Dubbo服务的发现流程源码分析

1、spring代理对象

当你用@DubboReference标识一个属性时，其实spring会生成了spring的代理对象注入进去。spring创建代理的过程在第四节课的时候我们已经讲过，同学们可以翻看第四节课的笔记看看代理的生成原理。

```
private void createLazyProxy() {
//set proxy interfaces
//see also:
org.apache.dubbo.rpc.proxy.AbstractProxyFactory.getProxy(org.apache.dubbo.rpc.Invoker<T>,
boolean)
//很明显这里会用spring的代理工厂生成代理对象
ProxyFactory proxyFactory = new ProxyFactory();
//定义哦了TargetSource类型实例，spring中会有该类调用其getTarget方法拿到目标对象，其实这里就会生成Dubbo
的代理
proxyFactory.setTargetSource(new DubboReferenceLazyInitTargetSource());
proxyFactory.addInterface(interfaceClass);
Class<?>[] internalInterfaces = AbstractProxyFactory.getInternalInterfaces();
for (Class<?> anInterface : internalInterfaces) {
    proxyFactory.addInterface(anInterface);
}
if (!StringUtils.isEquals(interfaceClass.getName(), interfaceName)) {
    //add service interface
    try {
        Class<?> serviceInterface = ClassUtils.forName(interfaceName, beanClassLoader);
        proxyFactory.addInterface(serviceInterface);
    } catch (ClassNotFoundException e) {
        // generic call maybe without service interface class locally
    }
}

//返回spring的代理
this.lazyProxy = proxyFactory.getProxy(this.beanClassLoader);
}
```

在spring中的JDKDynamicAopProxy类中就会回调TargetSource对象的getTarget方法，在getTarget方法中生成了dubbo的代理对象。

```
private class DubboReferenceLazyInitTargetSource extends AbstractLazyCreationTargetSource {
@Override
protected Object createObject() throws Exception {
    return getCallProxy();
}

@Override
public synchronized Class<?> getTargetClass() {
    return getInterfaceClass();
}
}

//父类的getTarget方法会被spring的advice调用到，又会回调子类的createObject方法，模板设计模式
@Override
public synchronized Object getTarget() throws Exception {
    if (this.lazyTarget == null) {
        logger.debug("Initializing lazy target object");
        this.lazyTarget = createObject();
    }
    return this.lazyTarget;
}
```

```
private Object getCallProxy() throws Exception {
    if (referenceConfig == null) {
        throw new IllegalStateException("ReferenceBean is not ready yet, please make sure
to call reference interface method after dubbo is started.");
    }
    //获取引用代理
    //get reference proxy
    return referenceConfig.get();
}
```

从源码中可以看到，最终服务的引用还是调到了referenceConfig.get()方法，那我们就来看看get方法是如何走的。

2、dubbo代理对象

前面我们知道，当我们用spring的代理对象调用的时候，最终会走到JDKDynamicAopProxy类中的invoke方法，然后调用getTarget方法生成了一个dubbo的代理对象，那么这个代理对象是如何生成的呢？我们看看生成dubbo代理对象的核心代码；

```
private T createProxy(Map<String, String> map) {
    //是不是injvm调用
    if (shouldJvmRefer(map)) {
        URL url = new ServiceConfigURL(LOCAL_PROTOCOL, LOCALHOST_VALUE, 0,
interfaceClass.getName()).addParameters(map);
        invoker = REF_PROTOCOL.refer(interfaceClass, url);
        if (logger.isInfoEnabled()) {
            logger.info("Using injvm service " + interfaceClass.getName());
        }
    } else {
        urls.clear();
        if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer
address, or register center's address.
            String[] us = SEMICOLON_SPLIT_PATTERN.split(url);
            if (us != null && us.length > 0) {
                for (String u : us) {
                    URL url = URL.valueOf(u);
                    if (StringUtils.isEmpty(url.getPath())) {
                        url = url.setPath(interfaceName);
                    }
                    if (UrlUtils.isRegistry(url)) {
                        urls.add(url.putAttribute(REFER_KEY, map));
                    } else {
                        URL peerURL = ClusterUtils.mergeUrl(url, map);
                        peerURL = peerURL.putAttribute(PEER_KEY, true);
                        urls.add(peerURL);
                    }
                }
            }
        } else { // assemble URL from register center's configuration
            // if protocols not injvm checkRegistry
            //如果协议不是injvm
            if (!LOCAL_PROTOCOL.equalsIgnoreCase(getProtocol())) {
                checkRegistry();
                List<URL> us = ConfigValidationUtils.loadRegistries(this, false);
                if (CollectionUtils.isNotEmpty(us)) {
                    for (URL u : us) {
                        URL monitorUrl = ConfigValidationUtils.loadMonitor(this, u);
                        if (monitorUrl != null) {
                            u = u.putAttribute(MONITOR_KEY, monitorUrl);
                        }
                        urls.add(u.putAttribute(REFER_KEY, map));
                    }
                }
                if (urls.isEmpty()) {
                    throw new IllegalStateException(
                            "No such any registry to reference " + interfaceName + " on the
consumer " + NetUtils.getLocalHost() +
                                    " use dubbo version " + Version.getVersion() +
                                    ", please config <dubbo:registry address=\"...\" /> to
your spring config.");
                }
            }
        }

        if (urls.size() == 1) {
            //生成MigrationInvoker对象，走包装类
            invoker = REF_PROTOCOL.refer(interfaceClass, urls.get(0));
        } else {
            List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();
            URL registryURL = null;
            for (URL url : urls) {
                // For multi-registry scenarios, it is not checked whether each
referInvoker is available.
                // Because this invoker may become available later.
                invokers.add(REF_PROTOCOL.refer(interfaceClass, url));

                if (UrlUtils.isRegistry(url)) {
                    registryURL = url; // use last registry url
                }
            }
            if (registryURL != null) { // registry url is available
                // for multi-subscription scenario, use 'zone-aware' policy by default
                String cluster = registryURL.getParameter(CLUSTER_KEY,
ZoneAwareCluster.NAME);
                // The invoker wrap sequence would be:
ZoneAwareClusterInvoker(StaticDirectory) -> FailoverClusterInvoker(RegistryDirectory,
routing happens here) -> Invoker
                invoker = Cluster.getCluster(cluster, false).join(new
StaticDirectory(registryURL, invokers));
            } else { // not a registry url, must be direct invoke.
                String cluster = CollectionUtils.isNotEmpty(invokers)
                        ?
                        (invokers.get(0).getUrl() != null ?
invokers.get(0).getUrl().getParameter(CLUSTER_KEY, ZoneAwareCluster.NAME) :
                                Cluster.DEFAULT)
                        : Cluster.DEFAULT;
                invoker = Cluster.getCluster(cluster).join(new StaticDirectory(invokers));
            }
        }
    }

    if (logger.isInfoEnabled()) {
        logger.info("Referred dubbo service " + interfaceClass.getName());
    }

    URL consumerURL = new ServiceConfigURL(CONSUMER_PROTOCOL, map.get(REGISTER_IP_KEY), 0,
map.get(INTERFACE_KEY), map);
    MetadataUtils.publishServiceDefinition(consumerURL);

    //生成代理对象
    // create service proxy
    return (T) PROXY_FACTORY.getProxy(invoker, ProtocolUtils.isGeneric(generic));
}
```

从代码来看看，其实在生成dubbo代理之前想要生成invoker对象，因为代理对象也要通过invoker来完成服务的远程调用的。所以我们先看看生成invoker的逻辑。

2.1、invoker对象的生成

2.1.1、描述

当我们用dubbo的代理对象进行调用的时候，实际上代理对象的advice一样是持有了invoker对象的引用的，实际上是用invoker对象进行后续逻辑的调用，所以我们第一步是要获取到进行rpc远程调用的invoker对象。

2.1.2、源码分析

源码是 invoker = REF_PROTOCOL.refer(interfaceClass, urls.get(0));，根据protocol对象调用refer方法获取到invoker对象，这里获取到的invoker对象是MigrationInvoker对象。其实refer流程跟之前分析的export流程有点类似，也是先走registry协议完成consumer的注册，然后在dubbo协议中开启了netty客户端的连接的。那么我们就一个个的分析

2.1.2.1、包装类的流转

registry协议对应的包装类的流转

2.1.2.1.1、QosProtocolWrapper

一样的是开启qos服务

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        startQosServer(url);
        return protocol.refer(type, url);
    }
    return protocol.refer(type, url);
}
```

2.1.2.1.2、ProtocolSerializationWrapper

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    return protocol.refer(type, url);
}
```

2.1.2.1.3、ProtocolSerializationWrapper

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        return protocol.refer(type, url);
    }
    return builder.buildInvokerChain(protocol.refer(type, url), REFERENCE_FILTER_KEY,
CommonConstants.CONSUMER);
}
```

2.1.2.1.4、ProtocolFilterWrapper

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        return protocol.refer(type, url);
    }

    Invoker<T> invoker = protocol.refer(type, url);
    if (StringUtils.isEmpty(url.getParameter(REGISTRY_CLUSTER_TYPE_KEY))) {
        invoker = new ListenerInvokerWrapper<>(invoker,
                Collections.unmodifiableList(
                        ExtensionLoader.getExtensionLoader(InvokerListener.class)
                                .getActivateExtension(url, INVOKER_LISTENER_KEY)));
    }
    return invoker;
}
```

2.1.2.2、InterfaceCompatibleRegistryProtocol

在这里完成了消费者注册流程

```java
@Override
@SuppressWarnings("unchecked")
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    //根据url获取注册的协议地址 zookeeper://xxx
    url = getRegistryUrl(url);
    //根据协议头获取注册逻辑类
    Registry registry = registryFactory.getRegistry(url);
    if (RegistryService.class.equals(type)) {
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    // group="a,b" or group="*"
    Map<String, String> qs = (Map<String, String>) url.getAttribute(REFER_KEY);
    String group = qs.get(GROUP_KEY);
    if (group != null && group.length() > 0) {
        if ((COMMA_SPLIT_PATTERN.split(group)).length > 1 || "*".equals(group)) {
            return doRefer(Cluster.getCluster(MergeableCluster.NAME), registry, type, url,
qs);
        }
    }

    //获取集群容错对象  默认是FailoverCluster
    Cluster cluster = Cluster.getCluster(qs.get(CLUSTER_KEY));
    return doRefer(cluster, registry, type, url, qs);
}
```

```java
protected <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL
url, Map<String, String> parameters) {
    Map<String, Object> consumerAttribute = new HashMap<>(url.getAttributes());
    consumerAttribute.remove(REFER_KEY);
    URL consumerUrl = new ServiceConfigURL(parameters.get(PROTOCOL_KEY) == null ? DUBBO :
parameters.get(PROTOCOL_KEY),
        null,
        null,
        parameters.get(REGISTER_IP_KEY),
        0, getPath(parameters, type),
        parameters,
        consumerAttribute);
    url = url.putAttribute(CONSUMER_URL_KEY, consumerUrl);
    //获取 MigrationInvoker对象
    ClusterInvoker<T> migrationInvoker = getMigrationInvoker(this, cluster, registry, type,
url, consumerUrl);
    return interceptInvoker(migrationInvoker, url, consumerUrl, url);
}
```

```
protected <T> Invoker<T> interceptInvoker(ClusterInvoker<T> invoker, URL url, URL
consumerUrl, URL registryURL) {
    //获取  MigrationRuleListener 监听类
    List<RegistryProtocolListener> listeners = findRegistryProtocolListeners(url);
    if (CollectionUtils.isEmpty(listeners)) {
        return invoker;
    }

    for (RegistryProtocolListener listener : listeners) {
        //调用MigrationRuleListener的 onRefer方法
        listener.onRefer(this, invoker, consumerUrl, registryURL);
    }
    return invoker;
}
```

```
@Override
public void onRefer(RegistryProtocol registryProtocol, ClusterInvoker<?> invoker, URL
consumerUrl, URL registryURL) {
    MigrationRuleHandler<?> migrationRuleHandler =
handlers.computeIfAbsent((MigrationInvoker<?>) invoker, _key -> {
        ((MigrationInvoker<?>) invoker).setMigrationRuleListener(this);
        return new MigrationRuleHandler<>((MigrationInvoker<?>) invoker, consumerUrl);
    });

    //迁移规则的处理器
    migrationRuleHandler.doMigrate(rule);
}
```

```
public synchronized void doMigrate(MigrationRule rule) {
    if (migrationInvoker instanceof ServiceDiscoveryMigrationInvoker) {
        refreshInvoker(MigrationStep.FORCE_APPLICATION, 1.0f, rule);
        return;
    }

    // initial step : APPLICATION_FIRST
    MigrationStep step = MigrationStep.APPLICATION_FIRST;
    float threshold = -1f;

    try {
        step = rule.getStep(consumerURL);
        threshold = rule.getThreshold(consumerURL);
    } catch (Exception e) {
        logger.error("Failed to get step and threshold info from rule: " + rule, e);
    }

    //核心代码
    if (refreshInvoker(step, threshold, rule)) {
        // refresh success, update rule
        setMigrationRule(rule);
    }
}
```

```
@Override
public void migrateToApplicationFirstInvoker(MigrationRule newRule) {
    CountDownLatch latch = new CountDownLatch(0);
    //RegistryDirectory  主要看这里
    refreshInterfaceInvoker(latch);
    //ServiceDiscoveryRegistryDirectory
    refreshServiceDiscoveryInvoker(latch);

    // directly calculate preferred invoker, will not wait until address notify
    // calculation will re-occurred when address notify later
    calcPreferredInvoker(newRule);
}
```

```
protected void refreshInterfaceInvoker(CountDownLatch latch) {
    clearListener(invoker);
    if (needRefresh(invoker)) {
        if (logger.isDebugEnabled()) {
            logger.debug("Re-subscribing interface addresses for interface " +
type.getName());
        }

        if (invoker != null) {
            invoker.destroy();
        }
        //获取 invoker对象。。在这里真正获取到了一个invoker对象被MigrationInvoker持有
        invoker = registryProtocol.getInvoker(cluster, registry, type, url);
    }
    setListener(invoker, () -> {
        latch.countDown();
        FrameworkStatusReporter.reportConsumptionStatus(
            createConsumptionReport(consumerUrl.getServiceInterface(),
consumerUrl.getVersion(), consumerUrl.getGroup(), "interface")
        );
        if (step == APPLICATION_FIRST) {
            calcPreferredInvoker(rule);
        }
    });
}
```

RegistryDirectory这个类是一个非常关键的类，该类中有整个客户端的服务列表，并且在这个类中有很多事件监听，可以监听到数据的变更然后刷新本地服务列表。

```
@Override
public <T> ClusterInvoker<T> getInvoker(Cluster cluster, Registry registry, Class<T> type,
URL url) {
    DynamicDirectory<T> directory = new RegistryDirectory<>(type, url);
    return doCreateInvoker(directory, cluster, registry, type);
}
```

```
protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    Map<String, String> parameters = new HashMap<String, String>
(directory.getConsumerUrl().getParameters());
    URL urlToRegistry = new ServiceConfigURL(
        parameters.get(PROTOCOL_KEY) == null ? DUBBO : parameters.get(PROTOCOL_KEY),
        parameters.remove(REGISTER_IP_KEY), 0, getPath(parameters, type), parameters);
    if (directory.isShouldRegister()) {
        directory.setRegisteredConsumerUrl(urlToRegistry);
        //把协议注册到 /dubbo/cn.enjoy.userService/consumers节点下面
        registry.register(directory.getRegisteredConsumerUrl());
    }
    //创建路由链
    directory.buildRouterChain(urlToRegistry);
    //订阅事件，对 configurations，providers，routes节点建立监听
    directory.subscribe(toSubscribeUrl(urlToRegistry));

    //返回默认的 FailoverClusterInvoker对象
    return (ClusterInvoker<T>) cluster.join(directory);
}
```

2.1.3、总结

从上面源码分析看到了，最终返回了一个invoker对象，invoker对象关系是：MigrationInvoker->MockClusterInvoker->FailoverClusterInvoker

2.2、dubbo生成代理对象

前面我们分析到了通过protocol.refer方法调用已经生成了一个invoker对象了，现在这个invoker要传递给一个代理对象，看看代理对象是如何生成的。

```
PROXY_FACTORY.getProxy(invoker, ProtocolUtils.isGeneric(generic));
```

根据SPI的规则，首先经过包装类StubProxyFactoryWrapper.getProxy方法，：

```
@Override
public <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException {
    T proxy = proxyFactory.getProxy(invoker, generic);
    if (GenericService.class != invoker.getInterface()) {
        URL url = invoker.getUrl();
        String stub = url.getParameter(STUB_KEY, url.getParameter(LOCAL_KEY));
        if (ConfigUtils.isNotEmpty(stub)) {
            Class<?> serviceType = invoker.getInterface();
            if (ConfigUtils.isDefault(stub)) {
                if (url.hasParameter(STUB_KEY)) {
                    stub = serviceType.getName() + "Stub";
                } else {
                    stub = serviceType.getName() + "Local";
                }
            }
            try {
                Class<?> stubClass = ReflectUtils.forName(stub);
                if (!serviceType.isAssignableFrom(stubClass)) {
                    throw new IllegalStateException("The stub implementation class " +
stubClass.getName() + " not implement interface " + serviceType.getName());
                }
                try {
                    Constructor<?> constructor = ReflectUtils.findConstructor(stubClass,
serviceType);
                    proxy = (T) constructor.newInstance(new Object[]{proxy});
                    //export stub service
                    URLBuilder urlBuilder = URLBuilder.from(url);
                    if (url.getParameter(STUB_EVENT_KEY, DEFAULT_STUB_EVENT)) {
                        urlBuilder.addParameter(STUB_EVENT_METHODS_KEY,
StringUtils.join(Wrapper.getWrapper(proxy.getClass()).getDeclaredMethodNames(), ","));
                        urlBuilder.addParameter(IS_SERVER_KEY, Boolean.FALSE.toString());
                        try {
                            export(proxy, (Class) invoker.getInterface(),
urlBuilder.build());
                        } catch (Exception e) {
                            LOGGER.error("export a stub service error.", e);
                        }
                    }
                } catch (NoSuchMethodException e) {
                    throw new IllegalStateException("No such constructor \"public " +
stubClass.getSimpleName() + "(" + serviceType.getName() + ")\" in stub implementation class
" + stubClass.getName(), e);
                }
            } catch (Throwable t) {
                LOGGER.error("Failed to create stub implementation class " + stub + " in
consumer " + NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion() + ",
cause: " + t.getMessage(), t);
                // ignore
            }
        }
    }
    return proxy;

}
```

然后包装类调到JavassistProxyFactory

```
@Override
public <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException {
    // when compiling with native image, ensure that the order of the interfaces remains
unchanged
    LinkedHashSet<Class<?>> interfaces = new LinkedHashSet<>();

    String config = invoker.getUrl().getParameter(INTERFACES);
    if (config != null && config.length() > 0) {
        String[] types = COMMA_SPLIT_PATTERN.split(config);
        for (String type : types) {
            // TODO can we load successfully for a different classloader?.
            interfaces.add(ReflectUtils.forName(type));
        }
    }

    if (generic) {
        if (GenericService.class.equals(invoker.getInterface()) ||
!GenericService.class.isAssignableFrom(invoker.getInterface())) {
            interfaces.add(com.alibaba.dubbo.rpc.service.GenericService.class);
        }

        try {
            // find the real interface from url
            String realInterface = invoker.getUrl().getParameter(Constants.INTERFACE);
            interfaces.add(ReflectUtils.forName(realInterface));
        } catch (Throwable e) {
            // ignore
        }
    }

    interfaces.add(invoker.getInterface());
    interfaces.addAll(Arrays.asList(INTERNAL_INTERFACES));
    //生成代理对象
    return getProxy(invoker, interfaces.toArray(new Class<?>[0]));
}
```

```
public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
    return (T) Proxy.getProxy(interfaces).newInstance(new
InvokerInvocationHandler(invoker));
}
```

从上述代码中可以看到，最终dubbo生成了一个jdk的动态代理类，当spring的JDKDynamicAopProxy中持有该对象调用方法时，最终代码逻辑会走到InvokerInvocationHandler类的invoke方法，InvokerInvocationHandler持有了通过protocol.refer方法创建的invoker对象，该invoker对象是可以通过rpc远程访问的。

2.3、RegistryDirectory服务列表的刷新

在调用过程中我们需要获取到一个调用的服务列表，而这个服务列表是由RegistryDirectory提供的，那么服务列表是如何有值的呢？

RegistryDirectory类是实现了NotifyListener接口的，最终会被zookeeper的事件回调到NotifyListener接口的notify方法，这个为什么会这样我们后续会讲到。

我们看看RegistryDirectory中的notify方法：

```
//事件监听回调
@Override
public synchronized void notify(List<URL> urls) {
    if (isDestroyed()) {
        return;
    }

    //对回调的协议分组
    // routes://
    // override://
    //dubbo://
    Map<String, List<URL>> categoryUrls = urls.stream()
            .filter(Objects::nonNull)
            .filter(this::isValidCategory)
            .filter(this::isNotCompatibleFor26x)
            .collect(Collectors.groupingBy(this::judgeCategory));

    List<URL> configuratorURLs = categoryUrls.getOrDefault(CONFIGURATORS_CATEGORY,
Collections.emptyList());
    this.configurators =
Configurator.toConfigurators(configuratorURLs).orElse(this.configurators);

    List<URL> routerURLs = categoryUrls.getOrDefault(ROUTERS_CATEGORY,
Collections.emptyList());
    //生成路由规则，加入到规则链中
    toRouters(routerURLs).ifPresent(this::addRouters);

    // providers
    List<URL> providerURLs = categoryUrls.getOrDefault(PROVIDERS_CATEGORY,
Collections.emptyList());
    /**
     * 3.x added for extend URL address
     */
    ExtensionLoader<AddressListener> addressListenerExtensionLoader =
ExtensionLoader.getExtensionLoader(AddressListener.class);
    List<AddressListener> supportedListeners =
addressListenerExtensionLoader.getActivateExtension(getUrl(), (String[]) null);
    if (supportedListeners != null && !supportedListeners.isEmpty()) {
        for (AddressListener addressListener : supportedListeners) {
            providerURLs = addressListener.notify(providerURLs, getConsumerUrl(),this);
        }
    }
    //刷新本地服务列表
    refreshOverrideAndInvoker(providerURLs);
}
```

刷新本地服务列表

```
//刷新本地服务列表
```

```
private void refreshInvoker(List<URL> invokerUrls) {
    Assert.notNull(invokerUrls, "invokerUrls should not be null");

    if (invokerUrls.size() == 1
            && invokerUrls.get(0) != null
            && EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol())) {
        this.forbidden = true; // Forbid to access
        this.invokers = Collections.emptyList();
        routerChain.setInvokers(this.invokers);
        destroyAllInvokers(); // Close all invokers
    } else {
        this.forbidden = false; // Allow to access
        Map<URL, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap; // local reference
        if (invokerUrls == Collections.<URL>emptyList()) {
            invokerUrls = new ArrayList<>();
        }
        if (invokerUrls.isEmpty() && this.cachedInvokerUrls != null) {
            invokerUrls.addAll(this.cachedInvokerUrls);
        } else {
            this.cachedInvokerUrls = new HashSet<>();
            this.cachedInvokerUrls.addAll(invokerUrls);//Cached invoker urls, convenient
for comparison
        }
        if (invokerUrls.isEmpty()) {
            return;
        }
        //创建url和invoker对象的映射关系 .这里会根据dubbo协议创建invoker读写
        Map<URL, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls);// Translate url
list to Invoker map

        /**
         * If the calculation is wrong, it is not processed.
         *
         * 1. The protocol configured by the client is inconsistent with the protocol of
the server.
         *    eg: consumer protocol = dubbo, provider only has other protocol
services(rest).
         * 2. The registration center is not robust and pushes illegal specification data.
         *
         */
        if (CollectionUtils.isEmptyMap(newUrlInvokerMap)) {
            logger.error(new IllegalStateException("urls to invokers error
.invokerUrls.size :" + invokerUrls.size() + ", invoker.size :0. urls :" + invokerUrls
                    .toString()));
            return;
        }

        //所有的invoker对象
        List<Invoker<T>> newInvokers = Collections.unmodifiableList(new ArrayList<>
(newUrlInvokerMap.values()));
        // pre-route and build cache, notice that route cache should build on original
Invoker list.
```

```
        // toMergeMethodInvokerMap() will wrap some invokers having different groups, those
wrapped invokers not should be routed.
        routerChain.setInvokers(newInvokers);
        //这个invokers就是我们的服务列表
        this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;
        this.urlInvokerMap = newUrlInvokerMap;

        try {
            destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap); // Close the unused
Invoker
        } catch (Exception e) {
            logger.warn("destroyUnusedInvokers error. ", e);
        }

        // notify invokers refreshed
        this.invokersChanged();
    }
}
```

2.4、服务列表invoker的生成

前面我们在刷新服务列表的notify方法中我们看到了

Map<URL, Invoker> newUrlInvokerMap = toInvokers(invokerUrls);方法，其实所有的DubboInvoker对象都是在这里生成的，netty客户端的启动，调用链handler的建立都是在这里。

```
private Map<URL, Invoker<T>> toInvokers(List<URL> urls) {
    Map<URL, Invoker<T>> newUrlInvokerMap = new ConcurrentHashMap<>();
    if (urls == null || urls.isEmpty()) {
        return newUrlInvokerMap;
    }
    String queryProtocols = this.queryMap.get(PROTOCOL_KEY);
    for (URL providerUrl : urls) {
        // If protocol is configured at the reference side, only the matching protocol is
selected
        if (queryProtocols != null && queryProtocols.length() > 0) {
            boolean accept = false;
            String[] acceptProtocols = queryProtocols.split(",");
            for (String acceptProtocol : acceptProtocols) {
                if (providerUrl.getProtocol().equals(acceptProtocol)) {
                    accept = true;
                    break;
                }
            }
            if (!accept) {
                continue;
            }
        }
        if (EMPTY_PROTOCOL.equals(providerUrl.getProtocol())) {
            continue;
        }
        if
(!ExtensionLoader.getExtensionLoader(Protocol.class).hasExtension(providerUrl.getProtocol()
)) {
            logger.error(new IllegalStateException("Unsupported protocol " +
providerUrl.getProtocol() +
                    " in notified url: " + providerUrl + " from registry " +
getUrl().getAddress() +
                    " to consumer " + NetUtils.getLocalHost() + ", supported protocol: " +

 ExtensionLoader.getExtensionLoader(Protocol.class).getSupportedExtensions()));
            continue;
        }
        URL url = mergeUrl(providerUrl);

        // Cache key is url that does not merge with consumer side parameters, regardless
of how the consumer combines parameters, if the server url changes, then refer again
        Map<URL, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference
        Invoker<T> invoker = localUrlInvokerMap == null ? null :
localUrlInvokerMap.remove(url);
        if (invoker == null) { // Not in the cache, refer again
            try {
                boolean enabled = true;
                if (url.hasParameter(DISABLED_KEY)) {
                    enabled = !url.getParameter(DISABLED_KEY, false);
                } else {
                    enabled = url.getParameter(ENABLED_KEY, true);
                }
                if (enabled) {
                    //生成invoker对象 。。核心代码，这里的协议是dubbo协议。
                    invoker = protocol.refer(serviceType, url);
                }
            } catch (Throwable t) {
                logger.error("Failed to refer invoker for interface:" + serviceType +
",url:(" + url + ")" + t.getMessage(), t);
            }
            if (invoker != null) { // Put new invoker in cache
                newUrlInvokerMap.put(url, invoker);
            }
        } else {
            newUrlInvokerMap.put(url, invoker);
        }
    }
    return newUrlInvokerMap;
}
```

关键代码就是：

//生成invoker对象 。。核心代码，这里的协议是dubbo协议。 invoker = protocol.refer(serviceType, url);

前面我们分析过，跟export方法的逻辑有点类似：

2.4.1、包装类的流转

2.4.1.1、QosProtocolWrapper

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        startQosServer(url);
        return protocol.refer(type, url);
    }
    return protocol.refer(type, url);
}
```

2.4.1.2、ProtocolSerializationWrapper

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    return protocol.refer(type, url);
}
```

2.4.1.3、ProtocolFilterWrapper

创建一个invoker的链，前面分析过，这里不赘述了

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        return protocol.refer(type, url);
    }
    return builder.buildInvokerChain(protocol.refer(type, url), REFERENCE_FILTER_KEY,
CommonConstants.CONSUMER);
}
```

2.4.1.4、ProtocolListenerWrapper

服务引用后触发事件，前面分析过不赘述了。

```java
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (UrlUtils.isRegistry(url)) {
        return protocol.refer(type, url);
    }

    Invoker<T> invoker = protocol.refer(type, url);
    if (StringUtils.isEmpty(url.getParameter(REGISTRY_CLUSTER_TYPE_KEY))) {
        invoker = new ListenerInvokerWrapper<>(invoker,
                Collections.unmodifiableList(
                        ExtensionLoader.getExtensionLoader(InvokerListener.class)
                                .getActivateExtension(url, INVOKER_LISTENER_KEY)));
    }
    return invoker;
}
```

2.4.2、DubboProtocol

在DubboProtocol中完成了netty客户端的连接，handler链路的调用，其实这块跟DubboProtocol中的服务发布过程的export方法很类似。核心代码在getClients方法中

```
@Override
public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws
RpcException {
    optimizeSerialization(url);

    // create rpc invoker.
    //创建用于远程调用的invoker对象 getClients(url)核心方法
    DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url),
invokers);
    invokers.add(invoker);

    return invoker;
}
```

```
private ExchangeClient[] getClients(URL url) {
    // whether to share connection

    boolean useShareConnect = false;

    int connections = url.getParameter(CONNECTIONS_KEY, 0);
    List<ReferenceCountExchangeClient> shareClients = null;
    // if not configured, connection is shared, otherwise, one connection for one service
    if (connections == 0) {
        useShareConnect = true;

        /*
         * The xml configuration should have a higher priority than properties.
         */
        String shareConnectionsStr = url.getParameter(SHARE_CONNECTIONS_KEY, (String)
null);
        //默认建立一个长连接
        connections = Integer.parseInt(StringUtils.isBlank(shareConnectionsStr) ?
ConfigUtils.getProperty(SHARE_CONNECTIONS_KEY,
                DEFAULT_SHARE_CONNECTIONS) : shareConnectionsStr);
        shareClients = getSharedClient(url, connections);
    }

    ExchangeClient[] clients = new ExchangeClient[connections];
    for (int i = 0; i < clients.length; i++) {
        if (useShareConnect) {
            clients[i] = shareClients.get(i);

        } else {
            //初始化客户端
            clients[i] = initClient(url);
        }
    }
```

```java
    return clients;
}
```

```java
private ExchangeClient initClient(URL url) {

    // client type setting.
    String str = url.getParameter(CLIENT_KEY, url.getParameter(SERVER_KEY,
DEFAULT_REMOTING_CLIENT));

    url = url.addParameter(CODEC_KEY, DubboCodec.NAME);
    // enable heartbeat by default
    url = url.addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT));

    // BIO is not allowed since it has severe performance issue.
    if (str != null && str.length() > 0 &&
!ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported client type: " + str + "," +
                " supported client type is " +
StringUtils.join(ExtensionLoader.getExtensionLoader(Transporter.class).getSupportedExtensio
ns(), " "));
    }

    ExchangeClient client;
    try {
        // connection should be lazy
        if (url.getParameter(LAZY_CONNECT_KEY, false)) {
            client = new LazyConnectExchangeClient(url, requestHandler);

        } else {
            //核心代码，建立连接
            client = Exchangers.connect(url, requestHandler);
        }

    } catch (RemotingException e) {
        throw new RpcException("Fail to create remoting client for service(" + url + "): "
+ e.getMessage(), e);
    }

    return client;
}
```

```java
@Override
public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {
    return new HeaderExchangeClient(Transporters.connect(url, new DecodeHandler(new
HeaderExchangeHandler(handler))), true);
}
```

```
public static Client connect(URL url, ChannelHandler... handlers) throws RemotingException
{
    if (url == null) {
        throw new IllegalArgumentException("url == null");
    }
    ChannelHandler handler;
    if (handlers == null || handlers.length == 0) {
        handler = new ChannelHandlerAdapter();
    } else if (handlers.length == 1) {
        handler = handlers[0];
    } else {
        handler = new ChannelHandlerDispatcher(handlers);
    }
    return getTransporter().connect(url, handler);
}
```

```
@Override
public Client connect(URL url, ChannelHandler handler) throws RemotingException {
    return new NettyClient(url, handler);
}
```

在NettyClient构造函数中又对handler进行了再一次包装，整个过程跟NettyServer类似我这里就不赘述了。