title: 精尽 Dubbo 源码分析 —— 服务暴露（一）之本地暴露（Injvm） date: 2018-03-07 tags: categories: Dubbo permalink: Dubbo/service-export-local

---

摘要: 原创出处 http://www.iocoder.cn/Dubbo/service-export-local/ 「芋道源码」欢迎转载，保留摘要，谢谢！



---

> 1. RocketMQ / MyCAT / Sharding-JDBC **所有**源码分析文章列表
> 2. RocketMQ / MyCAT / Sharding-JDBC **中文注释源码 GitHub 地址**
> 3. 您对于源码的疑问每条留言**都**将得到**认真**回复。**甚至不知道如何读源码也可以请教噢**。
> 4. **新的**源码解析文章**实时**收到通知。**每周更新一篇左右**。
> 5. **认真的**源码交流微信群。

---

# 1. 概述

Dubbo 服务暴露有两种方式

- 本地暴露，JVM 本地调用。配置如下：

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

- 远程暴露，网络远程通信。配置如下：

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

- 【可以不算】不暴露，配置如下：

```
<dubbo:service scope="none" />
```

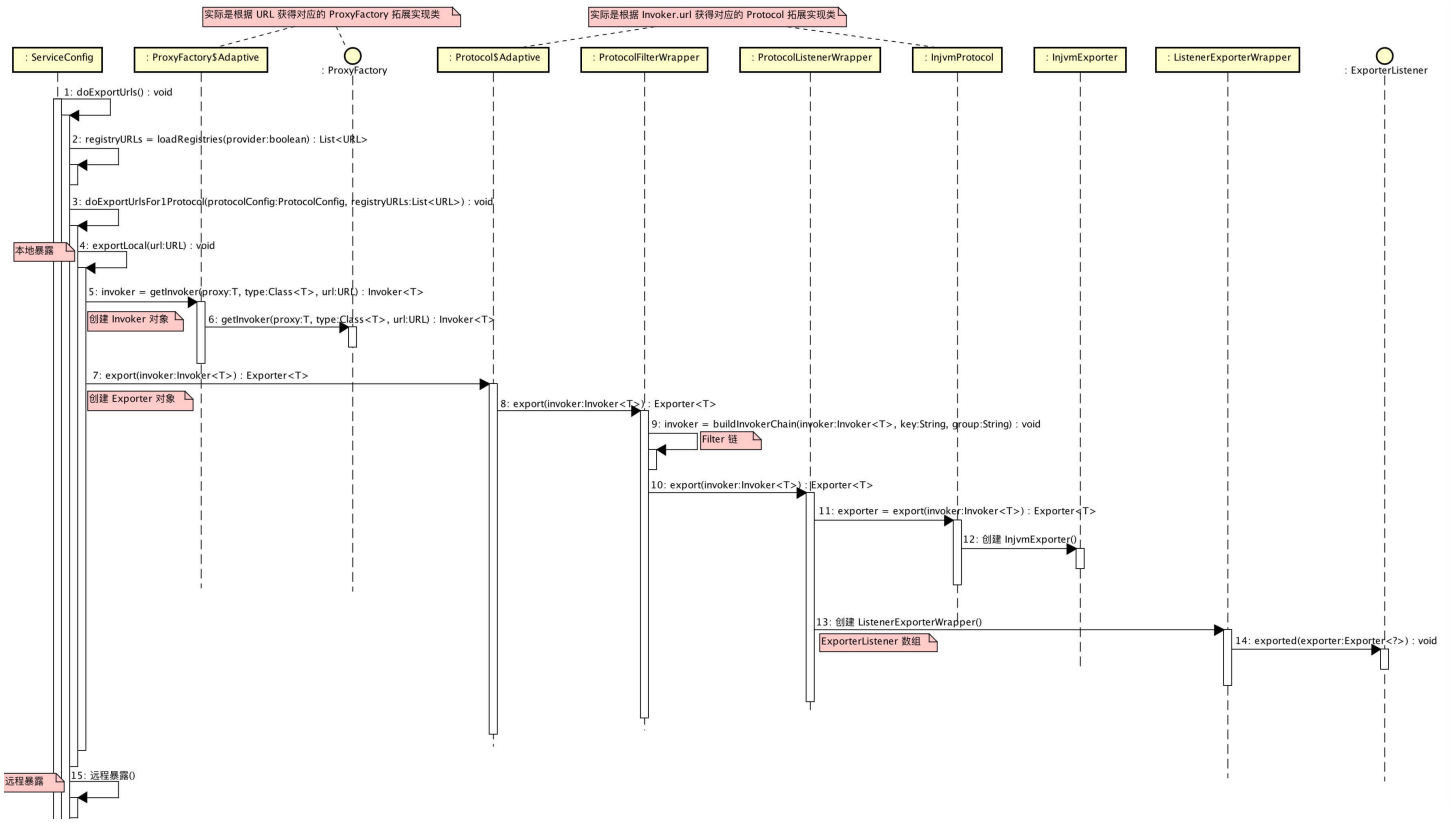
在不配置 `scope` 的情况下，**默认两种方式都暴露**。因为，Dubbo 自身无法确认应用中，是否存在本地引用的情况。 大多数情况下，我们不需要配置 `scope` 。如果胖友可以明确服务器消费引用服务的方式，也可以进行设置。

我们知道 Dubbo 提供了多种协议( Protocol )实现。

- **本文**仅分享本地暴露，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**会分享远程暴露，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. doExportUrls

本地暴露服务的顺序图如下：




在 《精尽 Dubbo 源码分析 —— API 配置（二）之服务提供者》 一文中，我们看到 `ServiceConfig#export()` 方法中，会在配置初始化完成后，调用顺序图的**起点** `#doExportUrls()` 方法，开始暴露服务。代码如下：

```
/**
 * 注册中心配置数组
 */
protected List<ProtocolConfig> protocols;
```

```
 1: /**
 2:  * 暴露 Dubbo URL
 3:  */
 4: @SuppressWarnings({"unchecked", "rawtypes"})
 5: private void doExportUrls() {
 6:     // 加载注册中心 URL 数组
 7:     List<URL> registryURLs = loadRegistries(true);
 8:     // 循环 `protocols` ，向逐个注册中心分组暴露服务。
 9:     for (ProtocolConfig protocolConfig : protocols) {
10:         doExportUrlsFor1Protocol(protocolConfig, registryURLs);
11:     }
12: }
```

- 第 7 行：调用 `#loadRegistries(provider)` 方法，加载注册中心的 com.alibaba.dubbo.common.URL` 数组。

    - 🙂 在 「2.1 loadRegistries」 详细解析。
- 第 8 至 11 行：循环 `protocols` ，调用 `#doExportUrlsFor1Protocol(protocolConfig, registryURLs)` 方法，使用**对应的协议**，逐个向**注册中心分组**暴露服务。在这个方法中，包含了本地和远程两种暴露方式。在下文中，我们会看到，本地暴露不会向注册中心注册服务，**因为仅仅用于 JVM 内部本地调用，内存中已经有相关信息**。
    - 🙂 在 「2.2 doExportUrlsFor1Protocol」 详细解析。

# 2.1 loadRegistries

> 友情提示，实际这个方法，放在服务的**远程暴露**一文分享较为合适。
> 因为，**本地暴露**无需向注册中心注册。
> 所以，该方法也可以放到**远程暴露**一文在看。

`#loadRegistries(provider)` 方法，加载注册中心 `com.alibaba.dubbo.common.URL` 数组。代码如下：

```
 1: /**
 2:  * 加载注册中心 URL 数组
 3:  *
 4:  * @param provider 是否是服务提供者
 5:  * @return URL 数组
 6:  */
 7: protected List<URL> loadRegistries(boolean provider) {
 8:     // 校验 RegistryConfig 配置数组。
 9:     checkRegistry();
10:     // 创建 注册中心 URL 数组
11:     List<URL> registryList = new ArrayList<URL>();
12:     if (registries != null && !registries.isEmpty()) {
13:         for (RegistryConfig config : registries) {
14:             // 获得注册中心的地址
15:             String address = config.getAddress();
16:             if (address == null || address.length() == 0) {
17:                 address = Constants.ANYHOST_VALUE;
18:             }
19:             String sysaddress = System.getProperty("dubbo.registry.address");
// 从启动参数读取
20:             if (sysaddress != null && sysaddress.length() > 0) {
21:                 address = sysaddress;
22:             }
23:             // 有效的地址
24:             if (address.length() > 0
25:                     && !RegistryConfig.NO_AVAILABLE.equalsIgnoreCase(address))
{
26:                 Map<String, String> map = new HashMap<String, String>();
```

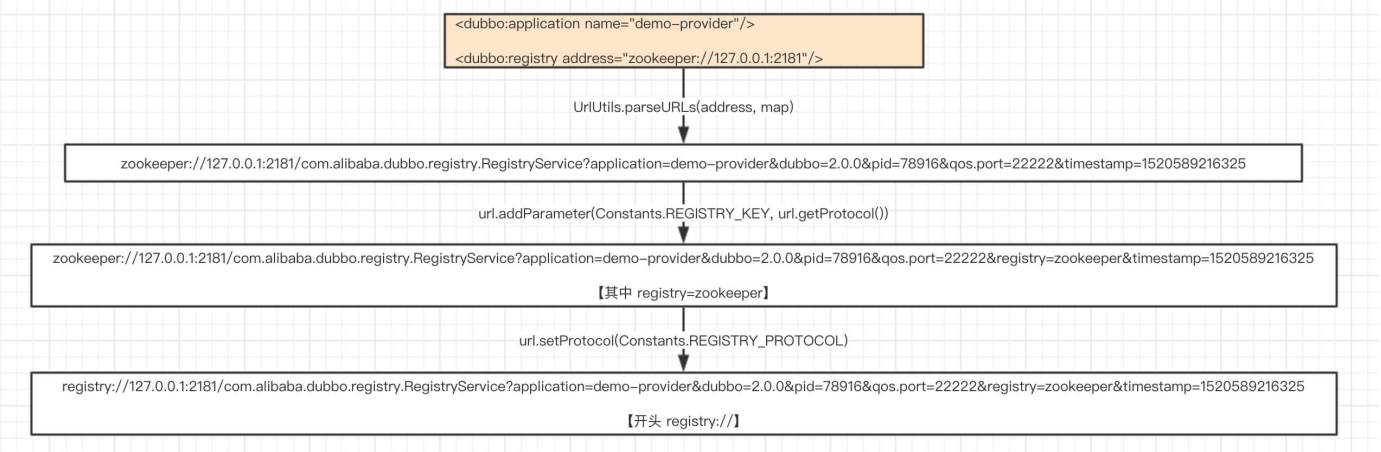
```
 27:                 // 将各种配置对象，添加到 `map` 集合中。
 28:                 appendParameters(map, application);
 29:                 appendParameters(map, config);
 30:                 // 添加 `path` `dubbo` `timestamp` `pid` 到 `map` 集合中。
 31:                 map.put("path", RegistryService.class.getName());
 32:                 map.put("dubbo", Version.getVersion());
 33:                 map.put(Constants.TIMESTAMP_KEY, String.valueOf(System.current
TimeMillis()));
 34:                 if (ConfigUtils.getPid() > 0) {
 35:                     map.put(Constants.PID_KEY, String.valueOf(ConfigUtils.getP
id()));
 36:                 }
 37:                 // 若不存在 `protocol` 参数，默认 "dubbo" 添加到 `map` 集合中。
 38:                 if (!map.containsKey("protocol")) {
 39:                     if (ExtensionLoader.getExtensionLoader(RegistryFactory.cla
ss).hasExtension("remote")) { // "remote"
 40:                         map.put("protocol", "remote");
 41:                     } else {
 42:                         map.put("protocol", "dubbo");
 43:                     }
 44:                 }
 45:                 // 解析地址，创建 Dubbo URL 数组。（数组大小可以为一）
 46:                 List<URL> urls = UrlUtils.parseURLs(address, map);
 47:                 // 循环 `url` ，设置 "registry" 和 "protocol" 属性。
 48:                 for (URL url : urls) {
 49:                     // 设置 `registry=${protocol}` 和 `protocol=registry` 到 UR
L
 50:                     url = url.addParameter(Constants.REGISTRY_KEY, url.getProt
ocol());
 51:                     url = url.setProtocol(Constants.REGISTRY_PROTOCOL);
 52:                     // 添加到结果
 53:                     if ((provider && url.getParameter(Constants.REGISTER_KEY,
true)) // 服务提供者 && 注册
 54:                             || (!provider && url.getParameter(Constants.SUBSCR
IBE_KEY, true))) { // 服务消费者 && 订阅
 55:                         registryList.add(url);
 56:                     }
 57:                 }
 58:             }
 59:         }
 60:     }
 61:     return registryList;
 62: }
```

- 第 9 行：调用 `#checkRegistry()` 方法，校验 RegistryConfig 配置。
  - 🙂 直接点击方法查看，较为简单，已经添加详细注释。
- 第 11 行：**创建**注册中心 URL 数组。

- 第 13 行：**循环**注册中心配置对象数组 `registries` 。
- 第 14 至 22 行：获得注册中心的地址 `address` 。
  - 第 19 至 22 行：从启动参数 `dubbo.registry.address` 读取，若存在，**最高优先级**，进行覆盖。
- 第 24 至 25 行：地址 `address` 是有效地址，包括 `address != N/A` 。
  - `"N/A"` 代表不配置注册中心。
- 第 26 行：创建参数集合 `map` 。
- 第 28 至 29 行：调用 `#appendParameters(map, config)` 方法，将各种配置对象，添加到 `map` 集合中。
  - 🙂 在 《精尽 Dubbo 源码分析 —— API 配置（一）之应用》「3.1 AbstractConfig」已经详细解析。
  - `RegistryConfig#getAddress()` 上有 `@Parameter(excluded = true)` ，因此 `Registry.address` 属性不会添加到 `map` 中。
- 第 30 至 36 行：添加 `path` `dubbo` `timestamp` `pid` 到 `map` 集合中。
- 第 37 至 44 行：若不存在 `protocol` 参数，缺省默认为 "dubbo" ，并添加到 `map` 集合中。
  - 第 39 至 40 行：**可以忽略**。因为， `remote` 这个拓展实现已经不存在。
- 第 46 行：调用 `UrlUtils#parseURLs(address, map)` 方法，解析 `address` ，创建 Dubbo URL 数组（数组大小可以为一）。
  - 🙂 已经添加了代码注释，胖友点击链接查看。
  - `address` 可以使用 `"|"` 或者 `";"` 作为分隔符，设置多个注册中心**分组**。**注意**，一个注册中心集群是一个分组，而不是多个。
  - 这个方法从名字上很难看出具体的含义，看下图的**注释**，相信胖友能理解。

```
/**
 * 解析单个 URL ，将 `defaults` 里的参数，合并到 `address` 中。
 *
 * 合并的逻辑如下：
 *
 * 我们可以把 `address` 认为是 url ；`defaults` 认为是 defaultURL 。
 * 若 url 有不存在的属性时，从 defaultURL 获得对应的属性，设置到 url 中。
 *
 * @param address 地址
 * @param defaults 默认参数集合
 * @return URL
 */
public static URL parseURL(String address, Map<String, String> defaults) {
```

- 第 47 至 57 行：循环 `urls` ，设置 "registry" 和 "protocol" 属性。
  - 第 49 至 51 行：设置 `registry=${protocol}` 和 `protocol=registry` 到 URL 。
  - 第 53 行：若是服务**提供者**，判断是否**只订阅不注册**。如果是，**不添加**结果到 `registryList` 中。对应 《Dubbo 用户指南 —— 只订阅》 文档。
  - 第 54 行：若是服务**消费者**，判断是否**只注册不订阅**。如果是，**不添加**到结果 `registryList` 。对应 《Dubbo 用户指南 —— 只注册》 文档。
  - 第 53 至 56 行：添加到结果 `registryList` 。
- 为了胖友更加容易理解，我们举个例子。




# 2.2 doExportUrlsFor1Protocol

`#doExportUrlsFor1Protocol(protocolConfig, registryURLs)` 方法，基于单个协议，暴露服务。**简化**代码如下：

```
 1: /**
 2:  * 基于单个协议，暴露服务
 3:  *
 4:  * @param protocolConfig 协议配置对象
 5:  * @param registryURLs 注册中心链接对象数组
 6:  */
 7: private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL>
registryURLs) {
 8:     // ... 【省略】 创建服务 URL 对象
 9:
10:     String scope = url.getParameter(Constants.SCOPE_KEY);
11:     // don't export when none is configured
12:     if (!Constants.SCOPE_NONE.toString().equalsIgnoreCase(scope)) {
13:
14:         // export to local if the config is not remote (export to remote only
when config is remote)
15:         if (!Constants.SCOPE_REMOTE.toString().equalsIgnoreCase(scope)) {
16:             exportLocal(url);
17:         }
18:         // export to remote if the config is not local (export to local only w
hen config is local)
19:         if (!Constants.SCOPE_LOCAL.toString().equalsIgnoreCase(scope)) {
20:             // ... 【省略】远程暴露
21:         }
22:     }
23:     this.urls.add(url);
24: }
```

- 第 8 行：【省略代码】创建服务 URL 对象。
  - 🙂 在《精尽 Dubbo 源码分析 —— API 配置（二）之服务提供者》「8. ServiceConfig」有详细解析。
- 第 15 至 17 行：当非 `"remote"` 时，调用 `#exportLocal(url)` 方法，本地暴露服务。
  - 🙂 在「2.2.1 exportLocal」详细解析。
- 第 19 至 21 行：【省略代码】非 `"local"` 时，远程暴露服务。在下一篇文章，详细分享。

## 2.2.1 exportLocal

`#exportLocal(url)` 方法，本地暴露服务。代码如下：

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

/**
 * {@link #interfaceName} 对应的接口类
 *
 * 非配置
 */
private Class<?> interfaceClass;
/**
 * Service 对象
 */
// reference to interface impl
private T ref;
/**
 * 服务配置暴露的 Exporter 。
 * URL ：Exporter 不一定是 1：1 的关系。
 * 例如 {@link #scope} 未设置时，会暴露 Local + Remote 两个，也就是 URL ：Exporter = 1：2
 *      {@link #scope} 设置为空时，不会暴露，也就是 URL ：Exporter = 1：0
 *      {@link #scope} 设置为 Local 或 Remote 任一时，会暴露 Local 或 Remote 一个，也就是 URL ：Exporter = 1：1
 *
 * 非配置。
 */
private final List<Exporter<?>> exporters = new ArrayList<Exporter<?>>();
```

```
 1: /**
 2:  * 本地暴露服务
 3:  *
 4:  * @param url 注册中心 URL
 5:  */
 6: @SuppressWarnings({"unchecked", "rawtypes"})
 7: private void exportLocal(URL url) {
 8:     if (!Constants.LOCAL_PROTOCOL.equalsIgnoreCase(url.getProtocol())) {
 9:         // 创建本地 Dubbo URL
10:         URL local = URL.valueOf(url.toFullString())
11:                 .setProtocol(Constants.LOCAL_PROTOCOL) // injvm
12:                 .setHost(LOCALHOST) // 本地
13:                 .setPort(0); // 端口=0
14:         // 添加服务的真实类名，例如 DemoServiceImpl ，仅用于 RestProtocol 中。
15:         ServiceClassHolder.getInstance().pushServiceClass(getServiceClass(ref)
);
16:         // 使用 ProxyFactory 创建 Invoker 对象
17:         // 使用 Protocol 暴露 Invoker 对象
18:         Exporter<?> exporter = protocol.export(proxyFactory.getInvoker(ref, (C
lass) interfaceClass, local));
19:         // 添加到 `exporters`
20:         exporters.add(exporter);
21:         logger.info("Export dubbo service " + interfaceClass.getName() + " to
local registry");
22:     }
23: }
```

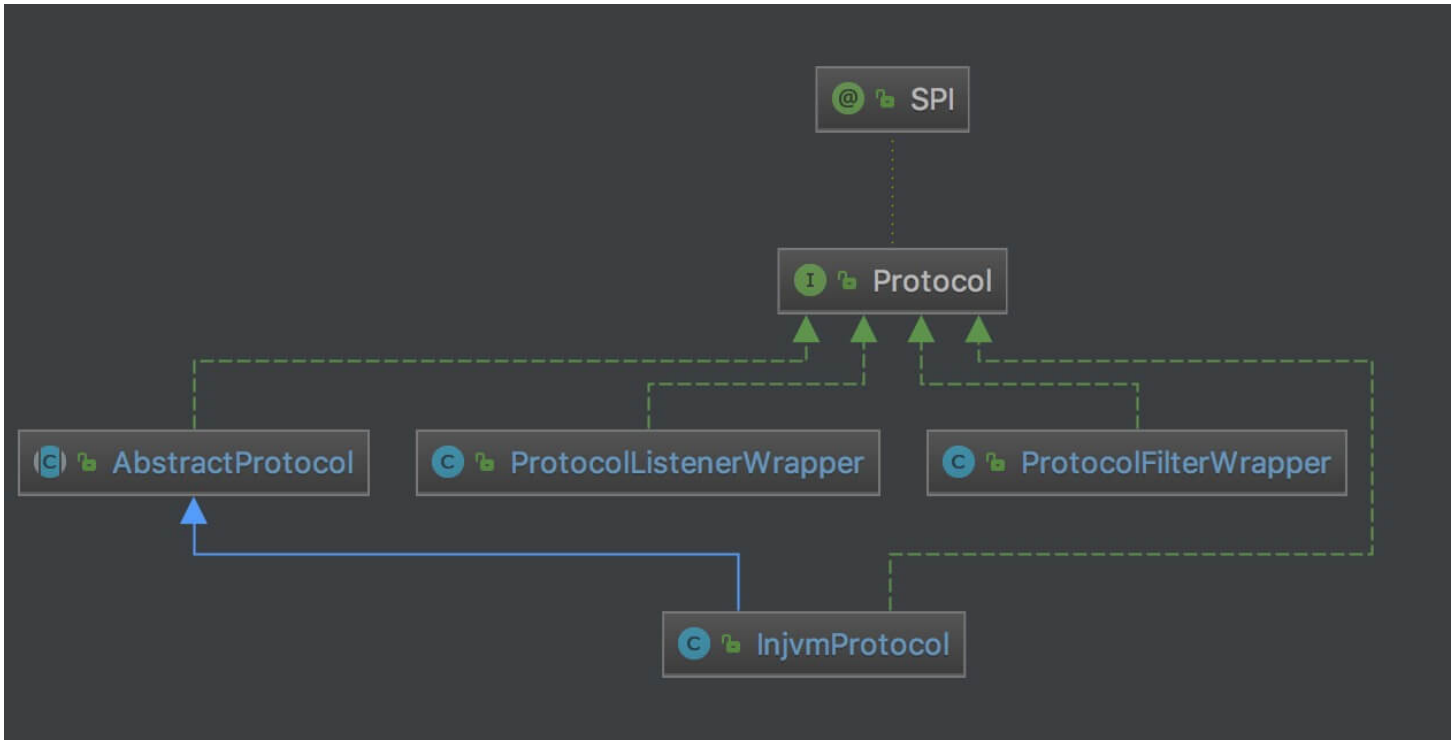
- `protocol` **静态**属性，自适应 Protocol 实现对象。
  - 不熟悉的胖友，请看 《精尽 Dubbo 源码分析 —— 拓展机制 SPI》 文章。
- `proxyFactory` **静态**属性，自适应 ProxyFactory 实现对象。
- `interfaceClass` 属性，Service 接口。
- `ref` 属性，Service 对象。
- `exporters` 属性，Exporter 集合。
- ========== 分割线 ==========
- 第 9 至 13 行：基于原有 `url` ，创建**新的**服务的本地 Dubbo URL 对象，并设置属性 `protocol=injvm` `host=127.0.0.1` `port=0` 。因为 `url` 在后面远程暴露服务会使用，所以要新创建。
- 第 15 行：添加服务的真实类名，例如 DemoServiceImpl ，仅用于 RestProtocol 中。
- 第 18 行：调用 `ProxyFactory#getInvoker(proxy, type, url)` 方法，创建 Invoker 对象。该 Invoker 对象，执行 `#invoke(invocation)` 方法时，内部会调用 Service 对象( `ref` )对应的调用方法。
  - 🙂 详细的实现，后面单独写文章分享。
- 第 18 行：调用 `Protocol#export(invoker)` 方法，暴露服务。

    - 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，`#export(...)` 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 20 行：添加到 `exporters` 集合中。

# 3. Protocol

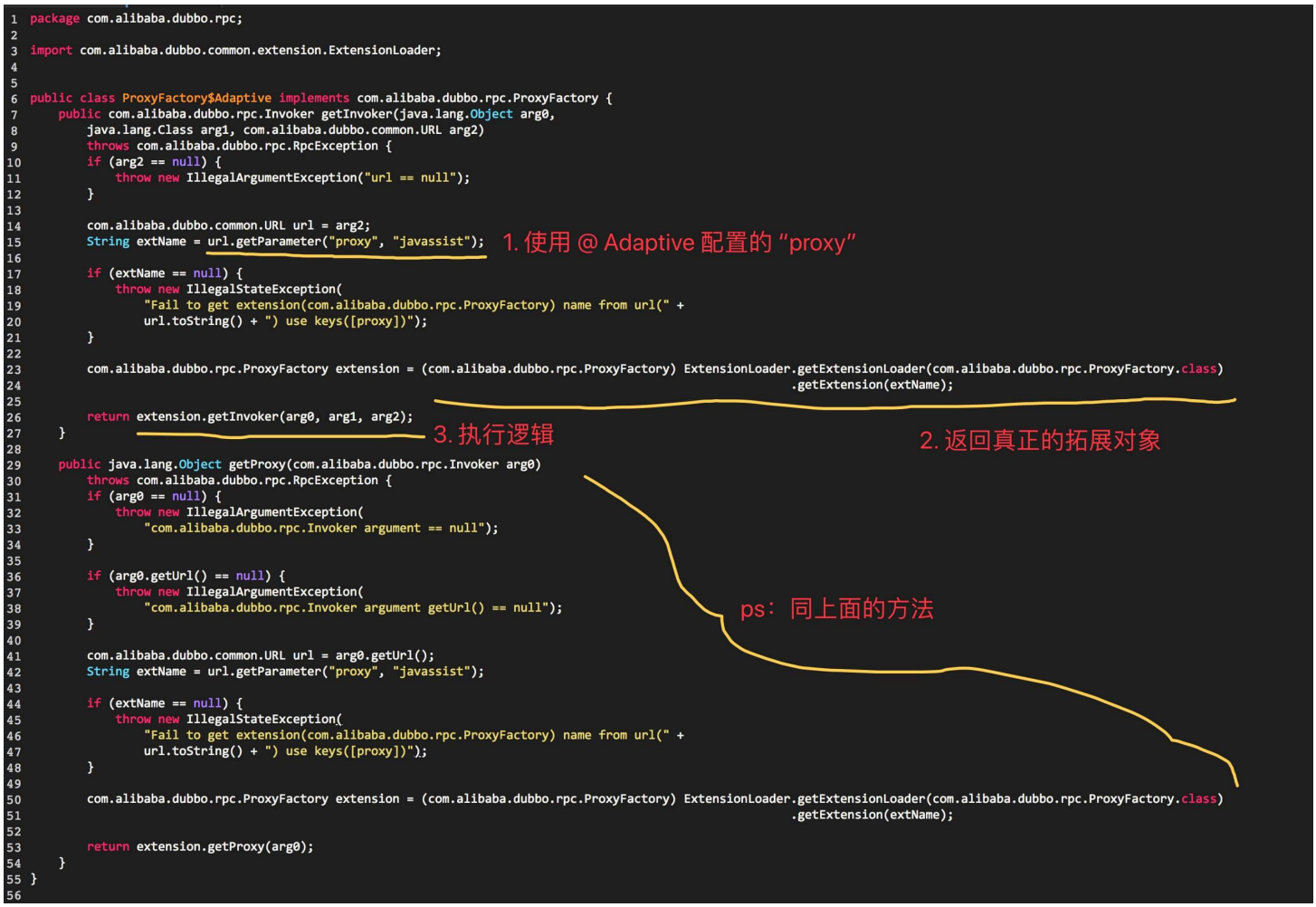
Protocol **接口**，在 《精尽 Dubbo 源码分析 —— 核心流程一览》「4.6 Protocol」 有详细解析。

本文涉及的 Protocol 类图如下：




## 3.1 Protocol$Adaptive

代码类似下图，笔者偷懒，就不去重新生成啦。




# 3.2 ProtocolFilterWrapper

`com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper` ，实现 Protocol 接口，Protocol 的 Wrapper 拓展实现类，用于给 Invoker 增加过滤链。

## 3.2.1 export

本文涉及的 `#export(invoker)` 方法，代码如下：

```
  1: public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
  2:     // 注册中心
  3:     if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
  4:         return protocol.export(invoker);
  5:     }
  6:     // 建立带有 Filter 过滤链的 Invoker ，再暴露服务。
  7:     return protocol.export(buildInvokerChain(invoker, Constants.SERVICE_FILTER
_KEY, Constants.PROVIDER));
  8: }
```

- 第 2 至 5 行：当 `invoker.url.protocol = registry` ，跳过，本地暴露服务不会符合这个判断。在远程暴露服务会符合暴露该判断，所以下一篇文章分享。
- 第 7 行：调用 `#buildInvokerChain(invoker, key, group)` 方法，创建带有 Filter 过滤链的 Invoker 对象。
  - 🙂 在 「3.2.2 buildInvokerChain」 详细解析。
- 第 7 行：调用 `protocol#export(invoker)` 方法，继续暴露服务。

## 3.2.2 buildInvokerChain

`#buildInvokerChain(invoker, key, group)` 方法，创建带 Filter 链的 Invoker 对象。代码如下：

```
 1: /**
 2:  * 创建带 Filter 链的 Invoker 对象
 3:  *
 4:  * @param invoker Invoker 对象
 5:  * @param key 获取 URL 参数名
 6:  * @param group 分组
 7:  * @param <T> 泛型
 8:  * @return Invoker 对象
 9:  */
10: private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, Stri
ng key, String group) {
11:     Invoker<T> last = invoker;
12:     // 获得过滤器数组
13:     List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class).ge
tActivateExtension(invoker.getUrl(), key, group);
14:     // 倒序循环 Filter ，创建带 Filter 链的 Invoker 对象
15:     if (!filters.isEmpty()) {
16:         for (int i = filters.size() - 1; i >= 0; i--) {
17:             final Filter filter = filters.get(i);
18:             final Invoker<T> next = last;
19:             last = new Invoker<T>() {
20:
21:                 public Class<T> getInterface() {
22:                     return invoker.getInterface();
23:                 }
24:
25:                 public URL getUrl() {
26:                     return invoker.getUrl();
27:                 }
28:
29:                 public boolean isAvailable() {
30:                     return invoker.isAvailable();
31:                 }
32:
```

```
 33:                 public Result invoke(Invocation invocation) throws RpcExceptio
n {
 34:                     return filter.invoke(next, invocation);
 35:                 }
 36:
 37:                 public void destroy() {
 38:                     invoker.destroy();
 39:                 }
 40:
 41:                 @Override
 42:                 public String toString() {
 43:                     return invoker.toString();
 44:                 }
 45:             };
 46:         }
 47:     }
 48:     return last;
 49: }
```

- `key` 属性，获得 URL 参数名。
  - 该参数用于获得 ServiceConfig 或 ReferenceConfig 配置的自定义过滤器。
  - 以 ServiceConfig 举例子，例如 `url = injvm://127.0.0.1/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&bind.ip=192.168.3.17&bind.port=20880&default.delay=-1&default.retries=0&default.service.filter=demo&delay=-1&dubbo=2.0.0&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=81844&qos.port=22222&service.filter=demo&side=provider&timestamp=1520682156043` 中， `service.filter=demo` ，这是笔者配置自定义的 DemoFilter 过滤器。

    ```
    <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoServ
    ice" filter="demo" />
    ```

    - x
- `group` 属性，分组。
  - 在暴露服务时， `group = provider` 。
  - 在引用服务时， `group = consumer` 。
- ========== 分割线 ==========

- 第 13 行：调用 `ExtensionLoader#getActivateExtension(url, key, group)` 方法，获得过滤器数组。
    - 🙂 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，`filters` 为：
        - EchoFilter
        - ClassLoaderFilter
        - GenericFilter
        - ContextFilter
        - TraceFilter
        - TimeoutFilter
        - MonitorFilter
        - ExceptionFilter
        - DemoFilter 【自定义】
- 第 15 至 47 行：**倒序**循环 Filter ，创建带 Filter 链的 Invoker 对象。因为是通过**嵌套**声明匿名类循环调用的方式，所以要倒序。胖友可以手工模拟下这个过程。通过这样的方式，实际过滤的顺序，还是我们上面看到的**正序**。

# 3.3 ProtocolListenerWrapper

`com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper` ，实现 Protocol 接口，Protocol 的 Wrapper 拓展实现类，用于给 Exporter 增加 ExporterListener ，监听 Exporter 暴露完成和取消暴露完成。

## 3.3.1 export

本文涉及的 `#export(invoker)` 方法，代码如下：

```
  1: public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
  2:     // 注册中心
  3:     if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
  4:         return protocol.export(invoker);
  5:     }
  6:     // 暴露服务，创建 Exporter 对象
  7:     Exporter<T> exporter = protocol.export(invoker);
  8:     // 获得 ExporterListener 数组
  9:     List<ExporterListener> listeners = Collections.unmodifiableList(ExtensionL
oader.getExtensionLoader(ExporterListener.class).getActivateExtension(invoker.getUr
l(), Constants.EXPORTER_LISTENER_KEY));
 10:     // 创建带 ExporterListener 的 Exporter 对象
```

```
11:     return new ListenerExporterWrapper<T>(exporter, listeners);
12: }
```

- 第 2 至 5 行：当 `invoker.url.protocol = registry` ，跳过，本地暴露服务不会符合这个判断。在远程暴露服务会符合暴露该判断，所以下一篇文章分享。
- 第 7 行：调用 `InjvmProtocol#export(invoker)` 方法，暴露本地服务，创建 InjvmExporter 对象。
  - 🙂 在 「4.2 InjvmExporter」 详细解析。
- 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 ExporterListener 的 ListenerExporterWrapper 对象。在这个过程中，会执行 `ExporterListener#exported(exporter)` 方法。
  - 🙂 在 「4.3 ListenerExporterWrapper」 详细解析。

# 3.4 AbstractProtocol

`com.alibaba.dubbo.rpc.protocol.AbstractProtocol` ，实现 Protocol 接口，协议抽象类。代码如下：

```
public abstract class AbstractProtocol implements Protocol {

    /**
     * Exporter 集合
     *
     * key: 服务键 {@link #serviceKey(URL)} 或 {@link URL#getServiceKey()} 。
     *      不同协议会不同
     */
    protected final Map<String, Exporter<?>> exporterMap = new ConcurrentHashMap<St
ring, Exporter<?>>();

    // ... 省略和本文无关的方法与属性

}
```

- `exporterMap` 属性，Exporter 集合。**该集合拥有该协议中，所有暴露中的 Exporter 对象**。其中 `key` 为服务键。不同协议的实现，生成的方式**略有差距**。例如：
  - InjvmProtocol 使用 `URL#getServiceKey()` 方法

    - DubboProtocol 使用 `#serviceKey(URL)` 方法。
    - 差别主要在于是否包含 `port` 。实际上，也是**一致的**。因为 InjvmProtocol 统一 `port=0` 。
- 该类中，**省略和本文无关的方法与属性**。

# 3.5 InjvmProtocol

`com.alibaba.dubbo.rpc.protocol.injvm.InjvmProtocol` ，实现 AbstractProtocol 抽象类，Injvm 协议实现类。

## 3.5.1 属性

属性相关，代码如下：

```
/**
 * 协议名
 */
public static final String NAME = Constants.LOCAL_PROTOCOL;
/**
 * 默认端口
 */
public static final int DEFAULT_PORT = 0;
/**
 * 单例。在 Dubbo SPI 中，被初始化，有且仅有一次。
 */
private static InjvmProtocol INSTANCE;

public InjvmProtocol() {
    INSTANCE = this;
}

public static InjvmProtocol getInjvmProtocol() {
    if (INSTANCE == null) {
        ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(InjvmProtoc
ol.NAME); // load
    }
    return INSTANCE;
}
```

- `NAME` **静态**属性， `injvm` ，协议名。
- `DEFAULT_PORT` **静态**属性，默认端口为 0 。
- `INSTANCE` **静态**属性，单例。通过 Dubbo SPI 加载创建，有且仅有一次。

- `#getInjvmProtocol()` **静态**方法，获得单例。

### 3.5.2 export

本文涉及的 `#export(invoker)` 方法，代码如下：

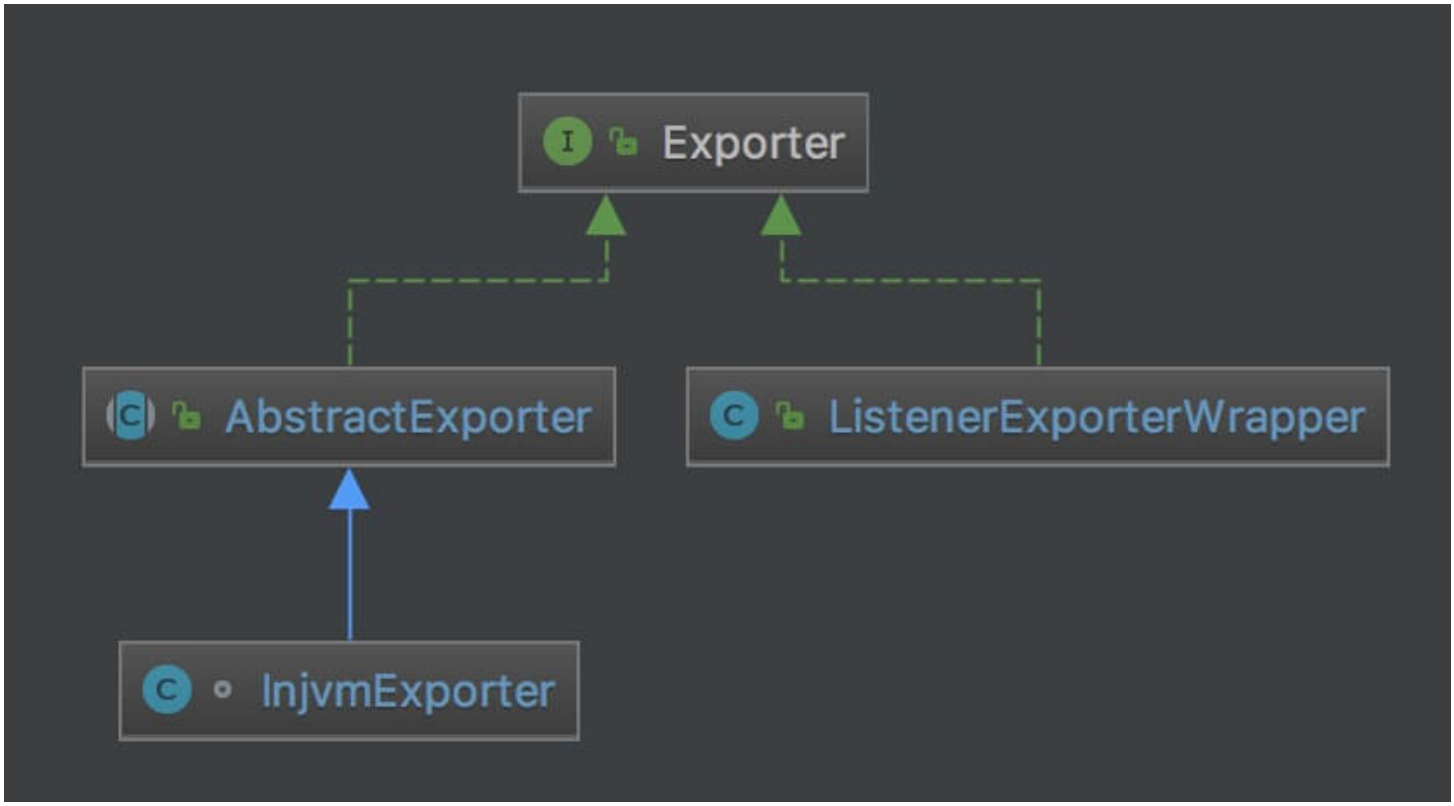
```
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    return new InjvmExporter<T>(invoker, invoker.getUrl().getServiceKey(), exporterMap);
}
```

- 创建 InjvmExporter 对象。

# 4. Exporter

Exporter **接口**，在 《精尽 Dubbo 源码分析 —— 核心流程一览》「4.7 Exporter」 有详细解析。

本文涉及的 Exporter 类图如下：




## 4.1 AbstractExporter

`com.alibaba.dubbo.rpc.protocol.AbstractExporter`，实现 Exporter 接口，Exporter 抽象类。代码如下：

```
public abstract class AbstractExporter<T> implements Exporter<T> {

    protected final Logger logger = LoggerFactory.getLogger(getClass());

    /**
     * Invoker 对象
     */
    private final Invoker<T> invoker;
    /**
     * 是否取消暴露服务
     */
    private volatile boolean unexported = false;

    public AbstractExporter(Invoker<T> invoker) {
        if (invoker == null)
            throw new IllegalStateException("service invoker == null");
        if (invoker.getInterface() == null)
            throw new IllegalStateException("service type == null");
        if (invoker.getUrl() == null)
            throw new IllegalStateException("service url == null");
        this.invoker = invoker;
    }

    @Override
    public Invoker<T> getInvoker() {
        return invoker;
    }

    @Override
    public void unexport() {
        if (unexported) {
            return;
        }
        unexported = true;
        getInvoker().destroy();
    }

    public String toString() {
        return getInvoker().toString();
    }

}
```

- `invoker` 属性，Invoker 对象。
  - `#getInvoker()` 方法，获得 Invoker 对象。
- `unexported` 属性，是否取消暴露服务。
  - `#unexport()` 方法，取消暴露服务。

## 4.2 InjvmExporter

`com.alibaba.dubbo.rpc.protocol.injvm.InjvmProtocol` ，实现 AbstractExporter 抽象类，Injvm Exporter 实现类。代码如下：

```
class InjvmExporter<T> extends AbstractExporter<T> {

    /**
     * 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;

    InjvmExporter(Invoker<T> invoker, String key, Map<String, Exporter<?>> exporterMap) {
        super(invoker);
        this.key = key;
        this.exporterMap = exporterMap;
        // 添加到 Exporter 集合
        exporterMap.put(key, this);
    }

    @Override
    public void unexport() {
        super.unexport();
        // 移除出 Exporter 集合
        exporterMap.remove(key);
    }

}
```

- `key` 属性，服务键。
- `exporterMap` 属性，Exporter 集合。在上文 `InjvmProtocol#export(invoker)` 方法中，我们可以看到，该属性就是 `AbstractProtocol.exporterMap` 属性。
  - **构造方法，发起**暴露，将自己添加到 `exporterMap` 中。
  - `#unexport()` 方法，**取消**暴露，将自己移除出 `exporterMap` 中。

## 4.3 ListenerExporterWrapper

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

```
public class ListenerExporterWrapper<T> implements Exporter<T> {

    private static final Logger logger = LoggerFactory.getLogger(ListenerExporterWr
apper.class);

    /**
     * 真实的 Exporter 对象
     */
    private final Exporter<T> exporter;
    /**
     * Exporter 监听器数组
     */
    private final List<ExporterListener> listeners;

    public ListenerExporterWrapper(Exporter<T> exporter, List<ExporterListener> lis
teners) {
        if (exporter == null) {
            throw new IllegalArgumentException("exporter == null");
        }
        this.exporter = exporter;
        this.listeners = listeners;
        // 执行监听器
        if (listeners != null && !listeners.isEmpty()) {
            RuntimeException exception = null;
            for (ExporterListener listener : listeners) {
                if (listener != null) {
                    try {
                        listener.exported(this);
                    } catch (RuntimeException t) {
                        logger.error(t.getMessage(), t);
                        exception = t;
                    }
                }
```

```
            }
            if (exception != null) {
                throw exception;
            }
        }
    }

    public Invoker<T> getInvoker() {
        return exporter.getInvoker();
    }

    public void unexport() {
        try {
            exporter.unexport();
        } finally {
            // 执行监听器
            if (listeners != null && !listeners.isEmpty()) {
                RuntimeException exception = null;
                for (ExporterListener listener : listeners) {
                    if (listener != null) {
                        try {
                            listener.unexported(this);
                        } catch (RuntimeException t) {
                            logger.error(t.getMessage(), t);
                            exception = t;
                        }
                    }
                }
                if (exception != null) {
                    throw exception;
                }
            }
        }
    }

}
```

- **构造方法**，循环 `listeners` ，执行 `ExporterListener#exported(listener)` 。若执行过程中发生异常 RuntimeException ，打印错误日志，继续执行，最终才抛出。
- `#unexport()` 方法，循环 `listeners` ，执行 `ExporterListener#unexported(listener)` 。若执行过程中发生异常 RuntimeException ，打印错误日志，继续执行，最终才抛出。

# 5. ExporterListener

`com.alibaba.dubbo.rpc.ExporterListener` ， Exporter 监听器。

代码如下：

```
@SPI
public interface ExporterListener {

    /**
     * The exporter exported.
     *
     * 当服务暴露完成
     *
     * @param exporter
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Protocol#export(Invoker)
     */
    void exported(Exporter<?> exporter) throws RpcException;

    /**
     * The exporter unexported.
     *
     * 当服务取消暴露完成
     *
     * @param exporter
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Exporter#unexport()
     */
    void unexported(Exporter<?> exporter);

}
```


## 5.1 ExporterListenerAdapter

`com.alibaba.dubbo.rpc.listener.ExporterListenerAdapter` ，实现 ExporterListener 接口，ExporterListener 适配器抽象类。代码如下：

```
public abstract class ExporterListenerAdapter implements ExporterListener {

    public void exported(Exporter<?> exporter) throws RpcException { }

    public void unexported(Exporter<?> exporter) throws RpcException { }

}
```

# 666. 彩蛋

周末偷懒了下，到了接近 24 点才写完。

明天继续加油，一个“暴露”的周末。