# Dubbo源码解析 — DIRECTORY和ROUTER

今天看一下`Directory`和`Router`。

我们直接从代码看起（一贯风格），先看后总结，对着总结再来看，相信会收获很多。我们先看`com.alibaba.dubbo.config.ReferenceConfig`的`createProxy`：

```
if (urls.size() == 1) {
    invoker = refprotocol.refer(interfaceClass, urls.get(0));
}
```

这里我们之前说过了，根据扩展点，我们知道这里的`protocol`目前是`RegistryProtocol`，点进去：

```
return doRefer(cluster, registry, type, url);
```

这里的`cluster`同样是扩展点，点进去：

```
RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
directory.setRegistry(registry);
directory.setProtocol(protocol);
// all attributes of REFER_KEY
Map<String, String> parameters = new HashMap<String, String>(directory.getUrl
().getParameters());
URL subscribeUrl = new URL(Constants.CONSUMER_PROTOCOL, parameters.remove(Con
stants.REGISTER_IP_KEY), 0, type.getName(), parameters);
if (!Constants.ANY_VALUE.equals(url.getServiceInterface())
        && url.getParameter(Constants.REGISTER_KEY, true)) {
    registry.register(subscribeUrl.addParameters(Constants.CATEGORY_KEY, Cons
tants.CONSUMERS_CATEGORY,
            Constants.CHECK_KEY, String.valueOf(false)));
```

```
}
directory.subscribe(subscribeUrl.addParameter(Constants.CATEGORY_KEY,
        Constants.PROVIDERS_CATEGORY
                + "," + Constants.CONFIGURATORS_CATEGORY
                + "," + Constants.ROUTERS_CATEGORY));

Invoker invoker = cluster.join(directory);
ProviderConsumerRegTable.registerConsuemr(invoker, url, subscribeUrl, directo
ry);
return invoker;
```

这里出现了第一个东西：`DIRECTORY`，这里会用`cluster`的`join`方法生成一个`Invoker`：`Invoker invoker = cluster.join(directory);`，这里的`cluster`debug一下，是`FailoverCluster`，生成的就是`FailoverClusterInvoker`。这里我们都没有发现这个`Directory`的作用，先别急，我们至少知道了`AbstractClusterInvoker`中有一个`Directory`的实例。

再看我们的`AbstractClusterInvoker`的`invoke`方法，这是`Dubbo`所有集群`invoker`的入口：

```
checkWhetherDestroyed();
LoadBalance loadbalance;
//注意这里
List<Invoker<T>> invokers = list(invocation);
if (invokers != null && invokers.size() > 0) {
    loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).getEx
tension(invokers.get(0).getUrl()
            .getMethodParameter(invocation.getMethodName(), Constants.LOADBAL
ANCE_KEY, Constants.DEFAULT_LOADBALANCE));
} else {
    loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).getEx
tension(Constants.DEFAULT_LOADBALANCE);
}
RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
return doInvoke(invocation, invokers, loadbalance);
```

这里有个`list`方法，返回的是`invoker`集合，`doInvoke`方法下沉到了子类，之前我们说过，这里会根据负载均衡策略选出一个`invoker`执行，那么我们看下`list`方法是如何选取`invoker`集合的：

```
//使用directory去选取
List<Invoker<T>> invokers = directory.list(invocation);
```

```
return invokers;
```

点进去进入到`AbstractDirectory`的`list`方法：

```
List<Invoker<T>> invokers = doList(invocation);
//本地的routers
List<Router> localRouters = this.routers; // local reference
if (localRouters != null && localRouters.size() > 0) {
    for (Router router : localRouters) {
        try {
            if (router.getUrl() == null || router.getUrl().getParameter(Const
ants.RUNTIME_KEY, false)) {
                invokers = router.route(invokers, getConsumerUrl(), invocatio
n);
            }
        } catch (Throwable t) {
            logger.error("Failed to execute router: " + getUrl() + ", cause:
" + t.getMessage(), t);
        }
    }
}
```

继续看一下`doList`，这里同样下沉到子类，子类有两个，分别是：`RegistryDirectory`和`StaticDirectory`，这两个的区别我们别的文章再说，这里简单提一下：`RegistryDirectory`中的`invoker`集合是动态的（实现了`NotifyListener`接口），而`StaticDirectory`中的`invoker`集合是固定的。

那么`StaticDirectory`返回的集合就是固定的，而`RegistryDirectory`是动态的，我们在注册中心对`invoker`做改动，都会引起`RegistryDirectory`中`invoker`集合的变化。

这里我们大概就能理解这个`Directory`，实际上，这个东西就是`Invoker`的集合。

`doList`结束，下面就出现了我们说的`Router`，默认有三种实现：`ConditionRouter`、`MockInvokersSelector`（虽然叫Selector）和`ScriptRouter`。这篇文章先不具体展开讨论每种`Router`的实现方式。

`router`返回的并不是一个`Invoker`，而是`invoker`集合，我们可以认为，`router`的作用就是把`Directory`中符合路由规则的`invoker`筛选出来，然后在这些`invoker`中，根据负载均衡策略和集群容错策略执行`invoke`方法。

总结一下：

**Director**：`invoker`的集合，是`AbstractClusterInvoker`的属性，可以认为是本集群下的所有`Invoker`。
**Router**：从`Directory`中挑选出符合要求的`Invoker`集合。
**LoadBalance**：从`Router`挑选的`invoker`中再选出**其中一个**作为最终选择去执行`invoke`方法。