Dubbo架构图（取自dubbo官网）：



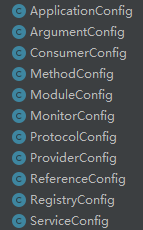
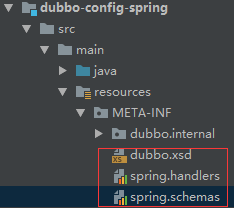
1. 加载、运行服务提供者
2. 向注册中心注册自己提供的服务。
3. 向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

# Dubbo融合Spring

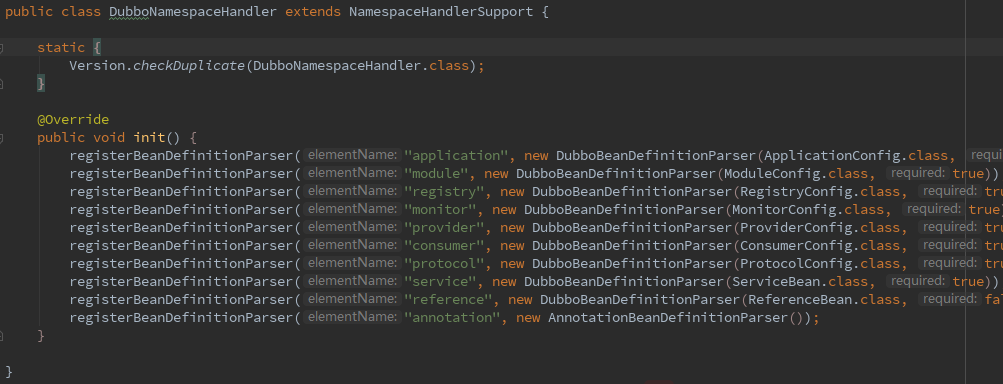
Dubbo采用Spring的配置方式，透明化的接入应用，对应用没有任何API侵入，只需用Spring加载Dubbo的配置即可。

dubbo-config-api模块定义了dubbo中核心的配置，本文后续会对服务提供者ServiceConfig（暴露服务）、服务消费者ReferenceConfig（引用服务）进行主要分析。

dubbo-config-spring模块中，采用Spring中的Schema扩展将dubbo标签解析为相应的Bean：

spring.handlers指定了具体的dubbo标签处理者：DubboNamespaceHandler

http\://dubbo.apache.org/schema/dubbo=com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler  


其中DubboBeanDefinitionParser则是具体的将配置属性、继承属性注入到相应Bean中，解析相关的代码就不细说了。

# Dubbo的SPI机制

1. public class ServiceConfig<T> extends AbstractServiceConfig {
2. private static final Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
3. private static final ProxyFactory proxyFactory = ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();
4. ...

在分析Dubbo源码过程中，发现很多使用ExtensionLoader来获取对象的地方，没搞明白这种加载方式可不利于继续阅读源码，就先让我们从这个SPI加载机制来入手吧。

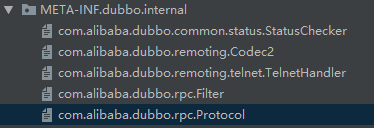
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

ExtensionLoader.getExtensionLoader(Protocol.class)获取到的是Protocol接口类的一个Loader。Loader下面的getAdaptiveExtension()方法则是获取适配的类对象。Dubbo使用这样的方式来获取对象是为了更好的拓展性。

具体来看看是怎么获取到适配对象的吧（部分代码省略）：

1. **public** T getAdaptiveExtension() {
2. Object instance = cachedAdaptiveInstance.get();
3. **if** (instance == **null**) {
4. //省略 双重判空锁
5. instance = createAdaptiveExtension();//1、创建适配对象，往下看
6. cachedAdaptiveInstance.set(instance);
7. //省略
8. }
9. **return** (T) instance;
10. }
11. private T createAdaptiveExtension() {
12. //省略try catch块 2、继续往下看getAdaptiveExtensionClass方法
13. return injectExtension((T) getAdaptiveExtensionClass().newInstance());
14. }
15. private Class<?> getAdaptiveExtensionClass() {
16. getExtensionClasses();//3、获取所有的拓展Class对象及默认实现类，下面会做简要介绍
17. if (cachedAdaptiveClass != null) {
18. return cachedAdaptiveClass;
19. }// 4、跳到createAdaptiveExtensionClass里创建适配类
20. return cachedAdaptiveClass = createAdaptiveExtensionClass();
21. }

getExtensionClasses方法会对META-INF/dubbo/internal/、META-INF/dubbo/、META-INF/services/ 三个文件夹下的文件进行扫描



比如当前Protocol加载的就是com.alibaba.dubbo.rpc.Protocol文件，文件中是Protocol接口的各个实现类的全路径。

其中有一种特殊的实现如ProtocolFilterWrapper，该类的构造参数是Protocol ，在Dubbo中如果构造参数只有一个且是该类实现的接口类型，则认为该类为Wapper包装类。Protocol就存在ProtocolFilterWrapper 和 ProtocolListenerWrapper 两个包装类。Dubbo中自定义的Filter就是通过ProtocolFilterWrapper对Invoker进行包装的。Dubbo中服务暴露的监听事件是通过ProtocolListenerWrapper包装实现的。

final SPI defaultAnnotation = type.getAnnotation(SPI.class);//获取SPI注解

if (defaultAnnotation != null) {

String value = defaultAnnotation.value();

if ((value = value.trim()).length() > 0) {

//省略校验

//获取默认实现类

if (names.length == 1) cachedDefaultName = names[0];

}

}

// 获取全部实现类

Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();

loadDirectory(extensionClasses, DUBBO\_INTERNAL\_DIRECTORY);

loadDirectory(extensionClasses, DUBBO\_DIRECTORY);

loadDirectory(extensionClasses, SERVICES\_DIRECTORY);

Protocol默认实现类就是接口上SPI注解的值“dubbo”在com.alibaba.dubbo.rpc.Protocol文件中对应的实现类。



获取到所有的实现类了后，接下来接着看怎么创建适配类

1. private Class<?> createAdaptiveExtensionClass() {
2. String code = createAdaptiveExtensionClassCode();//5、组装适配类的代码
3. ClassLoader classLoader = findClassLoader();
4. com.alibaba.dubbo.common.compiler.Compiler compiler = ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.common.compiler.Compiler.class).getAdaptiveExtension();
5. return compiler.compile(code, classLoader);//6、将组装code编译成class
6. }

通过createAdaptiveExtensionClassCode方法组装的code实际上就是一个代理类，通过行参中的URL对象（或者行参中能获取到URL对象）来选择具体调用哪一个实现类的该方法。以Protocol为例来看看组装的具体代码：

1. import com.alibaba.dubbo.common.extension.ExtensionLoader;
2. public class Protocol$Adaptive implements com.alibaba.dubbo.rpc.Protocol {
3. //省略 getDefaultPort、destroy 、refer方法 都是代理 不一一做分析，下面拿export方法开刀
4. public com.alibaba.dubbo.rpc.Exporter export(com.alibaba.dubbo.rpc.Invoker arg0) throws com.alibaba.dubbo.rpc.RpcException {
5. if (arg0 == null) throw new IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument == null");
6. //dubbo中拓展点实现方法都包含URL对象
7. //Dubbo中统一的URL模型：
8. // 1、有的配置信息都转换成URL参数
9. // 2、所有的元信息传输都采用URL
10. // 3、所有接口都能获取到URL
11. if (arg0.getUrl() == null)
12. throw new IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument getUrl() == null");
13. //获取URL对象，Dubbo中通过URL来控制对象调用的切换
14. com.alibaba.dubbo.common.URL url = arg0.getUrl();
15. //获取当前URL中指定的Protocol实现
16. String extName = (url.getProtocol() == null ? "dubbo" : url.getProtocol());
17. if (extName == null)
18. throw new IllegalStateException("Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url(" + url.toString() + ") use keys([protocol])");
19. //获取当前URL中指定的Protocol实现的对象,如果有Wapper对象，则返回用wapper包装后的对象
20. com.alibaba.dubbo.rpc.Protocol extension =
21. (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.Protocol.class).getExtension(extName);
22. return extension.export(arg0);//代理到获取到对象的此方法
23. }
24. }

# 服务的提供者ServiceConfig

1. 服务暴露触发点

Dubbo是通过Spring配置来植入应用的，ServiceBean是SreviceConfig的实现，在ServiceBean中监听了容器刷新事件onApplicationEvent(ContextRefreshedEvent event)。

1. @Override
2. public void onApplicationEvent(ContextRefreshedEvent event) {
3. //(支持ApplicationListener|非延迟暴露) | 已经暴露 | 不暴露
4. if (isDelay() && !isExported() && !isUnexported()) {
5. if (logger.isInfoEnabled()) {
6. logger.info("The service ready on spring started. service: " + getInterface());
7. }
8. //1、调用ServiceConfig export方法导出
9. export();
10. }
11. }

2、服务暴露

1. public synchronized void export() {
2. //省略继承provider属性代码
3. if (delay != null && delay > 0) {//配置了延迟暴露
4. delayExportExecutor.schedule(new Runnable() {
5. @Override
6. public void run() {
7. doExport();
8. }
9. }, delay, TimeUnit.MILLISECONDS);
10. } else {
11. doExport(); //2、继续往里走
12. }
13. }

doExport方法中大部分代码在进行校验，最后调用了doExportUrls方法进行导出，直接进入到doExportUrls中：

1. private void doExportUrls() {
2. List<URL> registryURLs = loadRegistries(true);//加载注册中心信息
3. for (ProtocolConfig protocolConfig : protocols) { //dubbo支持多协议暴露
4. doExportUrlsFor1Protocol(protocolConfig, registryURLs);
5. }
6. }

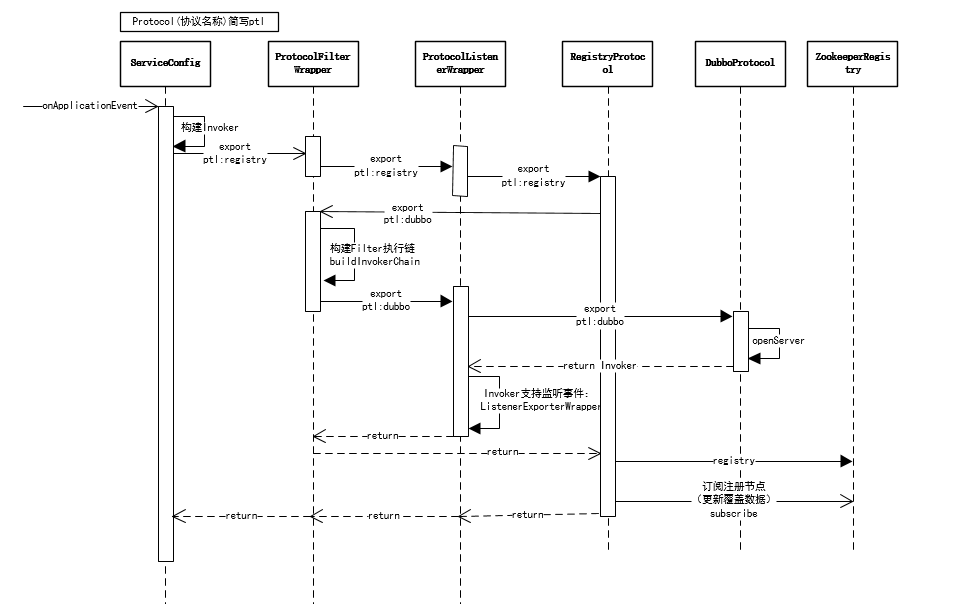
接下来进入到doExportUrlsFor1Protocol中，大部分构建URL的代码被省略掉。

1. private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {
2. //省略… 获取协议版本、发布版本、时间戳 等信息来构造URL
3. if (!Constants.SCOPE\_NONE.toString().equalsIgnoreCase(scope)) {
4. //暴露在本地
5. if (!Constants.SCOPE\_REMOTE.toString().equalsIgnoreCase(scope)) {
6. exportLocal(url);
7. }
8. //远程暴露
9. if (!Constants.SCOPE\_LOCAL.toString().equalsIgnoreCase(scope)) {
10. if (registryURLs != null && !registryURLs.isEmpty()) {
11. //向service配置的注册中心逐个暴露
12. for (URL registryURL : registryURLs) {
13. //省略 加载监控、URL中添加监控地址信息、日志
14. //代理层工厂创建Invoker，默认使用javassist代理
15. Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(Constants.EXPORT\_KEY, url.toFullString()));
16. //再将Invoker包装一层，并持有ServiceConfig对象
17. DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);
18. //按照指定协议获取暴露服务对象，protocol导出服务调用链下方用UML序列图展示
19. Exporter<?> exporter = protocol.export(wrapperInvoker);
20. exporters.add(exporter);
21. }
22. } else {
23. Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, url);
24. DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);
25. Exporter<?> exporter = protocol.export(wrapperInvoker);
26. exporters.add(exporter);
27. }
28. }
29. }
30. this.urls.add(url);
31. }

通过javassist代理工厂包装出来的类如下：

1. public class Wrapper20 extends Wrapper2 implements DC {
2. //省略代理类方法，属性相关的代码
4. //代理方法
5. public Object invokeMethod(Object var1, String var2, Class[] var3, Object[] var4) throws InvocationTargetException {
6. DemoServiceImpl var5; //代理对象 var1传入
7. try {
8. var5 = (DemoServiceImpl)var1;
9. } catch (Throwable var8) {
10. throw new IllegalArgumentException(var8);
11. }
12. try { //相关方法，只做一层代理
13. if ("funcOne".equals(var2) && var3.length == 0) {
14. return var5.funcOne();
15. }
16. if ("funcTwo".equals(var2) && var3.length == 1) {
17. return var5.funcTwo((Integer)var4[0]);
18. }
19. } catch (Throwable var9) {
20. throw new InvocationTargetException(var9);
21. }
22. throw new NoSuchMethodException("Not found method \"" + var2 + "\" in class com.xx.xx.samples.loader.service.impl.UserServiceImpl.");
23. }
24. }

服务暴露时序图：



# 服务的消费方ReferenceConfig

Dubbo对于ReferenceConfig接入Spring提供了ReferenceBean的实现类，实现FactoryBean接口通过getObject方法来获取远程调度代理对象。

1. @Override
2. public Object getObject() throws Exception {
3. return get();//调用父类ReferenceConfig的get方法获取
4. }
5. public synchronized T get() {
6. if (destroyed) {
7. throw new IllegalStateException("Already destroyed!");
8. }
9. if (ref == null) {
10. init();//初始化
11. }
12. return ref;
13. }
14. private void init() {
15. //构建参数,忽略...
16. ref = createProxy(map);//创建代理对象
17. //忽略
18. }
19. private T createProxy(Map<String, String> map) {
20. if (isJvmRefer) {
21. //忽略，本地引用
22. } else {
23. if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer address, or register center's address.
24. //有设置url参数，直接使用提供方IP引用，忽略
25. } else { // assemble URL from register center's configuration
26. List<URL> us = loadRegistries(false);//组装注册的URL
27. //URL加参数、判空，忽略
28. }
29. }
30. invoker = refprotocol.refer(interfaceClass, urls.get(0));//服务引用
31. //默认通过javassist创建代理对象
32. //如果配置了stub还要经过StubProxyFactoryWrapper进行包装一层
33. return (T) proxyFactory.getProxy(invoker);
34. }

在了解SPI拓展机制后这里Stub的创建看起来就很容易。proxyFactory.getProxy(invoker)创建出代理对象。

META-INF\dubbo\internal\com.alibaba.dubbo.rpc.ProxyFactory文件夹下的内容：

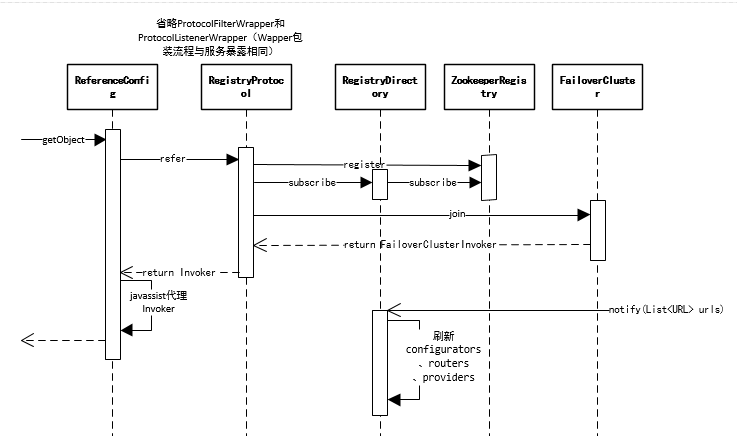
stub=com.alibaba.dubbo.rpc.proxy.wrapper.StubProxyFactoryWrapper

jdk=com.alibaba.dubbo.rpc.proxy.jdk.JdkProxyFactory

javassist=com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory

在stub、jdk、javassist中stub比较特殊，StubProxyFactoryWrapper构造函数的参数是ProxyFactory 按照之前讲的SPI机制，该类为一个包装类。这里就很轻松的看出来Stub对象是通过StubProxyFactoryWrapper这个包装类包装出来的了。

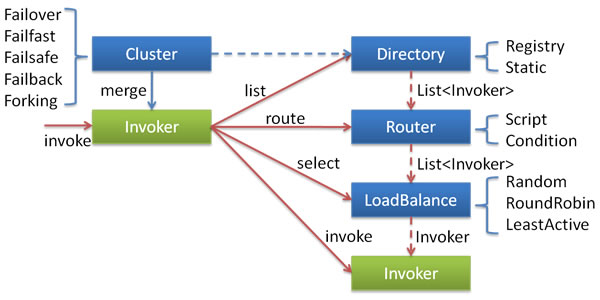
服务引用时序图（默认使用FailoverCluster）：



RegistryProtocol将自己注册到Consumer节点，再订阅相关节点（含provider），回调实现在RegistryDirectory的notify方法：

1. public synchronized void notify(List<URL> urls) {
2. //省略configurators、routers
3. // 刷新Provider列表、将provider的url列表转换为Invoker map
4. refreshInvoker(invokerUrls);
5. }

集群容错机制：



当消费方发起调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。

FailoverCluster支持失败自动切换，当出现失败，重试其它服务器。FailoverCluster的超时失败重试是通过FailoverClusterInvoker来实现的。

1. public class FailoverClusterInvoker<T> extends AbstractClusterInvoker<T> {
2. public FailoverClusterInvoker(Directory<T> directory) {
3. super(directory);
4. }
5. public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {
6. List<Invoker<T>> copyinvokers = invokers;
7. checkInvokers(copyinvokers, invocation);
8. int len = getUrl().getMethodParameter(invocation.getMethodName(), Constants.RETRIES\_KEY, Constants.DEFAULT\_RETRIES) + 1;//默认失败重试2次
9. if (len <= 0) {
10. len = 1;
11. }
12. // retry loop.
13. RpcException le = null; // last exception.
14. List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyinvokers.size()); // invoked invokers.
15. Set<String> providers = new HashSet<String>(len);
16. for (int i = 0; i < len; i++) {//重试循环体
17. //Reselect before retry to avoid a change of candidate `invokers`.
18. //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
19. if (i > 0) {
20. checkWhetherDestroyed();
21. copyinvokers = list(invocation);
22. // check again
23. checkInvokers(copyinvokers, invocation);
24. }
25. //负载算法，选择invoker
26. Invoker<T> invoker = select(loadbalance, invocation, copyinvokers, invoked);
27. invoked.add(invoker);
28. RpcContext.getContext().setInvokers((List) invoked);
29. try { //try块调用invoker
30. Result result = invoker.invoke(invocation);
31. if (le != null && logger.isWarnEnabled()) {
32. //error日志
33. }
34. return result;
35. } catch (RpcException e) {
36. if (e.isBiz()) { // 业务异常不做重试，直接抛出
37. throw e;
38. }
39. le = e;
40. } catch (Throwable e) {
41. le = new RpcException(e.getMessage(), e);
42. } finally {
43. providers.add(invoker.getUrl().getAddress());
44. }
45. }
46. throw new RpcException(le);
47. }
48. }

异常种类:

UNKNOWN\_EXCEPTION = 0;

NETWORK\_EXCEPTION = 1;

TIMEOUT\_EXCEPTION = 2;

BIZ\_EXCEPTION = 3;

FORBIDDEN\_EXCEPTION = 4;

SERIALIZATION\_EXCEPTION = 5;

重试异常抛出点：com.alibaba.dubbo.rpc.protocol.dubbo.DubboInvoker#doInvoke

1. protected Result doInvoke(final Invocation invocation) throws Throwable {
2. RpcInvocation inv = (RpcInvocation) invocation;
3. //忽略
4. try {
5. boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
6. boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
7. int timeout = getUrl().getMethodParameter(methodName, Constants.TIMEOUT\_KEY, Constants.DEFAULT\_TIMEOUT);
8. //忽略Oneway和Async 方式
9. RpcContext.getContext().setFuture(null);
10. return (Result) currentClient.request(inv, timeout).get();
11. } catch (TimeoutException e) {//超时异常
12. throw new RpcException(RpcException.TIMEOUT\_EXCEPTION, "Invoke remote method timeout. method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
13. } catch (RemotingException e) {
14. throw new RpcException(RpcException.NETWORK\_EXCEPTION, "Failed to invoke remote method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
15. }
16. }

负载均衡策略：

Dubbo中支持“一致性hash”、“轮询”、“最小活跃数”、“随机”四种均衡策略，其中“随机”是dubbo默认的负载均衡策略。

random=com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance

roundrobin=com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance

leastactive=com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance

consistenthash=com.alibaba.dubbo.rpc.cluster.loadbalance.ConsistentHashLoadBalance

下面选择RandomLoadBalance（随机）的doselect方法来看看怎么取Invoker的：

1. @Override
2. protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
3. int length = invokers.size(); // Number of invokers
4. int totalWeight = 0; // The sum of weights
5. //sameWeight标志所有的invoker权重是否都相同，都相同直接使用随机数
6. boolean sameWeight = true; // Every invoker has the same weight?
7. for (int i = 0; i < length; i++) {
8. int weight = getWeight(invokers.get(i), invocation);
9. totalWeight += weight; // Sum
10. if (sameWeight && i > 0
11. && weight != getWeight(invokers.get(i - 1), invocation)) {
12. sameWeight = false;
13. }
14. }
15. //存在权重不一致的情况
16. if (totalWeight > 0 && !sameWeight) {
17. int offset = random.nextInt(totalWeight);
18. // Return a invoker based on the random value.
19. for (int i = 0; i < length; i++) {//遍历权重值，看随机数落在哪一个invoker上
20. offset -= getWeight(invokers.get(i), invocation);
21. if (offset < 0) {
22. return invokers.get(i);
23. }
24. }
25. }
26. // 权重一致，直接使用随机数
27. return invokers.get(random.nextInt(length));
28. }

上面代码中权重不一致的情况，除了配置的时候weight属性不同以外，还有一个原因，那就是dubbo中存在预热的机制，在新注册到注册中心的provider权重会在10分钟（默认）内从0%均速增长到100%。下面来看看这段预热的代码：

1. protected int getWeight(Invoker<?> invoker, Invocation invocation) {
2. int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.WEIGHT\_KEY, Constants.DEFAULT\_WEIGHT);
3. if (weight > 0) {
4. //获取provider注册的时间戳
5. long timestamp = invoker.getUrl().getParameter(Constants.REMOTE\_TIMESTAMP\_KEY, 0L);
6. if (timestamp > 0L) {
7. int uptime = (int) (System.currentTimeMillis() - timestamp);
8. int warmup = invoker.getUrl().getParameter(Constants.WARMUP\_KEY, Constants.DEFAULT\_WARMUP);//获取总预热时长
9. if (uptime > 0 && uptime < warmup) {
10. weight = calculateWarmupWeight(uptime, warmup, weight);//计算权重
11. }
12. }
13. }
14. return weight;
15. }
16. static int calculateWarmupWeight(int uptime, int warmup, int weight) {
17. // 根据uptime、warmup 时间比值来释放权重
18. int ww = (int) ((float) uptime / ((float) warmup / (float) weight));
19. return ww < 1 ? 1 : (ww > weight ? weight : ww);
20. }