
                            listener.onError(t, originalInvoker, invocation);
                        }
                    }
                } finally {
                    listenableFilter.removeListener(invocation);
                }
            } else if (filter instanceof FILTER.Listener) {
                FILTER.Listener listener = (FILTER.Listener) filter;
                if (t == null) {
                    listener.onResponse(r, originalInvoker, invocation);
                } else {
                    listener.onError(t, originalInvoker, invocation);
                }
            }
        });
    }

    @Override
    public void destroy() {
        originalInvoker.destroy();
    }

    @Override
    public String toString() {
        return originalInvoker.toString();
    }
}
```



```
"main"@1 in group "main": RUNNING
invoke:172, AbstractInvoker (org.apache.dubbo.rpc.protocol)
invoke:78, ListenerInvokerWrapper (org.apache.dubbo.rpc.listener)
invokeWithContext:306, AbstractClusterInvoker (org.apache.dubbo.rpc.cluster.supp
doInvoke:86, FailoverClusterInvoker (org.apache.dubbo.rpc.cluster.support)
invoke:274, AbstractClusterInvoker (org.apache.dubbo.rpc.cluster.support)
invoke:89, MonitorFilter (org.apache.dubbo.monitor.support)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filt
invoke:51, FutureFilter (org.apache.dubbo.rpc.protocol.dubbo.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filt
invoke:101, ConsumerContextFilter (org.apache.dubbo.rpc.cluster.filter.support)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filt
invoke:93, AbstractCluster$ClusterFilterInvoker (org.apache.dubbo.rpc.cluster.su
invoke:95, MockClusterInvoker (org.apache.dubbo.rpc.cluster.support.wrapper)
invoke:278, MigrationInvoker (org.apache.dubbo.registry.client.migration)
invoke:95, InvokerInvocationHandler (org.apache.dubbo.rpc.proxy)
queryUser:-1, proxy1 (org.apache.dubbo.common.bytecode)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
```

从断点的栈帧来看，这里经过了3个filter，分别是，ConsumerContextFilter、FutureFilter、MonitorFilter。

2.1.5、AbstractClusterInvoker

然后代码走到了FailoverClusterInvoker的父类的invoke方法，里面定义了核心的流程

```java
@Override
public Result invoke(final Invocation invocation) throws RpcException {
    //判断是否销毁
    checkWhetherDestroyed();

    // binding attachments into invocation.
    //        Map<String, Object> contextAttachments =
RpcContext.getClientAttachment().getObjectAttachments();
    //        if (contextAttachments != null && contextAttachments.size() != 0) {
    //            ((RpcInvocation)
invocation).addObjectAttachmentsIfAbsent(contextAttachments);
    //        }

    //获取服务列表
    List<Invoker<T>> invokers = list(invocation);
    //spi 获取负载均衡类实例
    LoadBalance loadbalance = initLoadBalance(invokers, invocation);
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    return doInvoke(invocation, invokers, loadbalance);
}
```

在父类中定义了核心的流程，比如，获取服务列表，获取负载均衡器，然后掉钩子方法doInvoke，这里是一个典型的模板设计模式，不同的子类实例doInvoke掉到的是不同的子类逻辑，该方法是一个抽象的钩子方法。

2.1.6、FailoverClusterInvoker

doInvoke方法勾到了FailoverClusterInvoker类中，这个是默认的集群容错，在该类中定义了如何重试的策略，如果出现了RpcException那么就会重试。再一次调用。

```java
public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    List<Invoker<T>> copyInvokers = invokers;
    //invokers校验
    checkInvokers(copyInvokers, invocation);
    String methodName = RpcUtils.getMethodName(invocation);
    //计算调用次数
    int len = calculateInvokeTimes(methodName);
    // retry loop.
    RpcException le = null; // last exception.
    //记录已经调用过了的服务列表
    List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyInvokers.size()); // invoked
invokers.
    Set<String> providers = new HashSet<String>(len);
    for (int i = 0; i < len; i++) {
        //Reselect before retry to avoid a change of candidate `invokers`.
        //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
        //如果掉完一次后，服务列表更新了，再次获取服务列表
        if (i > 0) {
            checkWhetherDestroyed();
            copyInvokers = list(invocation);
            // check again
            checkInvokers(copyInvokers, invocation);
        }
        //根据负载均衡算法，选择一个服务调用
        Invoker<T> invoker = select(loadbalance, invocation, copyInvokers, invoked);
        //记录已经调用过的invoker
        invoked.add(invoker);
        RpcContext.getServiceContext().setInvokers((List) invoked);
        try {
            //具体的服务调用逻辑
            Result result = invokeWithContext(invoker, invocation);
            if (le != null && logger.isWarnEnabled()) {
                logger.warn("Although retry the method " + methodName
                        + " in the service " + getInterface().getName()
                        + " was successful by the provider " +
invoker.getUrl().getAddress()
                        + ", but there have been failed providers " + providers
                        + " (" + providers.size() + "/" + copyInvokers.size()
                        + ") from the registry " + directory.getUrl().getAddress()
                        + " on the consumer " + NetUtils.getLocalHost()
                        + " using the dubbo version " + Version.getVersion() + ". Last
error is: "
                        + le.getMessage(), le);
            }
```

```
                return result;
            } catch (RpcException e) {
                if (e.isBiz()) { // biz exception.
                    throw e;
                }
                le = e;
            } catch (Throwable e) {
                le = new RpcException(e.getMessage(), e);
            } finally {
                providers.add(invoker.getUrl().getAddress());
            }
        }
        throw new RpcException(le.getCode(), "Failed to invoke the method "
                + methodName + " in the service " + getInterface().getName()
                + ". Tried " + len + " times of the providers " + providers
                + " (" + providers.size() + "/" + copyInvokers.size()
                + ") from the registry " + directory.getUrl().getAddress()
                + " on the consumer " + NetUtils.getLocalHost() + " using the dubbo version "
                + Version.getVersion() + ". Last error is: "
                + le.getMessage(), le.getCause() != null ? le.getCause() : le);
}
```

2.1.7、AbstractInvoker

接着来到了DubboInvoker的父类逻辑

```
@Override
public Result invoke(Invocation inv) throws RpcException {
    // if invoker is destroyed due to address refresh from registry, let's allow the
current invoke to proceed
    if (isDestroyed()) {
        logger.warn("Invoker for service " + this + " on consumer " +
NetUtils.getLocalHost() + " is destroyed, "
                + ", dubbo version is " + Version.getVersion() + ", this invoker should not
be used any longer");
    }

    RpcInvocation invocation = (RpcInvocation) inv;

    //调用 invocation初始化
    // prepare rpc invocation
    prepareInvocation(invocation);

    //具体的rpc调用流程
    // do invoke rpc invocation and return async result
    AsyncRpcResult asyncResult = doInvokeAndReturn(invocation);

    //阻塞拿返回结果
    // wait rpc result if sync
    waitForResultIfSync(asyncResult, invocation);

    return asyncResult;
}
```

在doInvokeAndReturn方法中是走了远程rpc调用，然后在waitForResultIfSync里面如果是同步调用就掉了CompletableFuture的get方法完成了调用阻塞，这里就是异步调用然后用get的方式阻塞，异步调用框架的同步阻塞改造。我们看看waitForResultIfSync(asyncResult, invocation);方法。

```
private void waitForResultIfSync(AsyncRpcResult asyncResult, RpcInvocation invocation) {
    //如果非同步调用，要直接返回asyncResult
    if (InvokeMode.SYNC != invocation.getInvokeMode()) {
        return;
    }
    try {
        /*
         * NOTICE!
         * must call {@link java.util.concurrent.CompletableFuture#get(long, TimeUnit)}
because
         * {@link java.util.concurrent.CompletableFuture#get()} was proved to have serious
performance drop.
         */
        //同步调用，其实是在这里阻塞拿返回结果
        asyncResult.get(Integer.MAX_VALUE, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        throw new RpcException("Interrupted unexpectedly while waiting for remote result to
return! method: " +
                invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
    } catch (ExecutionException e) {
        Throwable rootCause = e.getCause();
        if (rootCause instanceof TimeoutException) {
            throw new RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke remote method
timeout. method: " +
                    invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
        } else if (rootCause instanceof RemotingException) {
            throw new RpcException(RpcException.NETWORK_EXCEPTION, "Failed to invoke remote
method: " +
                    invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
        } else {
            throw new RpcException(RpcException.UNKNOWN_EXCEPTION, "Fail to invoke remote
method: " +
                    invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
        }
    } catch (Throwable e) {
        throw new RpcException(e.getMessage(), e);
    }
}
```

2.1.8、DubboInvoker

接着我们看看DubboInvoker的远程调用逻辑，这个是父类的方法，同样是模板设计模式doInvoke是钩子方法，勾到了子类DubboInvoker的逻辑，

```java
private AsyncRpcResult doInvokeAndReturn(RpcInvocation invocation) {
    AsyncRpcResult asyncResult;
    try {
        //调用核心代码
        asyncResult = (AsyncRpcResult) doInvoke(invocation);
    } catch (InvocationTargetException e) {
        Throwable te = e.getTargetException();
        if (te != null) {
            // if biz exception
            if (te instanceof RpcException) {
                ((RpcException) te).setCode(RpcException.BIZ_EXCEPTION);
            }
            asyncResult = AsyncRpcResult.newDefaultAsyncResult(null, te, invocation);
        } else {
            asyncResult = AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
        }
    } catch (RpcException e) {
        // if biz exception
        if (e.isBiz()) {
            asyncResult = AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
        } else {
            throw e;
        }
    } catch (Throwable e) {
        asyncResult = AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
    }

    // set server context
    RpcContext.getServiceContext().setFuture(new FutureAdapter<>
(asyncResult.getResponseFuture()));

    return asyncResult;
}
```

我们看看DubboInvoker中的doInvoke逻辑

```java
@Override
protected Result doInvoke(final Invocation invocation) throws Throwable {
    RpcInvocation inv = (RpcInvocation) invocation;
    final String methodName = RpcUtils.getMethodName(invocation);
    inv.setAttachment(PATH_KEY, getUrl().getPath());
    inv.setAttachment(VERSION_KEY, version);

    ExchangeClient currentClient;
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
    try {
        //如果是单工通讯
        //@Method(name = "doKill",isReturn = false) 这样配置就是单工通讯
        boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
        int timeout = calculateTimeout(invocation, methodName);
        invocation.put(TIMEOUT_KEY, timeout);
        //单工就不需要等待返回结果
        if (isOneway) {
            boolean isSent = getUrl().getMethodParameter(methodName, Constants.SENT_KEY,
false);
            currentClient.send(inv, isSent);
            return AsyncRpcResult.newDefaultAsyncResult(invocation);
        } else {
            ExecutorService executor = getCallbackExecutor(getUrl(), inv);
            //request具体的rpc远程调用
            CompletableFuture<AppResponse> appResponseFuture =
                    currentClient.request(inv, timeout, executor).thenApply(obj ->
(AppResponse) obj);
            // save for 2.6.x compatibility, for example, TraceFilter in Zipkin uses
com.alibaba.xxx.FutureAdapter
            FutureContext.getContext().setCompatibleFuture(appResponseFuture);
            AsyncRpcResult result = new AsyncRpcResult(appResponseFuture, inv);
            result.setExecutor(executor);
            return result;
        }
    } catch (TimeoutException e) {
        throw new RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke remote method
timeout. method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
    } catch (RemotingException e) {
        throw new RpcException(RpcException.NETWORK_EXCEPTION, "Failed to invoke remote
method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " +
e.getMessage(), e);
    }
}
```

我们以双工调用为例来分析调用过程。

CompletableFuture appResponseFuture = currentClient.request(inv, timeout, executor).thenApply(obj -> (AppResponse) obj);

这行代码就是调用后端逻辑代码，我们看看request方法。

2.1.9、HeaderExchangeChannel

```
@Override
public CompletableFuture<Object> request(Object request, int timeout, ExecutorService
executor) throws RemotingException {
    if (closed) {
        throw new RemotingException(this.getLocalAddress(), null, "Failed to send request "
+ request + ", cause: The channel " + this + " is closed!");
    }
    // create request.
    //创建了Request对象，并生了一个流水id
    Request req = new Request();
    req.setVersion(Version.getProtocolVersion());
    req.setTwoWay(true);
    req.setData(request);
    DefaultFuture future = DefaultFuture.newFuture(channel, req, timeout, executor);
    try {
        channel.send(req);
    } catch (RemotingException e) {
        future.cancel();
        throw e;
    }
    return future;
}
```

```
public Request() {
    mId = newId();
}

private static long newId() {
    // getAndIncrement() When it grows to MAX_VALUE, it will grow to MIN_VALUE, and the
negative can be used as ID
    return INVOKE_ID.getAndIncrement();
}
```

DefaultFuture future = DefaultFuture.newFuture(channel, req, timeout, executor);该方法创建了一个CompletableFuture对象，并把对象缓存了起来，缓存的key就是刚刚生成Request对象的id，这个id会传递给消费端，然后消费端响应的时候又会传递回来的，然后会从缓存中根据传递回来的id拿到对应的CompletableFuture对象然后把阻塞的地方唤醒。

```
private DefaultFuture(Channel channel, Request request, int timeout) {
    this.channel = channel;
    this.request = request;
    //这里是生成请求流水，请求和响应的流水id
    this.id = request.getId();
    this.timeout = timeout > 0 ? timeout :
channel.getUrl().getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
    // put into waiting map.
    //通过id来找对应请求的future对象
    FUTURES.put(id, this);
    CHANNELS.put(id, channel);
}
```

2.1.10、NettyChannel

```
@Override
public void send(Object message, boolean sent) throws RemotingException {
    // whether the channel is closed
    super.send(message, sent);

    boolean success = true;
    int timeout = 0;
    try {
        //netty通讯，把请求发送到服务端
```

```
            ChannelFuture future = channel.writeAndFlush(message);
            if (sent) {
                // wait timeout ms
                timeout = getUrl().getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
                success = future.await(timeout);
            }
            Throwable cause = future.cause();
            if (cause != null) {
                throw cause;
            }
        } catch (Throwable e) {
            removeChannelIfDisconnected(channel);
            throw new RemotingException(this, "Failed to send message " +
PayloadDropper.getRequestWithoutData(message) + " to " + getRemoteAddress() + ", cause: " +
e.getMessage(), e);
        }
        if (!success) {
            throw new RemotingException(this, "Failed to send message " +
PayloadDropper.getRequestWithoutData(message) + " to " + getRemoteAddress()
                    + "in timeout(" + timeout + "ms) limit");
        }
    }
```

从上述描述可以看出，客户端的请求流程其实会有一个get方法的阻塞动作，然后把请求通过netty通讯的方式发送给服务端，然后有一个Request对象请求到了服务端，Request对象中有一个流水id，建立了一个流水id和CompletableFuture对象的映射关系，到时候会从这个映射关系中获取到CompletableFuture对象的。

2.2、服务端接收流程

服务的接收，前面分析过Netty的服务端有一个核心的处理handler，这个类就是NettyServerHandler，当客户端请求过来时，请求就会掉到channelRead方法，我们来看看这个方法：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);
    handler.received(channel, msg);
}
```

前面我们在服务的暴露流程那节课中我们分析了，这个handler是一个层级关系的handler，我们来分析一下这个层级调用。

2.2.1、MultiMessageHandler

这个是对不同类型的消息进行处理

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    if (message instanceof MultiMessage) {
        MultiMessage list = (MultiMessage) message;
        for (Object obj : list) {
            try {
                handler.received(channel, obj);
            } catch (ExecutionException e) {
                logger.error("MultiMessageHandler received fail.", e);
                handler.caught(channel, e);
            }
        }
    } else {
        handler.received(channel, message);
    }
}
```

2.2.2、HeartbeatHandler

这个是心跳的handler，设置读消息的时间戳用于在发送心跳的时候获取到最后读的时候跟当前时间对比，如果超过了心跳时间间隔设置就发送心跳消息。然后如果收到的请求是一个心跳请求则发送应答消息给对端系统。

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    //设置读数据的时间戳
    setReadTimestamp(channel);
    //如果是心跳请求
    if (isHeartbeatRequest(message)) {
        Request req = (Request) message;
        if (req.isTwoWay()) {
            //应答消息
            Response res = new Response(req.getId(), req.getVersion());
            res.setEvent(HEARTBEAT_EVENT);
            //发送应答给对方
            channel.send(res);
            if (logger.isInfoEnabled()) {
                int heartbeat = channel.getUrl().getParameter(Constants.HEARTBEAT_KEY, 0);
                if (logger.isDebugEnabled()) {
                    logger.debug("Received heartbeat from remote channel " +
channel.getRemoteAddress()
                            + ", cause: The channel has no data-transmission exceeds a
heartbeat period"
                            + (heartbeat > 0 ? ": " + heartbeat + "ms" : ""));
                }
            }
        }
        return;
    }
    //如果是收到的心跳应答消息
    if (isHeartbeatResponse(message)) {
        if (logger.isDebugEnabled()) {
            logger.debug("Receive heartbeat response in thread " +
Thread.currentThread().getName());
        }
        return;
    }
    handler.received(channel, message);
}
```

2.2.3、AllChannelHandler

这个是框架线程池隔离类，主要目的是把io线程转换成框架线程池，这样做的目的可以释放io线程，让io线程去处理其他用户的请求，提高整个系统的吞吐量。

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    ExecutorService executor = getPreferredExecutorService(message);
    try {
        executor.execute(new ChannelEventRunnable(channel, handler, ChannelState.RECEIVED,
message));
    } catch (Throwable t) {
        if(message instanceof Request && t instanceof RejectedExecutionException){
            sendFeedback(channel, (Request) message, t);
            return;
        }
        throw new ExecutionException(message, channel, getClass() + " error when process
received event .", t);
    }
}
```

可以看到，这里开启了线程池，我们都知道线程池也有线程和队列大小的，如果超过了这个大小是会抛出RejectedExecutionException的，可以看到代码是有在catch里面判断该异常的，如果有该异常出现则会发送应答消息给客户端的。告诉客户端服务端的线程池已经满了。

```
protected void sendFeedback(Channel channel, Request request, Throwable t) throws
RemotingException {
    if (request.isTwoWay()) {
        String msg = "Server side(" + url.getIp() + "," + url.getPort()
                + ") thread pool is exhausted, detail msg:" + t.getMessage();
        Response response = new Response(request.getId(), request.getVersion());
        response.setStatus(Response.SERVER_THREADPOOL_EXHAUSTED_ERROR);
        response.setErrorMessage(msg);
        channel.send(response);
        return;
    }
}
```

我们再看看线程类ChannelEventRunnable

```
@Override
public void run() {
    //如果是接受请求
    if (state == ChannelState.RECEIVED) {
        try {
            handler.received(channel, message);
        } catch (Exception e) {
            logger.warn("ChannelEventRunnable handle " + state + " operation error, channel
is " + channel
                    + ", message is " + message, e);
        }
    } else {
        switch (state) {
        case CONNECTED:
            try {
                handler.connected(channel);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error,
channel is " + channel, e);
            }
            break;
        case DISCONNECTED:
            try {
                handler.disconnected(channel);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error,
channel is " + channel, e);
            }
            break;
        case SENT:
            try {
                handler.sent(channel, message);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error,
channel is " + channel
                        + ", message is " + message, e);
            }
            break;
        case CAUGHT:
            try {
                handler.caught(channel, exception);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error,
channel is " + channel
                        + ", message is: " + message + ", exception is " + exception, e);
            }
            break;
        default:
            logger.warn("unknown state: " + state + ", message is " + message);
        }
    }

}
```

2.2.4、DecodeHandler

该handler主要是对请求过来的数据进行编解码序列化的，把网络传输过来的数据编程业务数据。默认采用的是用Hessian2做的序列化。序列化过程就是用流的方式编解码这里就不做过多的分析。

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    if (message instanceof Decodeable) {
        decode(message);
    }
    //如果是请求消息
    if (message instanceof Request) {
        decode(((Request) message).getData());
    }

    //如果是响应消息
    if (message instanceof Response) {
        decode(((Response) message).getResult());
    }

    handler.received(channel, message);
}
```

2.2.5、HeaderExchangeHandler

接着来到了消息交换层的handler逻辑，在之类对接收到消息调用了被代理方法，然后根据被代理方法的返回值回调了调用方的逻辑。

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    final ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
    //服务端接收到的请求消息
    if (message instanceof Request) {
        // handle request.
        Request request = (Request) message;
        if (request.isEvent()) {
            handlerEvent(channel, request);
        } else {
            //如果是双工通讯
            if (request.isTwoWay()) {
                //核心代码
                handleRequest(exchangeChannel, request);
            } else {
                handler.received(exchangeChannel, request.getData());
            }
        }
    } else if (message instanceof Response) {
        //如果是客户端接收到的响应消息
        handleResponse(channel, (Response) message);
    } else if (message instanceof String) {
        if (isClientSide(channel)) {
            Exception e = new Exception("Dubbo client can not supported string message: " +
message + " in channel: " + channel + ", url: " + channel.getUrl());
            logger.error(e.getMessage(), e);
        } else {
            String echo = handler.telnet(channel, (String) message);
            if (echo != null && echo.length() > 0) {
                channel.send(echo);
            }
        }
    } else {
        handler.received(exchangeChannel, message);
    }
}
```

我们可以看到核心代码就是handleRequest(exchangeChannel, request);，我们看看该方法

```
void handleRequest(final ExchangeChannel channel, Request req) throws RemotingException {
    //这里注意，id是从request对象中获取到的id，这个id响应数据时是需要返回给客户端的
    Response res = new Response(req.getId(), req.getVersion());
    if (req.isBroken()) {
        Object data = req.getData();

        String msg;
        if (data == null) {
            msg = null;
        } else if (data instanceof Throwable) {
            msg = StringUtils.toString((Throwable) data);
        } else {
            msg = data.toString();
        }
        res.setErrorMessage("Fail to decode request due to: " + msg);
        res.setStatus(Response.BAD_REQUEST);

        channel.send(res);
        return;
    }
    //获取到编解码后的数据
    // find handler by message class.
    Object msg = req.getData();
    try {
        //调用DubboProtocol中的ExchangeHandlerAdapter中的reply方法进行后续的filter调用
        CompletionStage<Object> future = handler.reply(channel, msg);
        //future完成后回调客户端，响应数据回去
        future.whenComplete((appResult, t) -> {
            try {
                if (t == null) {
                    res.setStatus(Response.OK);
                    res.setResult(appResult);
                } else {
                    res.setStatus(Response.SERVICE_ERROR);
                    res.setErrorMessage(StringUtils.toString(t));
                }
                //回调客户端响应数据
                channel.send(res);
            } catch (RemotingException e) {
                logger.warn("Send result to consumer failed, channel is " + channel + ", 
msg is " + e);
            }
        });
    } catch (Throwable e) {
        res.setStatus(Response.SERVICE_ERROR);
        res.setErrorMessage(StringUtils.toString(e));
        channel.send(res);
    }
}
```

我们可以看到创建了一个Response对象，对象的id就是Request对象传递过来的id，

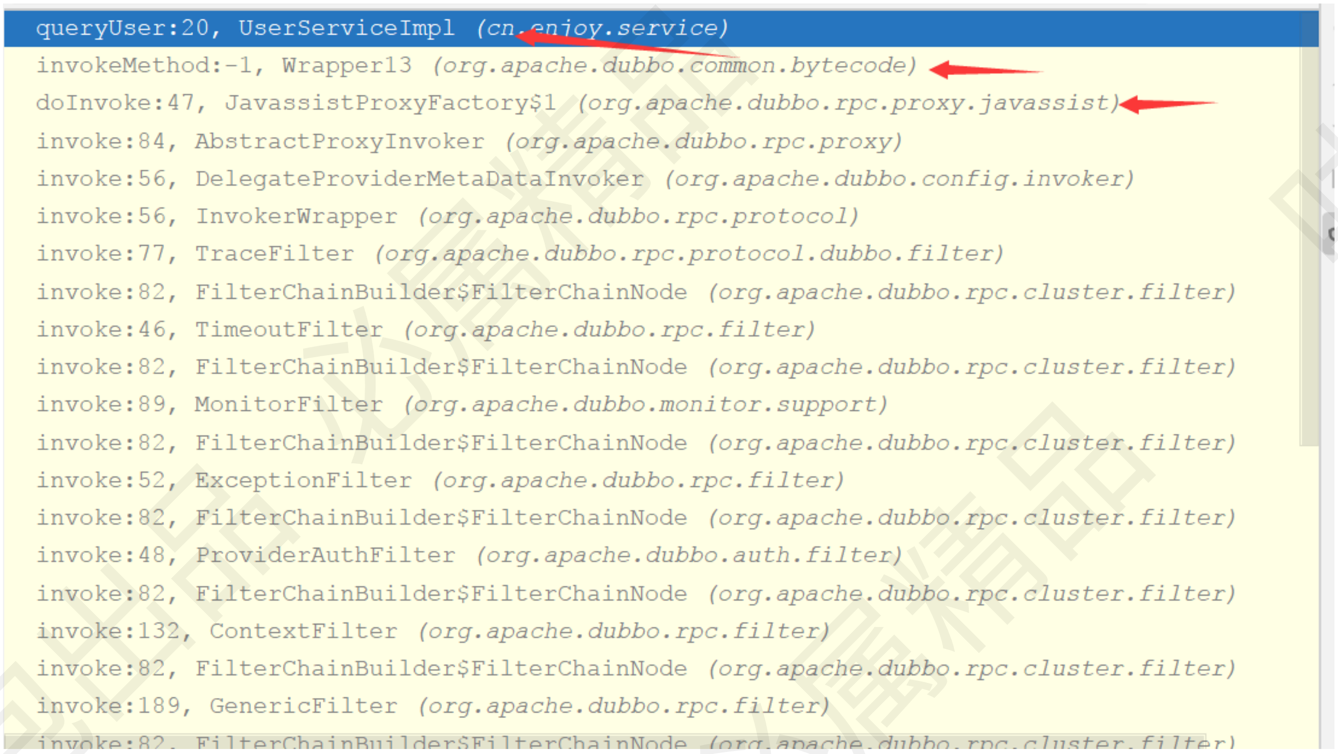
Response res = new Response(req.getId(), req.getVersion());

然后调用了这行代码CompletionStage future = handler.reply(channel, msg);

前面我们分析过，这个reply方法会调到DubboProtocol中的一个内部类，我们看看这个reply方法。

```
@Override
public CompletableFuture<Object> reply(ExchangeChannel channel, Object message) throws
RemotingException {

    if (!(message instanceof Invocation)) {
        throw new RemotingException(channel, "Unsupported request: "
                + (message == null ? null : (message.getClass().getName() + ": " +
message))
                + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider: "
+ channel.getLocalAddress());
    }

    Invocation inv = (Invocation) message;
    //获取到invoker，这个invoker就是FilterChainNode
    Invoker<?> invoker = getInvoker(channel, inv);
    // need to consider backward-compatibility if it's a callback
    if
(Boolean.TRUE.toString().equals(inv.getObjectAttachments().get(IS_CALLBACK_SERVICE_INVOKE))
) {
        String methodsStr = invoker.getUrl().getParameters().get("methods");
        boolean hasMethod = false;
        if (methodsStr == null || !methodsStr.contains(",")) {
            hasMethod = inv.getMethodName().equals(methodsStr);
        } else {
            String[] methods = methodsStr.split(",");
            for (String method : methods) {
                if (inv.getMethodName().equals(method)) {
                    hasMethod = true;
                    break;
                }
            }
        }
        if (!hasMethod) {
            logger.warn(new IllegalStateException("The methodName " + inv.getMethodName()
                    + " not found in callback service interface ,invoke will be ignored."
                    + " please update the api interface. url is:"
                    + invoker.getUrl()) + " ,invocation is :" + inv);
            return null;
        }
    }
    RpcContext.getServiceContext().setRemoteAddress(channel.getRemoteAddress());
    Result result = invoker.invoke(inv);
    return result.thenApply(Function.identity());
}
```

这个方法简单来讲就是掉了生产方的FilterChainNode逻辑，最后掉到了被代理方法，为什么能掉到被代理方法前面分析过，这里就不再赘述了。我们来看看调用栈帧


```
queryUser:20, UserServiceImpl (cn.enjoy.service)
invokeMethod:-1, Wrapper13 (org.apache.dubbo.common.bytecode)
doInvoke:47, JavassistProxyFactory$1 (org.apache.dubbo.rpc.proxy.javassist)
invoke:84, AbstractProxyInvoker (org.apache.dubbo.rpc.proxy)
invoke:56, DelegateProviderMetaDataInvoker (org.apache.dubbo.config.invoker)
invoke:56, InvokerWrapper (org.apache.dubbo.rpc.protocol)
invoke:77, TraceFilter (org.apache.dubbo.rpc.protocol.dubbo.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:46, TimeoutFilter (org.apache.dubbo.rpc.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:89, MonitorFilter (org.apache.dubbo.monitor.support)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:52, ExceptionFilter (org.apache.dubbo.rpc.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:48, ProviderAuthFilter (org.apache.dubbo.auth.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:132, ContextFilter (org.apache.dubbo.rpc.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
invoke:189, GenericFilter (org.apache.dubbo.rpc.filter)
invoke:82, FilterChainBuilder$FilterChainNode (org.apache.dubbo.rpc.cluster.filter)
```

调用完成以后就要回调调用方了

```
future.whenComplete((appResult, t) -> {
    try {
        if (t == null) {
            res.setStatus(Response.OK);
            res.setResult(appResult);
        } else {
            res.setStatus(Response.SERVICE_ERROR);
            res.setErrorMessage(StringUtils.toString(t));
        }
        //回调客户端响应数据
        channel.send(res);
    } catch (RemotingException e) {
        logger.warn("Send result to consumer failed, channel is " + channel + ", msg is " +
e);
    }
});
```

从上面的分析来看，生产方经过了一些列handler的处理，最终掉到了被代理逻辑，然后响应到了客户端了。

2.3、客户端接收响应

服务端处理完成后，服务端会把Response对象响应给客户端，客户端接收到响应以后接着处理，前面的课程我们分析了，netty客户端的核心handler是，NettyClientHandler，一样响应过来后会掉到channelRead方法。

在客户端接收到了Response对象后，一样的要经过服务的一些列handler的逻辑，这里我就不再赘述了。

这里我们只着重分析一下HeaderExchangeHandler

2.3.1、HeaderExchangeHandler

```java
@Override
public void received(Channel channel, Object message) throws RemotingException {
    final ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
    //服务端接收到的请求消息
    if (message instanceof Request) {
        // handle request.
        Request request = (Request) message;
        if (request.isEvent()) {
            handlerEvent(channel, request);
        } else {
            //如果是双工通讯
            if (request.isTwoWay()) {
                //核心代码
                handleRequest(exchangeChannel, request);
            } else {
                handler.received(exchangeChannel, request.getData());
            }
        }
    } else if (message instanceof Response) {
        //如果是客户端接收到的响应消息
        handleResponse(channel, (Response) message);
    } else if (message instanceof String) {
        if (isClientSide(channel)) {
            Exception e = new Exception("Dubbo client can not supported string message: " +
message + " in channel: " + channel + ", url: " + channel.getUrl());
            logger.error(e.getMessage(), e);
        } else {
            String echo = handler.telnet(channel, (String) message);
            if (echo != null && echo.length() > 0) {
                channel.send(echo);
            }
        }
    } else {
        handler.received(exchangeChannel, message);
    }
}
```

客户端接收到Response响应后，走的逻辑就是 handleResponse(channel, (Response) message);逻辑了，我们看看这个方法。

```java
static void handleResponse(Channel channel, Response response) throws RemotingException {
    if (response != null && !response.isHeartbeat()) {
        DefaultFuture.received(channel, response);
    }
}
```

```java
public static void received(Channel channel, Response response, boolean timeout) {
    try {
        //根据返回的id，从map中找到该请求对应的future对象
        DefaultFuture future = FUTURES.remove(response.getId());
        if (future != null) {
            Timeout t = future.timeoutCheckTask;
            if (!timeout) {
                // decrease Time
                t.cancel();
            }
            //收到服务端的回调后，唤醒get方法
            future.doReceived(response);
        } else {
            logger.warn("The timeout response finally returned at "
                    + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date()))
                    + ", response status is " + response.getStatus()
                    + (channel == null ? "" : ", channel: " + channel.getLocalAddress()
                    + " -> " + channel.getRemoteAddress()) + ", please check provider side
for detailed result.");
        }
    } finally {
        CHANNELS.remove(response.getId());
    }
}
```

```java
private void doReceived(Response res) {
    if (res == null) {
        throw new IllegalStateException("response cannot be null");
    }
    if (res.getStatus() == Response.OK) {
        //收到返回结果后，这里是去唤醒get方法的。唤醒必须是接收到返回结果后才去唤醒
        this.complete(res.getResult());
    } else if (res.getStatus() == Response.CLIENT_TIMEOUT || res.getStatus() ==
Response.SERVER_TIMEOUT) {
        this.completeExceptionally(new TimeoutException(res.getStatus() ==
Response.SERVER_TIMEOUT, channel, res.getErrorMessage()));
    } else {
        this.completeExceptionally(new RemotingException(channel, res.getErrorMessage()));
    }

    // the result is returning, but the caller thread may still waiting
    // to avoid endless waiting for whatever reason, notify caller thread to return.
    if (executor != null && executor instanceof ThreadlessExecutor) {
        ThreadlessExecutor threadlessExecutor = (ThreadlessExecutor) executor;
        if (threadlessExecutor.isWaiting()) {
            threadlessExecutor.notifyReturn(new IllegalStateException("The result has
returned, but the biz thread is still waiting" +
                    " which is not an expected state, interrupt the thread manually by
returning an exception."));
        }
    }
}
```

前面我们分析客户端的请求流程时，我们看到了一个get方法的阻塞，其实上述逻辑最终掉了**this.complete(res.getResult())**;方法，这个方法就会唤醒get方法然后把返回结果传递给get方法，这样就完成了同步调用流程了。