深入剖析 Tomcat

A Fool

2019

目录

第一章　一个简单的 Web 服务器

1.1　HTTP

超文本传输协议（HyperText Transfer Protocol）是基于“请求-响应”的协议。

HTTP 使用可靠地 TCP 连接，默认使用 TCP 80 端口。

Web 服务器不负责联系客户端或建立到一个客户端的回调连接，客户端或服务端可提前关闭连接。

1.1.1　HTTP 请求

一个 HTTP 请求包含：

- 请求方法——统一资源标识符（Uniform Resource Identifier，URI）——协议/版本
- 请求头
- 实体

HTTP1.1 支持 7 种请求方法：GET、POST、HEAD、OPTIONS、PUT、DELETE 和 TRACE。统一资源定位符（Uniform Resource Locator，URL）实际上是 URI 的一种类型。

请求头包含了客户端环境、请求实体正文的相关信息。各个请求头之间使用回车/换行（Carriage Return/LineFeed，CRLF）间隔开。

在请求头和请求实体正文之间有一个空行，该行只有 CRLF 符。

1.1.2　HTTP 响应

HTTP 响应三部分：

- 协议——状态码——描述
- 响应头
- 响应式体段

响应头和响应实体正文之间只包含了 CRLF 的一个空行分隔。

1.2 Socket 类

套接字使应用程序可以从网络中读取数据，可以向网络中写入数据。

Java 中的套接字：java.net.Socket，构造函数：

public Socket(java.lang.String host, int port)

host 为远程主机名称或 IP 地址，参数 port 是远程应用程序端口号。两者通信：

```
1  Socket socket = new Socket("127.0.0.1", 8080);
2  OutputStream os = socket.getOutputStream();
3  boolean autoflush = true;
4  PrintWriter out = new PrintWriter(socket.getOutputStream(),
       autoflush);
5  BufferedReader in = new BufferedReader(new InputStreamReader(socket
       .getInputStream()));
6
7  out.println("GET /index.jsp HTTP/1.1");
8  out.println("Host: localhost:8080");
9  out.println("Connection: Close");
10 out.println();
11
12 boolean loop =true;
13 StringBuffer sb = new StringBuffer(8096);
14 while (loop) {
15     if (in.ready()) {
16         int i = 0;
17         while (i != -1) {
18             i = in.read();
19             sb.append((char) i);
20         }
21         loop = false;
22     }
23     Thread.currentThread().sleep(50);
24 }
25 System.out.println(sb.toString());
26 socket.close();
```

1.2.1 ServerSocket 类

与 Socket 不同，当服务器套接字接收到连接请求后，会创建 Socket 实例来处理与客户端通信。

ServerSocket 提供 4 个构造函数，典型如下：

```
1 public ServerSocket(int port, int backlog, InetAddress
    bindingAddress)
```

Port 是服务器端口，IP 地址一般是 127.0.0.1（绑定地址），baklog 表示在服务器拒绝接受传入的请求之前，传入的连接请求的最大队列长度。

这里 IP 必须是 java.net.InetAddress 的实例，可以使用静态方法获取：

```
1 InetAddress.getByName("127.0.0.1")
```

当创建 ServerSocket 实例后，等待传入的连接请求，可以通过 ServerSocket 的 accept 方法完成，只有收到请求后才返回，返回值是一个 Socket 实例。

1.3 应用程序

发送指定目录静态资源，但不发送任何头信息，程序包含三个部分：

- HttpServer
- Request
- Response

1.3.1 HttpServer

```
1 public class HttpServer {
2
3     /** WEB_ROOT is the directory where our HTML and other files
          reside.
4     *  For this package, WEB_ROOT is the "webroot" directory under
          the working
5     *  directory.
```

```
6       *  The working directory is the location in the file system
7       *  from where the java command was invoked.
8       */
9       public static final String WEB_ROOT =
10      System.getProperty("user.dir") + File.separator  + "webroot";
11
12      // shutdown command
13      private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
14
15      // the shutdown command received
16      private boolean shutdown = false;
17
18      public static void main(String[] args) {
19          HttpServer server = new HttpServer();
20          server.await();
21      }
22
23      public void await() {
24          ServerSocket serverSocket = null;
25          int port = 8080;
26          try {
27              serverSocket =  new ServerSocket(port, 1, InetAddress.
                    getByName("127.0.0.1"));
28          }
29          catch (IOException e) {
30              e.printStackTrace();
31              System.exit(1);
32          }
33
34          // Loop waiting for a request
35          while (!shutdown) {
36              Socket socket = null;
37              InputStream input = null;
38              OutputStream output = null;
39              try {
40                  socket = serverSocket.accept();
```

```
41                  //收到请求获取java.io.*的实例
42                  input = socket.getInputStream();
43                  output = socket.getOutputStream();
44
45                  // create Request object and parse
46                  Request request = new Request(input);
47                  request.parse();
48
49                  // create Response object
50                  Response response = new Response(output);
51                  response.setRequest(request);
52                  response.sendStaticResource();
53
54                  // Close the socket
55                  socket.close();
56
57                  //check if the previous URI is a shutdown command
58                  shutdown = request.getUri().equals(SHUTDOWN_COMMAND
                        );
59              }
60              catch (Exception e) {
61                  e.printStackTrace();
62                  continue;
63              }
64          }
65      }
66  }
```

1.3.2 Request

Request 表示一个 HTTP 请求，可以传递 InputStream 创建 Request 对象，并调用 InputStream 的 read() 获取原始数据。

```
1  public class Request {
2
3      private InputStream input;
```

```
4     private String uri;
5
6     public Request(InputStream input) {
7         this.input = input;
8     }
9     //解析HTTP原始数据
10    public void parse() {
11        // Read a set of characters from the socket
12        StringBuffer request = new StringBuffer(2048);
13        int i;
14        byte[] buffer = new byte[2048];
15        try {
16            i = input.read(buffer);
17        }
18        catch (IOException e) {
19            e.printStackTrace();
20            i = -1;
21        }
22        for (int j=0; j<i; j++) {
23            request.append((char) buffer[j]);
24        }
25        System.out.print(request.toString());
26        uri = parseUri(request.toString());
27    }
28    //解析HTTP的URI, 按照消息头截取URI
29    private String parseUri(String requestString) {
30        int index1, index2;
31        index1 = requestString.indexOf(' ');
32        if (index1 != -1) {
33            index2 = requestString.indexOf(' ', index1 + 1);
34            if (index2 > index1)
35            return requestString.substring(index1 + 1, index2);
36        }
37        return null;
38    }
39
```

```
40      public String getUri() {
41          return uri;
42      }
43
44  }
```

1.3.3 Response

```
1   /*
2   HTTP Response = Status-Line
3   *(( general-header | response-header | entity-header ) CRLF)
4   CRLF
5   [ message-body ]
6   Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
7   */
8
9   public class Response {
10
11      private static final int BUFFER_SIZE = 1024;
12      Request request;
13      OutputStream output;
14
15      public Response(OutputStream output) {
16          this.output = output;
17      }
18
19      public void setRequest(Request request) {
20          this.request = request;
21      }
22      //将静态资源作为原始数据发送
23      public void sendStaticResource() throws IOException {
24          byte[] bytes = new byte[BUFFER_SIZE];
25          FileInputStream fis = null;
26          try {
```

```
27              File file = new File(HttpServer.WEB_ROOT, request.
                    getUri());
28              if (file.exists()) {
29                  fis = new FileInputStream(file);
30                  int ch = fis.read(bytes, 0, BUFFER_SIZE);
31                  while (ch!=-1) {
32                      output.write(bytes, 0, ch);
33                      ch = fis.read(bytes, 0, BUFFER_SIZE);
34                  }
35              }
36              else {
37                  // file not found
38                  String errorMessage = "HTTP/1.1 404 File Not Found\
                        r\n" +
39                  "Content-Type: text/html\r\n" +
40                  "Content-Length: 23\r\n" +
41                  "\r\n" +
42                  "<h1>File Not Found</h1>";
43                  output.write(errorMessage.getBytes());
44              }
45          }
46          catch (Exception e) {
47              // thrown if cannot instantiate a File object
48              System.out.println(e.toString() );
49          }
50          finally {
51              if (fis!=null)
52              fis.close();
53          }
54      }
55  }
```

第二章　一个简单的 servlet 容器

2.1　javax.servlet.Servlet 接口

Servlet 接口的 5 个方法：

```
1 void init(ServletConfig var1) throws ServletException;
2 ServletConfig getServletConfig();
3 void service(ServletRequest var1, ServletResponse var2) throws
     ServletException, IOException;
4 String getServletInfo();
5 void destroy();
```

其中 init()、service() 和 destroy() 与生命周期有关，实例化 servlet 后会调用 init() 方法，只调用 1 次，然后执行 service()，并且会被多次调用。

在 servlet 移除前，会调用 destroy()，清除自身持有的资源。

2.2　应用程序 1

一个完整的 servlet 责任：

1. 第一次调用，使用 init() 初始化；
2. 针对每一个 request 请求,创建一个 javax.serlvet.ServletRequest 和 javax.serlvet.ServletResponse 实例；
3. 调用 service()，将 ServletRequest 和 ServletResponse 作为参数传入；
4. 当关闭 servlet，调用 destroy()，卸载 servlet。

2.2.1　HttpServer1 类

既可以处理静态资源请求，也可以处理 servlet 资源请求。

```
1  public class HttpServer1 {
2
3      /** WEB_ROOT is the directory where our HTML and other files
           reside.
4      *  For this package, WEB_ROOT is the "webroot" directory under
           the working
5      *  directory.
6      *  The working directory is the location in the file system
7      *  from where the java command was invoked.
8      */
9      // shutdown command
10     private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
11
12     // the shutdown command received
13     private boolean shutdown = false;
14
15     public static void main(String[] args) {
16         HttpServer1 server = new HttpServer1();
17         server.await();
18     }
19
20     public void await() {
21         ServerSocket serverSocket = null;
22         int port = 8080;
23         try {
24             serverSocket =  new ServerSocket(port, 1, InetAddress.
                   getByName("127.0.0.1"));
25         }
26         catch (IOException e) {
27             e.printStackTrace();
28             System.exit(1);
29         }
30
31         // Loop waiting for a request
32         while (!shutdown) {
```

```
33              Socket socket = null;
34              InputStream input = null;
35              OutputStream output = null;
36              try {
37                  socket = serverSocket.accept();
38                  input = socket.getInputStream();
39                  output = socket.getOutputStream();
40
41                  // create Request object and parse
42                  Request request = new Request(input);
43                  request.parse();
44
45                  // create Response object
46                  Response response = new Response(output);
47                  response.setRequest(request);
48
49                  // check if this is a request for a servlet or a
                        static resource
50                  // a request for a servlet begins with "/servlet/"
51                  if (request.getUri().startsWith("/servlet/")) {
52                      ServletProcessor1 processor = new
                            ServletProcessor1();
53                      processor.process(request, response);
54                  }
55                  else {
56                      StaticResourceProcessor processor = new
                            StaticResourceProcessor();
57                      processor.process(request, response);
58                  }
59
60                  // Close the socket
61                  socket.close();
62                  //check if the previous URI is a shutdown command
63                  shutdown = request.getUri().equals(SHUTDOWN_COMMAND
                        );
64              }
```

```
65              catch (Exception e) {
66                  e.printStackTrace();
67                  System.exit(1);
68              }
69          }
70      }
71  }
```

2.2.2 Request 类

表示 servlet 的 service() 方法的参数 request 对象，这里只实现一部分。

```
1  public class Request implements ServletRequest {
2
3      private InputStream input;
4      private String uri;
5
6      public Request(InputStream input) {
7          this.input = input;
8      }
9
10     public String getUri() {
11         return uri;
12     }
13
14     private String parseUri(String requestString) {
15         int index1, index2;
16         index1 = requestString.indexOf(' ');
17         if (index1 != -1) {
18             index2 = requestString.indexOf(' ', index1 + 1);
19             if (index2 > index1)
20             return requestString.substring(index1 + 1, index2);
21         }
22         return null;
23     }
24
```

```
25      public void parse() {
26          // Read a set of characters from the socket
27          StringBuffer request = new StringBuffer(2048);
28          int i;
29          byte[] buffer = new byte[2048];
30          try {
31              i = input.read(buffer);
32          }
33          catch (IOException e) {
34              e.printStackTrace();
35              i = -1;
36          }
37          for (int j=0; j<i; j++) {
38              request.append((char) buffer[j]);
39          }
40          System.out.print(request.toString());
41          uri = parseUri(request.toString());
42      }
43
44      /* implementation of the ServletRequest*/
45      public Object getAttribute(String attribute) {
46          return null;
47      }
48
49      public Enumeration getAttributeNames() {
50          return null;
51      }
52
53      public String getRealPath(String path) {
54          return null;
55      }
56
57      public RequestDispatcher getRequestDispatcher(String path) {
58          return null;
59      }
60
```

```
61      public boolean isSecure() {
62          return false;
63      }
64
65      public String getCharacterEncoding() {
66          return null;
67      }
68
69      public int getContentLength() {
70          return 0;
71      }
72
73      public String getContentType() {
74          return null;
75      }
76
77      public ServletInputStream getInputStream() throws IOException {
78          return null;
79      }
80
81      public Locale getLocale() {
82          return null;
83      }
84
85      public Enumeration getLocales() {
86          return null;
87      }
88
89      public String getParameter(String name) {
90          return null;
91      }
92
93      public Map getParameterMap() {
94          return null;
95      }
96
```

```
97      public Enumeration getParameterNames() {
98          return null;
99      }
100
101     public String[] getParameterValues(String parameter) {
102         return null;
103     }
104
105     public String getProtocol() {
106         return null;
107     }
108
109     public BufferedReader getReader() throws IOException {
110         return null;
111     }
112
113     public String getRemoteAddr() {
114         return null;
115     }
116
117     public String getRemoteHost() {
118         return null;
119     }
120
121     public String getScheme() {
122         return null;
123     }
124
125     public String getServerName() {
126         return null;
127     }
128
129     public int getServerPort() {
130         return 0;
131     }
132
```

```
133     public void removeAttribute(String attribute) {
134     }
135
136     public void setAttribute(String key, Object value) {
137     }
138
139     public void setCharacterEncoding(String encoding)
140     throws UnsupportedEncodingException {
141     }
142 }
```

2.2.3 Response 类

与 Request 类似，这里只有部分实现，getWriter() 实现。

```
1  public class Response implements ServletResponse {
2
3      private static final int BUFFER_SIZE = 1024;
4      Request request;
5      OutputStream output;
6      PrintWriter writer;
7
8      public Response(OutputStream output) {
9          this.output = output;
10     }
11
12     public void setRequest(Request request) {
13         this.request = request;
14     }
15
16     /* This method is used to serve a static page */
17     public void sendStaticResource() throws IOException {
18         byte[] bytes = new byte[BUFFER_SIZE];
19         FileInputStream fis = null;
20         try {
```

```
21              /* request.getUri has been replaced by request.
                   getRequestURI */
22              File file = new File(Constants.WEB_ROOT, request.getUri
                   ());
23              fis = new FileInputStream(file);
24              /*
25              HTTP Response = Status-Line
26              *(( general-header | response-header | entity-header )
                   CRLF)
27              CRLF
28              [ message-body ]
29              Status-Line = HTTP-Version SP Status-Code SP Reason-
                   Phrase CRLF
30              */
31              int ch = fis.read(bytes, 0, BUFFER_SIZE);
32              while (ch!=-1) {
33                  output.write(bytes, 0, ch);
34                  ch = fis.read(bytes, 0, BUFFER_SIZE);
35              }
36          }
37          catch (FileNotFoundException e) {
38              String errorMessage = "HTTP/1.1 404 File Not Found\r\n"
                       +
39              "Content-Type: text/html\r\n" +
40              "Content-Length: 23\r\n" +
41              "\r\n" +
42              "<h1>File Not Found</h1>";
43              output.write(errorMessage.getBytes());
44          }
45          finally {
46              if (fis!=null)
47              fis.close();
48          }
49      }
50
51
```

```
52      /** implementation of ServletResponse  */
53      public void flushBuffer() throws IOException {
54      }
55
56      public int getBufferSize() {
57          return 0;
58      }
59
60      public String getCharacterEncoding() {
61          return null;
62      }
63
64      public Locale getLocale() {
65          return null;
66      }
67
68      public ServletOutputStream getOutputStream() throws IOException
            {
69          return null;
70      }
71
72      public PrintWriter getWriter() throws IOException {
73          // autoflush is true, println() will flush,
74          // but print() will not.
75          writer = new PrintWriter(output, true);
76          return writer;
77      }
78
79      public boolean isCommitted() {
80          return false;
81      }
82
83      public void reset() {}
84      public void resetBuffer() {}
85      public void setBufferSize(int size) {}
86      public void setContentLength(int length) {}
```

```
87      public void setContentType(String type) {}
88      public void setLocale(Locale locale) {}
89  }
```

如果 PrintWriter() 的第二个参数为真，对于 println() 的任何调用都会刷新输出，但是 print() 不会刷新输出，因此如果在 service() 中使用 print() 不会发送给浏览器。

2.2.4 StaticResourceProcessor

这里直接调用 response.sendStaticResource() 处理。

```
1  public class StaticResourceProcessor {
2
3      public void process(Request request, Response response) {
4          try {
5              response.sendStaticResource();
6          }
7          catch (IOException e) {
8              e.printStackTrace();
9          }
10     }
11 }
```

2.2.5 ServletProcessor1

```
1  public class ServletProcessor1 {
2
3      public void process(Request request, Response response) {
4
5          String uri = request.getUri();
6          String servletName = uri.substring(uri.lastIndexOf("/") +
               1);
7          URLClassLoader loader = null;
8
9          try {
```

```
10              // create a URLClassLoader
11              URL[] urls = new URL[1];
12              URLStreamHandler streamHandler = null;
13              File classPath = new File(Constants.WEB_ROOT);
14              // the forming of repository is taken from the
                   createClassLoader method in
15              // org.apache.catalina.startup.ClassLoaderFactory
16              String repository = (new URL("file", null, classPath.
                   getCanonicalPath() + File.separator)).toString() ;
17              // the code for forming the URL is taken from the
                   addRepository method in
18              // org.apache.catalina.loader.StandardClassLoader class
                   .
19              urls[0] = new URL(null, repository, streamHandler);
20              loader = new URLClassLoader(urls);
21          }
22          catch (IOException e) {
23              System.out.println(e.toString() );
24          }
25          Class myClass = null;
26          try {
27              myClass = loader.loadClass(servletName);
28          }
29          catch (ClassNotFoundException e) {
30              System.out.println(e.toString());
31          }
32
33          Servlet servlet = null;
34
35          try {
36              servlet = (Servlet) myClass.newInstance();
37              servlet.service((ServletRequest) request, (
                   ServletResponse) response);
38          }
39          catch (Exception e) {
40              System.out.println(e.toString());
```

```
41          }
42          catch (Throwable e) {
43              System.out.println(e.toString());
44          }
45
46      }
47  }
```

URI 的格式如：/serlvet/servletName，并通过字符串截取获取 servletName；
为了载入类，创建一个类加载器 java.net.URLClassLoader，是 java.lang.ClassLoader 直接子类，可以使用它的 loadClass() 载入 servlet 类；

```
1  public URLClassLoader(URL[] urls)
```

当载入类时要指明在哪查找类，若以“/”结尾，则指向目录，否则指向一个 JAR 文件；这里把目录称为仓库（repository）。
这里使用下面 URL 构造函数：

```
1  public URL(String protocol, String host, String file) throws
       MalformedURLException
2  //或
3  public URL(URL context, String spec, URLStreamHandler handler)
       throws MalformedURLException
```

如果只使用 new URL(null, aString, null) 不知道使用哪个构造函数，因此使用上述例子中 URLStreamHandler 区别构造函数。
加载类后，然后使用 newInstance() 创建新的实例，并调用它的服务程序 service()。

2.3 应用程序 2

```
1  servlet = (Servlet) myClass.newInstance();
2  servlet.service((ServletRequest) request, (ServletResponse)
       response);
```

上述代码使用了向上转型，但是这是不安全的做法，因为了解此 servlet 容器内部结构的程序员，可以将 ServletRequest 和 ServletResponse 向下转型，就可以调用它们的公共方法。

不能将 parse() 和 sendStaticResource() 设置为 private，但这两个方法在 Servlet 又应当是不可用，所以可以使用外观模式。

添加两个外观类：RequestFacade 和 ResponseFacade，前者实现 servletRequest 接口，在其构造函数中需要指定一个 Request 对象传递给 ServletRequest 对象引用（private）。然后将外观类传递给 service() 方法，这里 Servlet 仍可以向下转型 RequestFacade 对象，但只能访问 ServletRequest 提供的方法。

```
1  public class RequestFacade implements ServletRequest {
2
3      private ServletRequest request = null;
4
5      public RequestFacade(Request request) {
6          this.request = request;
7      }
8
9      /* implementation of the ServletRequest*/
10     public Object getAttribute(String attribute) {
11         return request.getAttribute(attribute);
12     }
13
14     public Enumeration getAttributeNames() {
15         return request.getAttributeNames();
16     }
17     //......
```

对于 servletProcessor2，改变部分：

```
1  Servlet servlet = null;
2  RequestFacade requestFacade = new RequestFacade(request);
3  ResponseFacade responseFacade = new ResponseFacade(response);
4  try {
5      servlet = (Servlet) myClass.newInstance();
6      servlet.service((ServletRequest) requestFacade, (
           ServletResponse) responseFacade);
7  }
8  catch (Exception e) {
```

```
9       System.out.println(e.toString());
10  }
```

第三章　连接器

Catalina 两个主要板块：连接器（connector）和容器（container）。
连接器解析 HTTP 请求头，使 servlet 实例能够获取到请求头、cookie 和请求参数/值等信息。

3.1　StringManager

Tomcat 处理错误消息的方法是将错误信息存储在一个 properties 文件中，便于读取和编辑，但是随着类的暴增，错误信息的激增，不方便维护，因此将 properties 划分到不同的包中，会产生 StringManager 多个实例。可以采用单例模式减少资源的消耗，并且使用 Hashtable 保存该单个实例。

3.2　应用程序

3 个模块：连接器模块、启动模块和核心模块。
启动模块：只有一个类（Bootstrap），负责启动应用程序。
连接器模块 5 类：

- 连接器及其支持类（HttpConnector 和 HttpProcessor）
- 表示 HTTP 请求的类（HttpRequest）及其支持类
- 表示 HTTP 响应的类（HttpResponse）及其支持类
- 外观类（HttpRequestFacade 和 HTTPResponseFacade）
- 常量类

核心模块：servletProcessor 类和 staticResourceProcessor。
注：
HTTP 请求对象使用 HTTPRequest 类表示，在调用 servlet 的 service() 时，HTTPRequest 会被向上转型，因此必须正确地设置每一个 HTTPRequest 会用到的参数：URI、查询字符串、参数、Cookie 和其他请求头。但连接器并不知道 servlet 会使用哪些，所以连接器必须解析所有从 HTTP 请求中获取的所有信息。

解析 HTTP 请求设计开销大的字符串操作，因此可以让连接器只解析用到的数据。

Tomcat 默认连接器和本节使用 SocketInputStream 类从套接字的 InputStream 中读取字节流。SocketInputStream 的 readRequestLine() 会返回 HTTP 请求中第一行内容（必须在 readHeader() 前调用），每回调用 readHeader() 会返回一个名/值对，可以重复调用直到获取所有请求头信息。两者返回值分别为 HttpRequestLine 实例和 HttpHeader 对象。

等待 HTTP 请求由 HTTPConnector 完成，创建 Request 和 Response 由 HttpProcessor 完成。HttpProcessor 使用 parse() 解析 HTTP 请求行和请求头信息，填充到 HttpRequest 对象的成员变量中，但不负责解析请求体和查询字符串，这些由 HttpRequest 完成。

3.2.1 启动应用程序

```
1 public final class Bootstrap {
2     public static void main(String[] args) {
3         HttpConnector connector = new HttpConnector();
4         connector.start();
5     }
6 }
```

实例化 HttpConnector，并启动线程。

```
1  public class HttpConnector implements Runnable {
2
3      boolean stopped;
4      private String scheme = "http";
5
6      public String getScheme() {
7          return scheme;
8      }
9
10     public void run() {
11         ServerSocket serverSocket = null;
12         int port = 8080;
13         try {
14             serverSocket =  new ServerSocket(port, 1, InetAddress.
                   getByName("127.0.0.1"));
```

```
15          }
16          catch (IOException e) {
17              e.printStackTrace();
18              System.exit(1);
19          }
20          while (!stopped) {
21              // Accept the next incoming connection from the server
                   socket
22              Socket socket = null;
23              try {
24                  socket = serverSocket.accept();
25              }
26              catch (Exception e) {
27                  continue;
28              }
29              // Hand this socket off to an HttpProcessor
30              HttpProcessor processor = new HttpProcessor(this);
31              processor.process(socket);
32          }
33      }
34
35      public void start() {
36          Thread thread = new Thread(this);
37          thread.start();
38      }
39  }
```

3.2.2 HttpConnector 类

HttpConnector 实现了 java.lang.Runnable 接口。

HttpProcessor 类的 process() 接受传入的 HTTP 请求的套接字，然后完成下面 4 个操作：

- 创建一个 HttpRequest 对象；
- 创建一个 HttpResponse 对象；
- 解析 HTTP 请求的第一行内容的请求头信息，填充 HttpRequest 对象；

- 将 HttpRequest 对象和 HttpResponse 对象传递给 servletProcessor 或 StaticResourceProcessor 的 process()。

HttpProcessor 的 process() 方法：

```
1  public void process(Socket socket) {
2      SocketInputStream input = null;
3      OutputStream output = null;
4      try {
5          input = new SocketInputStream(socket.getInputStream(),
               2048);
6          output = socket.getOutputStream();
7
8          // create HttpRequest object and parse
9          request = new HttpRequest(input);
10
11         // create HttpResponse object
12         response = new HttpResponse(output);
13         response.setRequest(request);
14
15         response.setHeader("Server", "Pyrmont Servlet Container");
16
17         parseRequest(input, output);
18         parseHeaders(input);
19
20         //check if this is a request for a servlet or a static
               resource
21         //a request for a servlet begins with "/servlet/"
22         if (request.getRequestURI().startsWith("/servlet/")) {
23             ServletProcessor processor = new ServletProcessor();
24             processor.process(request, response);
25         }
26         else {
27             StaticResourceProcessor processor = new
                   StaticResourceProcessor();
28             processor.process(request, response);
29         }
```

```
30
31          // Close the socket
32          socket.close();
33          // no shutdown for this application
34      }
35      catch (Exception e) {
36          e.printStackTrace();
37      }
38  }
```

先获取套接字的输入输出流，然后创建一个 HttpRequest 实例和一个 HttpResponse 实例，然后 HttpRequest 实例赋值给 HTTPResponse 实例；

发送响应头信息，然后会调用 setHeader() 设置头信息，再调用私有方法解析请求，可以根据 URI 模式判断是 Servlet 还是静态资源，最后关闭套接字。

3.2.3 创建 HttpRequest 对象

HttpRequest 的请求头、Cookie 和请求参数数据：

```
1  protected HashMap headers = new HashMap();
2  protected ArrayList cookies = new ArrayList();
3  protected ParameterMap parameters = null;
```

1. 读取套接字的输入流
   可以按照前面的 java.io.InputStream 类的 read() 方法进行获取方法、URI、HTTP 协议的版本信息。

```
1  byte[] buffer = new byte[2048];
2  try {
3      i = input.read(buffer);
4  }
```

这里使用 SocketInputStream 进行解析，使用它是为了调用其方法 readRequestLine() 和 readHeader()：

```
1  SocketInputStream input = null;
```

```
2  OutputStream output = null;
3  try {
4      input = new SocketInputStream(socket.getInputStream(),
           2048);
5      //......
6  }
```

2. 解析请求行

HttpProcessor 的 process() 会调用私有方法 parseRequest() 解析请求行，请求行的 URI 可以包含 0 或多个请求参数。

在 servlet/JSP 中，参数名 jsessionid 用于携带一个会话标识符，会话标识符通常是作为 Cookie 嵌入的，但当浏览器禁用 Cookie 时，可以将它嵌入到查询符中。

```
1   private void parseRequest(SocketInputStream input,
        OutputStream output)
2  throws IOException, ServletException {
3
4      // Parse the incoming request line
5      input.readRequestLine(requestLine);
6      String method =
7      new String(requestLine.method, 0, requestLine.methodEnd);
8      String uri = null;
9      String protocol = new String(requestLine.protocol, 0,
           requestLine.protocolEnd);
10
11     // Validate the incoming request line
12     if (method.length() < 1) {
13         throw new ServletException("Missing HTTP request method
               ");
14     }
15     else if (requestLine.uriEnd < 1) {
16         throw new ServletException("Missing HTTP request URI");
17     }
18     // Parse any query parameters out of the request URI
19     int question = requestLine.indexOf("?");
20     if (question >= 0) {
```

```
21          request.setQueryString(new String(requestLine.uri,
                question + 1,
22          requestLine.uriEnd - question - 1));
23          uri = new String(requestLine.uri, 0, question);
24      }
25      else {
26          request.setQueryString(null);
27          uri = new String(requestLine.uri, 0, requestLine.uriEnd
                );
28      }
29
30
31      // Checking for an absolute URI (with the HTTP protocol)
32      if (!uri.startsWith("/")) {
33          int pos = uri.indexOf("://");
34          // Parsing out protocol and host name
35          if (pos != -1) {
36              pos = uri.indexOf('/', pos + 3);
37              if (pos == -1) {
38                  uri = "";
39              }
40              else {
41                  uri = uri.substring(pos);
42              }
43          }
44      }
45
46      // Parse any requested session ID out of the request URI
47      String match = ";jsessionid=";
48      int semicolon = uri.indexOf(match);
49      if (semicolon >= 0) {
50          String rest = uri.substring(semicolon + match.length())
                ;
51          int semicolon2 = rest.indexOf(';');
52          if (semicolon2 >= 0) {
53              request.setRequestedSessionId(rest.substring(0,
```

```
                semicolon2));
54              rest = rest.substring(semicolon2);
55          }
56          else {
57              request.setRequestedSessionId(rest);
58              rest = "";
59          }
60          request.setRequestedSessionURL(true);
61          uri = uri.substring(0, semicolon) + rest;
62      }
63      else {
64          request.setRequestedSessionId(null);
65          request.setRequestedSessionURL(false);
66      }
67
68      // Normalize URI (using String operations at the moment)
69      String normalizedUri = normalize(uri);
70
71      // Set the corresponding request properties
72      ((HttpRequest) request).setMethod(method);
73      request.setProtocol(protocol);
74      if (normalizedUri != null) {
75          ((HttpRequest) request).setRequestURI(normalizedUri);
76      }
77      else {
78          ((HttpRequest) request).setRequestURI(uri);
79      }
80
81      if (normalizedUri == null) {
82          throw new ServletException("Invalid URI: " + uri + "'")
                ;
83      }
84  }
```

首先会调用 SocketInputStream 类的 readRequestLine() 方法，填充 HttpRequestLine 实例；

然后判断 URI 是否包含参数，如果包含参数，对参数进行处理，否则直接当做 url 处理。

查询字符串可能会包含一个会话标识符 jsessionid，若存在参数 jsessionid，则表明会话标识符在查询字符串中，而不再 Cookie 中。
然后，会将 URI 传入到 normalize() 方法中，对非正常 URL 进行修正。

3. 解析请求头

   - 可以通过其类的无参构造函数创建一个 HttpHeader 实例；
   - 创建了 HttpHeader 实例后，可以将其传递给 SocketInputStream 的 readHeader() 方法，然后会填充 HttpHeader 对象。
   - 要获取请求头的名字和值，可以使用

```
1 String name = new String(header.name, 0, header.nameEnd);
2 String value = new String(header.value, 0, header.valueEnd)
    ;
```

   - parseHeaders() 会有一个 while 循环，直到读完为止。
   - 另外，一些请求头包含属性设置信息，如“content-length”和“cookies”

4. 解析 Cookie
   Cookie 是由浏览器作为 HTTP 请求头的一部分发送的，对 Cookie 的解析可以通过 org.apache.catalina.util.RequestUtil 类的 parseCookieHeader() 方法完成。

```
1  public static Cookie[] parseCookieHeader(String header) {
2
3      if ((header == null) || (header.length() < 1))
4      return (new Cookie[0]);
5
6      ArrayList cookies = new ArrayList();
7      while (header.length() > 0) {
8          int semicolon = header.indexOf(';');
9          if (semicolon < 0)
10         semicolon = header.length();
11         if (semicolon == 0)
12         break;
13         String token = header.substring(0, semicolon);
```

```
14          if (semicolon < header.length())
15          header = header.substring(semicolon + 1);
16          else
17          header = "";
18          try {
19              int equals = token.indexOf('=');
20              if (equals > 0) {
21                  String name = token.substring(0, equals).trim()
                        ;
22                  String value = token.substring(equals+1).trim()
                        ;
23                  cookies.add(new Cookie(name, value));
24              }
25          } catch (Throwable e) {
26              ;
27          }
28      }
29
30      return ((Cookie[]) cookies.toArray(new Cookie[cookies.size
            ()]));
31
32  }
```

5. 获取参数在调用 javax.servlet.http.HttpservletRequest 的 getParameter()、getParameterMap()、getParameterNames() 或 getParameterValues() 方法之前，都不需要解析查询字符串或 HTTP 请求体来获得参数，因此在 HttpRequest 类中，这 4 个方法实现都会先调用 parseParameter() 方法。
   参数只解析一次，HTTPRequest 中使用一个 parsed 判断是否完成解析。
   参数可以出现在查询字符串或请求体中，若使用 GET 方法请求，所有参数在查询字符串中，若使用 POST，则请求体中也可能会有参数。
   这里使用 java.util.HashMap 的子类 ParameterMap 存放参数，有一个 locked 判断是否可以修改。

```
1   public final class ParameterMap extends HashMap {
2
3       public ParameterMap() {
```

```
4           super();
5       }
6
7       public ParameterMap(int initialCapacity) {
8           super(initialCapacity);
9       }
10
11      public ParameterMap(int initialCapacity, float loadFactor)
            {
12          super(initialCapacity, loadFactor);
13      }
14
15      public ParameterMap(Map map) {
16          super(map);
17      }
18
19      private boolean locked = false;
20
21      public boolean isLocked() {
22
23          return (this.locked);
24
25      }
26
27      public void setLocked(boolean locked) {
28          this.locked = locked;
29      }
30
31
32      public void clear() {
33          if (locked)
34          throw new IllegalStateException
35          (sm.getString("parameterMap.locked"));
36          super.clear();
37      }
38
```

```
39      public Object put(Object key, Object value) {
40          if (locked)
41          throw new IllegalStateException
42          (sm.getString("parameterMap.locked"));
43          return (super.put(key, value));
44      }
45
46      public void putAll(Map map) {
47          if (locked)
48          throw new IllegalStateException
49          (sm.getString("parameterMap.locked"));
50          super.putAll(map);
51      }
52
53      public Object remove(Object key) {
54          if (locked)
55          throw new IllegalStateException
56          (sm.getString("parameterMap.locked"));
57          return (super.remove(key));
58      }
59  }
```

由于参数可以在查询字符串或 HTTP 请求体中，因此 parseParameters() 必须对二者检查，当解析完成时，参数会存储到对象变量 parameters 中，下面是解析过程代码：

```
1   protected void parseParameters() {
2       if (parsed)
3           return;
4       ParameterMap results = parameters;
5       if (results == null)
6           results = new ParameterMap();
7       results.setLocked(false);
8       String encoding = getCharacterEncoding();
9       if (encoding == null)
10          encoding = "ISO-8859-1";
11
```

```
12      // Parse any parameters specified in the query string
13      String queryString = getQueryString();
14      try {
15          RequestUtil.parseParameters(results, queryString,
               encoding);
16      }
17      catch (UnsupportedEncodingException e) {
18          ;
19      }
20
21      // Parse any parameters specified in the input stream
22      String contentType = getContentType();
23      if (contentType == null)
24          contentType = "";
25      int semicolon = contentType.indexOf(';');
26      if (semicolon >= 0) {
27          contentType = contentType.substring(0, semicolon).trim
               ();
28      }
29      else {
30          contentType = contentType.trim();
31      }
32      if ("POST".equals(getMethod()) && (getContentLength() > 0)
33      && "application/x-www-form-urlencoded".equals(contentType))
            {
34          try {
35              int max = getContentLength();
36              int len = 0;
37              byte buf[] = new byte[getContentLength()];
38              ServletInputStream is = getInputStream();
39              while (len < max) {
40                  int next = is.read(buf, len, max - len);
41                  if (next < 0 ) {
42                      break;
43                  }
44                  len += next;
```

```
45                }
46                is.close();
47                if (len < max) {
48                    throw new RuntimeException("Content length
                      mismatch");
49                }
50                RequestUtil.parseParameters(results, buf, encoding)
                      ;
51            }
52            catch (UnsupportedEncodingException ue) {
53                ;
54            }
55            catch (IOException e) {
56                throw new RuntimeException("Content read fail");
57            }
58        }
59
60        // Store the final results
61        results.setLocked(true);
62        parsed = true;
63        parameters = results;
64    }
```

首先会判断是否解析过参数，如果没有，则进行解析；
打开 parameterMap 的锁，再检查字符串 encoding，若为 null，则设置为"ISO-8859-1"，然后使用 org.apache.Catalina.util.RequestUtil 的 parseParameters() 完成。
然后，parseParameters() 会检查 HTTP 请求体是否包含参数，若使用 POST，请求体包含参数，且“content-length”的值会大于 0，“content-type”的值为“pplication/x-www-form-urlencoded”。

3.2.4 创建 HttpResponse 对象

这里为了解决 print() 和 println() 不会自动将结果发送到客户端，使用一个新的 writer 来 flush() 数据，这里可以使用 ResponseStream 的帝乡来实例化 ResponseWriter，（实际上使用了 java.io.OutputStreamWriter）告诉 writer 输出流，并实现自动发送结果。
使用 OutputStreamWriter 会将传入的字符转换为使用指定字符集的字节数组。

```
1  public PrintWriter getWriter() throws IOException {
2      ResponseStream newStream = new ResponseStream(this);
3      newStream.setCommit(false);
4      OutputStreamWriter osr =
5          new OutputStreamWriter(newStream, getCharacterEncoding());
6      writer = new ResponseWriter(osr);
7      return writer;
8  }
```

3.2.5 静态资源处理器和 servlet 处理器

与之前的处理 servlet 的方法类似，只有一个方法：process()，但是不同的是，这里的参数是 HttpRequest 和 HttpResponse，而不是 Request 和 Response，并且这里使用外观类作为调用后，使用 finshResponse() 处理连接。

```
1  servlet = (Servlet) myClass.newInstance();
2  HttpRequestFacade requestFacade = new HttpRequestFacade(request);
3  HttpResponseFacade responseFacade = new HttpResponseFacade(response
       );
4  servlet.service(requestFacade, responseFacade);
5  ((HttpResponse) response).finishResponse();
```

第四章　Tomcat 的默认连接器

Tomcat 的连接器有默认连接器，还有 Coyote、mod_jk 和 mod_webapp 等，并必须满足下面要求：

- 实现 org.apache.catalina.Connector 接口；
- 负责创建实现了 org.apache.catalina.Request 接口的 request 对象；
- 负责创建实现了 org.apache.catalina.Response 接口的 response 对象。

默认连接器原理：它会等待引入的 HTTP 请求，创建 request 对象和 response 对象，然后调用 org.apache.catalina.Container 接口的 invoke() 方法，将 request 对象和 response 对象传给 servlet 容器。

相对之前的连接器，使用优化方法：使用一个对象池避免了频繁的创建对象带来的性能损耗，其次，使用字符数组代替字符串。

Tomcat 的默认连接器支持 HTTP1.1 以及以前版本。

4.1　HTTP1.1 的新特性

4.1.1　持久连接

HTTP1.1 之前，无论浏览器何时连接服务器，当服务器处理请求后，就会断开与浏览器的连接，但是页面的引用资源就需要另开新连接获取，因此处理过程很慢。

使用持久连接，下载页面后，服务器不会关闭连接，等待客户端请求引用资源，HTTP 1.1 默认使用持久连接，可以显式使用：

connection: keep-alive

4.1.2　块编码

建立持久连接后，服务器可以从多个资源发送字节流，客户端也可以使用该连接发送多个请求，因此，发送方必须在每个请求或响应加上“content-length”头信息，这样接收方才能解释接受到的字节信息，但发送方有时不知道发送多少个字节，servlet 容器可以在接受一部分信息就能响应消息，因此可以使得接收方不知道发送内容长度也可以解析内容。

在 HTTP 1.0 中，服务器可以不写“content-length”头信息，尽管往连接中写响应信息即可，发送完信息后，直接关闭连接，而客户端会一直读取内容，直到读方法返回-1。

HTTP 1.1 使用“transfer-encoding”的特殊请求头指明字节流将会分块发送。对每个块，块的长度（16 进制）会有一个回车，然后是回车/换行符（CR/LF），然后是具体的数据，一个事务以一个长度为 0 的块标记，表示事务完成。

4.1.3 状态码 100 的使用

使用 HTTP 1.1 的客户端可以向服务器发送请求体之前发送如下请求头，并等待服务器的确认：Except: 100-continue

当客户端发送一个较长请求体之前，不确定服务端是否会接受，就可能使用上述头信息，服务器发送下面表明接收：HTTP/1.1 100 Continue。

注：返回内容后面要加上 CRLF 字符，然后服务器继续读取输入流信息。

4.2 Connector 接口

Tomcat 连接器需要实现 org.apache.catalina.Connector 接口，并实现 getContainer()、setContainer()、createRequest() 和 createResponse() 方法。

setContainer() 将连接器与某个 servlet 容器相关联；getContainer() 返回当前连接器的 servlet 容器；createRequest() 为引入的 HTTP 请求创建 request 对象；createResponse() 创建一个 response 对象。

4.3 HttpConnector 类

4.3.1 创建服务器套接字

HttpConnector 类的 initialize() 方法会调用私有办法 open()，后者返回一个 ServerSocket 实例，赋值给成员变量 serverSocket，但没有使用它的构造函数，而是使用工厂 ServerSocketFactory 获取实例。

4.3.2 维护 HttpProcessor 实例

Tomcat 的默认连接器中，HttpConnector 实例有一个 HttpProcessor 对象池，每个 HttpProcessor 实例都运行在自己的线程中，以应对对个 HTTP 请求。

HTTPProcessor 对象池：

```
1 private Stack processors = new Stack();
```

```
protected int minProcessors = 5;
private int maxProcessors = 20;
```

创建的 HttpProcessor 实例数目由 minProcessors 和 maxProcessors 决定，默认 min=5，max=20。初始时，会创建 min 个实例等待，当请求数目超过时，会创建更多的 HttpProcessor 实例，直到 max，如果还不够，则忽略请求。

如果希望 HttpConnector 能够持续创建 HttpProcessor 实例，可以将 max 设置为负数，且 curProcessor 保存了当前 HttpProcessor 实例的数目。

HttpConnector 的 start() 创建初始数量 HttpProcessor 实例：

```
// Create the specified minimum number of processors
while (curProcessors < minProcessors) {
    if ((maxProcessors > 0) && (curProcessors >= maxProcessors))
        break;
    HttpProcessor processor = newProcessor();
    recycle(processor);
}
```

HttpProcessor 负责解析 HTTP 的请求行和请求头，它的构造函数会调用 HttpConnector 的 createRequest 和 createResponse 方法。

4.3.3 提供 HTTP 请求服务

对于每个引入的 HTTP 请求，通过调用私有方法 createProcessor() 获得一个 HttpProcessor 对象，可以从对象池中获取，如果对象池中有可用，则直接返回，否则判断 min 和 max 大小进行进一步生成。

```
if (processor == null) {
    try {
        log(sm.getString("httpConnector.noProcessor"));
        socket.close();
    } catch (IOException e) {
        ;
    }
    continue;
}
processor.assign(socket);
```

如果为空，说明连接已达上限，直接关闭 socket，否则设置 processor 的 socket。

4.4 HttpProcessor 类

HttpProcessor 的 process() 负责解析 HTTP 请求。

之前的 HttpConnector 只能等待 HttpProcessor 处理完毕后才能处理下一个请求（同步），因此，在默认连接器中，HttpProcessor 实现了 java.lang.Runnable 接口，可以在自己的“处理器线程”处理请求，而不阻塞 HttpConnector 线程。

run() 方法中 while 执行到 await() 方法时阻塞，await() 会阻塞处理器，直到它从 HttpConnector 中获取到了新的 Socket 对象，在调用 assign 之前一直阻塞，但 assign 并不是与 await 在同一个线程中。

```
synchronized void assign(Socket socket) {

    // Wait for the Processor to get the previous Socket
    while (available) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }

    // Store the newly available Socket and notify our thread
    this.socket = socket;
    available = true;
    notifyAll();

    if ((debug >= 1) && (socket != null))
    log(" An incoming request is being assigned");

}
private synchronized Socket await() {

    // Wait for the Connector to provide a new Socket
    while (!available) {
        try {
```

```
25                wait();
26            } catch (InterruptedException e) {
27            }
28        }
29
30        // Notify the Connector that we have received this Socket
31        Socket socket = this.socket;
32        available = false;
33        notifyAll();
34
35        if ((debug >= 1) && (socket != null))
36        log("  The incoming request has been awaited");
37
38        return (socket);
39
40    }
```

处理器线程刚启动，available 变为 false，会等待其他线程调用 notify() 或 notifyAll() 方法，直到连接器线程调用了实例 HttpProcessor 的 notifyAll() 方法。

当新的 Socket 对象通过 assign() 传入时，连接器线程会调用 HttpProcessor 实例的 assign() 方法，由于变量 available 的值为 false，因此会调出 while，传入的 Socket 会赋值给 HttpProcessor 的 socket 变量。

然后连接器线程会将 available 设置为 true，并调用 notifyAll()，唤醒处理器线程，这样处理器线程会跳出 while 循环，将成员变量赋值给一个局部变量，然后调用 notifyAll()，将局部变量返回，由其他程序继续处理。

注：

为什么 await() 方法使用局部变量 socket，而不直接返回？因为使用局部变量可以在当前 Socket 对象处理完之前，继续接收下一个 Socket 对象。

为什么 await() 需要调用 notifyAll() 方法？是为了防止出现另一个 Socket 对象已经到达，而此时变量 available 还是 true。在这种情况下，“连接器线程”会在 assign() 方法内的循环中阻塞，直到“处理器线程”调用 notifyAll() 方法。

4.5 Request 对象

RequestBase 类直接实现了 org.apache.catalina.Request，而 RequestBase 类是 HttpRequest 类的父类，最终实现类是 HttpRequestImpl 类，而 HttpRequestImpl 类继承自 HttpRequest 类。

4.6 Response 对象

里面的继承关系和 Request 类似。

4.7 处理请求

当 Socket 对象被赋值给 HttpProcessor 后，HttpProcessor 的 run() 会调用 process() 方法，process() 方法会执行 3 个操作：解析连接、解析请求、解析请求头。

process() 使用 ok 表示处理过程是否有错误发生，finishResponse 表示是否调用 Response 接口的 finishResponse() 方法，keepAlive 是否为持久连接，stopped 表示是否被连接器终止，http11 表示是否支持 HTTP1.1 客户端。

在 process() 中，先执行 request 和 response 的初始化操作，然后调用 parseConnection()、parseRequest() 和 parseHeader() 方法解析 HTTP 请求，parseConnection() 获取请求使用的协议，并根据协议版本设置是否持久连接，如果在 http 请求头中发现“Except: 100-continue”，则 parseHeader() 将 sendAck 设置为 true。

若请求协议 HTTP 1.1,而且客户端发出了“Except: 100-continue”,会对调用 ackRequest() 请求该方法进行响应，另外还会检查是否允许分块。

```
1 //响应信息
2 HTTP/1.1 100 Continue
```

完成解析后，process() 将 request 和 response 对象作为参数传入 servlet 容器的 invoke() 方法。

```
1 try {
2     ((HttpServletResponse) response).setHeader
3     ("Date", FastHttpDateFormat.getCurrentDate());
4     if (ok) {
5         connector.getContainer().invoke(request, response);
6     }
7 }
```

然后，如果 finishResponse 为 true，则调用 response 对象的 finishResponse() 方法和 request 对象的 finishRequest() 方法，将结果发送至客户端。最后，检查 response 的头信息“Connection”是否在 servlet 中设置为 close 或者协议为 1.0，如果任一种为真，则将 keepAlive 设置为 false，最后对对象 request 和 response 进行回收。

keepAlive 为 true，或前面解析没有错误，或 HttpProcessor 没有被终止，则 while 将继续运行，否则调用 shutdownInput()，并关闭套接字。

4.7.1 解析连接

parseConnection() 会从套接字中获取 Internet 地址，将其赋值给 HttpRequestImpl 对象，并且会检查是否使用了代理，将 Socket 对象赋值给 request 对象。

```
1  private void parseConnection(Socket socket)
2  throws IOException, ServletException {
3      if (debug >= 2)
4      log(" parseConnection: address=" + socket.getInetAddress() +
5      ", port=" + connector.getPort());
6      ((HttpRequestImpl) request).setInet(socket.getInetAddress());
7      if (proxyPort != 0)
8      request.setServerPort(proxyPort);
9      else
10     request.setServerPort(serverPort);
11     request.setSocket(socket);
12 }
```

4.7.2 解析请求

parseRequest() 与之前版本类似。

4.7.3 解析请求头

默认连接器中的 parseHeaders() 方法使用 org.apache.catalina.http 包中 HttpHeader 类和 DefaultHeader 类。

parseHeaders() 使用 while 读取所有 HTTP 的请求信息，while 循环调用 request 对象的 allocateHeader() 获取一个内容为空的 HttpHeader 实例，然后，该实例会被传入 SocketInputStream 的 readHeader() 方法中。

若所有请求头都已经读取过了，则 readHeader() 不会再给 HttpHeader 实例设置 name 属性，这时就可以退出 parseHeader() 方法。

如果一个 HttpHeader 有 name，那么它肯定有 value：

```
1  private void parseHeaders(SocketInputStream input) throws
       IOException, ServletException {
2      while (true) {
3          HttpHeader header = request.allocateHeader();
4
5          // Read the next header
6          input.readHeader(header);
7          if (header.nameEnd == 0) {
8              if (header.valueEnd == 0) {
9                  return;
10             } else {
11                 throw new ServletException
12                 (sm.getString("httpProcessor.parseHeaders.colon"));
13             }
14         }
15
16         String value = new String(header.value, 0, header.valueEnd)
               ;
```

然后，将读取到的请求头的 name 属性值与 DefaultHeaders 中的标准名称比较，这里比较的是字符数组而不是字符串。

```
1  // Set the corresponding request headers
2  if (header.equals(DefaultHeaders.AUTHORIZATION_NAME)) {
3      request.setAuthorization(value);
4  } else if (header.equals(DefaultHeaders.ACCEPT_LANGUAGE_NAME)) {
5      parseAcceptLanguage(value);
6  } else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
7      //parse cookie
8  } else if (header.equals(DefaultHeaders.CONTENT_LENGTH_NAME)) {
9      //get content legth
10 } else if (header.equals(DefaultHeaders.CONTENT_TYPE_NAME)) {
11     request.setContentType(value);
12 } else if (header.equals(DefaultHeaders.HOST_NAME)) {
13     //get host name
14 } else if (header.equals(DefaultHeaders.CONNECTION_NAME)) {
```

```
15      if (header.valueEquals
16      (DefaultHeaders.CONNECTION_CLOSE_VALUE)) {
17          keepAlive = false;
18          response.setHeader("Connection", "close");
19      }
20  //request.setConnection(header);
21  /*
22  if ("keep-alive".equalsIgnoreCase(value)) {
23  keepAlive = true;
24  }
25  */
26  } else if (header.equals(DefaultHeaders.EXPECT_NAME)) {
27      if (header.valueEquals(DefaultHeaders.EXPECT_100_VALUE))
28          sendAck = true;
29      else
30          throw new ServletException (sm.getString ("httpProcessor.
               parseHeaders.unknownExpectation"));
31  } else if (header.equals(DefaultHeaders.TRANSFER_ENCODING_NAME)) {
32      //request.setTransferEncoding(header);
33  }
34
35  request.nextHeader();
```

4.8 简单的 Container 应用程序

SimpleContainer 实现了 org.catalina.Container 接口，它可以与默认连接器进行关联，Bootstrap 用于启动该应用程序，但移除了连接器模块，以及 ServletProcessor 和 StaticResourceProcessor 的使用。

```
1  public class SimpleContainer implements Container {
2
3      public static final String WEB_ROOT =
4      System.getProperty("user.dir") + File.separator  + "webroot";
5
6      public SimpleContainer() {
```

```
7       }
8
9       public String getInfo() {
10          return null;
11      }
12
13      public Loader getLoader() {
14          return null;
15      }
16
17      public void setLoader(Loader loader) {
18      }
19
20      public Logger getLogger() {
21          return null;
22      }
23
24      public void setLogger(Logger logger) {
25      }
26
27      public Manager getManager() {
28          return null;
29      }
30
31      public void setManager(Manager manager) {
32      }
33
34      public Cluster getCluster() {
35          return null;
36      }
37
38      public void setCluster(Cluster cluster) {
39      }
40
41      public String getName() {
42          return null;
```

```
43      }
44
45      public void setName(String name) {
46      }
47
48      public Container getParent() {
49          return null;
50      }
51
52      public void setParent(Container container) {
53      }
54
55      public ClassLoader getParentClassLoader() {
56          return null;
57      }
58
59      public void setParentClassLoader(ClassLoader parent) {
60      }
61
62      public Realm getRealm() {
63          return null;
64      }
65
66      public void setRealm(Realm realm) {
67      }
68
69      public DirContext getResources() {
70          return null;
71      }
72
73      public void setResources(DirContext resources) {
74      }
75
76      public void addChild(Container child) {
77      }
78
```

```
79      public void addContainerListener(ContainerListener listener) {
80      }
81
82      public void addMapper(Mapper mapper) {
83      }
84
85      public void addPropertyChangeListener(PropertyChangeListener
           listener) {
86      }
87
88      public Container findChild(String name) {
89          return null;
90      }
91
92      public Container[] findChildren() {
93          return null;
94      }
95
96      public ContainerListener[] findContainerListeners() {
97          return null;
98      }
99
100     public Mapper findMapper(String protocol) {
101         return null;
102     }
103
104     public Mapper[] findMappers() {
105         return null;
106     }
107
108     public void invoke(Request request, Response response)
109     throws IOException, ServletException {
110
111         String servletName = ( (HttpServletRequest) request).
               getRequestURI();
112         servletName = servletName.substring(servletName.lastIndexOf
```

```
            ("/") + 1);
113         URLClassLoader loader = null;
114         try {
115             URL[] urls = new URL[1];
116             URLStreamHandler streamHandler = null;
117             File classPath = new File(WEB_ROOT);
118             String repository = (new URL("file", null, classPath.
                    getCanonicalPath() + File.separator)).toString() ;
119             urls[0] = new URL(null, repository, streamHandler);
120             loader = new URLClassLoader(urls);
121         }
122         catch (IOException e) {
123             System.out.println(e.toString() );
124         }
125         Class myClass = null;
126         try {
127             myClass = loader.loadClass(servletName);
128         }
129         catch (ClassNotFoundException e) {
130             System.out.println(e.toString());
131         }
132
133         Servlet servlet = null;
134
135         try {
136             servlet = (Servlet) myClass.newInstance();
137             servlet.service((HttpServletRequest) request, (
                    HttpServletResponse) response);
138         }
139         catch (Exception e) {
140             System.out.println(e.toString());
141         }
142         catch (Throwable e) {
143             System.out.println(e.toString());
144         }
145
```

```
146
147
148     }
149
150     public Container map(Request request, boolean update) {
151         return null;
152     }
153
154     public void removeChild(Container child) {
155     }
156
157     public void removeContainerListener(ContainerListener listener)
            {
158     }
159
160     public void removeMapper(Mapper mapper) {
161     }
162
163     public void removePropertyChangeListener(PropertyChangeListener
            listener) {
164     }
165
166 }
```

默认连接器会调用这里的 invoke()，invoke() 会创建一个类载入器，载入相关的 servlet 类，并调用 servlet 类的 service() 方法，该方法与之前的 ServletProcessor 的 process() 类似。

```
1  public final class Bootstrap {
2      public static void main(String[] args) {
3          HttpConnector connector = new HttpConnector();
4          SimpleContainer container = new SimpleContainer();
5          connector.setContainer(container);
6          try {
7              connector.initialize();
8              connector.start();
9
```

```
10              // make the application wait until we press any key.
11              System.in.read();
12          }
13          catch (Exception e) {
14              e.printStackTrace();
15          }
16      }
17  }
```

Bootstrap 类的 main() 方法分别创建了 org.apache.catalina.connector.http.HttpConnector 类和 SimpleContainer 的一个实例，然后调用连接器的 setContainer() 将连接器和 servlet 容器相关联，接下来，调用连接器的 initialize() 和 start() 方法。

第五章　servlet 容器

servlet 容器是用来处理请求 servlet 资源，并为 Web 客户端填充 response 对象的模块。Tomcat 中有 4 种容器：Engine、Host、Context 和 Wrapper。

5.1　Container 接口

Tomcat 的 servlet 容器必须实现 org.apache.catalina.Container 接口，需要将 servlet 容器的实例作为参数传入连接器的 setContainer() 方法中，以此使用 invoke()。

4 种容器：

- Engine：表示整个 Catania servlet 引擎；
- Host：表示包含一个或多个 Context 容器的虚拟主机；
- Context：表示一个 Web 应用程序，一个 Context 可以有多个 Wrapper；
- Wrapper：表示一个独立的 servlet。

4 个接口的标准实现分别是 StandardEngine 类、StandardHost 类、StandardContext 类和 StandardWrapper 类，均在 org.apache.catalina.core 包内。

一个容器可以有 0 个或多个低层级的子容器。一般情况下，一个 Context 会有一个或多个 Wrapper 实例，一个 Host 实例会有 0 个或多个 Context 实例，Wrapper 处于最底层，不在包含子容器。

可以调用 Container 接口的 addChild() 方法向某容器中添加子容器。

public void addChild(Container child);

Container 接口提供 findChild() 方法和 findChildren() 方法来查找子容器和所有子容器的某个容器。

```
1 public Container findChild(String name);
2 public Container[] findChildren();
```

Container 接口的设计满足：在部署应用时，Tomcat 可以通过编辑配置文件 (server.xml) 来决定使用哪种容器。

5.2 管道任务

相关接口：Pipeline、Value、ValueContext 和 Contained。

管道包含该 servlet 将要调用的任务，一个阀表示一个具体的执行任务。

在 servlet 容器中，有一个基础阀，可以添加任意数量阀，阀的数量指的是额外添加阀数量，基础阀总是最后一个执行。

一个 servlet 容器可以有一条管道，当调用了容器的 invoke()，容器会将处理工作交由阀继续执行任务，直到所有的阀都处理完成。