链滴

Motan_ 服务调用

作者：Lord-X

原文链接：https://ld246.com/article/1571974319555

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
- Motan心跳机制
- Motan负载均衡策略
- Motan高可用策略

---

0 @MotanReferer注解是个啥

被 @MotanReferer 标注的 setter 方法或 field 会被motan在启动时扫描，并为其创建动态代理，并动态代理的实例赋值给这个 field。远程服务的调用都是在这个代理中实现的。

下面以注解在 field 的情况为例，说明 @MotanReferer 的解析以及创建动态代理的过程：

```
@MotanReferer(basicReferer = "ad-commonBasicRefererConfigBean", application = "ad-filter
, version = "1.1.0")
private AdCommonRPC adCommonRPC;
```

关于如何扫描，在 Motan如何完成与Spring的集成 一文中有详细说明，这里不再赘述。

1 @MotanReferer注解的解析

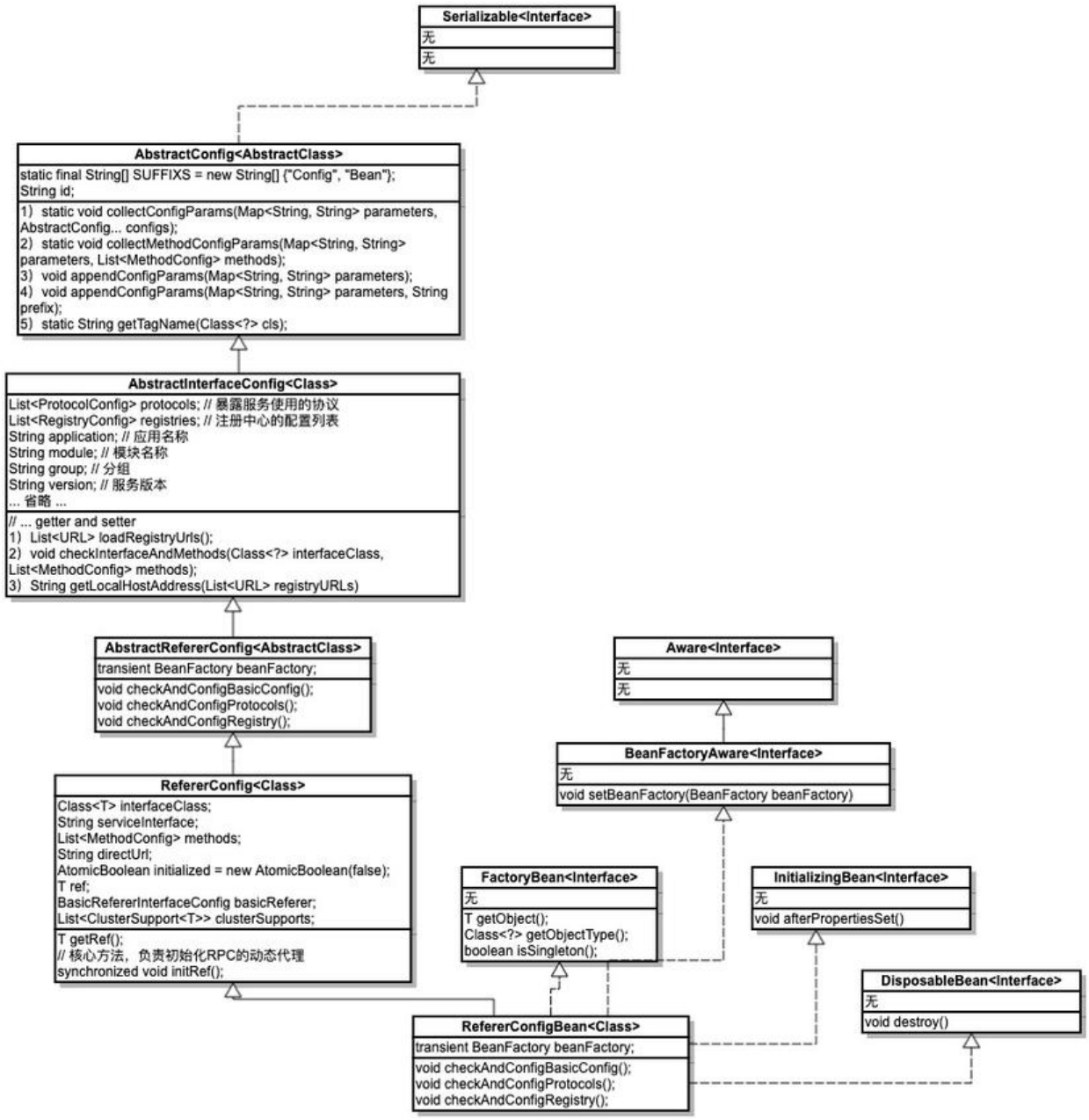
这个过程始于 AnnotationBean 中对扫描到的bean的field的解析。Motan会解析出带有 @MotanReferer 的field，应调用 AnnotationBean 的 refer 方法初始化并创建代理。field的解析如下：

```
Field[] fields = clazz.getDeclaredFields();
for (Field field : fields) {
    try {
        if (!field.isAccessible()) {
            field.setAccessible(true);
        }
        MotanReferer reference = field.getAnnotation(MotanReferer.class);
        if (reference != null) {
            // 调用 refer 方法初始化并创建动态代理
            // 并将field的引用指向这个代理对象
            Object value = refer(reference, field.getType());
            if (value != null) {
                field.set(bean, value);
            }
        }
    } catch (Throwable t) {
        throw new BeanInitializationException("Failed to init remote service reference at filed " +
field.getName()
                + " in class " + bean.getClass().getName(), t);
    }
}
```

2 @MotanReferer的初始化

类似于服务注册的过程，MotanReferer是通过 RefererConfigBean 类来管理配置、注册中心、URL HA、LoadBalance、Proxy等资源的。

还是先来个 RefererConfigBean 的UML，熟悉一下整个体系都有啥东西。



其中，注册中心、URL、Protocol、HA、LoadBalance策略等都是在 RefererConfig 的 clusterSupports 中管理的。

来继续看这个refer方法，这个方法中首先将 @MotanReferer 注解中的配置信息解析到 RefererConfigBean 中，然后依然是调用 afterPropertiesSet() 方法做一些校验，最后调用 RefererConfigBean 的 etRef() 方法，各个组件的初始化以及Proxy都在这里创建。

```
private <T> Object refer(MotanReferer reference, Class<?> referenceClass) {
    // 解析接口名
    String interfaceName;
    if (!void.class.equals(reference.interfaceClass())) {
        interfaceName = reference.interfaceClass().getName();
    } else if (referenceClass.isInterface()) {
        interfaceName = referenceClass.getName();
    } else {
```

```
        throw new IllegalStateException("The @Reference undefined interfaceClass or interface
ame, and the property type "
                + referenceClass.getName() + " is not a interface.");
    }
    String key = reference.group() + "/" + interfaceName + ":" + reference.version();
    RefererConfigBean<T> referenceConfig = referenceConfigs.get(key);
    if (referenceConfig == null) {
        referenceConfig = new RefererConfigBean<T>();
        referenceConfig.setBeanFactory(beanFactory);
        if (void.class.equals(reference.interfaceClass())
                && referenceClass.isInterface()) {
            referenceConfig.setInterface((Class<T>) referenceClass);
        } else if (!void.class.equals(reference.interfaceClass())) {
            referenceConfig.setInterface((Class<T>) reference.interfaceClass());
        }

        if (beanFactory != null) {
            // ... 省略，初始化 @MotanReferer的配置信息
            try {
                // 校验basicReferer配置、Protocol、Registry配置，与服务注册过程相似
                referenceConfig.afterPropertiesSet();
            } catch (RuntimeException e) {
                throw (RuntimeException) e;
            } catch (Exception e) {
                throw new IllegalStateException(e.getMessage(), e);
            }
        }
        referenceConfigs.putIfAbsent(key, referenceConfig);
        referenceConfig = referenceConfigs.get(key);
    }
    // 创建Proxy
    return referenceConfig.getRef();
}
```

getRef() 方法中实际是调用 initRef() 方法来创建Proxy的。

```
public synchronized void initRef() {
    // ... 校验 interface 和 protocols 是否非空
    checkInterfaceAndMethods(interfaceClass, methods);

    clusterSupports = new ArrayList<>(protocols.size());
    List<Cluster<T>> clusters = new ArrayList<>(protocols.size());
    String proxy = null;

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

    // 解析注册中心地址
    List<URL> registryUrls = loadRegistryUrls();
    // 解析本机IP
    String localIp = getLocalHostAddress(registryUrls);
    for (ProtocolConfig protocol : protocols) {
        Map<String, String> params = new HashMap<>();
        params.put(URLParamType.nodeType.getName(), MotanConstants.NODE_TYPE_REFERER)
```

```
        params.put(URLParamType.version.getName(), URLParamType.version.getValue());
        params.put(URLParamType.refreshTimestamp.getName(), String.valueOf(System.currentT
meMillis()));

        collectConfigParams(params, protocol, basicReferer, extConfig, this);
        collectMethodConfigParams(params, this.getMethods());

        String path = StringUtils.isBlank(serviceInterface) ? interfaceClass.getName() : serviceInter
ace;
        URL refUrl = new URL(protocol.getName(), localIp, MotanConstants.DEFAULT_INT_VALUE
 path, params);
        // 初始化ClusterSupport
        ClusterSupport<T> clusterSupport = createClusterSupport(refUrl, configHandler, registr
Urls);

        clusterSupports.add(clusterSupport);
        clusters.add(clusterSupport.getCluster());

        if (proxy == null) {
            // 获取创建proxy的方式，默认是JDK动态代理
            String defaultValue = StringUtils.isBlank(serviceInterface) ? URLParamType.proxy.getVa
ue() : MotanConstants.PROXY_COMMON;
            proxy = refUrl.getParameter(URLParamType.proxy.getName(), defaultValue);
        }
    }
    // 创建代理
    ref = configHandler.refer(interfaceClass, clusters, proxy);

    initialized.set(true);
}
```

可以发现，又调用了 configHandler.refer 方法，默认情况下，这个proxy参数的值是"jdk"，即使用JK自身的动态代理功能创建代理。

另外一个比较重要的类是 ClusterSupport，这个类封装了下面这些信息：

```
private static ConcurrentHashMap<String, Protocol> protocols = new ConcurrentHashMap<S
ring, Protocol>();
// 集群支持
private Cluster<T> cluster;
// 注册中心URL
private List<URL> registryUrls;
// 远程调用URL
private URL url;
private Class<T> interfaceClass;
// 使用的协议
private Protocol protocol;
private ConcurrentHashMap<URL, List<Referer<T>>> registryReferers = new ConcurrentHas
Map<URL, List<Referer<T>>>();
```

其中，cluster 在motan的具体实现实际上是 ClusterSpi 这个类，他封装了以下信息：

```
public class ClusterSpi<T> implements Cluster<T> {
    // 高可用策略
```

```
    private HaStrategy<T> haStrategy;
    // 负载均衡策略
    private LoadBalance<T> loadBalance;
    private List<Referer<T>> referers;
    private AtomicBoolean available = new AtomicBoolean(false);
    private URL url;
}
```

这些东西在上面的 createClusterSupport 方法中生成好了，这里先不关注Cluster、Ha、LoadBalance等，本文主要关注代理的创建及RPC调用。继续看 configHandler.refer 这个方法。

```
public <T> T refer(Class<T> interfaceClass, List<Cluster<T>> clusters, String proxyType) {
    ProxyFactory proxyFactory = ExtensionLoader.getExtensionLoader(ProxyFactory.class).getExtension(proxyType);
    return proxyFactory.getProxy(interfaceClass, clusters);
}
```

又是一个SPI扩展，默认使用的下面这个JDK的ProxyFactory。

```
@SpiMeta(name = "jdk")
public class JdkProxyFactory implements ProxyFactory {

    @Override
    @SuppressWarnings("unchecked")
    public <T> T getProxy(Class<T> clz, List<Cluster<T>> clusters) {
        return (T) Proxy.newProxyInstance(clz.getClassLoader(), new Class[]{clz}, new RefererInvocationHandler<>(clz, clusters));
    }
}
```

OK，到这里代理就创建好了。这个代理的实例最终会被 @MotanReferer 注解的field引用。初始化程结束。

这里要明确，最终是通过 RefererInvocationHandler 这个类创建的代理。

3 RPC调用

既然 RefererInvocationHandler 代理了我们的目标接口，那么接口的每个方法调用都会走到这个代类中。所以接下来主要关注代理是咋完成RPC调用的。

这里只给出关键代码：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // 省略 local method 部分

    DefaultRequest request = new DefaultRequest();
    request.setRequestId(RequestIdGenerator.getRequestId());
    request.setArguments(args);
    String methodName = method.getName();
    boolean async = false; // 异步调用支持，暂不关注
    if (methodName.endsWith(MotanConstants.ASYNC_SUFFIX) && method.getReturnType().equals(ResponseFuture.class)) {
        methodName = MotanFrameworkUtil.removeAsyncSuffix(methodName);
        async = true;
```

```
    }
    request.setMethodName(methodName);
    request.setParamtersDesc(ReflectUtil.getMethodParamDesc(method));
    request.setInterfaceName(interfaceName);

    return invokeRequest(request, getRealReturnType(async, this.clz, method, methodName), a
ync);
}
```

这个方法先将RPC的相关信息封装到 DefaultRequest 中，然后调用 invokeRequest 方法。

```
Object invokeRequest(Request request, Class returnType, boolean async) throws Throwable {
    RpcContext curContext = RpcContext.getContext();
    // 省略 初始化 RpcContext

    // 当 referer配置多个protocol的时候，比如A,B,C，
    // 那么正常情况下只会使用A，如果A被开关降级，那么就会使用B，B也被降级，那么会使用C
    for (Cluster<T> cluster : clusters) {
        // 如果开关处于关闭状态，不会去调用这个远程机器
        String protocolSwitcher = MotanConstants.PROTOCOL_SWITCHER_PREFIX + cluster.getU
l().getProtocol();
        Switcher switcher = switcherService.getSwitcher(protocolSwitcher);
        if (switcher != null && !switcher.isOn()) {
            continue;
        }

        request.setAttachment(URLParamType.version.getName(), cluster.getUrl().getVersion());
        request.setAttachment(URLParamType.clientGroup.getName(), cluster.getUrl().getGroup(
);
        // 带上client的application和module
        request.setAttachment(URLParamType.application.getName(), cluster.getUrl().getApplicat
on());
        request.setAttachment(URLParamType.module.getName(), cluster.getUrl().getModule());

        Response response = null;
        boolean throwException = Boolean.parseBoolean(cluster.getUrl().getParameter(URLPara
Type.throwException.getName(), URLParamType.throwException.getValue()));
        try {
            MotanFrameworkUtil.logEvent(request, MotanConstants.TRACE_INVOKE);
            // 执行调用
            response = cluster.call(request);
            if (async) {
                // 省略异步调用的支持
            } else {
                Object value = response.getValue();
                if (value != null && value instanceof DeserializableObject) {
                    try {
                        value = ((DeserializableObject) value).deserialize(returnType);
                    } catch (IOException e) {
                        LoggerUtil.error("deserialize response value fail! deserialize type:" + returnType
 e);
                        throw new MotanFrameworkException("deserialize return value fail! deserialize
type:" + returnType, e);
                    }
```

```
                }
                return value;
            }
        } catch (RuntimeException e) {
            // 异常处理，包括处理是否向上游服务抛出
        }
    }
    throw new MotanServiceException("Referer call Error: cluster not exist, interface=" + interfa
eName + " " + MotanFrameworkUtil.toString(request), MotanErrorMsgConstant.SERVICE_UN
OUND);
}
```

cluster.call()

```
public Response call(Request request) {
    if (available.get()) {
        try {
            // haStrategy是通过SPI来管理的，默认的HA策略是 failover
            // 即调用失败时，自动尝试其他服务器
            return haStrategy.call(request, loadBalance);
        } catch (Exception e) {
            return callFalse(request, e);
        }
    }
    return callFalse(request, new MotanServiceException(MotanErrorMsgConstant.SERVICE_UN
OUND));
}
```

haStrategy.call()

```
public Response call(Request request, LoadBalance<T> loadBalance) {
    // refer列表
    List<Referer<T>> referers = selectReferers(request, loadBalance);
    if (referers.isEmpty()) {
        throw new MotanServiceException(String.format("FailoverHaStrategy No referers for req
est:%s, loadbalance:%s", request,
                loadBalance));
    }
    URL refUrl = referers.get(0).getUrl();
    // 这里是配置中配置的 retries 重试次数，默认：0
    int tryCount =
            refUrl.getMethodParameter(request.getMethodName(), request.getParamtersDesc(),
RLParamType.retries.getName(),
                    URLParamType.retries.getIntValue());
    // 如果有问题，则设置为不重试
    if (tryCount < 0) {
        tryCount = 0;
    }

    for (int i = 0; i <= tryCount; i++) {
        Referer<T> refer = referers.get(i % referers.size());
        try {
            request.setRetries(i);
            return refer.call(request); // RPC
```

```
        } catch (RuntimeException e) {
            // 对于业务异常，直接抛出
            if (ExceptionUtil.isBizException(e)) {
                throw e;
            } else if (i >= tryCount) {
                throw e;
            }
            LoggerUtil.warn(String.format("FailoverHaStrategy Call false for request:%s error=%s",
equest, e.getMessage()));
        }
    }

    throw new MotanFrameworkException("FailoverHaStrategy.call should not come here!");
}
```

然后，refer.call()

```
public Response call(Request request) {
    if (!isAvailable()) {
        throw new MotanFrameworkException(this.getClass().getSimpleName() + " call Error: no
e is not available, url=" + url.getUri()
                + " " + MotanFrameworkUtil.toString(request));
    }
    // 增加目标server的连接数，用于loadBalance
    incrActiveCount(request);
    Response response = null;
    try {
        response = doCall(request); // do rpc
        return response;
    } finally {
        // 调用完要将目标server的连接数-1
        decrActiveCount(request, response);
    }
}
```

到这里，doCall 方法就是通过Netty调用RPC了。