03-Dubbo特性详解-多协议

2 多协议

2.0 学习目标

1 了解dubbo支持哪些协议，各协议的适用场景、性能

2 掌握dubbo、rmi、hessian协议的使用

3 掌握多协议使用用法

4 掌握协议的配置项

5 了解协议的扩展方式

6 掌握协议的源码构成、协作调用关系（工作原理）

2.1 支持协议详解

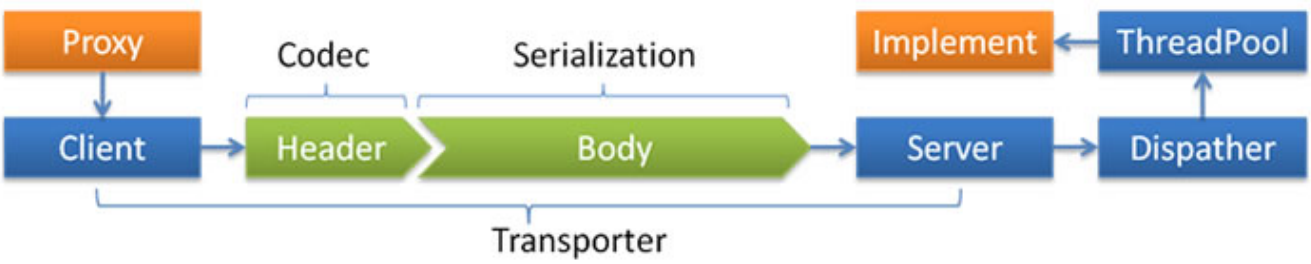
学习要点：

掌握各协议的适用场景、特性、适用约束。

dubbo://【掌握】

Dubbo 缺省协议采用单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。

反之，Dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。



- Transporter: mina, netty, grizzy
- Serialization: dubbo, hessian2, java, json
- Dispatcher: all, direct, message, execution, connection
- ThreadPool: fixed, cached

特性

缺省协议，使用基于 mina `1.1.7` 和 hessian `3.2.1` 的 tbremoting 交互。

- 连接个数：单连接
- 连接方式：长连接
- 传输协议：TCP
- 传输方式：NIO 异步传输
- 序列化：Hessian 二进制序列化
- 适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
- 适用场景：常规远程服务方法调用

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数及返回值不能自定义实现 `List`, `Map`, `Number`, `Date`, `Calendar` 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。
- Hessian 序列化，只传成员属性值和值的类型，不传方法或静态变量，兼容情况 [1][2]：

数据通讯	情况	结果
A->B	类A多一种 属性（或者说类B少一种 属性）	不抛异常，A多的那 个属性的值，B没有， 其他正常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A使用多 出来的枚举进行传输	抛异常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A不使用 多出来的枚举进行传输	不抛异常，B正常接 收数据
A->B	A和B的属性 名相同，但类型不相同	抛异常
A->B	serialId 不相同	正常传输

接口增加方法，对客户端无影响，如果该方法不是客户端需要的，客户端不需要重新部署。输入参数和结果集中增加属性，对客户端无影响，如果客户端并不需要新属性，不用重新部署。

输入参数和结果集属性名变化，对客户端序列化无影响，但是如果客户端不重新部署，不管输入还是输出，属性名变化的属性值是获取不到的。

总结：服务器端和客户端对领域对象并不需要完全一致，而是按照最大匹配原则。

配置

配置协议：

```
<dubbo:protocol name="dubbo" port="20880" />
```

设置默认协议：

```
<dubbo:provider protocol="dubbo" />
```

设置服务协议：

```
<dubbo:service protocol="dubbo" />
```

多端口：

```
<dubbo:protocol id="dubbo1" name="dubbo" port="20880" />
<dubbo:protocol id="dubbo2" name="dubbo" port="20881" />
```

配置协议选项：

```
<dubbo:protocol name="dubbo" port="9090" server="netty" client="netty"
codec="dubbo" serialization="hessian2" charset="UTF-8" threadpool="fixed"
threads="100" queues="0" iothreads="9" buffer="8192" accepts="1000"
payload="8388608" />
```

多连接配置：

Dubbo 协议缺省每服务每提供者每消费者使用单一长连接，如果数据量较大，可以使用多个连接。

```
<dubbo:service connections="1"/>
<dubbo:reference connections="1"/>
```

- `<dubbo:service connections="0">` 或 `<dubbo:reference connections="0">` 表示该服务使用 JVM 共享长连接。**缺省**
- `<dubbo:service connections="1">` 或 `<dubbo:reference connections="1">` 表示该服务使用独立长连接。
- `<dubbo:service connections="2">` 或`<dubbo:reference connections="2">` 表示该服务使用独立两条长连接。

为防止被大量连接撑挂，可在服务提供方限制大接收连接数，以实现服务提供方自我保护。

```
<dubbo:protocol name="dubbo" accepts="1000" />
```

`dubbo.properties` 配置：

```
dubbo.service.protocol=dubbo
```

常见问题

为什么要消费者比提供者个数多?

因 dubbo 协议采用单一长连接，假设网络为千兆网卡 [3]，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

为什么不能传大包?

因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡 [3:1]，每条连接最大 7MByte(不同的环境可能不一样，供参考)，单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

为什么采用异步单一长连接?

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步 IO，复用线程池，防止 C10K 问题。

在Dubbo协议中使用高效的Java序列化（Kryo和FST）

启用Kryo和FST

使用Kryo和FST非常简单，只需要在dubbo RPC的XML配置中添加一个属性即可：

```
<dubbo:protocol name="dubbo" serialization="kryo"/>
<dubbo:protocol name="dubbo" serialization="fst"/>
```

注册被序列化类

要让Kryo和FST完全发挥出高性能，最好将那些需要被序列化的类注册到dubbo系统中，例如，我们可以实现如下回调接口：

```
public class SerializationOptimizerImpl implements SerializationOptimizer
{

    public Collection<Class> getSerializableClasses() {
        List<Class> classes = new LinkedList<Class>();
        classes.add(BidRequest.class);
        classes.add(BidResponse.class);
        classes.add(Device.class);
        classes.add(Geo.class);
        classes.add(Impression.class);
        classes.add(SeatBid.class);
        return classes;
    }
}
```

然后在XML配置中添加：

```
<dubbo:protocol name="dubbo" serialization="kryo"
optimizer="org.apache.dubbo.demo.SerializationOptimizerImpl"/>
```

在注册这些类后，序列化的性能可能被大大提升，特别针对小数量的嵌套对象的时候。

当然，在对一个类做序列化的时候，可能还级联引用到很多类，比如Java集合类。针对这种情况，我们已经自动将JDK中的常用类进行了注册，所以你不需要重复注册它们（当然你重复注册了也没有任何影响），包括：

```
GregorianCalendar
InvocationHandler
BigDecimal
BigInteger
Pattern
BitSet
URI
UUID
HashMap
ArrayList
LinkedList
HashSet
TreeSet
Hashtable
Date
```

```
Calendar
ConcurrentHashMap
SimpleDateFormat
Vector
BitSet
StringBuffer
StringBuilder
Object
Object[]
String[]
byte[]
char[]
int[]
float[]
double[]
```

由于注册被序列化的类仅仅是出于性能优化的目的，所以即使你忘记注册某些类也没有关系。事实上，即使不注册任何类，Kryo和FST的性能依然普遍优于hessian和dubbo序列化。

rmi://

RMI 协议采用 JDK 标准的 `java.rmi.*` 实现，采用阻塞式短连接和 JDK 标准序列化方式。

注意：如果正在使用 RMI 提供服务给外部访问，同时应用里依赖了老的 common-collections 包的情况下，存在反序列化安全风险。（dubbo自身不依赖此包，请检查应用：将 commons-collections3 请升级到 3.2.2；将 commons-collections4 请升级到 4.1。新版本的 commons-collections 解决了该问题）

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：TCP
- 传输方式：同步传输
- 序列化：Java 标准二进制序列化
- 适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。
- 适用场景：常规远程服务方法调用，与原生RMI服务互操作

约束

- 参数及返回值需实现 `Serializable` 接口

- dubbo 配置中的超时时间对 RMI 无效，需使用 java 启动参数设置：`-Dsun.rmi.transport.tcp.responseTimeout=3000`，参见下面的 RMI 配置

dubbo.properties 配置

```
dubbo.service.protocol=rmi
```

RMI配置

```
java -Dsun.rmi.transport.tcp.responseTimeout=3000
```

更多 RMI 优化参数请查看 JDK 文档

接口

如果服务接口继承了 `java.rmi.Remote` 接口，可以和原生 RMI 互操作，即：

- 提供者用 Dubbo 的 RMI 协议暴露服务，消费者直接用标准 RMI 接口调用，
- 或者提供方用标准 RMI 暴露服务，消费方用 Dubbo 的 RMI 协议调用。

如果服务接口没有继承 `java.rmi.Remote` 接口：

- 缺省 Dubbo 将自动生成一个 `com.xxx.XxxService$Remote` 的接口，并继承 `java.rmi.Remote` 接口，并以此接口暴露服务，
- 但如果设置了 `<dubbo:protocol name="rmi" codec="spring" />`，将不生成 `$Remote` 接口，而使用 Spring 的 `RmiInvocationHandler` 接口暴露服务，和 Spring 兼容。

配置

定义 RMI 协议：

```
<dubbo:protocol name="rmi" port="1099" />
```

设置默认协议：

```
<dubbo:provider protocol="rmi" />
```

设置服务协议：

```
<dubbo:service protocol="rmi" />
```

多端口：

```
<dubbo:protocol id="rmi1" name="rmi" port="1099" />
<dubbo:protocol id="rmi2" name="rmi" port="2099" />

<dubbo:service protocol="rmi1" />
```

Spring 兼容性：

```
<dubbo:protocol name="rmi" codec="spring" />
```

hessian://

Hessian [1] 协议用于集成 Hessian 的服务，Hessian 底层采用 Http 通讯，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现。

Dubbo 的 Hessian 协议可以和原生 Hessian 服务互操作，即：

- 提供者用 Dubbo 的 Hessian 协议暴露服务，消费者直接用标准 Hessian 接口调用
- 或者提供方用标准 Hessian 暴露服务，消费方用 Dubbo 的 Hessian 协议调用。

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：Hessian二进制序列化
- 适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。
- 适用场景：页面传输，文件传输，或与原生hessian服务互操作

依赖

```
<dependency>
    <groupId>com.caucho</groupId>
    <artifactId>hessian</artifactId>
    <version>4.0.7</version>
</dependency>
```

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数及返回值不能自定义实现 `List`, `Map`, `Number`, `Date`, `Calendar` 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。

配置

定义 hessian 协议：

```
<dubbo:protocol name="hessian" port="8080" server="jetty" />
```

设置默认协议：

```
<dubbo:provider protocol="hessian" />
```

设置 service 协议：

```
<dubbo:service protocol="hessian" />
```

多端口：

```
<dubbo:protocol id="hessian1" name="hessian" port="8080" />
<dubbo:protocol id="hessian2" name="hessian" port="8081" />
```

直连：

```
<dubbo:reference id="helloService" interface="HelloWorld"
url="hessian://10.20.153.10:8080/helloWorld" />
```

---

1. Hessian 是 Caucho 开源的一个 RPC 框架，其通讯效率高于 WebService 和 Java 自带的序列化。 ↩

应用特别说明

Jetty

Dubbo长期未更新，hessian协议方式时，如果使用Jetty作为服务器，推荐的依赖版本jetty:6.1.26 在maven仓库上已不存在。引入较新的jetty版本，会报如下异常：

```
java.lang.ClassNotFoundException: org.mortbay.log.Logger
```

而 org.mortbay.log.Logger对应的jar也已很难找到。

Tomcat

Jetty不行，我们可以使用tomcat，但dubbo-2.6.6版本支持的tomcat版本tomcat-8.5.x及以下版本。

在spring boot 应用中使用时，如果是web应用，使用内嵌的tomcat，则可能需要降tomcat版本。

服务端

1、引入如下依赖：

```
        <!-- 选择使用hessian协议时需要 -->
        <dependency>
            <groupId>com.caucho</groupId>
            <artifactId>hessian</artifactId>
            <version>4.0.60</version>
        </dependency>
        <!-- 选择使用hessian协议时,使用tomcat作服务器 -->
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
            <version>8.5.40</version>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-el</artifactId>
            <version>8.5.40</version>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-websocket</artifactId>
            <version>8.5.40</version>
        </dependency>
```

配置协议的服务器使用tomcat

```yaml
dubbo:
  protocol:
    name: hessian
    server: tomcat
```

消费端

只需引入

```xml
        <!-- 选择使用hessian协议时需要 -->
        <dependency>
            <groupId>com.caucho</groupId>
            <artifactId>hessian</artifactId>
            <version>4.0.60</version>
        </dependency>
```

http://

基于 HTTP 表单的远程调用协议，采用 Spring 的 HttpInvoker 实现（`2.3.0` 以上版本支持）。

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：表单序列化
- 适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或URL传入参数，暂不支持传文件。
- 适用场景：需同时给应用程序和浏览器 JS 使用的服务。

约束

- 参数及返回值需符合 Bean 规范

配置

配置协议：

```
<dubbo:protocol name="http" port="8080" />
```

配置 Jetty Server (默认)：

```
<dubbo:protocol ... server="jetty" />
```

配置 Servlet Bridge Server (推荐使用)：

```
<dubbo:protocol ... server="servlet" />
```

配置 DispatcherServlet：

```
<servlet>
        <servlet-name>dubbo</servlet-name>
        <servlet-
class>org.apache.dubbo.remoting.http.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dubbo</servlet-name>
        <url-pattern>/*</url-pattern>
</servlet-mapping>
```

注意，如果使用 servlet 派发请求：

- 协议的端口 `<dubbo:protocol port="8080" />` 必须与 servlet 容器的端口相同，
- 协议的上下文路径 `<dubbo:protocol contextpath="foo" />` 必须与 servlet 应用的上下文路径相同。

应用说明

两种使用方式：

方式一：开启独立的http服务

服务端配置

```yaml
dubbo:
  application:
    name: service-app1
  registry:
    address: zookeeper://127.0.0.1:2181
  protocol:
    name: http
    server: tomcat
    port: 20880
```

默认使用Jetty服务，2.6.6版本下使用jetty不便（上面有讲到）。我们可以使用tomcat。

消费端：不需特殊配置

方式二：配置 Servlet Bridge Server，与应用本身的web应用共用http服务

服务端配置

1、配置dubbo协议配置：

```yaml
dubbo:
  application:
    name: service-app1
  registry:
    address: zookeeper://127.0.0.1:2181
  protocol:
    name: http
    server: servlet
    port: 8080
```

注意：是共用http服务，所以端口、应用上下文（如配置则需要和web应用一样）。

2、配置桥接Servlet

```
    @Bean
    public ServletRegistrationBean<DispatcherServlet>
getServletRegistrationBean() {

        ServletRegistrationBean<DispatcherServlet> bean = new
ServletRegistrationBean<DispatcherServlet>(
                new DispatcherServlet());
        bean.addUrlMappings("/*");
        bean.setLoadOnStartup(1);
        return bean;
    }
```

消费端：不需特殊配置

webservice://

基于 WebService 的远程调用协议，基于 Apache CXF [1] 的 `frontend-simple` 和 `transports-http` 实现 [2]。

可以和原生 WebService 服务互操作，即：

- 提供者用 Dubbo 的 WebService 协议暴露服务，消费者直接用标准 WebService 接口调用，
- 或者提供方用标准 WebService 暴露服务，消费方用 Dubbo 的 WebService 协议调用。

依赖

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-simple</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
    <version>2.6.1</version>
</dependency>
```

特性

- 连接个数：多连接

- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：SOAP 文本序列化
- 适用场景：系统集成，跨语言调用

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数尽量使用基本类型和 POJO

配置

配置协议：

```
<dubbo:protocol name="webservice" port="8080" server="jetty" />
```

配置默认协议：

```
<dubbo:provider protocol="webservice" />
```

配置服务协议：

```
<dubbo:service protocol="webservice" />
```

多端口：

```
<dubbo:protocol id="webservice1" name="webservice" port="8080" />
<dubbo:protocol id="webservice2" name="webservice" port="8081" />
```

直连：

```
<dubbo:reference id="helloService" interface="HelloWorld"
url="webservice://10.20.153.10:8080/com.foo.HelloWorld" />
```

WSDL：

```
http://10.20.153.10:8080/com.foo.HelloWorld?wsdl
```

Jetty Server (默认)：

```
<dubbo:protocol ... server="jetty" />
```

Servlet Bridge Server (推荐)：

```
<dubbo:protocol ... server="servlet" />
```

配置 DispatcherServlet：

```
<servlet>
        <servlet-name>dubbo</servlet-name>
        <servlet-
class>org.apache.dubbo.remoting.http.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dubbo</servlet-name>
        <url-pattern>/*</url-pattern>
</servlet-mapping>
```

注意，如果使用 servlet 派发请求：

- 协议的端口 `<dubbo:protocol port="8080" />` 必须与 servlet 容器的端口相同，
- 协议的上下文路径 `<dubbo:protocol contextpath="foo" />` 必须与 servlet 应用的上下文路径相同。

thrift://

当前 dubbo 支持 [1]的 thrift 协议是对 thrift 原生协议 [2] 的扩展，在原生协议的基础上添加了一些额外的头信息，比如 service name，magic number 等。

使用 dubbo thrift 协议同样需要使用 thrift 的 idl compiler 编译生成相应的 java 代码，后续版本中会在这方面做一些增强。

依赖

```
<dependency>
    <groupId>org.apache.thrift</groupId>
    <artifactId>libthrift</artifactId>
    <version>0.8.0</version>
</dependency>
```

配置

所有服务共用一个端口 [3]：

```
<dubbo:protocol name="thrift" port="3030" />
```

使用

可以参考 dubbo 项目中的示例代码

常见问题

- Thrift 不支持 null 值，即：不能在协议中传递 null 值

---

1. `2.3.0` 以上版本支持 ↩
2. Thrift 是 Facebook 捐给 Apache 的一个 RPC 框架 ↩
3. 与原生Thrift不兼容 ↩

memcached:// 【了解有这么回事】

【并未提供产品级使用实现】

基于 memcached [1] 实现的 RPC 协议 [2]。

注册 memcached 服务的地址

```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExten
sion();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("memcached://10.20.153.11/com.foo.BarService
?
category=providers&dynamic=false&application=foo&group=member&loadbalance=
consistenthash"));
```

在客户端引用

在客户端使用 [3]：

```
<dubbo:reference id="cache" interface="java.util.Map" group="member" />
```

或者，点对点直连：

```
<dubbo:reference id="cache" interface="java.util.Map"
url="memcached://10.20.153.10:11211" />
```

也可以使用自定义接口：

```
<dubbo:reference id="cache" interface="com.foo.CacheService"
url="memcached://10.20.153.10:11211" />
```

方法名建议和 memcached 的标准方法名相同，即：get(key), set(key, value), delete(key)。

如果方法名和 memcached 的标准方法名不相同，则需要配置映射关系 [4]：

```
<dubbo:reference id="cache" interface="com.foo.CacheService"
url="memcached://10.20.153.10:11211" p:set="putFoo" p:get="getFoo"
p:delete="removeFoo" />
```

---

1. Memcached 是一个高效的 KV 缓存服务器 ↩
2. `2.3.0` 以上版本支持 ↩
3. 不需要感知 Memcached 的地址 ↩
4. 其中 "p:xxx" 为 spring 的标准 p 标签 ↩

redis://【了解有这么回事】

【并未提供产品级使用实现】

基于 Redis [1] 实现的 RPC 协议 [2]。

注册 redis 服务的地址

```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExten
sion();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("redis://10.20.153.11/com.foo.BarService?
category=providers&dynamic=false&application=foo&group=member&loadbalance=
consistenthash"));
```

在客户端引用

在客户端使用 [3]：

```
<dubbo:reference id="store" interface="java.util.Map" group="member" />
```

或者，点对点直连：

```
<dubbo:reference id="store" interface="java.util.Map"
url="redis://10.20.153.10:6379" />
```

也可以使用自定义接口：

```
<dubbo:reference id="store" interface="com.foo.StoreService"
url="redis://10.20.153.10:6379" />
```

方法名建议和 redis 的标准方法名相同，即：get(key), set(key, value), delet(key)。

如果方法名和 redis 的标准方法名不相同，则需要配置映射关系 [4]：

```
<dubbo:reference id="cache" interface="com.foo.CacheService"
url="redis://10.20.153.10:6379" p:set="putFoo" p:get="getFoo"
p:delete="removeFoo" />
```

---

1. Redis 是一个高效的 KV 存储服务器 ↩
2. `2.3.0` 以上版本支持 ↩
3. 不需要感知 Redis 的地址 ↩
4. 其中 "p:xxx" 为 spring 的标准 p 标签 ↩

rest://

http://dubbo.apache.org/zh-cn/docs/user/references/protocol/rest.html

2.2 各协议性能测试报告【了解】

http://dubbo.apache.org/zh-cn/docs/user/perf-test.html

2.3 多协议使用

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

2.1.1 不同服务不同协议

不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://dubbo.apache.org/schema/dubbo
http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <dubbo:registry id="registry" address="10.20.141.150:9090"
username="admin" password="hello1234" />
    <!-- 多协议配置 -->
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:protocol name="rmi" port="1099" />
    <!-- 使用dubbo协议暴露服务 -->
    <dubbo:service interface="com.alibaba.hello.api.HelloService"
version="1.0.0" ref="helloService" protocol="dubbo" />
    <!-- 使用rmi协议暴露服务 -->
    <dubbo:service interface="com.alibaba.hello.api.DemoService"
version="1.0.0" ref="demoService" protocol="rmi" />
</beans>
```

2.1.2 多协议暴露服务

需要与 http 客户端互操作

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://dubbo.apache.org/schema/dubbo
http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <dubbo:registry id="registry" address="10.20.141.150:9090"
username="admin" password="hello1234" />
    <!-- 多协议配置 -->
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:protocol name="hessian" port="8080" />
    <!-- 使用多个协议暴露服务 -->
    <dubbo:service id="helloService"
interface="com.alibaba.hello.api.HelloService" version="1.0.0"
protocol="dubbo,hessian" />
</beans>
```

2.1.3 可配置属性 dubbo:protocol

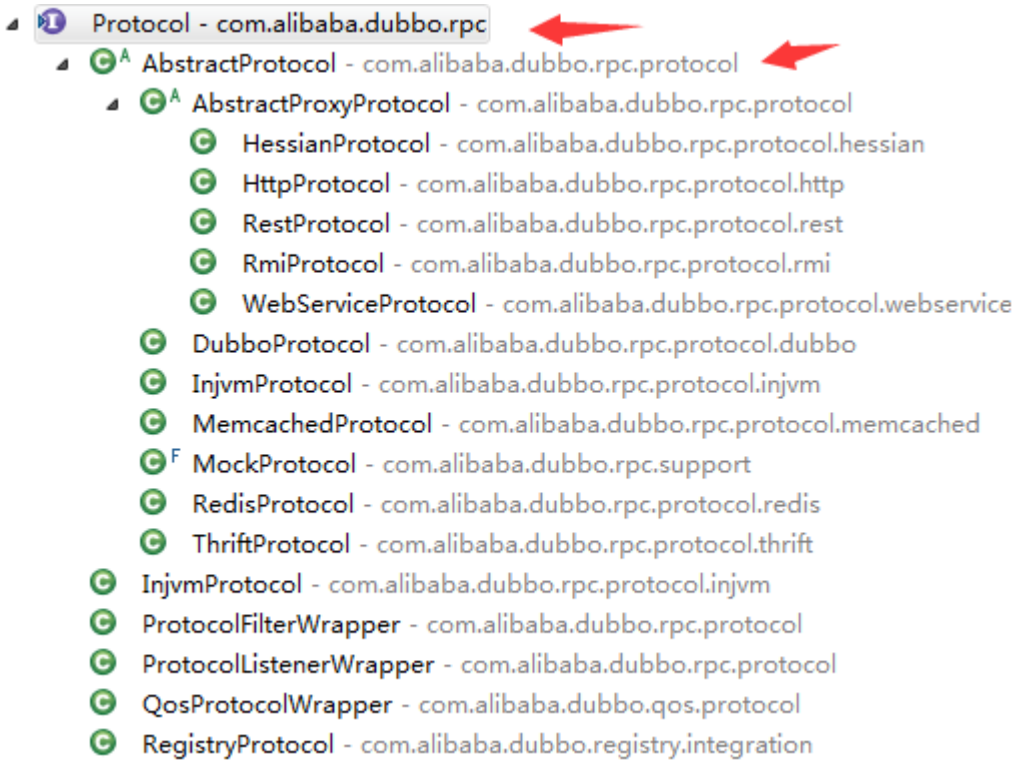
服务提供者协议配置。对应的配置类： `org.apache.dubbo.config.ProtocolConfig`。同时，如果需要支持多协议，可以声明多个 `<dubbo:protocol>` 标签，并在 `<dubbo:service>` 中通过 `protocol` 属性指定使用的协议。

属性	对应URL参数	类型	是否必填	缺省值	作用	描述	兼容性
id		string	可选	dubbo	配置关联	协议BeanId，可以在<dubbo:service protocol="">中引用此ID，如果ID不填，缺省和name属性值一样，重复则在name后加序号。	2.0.5以上版本
name		string	**必填**	dubbo	性能调优	协议名称	2.0.5以上版本
port		int	可选	dubbo协议缺省端口为20880，rmi协议缺省端口为1099，http和hessian协议缺省端口为80；如果**没有**配置port，则自动采用默认端口，如果配置为**-1**，则会分配一个没有被占用的端口。Dubbo 2.4.0+，分配的端口在协议缺省端口的基础上增长，确保端口段可控。	服务发现	服务端口	2.0.5以上版本
host		string	可选	自动查找本机IP	服务发现	-服务主机名，多网卡选择或指定VIP及域名时使用，为空则自动查找本机IP，-建议不要配置，让Dubbo自动获取本机IP	2.0.5以上版本
threadpool	threadpool	string	可选	fixed	性能调优	线程池类型，可选：fixed/cached	2.0.5以上版本
threads	threads	int	可选	200	性能调优	服务线程池大小(固定大小)	2.0.5以上版本
iothreads	threads	int	可选	cpu个数+1	性能调优	io线程池大小(固定大小)	2.0.5以上版本
accepts	accepts	int	可选	0	性能调优	服务提供方最大可接受连接数	2.0.5以上版本
payload	payload	int	可选	8388608(=8M)	性能调优	请求及响应数据包大小限制，单位：字节	2.0.5以上版本
codec	codec	string	可选	dubbo	性能调优	协议编码方式	2.0.5以上版本
serialization	serialization	string	可选	dubbo协议缺省为hessian2，rmi协议缺省为java，http协议缺省为json	性能调优	协议序列化方式，当协议支持多种序列化方式时使用，比如：dubbo协议的dubbo,hessian2,java,compactedjava，以及http协议的json等	2.0.5以上版本
accesslog	accesslog	string/boolean	可选		服务治理	设为true，将向logger中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件	2.0.5以上版本
path		string	可选		服务发现	提供者上下文路径，为服务path的前缀	2.0.5以上版本
transporter	transporter	string	可选	dubbo协议缺省为netty	性能调优	协议的服务端和客户端实现类型，比如：dubbo协议的mina,netty等，可以分拆为server和client配置	2.0.5以上版本
server	server	string	可选	dubbo协议缺省为netty，http协议缺省为servlet	性能调优	协议的服务器端实现类型，比如：dubbo协议的mina,netty等，http协议的jetty,servlet等	2.0.5以上版本

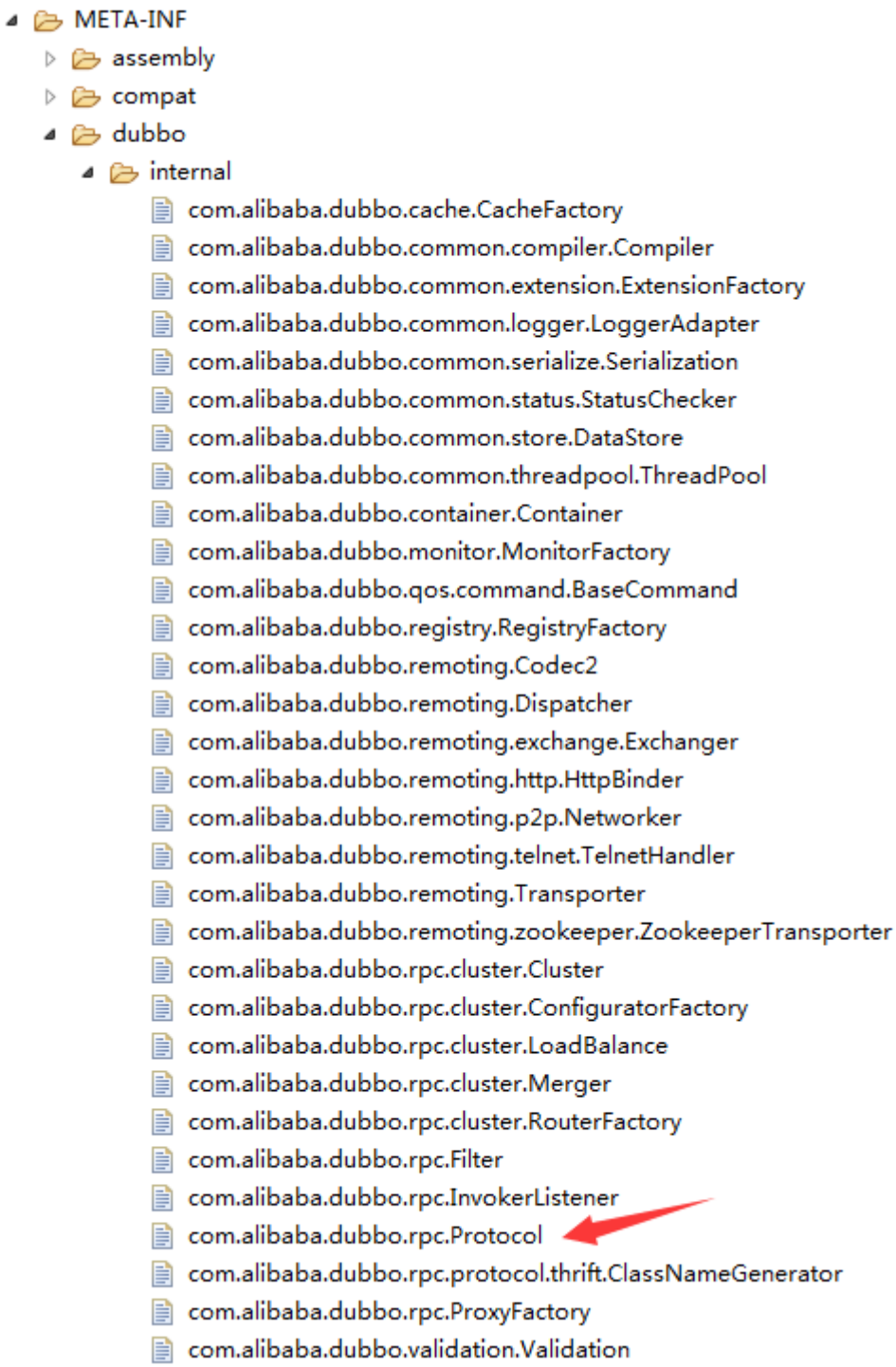
属性	对应URL参数	类型	是否必填	缺省值	作用	描述	兼容性
client	client	string	可选	dubbo协议缺省为netty	性能调优	协议的客户端实现类型，比如：dubbo协议的mina,netty等	2.0.5以上版本
dispatcher	dispatcher	string	可选	dubbo协议缺省为all	性能调优	协议的消息派发方式，用于指定线程模型，比如：dubbo协议的all, direct, message, execution, connection等	2.1.0以上版本
queues	queues	int	可选	0	性能调优	线程池队列大小，当线程池满时，排队等待执行的队列大小，建议不要设置，当线程池满时应立即失败，重试其它服务提供机器，而不是排队，除非有特殊需求。	2.0.5以上版本
charset	charset	string	可选	UTF-8	性能调优	序列化编码	2.0.5以上版本
buffer	buffer	int	可选	8192	性能调优	网络读写缓冲区大小	2.0.5以上版本
heartbeat	heartbeat	int	可选	0	性能调优	心跳间隔，对于长连接，当物理层断开时，比如拔网线，TCP的FIN消息来不及发送，对方收不到断开事件，此时需要心跳来帮助检查连接是否已断开	2.0.10以上版本
telnet	telnet	string	可选		服务治理	所支持的telnet命令，多个命令用逗号分隔	2.0.5以上版本
register	register	boolean	可选	true	服务治理	该协议的服务是否注册到注册中心	2.0.8以上版本
contextpath	contextpath	String	可选	缺省为空串	服务治理		2.0.6以上版本

2.4 协议实现源码

协议层的现有实现类如下图所示。我们如需要扩展自己的协议可参考它们。


Protocol - com.alibaba.dubbo.rpc
AbstractProtocol - com.alibaba.dubbo.rpc.protocol
AbstractProxyProtocol - com.alibaba.dubbo.rpc.protocol
HessianProtocol - com.alibaba.dubbo.rpc.protocol.hessian
HttpProtocol - com.alibaba.dubbo.rpc.protocol.http
RestProtocol - com.alibaba.dubbo.rpc.protocol.rest
RmiProtocol - com.alibaba.dubbo.rpc.protocol.rmi
WebServiceProtocol - com.alibaba.dubbo.rpc.protocol.webservice
DubboProtocol - com.alibaba.dubbo.rpc.protocol.dubbo
InjvmProtocol - com.alibaba.dubbo.rpc.protocol.injvm
MemcachedProtocol - com.alibaba.dubbo.rpc.protocol.memcached
MockProtocol - com.alibaba.dubbo.rpc.support
RedisProtocol - com.alibaba.dubbo.rpc.protocol.redis
ThriftProtocol - com.alibaba.dubbo.rpc.protocol.thrift
InjvmProtocol - com.alibaba.dubbo.rpc.protocol.injvm
ProtocolFilterWrapper - com.alibaba.dubbo.rpc.protocol
ProtocolListenerWrapper - com.alibaba.dubbo.rpc.protocol
QosProtocolWrapper - com.alibaba.dubbo.qos.protocol
RegistryProtocol - com.alibaba.dubbo.registry.integration

看协议的配置



2.5 协议扩展【了解】

扩展接口

- `org.apache.dubbo.rpc.Protocol`
- `org.apache.dubbo.rpc.Exporter`
- `org.apache.dubbo.rpc.Invoker`

```
public interface Protocol {
    /**
     * 暴露远程服务：<br>
     * 1. 协议在接收请求时，应记录请求来源方地址信息：
RpcContext.getContext().setRemoteAddress();<br>
     * 2. export()必须是幂等的，也就是暴露同一个URL的Invoker两次，和暴露一次没有区
别。<br>
     * 3. export()传入的Invoker由框架实现并传入，协议不需要关心。<br>
     *
     * @param <T> 服务的类型
     * @param invoker 服务的执行体
     * @return exporter 暴露服务的引用，用于取消暴露
     * @throws RpcException 当暴露服务出错时抛出，比如端口已占用
     */
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

    /**
     * 引用远程服务：<br>
     * 1. 当用户调用refer()所返回的Invoker对象的invoke()方法时，协议需相应执行同
URL远端export()传入的Invoker对象的invoke()方法。<br>
     * 2. refer()返回的Invoker由协议实现，协议通常需要在此Invoker中发送远程请求。
<br>
     * 3. 当url中有设置check=false时，连接失败不能抛出异常，需内部自动恢复。<br>
     *
     * @param <T> 服务的类型
     * @param type 服务的类型
     * @param url 远程服务的URL地址
     * @return invoker 服务的本地代理
     * @throws RpcException 当连接服务提供方失败时抛出
     */
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

}
```

扩展说明

RPC 协议扩展，封装远程调用细节。

契约：

- 当用户调用 `refer()` 所返回的 `Invoker` 对象的 `invoke()` 方法时，协议需相应执行同 URL 远端 `export()` 传入的 `Invoker` 对象的 `invoke()` 方法。
- 其中，`refer()` 返回的 `Invoker` 由协议实现，协议通常需要在此 `Invoker` 中发送远程请求，`export()` 传入的 `Invoker` 由框架实现并传入，协议不需要关心。

注意：

- 协议不关心业务接口的透明代理，以 `Invoker` 为中心，由外层将 `Invoker` 转换为业务接口。
- 协议不一定要是 TCP 网络通讯，比如通过共享文件，IPC 进程间通讯等。

扩展配置

```
<!-- 声明协议，如果没有配置id，将以name为id -->
<dubbo:protocol id="xxx1" name="xxx" />
<!-- 引用协议，如果没有配置protocol属性，将在ApplicationContext中自动扫描protocol配置 -->
<dubbo:service protocol="xxx1" />
<!-- 引用协议缺省值，当<dubbo:service>没有配置prototol属性时，使用此配置 -->
<dubbo:provider protocol="xxx1" />
```

已知扩展

- `org.apache.dubbo.rpc.injvm.InjvmProtocol`
- `org.apache.dubbo.rpc.dubbo.DubboProtocol`
- `org.apache.dubbo.rpc.rmi.RmiProtocol`
- `org.apache.dubbo.rpc.http.HttpProtocol`
- `org.apache.dubbo.rpc.http.hessian.HessianProtocol`

扩展示例

Maven项目结构：

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxProtocol.java (实现Protocol接口)
                |-XxxExporter.java (实现Exporter接口)
                |-XxxInvoker.java (实现Invoker接口)
    |-resources
        |-META-INF
            |-dubbo
                |-org.apache.dubbo.rpc.Protocol (纯文本文件，内容为：
xxx=com.xxx.XxxProtocol)
```

XxxProtocol.java：

```
package com.xxx;

import org.apache.dubbo.rpc.Protocol;

public class XxxProtocol implements Protocol {
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException
{
        return new XxxExporter(invoker);
    }
    public <T> Invoker<T> refer(Class<T> type, URL url) throws
RpcException {
        return new XxxInvoker(type, url);
    }
}
```

XxxExporter.java：

```
package com.xxx;

import org.apache.dubbo.rpc.support.AbstractExporter;

public class XxxExporter<T> extends AbstractExporter<T> {
    public XxxExporter(Invoker<T> invoker) throws RemotingException{
        super(invoker);
        // ...
```

```
    }
    public void unexport() {
        super.unexport();
        // ...
    }
}
```

XxxInvoker.java :

```
package com.xxx;

import org.apache.dubbo.rpc.support.AbstractInvoker;

public class XxxInvoker<T> extends AbstractInvoker<T> {
    public XxxInvoker(Class<T> type, URL url) throws RemotingException{
        super(type, url);
    }
    protected abstract Object doInvoke(Invocation invocation) throws
Throwable {
        // ...
    }
}
```

META-INF/dubbo/org.apache.dubbo.rpc.Protocol :

```
xxx=com.xxx.XxxProtocol
```