链滴

Motan_ 服务注册

作者：Lord-X
原文链接：https://ld246.com/article/1571825568410
来源网站：链滴

开源推荐

推荐一款一站式性能监控工具（开源项目）

Pepper-Metrics是跟一位同事一起开发的开源组件，主要功能是通过比较轻量的方式与常用开源组件jedis/mybatis/motan/dubbo/servlet）集成，收集并计算metrics，并支持输出到日志及转换成多时序数据库兼容数据格式，配套的grafana dashboard友好的进行展示。项目当中原理文档齐全，且部基于SPI设计的可扩展式架构，方便的开发新插件。另有一个基于docker-compose的独立demo项可以快速启动一套demo示例查看效果https://github.com/zrbcool/pepper-metrics-demo。如果家觉得有用的话，麻烦给个star，也欢迎大家参与开发，谢谢：）

进入正题...

Motan系列文章

- Motan如何完成与Spring的集成
- Motan的SPI插件扩展机制
- Motan服务注册
- Motan服务调用

本文将以 注解暴露服务 的方式探究Motan服务的注册过程。

0 @MotanService注解是个啥

以 @MotanService 注解标记的类，在应用启动时，会被Motan扫描，并作为服务的具体实现注册到册中心中。

就像下面这样：

```
@MotanService(export = "demoMotan:8002")
public class MotanDemoServiceImpl implements MotanDemoService {

    @Override
    public String hello(String name) {
        System.out.println(name);
        return "Hello " + name + "!";
    }

    @Override
    public User rename(User user, String name) throws Exception {
        Objects.requireNonNull(user);
        System.out.println(user.getId() + " rename " + user.getName() + " to " + name);
        user.setName(name);
        return user;
    }
}
```

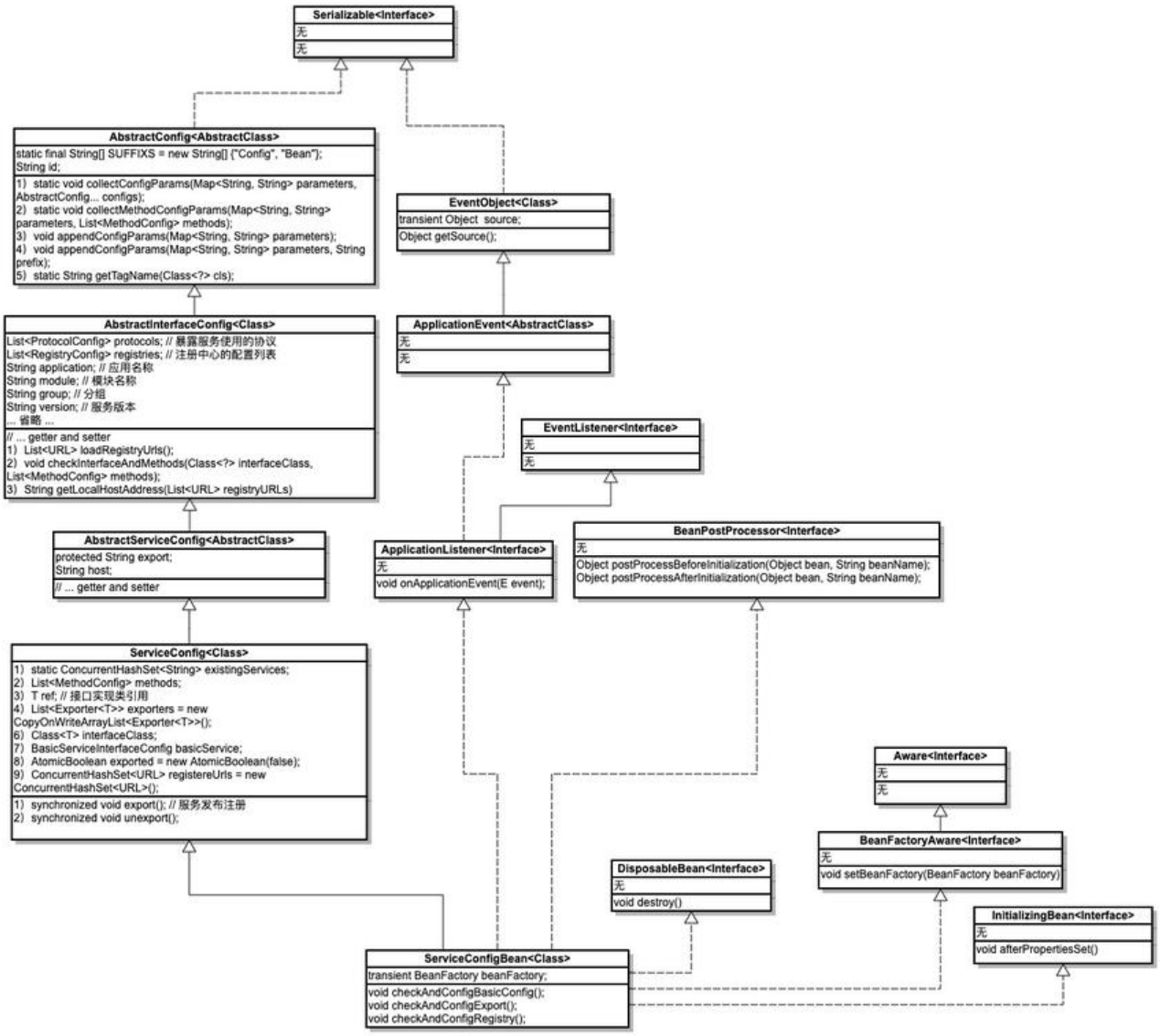
在 Motan如何完成与Spring的集成 一文中已经说过应用启动时，是如何扫描到 @MotanService 注标记的类的，这里不再赘述。

1 @MotanService的解析

@MotanService 的解析过程在 com.weibo.api.motan.config.springsupport.AnnotationBean 类的 postProcessAfterInitialization(Object bean, String beanName) 方法实现。

首先会解析配置信息，例如这个服务实现的接口是谁、以及application、moodule、group、version、ilter等的配置信息，最后会将这些信息封装到 com.weibo.api.motan.config.springsupport.Service onfigBean 的对象中。

先来看一下 ServiceConfigBean 的UML图：



图中最左侧的继承关系是 ServiceConfigBean 自身的继承关系，右侧的是Spring的相关扩展。这个这里先有个印象，我们继续上面的思路走。

Motan将配置信息封装到 ServiceConfigBean 后，调用了 afterPropertiesSet() 方法，由上图可知这个方法是 InitializingBean 接口中抽象方法的实现。

这个方法其实就干了下面三件事儿：

```
@Override
public void afterPropertiesSet() throws Exception {
    // 检查并配置basicConfig
    checkAndConfigBasicConfig();
    // 检查是否已经装配export，如果没有则到basicConfig查找
    checkAndConfigExport();
    // 检查并配置registry
    checkAndConfigRegistry();
}
```

● 检查配置解析过程后，ServiceConfigBean的 basicService 属性是否为空，如果是空，需要重新析并设置他的值。

如何重新设置他的值呢？从UML图中我们可以找到 basicService 的位置，在 ServiceConfig 类中的第个Field。字段类型是 BasicServiceInterfaceConfig。这个检查其实就是找到当前Spring容器中所有 BasicServiceInterfaceConfig 类型的bean，如果只找到一个，就把这个赋值到 basicService 上，如果多个，需要找到 BasicServiceInterfaceConfig 的 isDefault 属性为true的那个，并赋值。

- 检查 export 的值是否已经设置，如果没有设置，到 basicService 中查找。

这一步其实是检查 protocol，也就是 motan、motan2这些协议是否已经设置好，export字段的格为：protocol1:port1,protocol2:port2。对应到UML中，export字段在 AbstractServiceConfig 类。

这里同时会将 export 的值解析到 AbstractInterfaceConfig 的 protocols 字段中。

- 检查注册中心的配置

例如 zookeeper、consul 这些是否已经配置好。如果是空的话，还是从 basicService 中查找，并将果配置到 AbstractInterfaceConfig 的 registries 属性中。

这些都检查好以后，basicService、export、protocols、registries这些字段就初始化好了。然后会新创建出来的这个 ServiceConfigBean 实例添加到 AnnotationBean 的 serviceConfigs 属性中。

```
private final Set<ServiceConfigBean<?>> serviceConfigs = new ConcurrentHashSet<ServiceCnfigBean<?>>();
```

至此 @MotanService 解析完成，可以准备发布并注册服务了。

2 服务的启动

完成上述的解析和初始化后，会调用 ServiceConfigBean 的 export() 方法来发布并注册服务。

```
serviceConfig.export();
```

其实现如下：

```
public synchronized void export() { // 这里加了个并发的控制，锁使用的是 this
    // 如果已经发布过了，直接返回
    if (exported.get()) {
        LoggerUtil.warn(String.format("%s has already been expoted, so ignore the export reque
t!", interfaceClass.getName()));
        return;
    }

    // 检查暴露服务的类是否是某个接口的实现，如果不是则抛出异常
    // 检查暴露的方法是否在接口中存在，如果没有则抛出异常
    checkInterfaceAndMethods(interfaceClass, methods);

    // 解析注册中心地址，并将host、port等参数封装到URL类中
    List<URL> registryUrls = loadRegistryUrls();
    if (registryUrls == null || registryUrls.size() == 0) {
        throw new IllegalStateException("Should set registry config for service:" + interfaceClass.
etName());
    }

    // 解析协议和服务暴露的端口，默认为 `motan` 协议。Map的结构为：<协议, 端口>
```

```
    Map<String, Integer> protocolPorts = getProtocolAndPort();
    for (ProtocolConfig protocolConfig : protocols) {
        Integer port = protocolPorts.get(protocolConfig.getId());
        if (port == null) {
            throw new MotanServiceException(String.format("Unknow port in service:%s, protocol
%s", interfaceClass.getName(),
                    protocolConfig.getId()));
        }
        // 注册并暴露服务
        doExport(protocolConfig, port, registryUrls);
    }

    afterExport();
}
```

PS：URL这个类是Motan自己定义的类，Motan中几乎所有跟URL相关的东西都用它封装。

上述代码先初始化了暴露服务之前需要的一些数据：注册中心地址、服务协议、暴露端口等，真正执服务注册的是 doExport 方法。这个方法较长，这里只贴出关键部分。

```
private void doExport(ProtocolConfig protocolConfig, int port, List<URL> registryURLs) {
    // ... 省略 ...
    // 省略部分代码主要作用是处理下面这行URL中的参数，例如：protocolName -> motan，host
ddress -> 本机IP，port -> 暴露端口 等
    // map是解析出来的配置，以及一些默认配置，例如：
    /*
    "haStrategy" -> "failover"
    "module" -> "ad-common"
    "check" -> "false"
    "nodeType" -> "service"
    "version" -> "1.1.0"
    "filter" -> "cafTracing,pepperProfiler,sentinelProfiler"
    "minWorkerThread" -> "20"
    "retries" -> "1"
    "protocol" -> "motan"
    "application" -> "ad-common"
    "maxWorkerThread" -> "200"
    "shareChannel" -> "true"
    "refreshTimestamp" -> "1571821305290"
    "id" -> "ad-commonBasicServiceConfigBean"
    "export" -> "ad-commonProtocolConfigBean:8022"
    "requestTimeout" -> "30000"
    "group" -> "ad-common"
     */
    URL serviceUrl = new URL(protocolName, hostAddress, port, interfaceClass.getName(), map
;
    // 校验服务是否已经存在
    // 注册完成的服务会添加到一个set中，serviceExists方法就是检查这个set中是否已经包含了这个
务的描述符（描述符的格式大概是host、port、protocol、version、nodeType组合的字符串）
    // serviceUrl就是这个东西：motan://192.168.100.14:8022/com.coohua.ad.common.remote.a
i.AdCommonRPC?group=ad-common
    if (serviceExists(serviceUrl)) {
        LoggerUtil.warn(String.format("%s configService is malformed, for same service (%s) alre
dy exists ", interfaceClass.getName(),
```

```
            serviceUrl.getIdentity()));
        throw new MotanFrameworkException(String.format("%s configService is malformed, for
same service (%s) already exists ",
            interfaceClass.getName(), serviceUrl.getIdentity()), MotanErrorMsgConstant.FRAM
WORK_INIT_ERROR);
    }

    List<URL> urls = new ArrayList<URL>();

    // injvm 协议只支持注册到本地，其他协议可以注册到local、remote
    if (MotanConstants.PROTOCOL_INJVM.equals(protocolConfig.getId())) {
        // ... 省略，主要关注下面的注册中心暴露服务
    } else {
        for (URL ru : registryURLs) {
            urls.add(ru.createCopy()); // 这里是一个浅拷贝，只是new了一个URL，具体字段用的还是
前的引用。
        }
    }

    // registereUrls 是注册中心的URL
    for (URL u : urls) {
        u.addParameter(URLParamType.embed.getName(), StringTools.urlEncode(serviceUrl.toFu
lStr()));
        registereUrls.add(u.createCopy());
    }

    ConfigHandler configHandler = ExtensionLoader.getExtensionLoader(ConfigHandler.class).
etExtension(MotanConstants.DEFAULT_VALUE);

    // 到注册中心注册服务，urls是注册中心的地址
    exporters.add(configHandler.export(interfaceClass, ref, urls));
}
```

最后调用 configHandler.export 注册时，经过上面的解析过程，url的parameters参数中已经包含注册需要用到的信息，例如：

```
"path" -> "com.weibo.api.motan.registry.RegistryService"
"address" -> "192.168.103.254:2181"
"application" -> null
"name" -> "direct"
"connectTimeout" -> "3000"
"id" -> "ad-commonRegistryConfigBean"
"refreshTimestamp" -> "1571821250310"
"embed" -> "motan%3A%2F%2F192.168.100.14%3A8022%2Fcom.coohua.ad.common.remote
api.AdCommonRPC%3FhaStrategy%3Dfailover%26module%3Dad-common%26check%3Dfal
e%26nodeType%3Dservice%26version%3D1.1.0%26filter%3DcafTracing%2CpepperProfiler%
CsentinelProfiler%26minWorkerThread%3D20%26retries%3D1%26protocol%3Dmotan%26ap
lication%3Dad-common%26maxWorkerThread%3D200%26shareChannel%3Dtrue%26refresh
imestamp%3D1571821305290%26id%3Dad-commonBasicServiceConfigBean%26export%3D
d-commonProtocolConfigBean%3A8022%26requestTimeout%3D30000%26group%3Dad-co
mon%26"
"requestTimeout" -> "1000"
```

接下来看一下 configHandler.export 做了什么事情。

```
public <T> Exporter<T> export(Class<T> interfaceClass, T ref, List<URL> registryUrls) {
    // 解码url -> motan://192.168.100.14:8022/com.coohua.ad.common.remote.api.AdCommo
RPC?group=ad-common
    String serviceStr = StringTools.urlDecode(registryUrls.get(0).getParameter(URLParamType.
mbed.getName()));
    URL serviceUrl = URL.valueOf(serviceStr);

    // export service
    String protocolName = serviceUrl.getParameter(URLParamType.protocol.getName(), URLPa
amType.protocol.getValue());
    // SPI的方式拿到具体的Protocol实现，默认情况下拿到 motan 的 Protocol
    Protocol orgProtocol = ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(p
otocolName);
    Provider<T> provider = getProvider(orgProtocol, ref, serviceUrl, interfaceClass);

    Protocol protocol = new ProtocolFilterDecorator(orgProtocol);
    // 在这里走Motan的filter chain，并启动服务，filter chain通过调用 ProtocolFilterDecorator 的
ecorateWithFilter 方法实现
    // 走完filter chain后，会调用 orgProtocol 的 export 方法来暴露服务，这个方法的实现在 Abstra
tProtocol 类中
    Exporter<T> exporter = protocol.export(provider, serviceUrl);

    // 在注册中心中注册服务
    register(registryUrls, serviceUrl);

    return exporter;
}
```

在 AbstractProtocol 的 export 方法中会调用 createExporter 方法创建一个 Exporter 类的实例（体来说是 DefaultRpcExporter），在这个创建过程中会调用 NettyEndpointFactory 的 createServer 方法创建一个Server出来，并存放在exporter的server变量中。

然后调用 exporter 的 init 方法，在 init 方法中又调用了 doInit 方法，这个方法调用了 server.open() 至此，服务启动，并监听在本机指定的端口上。

```
@Override
protected boolean doInit() {
    boolean result = server.open();

    return result;
}
```

3 服务的注册

此时服务已经成功启动了，但还没注册到注册中心，所以还不能被发现。接下来，继续上面的代码，一下 register(registryUrls, serviceUrl); 这行代码干了啥。

此时两个参数的值分别是：

- registryUrls: zookeeper://192.168.103.254:2181/com.weibo.api.motan.registry.RegistryServi e?group=default_rpc
- serviceUrl: motan://192.168.100.14:8022/com.coohua.ad.common.remote.api.AdCommonR C?group=ad-common

这个方法的实现如下：

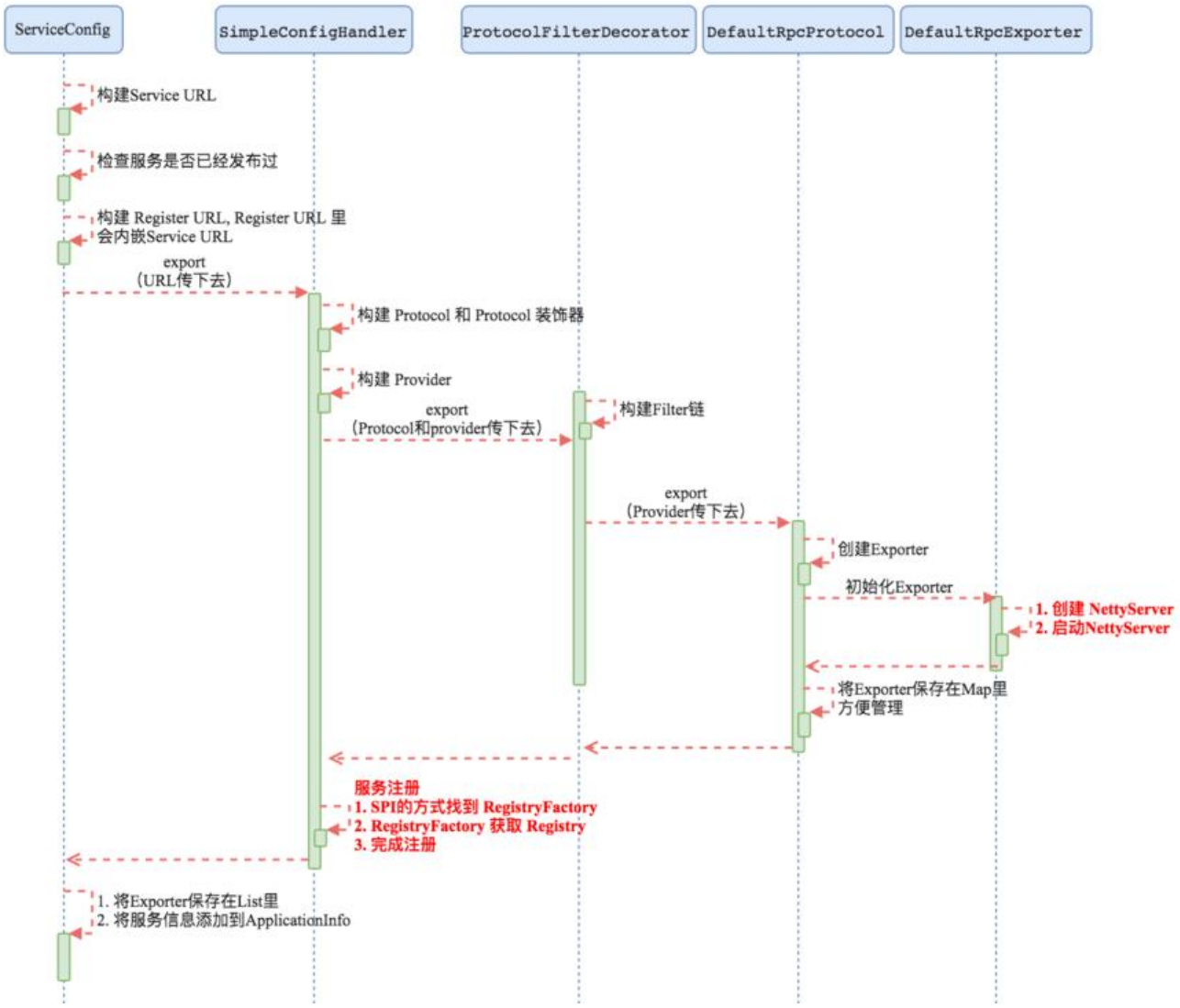
```
private void register(List<URL> registryUrls, URL serviceUrl) {

    for (URL url : registryUrls) {
        // 根据protocol的名称获取具体的 RegistryFactory ，这里以 zookeeper 为例
        RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.cl
ss).getExtension(url.getProtocol());
        if (registryFactory == null) {
            throw new MotanFrameworkException(new MotanErrorMsg(500, MotanErrorMsgCons
ant.FRAMEWORK_REGISTER_ERROR_CODE,
                    "register error! Could not find extension for registry protocol:" + url.getProtocol()
                            + ", make sure registry module for " + url.getProtocol() + " is in classpath!"));
        }
        // 尝试获取url对应registry已有的实例，如果没有，就创建一个
        // 这里zookeeper是用ZkClient管理的
        Registry registry = registryFactory.getRegistry(url);
        // 在zk中创建Node，完成服务的注册
        registry.register(serviceUrl);
    }
}
```

ZK中的注册结果：

```
[zk: localhost:2181(CONNECTED) 0] ls /motan/ad-common/com.coohua.ad.common.remote.
pi.AdCommonRPC/server
[192.168.100.14:8022]
```

至此，服务就可以被调用方发现了。

最后转载一张图总结一下上面的过程



图片来源：https://fdx321.github.io/2017/07/23/【Motan源码学习】4-服务发布和引用/