链滴

HttpClient4.3 教程 第二章 连接管理

作者：kenan

原文链接：https://ld246.com/article/1497843202457

来源网站：链滴

<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color
#FCFCFC;">
2.1.持久连接
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
两个主机建立连接的过程是很复杂的一个过程，涉及到多个数据包的交换，并且也很耗时间。Http
接需要的三次握手开销很大，这一开销对于比较小的http消息来说更大。但是如果我们直接使用已经
立好的http连接，这样花费就比较小，吞吐率更大。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
HTTP/1.1默认就支持Http连接复用。兼容HTTP/1.0的终端也可以通过声明来保持连接，实现连接
用。HTTP代理也可以在一定时间内保持连接不释放，方便后续向这个主机发送http请求。这种保持
接不释放的情况实际上是建立的持久连接。HttpClient也支持持久连接。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">

</p>
<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color
#FCFCFC;">
2.2.HTTP连接路由<span style="font-family:inherit;font-style:inherit;font-weight:inherit;verti
al-align:baseline;">
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
HttpClient既可以直接、又可以通过多个中转路由（hops）和目标服务器建立连接。HttpClient把
由分为三种plain（明文 ），tunneled（隧道）和layered（分层）。隧道连接中使用的多个中间代
被称作代理链。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
客户端直接连接到目标主机或者只通过了一个中间代理，这种就是Plain路由。客户端通过第一个
理建立连接，通过代理链tunnelling，这种情况就是Tunneled路由。不通过中间代理的路由不可能时t
nneled路由。客户端在一个已经存在的连接上进行协议分层，这样建立起来的路由就是layered路由
协议只能在隧道—>目标主机，或者直接连接（没有代理），这两种链路上进行分层。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">

</p>
<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertica
-align:baseline;background-color:#FCFCFC;">
2.2.1.路由计算
</h3>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
RouteInfo接口包含了数据包发送到目标主机过程中，经过的路由信息。HttpRoute类继承了Route
nfo接口，是RouteInfo的具体实现，这个类是不允许修改的。HttpTracker类也实现了RouteInfo接
，它是可变的，HttpClient会在内部使用这个类来探测到目标主机的剩余路由。HttpRouteDirector
个辅助类，可以帮助计算数据包的下一步路由信息。这个类也是在HttpClient内部使用的。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac

ground-color:#FCFCFC;">
    HttpRoutePlanner接口可以用来表示基于http上下文情况下，客户端到服务器的路由计算策略。HttpClient有两个HttpRoutePlanner的实现类。SystemDefaultRoutePlanner这个类基于java.net.ProxySelector，它默认使用jvm的代理配置信息，这个配置信息一般来自系统配置或者浏览器配置。DefaulProxyRoutePlanner这个类既不使用java本身的配置，也不使用系统或者浏览器的配置。它通常通过认代理来计算路由信息。

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.2.2. 安全的HTTP连接
</h3>

    为了防止通过Http消息传递的信息不被未授权的第三方获取、截获，Http可以使用SSL/TLS协议来证http传输安全，这个协议是当前使用最广的。当然也可以使用其他的加密技术。但是通常情况下，Http信息会在加密的SSL/TLS连接上进行传输。

<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color#FCFCFC;">
    2.3. HTTP连接管理器
</h2>

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.3.1. 管理连接和连接管理器
</h3>

    Http连接是复杂，有状态的，线程不安全的对象，所以它必须被妥善管理。一个Http连接在同一间只能被一个线程访问。HttpClient使用一个叫做Http连接管理器的特殊实体类来管理Http连接，这实体类要实现HttpClientConnectionManager接口。Http连接管理器在新建http连接时，作为工厂类管理持久http连接的生命周期;同步持久连接（确保线程安全，即一个http连接同一时间只能被一个线访问）。Http连接管理器和ManagedHttpClientConnection的实例类一起发挥作用，ManagedHttClientConnection实体类可以看做http连接的一个代理服务器，管理着I/O操作。如果一个Http连接释放或者被它的消费者明确表示要关闭，那么底层的连接就会和它的代理进行分离，并且该连接会被还给连接管理器。这是，即使服务消费者仍然持有代理的引用，它也不能再执行I/O操作，或者更改Http连接的状态。

    下面的代码展示了如何从连接管理器中取得一个http连接：

<pre>HttpClientContext context = HttpClientContext.create();

```
    HttpClientConnectionManager connMrg = new BasicHttpClientConnectionManager();
    HttpRoute route = new HttpRoute(new HttpHost("www.yeetrack.com", 80));
    // 获取新的连接. 这里可能耗费很多时间
    ConnectionRequest connRequest = connMrg.requestConnection(route, null);
    // 10秒超时
    HttpClientConnection conn = connRequest.get(10, TimeUnit.SECONDS);
    try {
        // 如果创建连接失败
        if (!conn.isOpen()) {
            // establish connection based on its route info
            connMrg.connect(conn, route, 1000, context);
            // and mark it as route complete
            connMrg.routeComplete(conn, route, context);
        }
        // 进行自己的操作.
    } finally {
        connMrg.releaseConnection(conn, null, 1, TimeUnit.MINUTES);
    }</pre>
```

    如果要终止连接，可以调用ConnectionRequest的cancel()方法。这个方法会解锁被ConnectionRequest类get()方法阻塞的线程。

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.3.2.简单连接管理器
</h3>

    BasicHttpClientConnectionManager是个简单的连接管理器，它一次只能管理一个连接。尽管这类是线程安全的，它在同一时间也只能被一个线程使用。BasicHttpClientConnectionManager会尽重用旧的连接来发送后续的请求，并且使用相同的路由。如果后续请求的路由和旧连接中的路由不匹，BasicHttpClientConnectionManager就会关闭当前连接，使用请求中的路由重新建立连接。如果前的连接正在被占用，会抛出java.lang.IllegalStateException异常。

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.3.3.连接池管理器
</h3>

    相对BasicHttpClientConnectionManager来说，PoolingHttpClientConnectionManager是个复杂的类，它管理着连接池，可以同时为很多线程提供http连接请求。Connections are pooled on a per route basis.当请求一个新的连接时，如果连接池有有可用的持久连接，连接管理器就会使用其的一个，而不是再创建一个新的连接。

    PoolingHttpClientConnectionManager维护的连接数在每个路由基础和总数上都有限制。默认每个路由基础上的连接不超过2个，总连接数不能超过20。在实际应用中，这个限制可能会太小了，其是当服务器也使用Http协议时。

    下面的例子演示了如果调整连接池的参数：

<pre>PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManage();
    // 将最大连接数增加到200
    cm.setMaxTotal(200);
    // 将每个路由基础的连接增加到20
    cm.setDefaultMaxPerRoute(20);
    //将目标主机的最大连接数增加到50
    HttpHost localhost = new HttpHost("www.yeetrack.com", 80);
    cm.setMaxPerRoute(new HttpRoute(localhost), 50);

CloseableHttpClient httpClient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();</pre>

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertica-align:baseline;background-color:#FCFCFC;">
    2.3.4.关闭连接管理器
</h3>

    当一个HttpClient的实例不在使用，或者已经脱离它的作用范围，我们需要关掉它的连接管理器，关闭掉所有的连接，释放掉这些连接占用的系统资源。

<pre>CloseableHttpClient httpClient = <...>
    httpClient.close();</pre>

<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color#FCFCFC;">
    2.4.多线程请求执行
</h2>

    当使用了请求连接池管理器（比如PoolingClientConnectionManager）后，HttpClient就可以同执行多个线程的请求了。

<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac

ground-color:#FCFCFC;">
    PoolingClientConnectionManager会根据它的配置来分配请求连接。如果连接池中的所有连接被占用了，那么后续的请求就会被阻塞，直到有连接被释放回连接池中。为了防止永远阻塞的情况发，我们可以把http.conn-manager.timeout的值设置成一个整数。如果在超时时间内，没有可用连接就会抛出ConnectionPoolTimeoutException异常。

<pre>PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManage();
    CloseableHttpClient httpClient = HttpClients.custom()
            .setConnectionManager(cm)
            .build();

// URL列表数组
String[] urisToGet = {
    "http://www.domain1.com/",
    "http://www.domain2.com/",
    "http://www.domain3.com/",
    "http://www.domain4.com/"
};

// 为每个url创建一个线程，GetThread是自定义的类
GetThread[] threads = new GetThread[urisToGet.length];
for (int i = 0; i < threads.length; i++) {
    HttpGet httpget = new HttpGet(urisToGet[i]);
    threads[i] = new GetThread(httpClient, httpget);
}

// 启动线程
for (int j = 0; j < threads.length; j++) {
    threads[j].start();
}

// join the threads
for (int j = 0; j < threads.length; j++) {
    threads[j].join();
}</pre>

    即使HttpClient的实例是线程安全的，可以被多个线程共享访问，但是仍旧推荐每个线程都要有自专用实例的HttpContext。

    下面是GetThread类的定义：

<pre>static class GetThread extends Thread {

    private final CloseableHttpClient httpClient;
    private final HttpContext context;
    private final HttpGet httpget;

```
    public GetThread(CloseableHttpClient httpClient, HttpGet httpget) {
        this.httpClient = httpClient;
        this.context = HttpClientContext.create();
        this.httpget = httpget;
    }

    @Override
    public void run() {
        try {
            CloseableHttpResponse response = httpClient.execute(
                    httpget, context);
            try {
                HttpEntity entity = response.getEntity();
            } finally {
                response.close();
            }
        } catch (ClientProtocolException ex) {
            // Handle protocol errors
        } catch (IOException ex) {
            // Handle I/O errors
        }
    }

}</pre>

<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <br />
</p>
<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color:#FCFCFC;">
    2.5. 连接回收策略
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    经典阻塞I/O模型的一个主要缺点就是只有当组侧I/O时，socket才能对I/O事件做出反应。当连接管理器收回后，这个连接仍然存活，但是却无法监控socket的状态，也无法对I/O事件做出反馈。如连接被服务器端关闭了，客户端监测不到连接的状态变化（也就无法根据连接状态的变化，关闭本地socket）。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    HttpClient为了缓解这一问题造成的影响，会在使用某个连接前，监测这个连接是否已经过时，如服务器端关闭了连接，那么连接就会失效。这种过时检查并不是100%有效，并且会给每个请求增加1到30毫秒额外开销。唯一一个可行的，且does not involve a one thread per socket model for idle connections的解决办法，是建立一个监控线程，来专门回收由于长时间不活动而被判定为失效的连。这个监控线程可以周期性的调用ClientConnectionManager类的closeExpiredConnections()方法关闭过期的连接，回收连接池中被关闭的连接。它也可以选择性的调用ClientConnectionManager的closeIdleConnections()方法来关闭一段时间内不活动的连接。
</p>
<pre>public static class IdleConnectionMonitorThread extends Thread {


    private final HttpClientConnectionManager connMgr;
```

```
    private volatile boolean shutdown;

    public IdleConnectionMonitorThread(HttpClientConnectionManager connMgr) {
        super();
        this.connMgr = connMgr;
    }

    @Override
    public void run() {
        try {
            while (!shutdown) {
                synchronized (this) {
                    wait(5000);
                    // 关闭失效的连接
                    connMgr.closeExpiredConnections();
                    // 可选的, 关闭30秒内不活动的连接
                    connMgr.closeIdleConnections(30, TimeUnit.SECONDS);
                }
            }
        } catch (InterruptedException ex) {
            // terminate
        }
    }

    public void shutdown() {
        shutdown = true;
        synchronized (this) {
            notifyAll();
        }
    }

}</pre>

<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    <br />
</p>
<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color
#FCFCFC;">
    2.6. 连接存活策略
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    Http规范没有规定一个持久连接应该保持存活多久。有些Http服务器使用非标准的Keep-Alive头
息和客户端进行交互，服务器端会保持数秒时间内保持连接。HttpClient也会利用这个头消息。如果
务器返回的响应中没有包含Keep-Alive头消息，HttpClient会认为这个连接可以永远保持。然而，很
服务器都会在不通知客户端的情况下，关闭一定时间内不活动的连接，来节省服务器资源。在某些情
下默认的策略显得太乐观，我们可能需要自定义连接存活策略。
</p>
<pre>ConnectionKeepAliveStrategy myStrategy = new ConnectionKeepAliveStrategy() {


    public long getKeepAliveDuration(HttpResponse response, HttpContext context) {
        // Honor 'keep-alive' header
```

```
        HeaderElementIterator it = new BasicHeaderElementIterator(
                response.headerIterator(HTTP.CONN_KEEP_ALIVE));
        while (it.hasNext()) {
            HeaderElement he = it.nextElement();
            String param = he.getName();
            String value = he.getValue();
            if (value != null &amp;amp;&amp;amp; param.equalsIgnoreCase("timeout")) {
                try {
                    return Long.parseLong(value) * 1000;
                } catch(NumberFormatException ignore) {
                }
            }
        }
        HttpHost target = (HttpHost) context.getAttribute(
                HttpClientContext.HTTP_TARGET_HOST);
        if ("www.naughty-server.com".equalsIgnoreCase(target.getHostName())) {
            // Keep alive for 5 seconds only
            return 5 * 1000;
        } else {
            // otherwise keep alive for 30 seconds
            return 30 * 1000;
        }
    }

};
CloseableHttpClient client = HttpClients.custom()
        .setKeepAliveStrategy(myStrategy)
        .build();</pre>

<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <br />
</p>
<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color:#FCFCFC;">
    2.7.socket连接工厂
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    Http连接使用java.net.Socket类来传输数据。这依赖于ConnectionSocketFactory接口来创建、始化和连接socket。这样也就允许HttpClient的用户在代码运行时，指定socket初始化的代码。PlainonnectionSocketFactory是默认的创建、初始化明文socket（不加密）的工厂类。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    创建socket和使用socket连接到目标主机这两个过程是分离的，所以我们可以在连接发生阻塞时，闭socket连接。
</p>
<pre>HttpClientContext clientContext = HttpClientContext.create();
    PlainConnectionSocketFactory sf = PlainConnectionSocketFactory.getSocketFactory();
    Socket socket = sf.createSocket(clientContext);
    int timeout = 1000; //ms
    HttpHost target = new HttpHost("www.yeetrack.com");
    InetSocketAddress remoteAddress = new InetSocketAddress(
```

```
        InetAddress.getByName("www.yeetrack.com", 80);
        //connectSocket源码中，实际没有用到target参数
        sf.connectSocket(timeout, socket, target, remoteAddress, null, clientContext);</pre>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <br />
</p>
<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.7.1.安全SOCKET分层
</h3>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    LayeredConnectionSocketFactory是ConnectionSocketFactory的拓展接口。分层socket工厂可以在明文socket的基础上创建socket连接。分层socket主要用于在代理服务器之间创建安全socket HttpClient使用SSLSocketFactory这个类实现安全socket，SSLSocketFactory实现了SSL/TLS分层请知晓，HttpClient没有自定义任何加密算法。它完全依赖于Java加密标准（JCE）和安全套接字（JSE）拓展。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <br />
</p>
<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.7.2.集成连接管理器
</h3>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    自定义的socket工厂类可以和指定的协议（Http、Https）联系起来，用来创建自定义的连接管理。
</p>
<pre>ConnectionSocketFactory plainsf = &lt;...&gt;
    LayeredConnectionSocketFactory sslsf = &lt;...&gt;
    Registry&lt;ConnectionSocketFactory&gt; r = RegistryBuilder.&lt;ConnectionSocketFactor&gt;create()
            .register("http", plainsf)
            .register("https", sslsf)
            .build();


HttpClientConnectionManager cm = new PoolingHttpClientConnectionManager(r);
HttpClients.custom()
        .setConnectionManager(cm)
        .build();</pre>

<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <br />
</p>
<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.7.3.SSL/TLS定制
</h3>
```

    HttpClient使用SSLSocketFactory来创建ssl连接。SSLSocketFactory允许用户高度定制。它可以受javax.net.ssl.SSLContext这个类的实例作为参数，来创建自定义的ssl连接。

<pre>HttpClientContext clientContext = HttpClientContext.create();
    KeyStore myTrustStore = <...>
    SSLContext sslContext = SSLContexts.custom()
            .useTLS()
            .loadTrustMaterial(myTrustStore)
            .build();
    SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(sslContext);</pre>

<h3 style="font-size:10px;color:#373737;font-family:Arial, sans-serif;font-weight:inherit;vertical-align:baseline;background-color:#FCFCFC;">
    2.7.4.域名验证
</h3>

    除了信任验证和在ssl/tls协议层上进行客户端认证，HttpClient一旦建立起连接，就可以选择性验目标域名和存储在X.509证书中的域名是否一致。这种验证可以为服务器信任提供额外的保障。X509ostnameVerifier接口代表主机名验证的策略。在HttpClient中，X509HostnameVerifier有三个实现。重要提示：主机名有效性验证不应该和ssl信任验证混为一谈。

<ul style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;background-color:#FCFCFC;">
    <li style="font-family:inherit;font-style:inherit;font-weight:inherit;vertical-align:baseline;">
        StrictHostnameVerifier: 严格的主机名验证方法和java 1.4,1.5,1.验证方法相同。和IE6的方式也大致相同。这种验证方式符合RFC 2818通配符。The hostname must match either the first CN, or any of the subject-alts. A wildcard can occur in the CN, and in any of the subject-alts.

    <li style="font-family:inherit;font-style:inherit;font-weight:inherit;vertical-align:baseline;">
        BrowserCompatHostnameVerifier: 这种验证主机名的方法，和Cul及firefox一致。The hostname must match either the first CN, or any of the subject-alts. A wildard can occur in the CN, and in any of the subject-alts.StrictHostnameVerifier和BrowserComptHostnameVerifier方式唯一不同的地方就是，带有通配符的域名（比如*.yeetrack.com),BrowserCmpatHostnameVerifier方式在匹配时会匹配所有的的子域名，包括 a.b.yeetrack.com .

    <li style="font-family:inherit;font-style:inherit;font-weight:inherit;vertical-align:baseline;">
        AllowAllHostnameVerifier: 这种方式不对主机名进行验证，验证能被关闭，是个空操作，所以它不会抛出javax.net.ssl.SSLException异常。HttpClient默认使用BrowerCompatHostnameVerifier的验证方式。如果需要，我们可以手动执行验证方式。
<pre>SSLContext sslContext = SSLContexts.createSystemDefault();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(
        sslContext,
        SSLConnectionSocketFactory.STRICT_HOSTNAME_VERIFIER);</pre>


```
    <br />
</p>
<h2 style="font-size:15px;font-family:Arial, sans-serif;vertical-align:baseline;background-color
#FCFCFC;">
    2.8.HttpClient代理服务器配置
</h2>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    尽管，HttpClient支持复杂的路由方案和代理链，它同样也支持直接连接或者只通过一跳的连接。
</p>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    使用代理服务器最简单的方式就是，指定一个默认的proxy参数。
</p>
<pre>HttpHost proxy = new HttpHost("someproxy", 8080);
    DefaultProxyRoutePlanner routePlanner = new DefaultProxyRoutePlanner(proxy);
    CloseableHttpClient httpclient = HttpClients.custom()
            .setRoutePlanner(routePlanner)
            .build();</pre>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    我们也可以让HttpClient去使用jre的代理服务器。
</p>
<pre>SystemDefaultRoutePlanner routePlanner = new SystemDefaultRoutePlanner(
        ProxySelector.getDefault());
    CloseableHttpClient httpclient = HttpClients.custom()
            .setRoutePlanner(routePlanner)
            .build();</pre>
<p style="font-family:Arial, sans-serif;font-size:15px;vertical-align:baseline;color:#373737;bac
ground-color:#FCFCFC;">
    又或者，我们也可以手动配置RoutePlanner，这样就可以完全控制Http路由的过程。
</p>
<pre><span style="font-size:small;"> HttpRoutePlanner routePlanner = new HttpRoutePlann
r() {


    public HttpRoute determineRoute(
            HttpHost target,
            HttpRequest request,
            HttpContext context) throws HttpException {
        return new HttpRoute(target, null,  new HttpHost("someproxy", 8080),
                "https".equalsIgnoreCase(target.getSchemeName()));
    }

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setRoutePlanner(routePlanner)
        .build();
    }
}

http://www.yeetrack.com/?p=782</span></pre>
```