芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2018-05-01
Dubbo

精尽 Dubbo 源码分析 —— 服务引用（一）之本地引用（Injvm）

本文基于 Dubbo 2.6.1 版本，望知悉。

1. 概述

Dubbo 服务引用，和 Dubbo 服务暴露一样，也有两种方式：

本地引用，JVM 本地调用。配置如下：

```
// 推荐
<dubbo:service scope="local" />
// 不推荐使用，准备废弃
<dubbo:service injvm="true" />
```

远程暴露，网络远程通信。配置如下：

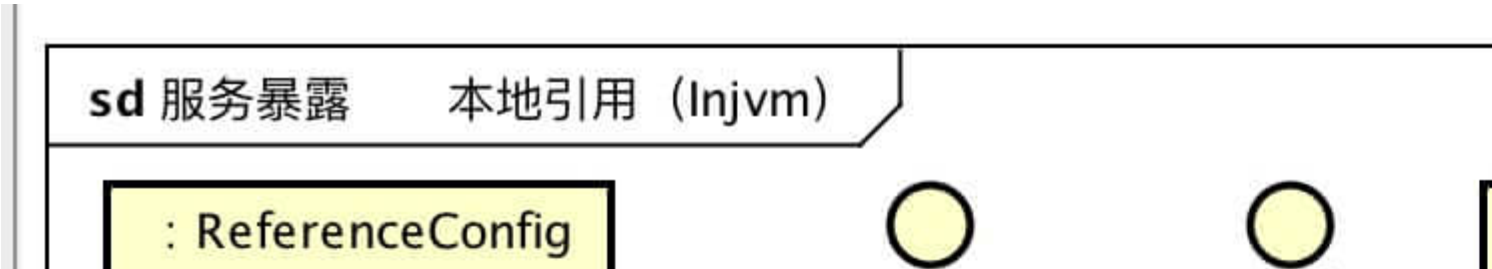
```
<dubbo:service scope="remote" />
```

我们知道 Dubbo 提供了多种协议（ Protocol ）实现。

本文仅分享本地引用，该方式仅使用 Injvm 协议实现，具体代码在 dubbo-rpc-injvm 模块中。下几篇会分享远程引用，该方式有多种协议实现，例如 Dubbo （ 默认协议 ）、Hessian 、Rest 等等。我们会每个协议对应一篇文章，进行分享。

2. createProxy

本地引用服务的顺序图如下：



在 《精尽 Dubbo 源码分析 —— API 配置（三）之服务消费者》 一文中，我们看到 ReferenceConfig#init() 方法中，会在配置初始化完成后，调用顺序图的起点 #createProxy(map) 方法，开始引用服务。代码如下：

```
/**
 * 自适应 Protocol 实现对象
 */
private static final Protocol refprotocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension()
/**
 * 自适应 ProxyFactory 实现对象
 */
private static final ProxyFactory proxyFactory = ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveEx

/**
 * 直连服务提供者地址
 */
// url for peer-to-peer invocation
private String url;
```

```
 1: private T createProxy(Map<String, String> map) {
 2:     URL tmpUrl = new URL("temp", "localhost", 0, map);
 3:     // 是否本地引用
 4:     final boolean isJvmRefer;
 5:     // injvm 属性为空，不通过该属性判断
 6:     if (isInjvm() == null) {
 7:         // 直连服务提供者，参见文档《直连提供者》http://dubbo.apache.org/zh-cn/docs/user/demos/explicit-target.h
 8:         if (url != null && url.length() > 0) { // if a url is specified, don't do local reference
 9:             isJvmRefer = false;
10:         // 通过 `tmpUrl` 判断，是否需要本地引用
11:         } else if (InjvmProtocol.getInjvmProtocol().isInjvmRefer(tmpUrl)) {
12:             // by default, reference local service if there is
13:             isJvmRefer = true;
14:         // 默认不是
15:         } else {
16:             isJvmRefer = false;
17:         }
18:     // 通过 injvm 属性。
19:     } else {
20:         isJvmRefer = isInjvm();
21:     }
22:
23:     // 本地引用
24:     if (isJvmRefer) {
25:         // 创建服务引用 URL 对象
26:         URL url = new URL(Constants.LOCAL_PROTOCOL, NetUtils.LOCALHOST, 0, interfaceClass.getName()).addParamete
27:         // 引用服务，返回 Invoker 对象
28:         invoker = refprotocol.refer(interfaceClass, url);
29:         if (logger.isInfoEnabled()) {
30:             logger.info("Using injvm service " + interfaceClass.getName());
31:         }
32:     // 正常流程，一般为远程引用
33:     } else {
34:         // ... 省略本文暂时不分享的服务远程引用
35:         }
36:     }
37:
38:     // 启动时检查
39:     Boolean c = check;
40:     if (c == null && consumer != null) {
41:         c = consumer.isCheck();
```

```
42:     }
43:     if (c == null) {
44:         c = true; // default true
45:     }
46:     if (c && !invoker.isAvailable()) {
47:         throw new IllegalStateException("Failed to check the status of the service " + interfaceName + ". No pro
48:     }
49:     if (logger.isInfoEnabled()) {
50:         logger.info("Refer dubbo service " + interfaceClass.getName() + " from url " + invoker.getUrl());
51:     }
52:
53:     // 创建 Service 代理对象
54:     // create service proxy
55:     return (T) proxyFactory.getProxy(invoker);
56: }
```

`map` 方法参数，URL 参数集合，包含服务引用配置对象的配置项。

========== 分割线 ==========

第 2 行：创建 URL 对象，重点在第四个参数，传入的是 `map` ，仅用于第 11 行，是否本地引用。

- `protocol = temp` 的原因是，在第 11 行，已经直接使用了 InjvmProtocol ，而不需要通过该值去获取。

第 4 行：是否本地引用变量 `isJvmRefer` 。

第 19 行 至 20 行：调用 `#isInjvm()` 方法，返回非空，说明配置了 `injvm` 配置项，直接使用配置项。

第 8 至 9 行：配置了 `url` 配置项，说明使用直连服务提供者的功能，则不使用本地使用。

- 《Dubbo 用户指南 —— 直连提供者》

第 11 至 13 行：调用 `InjvmProtocol#isInjvmRefer(url)` 方法，通过 `tmpUrl` 判断，是否需要本地引用。使用 `tmpUrl` ，相当于使用服务引用配置对象的配置项。该方法代码如下：

```
 1: /**
 2:  * 是否本地引用
 3:  *
 4:  * @param url URL
 5:  * @return 是否
 6:  */
 7: public boolean isInjvmRefer(URL url) {
 8:     final boolean isJvmRefer;
 9:     String scope = url.getParameter(Constants.SCOPE_KEY);
10:     // Since injvm protocol is configured explicitly, we don't need to set any extra flag, use normal refer
11:     // 当 `protocol = injvm` 时，本身已经是 jvm 协议了，走正常流程就是了。
12:     if (Constants.LOCAL_PROTOCOL.toString().equals(url.getProtocol())) {
13:         isJvmRefer = false;
14:     // 当 `scope = local` 或者 `injvm = true` 时，本地引用
15:     } else if (Constants.SCOPE_LOCAL.equals(scope) || (url.getParameter("injvm", false))) {
16:         // if it's declared as local reference
17:         // 'scope=local' is equivalent to 'injvm=true', injvm will be deprecated in the future release
18:         isJvmRefer = true;
19:     // 当 `scope = remote` 时，远程引用
20:     } else if (Constants.SCOPE_REMOTE.equals(scope)) {
21:         // it's declared as remote reference
22:         isJvmRefer = false;
23:     // 当 `generic = true` 时，即使用泛化调用，远程引用。
24:     } else if (url.getParameter(Constants.GENERIC_KEY, false)) {
25:         // generic invocation is not local reference
```

```
26:         isJvmRefer = false;
27:     // 当本地已经有该 Exporter 时，本地引用
28:     } else if (getExporter(exporterMap, url) != null) {
29:         // by default, go through local reference if there's the service exposed locally
30:         isJvmRefer = true;
31:     // 默认，远程引用
32:     } else {
33:         isJvmRefer = false;
34:     }
35:     return isJvmRefer;
36: }
```

- ========== 本地引用 ==========
  - 第 15 至 18 行：当 scope = local 或 injvm = true 时，本地引用。
  - 第 27 至 30 行：调用 #getExporter(url) 方法，判断当本地已经有 url 对应的 InjvmExporter 时，直接引用。 本地已有的服务，不必要使用远程服务，减少网络开销，提升性能。
    - 代码比较简单，已经添加中文注释，胖友点击链接查看。
    - InjvmProtocol#getExporter(url)
    - UrlUtils#isServiceKeyMatch(pattern, value)
  - ========== 远程引用 ==========
  - 第 10 至 13 行：当 protocol = injvm 时，本身已经是 Injvm 协议了，走正常流程即可。这是最特殊的，下面会更好的理解。另外，因为 #isInjvmRefer(url) 方法，仅有在 #createProxy(map) 方法中调用，因此实际也不会触发该逻辑。
  - 第 19 至 22 行：当 scope = remote 时，远程引用。
  - 第 23 至 26 行：当 generic = true 时，即使用泛化调用，远程引用。
    - 《Dubbo 用户指南 —— 泛化调用》
  - 第 31 至 34 行：默认，远程引用。

第 23 至 31 行：本地引用。

- 第 26 行：创建本地服务引用 URL 对象。
- 第 28 行：调用 Protocol#refer(interface, url) 方法，引用服务，返回 Invoker 对象。
  - 此处 Dubbo SPI 自适应的特性的好处就出来了，可以自动根据 URL 参数，获得对应的拓展实现。例如，invoker 传入后，根据 invoker.url 自动获得对应 Protocol 拓展实现为 InjvmProtocol 。
  - 实际上，Protocol 有两个 Wrapper 拓展实现类： ProtocolFilterWrapper、ProtocolListenerWrapper 。所以，#refer(...) 方法的调用顺序是 ：Protocol$Adaptive => ProtocolFilterWrapper => ProtocolListenerWrapper => InjvmProtocol 。
  - 详细的调用，在 「3. Protocol」 在解析。

第 32 至 36 行：正常流程，一般为远程引用。为什么是一般呢？如果我们配置 protocol = injvm ，实际走的是本地引用。例如：

```
<dubbo:reference protocol="injvm" >
</dubbo:reference>
```

- 当然，笔者建议，如果真的是需要本地应用，建议配置 scope = local 。这样，会更加明确和清晰。

第 38 至 51 行：若配置 check = true 配置项时，调用 Invoker#isAvailable() 方法，启动时检查

。

- 该方法在 「4.2 InjvmInvoker」 ，详细分享。
- 《Dubbo 用户指南 —— 启动时检查》

第 55 行：调用 ProxyFactory#getProxy(invoker) 方法，创建 Service 代理对象。该 Service 代理对象的内部，会调用 Invoker#invoke(Invocation) 方法，进行 Dubbo 服务的调用。

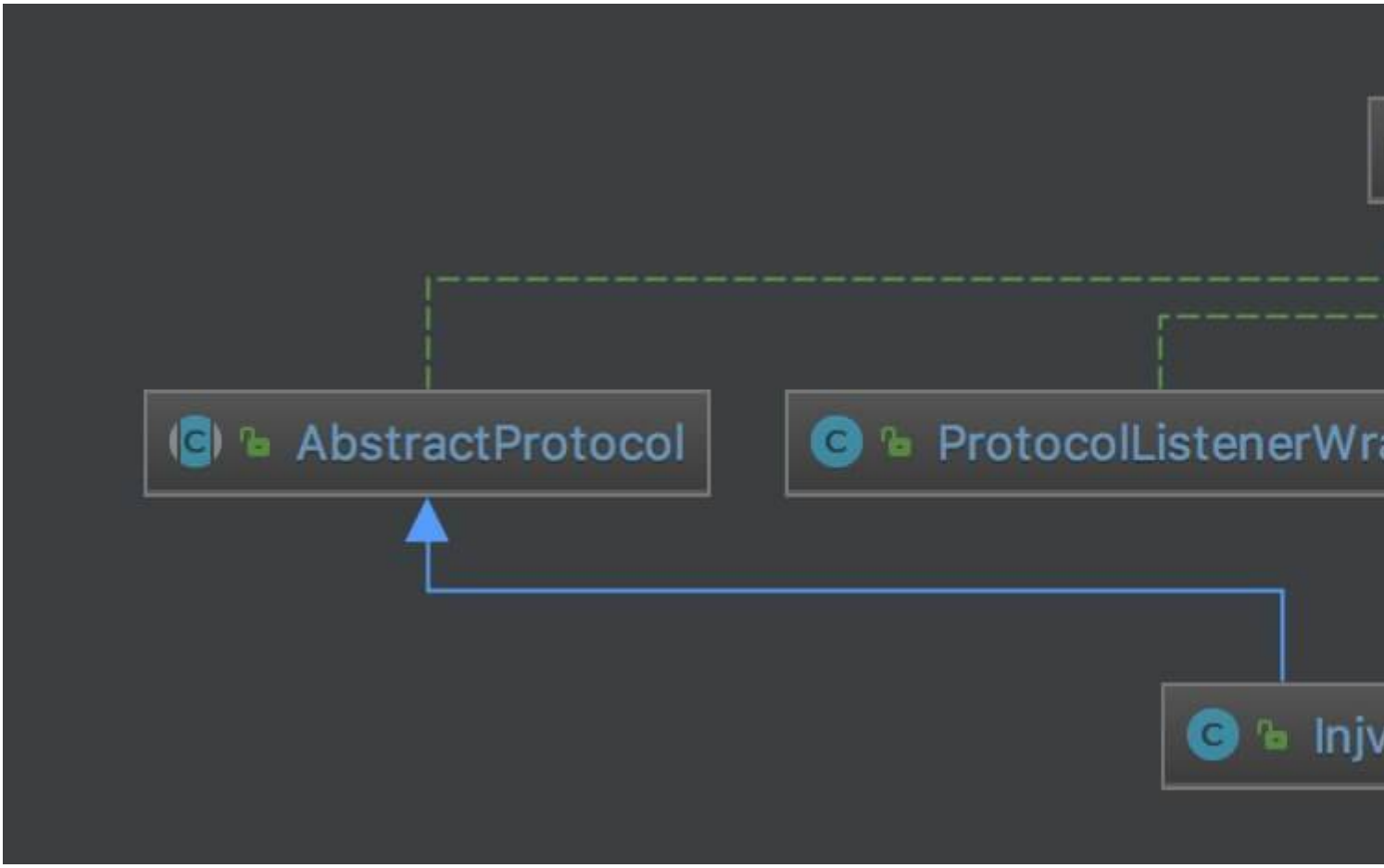
- 详细的实现，后面单独写文章分享。

3. Protocol

服务引用与暴露的 Protocol 很多类似点，本文就不重复叙述了。

建议不熟悉的胖友，请点击 《精尽 Dubbo 源码分析 —— 服务暴露（一）之本地暴露（Injvm）》「3. Protocol」 查看。

本文涉及的 Protocol 类图如下：



3.1 ProtocolFilterWrapper

3.1.1 refer

本文涉及的 #refer(type, url) 方法，代码如下：

```
1: public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
```

```
2:     // 注册中心
3:     if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
4:         return protocol.refer(type, url);
5:     }
6:     // 引用服务，返回 Invoker 对象
7:     // 给改 Invoker 对象，包装成带有 Filter 过滤链的 Invoker 对象
8:     return buildInvokerChain(protocol.refer(type, url), Constants.REFERENCE_FILTER_KEY, Constants.CONSUMER);
9: }
```

第 2 至 5 行：当 `invoker.url.protocol = registry` ，跳过，本地引用服务不会符合这个判断。在远程引用服务会符合暴露该判断，所以下一篇文章分享。
第 8 行：调用 `protocol#refer(type, url)` 方法，继续引用服务，最终返回 Invoker 。
第 8 行：在引用服务完成后，调用 `#buildInvokerChain(invoker, key, group)` 方法，创建带有 Filter 过滤链的 Invoker 对象。

3.1.2 buildInvokerChain

和 《精尽 Dubbo 源码分析 —— 服务暴露（一）之本地暴露（Injvm）》「3.1.3 buildInvokerChain」 基本一致，默认情况下，获得的 Filter 数组如下：

```
ConsumerContextFilter
FutureFilter
MonitorFilter
```

当然，因为传入的参数 `group` 不同，如果胖友自定义了自动激活的 Filter 只出现在 `group = consumer` ，那么服务消费者就会多一个该 Filter 实现。

3.2 ProtocolListenerWrapper

本文涉及的 `#refer(type, url)` 方法，代码如下：

```
1: public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
2:     // 注册中心协议
3:     if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
4:         return protocol.refer(type, url);
5:     }
6:     // 引用服务
7:     Invoker<T> invoker = protocol.refer(type, url);
8:     // 获得 InvokerListener 数组
9:     List<InvokerListener> listeners = Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(InvokerList
10:    // 创建 ListenerInvokerWrapper 对象
11:    return new ListenerInvokerWrapper<T>(invoker, listeners);
12: }
```

第 2 至 5 行：当 `invoker.url.protocol = registry` ，跳过，本地引用服务不会符合这个判断。在远程引用服务会符合暴露该判断，所以下一篇文章分享。
第 7 行：调用 `protocol#refer(type, url)` 方法，继续引用服务，最终返回 Invoker 。
第 9 行：调用 `ExtensionLoader#getActivateExtension(url, key, group)` 方法，获得监听器数组。
- 不熟悉的胖友，请看 《精尽 Dubbo 源码分析 —— 拓展机制 SPI》 文章。
- 继续以上面的例子为基础，`listeners` 为空。胖友可以自行实现 ExporterListener ，并进行配置 `@Activate` 注解，或者 XML 中 `listener` 属性。

第 11 行：创建带 InvokerListener 的 ListenerInvokerWrapper 对象。在这个过程中，会

执行 ExporterListener#referred(invoker) 方法。
    - 在 「4.3 ListenerInvokerWrapper」 详细解析。

3.3 InjvmProtocol

本文涉及的 #refer(type, url) 方法，代码如下：

```
public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {
    return new InjvmInvoker<T>(serviceType, url, url.getServiceKey(), exporterMap);
}
```

创建 InjvmInvoker 对象。注意，传入的 exporterMap 参数，包含所有的 InjvmExporter 对象。

4. Invoker

Exporter 接口，在 《精尽 Dubbo 源码分析 —— 核心流程一览》「4.1 Invoker」 有详细解析。

本文涉及的 Invoker 类图如下：

4.1 AbstractInvoker

com.alibaba.dubbo.rpc.protocol.AbstractInvoker ，实现 Invoker 接口，抽象 Invoker 类，主要提供了 Invoker 的通用属性和 #invoke(Invocation) 方法的通用实现。

本文主要涉及到它的通用属性，代码如下：

```
/**
 * 接口类型
 */
private final Class<T> type;
/**
 * 服务 URL
 */
private final URL url;
/**
 * 公用的隐式传参。在 {@link #invoke(Invocation)} 方法中使用。
 */
private final Map<String, String> attachment;
/**
 * 是否可用
 */
private volatile boolean available = true;
/**
 * 是否销毁
 */
private AtomicBoolean destroyed = new AtomicBoolean(false);
```

ps: #invoke(Invocation) 方法，在后续的文章分享。

4.2 InjvmInvoker

com.alibaba.dubbo.rpc.protocol.injvm.InjvmInvoker ，实现 AbstractInvoker 抽象类，Injvm Invoker 实现类。

4.2.1 属性

```
/**
 * 服务键
 */
private final String key;
/**
 * Exporter 集合
 *
 * key: 服务键
 *
 * 该值实际就是 {@link com.alibaba.dubbo.rpc.protocol.AbstractProtocol#exporterMap}
 */
private final Map<String, Exporter<?>> exporterMap;

InjvmInvoker(Class<T> type, URL url, String key, Map<String, Exporter<?>> exporterMap) {
    super(type, url);
    this.key = key;
    this.exporterMap = exporterMap;
}
```

key 属性，服务键。
exporterMap 属性，Exporter 集合。在 InjvmInvoker#invoke(invocation) 方法中，通过该 Invoker 的 key 属性，获得对应的 Exporter 对象。

4.2.2 isAvailable

#isAvailable() 方法，是否可用。代码如下：

```
@Override
public boolean isAvailable() {
    // 判断是否有 Exporter 对象
    InjvmExporter<?> exporter = (InjvmExporter<?>) exporterMap.get(key);
    if (exporter == null) {
        return false;
    } else {
        return super.isAvailable();
    }
}
```

开启 启动时检查 时，调用该方法，判断该 Invoker 对象，是否有对应的 Exporter 。若不存在，说明依赖服务不存在，检查不通过。

4.3 ListenerInvokerWrapper

com.alibaba.dubbo.rpc.listener.ListenerInvokerWrapper ，实现 Invoker 接口，具有监听器功能的 Invoker 包装器。代码如下：

```
public class ListenerInvokerWrapper<T> implements Invoker<T> {
```

```
private static final Logger logger = LoggerFactory.getLogger(ListenerInvokerWrapper.class);

/**
 * 真实的 Invoker 对象
 */
private final Invoker<T> invoker;
/**
 * Invoker 监听器数组
 */
private final List<InvokerListener> listeners;

public ListenerInvokerWrapper(Invoker<T> invoker, List<InvokerListener> listeners) {
    if (invoker == null) {
        throw new IllegalArgumentException("invoker == null");
    }
    this.invoker = invoker;
    this.listeners = listeners;
    // 执行监听器
    if (listeners != null && !listeners.isEmpty()) {
        for (InvokerListener listener : listeners) {
            if (listener != null) {
                try {
                    listener.referred(invoker);
                } catch (Throwable t) {
                    logger.error(t.getMessage(), t);
                }
            }
        }
    }
}

public Class<T> getInterface() {
    return invoker.getInterface();
}

public URL getUrl() {
    return invoker.getUrl();
}

public boolean isAvailable() {
    return invoker.isAvailable();
}

public Result invoke(Invocation invocation) throws RpcException {
    return invoker.invoke(invocation);
}

@Override
public String toString() {
    return getInterface() + " -> " + (getUrl() == null ? "" : getUrl().toString());
}

public void destroy() {
    try {
        invoker.destroy();
    } finally {
        // 执行监听器
        if (listeners != null && !listeners.isEmpty()) {
            for (InvokerListener listener : listeners) {
```

```
                    if (listener != null) {
                        try {
                            listener.destroyed(invoker);
                        } catch (Throwable t) {
                            logger.error(t.getMessage(), t);
                        }
                    }
                }
            }
        }
    }

}
```

构造方法，循环 listeners ，执行 InvokerListener#referred(invoker) 方法。 和 ListenerExporterWrapper 不同，若执行过程中发生异常 RuntimeException ，仅打印错误日志，继续执行，最终不抛出异常。

#unexport() 方法，循环 listeners ，执行 InvokerListener#destroyed(invoker) 。 和 ListenerExporterWrapper 不同，若执行过程中发生异常 RuntimeException ，仅打印错误日志，继续执行，最终不抛出异常。

5. InvokerListener

com.alibaba.dubbo.rpc.InvokerListener ， Invoker 监听器。

代码如下：

```
@SPI
public interface InvokerListener {

    /**
     * The invoker referred
     *
     * 当服务引用完成
     *
     * @param invoker
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Protocol#refer(Class, URL)
     */
    void referred(Invoker<?> invoker) throws RpcException;

    /**
     * The invoker destroyed.
     *
     * 当服务销毁引用完成
     *
     * @param invoker
     * @see com.alibaba.dubbo.rpc.Invoker#destroy()
     */
    void destroyed(Invoker<?> invoker);

}
```

5.1 InvokerListenerAdapter

com.alibaba.dubbo.rpc.listener.InvokerListenerAdapter ，实现 InvokerListener 接口，InvokerListener 适配器抽象类。代码如下：

```
public abstract class InvokerListenerAdapter implements InvokerListener {

    public void referred(Invoker<?> invoker) throws RpcException { }

    public void destroyed(Invoker<?> invoker) { }

}
```

5.2 DeprecatedInvokerListener

com.alibaba.dubbo.rpc.listener.DeprecatedInvokerListener ，实现 InvokerListenerAdapter 抽象类 ，引用废弃的服务时，打印错误日志提醒。代码如下：

```
@Activate(Constants.DEPRECATED_KEY)
public class DeprecatedInvokerListener extends InvokerListenerAdapter {

    private static final Logger LOGGER = LoggerFactory.getLogger(DeprecatedInvokerListener.class);

    public void referred(Invoker<?> invoker) throws RpcException {
        if (invoker.getUrl().getParameter(Constants.DEPRECATED_KEY, false)) {
            LOGGER.error("The service " + invoker.getInterface().getName() + " is DEPRECATED! Declare from " + invoke
        }
    }

}
```

`@Activate(Constants.DEPRECATED_KEY)` 注解，基于 Dubbo SPI Activate 机制加载。配置方式如下：

```
        <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService" deprecated="true" />
        ```
        * 通过设置 `"deprecated"` 为 `true` 来设置。
        * 该方式仅适用于**远程引用**服务。

    * 在 `#referred(invoker)` 方法中，打印错误日志，例如：

        ```Java
        [25/03/18 07:37:56:056 CST] main ERROR listener.DeprecatedInvokerListener:  [DUBBO] The service com.alibaba
        group:consumer
```

另外，本地引用服务的配置方式如下：

```
<dubbo:reference id="demoService" interface="com.alibaba.dubbo.demo.DemoService" protocol="injvm">
    <dubbo:parameter key="deprecated" value="true" />
</dubbo:reference>
```

因为，本地引用服务时，不是使用服务提供者的 URL ，而是服务消费者的 URL 。

666. 彩蛋

连续熬夜几天，要调整下作息了。

文章目录

回到首页