第三部分

网络协议

WebSocket 是一种为了提高 Web 应用程序的性能以及响应性而开发的先进的网络协议。我们将通过编写一个简单的示例应用程序来探索 Netty 对它们的支持。

在第 12 章中，通过构建一个可以在多个浏览器客户端之间进行实时通信的聊天室，你将学习到如何使用 WebSocket 来实现双向数据传输。你还将会看到如何在你的应用程序中通过检测客户端是否支持 WebSocket 协议，从而从 HTTP 协议切换到 WebSocket 协议。

通过对第 13 章中 Netty 对于用户数据报协议（UDP）的支持的学习，我们将结束第三部分。在这一章中，你将会构建可适用于多种实际用途的广播服务器和监视器客户端。

第 12 章　WebSocket

本章主要内容

- 实时 Web 的概念
- WebSocket 协议
- 使用 Netty 构建一个基于 WebSocket 的聊天室服务器

如果你有跟进 Web 技术的最新进展，你很可能就遇到过“实时 Web”这个短语，而如果你在工程领域中有实时应用程序的实战经验，那么你可能有点怀疑这个术语到底意味着什么。

因此，让我们首先澄清，这里并不是指所谓的硬实时服务质量（QoS），硬实时服务质量是保证计算结果将在指定的时间间隔内被递交。仅 HTTP 的请求/响应模式设计就使得其很难被支持，从过去所设计的各种方案中都没有提供一种能够提供令人满意的解决方案的事实中便可见一斑。

虽然已经有了一些关于正式定义实时Web服务[①]语义的学术讨论，但是被普遍接受的定义似乎还未出现。因此现在我们将采纳下面来自维基百科的非权威性描述：

> 实时 Web 利用技术和实践，使用户在信息的作者发布信息之后就能够立即收到信息，而不需要他们或者他们的软件周期性地检查信息源以获取更新。

简而言之，虽然全面的实时Web可能并不会马上到来，但是它背后的想法却助长了对于几乎瞬时获得信息的期望。我们将在本章中讨论的WebSocket[②]协议便是在这个方向上迈出的坚实的一步。

12.1　WebSocket 简介

WebSocket 协议是完全重新设计的协议，旨在为 Web 上的双向数据传输问题提供一个切

① “Real-time Web Services Orchestration and Choreography”：http://ceur-ws.org/Vol-601/EOMAS10_paper13.pdf。
② IETF RFC 6455, The WebSocket Protocol: http://tools.ietf.org/html/rfc6455。

实可行的解决方案，使得客户端和服务器之间可以在任意时刻传输消息，因此，这也就要求它们异步地处理消息回执。(作为 HTML5 客户端 API 的一部分，大部分最新的浏览器都已经支持了 WebSocket。)

Netty 对于 WebSocket 的支持包含了所有正在使用中的主要实现，因此在你的下一个应用程序中采用它将是简单直接的。和往常使用 Netty 一样，你可以完全使用该协议，而无需关心它内部的实现细节。我们将通过创建一个基于 WebSocket 的实时聊天应用程序来演示这一点。

12.2 我们的 WebSocket 示例应用程序

为了让示例应用程序展示它的实时功能，我们将通过使用 WebSocket 协议来实现一个基于浏览器的聊天应用程序，就像你可能在 Facebook 的文本消息功能中见到过的那样。我们将通过使得多个用户之间可以同时进行相互通信，从而更进一步。

图 12-1 说明了该应用程序的逻辑：

（1）客户端发送一个消息；

（2）该消息将被广播到所有其他连接的客户端。

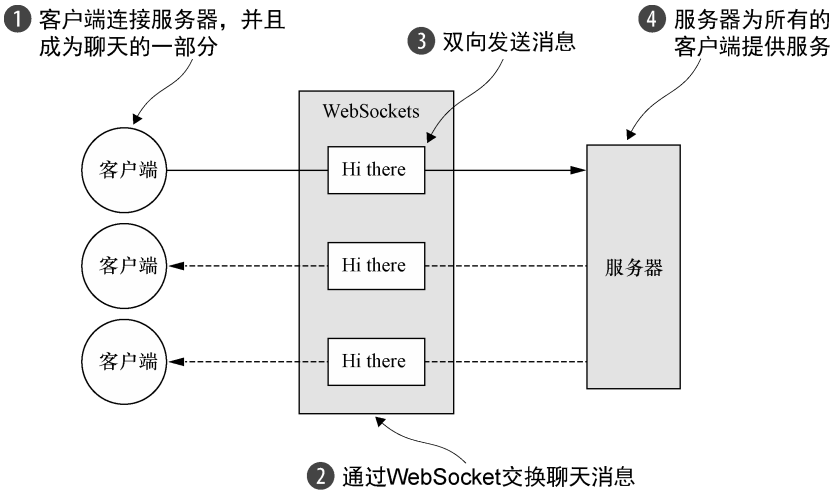

图 12-1 WebSocket 应用程序逻辑

这正如你可能会预期的一个聊天室应当的工作方式：所有的人都可以和其他的人聊天。在示例中，我们将只实现服务器端，而客户端则是通过 Web 页面访问该聊天室的浏览器。正如同你将在接下来的几页中所看到的，WebSocket 简化了编写这样的服务器的过程。

12.3 添加 WebSocket 支持

在从标准的HTTP或者HTTPS协议切换到WebSocket时，将会使用一种称为升级握手[①]的机制。因此，使用WebSocket的应用程序将始终以HTTP/S作为开始，然后再执行升级。这个升级动作发生的确切时刻特定于应用程序；它可能会发生在启动时，也可能会发生在请求了某个特定的URL之后。

我们的应用程序将采用下面的约定：如果被请求的 URL 以/ws 结尾，那么我们将会把该协议升级为 WebSocket；否则，服务器将使用基本的 HTTP/S。在连接已经升级完成之后，所有数据都将会使用 WebSocket 进行传输。图 12-2 说明了该服务器逻辑，一如在 Netty 中一样，它由一组 ChannelHandler 实现。我们将会在下一节中，解释用于处理 HTTP 以及 WebSocket 协议的技术时，描述它们。

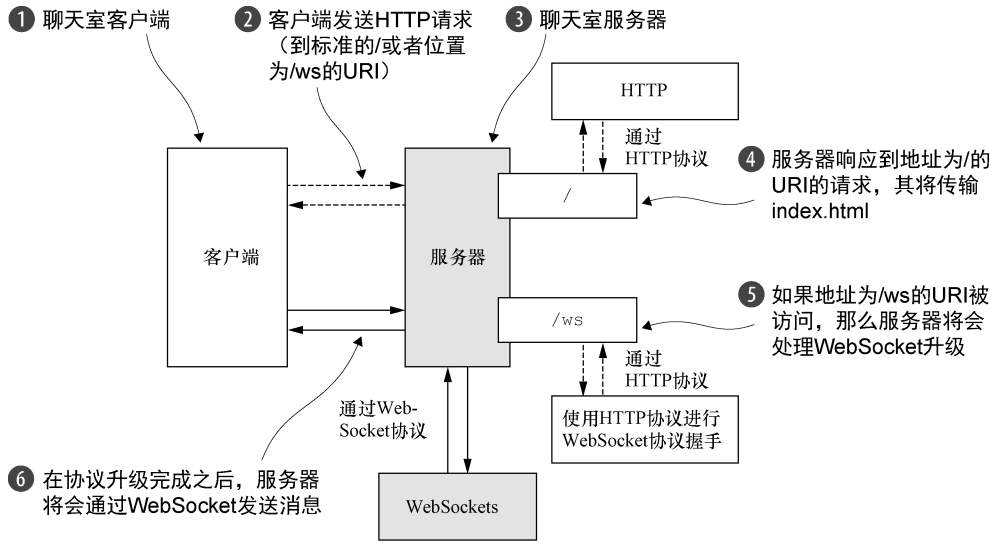

图 12-2 服务器逻辑

12.3.1 处理 HTTP 请求

首先，我们将实现该处理 HTTP 请求的组件。这个组件将提供用于访问聊天室并显示由连接的客户端发送的消息的网页。代码清单 12-1 给出了这个 HttpRequestHandler 对应的代码，其扩展了 SimpleChannelInboundHandler 以处理 FullHttpRequest 消息。需要注意的

① Mozilla 开发者网络，“Protocol upgrade mechanism”：https://developer.mozilla.org/en-US/docs/HTTP/Protocol_upgrade_mechanism。

是，channelRead0()方法的实现是如何转发任何目标 URI 为/ws 的请求的。

代码清单 12-1 HTTPRequestHandler

```
public class HttpRequestHandler
    extends SimpleChannelInboundHandler<FullHttpRequest> {
    private final String wsUri;
    private static final File INDEX;

    static {
        URL location = HttpRequestHandler.class
            .getProtectionDomain()
            .getCodeSource().getLocation();
        try {
            String path = location.toURI() + "index.html";
            path = !path.contains("file:") ? path : path.substring(5);
            INDEX = new File(path);
        } catch (URISyntaxException e) {
            throw new IllegalStateException(
                "Unable to locate index.html", e);
        }
    }

    public HttpRequestHandler(String wsUri) {
        this.wsUri = wsUri;
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
        FullHttpRequest request) throws Exception {
        if (wsUri.equalsIgnoreCase(request.getUri())) {
            ctx.fireChannelRead(request.retain());
        } else {
            if (HttpHeaders.is100ContinueExpected(request)) {
                send100Continue(ctx);
            }
            RandomAccessFile file = new RandomAccessFile(INDEX, "r");
            HttpResponse response = new DefaultHttpResponse(
                request.getProtocolVersion(), HttpResponseStatus.OK);
            response.headers().set(
                HttpHeaders.Names.CONTENT_TYPE,
                "text/plain; charset=UTF-8");
            boolean keepAlive = HttpHeaders.isKeepAlive(request);
            if (keepAlive) {
                response.headers().set(
                    HttpHeaders.Names.CONTENT_LENGTH, file.length());
                response.headers().set( HttpHeaders.Names.CONNECTION,
                    HttpHeaders.Values.KEEP_ALIVE);
            }
            ctx.write(response);
            if (ctx.pipeline().get(SslHandler.class) == null) {
                ctx.write(new DefaultFileRegion(
                    file.getChannel(), 0, file.length()));
            } else {
```

扩展 SimpleChannelInboundHandler 以处理 FullHttpRequest 消息

❶ 如果请求了 WebSocket 协议升级，则增加引用计数（调用 retain()方法），并将它传递给下一个 ChannelInboundHandler

❷ 处理 100 Continue 请求以符合 HTTP 1.1 规范

读取 index.html

如果请求了 keep-alive，则添加所需要的 HTTP 头信息

❸ 将 HttpResponse 写到客户端

❹ 将 index.html 写到客户端

```
                ctx.write(new ChunkedNioFile(file.getChannel()));
            }
            ChannelFuture future = ctx.writeAndFlush(
                LastHttpContent.EMPTY_LAST_CONTENT);
            if (!keepAlive) {
                future.addListener(ChannelFutureListener.CLOSE);
            }
        }
    }

    private static void send100Continue(ChannelHandlerContext ctx) {
        FullHttpResponse response = new DefaultFullHttpResponse(
            HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

❺ 写 LastHttpContent 并冲刷至客户端

如果没有请求 keep-alive，则在写操作完成后关闭 Channel ❻

如果该 HTTP 请求指向了地址为/ws 的 URI，那么 HttpRequestHandler 将调用 FullHttpRequest 对象上的 retain()方法，并通过调用 fireChannelRead(msg)方法将它转发给下一个 ChannelInboundHandler❶。之所以需要调用 retain()方法，是因为调用 channelRead()方法完成之后，它将调用 FullHttpRequest 对象上的 release()方法以释放它的资源。（参见我们在第 6 章中对于 SimpleChannelInboundHandler 的讨论。）

如果客户端发送了 HTTP 1.1 的 HTTP 头信息 Expect: 100-continue，那么 HttpRequestHandler 将会发送一个 100 Continue❷响应。在该 HTTP 头信息被设置之后，HttpRequestHandler 将会写回一个 HttpResponse❸给客户端。这不是一个 FullHttpResponse，因为它只是响应的第一个部分。此外，这里也不会调用 writeAndFlush()方法，在结束的时候才会调用。

如果不需要加密和压缩，那么可以通过将 index.html❹的内容存储到 DefaultFileRegion 中来达到最佳效率。这将会利用零拷贝特性来进行内容的传输。为此，你可以检查一下，是否有 SslHandler 存在于在 ChannelPipeline 中。否则，你可以使用 ChunkedNioFile。

HttpRequestHandler 将写一个 LastHttpContent❺来标记响应的结束。如果没有请求 keep-alive❻，那么 HttpRequestHandler 将会添加一个 ChannelFutureListener 到最后一次写出动作的 ChannelFuture，并关闭该连接。在这里，你将调用 writeAndFlush()方法以冲刷所有之前写入的消息。

这部分代码代表了聊天服务器的第一个部分，它管理纯粹的 HTTP 请求和响应。接下来，我们将处理传输实际聊天消息的 WebSocket 帧。

WEBSOCKET 帧 WebSocket 以帧的方式传输数据，每一帧代表消息的一部分。一个完整的消息可能会包含许多帧。

12.3.2 处理 WebSocket 帧

由 IETF 发布的 WebSocket RFC，定义了 6 种帧，Netty 为它们每种都提供了一个 POJO 实现。表 12-1 列出了这些帧类型，并描述了它们的用法。

表 12-1 **WebSocketFrame 的类型**

帧 类 型	描 述
BinaryWebSocketFrame	包含了二进制数据
TextWebSocketFrame	包含了文本数据
ContinuationWebSocketFrame	包含属于上一个 BinaryWebSocketFrame 或 TextWebSocketFrame 的文本数据或者二进制数据
CloseWebSocketFrame	表示一个 CLOSE 请求，包含一个关闭的状态码和关闭的原因
PingWebSocketFrame	请求传输一个 PongWebSocketFrame
PongWebSocketFrame	作为一个对于 PingWebSocketFrame 的响应被发送

我们的聊天应用程序将使用下面几种帧类型：

- CloseWebSocketFrame；
- PingWebSocketFrame；
- PongWebSocketFrame；
- TextWebSocketFrame。

TextWebSocketFrame 是我们唯一真正需要处理的帧类型。为了符合 WebSocket RFC，Netty 提供了 WebSocketServerProtocolHandler 来处理其他类型的帧。

代码清单 12-2 展示了我们用于处理 TextWebSocketFrame 的 ChannelInboundHandler，其还将在它的 ChannelGroup 中跟踪所有活动的 WebSocket 连接。

代码清单 12-2 处理文本帧

```
                                                    扩展 SimpleChannelInboundHandler，
                                                    并处理 TextWebSocketFrame 消息
public class TextWebSocketFrameHandler
    extends SimpleChannelInboundHandler<TextWebSocketFrame> {
    private final ChannelGroup group;

    public TextWebSocketFrameHandler(ChannelGroup group) {
        this.group = group;
```

```
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx,
        Object evt) throws Exception {
        if (evt == WebSocketServerProtocolHandler
            .ServerHandshakeStateEvent.HANDSHAKE_COMPLETE) {
            ctx.pipeline().remove(HttpRequestHandler.class);
            group.writeAndFlush(new TextWebSocketFrame(
                "Client " + ctx.channel() + " joined"));
            group.add(ctx.channel());
        } else {
            super.userEventTriggered(ctx, evt);
        }
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
        TextWebSocketFrame msg) throws Exception {
        group.writeAndFlush(msg.retain());
    }
}
```

重写 userEventTriggered() 方法以处理自定义事件

如果该事件表示握手成功，则从该 Channelipeline 中移除 HttpRequestHandler，因为将不会接收到任何 HTTP 消息了

❶通知所有已经连接的 WebSocket 客户端新的客户端已经连接上了

将新的 WebSocket Channel 添加到 ChannelGroup 中，以便它可以接收到所有的消息 ❷

❸增加消息的引用计数，并将它写到 ChannelGroup 中所有已经连接的客户端

TextWebSocketFrameHandler 只有一组非常少量的责任。当和新客户端的 WebSocket 握手成功完成之后❶，它将通过把通知消息写到 ChannelGroup 中的所有 Channel 来通知所有已经连接的客户端，然后它将把这个新 Channel 加入到该 ChannelGroup 中❷。

如果接收到了 TextWebSocketFrame 消息❸，TextWebSocketFrameHandler 将调用 TextWebSocketFrame 消息上的 retain() 方法，并使用 writeAndFlush() 方法来将它传输给 ChannelGroup，以便所有已经连接的 WebSocket Channel 都将接收到它。

和之前一样，对于 retain() 方法的调用是必需的，因为当 channelRead0() 方法返回时，TextWebSocketFrame 的引用计数将会被减少。由于所有的操作都是异步的，因此，writeAndFlush() 方法可能会在 channelRead0() 方法返回之后完成，而且它绝对不能访问一个已经失效的引用。

因为 Netty 在内部处理了大部分剩下的功能，所以现在剩下唯一需要做的事情就是为每个新创建的 Channel 初始化其 ChannelPipeline。为此，我们将需要一个 ChannelInitializer。

12.3.3 初始化 ChannelPipeline

正如你已经学习到的，为了将 ChannelHandler 安装到 ChannelPipeline 中，你扩展了 ChannelInitializer，并实现了 initChannel() 方法。代码清单 12-3 展示了由此生成的 ChatServerInitializer 的代码。

代码清单 12-3 初始化 ChannelPipeline

```
public class ChatServerInitializer extends ChannelInitializer<Channel> {
    private final ChannelGroup group;

    public ChatServerInitializer(ChannelGroup group) {
        this.group = group;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new HttpServerCodec());
        pipeline.addLast(new ChunkedWriteHandler());
        pipeline.addLast(new HttpObjectAggregator(64 * 1024));
        pipeline.addLast(new HttpRequestHandler("/ws"));
        pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
        pipeline.addLast(new TextWebSocketFrameHandler(group));
    }
}
```

扩展了 ChannelInitializer

将所有需要的 ChannelHandler 添加到 ChannelPipeline 中

对于 initChannel()方法的调用，通过安装所有必需的 ChannelHandler 来设置该新注册的 Channel 的 ChannelPipeline。这些 ChannelHandler 以及它们各自的职责都被总结在了表 12-2 中。

表 12-2 基于 WebSocket 聊天服务器的 ChannelHandler

ChannelHandler	职 责
HttpServerCodec	将字节解码为 HttpRequest、HttpContent 和 LastHttpContent。并将 HttpRequest、HttpContent 和 LastHttpContent 编码为字节
ChunkedWriteHandler	写入一个文件的内容
HttpObjectAggregator	将一个 HttpMessage 和跟随它的多个 HttpContent 聚合为单个 FullHttpRequest 或者 FullHttpResponse(取决于它是被用来处理请求还是响应)。安装了这个之后，ChannelPipeline 中的下一个 ChannelHandler 将只会收到完整的 HTTP 请求或响应
HttpRequestHandler	处理 FullHttpRequest（那些不发送到/ws URI 的请求）
WebSocketServerProtocolHandler	按照 WebSocket 规范的要求，处理 WebSocket 升级握手、PingWebSocketFrame 、 PongWebSocketFrame 和 CloseWebSocketFrame
TextWebSocketFrameHandler	处理 TextWebSocketFrame 和握手完成事件

Netty 的 WebSocketServerProtocolHandler 处理了所有委托管理的 WebSocket 帧类型以及升级握手本身。如果握手成功，那么所需的 ChannelHandler 将会被添加到 ChannelPipeline 中，而那些不再需要的 ChannelHandler 则将会被移除。

WebSocket 协议升级之前的 ChannelPipeline 的状态如图 12-3 所示。这代表了刚刚被 ChatServerInitializer 初始化之后的 ChannelPipeline。

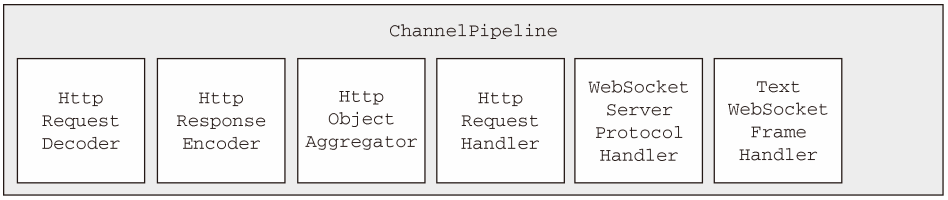

图 12-3 WebSocket 协议升级之前的 ChannelPipeline

当 WebSocket 协议升级完成之后，WebSocketServerProtocolHandler 将会把 HttpRequestDecoder 替换为 WebSocketFrameDecoder，把 HttpResponseEncoder 替换为 WebSocketFrameEncoder。为了性能最大化，它将移除任何不再被 WebSocket 连接所需要的 ChannelHandler。这也包括了图 12-3 所示的 HttpObjectAggregator 和 HttpRequestHandler。

图 12-4 展示了这些操作完成之后的ChannelPipeline。需要注意的是，Netty目前支持 4 个版本的WebSocket协议，它们每个都具有自己的实现类。Netty将会根据客户端（这里指浏览器）所支持的版本 [①]，自动地选择正确版本的WebSocketFrameDecoder和WebSocketFrameEncoder。

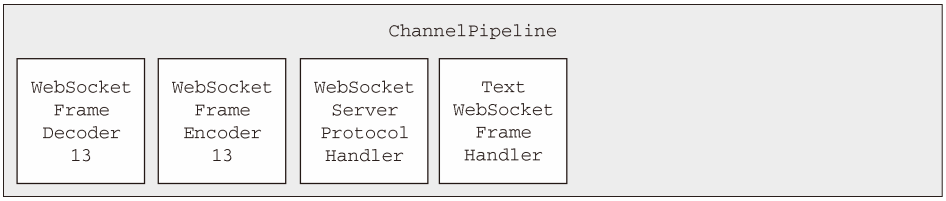

图 12-4 WebSocket 协议升级完成之后的 ChannelPipeline

12.3.4 引导

这幅拼图最后的一部分是引导该服务器，并安装 ChatServerInitializer 的代码。这将由 ChatServer 类处理，如代码清单 12-4 所示。

代码清单 12-4 引导服务器

```
public class ChatServer {
```

① 在这个例子中，我们假设使用了 13 版的 WebSocket 协议，所以图中展示的是 WebSocketFrameDecoder13 和 WebSocketFrameEncoder13。

```
private final ChannelGroup channelGroup =
    new DefaultChannelGroup(ImmediateEventExecutor.INSTANCE);
private final EventLoopGroup group = new NioEventLoopGroup();
private Channel channel;

public ChannelFuture start(InetSocketAddress address) {
    ServerBootstrap bootstrap = new ServerBootstrap();
    bootstrap.group(group)
        .channel(NioServerSocketChannel.class)
        .childHandler(createInitializer(channelGroup));
    ChannelFuture future = bootstrap.bind(address);
    future.syncUninterruptibly();
    channel = future.channel();
    return future;
}

 protected ChannelInitializer<Channel> createInitializer(
     ChannelGroup group) {
     return new ChatServerInitializer(group);
}

public void destroy() {
    if (channel != null) {
        channel.close();
    }
    channelGroup.close();
    group.shutdownGracefully();
}

public static void main(String[] args) throws Exception {
    if (args.length != 1) {
        System.err.println("Please give port as argument");
        System.exit(1);
    }
    int port = Integer.parseInt(args[0]);
    final ChatServer endpoint = new ChatServer();
    ChannelFuture future = endpoint.start(
        new InetSocketAddress(port));
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            endpoint.destroy();
        }
    });
    future.channel().closeFuture().syncUninterruptibly();
```

创建 DefaultChannelGroup，其将保存所有已经连接的 WebSocket Channel

引导服务器

创建 ChatServerInitializer

处理服务器关闭，并释放所有的资源

```
    }
}
```

这也就完成了该应用程序本身。现在让我们来测试它吧。

12.4 测试该应用程序

目录 chapter12 中的示例代码包含了你需要用来构建并运行该服务器的所有资源。(如果你还没有设置好你的包括 Apache Maven 在内的开发环境，参见第 2 章中的操作说明。)

我们将使用下面的 Maven 命令来构建和启动服务器：

```
mvn -PChatServer clean package exec:exec
```

项目文件 pom.xml 被配置为在端口 9999 上启动服务器。如果要使用不同的端口，可以通过编辑文件中对应的值，或者使用一个 System 属性来对它进行重写：

```
mvn -PChatServer -Dport=1111 clean package exec:exec
```

代码清单 12-5 展示了该命令主要的输出(无关紧要的行已经被删除了)。

代码清单 12-5 编译并运行 ChatServer

```
$ mvn -PChatServer clean package exec:exec

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting ChatServer on port 9999
```

你通过将自己的浏览器指向 http://localhost:9999 来访问该应用程序。图 12-5 展示了其在 Chrome 浏览器中的 UI。

图中展示了两个已经连接的客户端。第一个客户端是使用上面的界面连接的，第二个客户端则是通过底部的 Chrome 浏览器的命令行工具连接的。你会注意到，两个客户端都发送了消息，并且每个消息都显示在两个客户端中。

这是一个非常简单的演示，演示了 WebSocket 如何在浏览器中实现实时通信。

图 12-5 基于 WebSocket 的 ChatServer 的演示

如何进行加密

在真实世界的场景中，你将很快就会被要求向该服务器添加加密。使用 Netty，这不过是将一个 SslHandler 添加到 ChannelPipeline 中，并配置它的问题。代码清单 12-6 展示了如何通过扩展我们的ChatServerInitializer来创建一个SecureChatServerInitializer以完成这个需求。

代码清单 12-6 为 ChannelPipeline 添加加密

```
public class SecureChatServerInitializer extends ChatServerInitializer {
    private final SslContext context;

    public SecureChatServerInitializer(ChannelGroup group,
        SslContext context) {
        super(group);
        this.context = context;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        super.initChannel(ch);
        SSLEng.ine engine = context.newEngine(ch.alloc());
        engine.setUseClientMode(false);
```

扩展 ChatServerInitializer 以添加加密

调用父类的 initChannel()方法

```
        ch.pipeline().addFirst(new SslHandler(engine));
    }
}
```

将 SslHandler 添加到 ChannelPipeline 中

最后一步是调整 ChatServer 以使用 SecureChatServerInitializer，以便在 ChannelPipeline 中安装 SslHandler。这给了我们代码清单 12-7 中所展示的 SecureChatServer。

代码清单 12-7 向 ChatServer 添加加密

```
public class SecureChatServer extends ChatServer {
    private final SslContext context;

    public SecureChatServer(SslContext context) {
        this.context = context;
    }

    @Override
    protected ChannelInitializer<Channel> createInitializer(
        ChannelGroup group) {
        return new SecureChatServerInitializer(group, context);
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);
        SelfSignedCertificate cert = new SelfSignedCertificate();
        SslContext context = SslContext.newServerContext(
        cert.certificate(), cert.privateKey());

        final SecureChatServer endpoint = new SecureChatServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

SecureChatServer 扩展 ChatServer 以支持加密

返回之前创建的 SecureChatServer-Initializer 以启用加密

这就是为所有的通信启用 SSL/TLS 加密需要做的全部。和之前一样，可以使用 Apache Maven 来运行该应用程序，如代码清单 12-8 所示。它还将检索任何所需的依赖。

代码清单 12-8 启动 SecureChatServer

```
$ mvn -PSecureChatServer clean package exec:exec
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
```

```
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting SecureChatServer on port 9999
```

现在,你便可以从 `SecureChatServer` 的 HTTPS URL 地址 https://localhost:9999 访问它了。

12.5 小结

在本章中，你学习了如何使用 Netty 的 WebSocket 实现来管理 Web 应用程序中的实时数据。我们覆盖了其所支持的数据类型，并讨论了你可能会遇到的一些限制。尽管不可能在所有的情况下都使用 WebSocket，但是仍然需要清晰地认识到，它代表了 Web 技术的一个重要进展。

第 13 章　使用 UDP 广播事件

本章主要内容

- UDP 概述
- 一个示例广播应用程序

到目前为止，你所见过的绝大多数的例子都使用了基于连接的协议，如TCP。在本章中，我们将会把重点放在一个无连接协议即用户数据报协议（UDP）上，它通常用在性能至关重要并且能够容忍一定的数据包丢失的情况下[①]。

我们将会首先概述 UDP 的特性以及它的局限性。在这之后，我们将描述本章的示例应用程序，其将演示如何使用 UDP 的广播能力。我们还会使用一个编码器和一个解码器来处理作为广播消息格式的 POJO。在本章的结束时候，你将能够在自己的应用程序中使用 UDP。

13.1　UDP 的基础知识

面向连接的传输（如 TCP）管理了两个网络端点之间的连接的建立，在连接的生命周期内的有序和可靠的消息传输，以及最后，连接的有序终止。相比之下，在类似于 UDP 这样的无连接协议中，并没有持久化连接这样的概念，并且每个消息（一个 UDP 数据报）都是一个单独的传输单元。

此外，UDP 也没有 TCP 的纠错机制，其中每个节点都将确认它们所接收到的包，而没有被确认的包将会被发送方重新传输。

通过类比，TCP 连接就像打电话，其中一系列的有序消息将会在两个方向上流动。相反，UDP 则类似于往邮箱中投入一叠明信片。你无法知道它们将以何种顺序到达它们的目的地，或者它们是否所有的都能够到达它们的目的地。

UDP的这些方面可能会让你感觉到严重的局限性，但是它们也解释了为何它会比TCP快那么

① 最有名的基于 UDP 的协议之一便是域名服务（DNS），其将完全限定的名称映射为数字的 IP 地址。

多：所有的握手以及消息管理机制的开销都已经被消除了。显然，UDP很适合那些能够处理或者容忍消息丢失的应用程序，但可能不适合那些处理金融交易的应用程序[①]。

13.2 UDP 广播

到目前为止，我们所有的例子采用的都是一种叫作*单播*[②]的传输模式，定义为发送消息给一个由唯一的地址所标识的单一的网络目的地。面向连接的协议和无连接协议都支持这种模式。

UDP 提供了向多个接收者发送消息的额外传输模式：

- *多播*——传输到一个预定义的主机组；
- *广播*——传输到网络（或者子网）上的所有主机。

本章中的示例应用程序将通过发送能够被同一个网络中的所有主机所接收的消息来演示 UDP 广播的使用。为此，我们将使用特殊的受限广播地址或者零网络地址 `255.255.255.255`。发送到这个地址的消息都将会被定向给本地网络（`0.0.0.0`）上的所有主机，而不会被路由器转发给其他的网络。

接下来，我们将讨论该应用程序的设计。

13.3 UDP 示例应用程序

我们的示例程序将打开一个文件，随后将会通过 UDP 把每一行都作为一个消息广播到一个指定的端口。如果你熟悉类 UNIX 操作系统，你可能会认识到这是标准的 *syslog* 实用程序的一个非常简化的版本。UDP 非常适合于这样的应用程序，因为考虑到日志文件本身已经被存储在了文件系统中，因此，偶尔丢失日志文件中的一两行是可以容忍的。此外，该应用程序还提供了极具价值的高效处理大量数据的能力。

接收方是怎么样的呢？通过 UDP 广播，只需简单地通过在指定的端口上启动一个监听程序，便可以创建一个事件监视器来接收日志消息。需要注意的是，这样的轻松访问性也带来了潜在的安全隐患，这也就是为何在不安全的环境中并不倾向于使用 UDP 广播的原因之一。出于同样的原因，路由器通常也会阻止广播消息，并将它们限制在它们的来源网络上。

发布/订阅模式 类似于 *syslog* 这样的应用程序通常会被归类为发布/订阅模式：一个生产者或者服务发布事件，而多个客户端进行订阅以接收它们。

图 13-1 展示了整个系统的一个高级别视图，其由一个广播者以及一个或者多个事件监视器所组成。广播者将监听新内容的出现，当它出现时，则通过 UDP 将它作为一个广播消息进行传输。

---

① 基于 UDP 协议实现的一些可靠传输协议可能不在此范畴内，如 Quic、Aeron 和 UDT。——译者注

② 参见 http://en.wikipedia.org/wiki/Unicast。

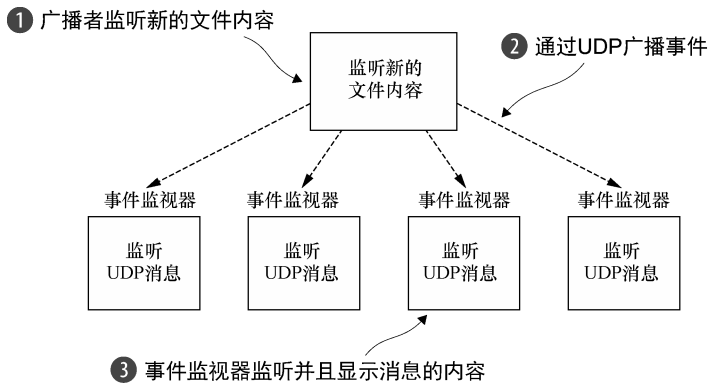

图 13-1 广播系统概览

所有的在该 UDP 端口上监听的事件监视器都将会接收到广播消息。

为了简单起见，我们将不会为我们的示例程序添加身份认证、验证或者加密。但是，要加入这些功能并使得其成为一个健壮的、可用的实用程序应该也不难。

在下一节中，我们将开始探讨该广播者组件的设计以及实现细节。

13.4 消息 POJO: LogEvent

在消息处理应用程序中，数据通常由 POJO 表示，除了实际上的消息内容，其还可以包含配置或处理信息。在这个应用程序中，我们将会把消息作为事件处理，并且由于该数据来自于日志文件，所以我们将它称为 `LogEvent`。代码清单 13-1 展示了这个简单的 POJO 的详细信息。

代码清单 13-1 `LogEvent` 消息

```
public final class LogEvent {
    public static final byte SEPARATOR = (byte) ':';
    private final InetSocketAddress source;
    private final String logfile;
    private final String msg;
    private final long received;

    public LogEvent(String logfile, String msg) {
        this(null, -1, logfile, msg);
    }

    public LogEvent(InetSocketAddress source, long received,
        String logfile, String msg) {
        this.source = source;
        this.logfile = logfile;
        this.msg = msg;
        this.received = received;
```

用于传出消息的构造函数

用于传入消息的构造函数

```
    }

    public InetSocketAddress getSource() {
        return source;
    }

    public String getLogfile() {
        return logfile;
    }

    public String getMsg() {
        return msg;
    }

    public long getReceivedTimestamp() {
        return received;
    }
}
```

返回发送 LogEvent 的源的 InetSocketAddress

返回所发送的 LogEvent 的日志文件的名称

返回消息内容

返回接收 LogEvent 的时间

定义好了消息组件，我们便可以实现该应用程序的广播逻辑了。在下一节中，我们将研究用于编码和传输 LogEvent 消息的 Netty 框架类。

13.5 编写广播者

Netty 提供了大量的类来支持 UDP 应用程序的编写。表 13-1 列出了我们将要使用的主要的消息容器以及 Channel 类型。

表 13-1 在广播者中使用的 Netty 的 UDP 相关类

名 称	描 述
interface AddressedEnvelope <M, A extends SocketAddress> extends ReferenceCounted	定义一个消息，其包装了另一个消息并带有发送者和接收者地址。其中 M 是消息类型；A 是地址类型
class DefaultAddressedEnvelope <M, A extends SocketAddress> implements AddressedEnvelope<M,A>	提供了 interface AddressedEnvelope 的默认实现
class DatagramPacket extends DefaultAddressedEnvelope <ByteBuf, InetSocketAddress> implements ByteBufHolder	扩展了 DefaultAddressedEnvelope 以使用 ByteBuf 作为消息数据容器
interface DatagramChannel extends Channel	扩展了 Netty 的 Channel 抽象以支持 UDP 的多播组管理
class NioDatagramChannnel extends AbstractNioMessageChannel implements DatagramChannel	定义了一个能够发送和接收 AddressedEnvelope 消息的 Channel 类型

Netty 的 DatagramPacket 是一个简单的消息容器，DatagramChannel 实现用它来和远程节点通信。类似于在我们先前的类比中的明信片，它包含了接收者（和可选的发送者）的地址以及消息的有效负载本身。

要将 LogEvent 消息转换为 DatagramPacket，我们将需要一个编码器。但是没有必要从头开始编写我们自己的。我们将扩展 Netty 的 MessageToMessageEncoder，在第 10 章和第 11 章中我们已经使用过了。

图 13-2 展示了正在广播的 3 个日志条目，每一个都将通过一个专门的 DatagramPacket 进行广播。

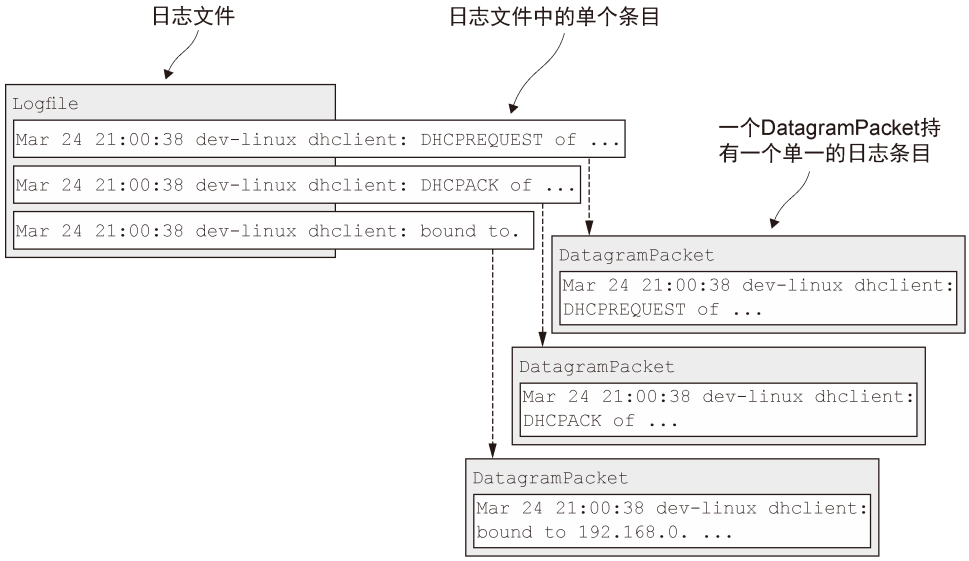

图 13-2 通过 DatagramPacket 发送的日志条目

图 13-3 呈现了该 LogEventBroadcaster 的 ChannelPipeline 的一个高级别视图，展示了 LogEvent 消息是如何流经它的。

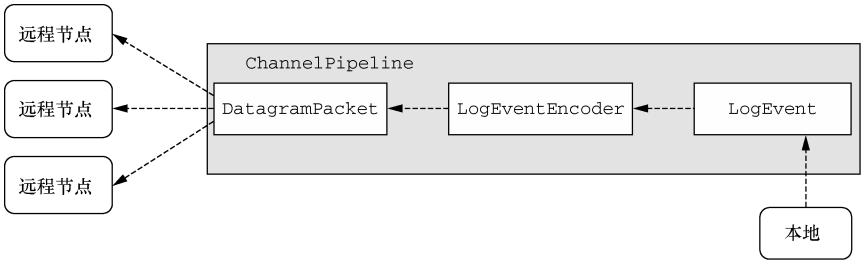

图 13-3 LogEventBroadcaster：ChannelPipeline 和 LogEvent 事件流

正如你所看到的，所有的将要被传输的数据都被封装在了 LogEvent 消息中。LogEvent-Broadcaster 将把这些写入到 Channel 中，并通过 ChannelPipeline 发送它们，在那里它们将会被转换（编码）为 DatagramPacket 消息。最后，他们都将通过 UDP 被广播，并由远程节点（监视器）所捕获。

代码清单 13-2 展示了我们自定义版本的 MessageToMessageEncoder，其将执行刚才所描述的转换。

代码清单 13-2 LogEventEncoder

```
public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {
    private final InetSocketAddress remoteAddress;

    public LogEventEncoder(InetSocketAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext,
        LogEvent logEvent, List<Object> out) throws Exception {
        byte[] file = logEvent.getLogfile().getBytes(CharsetUtil.UTF_8);
        byte[] msg = logEvent.getMsg().getBytes(CharsetUtil.UTF_8);
        ByteBuf buf = channelHandlerContext.alloc()
            .buffer(file.length + msg.length + 1);
        buf.writeBytes(file);
        buf.writeByte(LogEvent.SEPARATOR);
        buf.writeBytes(msg);
        out.add(new DatagramPacket(buf, remoteAddress));
    }
}
```

LogEventEncoder 创建了即将被发送到指定的 InetSocketAddress 的 DatagramPacket 消息

将文件名写入到 ByteBuf 中

添加一个 SEPARATOR

将日志消息写入 ByteBuf 中

将一个拥有数据和目的地地址的新 DatagramPacket 添加到出站的消息列表中

在 LogEventEncoder 被实现之后，我们已经准备好了引导该服务器，其包括设置各种各样的 ChannelOption，以及在 ChannelPipeline 中安装所需要的 ChannelHandler。这将通过主类 LogEventBroadcaster 完成，如代码清单 13-3 所示。

代码清单 13-3 LogEventBroadcaster

```
public class LogEventBroadcaster {
    private final EventLoopGroup group;
    private final Bootstrap bootstrap;
    private final File file;

    public LogEventBroadcaster(InetSocketAddress address, File file) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
        bootstrap.group(group).channel(NioDatagramChannel.class)
            .option(ChannelOption.SO_BROADCAST, true)
            .handler(new LogEventEncoder(address));
        this.file = file;
    }
```

引导该 NioDatagram-Channel（无连接的）

设置 SO_BROADCAST 套接字选项

```
    public void run() throws Exception {
        Channel ch = bootstrap.bind(0).sync().channel();
        long pointer = 0;
        for (;;) {
            long len = file.length();
            if (len < pointer) {
                // file was reset
                pointer = len;
            } else if (len > pointer) {
                // Content was added
                RandomAccessFile raf = new RandomAccessFile(file, "r");
                raf.seek(pointer);
                String line;
                while ((line = raf.readLine()) != null) {
                    ch.writeAndFlush(new LogEvent(null, -1,
                    file.getAbsolutePath(), line));
                }
                pointer = raf.getFilePointer();
                raf.close();
             }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.interrupted();
                break;
            }
        }
    }
    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new IllegalArgumentException();
        }

        LogEventBroadcaster broadcaster = new LogEventBroadcaster(
            new InetSocketAddress("255.255.255.255",
                Integer.parseInt(args[0])), new File(args[1]));
        try {
            broadcaster.run();
        }
        finally {
            broadcaster.stop();
        }
    }
}
```

绑定 Channel

启动主处理循环

如果有必要，将文件指针设置到该文件的最后一个字节

设置当前的文件指针，以确保没有任何的旧日志被发送

对于每个日志条目，写入一个 LogEvent 到 Channel 中

存储其在文件中的当前位置

休眠 1 秒，如果被中断，则退出循环；否则重新处理它

创建并启动一个新的 LogEventBroadcaster 的实例

这样就完成了该应用程序的广播者组件。对于初始测试，你可以使用*netcat*程序。在

UNIX/Linux 系统中，你能发现它已经作为 *nc* 被预装了。用于 Windows 的版本可以从 http://nmap.org/ncat 获取 ①。

netcat 非常适合于对这个应用程序进行基本的测试；它只是监听某个指定的端口，并且将所有接收到的数据打印到标准输出。可以通过下面所示的方式，将其设置为监听 UDP 端口 9999 上的数据：

```
$ nc -l -u -p 9999
```

现在我们需要启动我们的 `LogEventBroadcaster`。代码清单 13-4 展示了如何使用 `mvn` 来编译和运行该广播者应用程序。`pom.xml` 文件中的配置指向了一个将被频繁更新的文件，`/var/log/messages`（假设是一个 UNIX/Linux 环境），并将端口设置为了 9999。该文件中的条目将会通过 UDP 广播到那个端口，并在你启动了 netcat 的终端上打印出来。

代码清单 13-4 编译和启动 LogEventBroadcaster

```
$ chapter13> mvn clean package exec:exec LogEventBroadcaster
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building UDP Broadcast 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: target/chapter13-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
 LogEventBroadcaster running
```

要改变该日志文件和端口值，可以在启动 `mvn` 的时候通过 `System` 属性来指定它们。代码清单 13-5 展示了如何将日志文件设置为 `/var/log/mail.log`，并将端口设置为 `8888`。

代码清单 13-5 编译和启动 LogEventBroadcaster

```
$ chapter13> mvn clean package exec:exec -PLogEventBroadcaster /
-Dlogfile=/var/log/mail.log -Dport=8888 -....
....
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
LogEventBroadcaster running
```

当你看到 `LogEventBroadcaster running` 时，你便知道它已经成功地启动了。如果有错误发生，将会打印一个异常消息。一旦这个进程运行起来，它就会广播任何新被添加到该日志文件中的日志消息。

使用 *netcat* 对于测试来说是足够了，但是它并不适合于生产系统。这也就有了我们的应用程序的第二个部分——我们将在下一节中实现的广播监视器。

---

① 也可以使用 `scoop install netcat`。——译者注

13.6 编写监视器

我们的目标是将 *netcat* 替换为一个更加完整的事件消费者，我们称之为 LogEventMonitor。这个程序将：

（1）接收由 LogEventBroadcaster 广播的 UDP DatagramPacket；

（2）将它们解码为 LogEvent 消息；

（3）将 LogEvent 消息写出到 System.out。

和之前一样，该逻辑由一组自定义的 ChannelHandler 实现——对于我们的解码器来说，我们将扩展 MessageToMessageDecoder。图 13-4 描绘了 LogEventMonitor 的 ChannelPipeline，并且展示了 LogEvent 是如何流经它的。

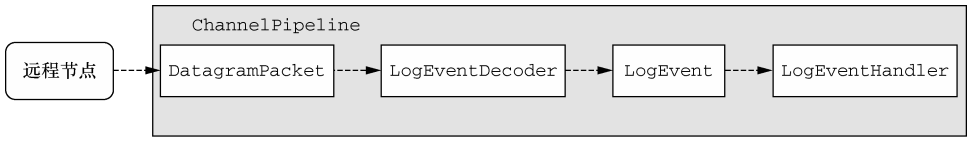

图 13-4 LogEventMonitor

ChannelPipeline 中的第一个解码器 LogEventDecoder 负责将传入的 DatagramPacket 解码为 LogEvent 消息（一个用于转换入站数据的任何 Netty 应用程序的典型设置）。代码清单 13-6 展示了该实现。

代码清单 13-6 LogEventDecoder

```
public class LogEventDecoder extends MessageToMessageDecoder<DatagramPacket> {

    @Override
    protected void decode(ChannelHandlerContext ctx,
        DatagramPacket datagramPacket, List<Object> out) throws Exception {
        ByteBuf data = datagramPacket.content();
        int idx = data.indexOf(0, data.readableBytes(),
            LogEvent.SEPARATOR);
        String filename = data.slice(0, idx)
            .toString(CharsetUtil.UTF_8);
        String logMsg = data.slice(idx + 1,
            data.readableBytes()).toString(CharsetUtil.UTF_8);

        LogEvent event = new LogEvent(datagramPacket.sender(),
            System.currentTimeMillis(), filename, logMsg);
        out.add(event);
    }
}
```

获取对 DatagramPacket 中的数据（ByteBuf）的引用

获取该 SEPARATOR 的索引

提取文件名

提取日志消息

构建一个新的 LogEvent 对象，并且将它添加到（已经解码的消息的）列表中

第二个 ChannelHandler 的工作是对第一个 ChannelHandler 所创建的 LogEvent 消息执行一些处理。在这个场景下，它只是简单地将它们写出到 System.out。在真实世界的应用程序中，你可能需要聚合来源于不同日志文件的事件，或者将它们发布到数据库中。代码清单 13-7 展示了 LogEventHandler，其说明了需要遵循的基本步骤。

代码清单 13-7 LogEventHandler

```
public class LogEventHandler
    extends SimpleChannelInboundHandler<LogEvent> {

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) throws Exception {
        cause.printStackTrace();
        ctx.close();
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
        LogEvent event) throws Exception {
        StringBuilder builder = new StringBuilder();
        builder.append(event.getReceivedTimestamp());
        builder.append(" [");
        builder.append(event.getSource().toString());
        builder.append("] [");
        builder.append(event.getLogfile());
        builder.append("] : ");
        builder.append(event.getMsg());
        System.out.println(builder.toString());
    }
}
```

扩展 SimpleChannelInbound-Handler 以处理 LogEvent 消息

当异常发生时，打印栈跟踪信息，并关闭对应的 Channel

创建 StringBuilder，并且构建输出的字符串

打印 LogEvent 的数据

LogEventHandler 将以一种简单易读的格式打印 LogEvent 消息，包括以下的各项：

- 以毫秒为单位的被接收的时间戳；
- 发送方的 InetSocketAddress，其由 IP 地址和端口组成；
- 生成 LogEvent 消息的日志文件的绝对路径名；
- 实际上的日志消息，其代表日志文件中的一行。

现在我们需要将我们的 LogEventDecoder 和 LogEventHandler 安装到 ChannelPipeline 中，如图 13-4 所示。代码清单 13-8 展示了如何通过 LogEventMonitor 主类来做到这一点。

代码清单 13-8 LogEventMonitor

```
public class LogEventMonitor {
    private final EventLoopGroup group;
    private final Bootstrap bootstrap;

    public LogEventMonitor(InetSocketAddress address) {
        group = new NioEventLoopGroup();
```

```
        bootstrap = new Bootstrap();
        bootstrap.group(group)
             .channel(NioDatagramChannel.class)
             .option(ChannelOption.SO_BROADCAST, true)
             .handler( new ChannelInitializer<Channel>() {
                 @Override
                 protected void initChannel(Channel channel)
                     throws Exception {
                     ChannelPipeline pipeline = channel.pipeline();
                     pipeline.addLast(new LogEventDecoder());
                     pipeline.addLast(new LogEventHandler());
                 }
             } )
             .localAddress(address);
    }

    public Channel bind() {
        return bootstrap.bind().syncUninterruptibly().channel();
    }

    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] main) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException(
            "Usage: LogEventMonitor <port>");
        }
        LogEventMonitor monitor = new LogEventMonitor(
            new InetSocketAddress(Integer.parseInt(args[0])));
        try {
            Channel channel = monitor.bind();
            System.out.println("LogEventMonitor running");
            channel.closeFuture().sync();
        } finally {
            monitor.stop();
        }
    }
}
```

引导该 NioDatagramChannel

设置套接字选项 SO_BROADCAST

将 LogEventDecoder 和 LogEventHandler 添加到 ChannelPipeline 中

绑定 Channel。注意，DatagramChannel 是无连接的

构造一个新的 LogEventMonitor

13.7 运行 LogEventBroadcaster 和 LogEventMonitor

和之前一样，我们将使用 Maven 来运行该应用程序。这一次你将需要打开两个控制台窗口，每个都将运行一个应用程序。每个应用程序都将会在直到你按下了 Ctrl+C 组合键来停止它之前一直保持运行。

首先，你需要启动 LogEventBroadcaster，因为你已经构建了该工程，所以下面的命令应该就足够了（使用默认值）：

```
$ chapter13> mvn exec:exec -PLogEventBroadcaster
```

和之前一样，这将通过 UDP 协议广播日志消息。

现在，在一个新窗口中，构建并且启动 LogEventMonitor 以接收和显示广播消息，如代码清单 13-9 所示。

代码清单 13-9 编译并启动 LogEventBroadcaster

```
$ chapter13> mvn clean package exec:exec -PLogEventMonitor
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building UDP Broadcast 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: target/chapter14-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
LogEventMonitor running
```

当你看到 LogEventMonitor running 时，你将知道它已经成功地启动了。如果有错误发生，则将会打印异常信息。

如代码清单 13-10 所示，当任何新的日志事件被添加到该日志文件中时，该终端都会显示它们。消息的格式则是由 LogEventHandler 创建的。

代码清单 13-10 LogEventMonitor 的输出

```
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
     dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
     port 67
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
     dev-linux dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
     dev-linux dhclient: bound to 192.168.0.50 -- renewal in 270 seconds.
1364217299382 [/192.168.0.38:63182] [[/var/log/messages] : Mar 25 13:59:38
     dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
      port 67
1364217299382 [/192.168.0.38:63182] [/[/var/log/messages] : Mar 25 13:59:38
     dev-linux dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:59:38
     dev-linux dhclient: bound to 192.168.0.50 -- renewal in 259 seconds.
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
     dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
     port 67
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
     dev-linux dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
     dev-linux dhclient: bound to 192.168.0.50 -- renewal in 285 seconds.
```

如果你不能访问 UNIX 的 syslog，那么你可以创建一个自定义的文件，并手动提供内容以观测该应用程序的反应。以使用 touch 命令来创建一个空文件作为开始，下面所展示的步骤使用了 UNIX 命令。

```
$ touch ~/mylog.log
```

现在再次启动 LogEventBroadcaster，并通过设置系统属性来将其指向该文件：

```
$ chapter13> mvn exec:exec -PLogEventBroadcaster -Dlogfile=~/mylog.log
```

一旦 LogEventBroadcaster 运行，你就可以手动将消息添加到该文件中，以在 LogEventMonitor 终端中查看广播输出。使用 echo 命令并将输出重定向到该文件，如下所示：

```
$ echo 'Test log entry' >> ~/mylog.log
```

你可以根据需要启动任意多的监视器实例，它们每一个都将接收并显示相同的消息。

13.8 小结

在本章中，我们使用 UDP 作为例子介绍了无连接协议。我们构建了一个示例应用程序，其将日志条目转换为 UDP 数据报并广播它们，随后这些被广播出去的消息将被订阅的监视器客户端所捕获。我们的实现使用了一个 POJO 来表示日志数据，并通过一个自定义的编码器来将这个消息格式转换为 Netty 的 DatagramPacket。这个例子说明了 Netty 的 UDP 应用程序可以很轻松地被开发和扩展用以支持专业化的用途。

在接下来的两章中，我们将把目光投向由知名公司的用户所提供的案例研究上，他们已使用 Netty 构建了工业级别的应用程序。