crazyhzm 265

Dubbo源码解析（三十）远程调用——rest协议

dubbo java 阅读约 44 分钟

远程调用——rest协议

> 目标：介绍rest协议的设计和实现，介绍dubbo-rpc-rest的源码。

前言

REST的英文名是RepresentationalState Transfer，它是一种开发风格，关于REST不清楚的朋友可以了解一下。在dubbo中利用的是红帽子RedHat公司的Resteasy来使dubbo支持REST风格的开发使用。在本文中主要讲解的是基于Resteasy来实现rest协议的实现。

源码分析

（一）RestServer

该接口是rest协议的服务器接口。定义了服务器相关的方法。

```
public interface RestServer {

    /**
     * 服务器启动
     * @param url
     */
    void start(URL url);

    /**
     * 部署服务器
     * @param resourceDef it could be either resource interface or resource impl
     */
    void deploy(Class resourceDef, Object resourceInstance, String contextPath);

    /**
     * 取消服务器部署
     * @param resourceDef
     */
    void undeploy(Class resourceDef);

    /**
     * 停止服务器
     */
    void stop();
}
```

（二）BaseRestServer

该类实现了RestServer接口，是rest服务的抽象类，把getDeployment和doStart方法进行抽象，让子类专注于中这两个方法的实现。

1.start

```
    @Override
    public void start(URL url) {
        // 支持两种 Content-Type
        getDeployment().getMediaTypeMappings().put("json", "application/json");
        getDeployment().getMediaTypeMappings().put("xml", "text/xml");
//        server.getDeployment().getMediaTypeMappings().put("xml", "application/xml");
        // 添加拦截器
        getDeployment().getProviderClasses().add(RpcContextFilter.class.getName());
        // TODO users can override this mapper, but we just rely on the current priority strategy of resteasy
        // 异常类映射
        getDeployment().getProviderClasses().add(RpcExceptionMapper.class.getName());

        // 添加需要加载的类
        loadProviders(url.getParameter(Constants.EXTENSION_KEY, ""));

        // 开启服务器
        doStart(url);
    }
```

2.deploy

```
@Override
public void deploy(Class resourceDef, Object resourceInstance, String contextPath) {
    // 如果
    if (StringUtils.isEmpty(contextPath)) {
        // 添加自定义资源实现端点，部署服务器
        getDeployment().getRegistry().addResourceFactory(new DubboResourceFactory(resourceInstance, resourceDef));
    } else {
        // 添加自定义资源实现端点。指定contextPath
        getDeployment().getRegistry().addResourceFactory(new DubboResourceFactory(resourceInstance, resourceDef),
contextPath);
    }
}
```

3.undeploy

```
@Override
public void undeploy(Class resourceDef) {
    // 取消服务器部署
    getDeployment().getRegistry().removeRegistrations(resourceDef);
}
```

4.loadProviders

```
protected void loadProviders(String value) {
    for (String clazz : Constants.COMMA_SPLIT_PATTERN.split(value)) {
        if (!StringUtils.isEmpty(clazz)) {
            getDeployment().getProviderClasses().add(clazz.trim());
        }
    }
}
```

该方法是把类都加入到ResteasyDeployment的providerClasses中，加入各类组件。

（三）DubboHttpServer

该类继承了BaseRestServer，实现了doStart和getDeployment方法，当配置选择servlet、jetty或者tomcat作为远程通信的实现时，实现的服务器类

1.属性

```
/**
 * HttpServletDispatcher实例
 */
private final HttpServletDispatcher dispatcher = new HttpServletDispatcher();
/**
 * Resteasy的服务部署器
 */
private final ResteasyDeployment deployment = new ResteasyDeployment();
/**
 * http绑定者
 */
private HttpBinder httpBinder;
/**
 * http服务器
 */
private HttpServer httpServer;
```

2.doStart

```
@Override
protected void doStart(URL url) {
    // TODO jetty will by default enable keepAlive so the xml config has no effect now
    // 创建http服务器
    httpServer = httpBinder.bind(url, new RestHandler());

    // 获得ServletContext
    ServletContext servletContext = ServletManager.getInstance().getServletContext(url.getPort());
    // 如果为空 ， 则获得默认端口对应的ServletContext对象
    if (servletContext == null) {
        servletContext = ServletManager.getInstance().getServletContext(ServletManager.EXTERNAL_SERVER_PORT);
    }
    // 如果还是为空 ， 则抛出异常
    if (servletContext == null) {
        throw new RpcException("No servlet context found. If you are using server='servlet', " +
                "make sure that you've configured " + BootstrapListener.class.getName() + " in web.xml");
    }

    // 设置属性部署器
    servletContext.setAttribute(ResteasyDeployment.class.getName(), deployment);

    try {
        // 初始化
        dispatcher.init(new SimpleServletConfig(servletContext));
    } catch (ServletException e) {
        throw new RpcException(e);
```

3.RestHandler

```
private class RestHandler implements HttpHandler {

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException
{
        // 设置远程地址
        RpcContext.getContext().setRemoteAddress(request.getRemoteAddr(), request.getRemotePort());
        // 请求相关的服务
        dispatcher.service(request, response);
    }
}
```

该内部类是服务请求的处理器

4.SimpleServletConfig

```
private static class SimpleServletConfig implements ServletConfig {

    // ServletContext对象
    private final ServletContext servletContext;

    public SimpleServletConfig(ServletContext servletContext) {
        this.servletContext = servletContext;
    }

    @Override
    public String getServletName() {
        return "DispatcherServlet";
    }

    @Override
    public ServletContext getServletContext() {
        return servletContext;
    }

    @Override
    public String getInitParameter(String s) {
        return null;
    }

    @Override
    public Enumeration getInitParameterNames() {
```

该内部类是配置类。

（四）NettyServer

该类继承了BaseRestServer，当配置了netty作为远程通信的实现时，实现的服务器。

```
public class NettyServer extends BaseRestServer {

    /**
     * NettyJaxrsServer对象
     */
    private final NettyJaxrsServer server = new NettyJaxrsServer();

    @Override
    protected void doStart(URL url) {
        // 获得ip
        String bindIp = url.getParameter(Constants.BIND_IP_KEY, url.getHost());
        if (!url.isAnyHost() && NetUtils.isValidLocalHost(bindIp)) {
            // 设置服务的ip
            server.setHostname(bindIp);
        }
        // 设置端口
        server.setPort(url.getParameter(Constants.BIND_PORT_KEY, url.getPort()));
        // 通道选项集合
        Map<ChannelOption, Object> channelOption = new HashMap<ChannelOption, Object>();
        // 保持连接检测对方主机是否崩溃
        channelOption.put(ChannelOption.SO_KEEPALIVE, url.getParameter(Constants.KEEP_ALIVE_KEY,
Constants.DEFAULT_KEEP_ALIVE));
        // 设置配置
        server.setChildChannelOptions(channelOption);
        // 设置线程数，默认为200
        server.setExecutorThreadCount(url.getParameter(Constants.THREADS_KEY, Constants.DEFAULT_THREADS));
```

（五）DubboResourceFactory

该类实现了ResourceFactory接口，是资源工程实现类，封装了以下两个属性，实现比较简单。

```
/**
 * 资源类
 */
private Object resourceInstance;
/**
 * 扫描的类型
 */
private Class scannableClass;
```

（六）RestConstraintViolation

该类是当约束违反的实体类，封装了以下三个属性，具体使用可以看下面的介绍。

```
/**
 * 地址
 */
private String path;
/**
 * 消息
 */
private String message;
/**
 * 值
 */
private String value;
```

（七）RestServerFactory

该类是服务器工程类，用来提供相应的实例，里面逻辑比较简单。

```
public class RestServerFactory {

    /**
     * http绑定者
     */
    private HttpBinder httpBinder;

    public void setHttpBinder(HttpBinder httpBinder) {
        this.httpBinder = httpBinder;
    }

    /**
     * 创建服务器
     * @param name
     * @return
     */
    public RestServer createServer(String name) {
        // TODO move names to Constants
        // 如果是servlet或者jetty或者tomcat，则创建DubboHttpServer
        if ("servlet".equalsIgnoreCase(name) || "jetty".equalsIgnoreCase(name) ||
"tomcat".equalsIgnoreCase(name)) {
            return new DubboHttpServer(httpBinder);
        } else if ("netty".equalsIgnoreCase(name)) {
            // 如果是netty，那么直接创建netty服务器
            return new NettyServer();
        } else {
```

可以看到，根据配置的不同，来创建不同的服务器实现。

（八）RpcContextFilter

该类是过滤器。增加了对协议头大小的限制。

```
public class RpcContextFilter implements ContainerRequestFilter, ClientRequestFilter {

    /**
     * 附加值key
     */
    private static final String DUBBO_ATTACHMENT_HEADER = "Dubbo-Attachments";

    // currently we use a single header to hold the attachments so that the total attachment size limit is about
8k
    /**
     * 目前我们使用单个标头来保存附件，以便总附件大小限制大约为8k
     */
    private static final int MAX_HEADER_SIZE = 8 * 1024;

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        // 获得request
        HttpServletRequest request = ResteasyProviderFactory.getContextData(HttpServletRequest.class);
        // 把它放到rpc上下文中
        RpcContext.getContext().setRequest(request);

        // this only works for servlet containers
        if (request != null && RpcContext.getContext().getRemoteAddress() == null) {
            // 设置远程地址
            RpcContext.getContext().setRemoteAddress(request.getRemoteAddr(), request.getRemotePort());
        }
```

可以看到有两个filter的方法实现，第一个是解析对于附加值，并且放入上下文中。第二个是对协议头大小的限制。

（九）RpcExceptionMapper

该类是异常的处理类。

```
public class RpcExceptionMapper implements ExceptionMapper<RpcException> {

    @Override
    public Response toResponse(RpcException e) {
        // TODO do more sophisticated exception handling and output
        // 如果是约束违反异常
        if (e.getCause() instanceof ConstraintViolationException) {
            return handleConstraintViolationException((ConstraintViolationException) e.getCause());
        }
        // we may want to avoid exposing the dubbo exception details to certain clients
        // TODO for now just do plain text output
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("Internal server error: " +
e.getMessage()).type(ContentType.TEXT_PLAIN_UTF_8).build();
    }

    /**
     * 处理参数不合法的异常
     * @param cve
     * @return
     */
    protected Response handleConstraintViolationException(ConstraintViolationException cve) {
        // 创建约束违反记录
        ViolationReport report = new ViolationReport();
        // 遍历约束违反
        for (ConstraintViolation cv : cve.getConstraintViolations()) {
            // 添加记录
```

主要是处理参数不合法的异常。

（十）ViolationReport

该类是约束违反的记录类，其中就封装了一个约束违反的集合。

```java
public class ViolationReport implements Serializable {

    private static final long serialVersionUID = -130498234L;

    /**
     * 约束违反集合
     */
    private List<RestConstraintViolation> constraintViolations;

    public List<RestConstraintViolation> getConstraintViolations() {
        return constraintViolations;
    }

    public void setConstraintViolations(List<RestConstraintViolation> constraintViolations) {
        this.constraintViolations = constraintViolations;
    }

    public void addConstraintViolation(RestConstraintViolation constraintViolation) {
        if (constraintViolations == null) {
            constraintViolations = new LinkedList<RestConstraintViolation>();
        }
        constraintViolations.add(constraintViolation);
    }
}
```

（十一）RestProtocol

该类继承了AbstractProxyProtocol，是rest协议实现的核心。

1.属性

```java
/**
 * 默认端口号
 */
private static final int DEFAULT_PORT = 80;

/**
 * 服务器集合
 */
private final Map<String, RestServer> servers = new ConcurrentHashMap<String, RestServer>();

/**
 * 服务器工厂
 */
private final RestServerFactory serverFactory = new RestServerFactory();

// TODO in the future maybe we can just use a single rest client and connection manager
/**
 * 客户端集合
 */
private final List<ResteasyClient> clients = Collections.synchronizedList(new LinkedList<ResteasyClient>());

/**
 * 连接监控
 */
private volatile ConnectionMonitor connectionMonitor;
```

2.doExport

```
@Override
protected <T> Runnable doExport(T impl, Class<T> type, URL url) throws RpcException {
    // 获得地址
    String addr = getAddr(url);
    // 获得实现类
    Class implClass = (Class) StaticContext.getContext(Constants.SERVICE_IMPL_CLASS).get(url.getServiceKey());
    // 获得服务
    RestServer server = servers.get(addr);
    if (server == null) {
        // 创建服务器
        server = serverFactory.createServer(url.getParameter(Constants.SERVER_KEY, "jetty"));
        // 开启服务器
        server.start(url);
        // 加入集合
        servers.put(addr, server);
    }

    // 获得contextPath
    String contextPath = getContextPath(url);
    // 如果以servlet的方式
    if ("servlet".equalsIgnoreCase(url.getParameter(Constants.SERVER_KEY, "jetty"))) {
        // 获得ServletContext
        ServletContext servletContext =
ServletManager.getInstance().getServletContext(ServletManager.EXTERNAL_SERVER_PORT);
        // 如果为空，则抛出异常
        if (servletContext == null) {
```

该方法是服务暴露的方法。

3.doRefer

```
protected <T> T doRefer(Class<T> serviceType, URL url) throws RpcException {
    // 如果连接监控为空，则创建
    if (connectionMonitor == null) {
        connectionMonitor = new ConnectionMonitor();
    }

    // TODO more configs to add
    // 创建http连接池
    PoolingHttpClientConnectionManager connectionManager = new PoolingHttpClientConnectionManager();
    // 20 is the default maxTotal of current PoolingClientConnectionManager
    // 最大连接数
    connectionManager.setMaxTotal(url.getParameter(Constants.CONNECTIONS_KEY, 20));
    // 最大的路由数
    connectionManager.setDefaultMaxPerRoute(url.getParameter(Constants.CONNECTIONS_KEY, 20));

    // 添加监控
    connectionMonitor.addConnectionManager(connectionManager);
    // 新建请求配置
    RequestConfig requestConfig = RequestConfig.custom()
            .setConnectTimeout(url.getParameter(Constants.CONNECT_TIMEOUT_KEY,
Constants.DEFAULT_CONNECT_TIMEOUT))
            .setSocketTimeout(url.getParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT))
            .build();

    // 设置socket配置
    SocketConfig socketConfig = SocketConfig.custom()
```

该方法是服务引用的实现。

4.ConnectionMonitor

```
protected class ConnectionMonitor extends Thread {
    /**
     * 是否关闭
     */
    private volatile boolean shutdown;
    /**
     * 连接池集合
     */
    private final List<PoolingHttpClientConnectionManager> connectionManagers = Collections.synchronizedList(new
LinkedList<PoolingHttpClientConnectionManager>());

    public void addConnectionManager(PoolingHttpClientConnectionManager connectionManager) {
        connectionManagers.add(connectionManager);
    }

    @Override
    public void run() {
        try {
            while (!shutdown) {
                synchronized (this) {
                    wait(1000);
                    for (PoolingHttpClientConnectionManager connectionManager : connectionManagers) {
                        // 关闭池中所有过期的连接
                        connectionManager.closeExpiredConnections();
                        // TODO constant
                        // 关闭池中的空闲连接
```

该内部类是处理连接的监控类，当连接过期获取空间的时候，关闭它。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了远程调用中关于rest协议实现的部分，关键是要对Resteasy的使用需要有所了解，其他的思路跟其他协议实现差距不大。接下来我将开始对rpc模块关于rmi协议部分进行讲解。

crazyhzm
265