

Dubbo源码解析（十四）远程通信——Http

 [java](#) [dubbo](#) [http](#) [tomcat](#) [jetty](#) 阅读约 28 分钟

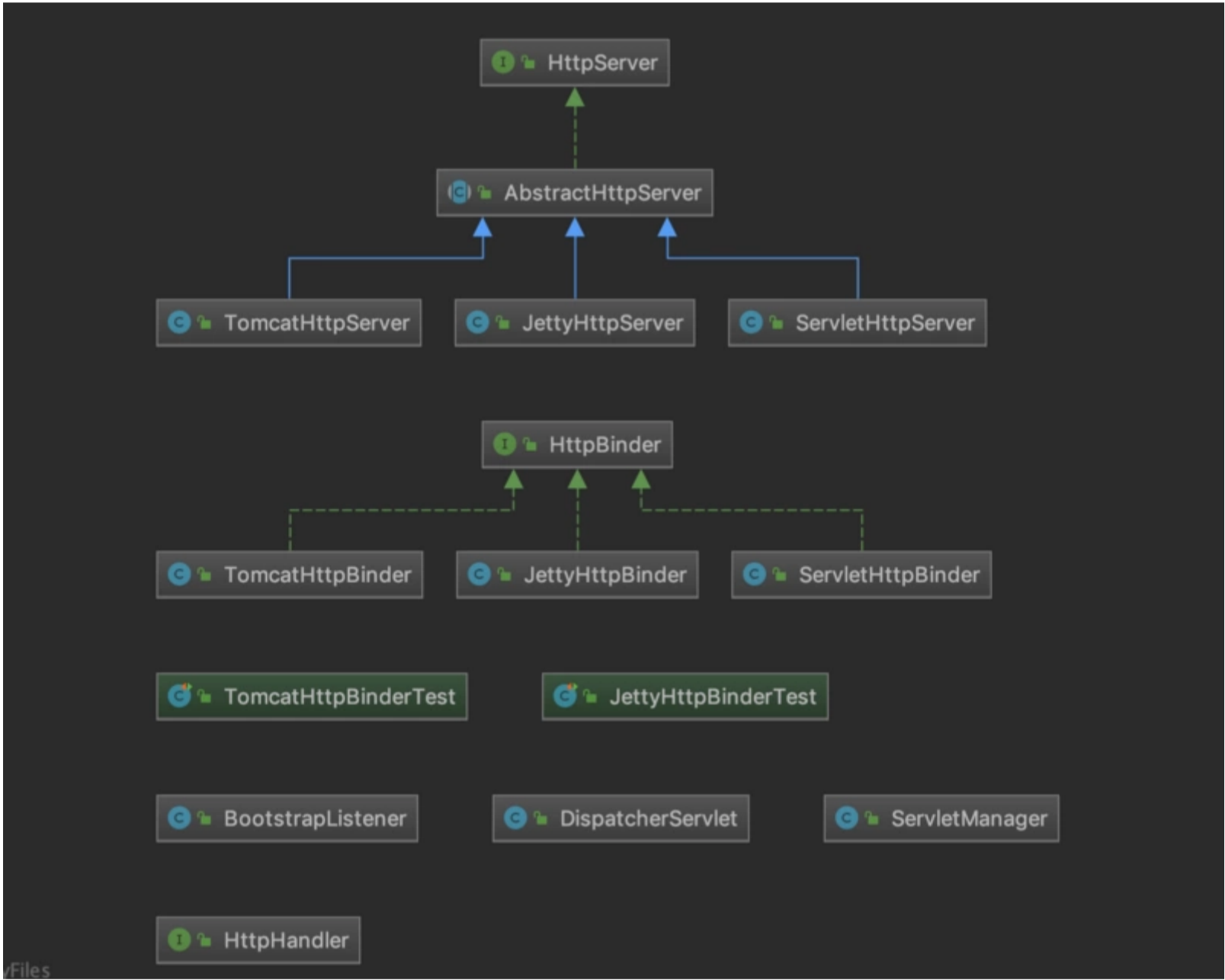
远程通讯——Http

目标：介绍基于Http的来实现的远程通信、介绍dubbo-remoting-http内的源码解析。

前言

本文我们讲解的是如何基于Tomcat或者Jetty实现HTTP服务器。Tomcat和Jetty都是一种servlet引擎，Jetty要比Tomcat的架构更简单一些。关于它们之间的比较，我觉得google一些更加方便，我就不多废话了。

下面是dubbo-remoting-http包下的类图：



可以看到这个包下只提供了服务端实现，并没有客户端实现。

源码分析

（一）HttpServer

```
public interface HttpServer extends Resetable {

    /**
     * get http handler.
     * 获得http的处理类
     * @return http handler.
     */
    HttpHandler getHttpHandler();

    /**
     * get url.
     * 获得url
     * @return url
     */
    URL getUrl();

    /**
     * get local address.
     * 获得本地服务器地址
     * @return local address.
     */
    InetSocketAddress getLocalAddress();

    /**
     * close the channel.
     * 关闭通道
     */
}
```

该接口是http服务器的接口，定义了服务器相关的方法，都比较好理解。

（二）AbstractHttpServer

该类实现接口HttpServer，是http服务器接口的抽象类。

```
/**
 * url
 */
private final URL url;

/**
 * http服务器处理器
 */
private final HttpHandler handler;

/**
 * 该服务器是否关闭
 */
private volatile boolean closed;

public AbstractHttpServer(URL url, HttpHandler handler) {
    if (url == null) {
        throw new IllegalArgumentException("url == null");
    }
    if (handler == null) {
        throw new IllegalArgumentException("handler == null");
    }
    this.url = url;
    this.handler = handler;
}
```

该类中有三个属性，关键是看在后面怎么用到，因为该类是抽象类，所以该类中的方法并没有实现具体的逻辑，我们继续往下看它的三个子类。

（三）TomcatHttpServer

该类是基于Tomcat来实现服务器的实现类，它继承了AbstractHttpServer。

1.属性

```
/**
 * 内嵌的tomcat对象
 */
private final Tomcat tomcat;

/**
 * url对象
 */
private final URL url;
```

该类的两个属性，关键是内嵌的tomcat。

2.构造方法

```
public TomcatHttpServer(URL url, final HttpHandler handler) {
    super(url, handler);

    this.url = url;
    // 添加处理器
    DispatcherServlet.addHttpHandler(url.getPort(), handler);
    // 获得java.io.tmpdir的绝对路径目录
    String baseDir = new File(System.getProperty("java.io.tmpdir")).getAbsolutePath();
    // 创建内嵌的tomcat对象
    tomcat = new Tomcat();
    // 设置根目录
    tomcat.setBaseDir(baseDir);
    // 设置端口号
    tomcat.setPort(url.getPort());
    // 给默认的http连接器。设置最大线程数
    tomcat.getConnector().setProperty(
        "maxThreads", String.valueOf(url.getParameter(Constants.THREADS_KEY,
            Constants.DEFAULT_THREADS)));
    // tomcat.getConnector().setProperty(
    // "minSpareThreads", String.valueOf(url.getParameter(Constants.THREADS_KEY,
    Constants.DEFAULT_THREADS)));

    // 设置最大的连接数
    tomcat.getConnector().setProperty(
        "maxConnections", String.valueOf(url.getParameter(Constants.ACCEPTS_KEY, -1)));
}
```

该方法的构造函数中就启动了tomcat，前面很多都是对tomcat的启动参数以及配置设置。如果有过tomcat配置经验的朋友应该看起来很简单。

3.close

```
@Override
public void close() {
    super.close();

    // 移除相关的servlet上下文
    ServletManager.getInstance().removeServletContext(url.getPort());

    try {
        // 停止tomcat
        tomcat.stop();
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
}
```

该方法是关闭服务器的方法。调用了tomcat.stop

（四）JettyHttpServer

该类是基于Jetty来实现服务器的实现类，它继承了AbstractHttpServer。

1.属性

```
/**
 * 内嵌的Jetty服务器对象
 */
private Server server;

/**
 * url 对象
 */
private URL url;
```

该类的两个属性，关键是内嵌的sever，它是内嵌的etty服务器对象。

2.构造方法

```
public JettyHttpServer(URL url, final HttpHandler handler) {
    super(url, handler);
    this.url = url;
    // TODO we should leave this setting to slf4j
    // we must disable the debug logging for production use
    // 设置日志
    Log.setLog(new StdErrLog());
    // 禁用调试用的日志
    Log.getLog().setDebugEnabled(false);

    // 添加http服务器处理器
    DispatcherServlet.addHttpHandler(url.getParameter(Constants.BIND_PORT_KEY, url.getPort()), handler);

    // 获得线程数
    int threads = url.getParameter(Constants.THREADS_KEY, Constants.DEFAULT_THREADS);
    // 创建线程池
    QueuedThreadPool threadPool = new QueuedThreadPool();
    // 设置线程池配置
    threadPool.setDaemon(true);
    threadPool.setMaxThreads(threads);
    threadPool.setMinThreads(threads);

    // 创建选择NIO连接器
    SelectChannelConnector connector = new SelectChannelConnector();

    // 获得绑定的ip
```

可以看到它跟TomcatHttpServer中构造函数不同的是API的不同，不过思路差不多，先设置启动参数和配置，然后启动jetty服务器。

3.close

```
@Override
public void close() {
    super.close();

    // 移除 ServletContext 对象
    ServletManager.getInstance().removeServletContext(url.getParameter(Constants.BIND_PORT_KEY, url.getPort()));

    if (server != null) {
        try {
            // 停止服务器
            server.stop();
        } catch (Exception e) {
            logger.warn(e.getMessage(), e);
        }
    }
}
```

该方法是关闭服务器的方法，调用的是server的stop方法。

（五）ServletHttpServer

该类继承了AbstractHttpServer，是基于Servlet的服务器实现类。

```
public class ServletHttpServer extends AbstractHttpServer {

    public ServletHttpServer(URL url, HttpHandler handler) {
        super(url, handler);
        // /把 HttpHandler 到 DispatcherServlet 中，默认端口为8080
        DispatcherServlet.addHttpHandler(url.getParameter(Constants.BIND_PORT_KEY, 8080), handler);
    }

}
```

该类就一个构造方法。就是把服务器处理器注册到DispatcherServlet上。

（六）HttpBinder

```
@SPI("jetty")
public interface HttpBinder {

    /**
     * bind the server.
     * 绑定到服务器
     * @param url server url.
     * @return server.
     */
    @Adaptive({Constants.SERVER_KEY})
    HttpServer bind(URL url, HttpHandler handler);

}
```

该接口是http绑定器接口，其中就定义了一个方法就是绑定方法，并且返回服务器对象。该接口是一个可扩展接口，默认实现JettyHttpBinder。它有三个实现类，请往下看。

（七）TomcatHttpBinder

```
public class TomcatHttpBinder implements HttpBinder {

    @Override
    public HttpServer bind(URL url, HttpHandler handler) {
        // 创建一个TomcatHttpServer
        return new TomcatHttpServer(url, handler);
    }

}
```

一眼就看出来，就是创建了个基于Tomcat实现的服务器TomcatHttpServer对象。具体的就往上看TomcatHttpServer里面的实现。

（八）JettyHttpBinder

```
public class JettyHttpBinder implements HttpBinder {

    @Override
    public HttpServer bind(URL url, HttpHandler handler) {
        // 创建JettyHttpServer实例
        return new JettyHttpServer(url, handler);
    }

}
```

一眼就看出来，就是创建了个基于Jetty实现的服务器JettyHttpServer对象。具体的就往上看JettyHttpServer里面的实现。

(九) ServletHttpBinder

```
public class ServletHttpBinder implements HttpBinder {

    @Override
    @Adaptive()
    public HttpServer bind(URL url, HttpHandler handler) {
        // 创建ServletHttpServer对象
        return new ServletHttpServer(url, handler);
    }

}
```

创建了一个基于servlet实现的服务器ServletHttpServer对象。并且方法上加入了Adaptive，用到了dubbo SPI机制。

(十) BootstrapListener

```
public class BootstrapListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        // context创建的时候，把ServletContext添加到ServletManager
        ServletManager.getInstance().addServletContext(ServletManager.EXTERNAL_SERVER_PORT,
servletContextEvent.getServletContext());
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        // context销毁到时候，把servletContextEvent移除
        ServletManager.getInstance().removeServletContext(ServletManager.EXTERNAL_SERVER_PORT);
    }

}
```

该类实现了ServletContextListener，是启动监听器，当context创建和销毁的时候对ServletContext做处理。不过需要配置BootstrapListener到web.xml，通过这样的方式，让外部的ServletContext对象，添加到ServletManager中。

(十一) DispatcherServlet

```
public class DispatcherServlet extends HttpServlet {

    private static final long serialVersionUID = 5766349180380479888L;
    /**
     * http服务器处理器
     */
    private static final Map<Integer, HttpHandler> handlers = new ConcurrentHashMap<Integer, HttpHandler>();
    /**
     * 单例
     */
    private static DispatcherServlet INSTANCE;

    public DispatcherServlet() {
        DispatcherServlet.INSTANCE = this;
    }

    /**
     * 添加处理器
     * @param port
     * @param processor
     */
    public static void addHttpHandler(int port, HttpHandler processor) {
        handlers.put(port, processor);
    }

    public static void removeHttpHandler(int port) {
```

该类继承了HttpServlet，是服务请求调度servlet类，主要是service方法，根据请求来调度不同的处理器去处理请求，如果没有该处理器，则报错404

(十二) ServletManager

```
public class ServletManager {

    /**
     * 外部服务器端口, 用于 `servlet` 的服务器端口
     */
    public static final int EXTERNAL_SERVER_PORT = -1234;

    /**
     * 单例
     */
    private static final ServletManager instance = new ServletManager();

    /**
     * ServletContext 集合
     */
    private final Map<Integer, ServletContext> contextMap = new ConcurrentHashMap<Integer, ServletContext>();

    public static ServletManager getInstance() {
        return instance;
    }

    /**
     * 添加ServletContext
     * @param port
     * @param servletContext
     */
}
```

该类是servlet的管理器，管理着ServletContext。

(十三) HttpHandler

```
public interface HttpHandler {

    /**
     * invoke.
     * HTTP 请求处理
     * @param request request.
     * @param response response.
     * @throws IOException
     * @throws ServletException
     */
    void handle(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException;

}
```

该接口是HTTP 处理器接口，就定义了一个处理请求的方法。

后记

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

该文章讲解了内嵌tomcat和jetty的来实现的http服务器，关键需要对tomcat和jetty的配置有所了解。下一篇我会讲解基于mina实现远程通信部分。

阅读 530 • 更新于 11月8日