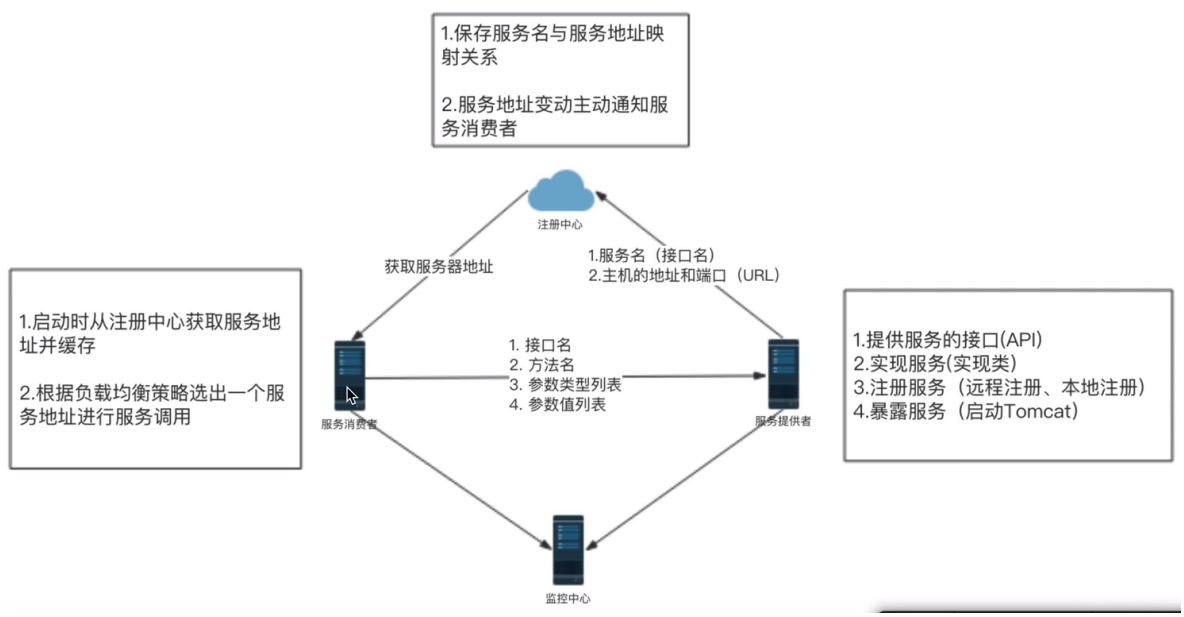
Dubbo

0、前言

dubbo RPC框架帮我们做一个代理，把编解码、联通网络等封装起来，对于上层应用来或者说开发者来说是透明的，方便我们调用，开发和维护。

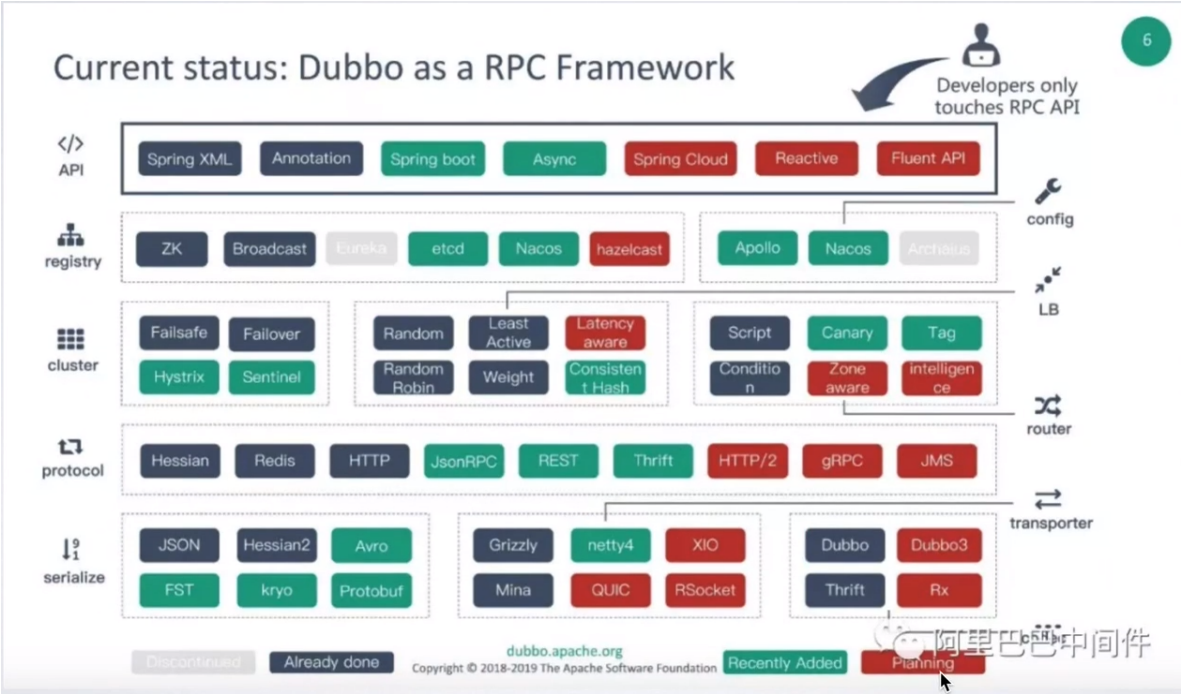
底层进行通信用的是netty框架，基于java的NIO实现的。简化了TCP和UDP的开发。

1、Dubbo架构图



本地注册，主要是将服务的地址和服务的实现类关联绑定起来，注册到本地的服务列表中。消费者消费服务时，通过服务端的服务列表，用服务的地址就可以找到对应的实现类。

2、Dubbo生态图



3、Dubbo SPI

1）扩展适配模块（getAdaptiveExtension）

【ExtensionLoader】扩展加载器类构造函数

初始化获取一个适配工厂（持有所有实现了ExtensionFactory接口的扩展工厂，如Dubbo、Spring、My），目的是在依赖注入时，根据注入的对象，适配出一个合适的扩展工厂，由扩展工厂生产扩展代理（持有扩展实例）。

```
private ExtensionLoader(Class<?> type) {
        this.type = type;
        objectFactory = (type == ExtensionFactory.class ? null :
ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension(
));
}
```

【Provider接收到Consumer请求，动态选择扩展实例】

```
// 1.获取"Driver.class"的扩展加载器
ExtensionLoader<Driver> extensionLoader =
ExtensionLoader.getExtensionLoader(Driver.class);
// 2.根据 SPI Key "trucker" 获取一个扩展实例
Driver driver = extensionLoader.getExtension("trucker");
// 3.根据 SPI Key "black"，动态注入扩展代理CarAdaptive（实现Car接口，持有BlackCar实例）
到Driver中
Map<String, String> map = new HashMap<>();
map.put("carType", "black");
URL url = new URL(protocol, host, port, map);
// 运行时，扩展代理CarAdaptive根据url中的"carType"取出"black"（SPI Key）对应的扩展实例
// 再执行目标方法
driver.driverCar(url);
```

【包装类】

```
public class Trucker implements Driver {
   private Car car;

   public void setCar(Car car) {
      this.car = car;
   }

   @Override
   public void driveCar(URL url) {
      car.getColor(url);
   }
}
```

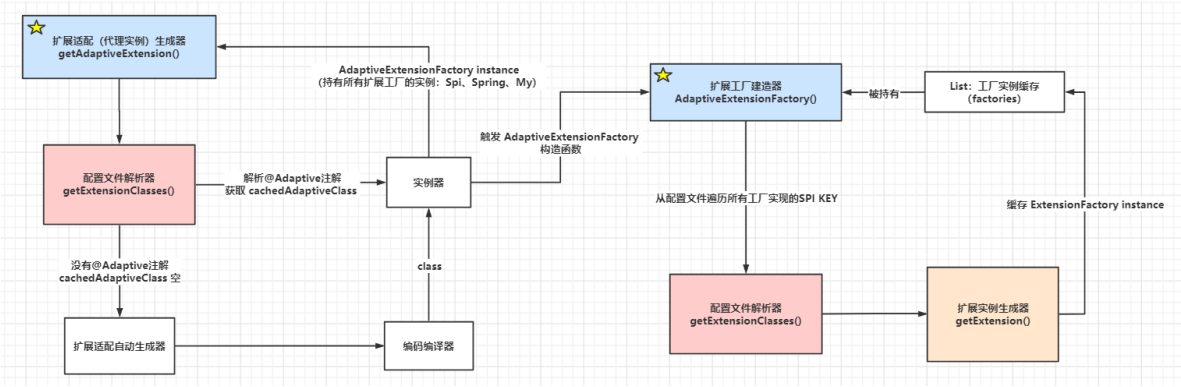
【扩展代理（此处是由Dubbo自动生成的）】

```
public class Car$Adaptive implements com.xxx.Car {
   public void getColor() {
      org.apache.dubbo.common.URL url = arg0;
      String extName = url.getParameter("carType", "black");
      com.xxx.Car extension =
ExtensionLoader.getExtensionLoader(Car.class).getExtension(extName);
      extension.getColor(arg0);
   }
}
```

【模块调用链】

```
T getAdaptiveExtension()
---> T createAdaptiveExtension()
     ---> Class<?> getAdaptiveExtensionClass()

          ---> Map<String, Class<?>> getExtensionClasses()
               ---> Map<String, Class<?>> loadExtensionClasses()
                    ---> void cacheDefaultExtensionName() // 解析@SPI默认SPI Key
                    ---> void loadDirectory(...)
                         ---> void loadResource(...)
                              ---> void loadClass(...)
                                   ---> void cacheAdaptiveClass(...) // 解析
@Adaptive注解类
                                   ---> void cacheWrapperClass(...)  // 解析包装类
                                   ---> void cacheClass(...)         // 解析普通类

               ---> return cachedAdaptiveClass // 若有@Adaptive注解，直接返回注解的
类
               ---> Class<?> createAdaptiveExtensionClass() // 若没有，则Dubbo自动
生成扩展适配
               ---> T injectExtension(T instance) // 注入并返回实例
```

【架构流程图】



扩展适配生成器，目的是获取@Adaptive注解的实例扩展对象，该对象本质是一个代理对象。因为其实现了某个使用@SPI注解的接口（通过解析配置文件获取对应实现），并且重载了接口方法，在方法内部通过spi机制（getExtension()方法实现），在运行时动态地根据传入的参数（SPI Key）来获得最终的扩展实例，并且由该实例执行目标方法。

扩展适配生成器在ExtensionLoader类构造时便使用到。ExtensionLoader是实现 Dubbo SPI 机制的关键类，其通过配置文件解析获得扩展实例的集合与@Adaptive注解的类，同时基于反射实现了扩展实例的依赖注入IOC和AOP功能。为了解决不同类型的依赖的注入，比如除了Dubbo自身的依赖外，还有Spring的依赖，第三方的依赖、甚至我们通过继承ExtensionFactory自定义我们的适配器来实现自定义的依赖等，Dubbo基于前文提到的扩展适配生成器，来生成不同的扩展工厂，以便这些工厂可以很灵活地通过传入的URL动态生成不同的扩展代理。

而扩展工厂的获得，便是在ExtensionLoader扩展加载器构造时去获得的。因为ExtensionFactory扩展工厂本身也是一个@SPI注解的抽象接口，其具体实现的类全路径名也定义在配置文件中，所以在获取其扩展实例时，同样需要解析配置文件，而解析过程中会判断是否有@Adaptive注解的类，有则声明其为扩展工厂实例。

2）扩展模块（getExtension）

【举例】

运行时注入依赖，并执行目标方法

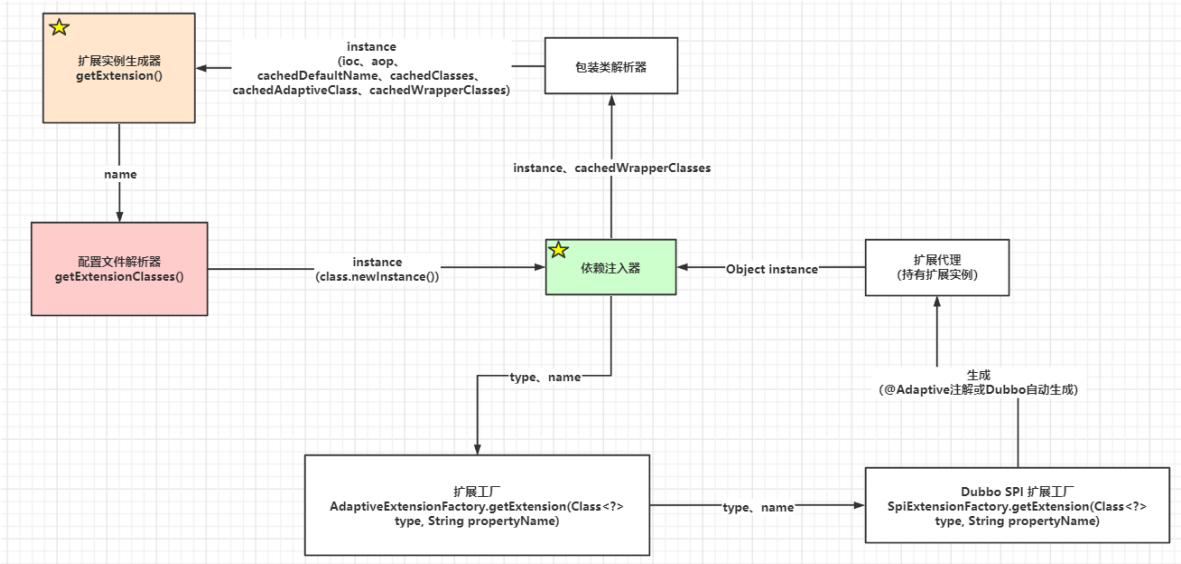
```
// 获取"trucker"的扩展
ExtensionLoader<Driver> extensionLoader =
ExtensionLoader.getExtensionLoader(Driver.class);
Driver driver = extensionLoader.getExtension("trucker");

// 根据 SPI Key "black"，动态注入扩展代理CarAdaptive（实现Car接口，持有BlackCar实例）到
Driver中
Map<String, String> map = new HashMap<>();
map.put("carType", "black");
URL url = new URL(protocol, host, port, map);
// 运行时，扩展代理CarAdaptive根据url中的"carType"取出"black"（SPI Key）对应的扩展实例
// 再执行目标方法
driver.driverCar(url);

// 扩展代理（此处是由Dubbo自动生成的）
public class Car$Adaptive implements com.xxx.Car {
   public void getColor() {
      org.apache.dubbo.common.URL url = arg0;
      String extName = url.getParameter("carType", "black");
```

```
        com.xxx.Car extension =
ExtensionLoader.getExtensionLoader(Car.class).getExtension(extName);
        extension.getColor(arg0);
    }
}
```

【模块调用链】

```
T getExtension(String name)
---> T createExtension(String name)

     ---> Map<String, Class<?>> getExtensionClasses()
          ---> Map<String, Class<?>> loadExtensionClasses()
               ---> void cacheDefaultExtensionName() // 解析@SPI默认SPI Key
               ---> void loadDirectory(...)
                    ---> void loadResource(...)
                         ---> void loadClass(...)
                              ---> void cacheAdaptiveClass(...) // 解析@Adaptive
注解类
                              ---> void cacheWrapperClass(...)  // 解析包装类
                              ---> void cacheClass(...)         // 解析普通类

     ---> T injectExtension(T instance) // 注入上述解析出来的扩展实例，并返回
```

【架构流程图】



在Dubbo中，只有AdaptiveExtensionFactory类被@Adaptive注解，所以Dubbo不需自动为其生成一个便可以声明它是一个扩展工厂实例。在该实例构造时，会再次通过配置解析的方式获取所有实现了ExtensionFacyory的SPI Keys，然后通过getExtension(key)的方式获取所有的扩展工厂实现并缓存。

这一步的目的是使AdaptiveExtensionFactory这个对象持有所有ExtensionFactory的实例，以便在注入时，可以根据对象的类型适配选择出一个合适的扩展工厂，再由扩展工厂生产出扩展代理。

运行时，如上述代码 driver.driveCar(url) ，扩展代理根据url中的参数，取得 SPI Key，再获取扩展实现，从而通过扩展实现，执行目的方法。就像这个对象名字Adaptive ExtensionFactory一样，该对象目的就是为了适配正确的扩展工厂。

3）子功能模块

【配置文件解析器】

- 解析 @SPI 注解的默认 SPI Key
- 解析 @Adaptive 注解的类，缓存为扩展适配 cachedAdaptiveClass（只有一个）
- 解析出包装类，缓存为 cachedWrapperClasses
- 解析出普通类

模块调用链：

```
Map<String, Class<?>> getExtensionClasses()
---> Map<String, Class<?>> loadExtensionClasses()
     ---> void cacheDefaultExtensionName()  // 解析 @SPI 注解的默认 SPI Key
     ---> void loadDirectory(...)
          ---> void loadResource(...)
               ---> void loadClass(...)
                    ---> void cacheAdaptiveClass(...) // 解析 @Adaptive 注解的类
                    ---> void cacheWrapperClass(...) // 解析出包装类
                    ---> void cacheClass(...) // 解析出普通类
```

配置文件解析路径：

- META-INF/dubbo/internal
- META-INF/dubbo
- META-INF/services

【依赖注入器（IOC）】

- 解析set方法参数，取出依赖的类型（.class对象，按需实例化）
- 解析set方法名，取出注入的属性
- 从扩展工厂取出扩展代理（持有扩展实例）
- 反射调用完成set方法的依赖注入，得到instance实例

【包装类解析器（AOP）】

- 将instance封装到包装类cachedWrapperClasses中
- 反射创建包装类实例，注入依赖
- 再将包装类赋值给instance返回

【ExtensionFactory 扩展工厂】

- SpiExtensionFactory：Dubbo自己的Spi去加载Extension
- SpringExtensionFactory：从Spring容器中去加载Extension
- AdaptiveExtensionFactory：自适应的AdaptiveExtensionLoader
- MyExtensionFactory：自定义扩展工厂

【总结】

1）问题的出现

Dubbo SPI 是对 Java SPI 机制的一种改进，是一种对同个接口能得到多种扩展实例的实现方案。它之所以要改进，主要是为了解决2个问题。一个是解决加载扩展点的时候的性能问题，另一个是解决依赖注入的问题，即 IOC 和 AOP 的功能实现的问题。

2）全部加载部分需要、键值对与缓存

首先为什么需要解决性能问题，是因为 Java SPI 在加载某个扩展点的时候，是一次性将所有的扩展实例都加载并实例化的，如果扩展实例很多那么加载的开销很大，而且当中如果只有少部分实例需要用到，其他大部分不需要的话，那么这种开销无疑是很浪费的。所以 Dubbo SPI 做了改进，一方面将配置文件做成了 key-value 的形式，这样的话就支持通过某个 key 来获得对应的扩展实例，就不用全部都去加载；另一方面，与 Java SPI 不同的是，Dubbo SPI 首先将具体的扩展类先加载到 JVM 并且缓存起来，在需要真正使用的时候，才进行实例化，这样的按需实例化也降低了系统资源的开销，整体上做到了性能的提升。

3）加载扩展实现类

所以当应用启动的时候，首先会去 Dubbo 指定的路径下，把所有的扩展类都进行加载并缓存起来，再按需进行实例化，那么缓存中就有类对象的缓存和实例对象的缓存。当需要某个对象的时候，首先会去实例缓存中找，如果没有再从类对象中找到类进行实例化。

4）依赖注入（构造器和setter函数）

对象实例化后就需要进行初始化，初始化就需要对属性的值进行设置，就是进行一个依赖注入，依赖注入常用的是构造器注入和setter函数注入，那么首先会进行构造器注入，这里就是为了解决一个 AOP 的问题，当一个对象的构造函数中包含有其他的扩展点作为参数时，那么 Dubbo 认为这就是一个 AOP 对象，因为它把抽象的公共逻辑抽取出来放到当前类，然后在构造器注入一个真正目标对象，这就类似动态代理在目标方法周围织入公共逻辑对目标进行增强，然后 Dubbo 也会从缓存或者新建一个对象进行设置，然后返回。另一个依赖注入的方式是 setter 函数，Dubbo 会通过反射获取所有方法，然后找到以 set 开头的函数，解析它的参数的类型和属性的名称，然后在缓存中或者新建一个对象通过反射设置进去。

5）适配器模式（注解与自动生成）

正常来说，这里就完成了 AOP 和 IOC 的功能。但有一个问题，上面注入的都是对象，可以在缓存中找到，但如果是接口呢，有多个实现但不知道当下要注入哪一个的时候，这个时候就需要用到另一个特性功能，就是 Dubbo 的自适应功能，自适应的意思就是通过动态地接受一个 SPI KEY，自动生成匹配的扩展实例来适应当前的业务场景。如果是我们自己实现的话，我会写一个对象，对象中有一个函数，功能是通过输入一个 Class 对象，以及一个 String 类型的 SPI KEY，然后输出一个具体的扩展实例，那么这个对象就是一个适配器对象。那么其实 Dubbo 也是这么做的，只不过它更加简洁，它提供两种方式，一种是用 @Adaptive 注解来修饰刚才说的那些适配器对象，当应用启动时 Dubbo 会扫描并解析 @Adaptive 注解然后把对应的适配器类缓存起来，另一种方式是没有注解时，Dubbo 会自动通过生成代码字符串然后进行自动编译，自动生成一个适配器对象的方式，两种方式总之都是要生成适配器对象，然后适配器对象根据动态传入的 SPI KEY，再找到对应的扩展实例。

6）适配器工厂（Dubbo SPI、Spring、My）

但这里的 SPI KEY，其实可以有不同框架的实现，比如 Dubbo 本身容器管理的实例，以及 Spring 容器管理的实例等，换句话说，当注入的对象是 Spring 容器管理的一个接口时，那么就需要到 Spring 的依赖下的 SPI 路径去找到对应的扩展实例，所以这里是典型的工厂模式，要有多个工厂根据不同业务场景产生不同实例，所以这里首先还需要工厂对象，而工厂可以有 Dubbo、Spring、以及用户自定义的，所以工厂同时又是一个 @SPI 注解修饰的抽象接口，也需要通过 SPI 机制去获取不同的实现。刚才说了 Dubbo 有两种方式生成适配器一种是注解一种是自动生成，Dubbo 是用一个 @Adaptive 注解了一个 AdativeExtensionFactory 对象，然后由这个对象去实例化所有的工厂对象并缓存起来。再回到依赖注入的场景时，就会通过属性的类型即类对象，以及动态传入的一个 SPI KEY，首先匹配到合适的工厂，再由工厂生成一个适配器对象，然后适配器对象根据传入的 SPI KEY 基于 SPI 机制找到扩展实例，最终执行实例方法。

> **问题：性能、依赖注入（IOC、AOP）**
>
> **性能解决：全部加载部分需要、键值对与缓存**
>
> **加载扩展实现类**
>
> **依赖注入（构造器和setter函数）**

> **适配器模式（注解与自动生成）**
>
> **适配器工厂（Dubbo SPI、Spring、My）**

@Activate

@Adaptive 只能激活一个实现，而 @Activate 可以可以根据不同条件激活多个扩展实现。

@Adaptive

- 注解在类上，则会将类进行缓存，作为一个适配器的默认实现，不用再通过 Dubbo 自动生成
- 注解在方法，且类没有的话，则 Dubbo 会自动生成一个适配器对象，包含所注解的方法

@SPI

可声明一个接口的多个实现需要通过 Dubbo SPI 来加载和获取，可以有默认值，优先级低于 @Adaptive

SpringExtensionFactory

是一个 Spring 的适配器工厂对象，用来生成基于 Spring SPI 的适配器（到 Spring 的路径下找扩展实例）

工厂提供了可以保存 Spring 上下文的方法，该上下文是在服务发布和引用的时候被保存下来的。在获取实例时，会根据名字再根据类型去 Spring 的 IOC 容器找对象。

getExtension()

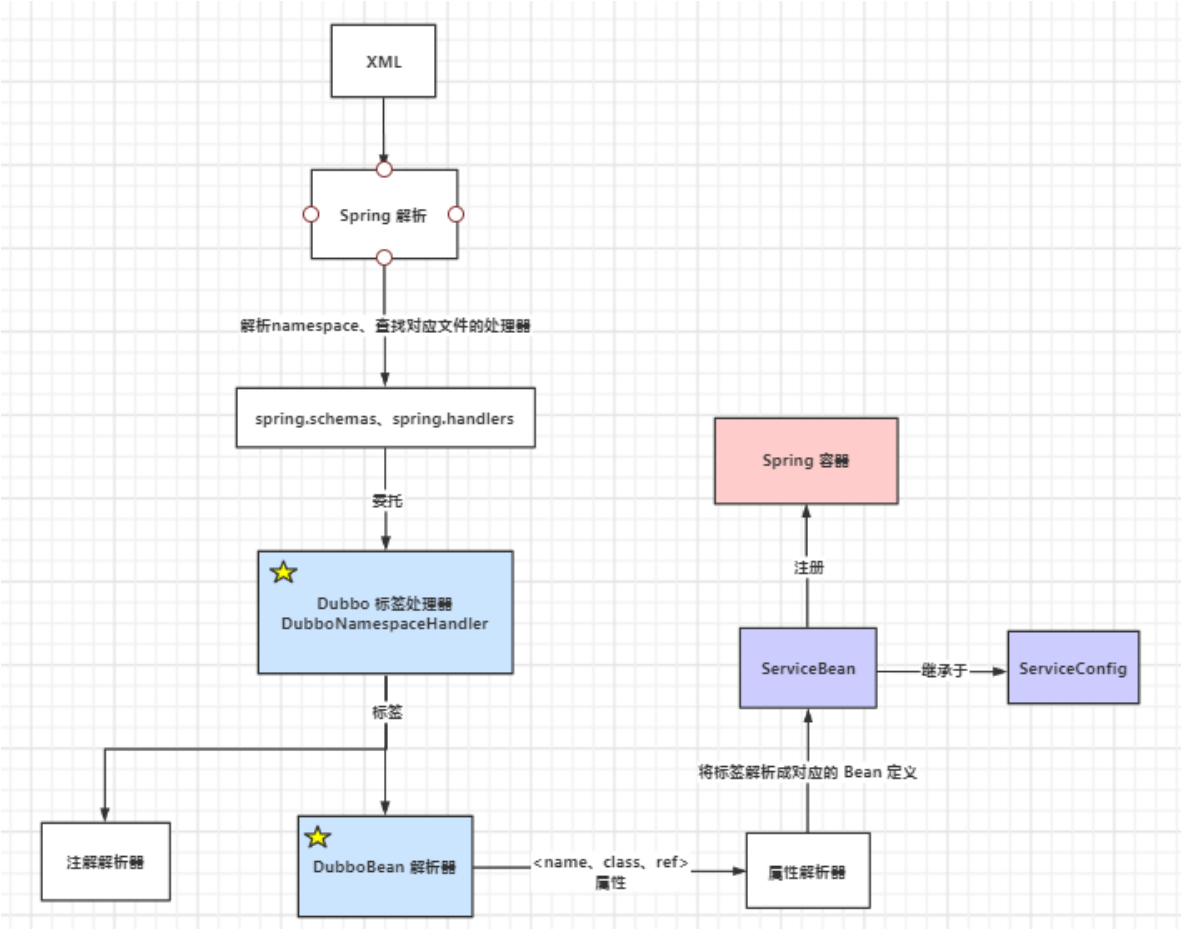
获取路径下的所有扩展实例。扫描并加载扩展实例，完成缓存、实例化、AOP、IOC 等功能。

getAdaptiveExtension()

获取所有适配器对象。通过 getExtension 方法获取路径下所有扩展实例，扫描 @Adaptive 注解并缓存，并根据需要实例化适配器对象，如果没有注解的会自动生成编译一个。涉及到多个实例工厂的加载。

4、标签解析

【架构流程图】



Spring启动流程中包含对XML文件的解析，在解析到 namespace 标签时，比如 dubbo:service 标签，就会查找对应的 spring.schemas 和 spring.handlers 文件，在dubbo中，对于dubbo标签的解析，已在这两个文件中指定了一个标签处理器 DubboNamespaceHandler。

DubboNamespaceHandler 主要把不同的标签交给不同的解析器去处理，比如和服务相关的service标签就是交给 DubboBean 解析器来解析，其他的还有一个注解解析器。DubboBean 解析器还包含了一个属性解析器，主要解析比如 dubbo:service 标签中的name、class、ref等属性。

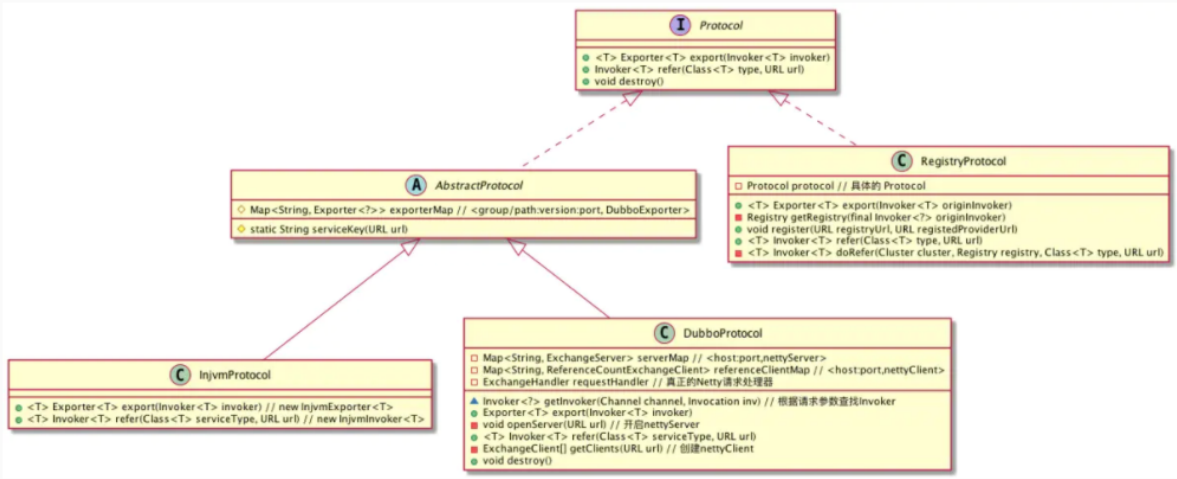
经过一顿操作之后，标签解析器就会把标签解析成对应的bean，并注册到 Spring 容器中。

> **命名空间、配置文件、标签解析器、属性解析器（name、class、ref）、注册到 IOC**

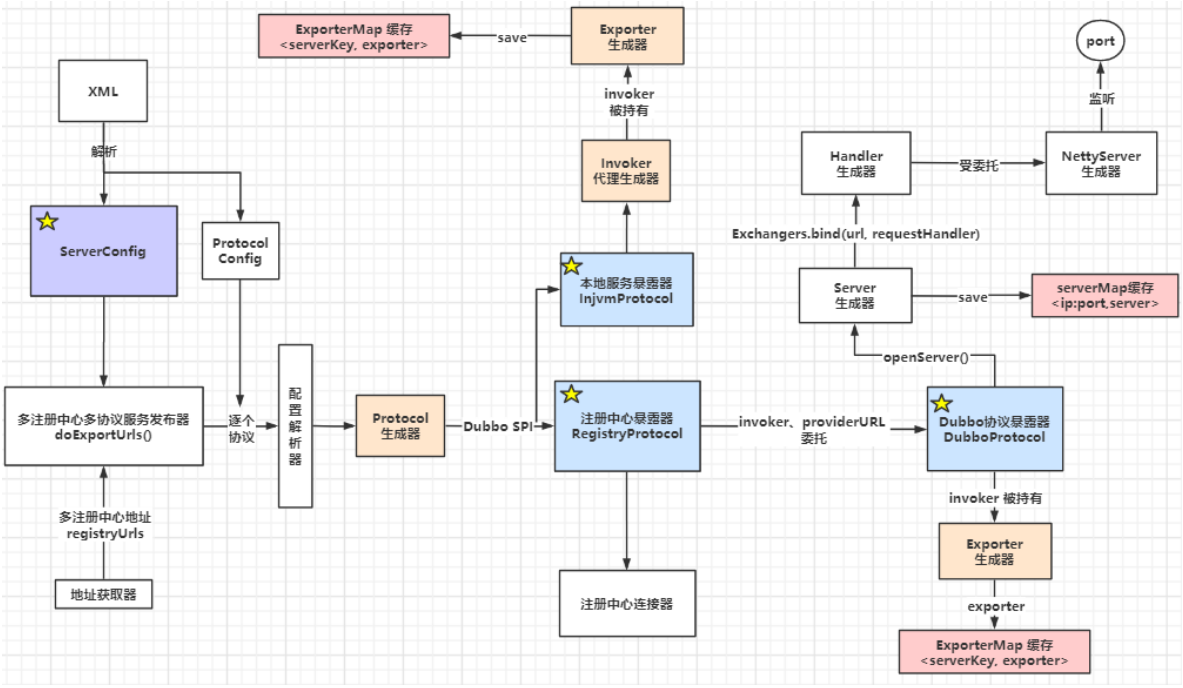
5、服务暴露

【核心概念】

- **Invoker**：是 Dubbo 的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起 invoke 调用。
- **Protocol**：是 Invoker 暴露和引用的主功能入口，它负责 Invoker 的生命周期管理。


Protocol
<T> Exporter<T> export(Invoker<T> invoker)
Invoker<T> refer(Class<T> type, URL url)
void destroy()
AbstractProtocol
Map<String, Exporter<?>> exporterMap // <group/path:version:port, DubboExporter>
static String serviceKey(URL url)
RegistryProtocol
Protocol protocol // 具体的 Protocol
<T> Exporter<T> export(Invoker<T> originInvoker)
Registry getRegistry(final Invoker<?> originInvoker)
void register(URL registryUrl, URL registedProviderUrl)
<T> Invoker<T> refer(Class<T> type, URL url)
<T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url)
InjvmProtocol
<T> Exporter<T> export(Invoker<T> invoker) // new InjvmExporter<T>
<T> Invoker<T> refer(Class<T> serviceType, URL url) // new InjvmInvoker<T>
DubboProtocol
Map<String, ExchangeServer> serverMap // <host:port,nettyServer>
Map<String, ReferenceCountExchangeClient> referenceClientMap // <host:port,nettyClient>
ExchangeHandler requestHandler // 真正的Netty请求处理器
Invoker<?> getInvoker(Channel channel, Invocation inv) // 根据请求参数查找Invoker
Exporter<T> export(Invoker<T> invoker)
void openServer(URL url) // 开启nettyServer
<T> Invoker<T> refer(Class<T> serviceType, URL url)
ExchangeClient[] getClients(URL url) // 创建nettyClient
void destroy()

【架构流程图】


ExporterMap 缓存
<serverKey, exporter>
save
Exporter
生成器
port
XML
invoker
被持有
监听
解析
Invoker
代理生成器
Handler
生成器
受委托
NettyServer
生成器
ServerConfig
Protocol
Config
Exchangers.bind(url, requestHandler)
本地服务暴露器
InjvmProtocol
Server
生成器
save
serverMap缓存
<ip:port,server>
openServer()
多注册中心多协议服务发布器
doExportUrls()
逐个
协议
配置解析器
Protocol
生成器
Dubbo SPI
注册中心暴露器
RegistryProtocol
invoker、providerURL
委托
Dubbo协议暴露器
DubboProtocol
多注册中心地址
registryUrls
invoker 被持有
Exporter
生成器
地址获取器
注册中心连接器
exporter
ExporterMap 缓存
<serverKey, exporter>

【核心调用链】

```
ServiceConfig
// 参数获取
void export(
  // 参数校验
  void doExport()
    // 获取注册中心地址，进行多注册中心多协议发布
    void doExportUrls()
      // 解析配置，填充map；本地暴露；获取代理invoker；protocol远程暴露;
      void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs)
        // 本地暴露
        void exportLocal(URL url)

        // 注册中心暴露：委托DubboProtocol暴露；创建注册中心，建立连接；注册服务信息到注册中心；订阅节点；销毁服务；
        RegistryProtocol#Exporter<T> export(final Invoker<T> originInvoker)

          // 委托DubboProtocol暴露
          ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker, URL providerUrl)
            // 构造服务serverKey；构造DubboExporter封装invoker（持有URL和REF）；维护exporterMap（serverKey, dubboExporter）
            DubboProtocol#ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker, URL providerUrl)
              // 创建服务器server（如netty），监听端口；创建serverMap（ip:port, server）
              void openServer(URL url)
                // 创建服务器server
                ProtocolServer createServer(URL url)
                  // 绑定requestHandler
                  Exchangers#ExchangeServer bind(URL url, ExchangeHandler handler)
                    // 传输层绑定
                    Transporters#RemotingServer bind(URL url, ChannelHandler... handlers)
                      // 选择具体协议绑定（netty）：创建netty服务器和客户端，监听端口，处理请求
                      NettyTransporter#RemotingServer bind(URL url, ChannelHandler handler)

          // 创建注册中心实例，建立TCP连接
          Registry getRegistry(final Invoker<?> originInvoker)
          // 注册服务元数据信息到注册中心（zookeeper）
          void register(URL registryUrl, URL registeredProviderUrl)
```

【小结】

标签解析

Dubbo 配置文件中的dubbo:service标签经过解析后会变成Spring中的bean，并注册到Spring的上下文。然后就要进行服务的暴露。Dubbo可以支持多注册中心多协议的暴露，多注册中心包括zookeeper、Eurkes等，所以dubbo注册中心的获取是基于 Dubbo SPI 机制来选择的，当然也可以同时配置多个配置中心。像这种可以支持有多种不同扩展的情况，在Dubbo中都是基于SPI机制来选择最终的扩展实例的，比如配置中心、协议等。

URL、Invoker、Exporter

得到注册中心地址后，再根据解析出来的协议配置，得到配置的多个协议。这个过程也是基于Dubbo SPI机制的。
如果是多协议的情况，就遍历每一个协议，使用每种协议分别对多个注册中心暴露服务。暴露的过程，首先是解析由xml解析后产生的各种config，来构造一个URL。这里的URL，是配置信息的基本载体，包含了注册中心的信息，服务器的地址、服务提供者的信息、服务者提供的服务的信息，比如接口名、方法名、参数等。

这个URL是各阶段信息交流的一个通用载体，用于在服务暴露过程的各个阶段中，构造其他的关键结构体，比如服务实例的代理对象invoker，这个invoker也是基于SPI通过不同的代理机制来生成的一个代理对象，它通过一个Wrapper包装类，封装了invokeMethod方法，这个方法代理了真实服务的实现逻辑。调用时通过客户端的请求，请求中有目标的接口和方法名以及方法参数，通过这几个来参数匹配出真实服务并进行调用，所以Invoker简单说可以看成是一个持有真实服务实现的代理对象。还有一个是在服务最终暴露阶段，会把invoker封装成exporter对象，它封装并管理着invoker生命周期的exporter等。

本地暴露

在构造完URL后，就进入服务真正暴露的阶段。在Dubbo中，服务的暴露是在Protocol层进行的，Protocol也是一个扩展，基于SPI机制可以扩展出本地协议InjvmProtocol、注册中心协议RegistryProtocol、Dubbo协议DubboProtocol等。本地服务使用的事InjvmProtocol协议进行暴露，作用域主要是针对在同个JVM既有Provider和Consumer，然后Consumer消费了该JVM上的Provider的情

况，属于进程间的通信，不需要远程RPC通信。所以为了提高同个JVM的服务消费效率，使用了本地暴露。具体的就是将ref封装并生成一个代理对象invoker，然后再封装到exporter对象的exporterMap中进行维护。

简单地说，这里其实就是把一个服务的标识serverKey，和对应的exporter的映射关系（程序中用map维护）缓存到内存中，下次客户端调用时，如果判断是本地服务，那么就根据客户端请求生成的服务标识serverKey，在exporterMap中直接取得对应的exporter，再通过exporter取得invoker，进行调用即可。

ExporterMap

这里有必要说一下的就是这个exporterMap，是一个映射，存储了服务的标识（可以由服务地址、服务接口名和方法名等规则生成）和对应的exporter，它是存储于Protocol层的一个缓存Map，程序中具体是AbstractProtocol 抽象协议对象中，无论是哪个扩展实现了它，都可以从这个抽象对象中取得该协议下的所有服务和exporter的对应关系。本地暴露就是把服务放到jvm内存中，以便本地消费者直接获取。

远程暴露 - DubboProtocol

然后就是进入远程暴露，通过注册中心暴露器 RegistryProtocol 进行暴露，注册中心暴露首先委托给具体的协议进行暴露，这里因为有多个协议扩展，比如Dubbo协议、Http协议等，所以依然是基于SPI协议选择一个扩展，比如DubboProtocol协议暴露，dubbo协议底层基于netty框架搭建服务器，因为netty的多路复用和事件驱动模型可以很好地支持非阻塞的高并发请求，所以首先是创建一个nettyServer服务器，监听端口并且绑定一个requestHandler，把客户端的请求交给这个handler去具体处理。

同时，在创建nettyServer服务器之前，在dubboProtocol协议层同样会把invoker封装到exporter的exporterMap中，将服务的标识serverKey和对应的exporterMap的映射关系维护起来。简单的说，就是exporter持有invoker。之后接收到客户端请求后，就可以提取客户端的请求依据相同的规则构造服务标识serverKey，然后在这个exporterMap中找到对应的exporter，再找到invoker进行方法调用。通过Dubbo协议暴露，其实就是封装exporter，缓存起来。再创建nettyServer准备接收客户端的请求。

远程暴露 - RegistryProtocol

然后刚才说到注册中心暴露器进行暴露，在经过具体协议Dubbo暴露成功后，才会把服务的信息注册到注册中心中，以便客户端消费者可以先从注册中心根据服务名拿到服务的地址，再发送rpc请求到搭建好的nettyServer中来。所以注册中心暴露器 RegistryProtocol 还需要根据URL中的注册中心地址实例化一个注册中心，然后注册服务的元数据信息，再订阅相关的节点，监听服务属性变更事件。

所以这里简单来讲，其实注册中心协议RegistryProtocol这一层是对DubboProtocol这一层的一个封装，相当于是他的AOP切面对象，在RegistryProtocol这一层的核心功能就是通过DubboProtocol去暴露协议，然后之后的AOP功能就是搭建一个注册中心，然后向注册中心注册和订阅数据等，到这里，服务的暴露过程就介绍完了。

服务暴露

1）标签解析、获取注册中心地址与协议信息、构造URL

Dubbo 配置文件中的dubbo:service标签经过解析后会变成Spring中的bean，并注册到Spring的上下文。然后就要进行服务的暴露。Dubbo 可以支持多注册中心多协议的暴露，所以首先从配置文件获取到多个配置中心的地址，以及多种协议，然后遍历每一个协议，使用每种协议分别对多个注册中心暴露服务。

Provider 的服务暴露最终是要向注册中心发送例如一个服务名与一个API调用地址的映射关系，所以围绕这种关系构成，就需要解析配置文件的参数然后组成这样一个 API 地址，在 Dubbo 中是用 URL 的对象来保存 API 地址的，包含了 Provider 这边的一些服务信息，比如接口名、方法名、参数等。

2）本地暴露（定义、封装invoker、缓存exporter）

在构造完URL后，就进入服务真正暴露的阶段。Dubbo 的层级结构中有一层叫 Protocol 协议层，服务暴露就是在这一层进行的。这一层又分为两个模块，一个是本地暴露模块、和远程暴露模块。本地暴露模块，意思是在同个 JVM 中既有服务提供者也有服务消费者，那么服务提供者只需要把服务信息缓存在 JVM 本地，然后消费者需要消费的时候，就直接取本地 JVM 的缓存的服务进行调用就行，以提升服务调用的效率。这个过程发生在Protocol 层的 InjvmProtocol 协议模块中，具体的就是将真正服务的实现对象封装到一个代理对象 invoker 中，然后再封装到一个 exporter 对象中，然后放到一个 Map 里，Map 的 value 就是 exporter 对象，key 就是一个服务的标识 serverKey，是用接口名、方法名、参数名等规则形成的，来唯一标识一个服务的 key。那么下次客户端调用时，如果判断是本地服务，那么就根据客户端请求生成的服务标识serverKey，在exporterMap中直接取得对应的exporter，再通过exporter取得invoker，进行调用即可。

invoker 和 exporter 的关系是，invoker 是一个代理对象，持有真正服务的实现者，通过接受客户端请求，解析出请求中的服务信息比如接口名、方法名、参数等来匹配出一个服务的实现者，然后调用服务的实现方法，最终完成一次 RPC 调用。而 exporter 封装了 invoker，负责管理其生命周期，并作为最终的服务包装对象暴露给消费者。

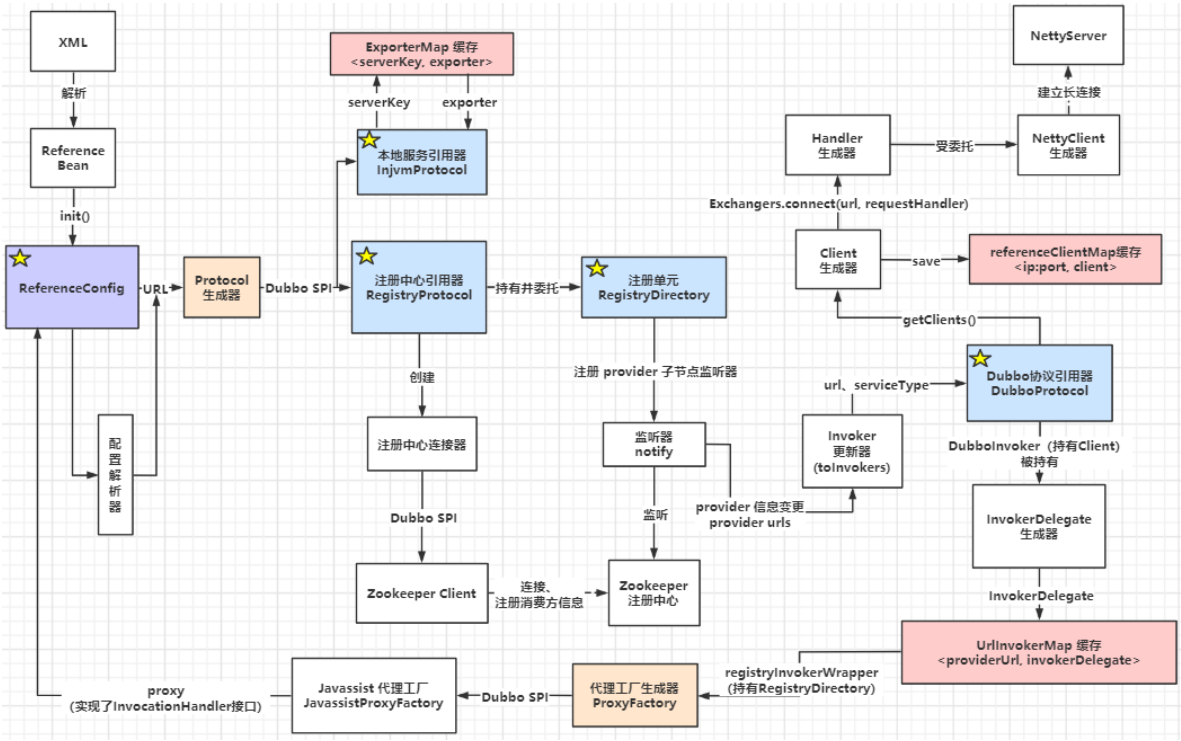
3）远程暴露（DubboProtocol 创建 netty 服务器、缓存 exporter，RegistryProtocol 创建注册中心并发布服务信息）

然后进行远程暴露，就是将服务信息注册到注册中心，可以通过 HTTP 协议或者 Dubbo 协议进行暴露，也是通过 SPI 来选择的，默认是 Dubbo 协议，那么就会委托给具体的 DubboProtocol 模块去进行。Dubbo 底层通信基于 Netty 框架的，所以首先是创建一个nettyServer服务器，监听端口并且绑定一个requestHandler，把客户端的请求交给这个handler去具体处理。同时 DubboProtocol 模块会把 invoker 就是服务的代理执行对象封装到 exporter，像之前本地暴露一样，也将服务的标识和对应的 exporter 的对应关系缓存到一个 Map。之后接收到客户端请求后，就可以提取客户端的请求依据相同的规则构造服务标识serverKey，然后在这个exporterMap中找到对应的exporter，再找到invoker进行方法调用。在经过 DubboProtocol 模块搭建好 netty 服务器以及缓存好服务标识与对应的代理对象后，再通过 RegistryProtocol 模块创建注册中心，并发布服务信息到注册中心去。

> **服务暴露**
>
> **1）标签解析、获取注册中心地址与协议信息、构造URL**
>
> **2）本地暴露（定义、封装invoker、缓存exporter）**
>
> **3）远程暴露（DubboProtocol 创建 netty 服务器、缓存 exporter，RegistryProtocol 创建注册中心并发布服务信息）**

6、服务引用

【架构流程图】



【调用链】

```
FactoryBean.getObject()
ReferenceConfig.init()
  // 创建动态接口代理，持有Invoker和netty客户端，承担实际服务调用的功能
  ref = createProxy()
    // 本地服务引用，直接内存中通过serviceKey从exporterMap取得exporter和invoker
    invoker = InjvmProtocol.refer()
    // 服务远程引用，实现AOP引用功能，即除了发起真正远程引用外，还需要连接注册中心，
    // 订阅服务提供者的节点信息，创建节点监听器，处理回调数据等
    invoker = RegistryProtocol.refer()
      registryInvokerWrapper(registryDirectory) = doRefer()
        // 创建注册单元，专门负责监听provider节点信息，承担服务的发现功能，并据此创建invoker
        registryDirectory = new RegistryDirectory()
        // 节点回调函数（监听器）
        RegistryDirectory.notify()
          // 服务的发现，创建invoker
          RegistryDirectory.notify.Map<String, Invoker<T>> toInvokers();
            // 将invoker封装为InvokerDelegate，持有invoker和对应的provider的服务地址
            InvokerDelegate = DubboProtocol.refer()
              // 创建invoker的同时，创建nettyClient
              ExchangeClient[] clients = getClients()
              // 将client封装到dubboInvoker中，即invoker持有client，承担连接与发送的功能
              dubboInvoker = new DubboInvoker(clients)
        // 注册消费信息到注册中心
        registry.register()
        // 注册单元订阅服务提供者、路由和动态配置
        directory.subscribe()
    proxy = ProxyFactory.getProxy(invoker)
```

【小结】

接口代理

客户端的服务引用过程，主要是通过一个代理对象发起的，这个代理对象持有一个invoker对象，当调用发生时，代理对象将委托这个持有的invoker发起实际的调用，而这个invoker又持有了可以发起远程连接并发送请求的nettyClient客户端，所以说，发起服务调用的核心，是这个invoker对象，它承担了连接服务端发起请求并返回结果的功能，通过代理的方式屏蔽了对上层的调用细节，所以看起来一个

RPC请求就像一个本地调用一样。

所以服务的调用，首先需要把Invoker准备好。这个流程是，首先spring将消费端的xml配置文件解析成bean并注册在容器中。服务端调用时，首先通过getObject方法从spring工厂中获得服务的引用对象ReferenceBean，由于ReferenceBean继承于ReferenceConfig，在getObject方法发生时，便触发了ReferenceConfig的初始化，初始化的工作核心是通过createProxy方法获得一个接口代理，这个代理绑定了一个handler对象，所以对代理发起调用，实际上都会交给这个handler去处理，这个handler持有一个invoker对象，就像上文说的，调用发生时实际上底层是通过invoker来发起的。

引用不是真正，因为通过远程协议引用到远端的真正服务，只是封装起来，屏蔽了发起远程请求的过程，看起来

invoker创建

所以下面是围绕invoker的创建来展开的。首先会进入Protocol层，调用refer方法发起服务的引用，返回一个invoker对象。protocol层是协议层，有不同的协议扩展，比如用于本地引用的InjvmProtocol协议，用于注册中心做AOP引用功能的RegistryProtocol协议，以及实际发起远程调用的DubboProtocol协议等，具体扩展的实现是基于Dubbo SPI机制，通过传入的参数的SPI Key关键字来适配的。

本地引用

所以，在protocol层，首先会判断这个引用是否属于本地引用。这里有必要说一下，配置文件的标签经过解析后会生成若干个分别与标签相对应的config对象，然后在protocol引用阶段，会将这些config进行属性的提取，将属性和对应的值提取成key-value的形式维护在一个属性配置Map中。判断是否本地引用，就是查看map中的consumer属性的地址是否和当前jvm的地址一样，包括ip和端口，如果是则属于本地引用，则直接根据reference属性，根据要引用的接口和方法等生成一个serverKey，然后在内存中的存储的map中取出对应的本地invoker即可。这里内存存储的map，是属于服务暴露过程中本地暴露的阶段产生的对象，服务暴露时首先会将服务的标识serverKey，比如根据接口名接口方法等生成的一个标识，与对应的服务暴露者exporter一起维护在map中，而exporter又持有代理对象invoker（持有具体服务实现），所以当服务引用时判断到是同属一个jvm的引用，则直接会在该内存中取得的暴露者exporter，再通过exporter取得代理对象invoker返回，然后执行服务的调用逻辑。

注册中心引用 - 注册单元设置监听处理回调

刚才说的是在protocol层首先进行服务的本地引用的过程，如果服务是远程引用，还需要区分是单注册中心还是多注册中心的引用，因为这两者核心基本原理相同，所以我就以单注册中心的情况来说明服务的引用过程。首先会通过Dubbo SPI机制取得RegistryProtocol扩展，执行refer方法和doRefer方法。在doRefer方法中，首先会创建一个注册单元RegistryDirectory ，它实现了一个监听的接口，会有一个回调函数notify，专门负责监听provider节点信息，进行服务的发现，当dubbo消费端第一次发起对注册中心的provider的节点的订阅时，会做一次全量的数据拉取操作，同时会触发这个回调函数notify，然后根据回调到的provider的信息通过toInvokers方法创建invoker。

DubboProtocol引用

这个invoker持有一个服务的调用过程，是通过DubboProtocol的refer方法实现的。也就是说invoker的创建又转交由DubboProtocol来实现，首先会创建一个客户端client，底层其实是创建了一个nettyClient，负责向服务端发起连接和请求，并接收返回的客户端。然后将这个client封装到dubboInvoker中返回，所以刚才说Invoker持有服务的调用过程，实际上就是指它持有client，承担发起服务调用的功能。接着这个dubboInvoker又封装了一层成为InvokerDelegate，然后将这个InvokerDelegate与providerUrl做一个映射存储在注册单元即RegistryDirectory的UrlInvokerMap中。这样做的好处就是，在服务发起调用时，首先会根据请求生成一个RPC调用结构rpcInvocation，然后通过proxy代理将请求交给handler对象，handler持有registryInvokerWrapper，即invoker的包装类，这个包装类持有注册单元，再通过rpcinvocation就可以在注册单元RegistryDirectory根据providerUrl取出相应的Invoker。

注册中心引用 - 连接注册中心、注册消费端消息、订阅服务端节点

到这里就就是服务引用过程通过具体协议DubboProtocol进行引用的过程，通过DubboProtocol的refer方法，最终将返回一个invoker对象，它持有一个服务的调用过程。而这个invoker也会被缓存在protocol层的一个map中，在服务发起真正调用时，可以通过服务的信息比如服务地址providerUrl来取得这个invoker发起调用。注册中心引用即RegistryProtocol除了通过DubboProtocol发起真正引用外，还有其他的AOP功能，之后就是根据注册中心的地址，创建一个注册中心实例，连接上，并且设置对于provider节点的监听和回调处理逻辑，并且注册消费端这边的信息到注册中心上去，同时订阅其他节点，比如路由或者配置节点等，以便服务端信息变更时消费端可以有相应的反应。

【总结】

服务的引用，就是获取一个引用对象，即代理对象。这个引用不是真正的服务实现，而是通过协议的方式引用到远端的真正服务，只是封装起来，屏蔽了发起远程请求的过程，看起来像是本地调用一样。

1）本地引用（服务标识、invoker）

服务的引用包括本地引用和远程引用，和之前服务的暴露一样，也是在 protocol 层实现的，服务的本地引用指的是服务的提供者和消费者在同一个 JVM 中，那么消费者只需要根据所要请求的服务的接口名、方法名和参数等相关信息组成一个服务的唯一标识，然后在本地的服务缓存中根据服务标识这个key，取出对应的服务的代理对象invoker，再直接调用即可。因为之前的服务的本地暴露过程中，就已经将相关的服务信息缓存起来了。Dubbo 是如何判断是否属于本地引用的呢，它是根据我们配置文件中的服务提供者的ip与本地ip做对比，如果一样，就说明是本地引用。

2）远程引用（注册单元进行服务的发现）

如果不是本地服务，那么就要进行远程引用。要进行服务引用，就需要有远端服务提供者的服务信息，比如每个服务名与对应的 API 地址等，就需要去注册中心拉取服务信息，然后缓存在本地服务列表。消费者再根据将要引用的服务信息到本地服务列表中找到对应的API，再发起调用。所以这里面需要两个角色，一个是负责服务信息的拉取，一个是负责服务的调用。对应到 Dubbo 中，分别是注册单元 RegistryDirectory 对象和具体协议的引用者 DubboProtocol 对象。注册单元负责服务的发现，当消费端第一次订阅注册中心 provider 节点信息时，会做一次全量的拉取，然后回调到注册单元的回调函数中，再根据这些回调到的服务提供者的信息创建不同的服务标识对应的 invoker，最终形成不同服务的服务标识（服务的地址）对应不同 invoker 对象的局面，其实很容易理解，因为不同的服务对应不同的调用过程，本质是因为 API 地址不一样。

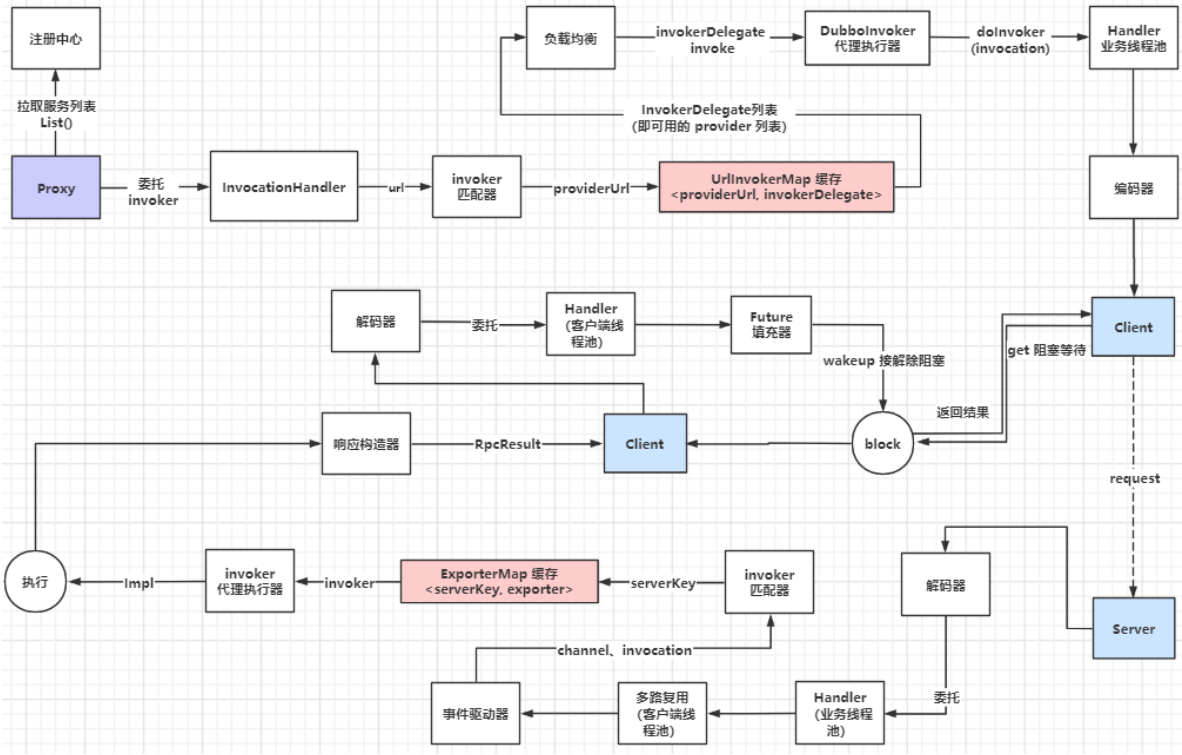
3）远程引用（DubboProtocol 进行服务的调用、创建 Netty 客户端、缓存服务 invoker）

服务信息拉取到了之后，就需要发起调用。通过 DubboProtocol 模块首先创建一个 netty 客户端，负责向服务提供者发起请求并接收响应，然后把 netty 客户端封装到 invoker 中，所以说 invoker 才是真正服务引用的底层实现，承担着服务的调用功能。同时 DubboProtocol 将不同的服务标识与不同的 invoker 对象做一个映射并缓存到注册单元的一个 Map 中。当服务发起调用时，首先会将请求发给代理对象，而代理对象通过请求的参数组成一个服务标识，再通过注册单元中缓存有服务信息的 Map 匹配得到一个最终的 invoker，而 invoker 持有 netty 客户端，就通过它向服务提供者发起请求，并接收返回的结果，这就是一次服务引用的过程。如果 provider 是集群部署的话，就会取到多个 provider 信息和对应的 invoker 对象，dubbo 会根据负载均衡策略选择其中一个 invoker 发起调用。

> **服务引用**
>
> **1）本地引用（服务标识、invoker）**
>
> **2）远程引用（注册单元进行服务的发现）**
>
> **3）远程引用（DubboProtocol 进行服务的调用、创建 Netty 客户端、缓存服务 invoker）**

7、服务调用

【架构流程图】

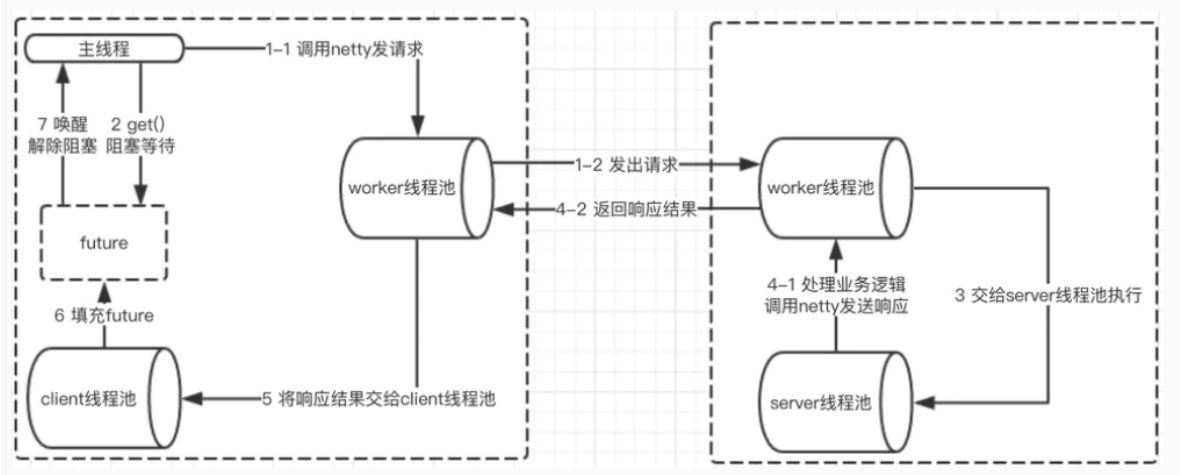


服务发起调用时，其实是通过一个代理对象发起的，是在服务引用阶段通过ReferenceConfig对象的createProxy方法获得的，该对象通过内置的handler处理器持有最终的invoker对象，通过该对象发起真正的请求。

代理首先会通过List方法向注册中心拉取服务列表，然后根据调用的请求生成一个RPC调用结构rpcInvocation，然后通过proxy代理将请求交给handler对象，handler持有registryInvokerWrapper，即invoker的包装类，这个包装类是在protocol层通过RegistryProtocol的refer方法得到的。这个包装类持有注册单元RegistryDirectory，注册单元在创建时就对注册中心的provider的节点进行监听，在消费端第一次订阅数据时，通过回调的数据创建了invoker，然后将服务的地址和Invoker的映射关系Map保存在内存中，所以在服务真正调用时就可以通过rpcinvocation在注册单元RegistryDirectory根据providerUrl取出相应的Invoker。

这个时候如果是集群部署的话，服务列表就会有多个provider，而每个provider对应一个Invoker，所以这里会取到多个invoker，再根据预设的负载均衡策略，通过Dubbo SPI生成负载均衡器进行负载均衡，选出一个invoker进行调用。此时层层解封装后，这个invoker是一个DubboInvoker，是在DubboProtocol服务引用时生产的，它持有一个可以向服务端发送请求的客户端client，执行代理的invoker方法，最终将调用DubboInvoker的doInvoker方法，然后使用Clinet向服务端发起连接并发送请求。

【响应结果异步转同步流程图】



consumer 端用户线程在发出请求之前会先创建一个 `DefaultFuture` 对象；并将 `requestID` 作为 DefaultFuture 对象的 key 存储在 `Map FUTURES` 中（注意：每一个 `requestID` 是一个请求的唯一标识，最后相应的响应 Response 的 `responseID` 就等于这个 `requestID`）之后调用 Netty 编码并发出请求，然后马上调用 DefaultFuture#get 进行阻塞等待（阻塞等待 response 不为空）；

client将请求进一步转发给工作线程池的handler，hanlder封装了nettyclient，编码后通过调用nettyClient发起请求，并调用get函数将线程阻塞，等待应答。服务端收到请求后，进行解码，处理后返回。该过程是经过IO多路复用和事件驱动机制，是一个用来处理非阻塞型的可以应付高并发的线程模型，处理客户端的连接和读写请求，不断轮询多路请求的事件是否就绪，一旦就绪就通过事件驱动，回调结果。

这个过程具体就是服务端收到请求后，根据请求也构造出一个RPC请求结构rpcInvocation，然后根据请求构造一个服务标识serverKey，在DubboProtocol层取出之前服务暴露时，将服务的invoker和serverKey进行缓存的ExporterMap，根据serverKey取出服务暴露者exporter，因为exporter持有invoker，再取出invoker，然后对这个代理对象发起调用。代理会将执行转交给内部的一个Wrapper包装类，该包装类持有服务的真正实现，同时通过客户端的请求，请求中有目标的接口和方法名以及方法参数，通过这几个来参数匹配出真实服务并进行调用，并返回结果。

server 线程池处理完成之后，调用 Netty 编码并发送响应消息给 consumer 端。consumer 端接收到响应后，解码，然后交给 client 线程池处理，client 线程池从 `Map FUTURES` 中获取 `key=responseID` 的 `DefaultFuture` 对象，然后将响应消息填充到其 response 属性后，唤醒 consumer 端阻塞的用户线程。最后 consumer 得到了响应。

【总结】

服务调用

消费端发起调用时，会生成一个 Future 对象来接收返回结果，并生成一个 requestID ，然后将这个 ID 与 Future 对象绑定在一起存到 Map，意思是用 requestID 来唯一标识每次请求，并将请求结果放到对应的 Future 中。消费端的线程发起调用后就进入阻塞等待返回结果。

服务端收到请求后，根据请求的参数组成一个服务标识，以该标识为 key，取出对应的 exporter，因为之前服务暴露的时候就将服务信息封装在 exporter 并与服务标识绑定在一起缓存，所以这里只要取出 exporter，再取出其中的 invoker 对象，invoker 对象是一个代理，持有一个服务的真正实现，然后调用服务的实现并返回结果。

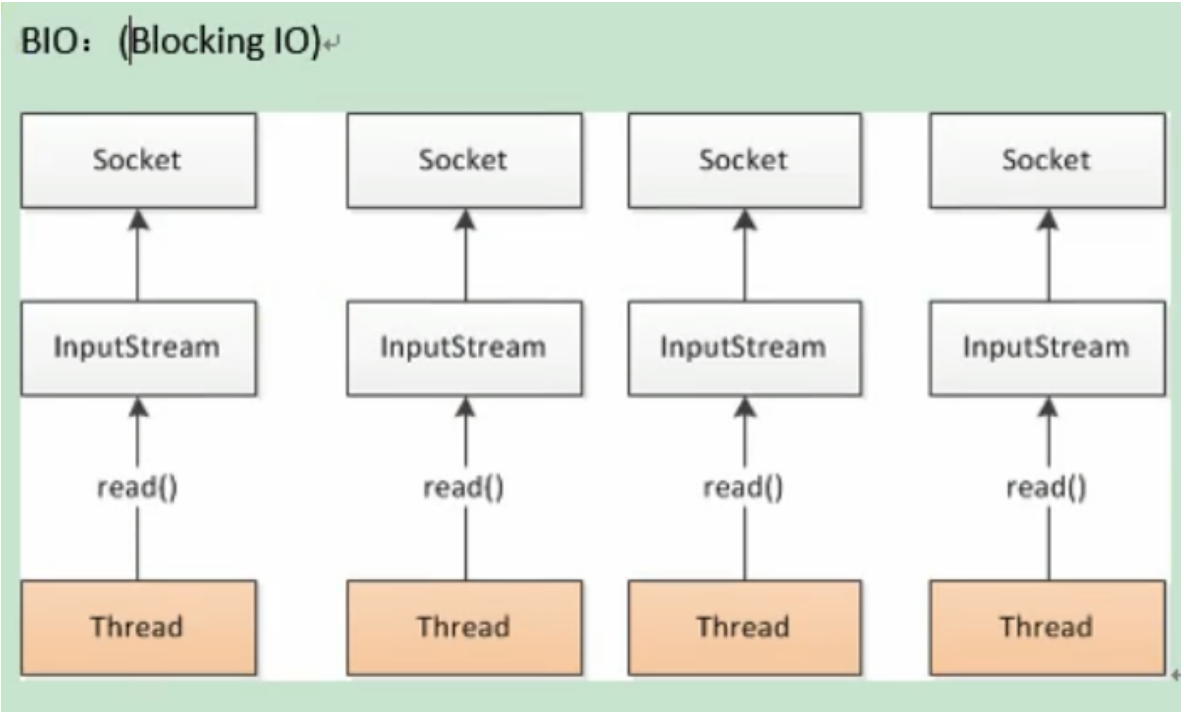
客户端收到响应后，根据 requestID 找到 Future，将结果填充进去，唤醒正在阻塞的用户线程，最终得到响应。

> **绑定 Future 与 requestID**
>
> **取出 exporter**
>
> **填充 Future，唤醒线程**

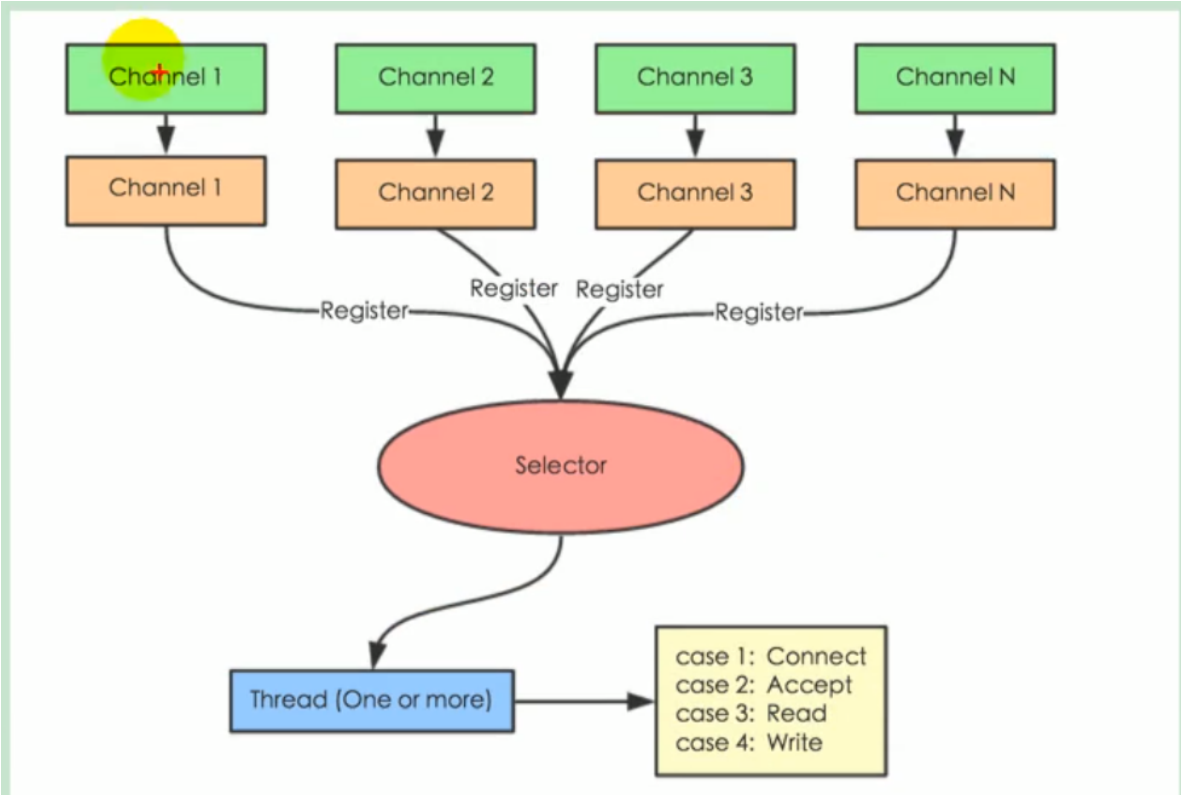
1）线程模型

【BIO】

每一个线程对应一个客户端请求，读取了客户端请求后，需要进行业务处理，此时线程需要等待业务处理完毕，再返回给客户端后，线程才可以释放，这样的话就无法处理高并发的情形，因为线程会阻塞，所以称为BIO，阻塞型IO。



【NIO】



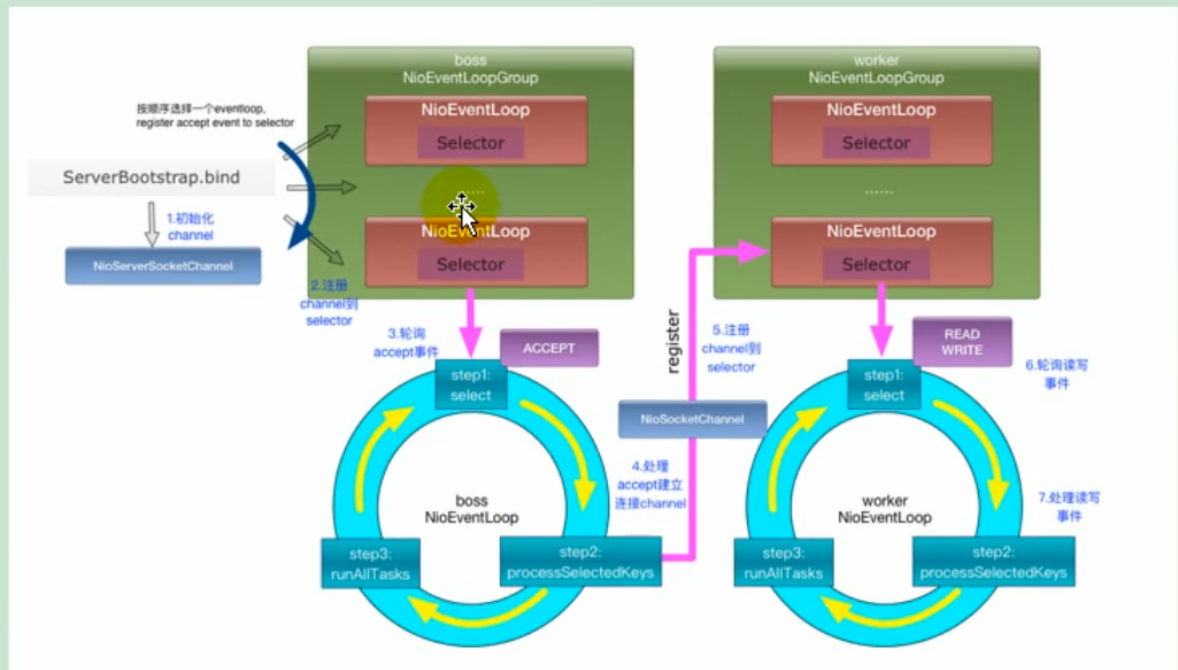
一个select可以注册很多个通道（请求），同时监听多个通道，当发现通道的数据准备好了，比如connect或者accept，再开一个线程轮询处理，将不同的事件交给特定的事件处理器去处理。

多个事件是被事件分发器分发到事件队列里面来维护的，而事件的具体处理可以由一个或多个线程来进行处理。

这样的话，只用一个线程，就可以监听多路请求，而且该线程无需阻塞，当当前通道的数据没有准备好，它会去轮询其他的通道，直到找到就绪的事件通道进行处理。这就是NIO非阻塞型IO。

【Netty】

整体架构是反应器模型，epoll 机制。



netty服务器启动，监听端口，比如dubbo的20880。所有给该端口发送的数据，netty服务器都能收到。

启动后，netty会初始化服务器的通道，就是多路复用模型里面的channel，每个通道对应一个客户端socket请求。初始化好了以后就会注册accpet事件到select即多路复用选择器中进行监听。

这个选择器负责轮询监听accept事件。 一旦有某个客户端的数据就绪，即准备好进行accept事件了，则netty就会读取客户端的数据，然后与客户端建立连接，生成另一个socket通道（NIOSocketChannel），在内核中即表现为accept完成后，给你返回一个新的文件描述符，表示accept已完成，连接已建立，可以进行数据通信。

所以对于多路复用选择器来说，这个通道与上一个通道虽然同属一个请求的发起者，但是从功能上讲，这里已经细分为两个请求，即两个通道，代表两种不同的事件，既然是事件，就都需要进行注册。

所以accept之后netty建立了新的通道，同样需要注册到select选择器中，只不过上次注册的是对于accept的监听，这次注册关于read和write的监听，netty就会轮询多个渠道监听这两个事件，一旦通道发生读写准备就绪事件，netty就会从通道进行数据的读写。

但无论是对于accpet还是write和read来说，具体的事件处理都不是直接由这个负责轮询的线程来执行的，这个线程只会轮询就绪事件产生，然后将对应事件生成一个任务，放到待执行的事件任务队列里面，由其他的线程即具体的事件处理器来负责事件的执行和处理。

但刚才讲到一个事件注册到选择器，线程轮询就绪事件，然后处理，这里如果多个事件只用一个选择器来维护，那么线程在轮询时开销会比较大，效率比较低，所以为了方便管理和提升轮询效率，netty将事件做一个划分，用不同类的选择器来维护。

在netty中，就划分为不同的线程组，boss线程组是主线程组，专门用来负责处理客户端的连接请求，比如accept，而worker线程组是工作线程组，用来负责连接之后数据的读写交互和业务事件的逻辑处理工作。这样通过不同的线程组就可以维护不同的选择器来维护不同的事件。

【Netty 实现】

基于非阻塞的多路复用与事件驱动模型，使用 epoll 机制。简单地说，就是通过建立一个网卡设备到用户进程的一个回调，当有数据到达网卡时，内核就会把就绪事件放到内核空间的一个就绪队列中。而用户进程在处理完连接后就会不断轮询这个队列是否有事件的到来。

在 Netty 中，epoll 这种机制就是多路复用IO，就是可以用一个线程去同时监听多个客户端的数据是否到达，如果当前客户端没有数据，则轮询下一个，不停地遍历这些客户端。所以 Netty 也是起一个扫描线程，不停监听就绪队列的事件，当有事件到达，就把事件取出，然后交给对应的事件处理器，每个事件处理器对应一个线程，这样的话扫描的那个线程就只需要负责把事件分发出去就行，然后继续轮询，提高了轮询的效率，这就是 Netty 的事件驱动机制。

而事件的类型比如有连接事件、读写事件等，需要有不同的线程去处理，而 Netty 进一步将这些线程进行划分，分为不同的线程组。其中主线程组负责客户端的连接事件，即专门处理客户端的连接；而另一个工作线程组用来负责读写事件等，这样区分开来便于维护，也间接地提升了性能。

> **非阻塞多路复用、事件驱动、epoll**
>
> **多路复用、两种线程**
>
> **两种线程组**

8、手写dubbo

- Provider模块：提供API、实现API、暴露(启动tomcat,nettyServer)、服务本地注册、服务注册中心注册
- Consumer模块：拿接口名从注册中心获取服务地址、调用服务
- Registry模块：保存服务配置信息（服务名：List<URL>）
- RpcProtocol模块：基于Tomcat的HttpProtocol、基于Netty的DubboProtocol
- Framework模块：框架实现

```java
import framework.URL;
import protocol.http.HttpServer;
import provider.api.HelloService;
import provider.impl.HelloServiceImpl;
import register.RemoteMapRegister;

/**
 * **************书山有路勤为径***************
 * 鲁班学院
 * 往期资料加木兰老师  QQ: 2746251334
 * VIP课程加安其拉老师 QQ: 3164703201
 * 讲师: 周瑜老师
 * **************学海无涯苦作舟***************
 */
public class Provider {

    public static void main(String[] args) {
        // 1. 本地注册
        // {服务名: 实现类}
        LocalRegister.regist(HelloService.class.getName(), HelloServiceImpl.class);


        // 2. 远程注册
        // {服务名: List<URL>}
        URL url = new URL( hostname: "localhost",  port: 8080);
        RemoteMapRegister.register(HelloService.class.getName(), url);


        // 3. 启动tomcat
        HttpServer httpServer = new HttpServer();
        httpServer.start( hostname: "localhost",  port: 8080);

    }
}
```

```java
import java.lang.reflect.Method;

/**
 * **************书山有路勤为径***************
 * 鲁班学院
 * 往期资料加木兰老师  QQ: 2746251334
 * VIP课程加安其拉老师 QQ: 3164703201
 * 讲师: 周瑜老师
 * **************学海无涯苦作舟***************
 */
public class HttpServerHandler {

    public void handler(HttpServletRequest req, HttpServletResponse resp) {
        // 处理请求, 返回结果

        try {
            InputStream inputStream = req.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(inputStream);

            Invocation invocation = (Invocation) ois.readObject();

            Class implClass = LocalRegister.get(invocation.getInterfaceName());

            Method method = implClass.getMethod(invocation.getMethodName(), invocation.getParamTypes());

            String result = (String) method.invoke(implClass.newInstance(), invocation.getParams());


            IOUtils.write(result, resp.getOutputStream());


        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
```

```java
package consumer;

import framework.Invocation;
import protocol.http.HttpClient;
import provider.api.HelloService;

/**
 * **************书山有路勤为径***************
 * 鲁班学院
 * 往期资料加木兰老师  QQ: 2746251334
 * VIP课程加安其拉老师 QQ: 3164703201
 * 讲师: 周瑜老师
 * **************学海无涯苦作舟***************
 */
public class Consumer {

    public static void main(String[] args) {


        HttpClient httpClient = new HttpClient();
        Invocation invocation = new Invocation(HelloService.class.getName(),  methodName: "sayHello", new Class[]{St
        String result = httpClient.send( hostname: "localhost",  port: 8080, invocation);
        System.out.println(result);



    }
}
```

```java
public class HttpClient {

    public String send(String hostname, Integer port, Invocation invocation) {

        try {
            URL url = new URL( protocol: "http", hostname, port,  file: "/");
            HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();

            httpURLConnection.setRequestMethod("POST");
            httpURLConnection.setDoOutput(true);

            OutputStream outputStream = httpURLConnection.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(outputStream);

            oos.writeObject(invocation);
            oos.flush();
            oos.close();

            InputStream inputStream = httpURLConnection.getInputStream();
            String result = IOUtils.toString(inputStream);

            return result;

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;

    }
}
```

客户端基于java的动态代理

```
main
  java
    consumer
      Consumer
    framework
      Invocation
      ProxyFactory
      URL
    protocol
      dubbo
      http
        Dispatche
        HttpClient
          send(S
        HttpServe
```

```java
* 讲师: 周瑜老师
* **************学海无涯苦作舟****************
*/
public class ProxyFactory {

    public static <T> T getProxy(final Class interfaceClass) {

        return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Class[]{interfaceClass}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                HttpClient httpClient = new HttpClient();
                Invocation invocation = new Invocation(interfaceClass.getName(), method.getName(), method.getParameterTypes(), args);
                String result = httpClient.send( hostname: "localhost",  port: 8080, invocation);

                return result;
            }
        });

    }
}
```

```java
public class Consumer {

    public static void main(String[] args) {

        HelloService helloService = ProxyFactory.getProxy(HelloService.class);

        System.out.println(helloService.sayHello( username: "123123123"));



    }
}
```

类比dubbo的写法：

其实 demoService 这个 spring bean 本质上也是一个代理对象。

```java
lic class Application {
/**
 * In order to make sure multicast registry works, need to specify '-Djava.net.preferIPv4Stack=true' before
 * launch the application
 */
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "spring/dubbo-consumer.xml");
    context.start();
    DemoService demoService = context.getBean( name: "demoService", DemoService.class);
    String hello = demoService.sayHello( name: "world");
    System.out.println("result: " + hello);
}
```

模拟远程注册中心

```
provider
  api
    HelloService
  impl
    HelloService
  LocalRegister
  Provider
  register
    RemoteMapReg
  resources
test
target
tomcat.8080
pom.xml
```

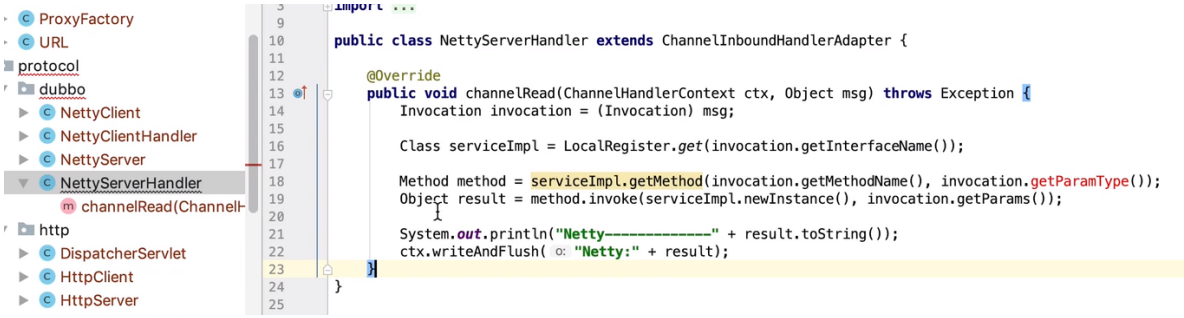
```java
*/
public class RemoteMapRegister {

    private static Map<String, List<URL>> REGISTER = new HashMap<>();

    public static void register(String interfaceName, URL url) {
        List<URL> list = Collections.singletonList(url);
        REGISTER.put(interfaceName, list);
    }


    public static URL random(String interfaceName) {

        List<URL> list = REGISTER.get(interfaceName);
        Random random =new Random();
        int n = random.nextInt(list.size());
        return list.get(n);
    }
}
```

模拟通过随机的负载均衡策略，从远程注册中心中取得服务的地址，

```java
public class ProxyFactory {

    public static <T> T getProxy(final Class interfaceClass) {

        return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Class[]{interfaceClass}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                HttpClient httpClient = new HttpClient();
                Invocation invocation = new Invocation(interfaceClass.getName(), method.getName(), method.getParameterTypes(), args);

                URL url = RemoteMapRegister.random(interfaceClass.getName());
                String result = httpClient.send(url.getHostname(), url.getPort(), invocation);

                return result;
            }
        });
```

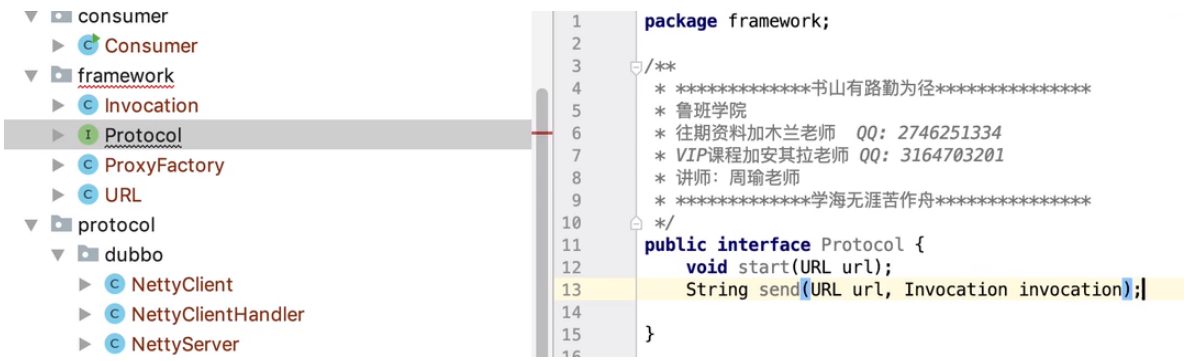
再将地址的自动获取封装到代理层，服务的消费方调用时，只需要设置负载均衡策略即可，无需关心具体实现，所以代理层又多了负载均衡的功能。

实现protocol协议层的dubbo协议



dubbo协议底层通信主要采用了netty框架来进行，核心是IO多路复用和事件驱动机制，每个请求通过一个channel来发送和接收响应结果。

协议选择抽象接口



实现

```
*/
public class DubboProtocol implements Protocol {

    @Override
    public void start(URL url) {
        new NettyServer().start(url.getHostname(), url.getPort());
    }

    @Override
    public String send(URL url, Invocation invocation) {
        return new NettyClient<>().send(url.getHostname(), url.getPort(), invocation);
    }
}
```

```
public class HttpProtocol implements Protocol {

    @Override
    public void start(URL url) { new HttpServer().start(url.getHostname(), url.getPort()); }

    @Override
    public String send(URL url, Invocation invocation) {
        return new HttpClient().send(url.getHostname(), url.getPort(), invocation);
    }
}
```

工厂模式

通过系统属性的设置来获取对应的实现类

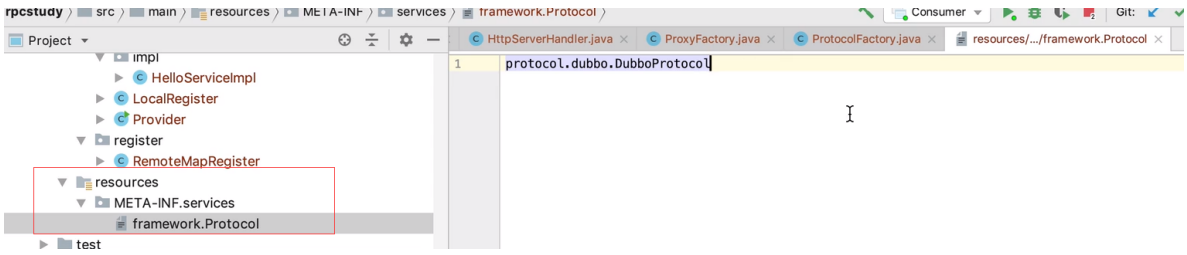
```java
public class ProtocolFactory {

    public static Protocol getProtocol() {

        // 工厂模式
        String name = System.getProperty("protocolName");
        if (name == null || name.equals("")) name = "http";
        switch (name) {
            case "http":
                return new HttpProtocol();
            case "dubbo":
                return new DubboProtocol();
            default:
                break;
        }
        return new HttpProtocol();
    }
}
```

再把获取实现类（执行者）的逻辑封装到代理层

```java
public class ProxyFactory {

    public static <T> T getProxy(final Class interfaceClass) {

        return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Class[]{interfaceClass}, ne
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                Protocol protocol = ProtocolFactory.getProtocol();
                Invocation invocation = new Invocation(interfaceClass.getName(), method.getName(), metho

                URL url = RemoteMapRegister.random(interfaceClass.getName());
                String result = protocol.send(url, invocation);

                return result;
            }
        });

    }
```

Java SPI

```java
public class ProtocolFactory {

    public static Protocol getProtocol() {

        // java spi
        ServiceLoader<Protocol> serviceLoader = ServiceLoader.load(Protocol.class);
        Iterator<Protocol> iterator = serviceLoader.iterator();
        return iterator.next();
```



基于ServiceLoader从Protocol接口中获取指定的实现类，这个实现类被定义到了资源文件目录下的一个文件，该文件名需要和接口的全限定名称一致，文件的内容就是接口实现类的全限定名称。

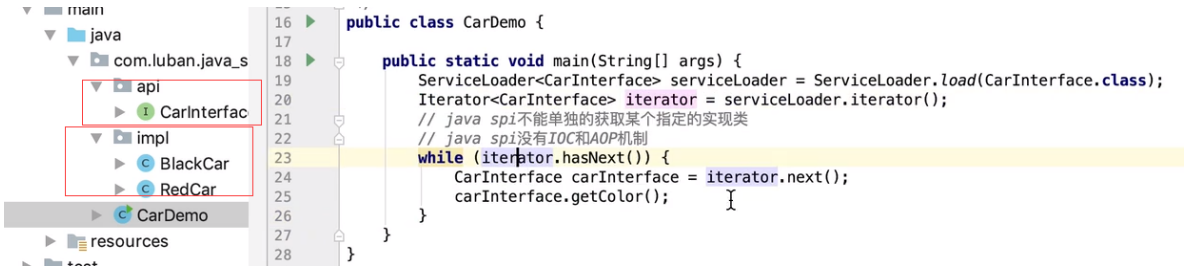
mysql驱动类JDBC需要加载，它是面向接口编程，实现了Driver这个驱动接口，然后要选择具体的驱动实现类，也是通过这种java spi机制来选择，通过在资源目录文件下指定实现类的路径的方式，比如可以是mysql或者sqlite。

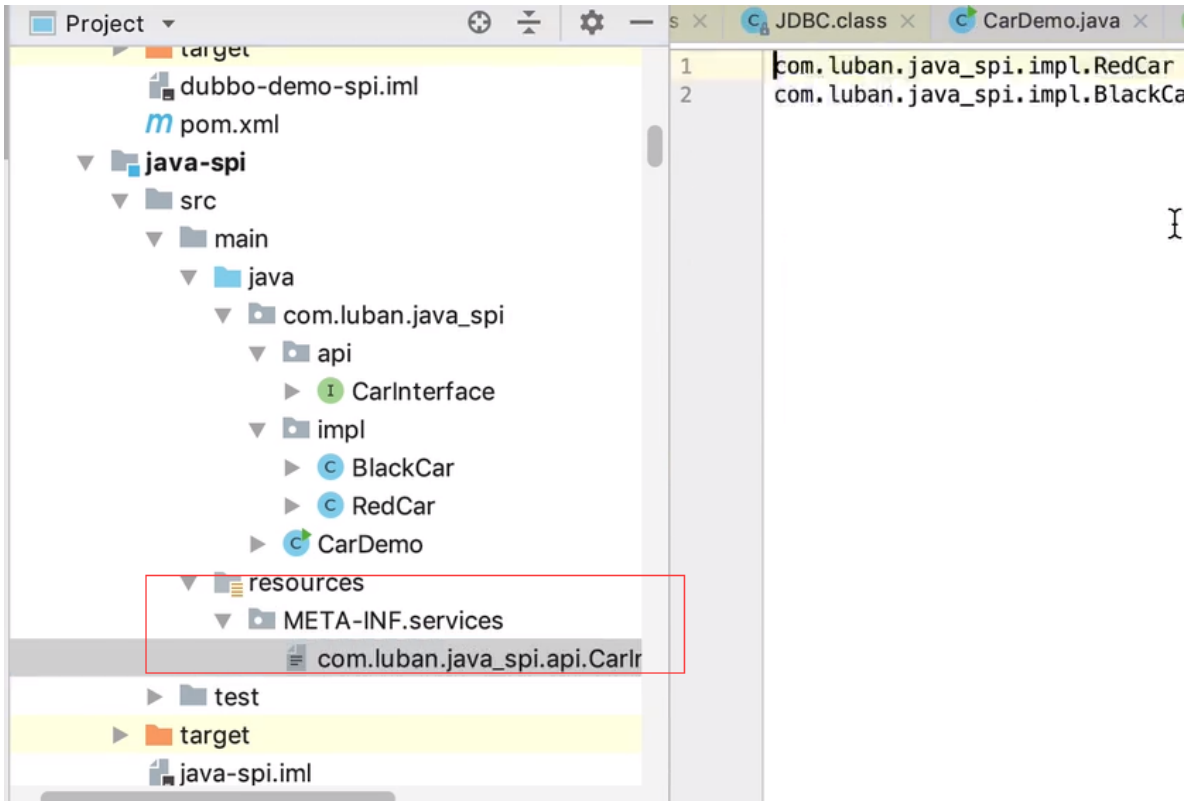
Java SPI 存在的问题

1）无法通过key-value的形式，在配置文件中体现，从而在获取接口的时候，不用通过遍历判断的方式，而是通过取相应key的value的形式来获取指定的接口的实现类。

2）无法解决依赖注入的问题。

3）无法实现AOP功能。





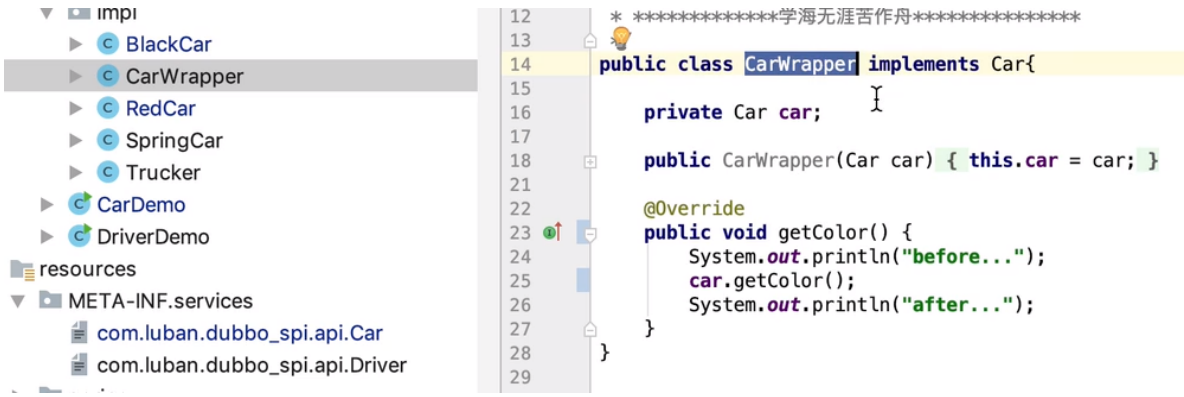
Dubbo SPI 应用

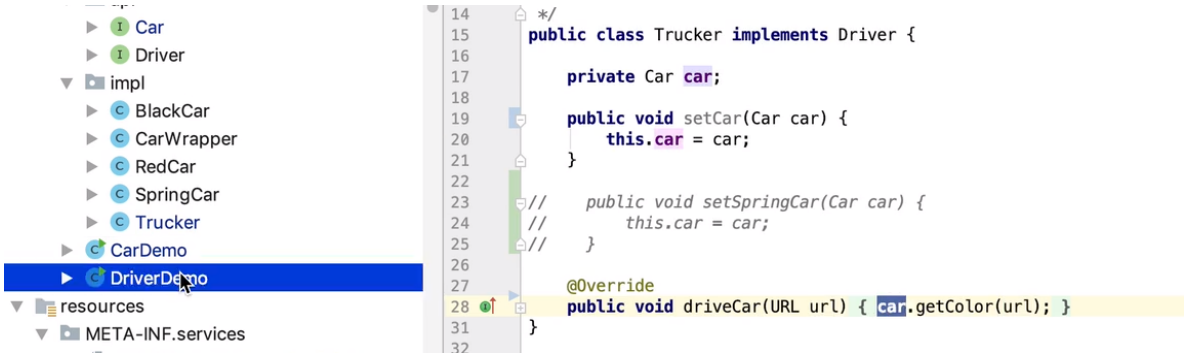
- 协议层协议的选择
- 注册中心的选择

Java SPI 原理

- 解析配置文件，从文件中遍历取出实现类全限定名
- 遍历上诉类全限定名，获取类对象（.class）
- 由类对象生成实例（class.newInstance()）

实现AOP





在dubbo中，是通过URL来传递接口信息的，接口信息中，就会包含扩展实现类的具体实现对象，通过@Adative注解来指定一个key，通过在配置文件中配置key-value的形式，那么dubbo就会根据这个key去取得对应的value，也即实现类的全限定名，然后反射出一个实现类的实例，以此来确定接口的执行者。

getExtensionLoader()

从缓存（Map）中，根据传入的接口class文件，获取对应的ExtensionLoader对象。

ExtensionLoader.getAdaptiveExtension()是为了获得一个代理类。

ExtensionLoader.getExtension()是为了根据全限定类名获得一个类的实例，并进行缓存。而全限定类名的获得，是根据函数的入参比如name，这个比如可以是配置文件的key，然后根据这个key读取配置文件的value，即是全限定的类名。

getExtension方法还支持获取一个默认的实现类，根据入参name为字符串true。这种实现需要在SPI注解上声明默认的实现类的key值，以便在解析配置文件时，可以根据这个key来获取对应的value，即获得一个全限定类名，再根据这个类名反射得到一个实例对象。

```java
/unchecked/
public T getExtension(String name) {
    if (StringUtils.isEmpty(name)) {
        throw new IllegalArgumentException("Extension name == null");
    }
    if ("true".equals(name)) {
        return getDefaultExtension();
    }
    Holder<Object> holder = getOrCreateHolder(name);
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                instance = createExtension(name);
                holder.set(instance);
            }
        }
    }
    return (T) instance;
```

```java
*/
@SPI(value = "black")
public interface Car {

    @Adaptive(value = "carType")
    public void getColor(URL url);
}
```

```java
*/
public class CarDemo {

    public static void main(String[] args) {
        // 每个接口对应一个ExtensionLoader
        ExtensionLoader<Car> extensionLoader = ExtensionLoader.getExtensionLoader(Car.class);
        Car redCar = extensionLoader.getExtension( name: "true");
        redCar.getColor( url: null);

//        Car blackCar = extensionLoader.getExtension("black");
//        blackCar.getColor();
    }
}
```

createExtension获得实现类的实例，包含ioc和aop

```java
/unchecked/
// 返回name所对应的实现类的实例
// 包含依赖注入和AOP
private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (CollectionUtils.isNotEmpty(wrapperClasses)) {
            for (Class<?> wrapperClass : wrapperClasses) {
                instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    } catch (Throwable t) {
```

获取扩展类（包括封装类对象）

解析配置文件并缓存到map中，key是全限定类名，value是对应的class

```java
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get();
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

loadExtensionClasses()

包括解析配置文件，根据全限定类名反射得到类的实例对象.class对象；以及获取实现类的封装类，即AOP的执行者。

```java
// synchronized in getExtensionClasses
private Map<String, Class<?>> loadExtensionClasses() {
    cacheDefaultExtensionName();

    Map<String, Class<?>> extensionClasses = new HashMap<>();
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName().replace( target: "org.apache",  replacement: "com
    loadDirectory(extensionClasses, DUBBO_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, DUBBO_DIRECTORY, type.getName().replace( target: "org.apache",  replacement: "com.alibaba"
    loadDirectory(extensionClasses, SERVICES_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, SERVICES_DIRECTORY, type.getName().replace( target: "org.apache",  replacement: "com.alibal
    return extensionClasses;
}
```

```java
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL resourceURL, Class<?> clazz, String name) throc
    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException("Error occurred when loading extension class (interface: " +
                type + ", class line: " + clazz.getName() + "), class "  type: "interface com.luban.dubbo_spi.api.Car"
                + clazz.getName() + " is not subtype of interface.");
    }
    if (clazz.isAnnotationPresent(Adaptive.class)) {  clazz: "class com.luban.dubbo_spi.impl.RedCar"
        cacheAdaptiveClass(clazz);
    } else if (isWrapperClass(clazz)) {
        cacheWrapperClass(clazz);
    } else {
        clazz.getConstructor();
        if (StringUtils.isEmpty(name)) {
            name = findAnnotationName(clazz);
            if (name.length() == 0) {
                throw new IllegalStateException("No such extension name for the class " + clazz.getName() + " in the confi
            }
        }

        String[] names = NAME_SEPARATOR.split(name);
        if (ArrayUtils.isNotEmpty(names)) {
            cacheActivateClass(clazz, names[0]);
```

配置文件中，除了可以配置有实现类之外，还可以配置实现类的封装类，即AOP执行对象。而解析的过程，有几种，除了可以直接解析配置文件得到value外，也可以解析由@Adaptive注解修饰的方法，区别是优先级。如果是由@Adaptive注解修饰的类，则为一个代理实现类。

```java
*/
private boolean isWrapperClass(Class<?> clazz) {
    try {
        clazz.getConstructor(type);
        return true;
    } catch (NoSuchMethodException e) {
        return false;
    }
}
```

通过判断是否有依赖注入对象的构造函数来判断是否为封装类，即是否为可以实现AOP功能的执行对象。

injectExtension()实现以来注入

```java
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            for (Method method : instance.getClass().getMethods()) {
                if (isSetter(method)) {
                    /**
                     * Check {@link DisableInject} to see if we need auto injection for this property
                     */
                    if (method.getAnnotation(DisableInject.class) != null) {
                        continue;
                    }
                    Class<?> pt = method.getParameterTypes()[0];
                    if (ReflectUtils.isPrimitives(pt)) {
                        continue;
                    }
                    try {
                        String property = getSetterProperty(method);
                        Object object = objectFactory.getExtension(pt, property);
                        if (object != null) {
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
```

getAdaptiveExtension()

```java
ExtensionLoader.java  Trucker.java  Extension.java  StringUtils.java  CarWrapper.java  BlackCar.java  Adaptive.java
cachedWrapperClasses   Match Case  Words  Regex  ?  5 matches
92      private volatile Class<?> cachedAdaptiveClass = null;
93      private String cachedDefaultName;
94      private volatile Throwable createAdaptiveInstanceError;
95
96      private Set<Class<?>> cachedWrapperClasses;
97
98      private Map<String, IllegalStateException> exceptions = new ConcurrentHashMap<>();
99
100     private ExtensionLoader(Class<?> type) {
101         this.type = type;
102         // ExtensionFactory表示扩展机制的工厂，在Dubbo里面有SPI扩展机制，也有Spring扩展机制
103         // 对于一个接口，比如Car接口，有两种实现类，一种就是我们自定义的实现类，比如RedCar，还有一种就是代理类，对于代理类，可以由我们自己实现，
104         objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.getExtensionLoader(ExtensionFactory.class).ge
105     }
106
107 @   private static <T> boolean withExtensionAnnotation(Class<T> type) { return type.isAnnotationPresent(SPI.class); }
110
111     /unchecked/
112 @   public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
113         if (type == null) {
114             throw new IllegalArgumentException("Extension type == null");
115         }
```

```java
/unchecked/
public T getAdaptiveExtension() {
    Object instance = cachedAdaptiveInstance.get();
    if (instance == null) {
        if (createAdaptiveInstanceError == null) {
            synchronized (cachedAdaptiveInstance) {
                instance = cachedAdaptiveInstance.get();
                if (instance == null) {
                    try {
                        instance = createAdaptiveExtension();
                        cachedAdaptiveInstance.set(instance);
                    } catch (Throwable t) {
                        createAdaptiveInstanceError = t;
                        throw new IllegalStateException("Failed to create adaptive instance: " + t.toString(), t);
                    }
                }
            }
        } else {
            throw new IllegalStateException("Failed to create adaptive instance: " + createAdaptiveInstanceError.toString(
        }
```

createAdaptiveExtension()

```java
private T createAdaptiveExtension() {
    try {
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can't create adaptive extension " + type + ", cause: " + e.getMessage(), e);
    }
}

private Class<?> getAdaptiveExtensionClass() {
    getExtensionClasses();
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }
    return cachedAdaptiveClass = createAdaptiveExtensionClass();
}

private Class<?> createAdaptiveExtensionClass() {
    String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();
    ClassLoader classLoader = findClassLoader();
    org.apache.dubbo.common.compiler.Compiler compiler = ExtensionLoader.getExtensionLoader(org.apache.dubbo.common.compil
    return compiler.compile(code, classLoader);
}
```

getExtensionClasses()方法，即从配置文件中解析出实现类，并且如果有用@Adaptivetion修饰的类，则为实现类的一个代理类。

```java
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get();
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

getAdaptiveExtensionClass()方法，如果cachedAdaptoveClass不为空，即我们自己实现的代理类；如果为空，即没有手动实现也没有@Adaptivetion注解，则Dubbo就会自动生成一个代理类，构造代码，编译一个class对象。接着以来注入就是寻找set方法，再反射注入。

9、其他总结

【服务暴露】

RegistryProtocol首先执行doLocalExport进行本地方法暴露，然后再进行服务的注册信息的注册，通过registerProvider方法将服务的信息注册到一张ProviderConsumerRegTable提供者消费者注册表中。

首先暴露本地服务，获取要暴露的执行者invoker，然后通过dubboProtocol进行进一步暴露，首先封装一个暴露器，进行属性的设置，然后执行openServer方法，首先获取服务的地址，然后创建一个ExchangeServer服务器，调用bind方法绑定端口和一个请求的处理器，再往底层走就是调用netty的方法创建一个Netty服务器。

所以在这里，暴露本地服务，首先dubboProtocol通过创建一个服务器，其实就是启动一个netty服务器，监听20880端口。

然后就是注册提供者。RegistryProtocol通过ProviderConsume这个类的注册方法来实现。这个类维护了两个本地的ConcurrentHashMap，分别是服务提供的执行者和服务消费的提供者，保存了每个url地址（key）对应的服务提供者和消费者的执行器（value），而执行器才有我们服务的具体实现，接口的实现类。然后再将服务的地址通过zookeeper客户端注册到zookeeper中完成整个服务的暴露。

至此，服务就暴露出来了。这里有两步，首先dubboProtocol暴露本地服务，会启动netty的服务器来监听20880，然后RegistryProtocol将服务的地址和服务的实现绑定在一起保存在本地，之后再注册到zookeeper。

所以消费端发起远程调用时，首先会通过zookeeper查找服务的地址。然后发起调用，生产端dubbo在启动后会将服务在容器中准备好，通过监听bean生命周期的一些回调事件比如属性设置完，对象初始化后开始做配置文件的解析，然后开始暴露服务。

暴露过程是通过启动netty服务监听20880端口，接收消费端的RPC请求，然后找到上述的提供者消费者注册表，通过服务的地址，取出服务的具体执行者即具体实现，然后完成调用，最终返回结果。

【服务引用】

首先服务的注入是一个工厂bean，需要调用getObject方法获取，接着进行初始化，解析配置文件。然后调用createProxy根据配置文件信息来创建一个代理对象。

进入DubboProtocol，执行refer方法，首先通过getClients获取客户端。获取客户端的具体做法是，创建并初始化一个连接客户端，连接到服务地址，本质上是创建一个netty客户端，连接到netty服务器，进行通信。

RegistryProtocol注册中心的refer方法，根据注册中心的地址得到注册中心。然后执行doRefer方法，开始去注册中心订阅服务提供者提供的服务。

订阅完成后，再将该invoker和订阅的地址注册到提供者消费者注册表中。消费者注册表中记录的是消费者地址对应的是哪一个代理，即invoker。

然后返回invoker执行者，这个执行者就包含了客户端的连接属性，同时这个invoker也封装了远程服务的调用地址。

所以，createProxy方法其实就是根据配置创建了一个代理对象，这个对象的核心包含一个可以和远程连接的客户端，还包含了远程服务的调用信息。

【服务调用】

引用服务创建好了代理对象，代理开始进行服务的调用。此时消费者注入使用的bean就是一个代理对象，代理对象层层封装了各种invoker，其中就包含了服务的地址和要执行的方法信息。

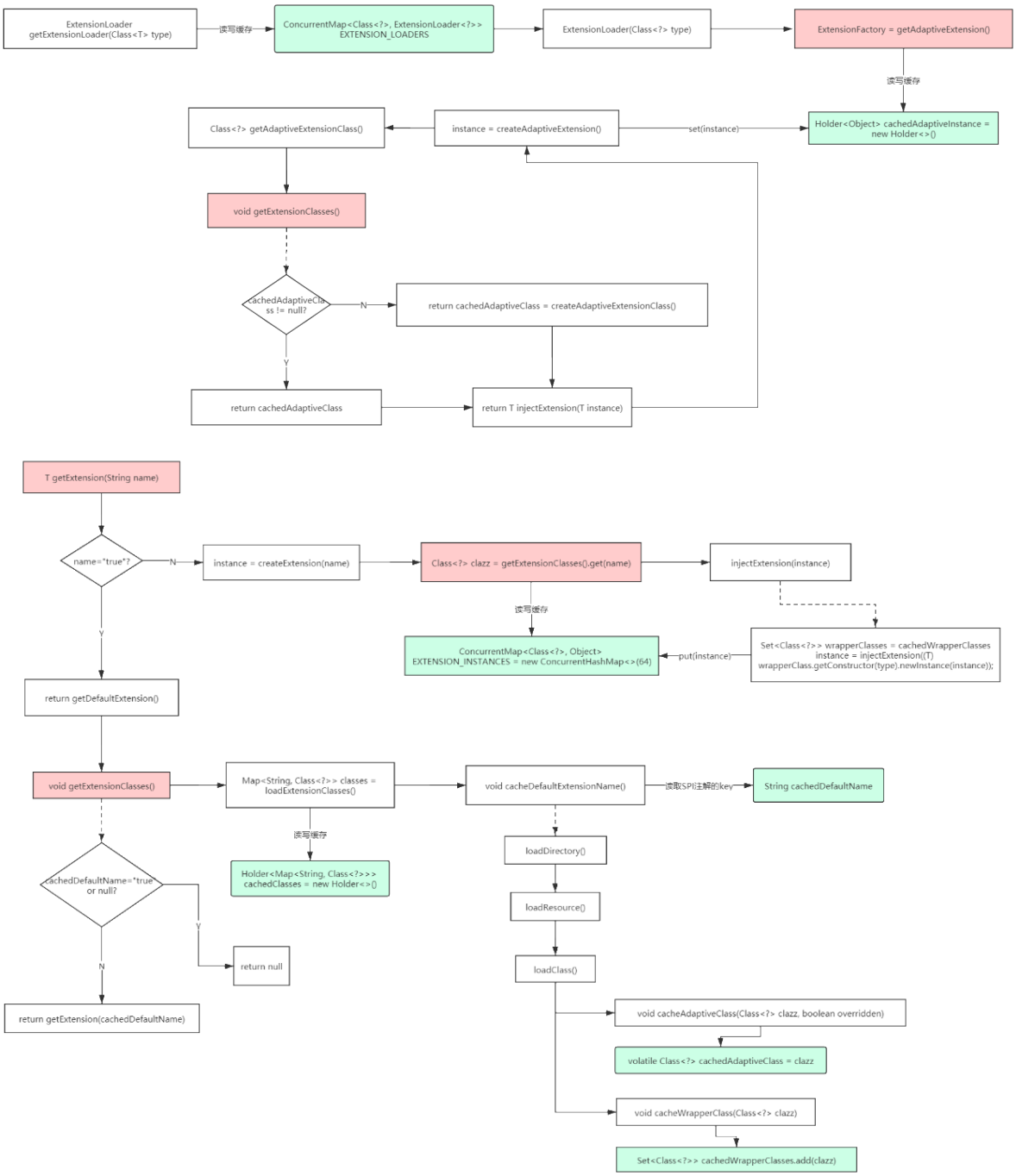
首先会获取方法的信息，比如方法名称和方法参数等，然后将这些具体的调用信息封装成一个RpcInvocation RPC执行对象。然后执行一系列filter的过滤。

然后执行list方法，根据RPC执行对象，从注册中心中找出有几个可以执行的invoker对象，实际上就是找出有几个服务的具体实现。然后根据预设的配置获取负载均衡策略，根据负载均衡策略获取具体的invoker。然后再执行一系列filter的过滤。

最后进入DubboInvoker中，根据RPC执行对象，获取方法的执行信息，比如方法名称和方法参数等。再获取一个客户端，这个客户端在服务的引用时创建过，该对象是用来和dubbo服务的提供者建立通信连接的，保存了服务的连接信息。再通过客户端的request方法发起请求，进入底层通信，最后通过netty客户端，把请求发到服务端的netty通道中，进入多路复用，服务端处理完再返回结果。

消费端拿到响应结果后，进行解码。

【Dubbo SPI】



【getExtensionLoader(Class type)】

getExtensionLoader方法 这是一个静态工厂方法，入参是一个可扩展的接口，返回一个该接口的ExtensionLoader实体类。通过这个实体类，可以根据name获得具体的扩展，也可以获得一个自适应扩展。

【变量】

// ExtensionLoader类加载器加载时
// 1.缓存接口类与对应的扩展加载器的映射
ConcurrentMap<Class, ExtensionLoader> EXTENSION_LOADERS = new ConcurrentHashMap<>(64)
// 2.缓存@Adaptive注解修饰的代理实现类，装载在扩展工厂中 ExtensionFactory
Holder cachedAdaptiveInstance = new Holder<>()

【调用链】

1.ExtensionLoader getExtensionLoader(Class type) 获取类扩展器
1）缓存有，取出扩展加载器
2）缓存没有，新建扩展加载器置入

2.ExtensionLoader(Class<?> type) 类扩展器构造函数
通过getAdaptiveExtension创建扩展工厂（ExtensionFactory = getAdaptiveExtension()）

【ExtensionFactory 有四个实现】

- SpiExtensionFactory：Dubbo自己的Spi去加载Extension
- SpringExtensionFactory：从Spring容器中去加载Extension
- AdaptiveExtensionFactory：自适应的AdaptiveExtensionLoader
- MyExtensionFactory：自定义扩展工厂

3.getAdaptiveExtension() 获取自适应扩展器
1）解析配置文件和注解，获取接口对应的代理实现类

【配置文件】

- META-INF/dubbo/internal
- META-INF/dubbo
- META-INF/services

2）如果没有配置文件和注解，则dubbo自动生成一个代理实现类

【原理】

- Dubbo需要在运行时根据方法参数来决定该使用哪个扩展，所以有了扩展点自适应实例。
- 其实是一个扩展点的代理，将扩展的选择从Dubbo启动时，延迟到RPC调用时。
- Dubbo中每一个扩展点都有一个自适应类，如果没有显式提供，Dubbo会自动为我们创建一个，默认使用Javaassist。

AdaptiveExtensionLoader类有@Adaptive注解。Dubbo会为每一个扩展创建一个自适应实例。如果扩展类上有@Adaptive，会使用该类作为自适应类。如果没有，Dubbo会为我们创建一个。

所以ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension())会返回一个AdaptiveExtensionLoader实例，作为自适应扩展实例。 AdaptiveExtensionLoader会遍历所有的ExtensionFactory实现，尝试着去加载扩展。如果找到了，返回。如果没有，在下一个ExtensionFactory中继续找。

Dubbo内置了两个ExtensionFactory，分别从Dubbo自身的扩展机制和Spring容器中去寻找。由于ExtensionFactory本身也是一个扩展点，我们可以实现自己的ExtensionFactory，让Dubbo的自动装配支持我们自定义的组件。比如，我们在项目中使用了Google的guice这个 IOC 容器。我们可以实现自己的GuiceExtensionFactory，让Dubbo支持从guice容器中加载扩展。

【getExtension(String name)】

【变量】

// 缓存代理实现类和对应实例的映射（实例包含了实现了ioc和aop功能，即完成了依赖注入和封装类的包装）
ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES = new ConcurrentHashMap<>(64)
// 获取接口代理实现类的默认key
String cachedDefaultName
// 缓存代理实现类key和对应的通过配置和注解解析出来的代理实现类
Holder<Map<String, Class<?>>> cachedClasses = new Holder<>()

// 缓存@Adaptive注解解析出来的代理实现类
volatile Class<?> cachedAdaptiveClass = clazz
// 缓存Aop执行者，即封装类
Set<Class<?>> cachedWrapperClasses.add(clazz)

【依赖注入】
Dubbo的方案是查找Java标准的setter方法。即方法名以set开始，只有一个参数。如果扩展类中有这样的set方法，Dubbo会对其进行依赖注入，类似于Spring的set方法注入。

【AOP】

在Dubbo中，有一种特殊的类，被称为Wrapper类。通过装饰者模式，使用包装类包装原始的扩展点实例。在原始扩展点实现前后插入其他逻辑，实现AOP功能。
在Dubbo中Wrapper类也是一个扩展点，和其他的扩展点一样，也是在META-INF文件夹中配置的

```
private T createExtension(String name) {
    // 从配置文件中加载所有的拓展类，可得到“配置项名称”到“配置类”的映射关系表
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            // 通过反射创建实例
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        // 向实例中注入依赖
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
            // 循环创建 Wrapper 实例
            for (Class<?> wrapperClass : wrapperClasses) {
                // 将当前 instance 作为参数传给 Wrapper 的构造方法，并通过反射创建
Wrapper 实例。
                // 然后向 Wrapper 实例中注入依赖，最后将 Wrapper 实例再次赋值给
instance 变量
                instance = injectExtension(
                    (T)
wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException("...");
    }
}
```

【调用链】

getExtension(String name)
1.name不是"true",
1）getExtensionClasses()
解析配置文件和注解，获取接口对应的代理实现类
2）injectExtension(instance)
完成依赖注入和封装类的包装，缓存到EXTENSION_INSTANCES

2.name是"true"

1）getExtensionClasses()

解析配置文件和注解，获取接口对应的代理实现类

2）getDefaultExtension()

获取默认扩展器

【主要方法】

```
private Map<String, Class<?>> loadExtensionClasses() {
    // 获取 SPI 注解，这里的 type 变量是在调用 getExtensionLoader 方法时传入的
    final SPI defaultAnnotation = type.getAnnotation(SPI.class);
    if (defaultAnnotation != null) {
        String value = defaultAnnotation.value();
        if ((value = value.trim()).length() > 0) {
            // 对 SPI 注解内容进行切分
            String[] names = NAME_SEPARATOR.split(value);
            // 检测 SPI 注解内容是否合法，不合法则抛出异常
            if (names.length > 1) {
                throw new IllegalStateException("more than 1 default extension
name on extension...");
            }

            // 设置默认名称，参考 getDefaultExtension 方法
            if (names.length == 1) {
                cachedDefaultName = names[0];
            }
        }
    }

    Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
    // 加载指定文件夹下的配置文件
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
    loadDirectory(extensionClasses, DUBBO_DIRECTORY);
    loadDirectory(extensionClasses, SERVICES_DIRECTORY);
    return extensionClasses;
}
```

loadExtensionClasses 方法总共做了两件事情，一是对 SPI 注解进行解析，二是调用 loadDirectory 方法加载指定文件夹配置文件。

```
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL
resourceURL,
    Class<?> clazz, String name) throws NoSuchMethodException {

    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException("...");
    }

    // 检测目标类上是否有 Adaptive 注解
    if (clazz.isAnnotationPresent(Adaptive.class)) {
        if (cachedAdaptiveClass == null) {
            // 设置 cachedAdaptiveClass缓存
            cachedAdaptiveClass = clazz;
        } else if (!cachedAdaptiveClass.equals(clazz)) {
            throw new IllegalStateException("...");
        }
```

```
    // 检测 clazz 是否是 Wrapper 类型
    } else if (isWrapperClass(clazz)) {
        Set<Class<?>> wrappers = cachedWrapperClasses;
        if (wrappers == null) {
            cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
            wrappers = cachedWrapperClasses;
        }
        // 存储 clazz 到 cachedWrapperClasses 缓存中
        wrappers.add(clazz);

    // 程序进入此分支，表明 clazz 是一个普通的拓展类
    } else {
        // 检测 clazz 是否有默认的构造方法，如果没有，则抛出异常
        clazz.getConstructor();
        if (name == null || name.length() == 0) {
            // 如果 name 为空，则尝试从 Extension 注解中获取 name，或使用小写的类名作为
name
            name = findAnnotationName(clazz);
            if (name.length() == 0) {
                throw new IllegalStateException("...");
            }
        }
        // 切分 name
        String[] names = NAME_SEPARATOR.split(name);
        if (names != null && names.length > 0) {
            Activate activate = clazz.getAnnotation(Activate.class);
            if (activate != null) {
                // 如果类上有 Activate 注解，则使用 names 数组的第一个元素作为键，
                // 存储 name 到 Activate 注解对象的映射关系
                cachedActivates.put(names[0], activate);
            }
            for (String n : names) {
                if (!cachedNames.containsKey(clazz)) {
                    // 存储 Class 到名称的映射关系
                    cachedNames.put(clazz, n);
                }
                Class<?> c = extensionClasses.get(n);
                if (c == null) {
                    // 存储名称到 Class 的映射关系
                    extensionClasses.put(n, clazz);
                } else if (c != clazz) {
                    throw new IllegalStateException("...");
                }
            }
        }
    }
}
```

loadClass方法主要用于操作缓存。

```
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            // 遍历目标类的所有方法
            for (Method method : instance.getClass().getMethods()) {
                // 检测方法是否以 set 开头，且方法仅有一个参数，且方法访问级别为 public
```

```
                if (method.getName().startsWith("set")
                    && method.getParameterTypes().length == 1
                    && Modifier.isPublic(method.getModifiers())) {
                    // 获取 setter 方法参数类型
                    Class<?> pt = method.getParameterTypes()[0];
                    try {
                        // 获取属性名，比如 setName 方法对应属性名 name
                        String property = method.getName().length() > 3 ?
                            method.getName().substring(3, 4).toLowerCase() +
                                method.getName().substring(4) : "";
                        // 从 ObjectFactory 中获取依赖对象
                        Object object = objectFactory.getExtension(pt,
property);
                        if (object != null) {
                            // 通过反射调用 setter 方法设置依赖
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        logger.error("fail to inject via method...");
                    }
                }
            }
        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}
```

Dubbo IOC 是通过 setter 方法注入依赖。Dubbo 首先会通过反射获取到实例的所有方法，然后再遍历方法列表，检测方法名是否具有 setter 方法特征。若有，则通过 ObjectFactory 获取依赖对象，最后通过反射调用 setter 方法将依赖设置到目标对象中。

在上面代码中，objectFactory 变量的类型为 AdaptiveExtensionFactory，AdaptiveExtensionFactory 内部维护了一个 ExtensionFactory 列表，用于存储其他类型的 ExtensionFactory。Dubbo 目前提供了两种 ExtensionFactory，分别是 SpiExtensionFactory 和 SpringExtensionFactory。前者用于创建自适应的拓展，后者是用于从 Spring 的 IOC 容器中获取所需的拓展。

【总结】

Dubbo是在运行时根据方法参数来决定该使用哪个扩展，使用哪个接口的代理实现类的实例。比如dubbo在协议层protocol就支持不同协议的扩展，比如dubbo协议、http协议等，还有负载均衡策略的不同扩展，比如根据扩展点名称得到扩展类，对于LoadBalance，根据random得到RandomLoadBalance类。

所以他的关键在于动态选择扩展，而这种方法的实现是基于Dubbo的SPI机制，这种SPI机制和Java原生的SPI的不同点在于，java的spi只是在配置文件中定义一个或若干个代理实现类的全限定路径，并没有像我们常见的比如spring配置文件那种key-value的参数配置参数，这样就会使得用java-spi机制定义时，在获取这些扩展的实现类的时候，不能直接根据key来获取我们希望的value即实现类，而是通过遍历取出后再判断过滤，其次，java-spi这种方式也不能支持在一个扩展中直接通过依赖注入的方式注入另一个扩展，而且也不支持类似AOP的面向切面编程，所以Dubbo SPI就是为了解决这几类问题而产生的。

Dubbo SPI可以基于反射机制完成依赖注入，也可以解决AOP问题，同时SPI的配置文件中也增加了key-value的配置方式，还支持用特殊的比如@Adaptive注解来声明一个代理实现类，这些方式，就直接决定了dubbo可以很方便地通过传入一个扩展实现类的关键字key，来获取对应的代理实现，然后执行代理逻辑。这就是dubbo sip的基本原理。

再扩展一点讲，我们根据key获取代理实现对象的过程，是发生在运行时根据key来选择实现类的。dubbo会提前准备好这些key和实现类的对应关系，缓存起来，常见的就是用了一些map结构来存储。而dubbo要获取这些对应关系的前提，首先就需要知道有哪些key和实现类，这些是通过解析配置文件来完成的。另外一种方式就是，通过给某个类加上@Adaptive注解，就可以在dubbo声明为一个代理实现类，这些也会最终被添加到缓存中。

而这个场景多见于在解决依赖注入问题的时候，dubbo首先需要知道有哪些依赖，这些依赖的类型是什么，比如可以是dubbo spi定义的类型、可以是spring bean的类型，也可以是自己定义的通过@Adaptive注解声明的型。这个时候因为类型多样，所以dubbo中会有一个ExtensionFactory 扩展工厂，来生成对应的不同的类型。也可以实现自己的ExtensionFactory，让Dubbo的自动装配支持我们自定义的组件。

所以当上面说的都准备好了以后，也就是扩展对象的关键字key和对应的代理实现类实例都缓存好了，运行时就可以通过key来取得对应的接口代理实现实例，再调用方法执行，就是让代理去执行，返回结果。

10、总结

【Dubbo SPI】

1）问题的出现

Dubbo SPI 是对 Java SPI 机制的一种改进，是一种对同个接口能得到多种扩展实例的实现方案。它之所以要改进，主要是为了解决2个问题。一个是解决加载扩展点的时候的性能问题，另一个是解决依赖注入的问题，即 IOC 和 AOP 的功能实现的问题。

2）全部加载部分需要、键值对与缓存

首先为什么需要解决性能问题，是因为 Java SPI 在加载某个扩展点的时候，是一次性将所有的扩展实例都加载并实例化的，如果扩展实例很多那么加载的开销很大，而且当中如果只有少部分实例需要用到，其他大部分不需要的话，那么这种开销无疑是很浪费的。所以 Dubbo SPI 做了改进，一方面将配置文件做成了 key-value 的形式，这样的话就支持通过某个 key 来获得对应的扩展实例，就不用全部都去加载；另一方面，与 Java SPI 不同的是，Dubbo SPI 首先将具体的扩展类先加载到 JVM 并且缓存起来，在需要真正使用的时候，才进行实例化，这样的按需实例化也降低了系统资源的开销，整体上做到了性能的提升。

3）加载扩展实现类

所以当应用启动的时候，首先会去 Dubbo 指定的路径下，把所有的扩展类都进行加载并缓存起来，再按需进行实例化，那么缓存中就有类对象的缓存和实例对象的缓存。当需要某个对象的时候，首先会去实例缓存中找，如果没有再从类对象中找到类进行实例化。

4）依赖注入（构造器和setter函数）

IOC 功能就是属性的依赖注入，是在对象实例化之后进行的，一般是通过 set 函数注入属性对象。对象实例化后就需要进行初始化，初始化就需要对属性的值进行设置，就是进行一个依赖注入，依赖注入常用的是构造器注入和setter函数注入，那么首先会进行构造器注入，这里就是为了解决一个 AOP 的问题，当一个对象的构造函数中包含有其他的扩展点作为参数时，那么 Dubbo 认为这就是一个 AOP 对象，因为它把抽象的公共逻辑抽取出来放到当前类，然后在构造器注入一个真正目标对象，这就类似动态代理在目标方法周围织入公共逻辑对目标进行增强，然后 Dubbo 也会从缓存或者新建一个对象进行设置，然后返回。另一个依赖注入的方式是 setter 函数，Dubbo 会通过反射获取所有方法，然后找到以 set 开头的函数，解析它的参数的类型和属性的名称，然后在缓存中或者新建一个对象通过反射设置进去。

5）适配器模式（注解与自动生成）

正常来说，这里就完成了 AOP 和 IOC 的功能。但有一个问题，上面注入的都是对象，可以在缓存中找到，但如果是接口呢，有多个实现但不知道当下要注入哪一个的时候，这个时候就需要用到另一个特性功能，就是 Dubbo 的自适应功能，自适应的意思就是通过动态地接受一个 SPI KEY，自动生成匹配的扩展实例来适应当前的业务场景。如果是我们自己实现的话，我会写一个对象，对象中有一个函数，功能是通过输入一个 Class 对象作为要匹配实例的一个抽象接口，以及一个 String 类型的 SPI KEY，然后在路径下找到扩展对象的类路径并实例化，最终输出一个具体的扩展实例，那么这个对象就是一个适配器对象。那么其实 Dubbo 也是这么做的，只不过它更加简洁，它提供两种方式，一种是用 @Adaptive 注解来修饰刚才说的那些适配器对象，当应用启动时 Dubbo 会扫描并解析 @Adaptive 注解然后把对应的适配器类缓存起来，然后在遇到刚才说的那种注入的属性是接口时，就会先生成一个这样的适配器对象，然后由这个对象根据传入的参数来决定要生成哪一个扩展实例。这就类似于 SpringBoot 中为了实现自动装配而诞生的@SprintBootApplication注解，包含了@EnableAutoConfigution注解，包含了@Import注解，然后生成一个适配器对象进行SPI。另一种方式是没有注解时，Dubbo 会自动通过生成代码字符串然后进行自动编译，动态地生成一个适配器对象的方式，两种方式总之都是要生成适配器对象，然后适配器对象根据动态传入的 SPI KEY，再找到对应的扩展实例。

6）适配器工厂（Dubbo SPI、Spring、My）

但这里的 SPI KEY，其实可以有不同框架的实现，比如 Dubbo 本身容器管理的实例，以及 Spring 容器管理的实例等，换句话说，当注入的对象是 Spring 容器管理的一个接口时，那么就需要到 Spring 的依赖下的 SPI 路径去找到对应的扩展实例，所以这里是典型的工厂模式，要有多个工厂根据不同业务场景产生不同实例，所以这里首先还需要工厂对象，而工厂可以有 Dubbo、Spring、以及用户自定义的，所以工厂同时又是一个 @SPI 注解修饰的抽象接口，也需要通过 SPI 机制去获取不同的实现。刚才说了 Dubbo 有两种方式生成适配器一种是注解一种是自动生成，Dubbo 是用一个 @Adaptive 注解了一个 AdativeExtensionFactory 对象，然后由这个对象去实例化所有的工厂对象并缓存起来。再回到依赖注入的场景时，就会通过属性的类型即类对象，以及动态传入的一个 SPI KEY，首先匹配到合适的工厂，再由工厂生成一个适配器对象，然后适配器对象根据传入的 SPI KEY 基于 SPI 机制找到扩展实例，最终执行实例方法。

> **问题：性能、依赖注入（IOC、AOP）**
>
> **性能解决：全部加载部分需要、键值对与缓存**
>
> **加载扩展实现类**
>
> **依赖注入（构造器和setter函数）**
>
> **适配器模式（注解与自动生成）**
>
> **适配器工厂（Dubbo SPI、Spring、My）**

【标签解析】

Spring启动流程中包含对XML文件的解析，在解析到 namespace 标签时，比如 dubbo:service 标签，就会查找对应的 spring.schemas 和 spring.handlers 文件，在dubbo中，对于dubbo标签的解析，已在这两个文件中指定了一个标签处理器 DubboNamespaceHandler。

DubboNamespaceHandler 主要把不同的标签交给不同的解析器去处理，比如和服务相关的service标签就是交给 DubboBean 解析器来解析，其他的还有一个注解解析器。DubboBean 解析器还包含了一个属性解析器，主要解析比如 dubbo:service 标签中的name、class、ref等属性。

经过一顿操作之后，标签解析器就会把标签解析成对应的bean，并注册到 Spring 容器中。

> **命名空间、配置文件、标签解析器、属性解析器（name、class、ref）、注册到 IOC**

【服务暴露】

1）标签解析、获取注册中心地址与协议信息、构造URL

Dubbo 配置文件中的dubbo:service标签经过解析后会变成Spring中的bean，并注册到Spring的上下文。然后根据配置文件信息构造一个 URL 对象，最后进行服务的暴露。Dubbo 可以支持多注册中心多协议的暴露，所以首先从配置文件获取到多个配置中心的地址，以及多种协议，然后遍历每一个协议，使用每种协议分别对多个注册中心暴露服务。

Provider 的服务暴露最终是要向注册中心发送例如一个服务名与一个API调用地址的映射关系，所以围绕这种关系构成，就需要解析配置文件的参数然后组成这样一个 API 地址，在 Dubbo 中是用 URL 的对象来保存 API 地址的，包含了 Provider 这边的一些服务信息，比如接口名、方法名、参数等。

2）本地暴露（定义、封装invoker、缓存exporter）

在构造完URL后，就进入服务真正暴露的阶段。Dubbo 的层级结构中有一层叫 Protocol 协议层，服务暴露就是在这一层进行的。这一层又分为两个模块，一个是本地暴露模块、和远程暴露模块。本地暴露模块，意思是在同个 JVM 中既有服务提供者也有服务消费者，那么服务提供者只需要把服务信息缓存在 JVM 本地，然后消费者需要消费的时候，就直接取本地 JVM 的缓存的服务进行调用就行，以提升服务调用的效率。这个过程发生在Protocol 层的 InjvmProtocol 协议模块中，具体的就是将真正服务的实现对象封装到一个代理对象 invoker 中，然后再封装到一个 exporter 对象中，用 exporter 来管理 Invoker 的生命周期，然后放到一个 Map 里，Map 的 value 就是 exporter 对象，key 就是一个服务的标识 serverKey，是用接口名、方法名、参数名等规则形成的，来唯一标识一个服务的 key。那么下次客户端调用时，如果判断是本地服务，那么就根据客户端请求生成的服务标识serverKey，在exporterMap中直接取得对应的exporter，再通过exporter取得invoker，进行调用即可。

invoker 和 exporter 的关系是，invoker 是一个代理对象，持有真正服务的实现者，通过接受客户端请求，解析出请求中的服务信息比如接口名、方法名、参数等来匹配出一个服务的实现者，然后调用服务的实现方法，最终完成一次 RPC 调用。而 exporter 封装了 invoker，负责管理其生命周期，并作为最终的服务包装对象暴露给消费者。

3）远程暴露（DubboProtocol 创建 netty 服务器、缓存 exporter，RegistryProtocol 创建注册中心并发布服务信息）

然后进行远程暴露，就是将服务信息注册到注册中心，可以通过 HTTP 协议或者 Dubbo 协议进行暴露，也是通过 SPI 来选择的，默认是 Dubbo 协议，那么就会委托给具体的 DubboProtocol 模块去进行。Dubbo 底层通信基于 Netty 框架的，所以首先是创建一个nettyServer服务器，监听端口并且绑定一个requestHandler，把客户端的请求交给这个handler去具体处理。同时 DubboProtocol 模块会把 invoker 就是服务的代理执行对象封装到 exporter，像之前本地暴露一样，也将服务的标识和对应的 exporter 的对应关系缓存到一个 Map。之后接收到客户端请求后，就可以提取客户端的请求依据相同的规则构造服务标识serverKey，然后在这个exporterMap中找到对应的exporter，再找到invoker进行方法调用。在经过 DubboProtocol 模块搭建好 netty 服务器以及缓存好服务标识与对应的代理对象后，再通过 RegistryProtocol 模块创建注册中心，并发布服务信息到注册中心去。

> **服务暴露**
>
> **1）标签解析、获取注册中心地址与协议信息、构造URL**
>
> **2）本地暴露（定义、封装invoker、缓存exporter）**
>
> **3）远程暴露（DubboProtocol 创建 netty 服务器、缓存 exporter，RegistryProtocol 创建注册中心并发布服务信息）**

【服务引用】

服务的引用，就是获取一个引用对象，即代理对象。这个引用不是真正的服务实现，而是通过协议的方式引用到远端的真正服务，只是封装起来，屏蔽了发起远程请求的过程，看起来像是本地调用一样。

1）本地引用（服务标识、invoker）

服务的引用包括本地引用和远程引用，和之前服务的暴露一样，也是在 protocol 层实现的，服务的本地引用指的是服务的提供者和消费者在同一个 JVM 中，那么消费者只需要根据所要请求的服务的接口名、方法名和参数等相关信息组成一个服务的唯一标识，然后在本地的服务缓存中根据服务标识这个key，取出对应的服务的代理对象invoker，再直接调用即可。因为之前的服务的本地暴露过程中，就已经将相关的服务信息缓存起来了。Dubbo 是如何判断是否属于本地引用的呢，它是根据我们配置文件中的服务提供者的ip与本地ip做对比，如果一样，就说明是本地引用。

2）远程引用（注册单元进行服务的发现）

如果不是本地服务，那么就要进行远程引用。要进行服务引用，就需要有远端服务提供者的服务信息，比如每个服务名与对应的 API 地址等，就需要去注册中心拉取服务信息，然后缓存在本地服务列表。消费者再根据将要引用的服务信息到本地服务列表中找到对应的API，再发起调用。所以这里面需要两个角色，一个是负责服务信息的拉取，一个是负责服务的调用。对应到 Dubbo 中，分别是注册单元 RegistryDirectory 对象和具体协议的引用者 DubboProtocol 对象。注册单元负责服务的发现，当消费端第一次订阅注册中心 provider 节点信息时，会做一次全量的拉取，然后回调到注册单元的回调函数中，再根据这些回调到的服务提供者的信息创建不同的服务标识对应的 invoker，最终形成不同服务的服务标识（服务的地址）对应不同 invoker 对象的局面，其实很容易理解，因为不同的服务对应不同的调用过程，本质是因为 API 地址不一样。

注册单元负责拉取注册中心中服务提供者的服务列表，根据列表中的每个服务来创建一个 invoker 代理对象，这个代理对象承担了服务的调用功能。具体的 invoker 的创建就交给了一个 DubboProtocol 模块来进行。同样地，在服务暴露过程中也是在这个模块创建 nettyserver，那么这里服务的引用就是创建一个 nettyclient。负责向服务提供者发起请求并接收响应，然后把 netty 客户端封装到 invoker 中，所以说 invoker 才是真正服务引用的底层实现，承担着服务的调用功能。

3）远程引用（DubboProtocol 进行服务的调用、创建 Netty 客户端、缓存服务 invoker）

服务信息拉取到了之后，就需要发起调用。通过 DubboProtocol 模块首先创建一个 netty 客户端，负责向服务提供者发起请求并接收响应，然后把 netty 客户端封装到 invoker 中，所以说 invoker 才是真正服务引用的底层实现，承担着服务的调用功能。同时 DubboProtocol 将不同的服务标识与不同的 invoker 对象做一个映射并缓存到注册单元的一个 Map 中。当服务发起调用时，首先会将请求发给代理对象，而代理对象通过请求的参数组成一个服务标识，再通过注册单元中缓存有服务信息的 Map 匹配得到一个最终的 invoker，而 invoker 持有 netty 客户端，就通过它向服务提供者发起请求，并接收返回的结果，这就是一次服务引用的过程。如果 provider 是集群部署的话，就会取到多个 provider 信息和对应的 invoker 对象，dubbo 会根据负载均衡策略选择其中一个 invoker 发起调用。

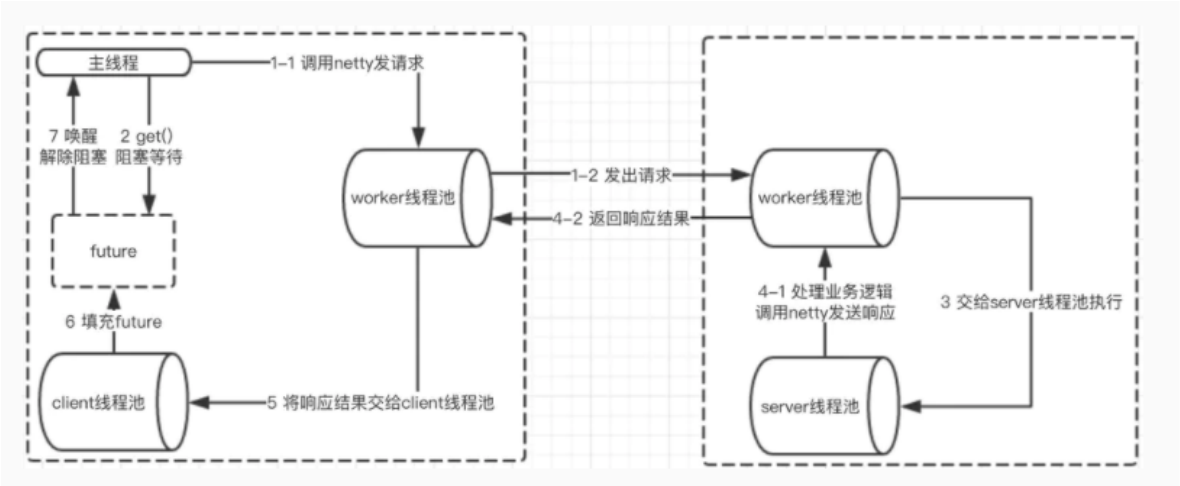
> **服务引用**
>
> **1）本地引用（服务标识、invoker）**
>
> **2）远程引用（注册单元进行服务的发现）**
>
> **3）远程引用（DubboProtocol 进行服务的调用、创建 Netty 客户端、缓存服务 invoker）**

【服务调用】

消费端发起调用时，会生成一个 Future 对象来接收返回结果，并生成一个 requestID ，然后将这个 ID 与 Future 对象绑定在一起存到 Map，意思是用 requestID 来唯一标识每次请求，并将请求结果放到对应的 Future 中。消费端的线程发起调用后就进入阻塞等待返回结果。

服务端收到请求后，根据请求的参数组成一个服务标识，以该标识为 key，取出对应的 exporter，因为之前服务暴露的时候就将服务信息封装在 exporter 并与服务标识绑定在一起缓存，所以这里只要取出 exporter，再取出其中的 invoker 对象，invoker 对象是一个代理，持有一个服务的真正实现，然后调用服务的实现并返回结果。

客户端收到响应后，根据 requestID 找到 Future，将结果填充进去，唤醒正在阻塞的用户线程，最终得到响应。



> **绑定 Future 与 requestID**
>
> **取出 exporter**
>
> **填充 Future，唤醒线程**

【Netty 实现】

基于非阻塞的多路复用与事件驱动模型，使用 epoll 机制。简单地说，就是通过建立一个网卡设备到用户进程的一个回调，当有数据到达网卡时，内核就会把就绪事件放到内核空间的一个就绪队列中。而用户进程在处理完连接后就会不断轮询这个队列是否有事件的到来。

在 Netty 中，epoll 这种机制就是多路复用IO，就是可以用一个线程去同时监听多个客户端的数据是否到达，如果当前客户端没有数据，则轮询下一个，不停地遍历这些客户端。所以 Netty 也是起一个扫描线程，不停监听就绪队列的事件，当有事件到达，就把事件取出，然后交给对应的事件处理器，每个事件处理器对应一个线程，这样的话扫描的那个线程就只需要负责把事件分发出去就行，然后继续轮询，提高了轮询的效率，这就是 Netty 的事件驱动机制。

而事件的类型比如有连接事件、读写事件等，需要有不同的线程去处理，而 Netty 进一步将这些线程进行划分，分为不同的线程组。其中主线程组负责客户端的连接事件，即专门处理客户端的连接；而另一个工作线程组用来负责读写事件等，这样区分开来便于维护，也间接地提升了性能。

> **非阻塞多路复用、事件驱动、epoll**
>
> **多路复用、两种线程**
>
> **两种线程组**

11、Q&A

1、Dubbo 有几种负载均衡策略

基于权重的随机负载均衡（默认）

基于权重的轮询负载（比如节点abc，权重为321，则a轮询3次，b轮询2次，c轮询1次）

最少活跃调用数。响应快的提供者接受越多请求，响应慢的接受越少请求。

一致性hash，根据某个参数进行 hash，分配到某个节点上。

2、常用配置

超时时间、重试次数、启动时是否检查服务存在、服务直连地址

配置项	作用
timeout	超时时间，单位为毫秒， 默认1000ms
retries	重试次数，不包括第一次调用，默认为2，0表示不重试
check	启动时检查提供者是否存在，true表示不存在时报错，false表示启动时不检查，默认为true
url	点对点直连服务提供者，绕过注册中心，以服务接口为单位

提供者（应用名、注册中心地址registry、协议端口protocol、暴露的服务接口service、方法、实例bean）

```
<!--配置当前dubbo应用程序的名称，可自定义，但必须唯一，一般使用项目名-->
<dubbo:application name="dubbo-provider"/>

<!--指定注册中心地址-->
<dubbo:registry address="zookeeper://127.0.0.1:2181"/>

<!--配置dubbo协议和端口（通过该端口来提供服务）-->
<dubbo:protocol name="dubbo" port="8888"/>

<!--指定要暴露的服务接口-->
<dubbo:service interface="com.itany.service.UserService" ref="userService"/>

<bean id="userService" class="com.itany.service.impl.UserServiceImpl"/>
```

引用者（应用名、注册中心地址registry、引用接口reference）

```
<dubbo:application name="dubbo-consumer"/>

<dubbo:registry address="zookeeper://127.0.0.1:2181"/>

<!--指定要引用的服务-->
<dubbo:reference id="userService" interface="com.itany.service.UserService"/>

<bean class="com.itany.controller.UserController">
    <property name="userService" ref="userService" />
</bean>
```

3、配置的优先级关系

dubbo属性可以配置在如下六个位置：

消费者（提供者）：引用方法处、接口处、全局配置处

方法优先，接口次之，全局配置再次之。如果级别一样，则消费方优先，提供方次之。

注：建议由服务提供者设置超时时间

4、几种配置方式

1）XML 配置方式，通过 Dubbo 的比如 service、protocol、reference 等标签来定义提供者和引用者的信息。

2）dubbo.properties 的属性配置文件方式，可以用来放置比如应用名、注册中心地址等公共配置信息。

3）注解的方式：

提供者（配置服务所在的包名和扫描器、用 dubbo 特有的 @service 注解标识某个服务）

```
<!--dubbo组件的扫包-->
<dubbo:annotation package="com.itany.service.impl"/>

<!--spring组件的扫包-->
<context:component-scan base-package="com.itany.service.impl"/>
```

```
// 方式1
// @Service
// @com.alibaba.dubbo.config.annotation.Service
// 方式2
@Component
@Service
public class UserServiceImpl implements UserService {
```

引用者（配置引用相关的类的包名以及扫描器、使用 @Reference 注解注入远程服务对象）

```
<dubbo:annotation package="com.itany.controller"/>

<context:component-scan base-package="com.itany.controller"/>
```

```
@Controller
public class UserController {

    // 使用dubbo的@Reference注入远程服务对象
    @Reference
    private UserService userService;
```

4）API 配置方式

5、Dubbo Provider 如何设置并发

1）通过 executes 参数来设置服务端某个接口或者接口中的某个方法的并发执行次数

```
<dubbo:service interface="com.foo.BarService" executes="10" />

<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

2）通过 actives 参数设置客户端（服务端）某个接口或接口中的某个方法的并发请求次数

```
<dubbo:reference interface="com.foo.BarService" actives="10" />
<dubbo:dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

参考资料

解密Dubbo ——深入解析Dubbo RPC 原理
https://www.jianshu.com/p/bd34e6ae06cf

第8章 Dubbo 服务引用流程的设计与实现
https://www.jianshu.com/p/cbea352ab735

第9章 Dubbo 服务通信流程的设计与实现
https://www.jianshu.com/p/a3b6859ca07f

第4章 Dubbo SPI 的设计与实现
https://www.jianshu.com/p/08b41189eb4c

第7章 Dubbo 服务暴露流程的设计与实现
https://www.jianshu.com/p/1e95c541d1e6

Dubbo之服务暴露源码分析
https://www.jianshu.com/p/8a5d4e3fee8b

第10章 Dubbo 代理层的设计与实现
https://www.jianshu.com/p/25a3108385fa

Dubbo可扩展机制源码解析
http://dubbo.apache.org/zh-cn/blog/introduction-to-dubbo-spi-2.html

官网
http://dubbo.apache.org/zh-cn/docs/source_code_guide/dubbo-spi.html
http://dubbo.apache.org/zh-cn/docs/source_code_guide/adaptive-extension.html
http://dubbo.apache.org/zh-cn/docs/source_code_guide/export-service.html
http://dubbo.apache.org/zh-cn/docs/source_code_guide/refer-service.html
http://dubbo.apache.org/zh-cn/docs/source_code_guide/router.html
http://dubbo.apache.org/zh-cn/docs/source_code_guide/service-invoking-process.html