SSSS

咕泡学院 VIP 课： Dubbo 常用配置文件分析及核心源码阅读

多版本支持

多版本支持的演示，请参考课堂演示的代码。

设置不同版本的目的，就是要考虑到接口升级以后带来的兼容问题。在 Dubbo 中配置不同版本的接口，会在 Zookeeper 地址中有多个协议 url 的体现，具体内容如下

```
dubbo://192.168.11.1:20880%2Fcom.gupaoedu.dubbo.IGpHello%3Fanyhost%3Dtrue%26application%3Dhello-world-app%26dubbo%3D2.5.6%26generic%3Dfalse%26interface%3Dcom.gupaoedu.dubbo.IGpHello%26methods%3DsayHello%26pid%3D60700%26revision%3D1.0.0%26side%3Dprovider%26timestamp%3D
```

1529498478644%26version%3D1.0.0

dubbo://192.168.11.1%3A20880%2Fcom.gupaoedu.dubbo.IGpHello2%3Fanyhost%3Dtrue%26application%3Dhello-world-app%26dubbo%3D2.5.6%26generic%3Dfalse%26interface%3Dcom.gupaoedu.dubbo.IGpHello%26methods%3DsayHello%26pid%3D60700%26revision%3D1.0.1%26side%3Dprovider%26timestamp%3D1529498488747%26version%3D1.0.1

主机绑定

在发布一个 Dubbo 服务的时候，会生成一个 dubbo://ip:port 的协议地址，那么这个 IP 是根据什么生成的呢？大家可以在 ServiceConfig.java 代码中找到如下代码;可以发现，在生成绑定的主机的时候，会通过一层一层的判断，直到获取到合法的 ip 地址。

1. NetUtils.isInvalidLocalHost(host)， 从配置文件中获取 host

```
2.    host =
  InetAddress.getLocalHost().getHostAddress(
  );
```

```
3.    Socket socket = new Socket();
  try {
      SocketAddress addr = new
  InetSocketAddress(registryURL.getHost(),
  registryURL.getPort());
      socket.connect(addr, 1000);
      host =
  socket.getLocalAddress().getHostAddress();
      break;
  } finally {
      try {
          socket.close();
      } catch (Throwable e) {}
  }
```

```
public static String getLocalHost(){
    InetAddress address = getLocalAddress();
    return address == null ? LOCALHOST : address.getHostAddress();
}
```

4.

集群容错

什么是容错机制？ 容错机制指的是某种系统控制在一定范围内的一种允许或包容犯错情况的发生，举个简单例子，我们在电脑上运行一个程序，有时候会出现无响应的

情况，然后系统会弹出一个提示框让我们选择，是立即结束还是继续等待，然后根据我们的选择执行对应的操作，这就是“容错”。

在分布式架构下，网络、硬件、应用都可能发生故障，由于各个服务之间可能存在依赖关系，如果一条链路中的其中一个节点出现故障，将会导致雪崩效应。为了减少某一个节点故障的影响范围，所以我们才需要去构建容错服务，来优雅的处理这种中断的响应结果

Dubbo 提供了 6 种容错机制，分别如下

1. failsafe 失败安全，可以认为是把错误吞掉（记录日志）
2. failover(默认)　重试其他服务器；　retries（2）
3. failfast 快速失败，　失败以后立马报错
4. failback　失败后自动恢复。
5. forking　forks. 设置并行数
6. broadcast　广播，任意一台报错，则执行的方法报错

配置方式如下，通过 cluster 方式，配置指定的容错方案

```
<!-- 声明需要暴露的服务接口 -->
<dubbo:reference id="demoService" interface="com.gupaoedu.dubbo.IGpHello"
                 registry="zookeeper" version="1.0.0"
cluster="failsafe"/>
```

服务降级

降级的目的是为了保证核心服务可用。

降级可以有几个层面的分类： 自动降级和人工降级； 按照功能可以分为：读服务降级和写服务降级；

1. 对一些非核心服务进行人工降级，在大促之前通过降级开关关闭哪些推荐内容、评价等对主流程没有影响的功能
2. 故障降级，比如调用的远程服务挂了，网络故障、或者 RPC 服务返回异常。 那么可以直接降级，降级的方案比如设置默认值、采用兜底数据（系统推荐的行为广告挂了，可以提前准备静态页面做返回）等等
3. 限流降级，在秒杀这种流量比较集中并且流量特别大的情况下，因为突发访问量特别大可能会导致系统支撑不了。这个时候可以采用限流来限制访问量。当达到阀值时，后续的请求被降级，比如进入排队页面，比如跳转到错误页（活动太火爆，稍后重试等）

dubbo 的降级方式： Mock

实现步骤

1. 在 client 端创建一个 TestMock 类，实现对应 IGpHello 的接口（需要对哪个接口进行 mock，就实现哪个），名称必须以 Mock 结尾
2. 在 client 端的 xml 配置文件中，添加如下配置，增加一个 mock 属性指向创建的 TestMock
3. 模拟错误（设置 timeout），模拟超时异常，运行测试

代码即可访问到 TestMock 这个类。当服务端故障解除以后，调用过程将恢复正常

```
<!-- 声明需要暴露的服务接口 -->
<dubbo:reference id="demoService"
                 interface="com.gupaoedu.dubbo.IGpHello"
                 registry="zookeeper"

                 mock="com.gupaoedu.dubbo.TestMock" timeout="50"/>
```

配置优先级别

- 以 timeout 为例，显示了配置的查找顺序，其它 retries, loadbalance 等类似。

  1. 方法级优先，接口级次之，全局配置再次之。

  2. 如果级别一样，则消费方优先，提供方次之。

  其中，服务提供方配置，通过 URL 经由注册中心传递给消费方。

- 建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一个消费方同时引用多个服务，就不需要关心每个服务的超时设置。

Dubbo SPI 和 JAVA SPI 的使用和对比

在 Dubbo 中，SPI 是一个非常核心的机制，贯穿在几乎所有的流程中。搞懂这块内容，是接下来了解 Dubbo 更多源码的关键因素。

Dubbo 是基于 Java 原生 SPI 机制思想的一个改进，所以，先从 JAVA SPI 机制开始了解什么是 SPI 以后再去学习 Dubbo 的 SPI，就比较容易了

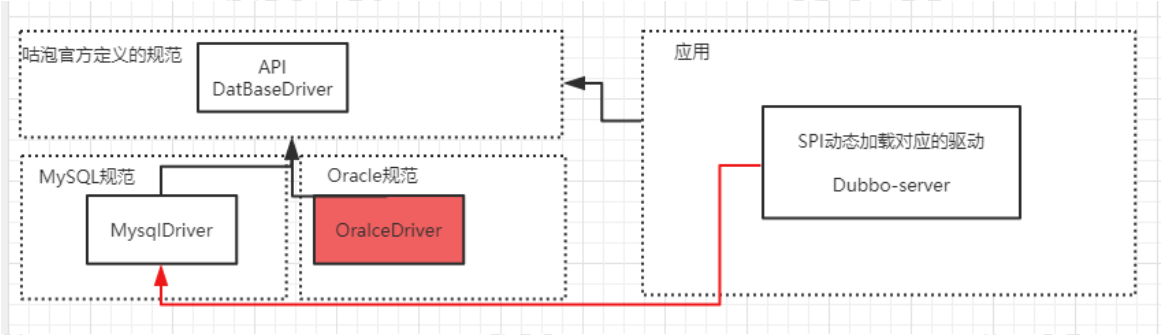
关于 JAVA 的 SPI 机制

SPI 全称（service provider interface），是 JDK 内置的一种服务提供发现机制，目前市面上有很多框架都是用它来做服务的扩展发现，大家耳熟能详的如 JDBC、日志框架都有用到；

简单来说，它是一种动态替换发现的机制。举个简单的例子，如果我们定义了一个规范，需要第三方厂商去实现，那么对于我们应用方来说，只需要集成对应厂商的插件，既可以完成对应规范的实现机制。 形成一种插拔式的扩展手段。

实现一个 SPI 机制

实现的代码的流程图如下



代码演示

《详见课堂中的代码实现过程，文档中就不重复写了》

SPI 规范总结

实现 SPI，就需要按照 SPI 本身定义的规范来进行配置，SPI 规范如下

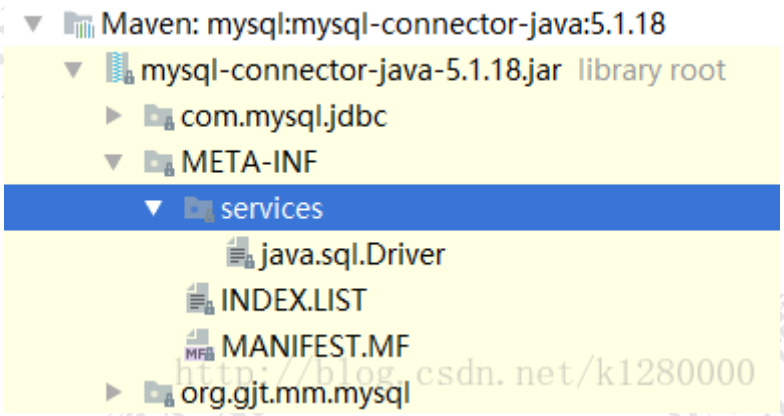
1. 需要在 classpath 下创建一个目录，该目录命名必须是：META-INF/services
2. 在该目录下创建一个 properties 文件，该文件需要满足以下几个条件
   a) 文件名必须是扩展的接口的全路径名称
   b) 文件内部描述的是该扩展接口的所有实现类
   c) 文件的编码格式是 UTF-8
3. 通过 java.util.ServiceLoader 的加载机制来发现

SPI 的实际应用

SPI 在很多地方有应用，大家可以看看最常用的 java.sql.Driver 驱动。JDK 官方提供了 java.sql.Driver 这个驱动扩展点，但是你们并没有看到 JDK 中有对应的 Driver 实现。 那在哪里实现呢?

以连接 Mysql 为例，我们需要添加 mysql-connector-java 依赖。然后，你们可以在这个 jar 包中找到 SPI 的配置信息。

如下图，所以 java.sql.Driver 由各个数据库厂商自行实现。这就是 SPI 的实际应用。当然除了这个意外，大家在 spring 的包中也可以看到相应的痕迹



SPI 的缺点

1. JDK 标准的 SPI 会一次性加载实例化扩展点的所有实现，什么意思呢？就是如果你在 META-INF/service 下的文件里面加了 N 个实现类，那么 JDK 启动的时候都会一次性全部加载。那么如果有的扩展点实现初始化很耗时或者如果有些实现类并没有用到，那么会很浪费资源
2. 如果扩展点加载失败，会导致调用方报错，而且这个错误很难定位到是这个原因

Dubbo 优化后的 SPI 实现

基于 Dubbo 提供的 SPI 规范实现自己的扩展

在了解 Dubbo 的 SPI 机制之前，先通过一段代码初步了解

Dubbo 的实现方式，这样，我们就能够形成一个对比，得到这两种实现方式的差异

《代码实现，参考视频直播+课后的视频，这里偷懒不写了》

Dubbo 的 SPI 机制规范

大部分的思想都是和 SPI 是一样，只是下面两个地方有差异。

1. 需要在 resource 目录下配置 META-INF/dubbo 或者 META-INF/dubbo/internal 或者 META-INF/services，并基于 SPI 接口去创建一个文件
2. 文件名称和接口名称保持一致，文件内容和 SPI 有差异，内容是 KEY 对应 Value

Dubbo SPI 机制源码阅读

很多同学在听完我讲的课程以后，不知道为什么分析这块代码的源码。其实前面有提到过，我们必须要知道，下面这两段代码，到底做了什么，以及会返回一个什么样的结果，如果这个不清楚，后续的代码阅读，你就没办法清晰学习。

ps: 源码阅读，带着疑问去了解【为什么传入一个 myProtocol 就能获得自定义的 DefineProtocol 对象】、

getAdaptiveExtension 是一个什么东西?

```
1. Protocol protocol = ExtensionLoader.
   getExtensionLoader(Protocol.class).
   getExtension("myProtocol");
2. Protocol protocol =
   ExtensionLoader.getExtensionLoader(Protocol.class).
   getAdaptiveExtension();
```

源码阅读入口

接下来的源码分析，是基于下面这段代码作为入口，至于为什么不用上面提到的第一段代码作为入口。理由如下

1. 在下节课讲的分析 dubbo 源码之服务发布过程中，会有下面这段代码的体现。
2. 分下完下面这段代码，对于第一段代码的理解，会很容易

```
Protocol protocol =
ExtensionLoader.getExtensionLoader(Protocol.class).
getAdaptiveExtension();
```

- 把上面这段代码分成两段，一段是 getExtensionLoader、另一段是 getAdaptiveExtension。 初步猜想一下；
  - 第一段是通过一个 Class 参数去获得一个 ExtensionLoader 对象，有点类似一个工厂模式。

- 第二段 getAdaptiveExtension，去获得一个自适应的扩展点

Extension 源码的结构

了解源码结构，建立一个全局认识。结构图如下

初步了解这些代码在扩展点中的痕迹。

Protocol 源码

一下是 Protocol 的源码，在这个源码中可以看到有两个注解，一个是在类级别上的 @SPI("dubbo")，另一个是 @Adaptive

@SPI 表示当前这个接口是一个扩展点，可以实现自己的扩展实现，默认的扩展点是 DubboProtocol。

@Adaptive 表示一个自适应扩展点，在方法级别上，会动态生成一个适配器类

```
@SPI("dubbo")
public interface Protocol {


    /**
     * 获取缺省端口，当用户没有配置端口时使用。
     *
     * @return 缺省端口
     */
    int getDefaultPort();


    /**
     * 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
     */
    @Adaptive
    <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，并内部自动恢复。<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
     */
    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;


    /**
     * 释放协议：<br>
     * 1. 取消该协议所有已经暴露和引用的服务。<br>
     * 2. 释放协议所占用的所有资源，比如连接和端口。<br>
     * 3. 协议在释放后，依然能暴露和引用新的服务。<br>
     */
    void destroy();
}
```

getExtensionLoader

该方法需要一个 Class 类型的参数，该参数表示希望加载

的扩展点类型，该参数必须是接口，且该接口必须被@SPI 注解注释，否则拒绝处理。检查通过之后首先会检查 ExtensionLoader 缓存中是否已经存在该扩展对应的 ExtensionLoader，如果有则直接返回，否则创建一个新的 ExtensionLoader 负责加载该扩展实现，同时将其缓存起来。可以看到对于每一个扩展，dubbo 中只会有一个对应的 ExtensionLoader 实例

```
@SuppressWarnings("unchecked")
public static <T> ExtensionLoader<T>
getExtensionLoader(Class<T> type) {
    if (type == null)
        throw new
IllegalArgumentException("Extension type ==
null");
    if(!type.isInterface()) {
        throw new
IllegalArgumentException("Extension type(" +
type + ") is not interface!");
    }
    if(!withExtensionAnnotation(type)) {
        throw new
```

```
IllegalArgumentException("Extension type(" +
type +
                ") is not extension, because
WITHOUT @" + SPI.class.getSimpleName() + "
Annotation!");
    }


    ExtensionLoader<T> loader =
(ExtensionLoader<T>)
EXTENSION_LOADERS.get(type);
    if (loader == null) {
        EXTENSION_LOADERS.putIfAbsent(type,
new ExtensionLoader<T>(type));
        loader = (ExtensionLoader<T>)
EXTENSION_LOADERS.get(type);
    }
    return loader;
}
```

ExtensionLoader 提供了一个私有的构造函数，并且在这里面对两个成员变量 type/objectFactory 进行赋值。而 objectFactory 赋值的意义是什么呢？先留个悬念

```
private ExtensionLoader(Class<?> type) {
```

```
    this.type = type;
    objectFactory = (type ==
ExtensionFactory.class ? null :

ExtensionLoader.getExtensionLoader(Extension
Factory.class).
                    getAdaptiveExtension());
}
```

getAdaptiveExtension

通过 getExtensionLoader 获得了对应的 ExtensionLoader 实例以后，再调用 getAdaptiveExtension()方法来获得一个自适应扩展点。接下来我们来看看代码的实现

ps：简单对自适应扩展点做一个解释，大家一定了解过适配器设计模式，而这个自适应扩展点实际上就是一个适配器。

这个方法里面主要做几个事情：

1. 从 cacheAdaptiveInstance 这个内存缓存中获得一个对象实例
2. 如果实例为空，说明是第一次加载，则通过双重检查锁的方式去创建一个适配器扩展点

```
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("fail to create
adaptive instance: " + t.toString(), t);
```

```
                    }

                }

            }

        }

        else {

            throw new
IllegalStateException("fail to create
adaptive instance: " +
createAdaptiveInstanceError.toString(),
createAdaptiveInstanceError);

        }

    }

    return (T) instance;

}
```

createAdaptiveExtension

这段代码里面有两个结构，一个是 injectExtension.　另一个是 getAdaptiveExtensionClass()

我们需要先去了解 getAdaptiveExtensionClass 这个方法做了什么？很显然，从后面的.newInstance 来看，应该是获得一个类并且进行实例)

```
private T createAdaptiveExtension() {
    try {
        //可以实现扩展点的注入
        return injectExtension((T)
getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can
not create adaptive extenstion " + type + ",
cause: " + e.getMessage(), e);
    }
}
```

getAdaptiveExtensionClass

从类名来看，是获得一个适配器扩展点的类。

在这段代码中，做了两个事情

1. getExtensionClasses() 加载所有路径下的扩展点
2. createAdaptiveExtensionClass() 动态创建一个扩展点

cachedAdaptiveClass 这里有个判断，用来判断当前 Protocol 这个扩展点是否存在一个自定义的适配器，如果有，则直接返回自定义适配器，否则，就动态创建，这个值是在 getExtensionClasses 中赋值的，这块代码我们稍后再

看

```
private Class<?> getAdaptiveExtensionClass()
{
    getExtensionClasses();
    //TODO  不一定?
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }
    return cachedAdaptiveClass =
createAdaptiveExtensionClass();
}
```

createAdaptiveExtensionClass

动态生成适配器代码，以及动态编译

1. createAdaptiveExtensionClassCode，动态创建一个字节码文件。返回 code 这个字符串
2. 通过 compiler.compile 进行编译（默认情况下使用的是 javassist)
3. 通过 ClassLoader 加载到 jvm 中

```
//创建一个适配器扩展点。（创建一个动态的字节码文件）
private Class<?>
```

```
createAdaptiveExtensionClass() {
    //生成字节码代码
    String code =
createAdaptiveExtensionClassCode();
    //获得类加载器
    ClassLoader classLoader =
findClassLoader();

com.alibaba.dubbo.common.compiler.Compiler
compiler =
ExtensionLoader.getExtensionLoader(com.aliba
ba.dubbo.common.compiler.Compiler.class).get
AdaptiveExtension();
    //动态编译字节码
    return compiler.compile(code,
classLoader);
}
```

CODE 的字节码内容

```
public class Protocol$Adaptive implements
com.alibaba.dubbo.rpc.Protocol {
   public void destroy() {
      throw new
UnsupportedOperationException("method public
abstract void
com.alibaba.dubbo.rpc.Protocol.destroy() of
```

```
interface com.alibaba.dubbo.rpc.Protocol is not
adaptive method!");
    }

    public int getDefaultPort() {
        throw new
UnsupportedOperationException("method public
abstract int
com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of
interface com.alibaba.dubbo.rpc.Protocol is not
adaptive method!");
    }

    public com.alibaba.dubbo.rpc.Invoker
refer(java.lang.Class arg0,
com.alibaba.dubbo.common.URL arg1) throws
com.alibaba.dubbo.rpc.RpcException {
        if (arg1 == null) throw new
IllegalArgumentException("url == null");
        com.alibaba.dubbo.common.URL url = arg1;
        String extName = (url.getProtocol() == null ?
"dubbo" : url.getProtocol());
        if (extName == null)
            throw new IllegalStateException("Fail to
get extension(com.alibaba.dubbo.rpc.Protocol) name
from url(" + url.toString() + ") use
keys([protocol])");
        com.alibaba.dubbo.rpc.Protocol extension =
(com.alibaba.dubbo.rpc.Protocol)
ExtensionLoader.getExtensionLoader(com.alibaba.dubb
o.rpc.Protocol.class).getExtension(extName);
        return extension.refer(arg0, arg1);
    }

    public com.alibaba.dubbo.rpc.Exporter
export(com.alibaba.dubbo.rpc.Invoker arg0) throws
com.alibaba.dubbo.rpc.RpcException {
        if (arg0 == null) throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Inv
oker argument == null");
        if (arg0.getUrl() == null)
            throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Inv
oker argument getUrl() == null");
```

```
        com.alibaba.dubbo.common.URL       url       =
arg0.getUrl();
        String extName = (url.getProtocol() == null ?
"dubbo" : url.getProtocol());
        if (extName == null)
            throw new IllegalStateException("Fail to
get extension(com.alibaba.dubbo.rpc.Protocol) name
from    url("    +    url.toString()    +    ")    use
keys([protocol])");
        com.alibaba.dubbo.rpc.Protocol  extension  =
(com.alibaba.dubbo.rpc.Protocol)
ExtensionLoader.getExtensionLoader(com.alibaba.dubb
o.rpc.Protocol.class).getExtension(extName);
        return extension.export(arg0);
    }
}
```

Protocol$Adaptive 的主要功能

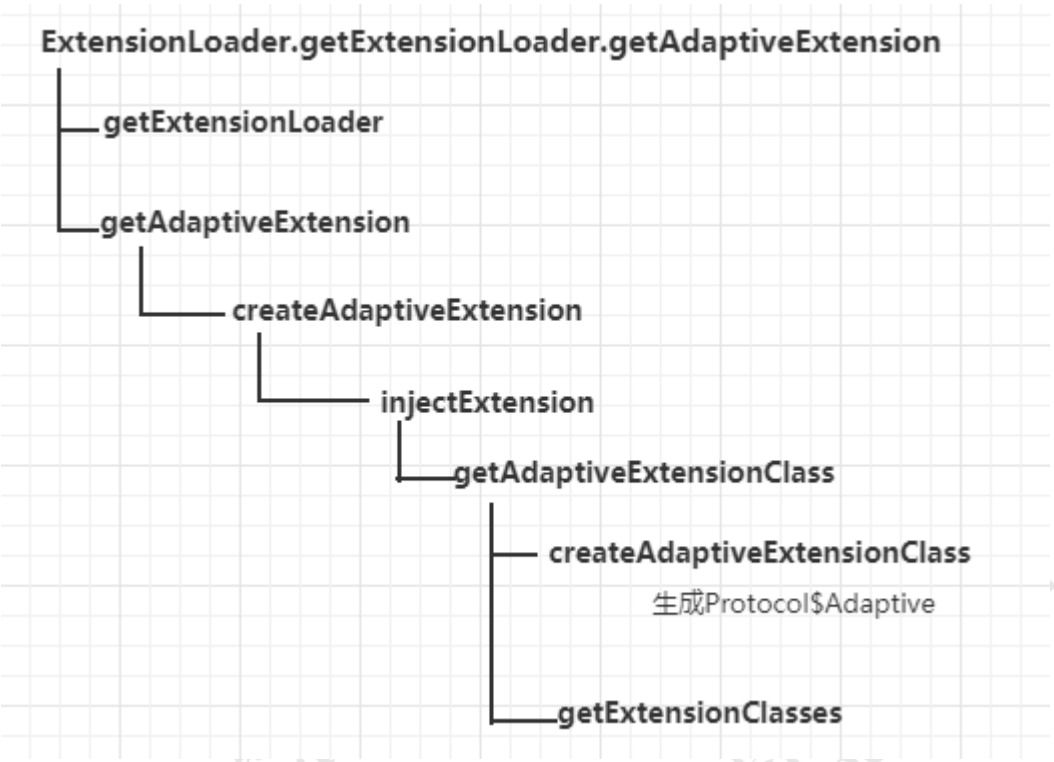
1. 从 url 或扩展接口获取扩展接口实现类的名称；

2. 根据名称，获取实现类 ExtensionLoader.getExtensionLoader(扩展接口类).getExtension(扩展接口实现类名称)，然后调用实现类的方法。

需要明白一点 dubbo 的内部传参基本上都是基于 Url 来实现的，也就是说 Dubbo 是基于 URL 驱动的技术

所以，适配器类的目的是在运行期获取扩展的真正实现来调用，解耦接口和实现，这样的话要不我们自己实现适配器类，要不 dubbo 帮我们生成，而这些都是通过 Adpative 来实现。

到目前为止，我们的 AdaptiveExtension 的主线走完了，可以简单整理一下他们的调用关系如下



我们再回过去梳理下代码，实际上在调用 createAdaptiveExtensionClass 之前，还做了一个操作。是执行 getExtensionClasses 方法，我们来看看这个方法做了什么事情

getExtensionClasses

getExtensionClasses 这个方法，就是加载扩展点实现类了。这段代码本来应该先看的，但是担心先看这段代码会容易导致大家不好理解。我就把顺序交换了下

这段代码主要做如下几个事情

1. 从 cachedClasses 中获得一个结果，这个结果实际上就是所有的扩展点类，key 对应 name，value 对应 class
2. 通过双重检查锁进行判断

3. 调用 loadExtensionClasses，去加载左右扩展点的实现

```
//加载扩展点的实现类
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

从不同目录去加载扩展点的实现，在最开始的时候讲到过的。 META-INF/dubbo ； META-INF/internal ； META-INF/services

主要逻辑

1. 获得当前扩展点的注解，也就是 Protocol.class 这个类的注解，@SPI
2. 判断这个注解不为空，则再次获得@SPI 中的 value 值
3. 如果 value 有值，也就是@SPI(“dubbo”)，则讲这个 dubbo 的值赋给 cachedDefaultName。这就是为什么我们能够通过 ExtensionLoader.getExtensionLoader(Protocol.class).getDefaultExtension() ,能够获得 DubboProtocol 这个扩展点的原因
4. 最后，通过 loadFile 去加载指定路径下的所有扩展点。也 就 是 META-INF/dubbo;META-INF/internal;META-INF/services

```
// 此方法已经 getExtensionClasses 方法同步过。
private Map<String, Class<?>>
loadExtensionClasses() {
```

```
    //type->Protocol.class
    //得到SPI的注解
    final SPI defaultAnnotation =
type.getAnnotation(SPI.class);
    if(defaultAnnotation != null) { //如果不
等于空.
        String value =
defaultAnnotation.value();
        if(value != null && (value =
value.trim()).length() > 0) {
            String[] names =
NAME_SEPARATOR.split(value);
            if(names.length > 1) {
                throw new
IllegalStateException("more than 1 default
extension name on extension " +
type.getName()
                        + ": " +
Arrays.toString(names));
            }
            if(names.length == 1)
cachedDefaultName = names[0];
```

```
        }
    }
    Map<String, Class<?>> extensionClasses
= new HashMap<String, Class<?>>();
    loadFile(extensionClasses,
DUBBO_INTERNAL_DIRECTORY);
    loadFile(extensionClasses,
DUBBO_DIRECTORY);
    loadFile(extensionClasses,
SERVICES_DIRECTORY);
    return extensionClasses;
}
```

loadFile

解析指定路径下的文件，获取对应的扩展点，通过反射的方式进行实例化以后，put 到 extensionClasses 这个 Map 集合中

```
private void loadFile(Map<String, Class<?>>
extensionClasses, String dir) {
    String fileName = dir + type.getName();
    try {
```

```
        Enumeration<java.net.URL> urls;
        ClassLoader classLoader =
findClassLoader();
        if (classLoader != null) {
            urls =
classLoader.getResources(fileName);
        } else {
            urls =
ClassLoader.getSystemResources(fileName);
        }
        if (urls != null) {
            while (urls.hasMoreElements()) {
                java.net.URL url =
urls.nextElement();
                try {
                    BufferedReader reader = new
BufferedReader(new
InputStreamReader(url.openStream(), "utf-
8"));
                    try {
                        String line = null;
                        while ((line =
```

```
reader.readLine()) != null) {
                        final int ci =
line.indexOf('#');
                        if (ci >= 0) line =
line.substring(0, ci);
                        line = line.trim();
                        if (line.length() >
0) {
                            try {
                                String name =
null;
                                int i =
line.indexOf('=');
                                if (i > 0) {//
文件采用 name=value 方式，通过 i 进行分割
                                    name =
line.substring(0, i).trim();
                                    line =
line.substring(i + 1).trim();
                                }
                                if
(line.length() > 0) {
```

```
                                        Class<?>
clazz = Class.forName(line, true,
classLoader);
                                        //加载对应的
实现类，并且判断实现类必须是当前的加载的扩展点的实现
                                        if (!
type.isAssignableFrom(clazz)) {
                                            throw
new IllegalStateException("Error when load
extension class(interface: " +

type + ", class line: " + clazz.getName() +
"), class "
                                                +
clazz.getName() + "is not subtype of
interface.");
                                        }

                                        //判断是否有
自定义适配类，如果有，则在前面讲过的获取适配类的时
候，直接返回当前的自定义适配类，不需要再动态创建
```

```
                                        // 还记得在前面讲过的 getAdaptiveExtensionClass 中有一个判断吗？是用来判断 cachedAdaptiveClass 是不是为空的。如果不为空，表示存在自定义扩展点。也就不会去动态生成字节码了。这个地方可以得到一个简单的结论；
                                        // @Adaptive 如果是加在类上， 表示当前类是一个自定义的自适应扩展点
                                        //如果是加在方法级别上，表示需要动态创建一个自适应扩展点，也就是 Protocol$Adaptive
                                        if (clazz.isAnnotationPresent(Adaptive.class)) {

if(cachedAdaptiveClass == null) {

cachedAdaptiveClass = clazz;
                                        } else if (! cachedAdaptiveClass.equals(clazz)) {

throw new IllegalStateException("More than 1 adaptive class found: "
```

```
+ cachedAdaptiveClass.getClass().getName()

+ ", " + clazz.getClass().getName());
                                        }
                                    } else {
                                        try {
                                            //如果没有 Adaptive 注解，则判断当前类是否带有参数是 type 类型的构造函数，如果有，则认为是

//wrapper 类。这个 wrapper 实际上就是对扩展类进行装饰.
                                            //可以在 dubbo-rpc-api/internal 下找到 Protocol 文件，发现 Protocol 配置了 3 个装饰
                                            //分别是,filter/listener/mock. 所以 Protocol 这个实例来说，会增加对应的装饰器

clazz.getConstructor(type);//
                                            //得
```

```
到带有public DubboProtocol(Protocol protocol)
的扩展点。进行包装

Set<Class<?>> wrappers =
cachedWrapperClasses;
                                        if
(wrappers == null) {

cachedWrapperClasses = new
ConcurrentHashSet<Class<?>>();

wrappers = cachedWrapperClasses;
                                        }

wrappers.add(clazz);//包装类
ProtocolFilterWrapper(ProtocolListenerWrappe
r(Protocol))
                                    } catch
(NoSuchMethodException e) {

clazz.getConstructor();
                                        if
```

```
(name == null || name.length() == 0) {

name = findAnnotationName(clazz);

if (name == null || name.length() == 0) {

if (clazz.getSimpleName().length() >
type.getSimpleName().length()

&&
clazz.getSimpleName().endsWith(type.getSimpl
eName())) {

name = clazz.getSimpleName().substring(0,
clazz.getSimpleName().length() -
type.getSimpleName().length()).toLowerCase()
;

} else {

throw new IllegalStateException("No such
extension name for the class " +
```

```
clazz.getName() + " in the config " + url);


}
                                                    }
                                                }


String[] names = NAME_SEPARATOR.split(name);
                                                if
(names != null && names.length > 0) {


Activate activate =
clazz.getAnnotation(Activate.class);


if (activate != null) {


cachedActivates.put(names[0], activate);
                                                }


for (String n : names) {


if (! cachedNames.containsKey(clazz)) {
```

```
cachedNames.put(clazz, n);

}

Class<?> c = extensionClasses.get(n);

if (c == null) {

extensionClasses.put(n, clazz);

} else if (c != clazz) {

throw new IllegalStateException("Duplicate
extension " + type.getName() + " name " + n
+ " on " + c.getName() + " and " +
clazz.getName());

}
                                                    }
                                                }
                                            }
                                        }
```

```
                                    }
                                } catch (Throwable
t) {

IllegalStateException e = new
IllegalStateException("Failed to load
extension class(interface: " + type + ",
class line: " + line + ") in " + url + ",
cause: " + t.getMessage(), t);

exceptions.put(line, e);
                                }
                            }
                        } // end of while read
lines
                    } finally {
                        reader.close();
                    }
                } catch (Throwable t) {
                    logger.error("Exception when
load extension class(interface: " +
                                        type + ",
```

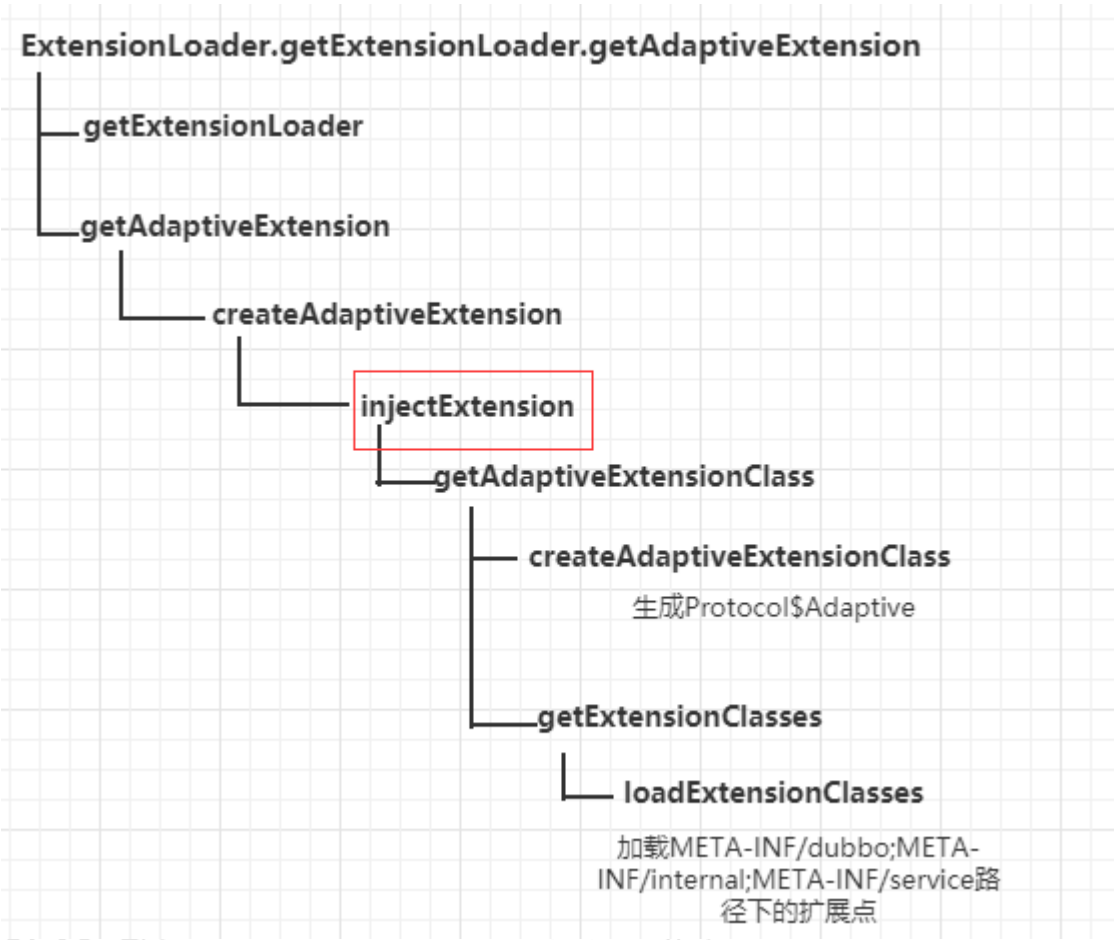
```
class file: " + url + ") in " + url, t);
                }
            } // end of while urls
        }
    } catch (Throwable t) {
        logger.error("Exception when load
extension class(interface: " +
                type + ", description file: " +
fileName + ").", t);
    }
}
```

阶段性小结

截止到目前，我们已经把基于 Protocol 的自适应扩展点看完了。也明白最终这句话应该返回的对象是什么了.

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

也就是，这段代码中，最终的 protocol 应该等于= Protocol$Adaptive



injectExtension

还记得这段代码吗？上面分析的代码，只是知道，最终 getAdaptiveExtensionClass.new Instance 获得一个自适应扩展点。而还有一段代码

injectExtension 没有讲。简单来说，这个方法的作用，是为这个自适应扩展点进行依赖注入。类似于 spring 里面的依赖注入功能。

为适配器类的 setter 方法插入其他扩展点或实现。(这块代码课堂上还没讲，大家抽空先看看)

```
private T createAdaptiveExtension() {
    try {
        //可以实现扩展点的注入
        return                    injectExtension((T)
getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can not
create adaptive extenstion " + type + ", cause:
" + e.getMessage(), e);
    }
}
```

最后留一个问题给大家?

getExtensionLoader 这个方法中，会调用 ExtensionLoader 的私有构造方法进行初始化，其中有一个 objectFactory. 这个是干嘛的呢？先想想~