Jetty开发指导：HTTP Client

介绍

Jetty HTTP client模块提供易用的API、工具类和一个高性能、异步的实现来执行HTTP和HTTPS请求。

Jetty HTTP client模块要求Java版本1.7或者更高，Java 1.8的应用能用lambda表达式在一些HTTP client API中。

Jetty HTTP client被实现和提供一个异步的API，不会因为I/O时间阻塞，因此使它在线程的利用上更有效率，并很适合用于负载测试和并行计算。

然而，有时你所有需要做的是对一个资源执行一个GET请求，HTTP client也提供了一个异步API，发起请求的线程将阻塞直到请求处理完成。

在外部来看，Jetty HTTP client提供：

1）重定向支持；重定向编码例如302或者303被自动跟随；

2）Cookies支持；被服务端送的cookies在匹配的请求中被送回到服务端；

3）认证支持；HTTP “Basic”和“Digest”热症被支持，其它的可以增加；

4）前转协议支持。

初始化

主要的类的名称是org.eclipse.jetty.client.HttpClient，同Jetty 7和Jetty 8中一样（虽然它和Jetty 7和Jetty 8中的同名类不是向后兼容的）。

你可以将一个HttpClient实例看作一个浏览器实例。像一个浏览器，它能发起请求到不同的域，它管理重定向、cookies和认证，你能用代理配置它，并且他提供给你你发起的请求的响应。

为了使用HttpClient，你必须初始化它、配置它、然后启动它：

```
// Instantiate HttpClient
HttpClient httpClient = new HttpClient();

// Configure HttpClient, for example:
httpClient.setFollowRedirects(false);

// Start HttpClient
httpClient.start();
```

你能创建多个HttpClient的实例；原因可能是你想指定不同的配置参数（例如，一个实例被配置为前转代理而另一个不是），或者因为你想有两个实例履行象两个不同的浏览器，因此有不同的cookies、不同的认证证书等等。

当你用参数构造器创建一个HttpClient实例时，你仅能履行简单的HTTP请求，并且你将不能履行HTTPS请求。

为了履行HTTPS请求，你首先应该创建一个SslContextFactory，配置它，并且传递它到HttpClient的构造器。当用一个SslContextFactory创建时，HttpClient将能履行HTTP和HTTPS请求到任何域。

```
// Instantiate and configure the SslContextFactory
SslContextFactory sslContextFactory = new SslContextFactory();

// Instantiate HttpClient with the SslContextFactory
HttpClient httpClient = new HttpClient(sslContextFactory);

// Configure HttpClient, for example:
httpClient.setFollowRedirects(false);

// Start HttpClient
httpClient.start();
```

API介绍

阻塞API

为了履行一个HTTP请求更简单的方法是：

```
ContentResponse response = httpClient.GET(<a target=_blank href="http://domain.com/path?query">http://domain.com/path?query</a>);
```

方法HttpClient.GET(...)履行一个HTTP GET请求到一个给定的URI，成功后返回一个ContentResponse。
ContentResponse对象包含HTTP响应信息：状态码、headers和可能的内容。内容长度默认被限制到2M；下面“响应内容处理”会介绍怎么处理更大的内容。
如果你想定制请求，例如通过发起一个HEAD请求代替一个GET，并且仿真一个浏览器用户代理，你能使用这种方式：

```
ContentResponse response = httpClient.newRequest("http://domain.com/path?query")
    .method(HttpMethod.HEAD)
    .agent("Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:17.0) Gecko/20100101 Firefox/17.0")
    .send();
```

下面是采用简写的方式：

```
Request request = httpClient.newRequest("http://domain.com/path?query");
request.method(HttpMethod.HEAD);
request.agent("Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:17.0) Gecko/20100101 Firefox/17.0");
ContentResponse response = request.send();
```

你首先用httpClient.newRequest(...)创建了一个请求对象，然后你定制它。当请求对象被定制后，你调用Request.send()，当请求处理玩陈过后，返回ContentResponse。
简单的POST请求也有一个简写的方法：

```
ContentResponse response = httpClient.POST("http://domain.com/entity/1")
    .param("p", "value")
    .send();
```

POST的参数值被自动的URL编码。
Jetty HTTP client自动地跟随重定向，因此自动地处理这种典型的web模式POST/Redirect/GET，并且响应对象包含了GET请求的响应的内容。跟随重定向是一个特征，你能针对每个请求或者全局来激活/停止。
文件上传也需要一行，使用JDK 7的java.nio.file类：

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .file(Paths.get("file_to_upload.txt"), "text/plain")
    .send();
```

也可以增加一个超时时间：

```
ContentResponse response = httpClient.newRequest("http://domain.com/path?query")
    .timeout(5, TimeUnit.SECONDS)
    .send();
```

在上面的例子中，当超过5秒后，请求被终止并抛出一个java.util.concurrent.TimeoutException异常。

异步API

到目前为止我们展示了怎么使用Jetty HTTP client的阻塞API，即发起请求的线程阻塞直到请求被处理完成。在这节我们将看看Jetty HTTP client的异步、非阻塞API，非常适合大数据下载、请求/响应的并行处理、在性能和有效的线程、资源的利用是一个关键因素的所有场合。
异步API在请求和响应处理的各个阶段都依赖于对回调listener的调用。这些listener被应用实现，可以履行任何应用逻辑。实现在处理请求或者响应的线程中调用这些listener。因此，如果在这些listener中的应用代码需要花费较长时间，请求或者响应的处理将被阻塞。
如果你需要在一个listener内执行耗时操作，你必须使用你自己的线程，并且记住要深度拷贝listener提供的任何数据，因为当listener返回后，这些数据可能回收/清除/销毁。

请求和响应在两个不同的线程中执行，因此可以并行的执行。这个并行处理的一个典型例子是一个回显服务器，一个大的上传和大的回显下载同时进行。注意，记住响应可以在请求之前被处理和完成；一个典型的例子是被服务器触发一个快速响应的一个大的上载（例如一个error）：当请求内容任然在上载时，响应已经到达和被完成了。

应用线程调用Request.send(CompleteListener)履行请求的处理，直到或者请求被充分地处理或者由于阻塞在I/O而返回（因此从不阻塞）。如果它将阻塞在I/O，线程请求I/O系统当I/O完成时发出一个事件，然后返回。当如此一个事件被触发，一个来自HttpClient线程池的线程将恢复响应的处理。

响应被线程处理，这些线程或者是触发字节码已经被准备好的I/O系统线程，或者是来自HttpClient线程池的一个线程（这通过HttpClient.isDispatchIO()属性控制）。响应持续处理直到响应被处理完成或者阻塞在I/O。如果它阻塞在I/O，线程请求I/O系统在I/O准备好后发出一个时间，然后返回。当如此一个事件被触发，一个来自HttpClient线程池的线程将恢复响应的处理。

当请求和响应都处理完成后，完成最后处理的线程（通常是处理响应的线程，但也可能是处理请求的线程——如果请求比响应的处理花费更多的时间）将取下一个请求进行处理。

一个抛弃响应内容的异步GET请求能这样实现：

```
httpClient.newRequest("http://domain.com/path")
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
```

方法Request.send(Response.CompleteListener)返回void，并且不阻塞；当请求/响应处理完成后Response.CompleteListener将被通知，result参数可以获取到响应对象。

你能用JDK 8的lambda表达式写同样的代码：

```
httpClient.newRequest("http://domain.com/path")
    .send((result) -> { /* Your logic here */ });
```

你也能为它指定一个总的超时时间：

```
Request request = httpClient.newRequest("http://domain.com/path")
    .timeout(3, TimeUnit.SECONDS)
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
```

上面的例子为请求/响应处理指定了一个超时时间3秒。

HTTP client API广泛的使用listener为所有可能的请求和响应时间提供钩子，在JDK 8的lambda表达式中，他们变得更易使用：

```
httpClient.newRequest("http://domain.com/path")
    // Add request hooks
    .onRequestQueued((request) -> { ... })
    .onRequestBegin((request) -> { ... })
    ... // More request hooks available

    // Add response hooks
```

```
    .onResponseBegin((response) -> { ... })
    .onResponseHeaders((response) -> { ... })
    .onResponseContent((response, buffer) -> { ... })
    ... // More response hooks available

    .send((result) -> { ... });
```

这使得Jetty HTTP client很适合HTTP负载测试，例如，你能精确的知道请求/响应处理的每一步花费的时间（因此知道请求/响应时间被真正消耗的地方）。
了解请求事件请查看Request.Listener，了解响应事件请查看Response.Listener。

内容处理

请求内容处理

Jetty HTTP client提供了许多现成的工具类处理请求内容。
你能提供这些格式的请求内容：String、byte[]、ByteBuffer、java.nio.file.Path、 InputStream，并提供你的org.eclipse.jetty.client.api.ContentProvider的实现。下面是一个例子，使用java.nio.file.Paths提供请求内容：

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .file(Paths.get("file_to_upload.txt"), "text/plain")
    .send();
```

这等价于这样使用PathContentProvider工具类：

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new PathContentProvider(Paths.get("file_to_upload.txt")), "text/plain")
    .send();
```

同样，你能通过InputStreamContentProvider工具类使用FileInputStream：

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new InputStreamContentProvider(new FileInputStream("file_to_upload.txt")), "text/plain")
    .send();
```

由于InputStream是阻塞的，因此请求的发送将阻塞，可以考虑使用异步API。
如果你已经将内容读到内存中，你能使用BytesContentProvider工具类将它作为byte[]传入：

```
byte[] bytes = ...;
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new BytesContentProvider(bytes), "text/plain")
    .send();
```

如果请求内容不是立即可用的，你能用DeferredContentProvider：

```
DeferredContentProvider content = new DeferredContentProvider();
httpClient.newRequest("http://domain.com/upload")
    .content(content)
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
```

```
// Content not available yet here

...

// An event happens, now content is available
byte[] bytes = ...;
content.offer(ByteBuffer.wrap(bytes));

...

// All content has arrived
content.close();
```

提供请求内容的另一个方法是使用OutputStreamContentProvider，当请求内容可用时允许应用写请求内容到OutputStreamContentProvider提供的OutputStream：

```
OutputStreamContentProvider content = new OutputStreamContentProvider();

// Use try-with-resources to close the OutputStream when all content is written
try (OutputStream output = content.getOutputStream())
{
    client.newRequest("localhost", 8080)
            .content(content)
            .send(new Response.CompleteListener()
            {
                @Override
                public void onComplete(Result result)
                {
                    // Your logic here
                }
            });

    ...

    // Write content
    writeContent(output);
}
// End of try-with-resource, output.close() called automatically to signal end of content
```

响应内容处理

Jetty HTTP client允许应用使用多种方式处理响应内容。

第一种方式是缓存响应内容在内存中；使用阻塞API，在一个ContentResponse 中内容的最大缓存是2MiB。

如果你想控制响应内容的长度（例如限制到小于2MiB的默认值），那么你能用一个org.eclipse.jetty.client.util.FutureResponseListener：

```
Request request = httpClient.newRequest("http://domain.com/path");

// Limit response content buffer to 512 KiB
FutureResponseListener listener = new FutureResponseListener(request, 512 * 1024);
```

```
request.send(listener);

ContentResponse response = listener.get(5, TimeUnit.SECONDS);
```

如果响应内容长度逸出，响应想被终止，一个异常将被方法get()抛出。

如果你正在使用异步API，你能用BufferingResponseListener工具类：

```
httpClient.newRequest("http://domain.com/path")
        // Buffer response content up to 8 MiB
        .send(new BufferingResponseListener(8 * 1024 * 1024)
        {
            @Override
            public void onComplete(Result result)
            {
                if (!result.isFailed())
                {
                    byte[] responseContent = getContent();
                    // Your logic here
                }
            }
        });
```

第二种方法最有效率（因为它避免了内容拷贝），并允许你指定一个Response.ContentListener，或者一个子类，处理到达的内容：

```
ContentResponse response = httpClient
        .newRequest("http://domain.com/path")
        .send(new Response.Listener.Empty()
        {
            @Override
            public void onContent(Response response, ByteBuffer buffer)
            {
                // Your logic here
            }
        });
```

第三种方法允许你等待响应，然后用InputStreamResponseListener工具类输出内容：

```
InputStreamResponseListener listener = new InputStreamResponseListener();
httpClient.newRequest("http://domain.com/path")
        .send(listener);

// Wait for the response headers to arrive
Response response = listener.get(5, TimeUnit.SECONDS);

// Look at the response
if (response.getStatus() == 200)
{
    // Use try-with-resources to close input stream.
    try (InputStream responseContent = listener.getInputStream())
    {
        // Your logic here
```

```
    }
}
```

其它特征

Cookies支持

Jetty HTTP client原生的支持cookie。HttpClient实例从HTTP响应收到cookie，然后存储他们在java.net.CookieStore中，这个类属于JDK。当新请求被创建，cookie缓存被查阅，如果存在匹配的cookie（即，coolie没有逸出，且匹配域和请求路径），这些cookie将被添加到请求。
应用能通过编程进入cookie缓存，查找设置的cookie：

```
CookieStore cookieStore = httpClient.getCookieStore();
List<HttpCookie> cookies = cookieStore.get(URI.create(<a target=_blank href="http://domain.com/path">http://domain.com/path</a>));
```

应用也能通过编程设置cookie，如果他们从一个HTTP响应返回：

```
CookieStore cookieStore = httpClient.getCookieStore();
HttpCookie cookie = new HttpCookie("foo", "bar");
cookie.setDomain("domain.com");
cookie.setPath("/");
cookie.setMaxAge(TimeUnit.DAYS.toSeconds(1));
cookieStore.add(URI.create("http://domain.com"), cookie);
```

你能移除不想再使用的cookies：

```
CookieStore cookieStore = httpClient.getCookieStore();
URI uri = URI.create("http://domain.com");
List<HttpCookie> cookies = cookieStore.get(uri);
for (HttpCookie cookie : cookies)
    cookieStore.remove(uri, cookie);
```

如果你想完全地禁用cookie处理，你能安装一个HttpCookieStore.Empty实例：

```
httpClient.setCookieStore(new HttpCookieStore.Empty());
```

你能激活cookie过滤，通过安装一个履行过滤逻辑的cookie缓存：

```
httpClient.setCookieStore(new GoogleOnlyCookieStore());


public class GoogleOnlyCookieStore extends HttpCookieStore
{
    @Override
    public void add(URI uri, HttpCookie cookie)
    {
        if (uri.getHost().endsWith("google.com"))
            super.add(uri, cookie);
    }
}
```

上面的例子将仅保留来自google.com域或者子域的cookies。

认证支持

Jetty HTTP client支持"Basic"和"Digest"认证机制，在RFC 2617中定义。
你能在HTTP client实例中配置认证证书如下：

```
URI uri = new URI("http://domain.com/secure");
String realm = "MyRealm";
String user = "username";
String pass = "password";

// Add authentication credentials
AuthenticationStore auth = httpClient.getAuthenticationStore();
auth.addAuthentication(new BasicAuthentication(uri, realm, user, pass));

ContentResponse response = httpClient
        .newRequest(uri)
        .send()
        .get(5, TimeUnit.SECONDS);
```

成功的认证被缓存，但是你能清除它们，迫使重新认证：

```
httpClient.getAuthenticationStore().clearAuthenticationResults();
```

代理支持

Jetty的HTTP client能被配置使用代理。
两种类型的代理是原生的支持的：HTTP代理（通过类org.eclipse.jetty.client.HttpProxy提供）和SOCKS 4代理（通过类org.eclipse.jetty.client.Socks4Proxy提供）。其它实现可以通过子类ProxyConfiguration.Proxy来写。
一个典型的配置如下：

```
ProxyConfiguration proxyConfig = httpClient.getProxyConfiguration();
HttpProxy proxy = new HttpProxy("proxyHost", proxyPort);
// Do not proxy requests for localhost:8080
proxy.getExcludedAddresses().add("localhost:8080");

httpClient.setProxyConfiguration(proxyConfig);

ContentResponse response = httpClient.GET(uri);
```

你指定代理的主机和端口，也设置你不想被代理的地址，然后在HttpClient实例上设置代理配置。