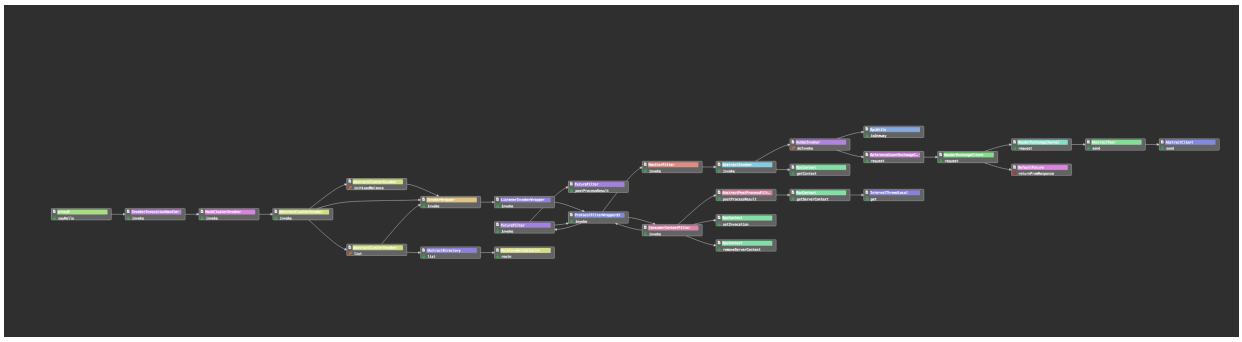
写在前面

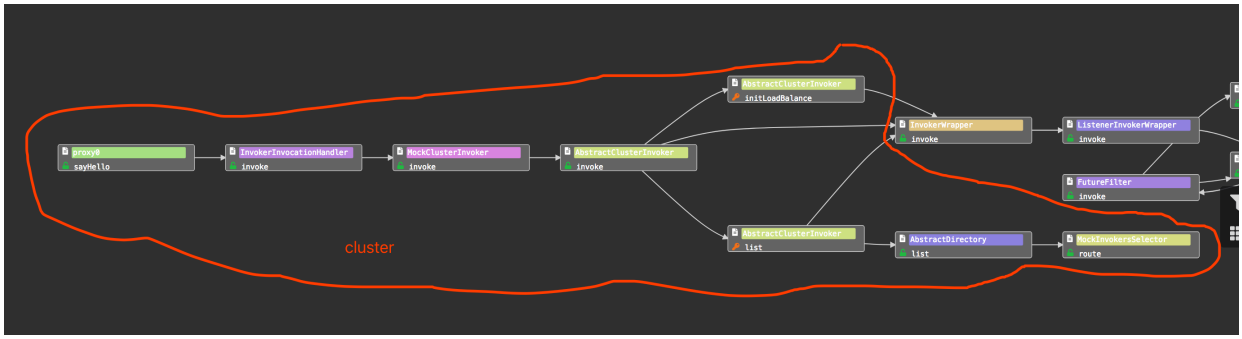
在分析整个调用过程时会发现 Invoker 和 Invocation 贯穿始末，层层调用的invoke的方法更是让人不知所措。先补充下dubbo的哲学思想，Invoker 作为dubbo 真正的执行单元，且可以同时为provider和consumer使用。Invoker 是实体域，它是 Dubbo 的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起 invoke 调用。个人觉得这种思想的优处是代码组织起来更方便，有很好的扩展性，但是可读性会变很差，除非功能描述能再类上完全体现出来，这要求了实现类必须遵守最小化和单一职责。Invocation 则是会话域，它持有调用过程中的变量，比如方法名，参数等。附上Invoker 类图

下图是主线程的一次调用过程，略去部分流程，根据测dubbo分层设计的思想，逐层对这个过程进行分析，本次我只关注调用过程，dubbo顶层的 配置层（Config）服务代理层（Proxy）不做分析，直接从集群层（Cluster）开始入手，其实下面分析的过程大致是三层，集群层（Cluster），远程调用层（Protocol）、信息交换层（Exchange）



集群层（Cluster）

下图是集群层涉及到的方法调用，Cluster是外围概念，所以Cluster的目的是将多个Invoker伪装成一个Invoker，这样其它人只要关注Protocol层Invoker即可。



InvokerInvocationHandler.invoke( )

服务的代理类，主要是判断了是不是object 的基本方法并判断了是否为异步调用，最后将方法签名和参数封装成了 Invocation 传递给了MockClusterInvoker

```
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    String methodName = method.getName();
    Class<?>[] parameterTypes = method.getParameterTypes();

    // 如果是Object的基本方法直接返回
    if (method.getDeclaringClass() == Object.class) {
        return method.invoke(invoker, args);
    }
    if ("toString".equals(methodName) && parameterTypes.length == 0) {
        return invoker.toString();
    }
    if ("hashCode".equals(methodName) && parameterTypes.length == 0) {
        return invoker.hashCode();
    }
    if ("equals".equals(methodName) && parameterTypes.length == 1) {
        return invoker.equals(args[0]);
    }

    RpcInvocation invocation;
    // 如果方法有@AsyncFor注解 && 方法名以异步后缀结尾 && 返回类型是
FutureReturnType
    // 则认为该方法是异步调用
    if (RpcUtils.hasGeneratedFuture(method)) {
        Class<?> clazz = method.getDeclaringClass();
        String syncMethodName = methodName.substring(0, methodName.length()
- Constants.ASYNC_SUFFIX.length());
        Method syncMethod = clazz.getMethod(syncMethodName,
method.getParameterTypes());
        invocation = new RpcInvocation(syncMethod, args);
        invocation.setAttachment(Constants.FUTURE_GENERATED_KEY, "true");
        invocation.setAttachment(Constants.ASYNC_KEY, "true");
    } else {
        invocation = new RpcInvocation(method, args);
        //这是干啥的？？
        if (RpcUtils.hasFutureReturnType(method)) {
            invocation.setAttachment(Constants.FUTURE_RETURNTYPE_KEY,
"true");
            invocation.setAttachment(Constants.ASYNC_KEY, "true");
        }
    }
    //同步调用MockClusterInvoker 的invoke方法
    return invoker.invoke(invocation).recreate();
}
```

MockClusterInvoker.invoke()

从名字即可看出MockClusterInvoker是支持 mock的 ClusterInvoker，这里主要做的事就是从url里判断这次调用是否为mock，MockClusterInvoker其实也是一个包装类，内部封装了RegistoryDerectory和一个 Invoker接口

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

    //检测url里是否有mock参数
    String value =
directory.getUrl().getMethodParameter(invocation.getMethodName(),
Constants.MOCK_KEY, Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || value.equalsIgnoreCase("false")) {
        //no mock
        //走真实调用场景
        result = this.invoker.invoke(invocation);
    } else if (value.startsWith("force")) {
        if (logger.isWarnEnabled()) {
            logger.warn("force-mock: " + invocation.getMethodName() + "
force-mock enabled , url : " + directory.getUrl());
        }
        //force:direct mock
        //强制mock
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock
        //mock 调用失败的场景
        try {
            result = this.invoker.invoke(invocation);
        } catch (RpcException e) {
            if (e.isBiz()) {
                throw e;
            } else {
                if (logger.isWarnEnabled()) {
                    logger.warn("fail-mock: " + invocation.getMethodName()
+ " fail-mock enabled , url : " + directory.getUrl(), e);
                }
                result = doMockInvoke(invocation, e);
            }
        }
    }
    return result;
}
```

FailOverClusterInvoker.invoke()--其实调用的是其父类AbstractClusterInvoker.invoke()

FailOverClusterInvoker的功能是支持 失败重试机制的ClusterInvoker

其作用就是获取可用provider对应的invoker，这些invoker其实是从RegisterDirectory注册目录中获取的，最后选择出负载算法（默认随机）

```
@Override
public Result invoke(final Invocation invocation) throws RpcException {

    //检查clusterInvoker 是否被销毁，通过 类型为AtomicBoolean 的destroy 变量来判断
    checkWhetherDestroyed();

    // binding attachments into invocation.
    // debug 时， 返回的contextAttachments 为空，但从上面原始注释看不出其具体作用
    Map<String, String> contextAttachments =
RpcContext.getContext().getAttachments();
    if (contextAttachments != null && contextAttachments.size() != 0) {
        ((RpcInvocation) invocation).addAttachments(contextAttachments);
    }

    //列举出可用的invoker，里面写的有点绕没看懂
    List<Invoker<T>> invokers = list(invocation);

    //根据服务提供者列表（invoker）和调用参数（invocation）决定使用哪种负载均衡方式，
默认随机
    //loadbalance 是通过ExtensionLoader 加载获得的
    LoadBalance loadbalance = initLoadBalance(invokers, invocation);

    //异步相关，不做深入研究
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    return doInvoke(invocation, invokers, loadbalance);
}
```

FailOverClusterInvoker.doInvoke()

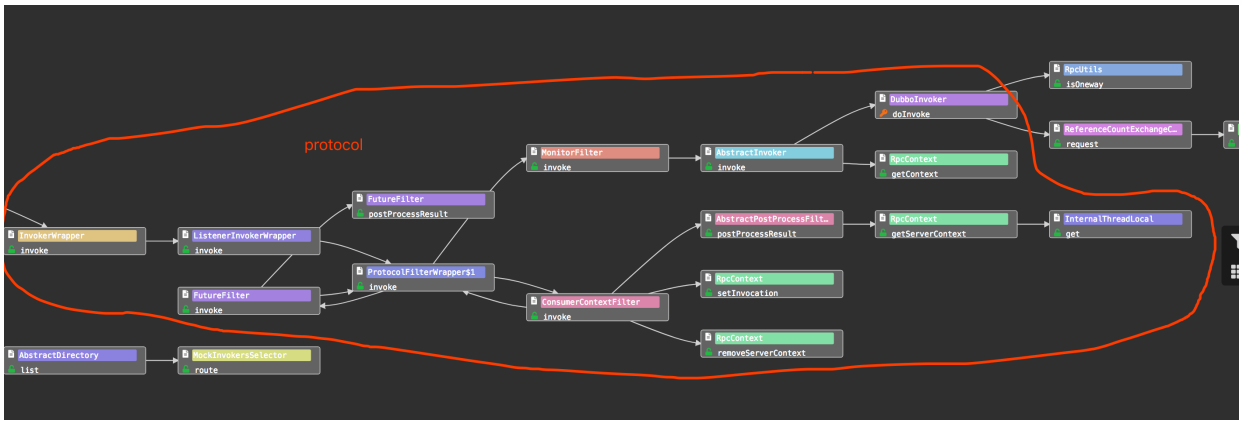
实现了失败重试的功能，重试次数是从url中获取的。从provider提供的invoker list中选择一个调用

```
public Result doInvoke(Invocation invocation, final List<Invoker<T>>
invokers, LoadBalance loadbalance) throws RpcException {
    List<Invoker<T>> copyinvokers = invokers;
    checkInvokers(copyinvokers, invocation);
    String methodName = RpcUtils.getMethodName(invocation);

    //从url中获取重试的次数
    int len = getUrl().getMethodParameter(methodName,
Constants.RETRIES_KEY, Constants.DEFAULT_RETRIES) + 1;
    if (len <= 0) {
        len = 1;
    }
    // retry loop.
    RpcException le = null; // last exception.

    //用于存放已经调用过的invoker
    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.
        if (i > 0) {
            checkWhetherDestroyed();
            copyinvokers = list(invocation);
            // check again
            //再做校验的目的的是 invokers有可能发生变化
            checkInvokers(copyinvokers, invocation);
        }

        //从候选invoker中选择一个
        Invoker<T> invoker = select(loadbalance, invocation, copyinvokers,
invoked);
        invoked.add(invoker);
        RpcContext.getContext().setInvokers((List) invoked);
        try {
            //调用RegistryDirectory 的 invoke方法
            Result result = invoker.invoke(invocation);
            if (le != null && logger.isWarnEnabled()) {
                logger.warn(#$%^&);
            }
            return result;
        } catch (RpcException e) {
            if (e.isBiz()) { // biz exception.
                throw e;
            }
            le = e;
        } catch (Throwable e) {
            le = new RpcException(e.getMessage(), e);
        } finally {
            providers.add(invoker.getUrl().getAddress());
        }
    }
    throw new RpcException($%^&*^);
}
```

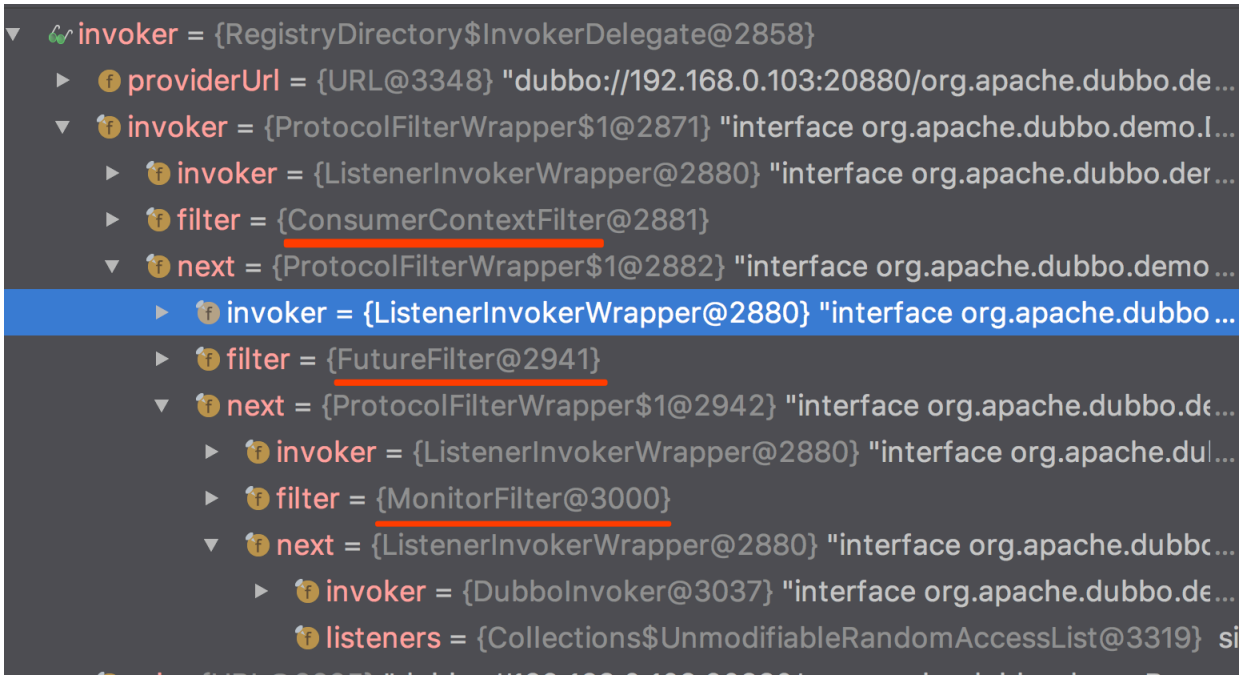
远程调用层（Protocol）：

Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。下图是protocal涉及的方法调用过程



从上面选出的invoker 实际上是个包装类，内部实现了责任链模式，增加了filter过滤，如下图，默认实现下面三个过滤器，分别是

- ConsumerContextFilter
- FutureFilter
- MonitorFilter



ConsumerContextFilter 过滤器对Rpc调用上下文进行设值

```
public Result invoke(Invoker<?> invoker, Invocation invocation) throws
RpcException {
    RpcContext.getContext()
            .setInvoker(invoker)
            .setInvocation(invocation)
            .setLocalAddress(NetUtils.getLocalHost(), 0)
            .setRemoteAddress(invoker.getUrl().getHost(),
                    invoker.getUrl().getPort());
    if (invocation instanceof RpcInvocation) {
        ((RpcInvocation) invocation).setInvoker(invoker);
    }
    try {
        // TODO should we clear server context?
        RpcContext.removeServerContext();
        return postProcessResult(invoker.invoke(invocation), invoker,
invocation);
    } finally {
        // TODO removeContext? but we need to save future for
RpcContext.getFuture() API. If clear attachments here, attachments will not
available when postProcessResult is invoked.
        RpcContext.getContext().clearAttachments();
    }
}
```

FutureFilter 异步回调过滤器，同步调用场景没有作用

```
@Override
public Result invoke(final Invoker<?> invoker, final Invocation invocation)
throws RpcException {
    fireInvokeCallback(invoker, invocation);
    // need to configure if there's return value before the invocation in
order to help invoker to judge if it's
    // necessary to return future.
    return postProcessResult(invoker.invoke(invocation), invoker,
invocation);
}
```

MonitorFilter 监控过滤器，监控耗时等信息

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws
RpcException {
    if (invoker.getUrl().hasParameter(Constants.MONITOR_KEY)) {
        RpcContext context = RpcContext.getContext(); // provider must
fetch context before invoke() gets called
        String remoteHost = context.getRemoteHost();
        long start = System.currentTimeMillis(); // record start timestamp
        getConcurrent(invoker, invocation).incrementAndGet(); // count up
        try {
            Result result = invoker.invoke(invocation); // proceed
invocation chain
            collect(invoker, invocation, result, remoteHost, start, false);
            return result;
        } catch (RpcException e) {
            collect(invoker, invocation, null, remoteHost, start, true);
            throw e;
        } finally {
            getConcurrent(invoker, invocation).decrementAndGet(); // count
down
        }
    } else {
        return invoker.invoke(invocation);
    }
}
```

invocation经过层层过滤后，经过DubboInvoker的doInvoker 将invocation传递给信息交换层（Exchange）的ReferenceCountExchangeClient

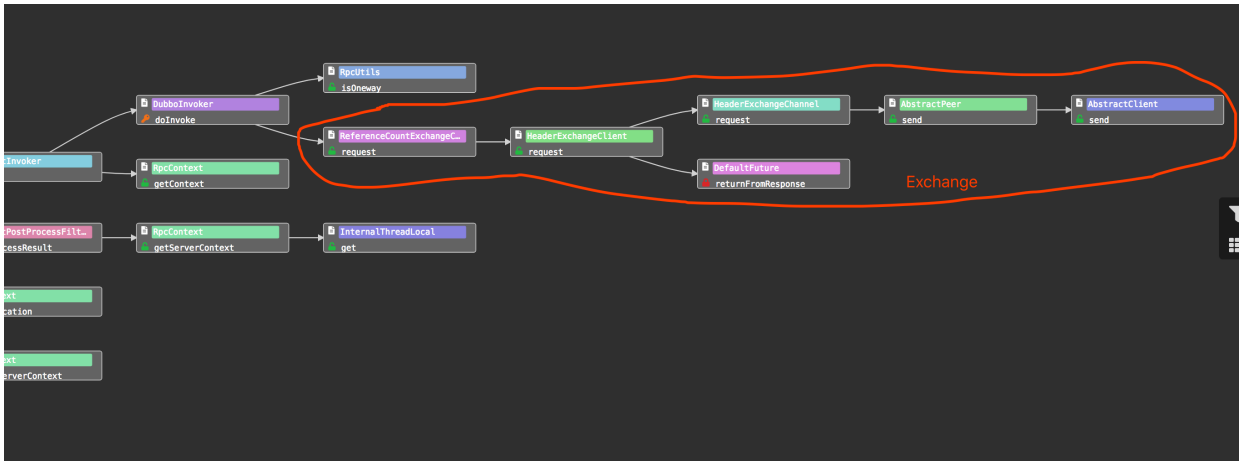
```
@Override
protected Result doInvoke(final Invocation invocation) throws Throwable {
    RpcInvocation inv = (RpcInvocation) invocation;
    final String methodName = RpcUtils.getMethodName(invocation);
    inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());
    inv.setAttachment(Constants.VERSION_KEY,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

    ExchangeClient currentClient;
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
    try {
        //是否是异步 ，有个疑问，前面已经判断了多次，怎么还判断
        boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
        // 是否用future框架
        boolean isAsyncFuture = RpcUtils.isGeneratedFuture(inv) ||
RpcUtils.isFutureReturnType(inv);
        //是否是单程
```

```
        boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
        int timeout = getUrl().getMethodParameter(methodName,
Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT);
        if (isOneway) {
            boolean isSent = getUrl().getMethodParameter(methodName,
Constants.SENT_KEY, false);
            currentClient.send(inv, isSent);
            RpcContext.getContext().setFuture(null);
            return new RpcResult();
        } else if (isAsync) {
            ResponseFuture future = currentClient.request(inv, timeout);
            // For compatibility
            FutureAdapter<Object> futureAdapter = new FutureAdapter<>
(future);
            RpcContext.getContext().setFuture(futureAdapter);

            Result result;
            if (isAsyncFuture) {
                // register resultCallback, sometimes we need the asyn
result being processed by the filter chain.
                result = new AsyncRpcResult(futureAdapter,
futureAdapter.getResultFuture(), false);
            } else {
                result = new SimpleAsyncRpcResult(futureAdapter,
futureAdapter.getResultFuture(), false);
            }
            return result;
        } else {
            RpcContext.getContext().setFuture(null);
            //invocation继续向下传递
            return (Result) currentClient.request(inv, timeout).get();
        }
    } 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);
    }
}
```

信息交换层（Exchange）

封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。

从名字的Exchange 可以理解为 将Invocation封装成Request，下图为封装过程调用。invocation 经过ReferenceCountExchangeClient、HeaderExchangeClient、HeaderExchangeChannel传递最终变成了 request。过程比较简单就不赘述了



网络传输层（Transport）

接下来就是网络传输层（Transport），抽象mina和netty为统一接口，逻辑则是经过几次 send方法将 Request 封装成Message调用nettty了。

总结

前半程以invoker 为主线实现支持mock、重试等机制，以及监控，异步回调等默认过滤机制

后半程invocation 为主线 将方法签名封装成 RPCInvocation ，通过信息交换层 将invocation封装成Request，再由网络传输层，将Request封装成message 有netty发出

参考资料：

https://zhouxiaowu.coding.me/2018/06/28/Dubbo%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B%E6%9C%8D%E5%8A%A1%E6%B6%88%E8%B4%B9%E8%BF%87%E7%A8%8B/

http://yeming.me/2018/07/31/dubbo4/

https://github.com/aalansehaiyang/technology-talk/blob/master/middle-software/dubbo-sourcecode.md

https://cdn2.jianshu.io/p/d8338935a60e

https://my.oschina.net/u/1263326/blog/614852

https://www.cnblogs.com/aspirant/p/9002663.html