36 如何增加安全性

更新时间：2020-08-27 10:04:18

> 人的差异在于业余时间。——爱因斯坦

前言

你好，我是彤哥。

上一节，我们一起从扩展性的角度将实战项目的通信协议修改为了 WebSocket，有了 WebSocket，服务端可能很轻松地适配各端各语言，但是，如果我们的应用是暴露在外网的，还缺乏安全性。

因此，本节，我将从安全性的角度来给实战项目添加上安全的外衣，增加连接和数据的安全性。

好了，让我们开始今天的学习吧。

何为安全性？

安全，是一个非常广泛地词，从广义上来讲，对于我们的应用，分为内部安全和外部安全。

内部安全，即系统本身是不是安全，有没有 OOM 的风险，数据是否加密存储，等等。

外部安全，即外部系统访问该系统是不是安全，连接是否可靠，是否是非法连接，传输数据是否安全，等等。

本节，我们讨论的主题主要是外部安全，对于外部安全，我把它分成三个部分：连接可靠性、连接合法性、数据安全性，我们一起来学习 Netty 是怎么解决这些问题的。

连接可靠性

身为互联网人，你肯定听过一个词 ——keepalive，在不同的场合，它可能还会被称为心跳监测、空闲检测等，它的主要目的是为了告诉服务端，我还活着，不要关闭我的连接。

常用的处理方法是客户端定时的给服务端发送一个心跳包，服务端隔一定时间没收到这个客户端的心跳包，则认为它死了，可以把它关闭了。

但是，这样有个缺陷，如果客户端本身一直在和服务端交互，这种定时发送的方式就有点浪费带宽和流量，所以，比较好的方法是，客户端检测一定时间没跟服务端交互了才发送一个心跳包，服务端隔一定时间没收到客户端的请求了才认为它可以关闭了。

那么，在 Netty 中，我们该如何实现呢？

其实，Netty 天然就支持空闲检测，不过还缺少心跳的部分，正好上一节介绍的 WebSocket 是支持心跳协议的，两者结合起来就达到了“空闲检测 + 心跳”的目的。

> 在 WebSocket 中，是通过 Ping/Pong 这一对消息来表示心跳的，它们分别是 PingWebSocketFrame 和 PongWebSocketFrame。

好了，让我们先看服务端的实现：

```
@Slf4j
public class ServerIdleCheckHandler extends IdleStateHandler {
    // 重写构造方法
    public ServerIdleCheckHandler() {
        // 10秒钟没收到读事件就认为是空闲了
        super(10, 0, 0);
    }

    @Override
    protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) throws Exception {
        if (evt == IdleStateEvent.FIRST_READER_IDLE_STATE_EVENT) {
            log.error("idle check close the client");
            // 检测到读空闲则关闭连接
            ctx.close();
            return;
        }
        super.channelIdle(ctx, evt);
    }
}
```

是不是非常简单？！只需要继承 IdleStateHandler 即可，重写构造方法指定空闲时间，并在 channelIdle () 方法中处理空闲事件。

然后，别忘了将其添加到 Pipeline 中，可以把它放在前面：

```
ChannelPipeline p = ch.pipeline();
// 打印日志
p.addLast(new LoggingHandler(LogLevel.INFO));

// 空闲检测 ********
p.addLast(new ServerIdleCheckHandler());

// 添加Http协议编解码器、处理器
p.addLast(new HttpServerCodec());
p.addLast(new HttpObjectAggregator(65536));
// 添加WebSocket处理器
p.addLast(new WebSocketServerCompressionHandler());
p.addLast(new WebSocketServerProtocolHandler(WEBSOCKET_PATH, null, true));
// websocket编解码器
p.addLast(new BinaryWebSocketFrameDecoder());
p.addLast(new BinaryWebSocketFrameEncoder());

// 二次编解码器
p.addLast(new MahjongProtocolDecoder());
p.addLast(new MahjongProtocolEncoder());
// 处理器
p.addLast(new MahjongServerHandler());
```

好了，再来客户端的实现：

```
public class ClientIdleCheckHandler extends IdleStateHandler {
    public ClientIdleCheckHandler() {
        super(0, 5, 0);
    }
}
```

客户端除了要实现空闲检测外，还需要发送 Ping 请求，而 Ping 请求是 WebSocket 的东西，它需要经过 WebSocket 的进一步编解码，所以还需要写一个处理空闲的 Handler，要放在 WebSocket 处理器的后面：

```
@Slf4j
public class PingPongHandler extends SimpleChannelInboundHandler<PongWebSocketFrame> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, PongWebSocketFrame msg) throws Exception {
        // 返回pong响应
        log.info("client received pong response");
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
        // 检测到是写空闲事件
        if (evt == IdleStateEvent.FIRST_WRITER_IDLE_STATE_EVENT) {
            log.info("idle check, send ping request");

            // 发送ping请求
            ByteBuf buffer = ctx.alloc().buffer();
            buffer.writeBytes(new byte[] {6, 6, 6});
            PingWebSocketFrame frame = new PingWebSocketFrame(buffer);
            ctx.writeAndFlush(frame);
        }
        super.userEventTriggered(ctx, evt);
    }
}
```

在这个处理器中，检测到空闲了就发一个 Ping 请求给服务端，并处理 Pong 响应。

然后，把这两个 Handler 分别添加到 Pipeline 中：

```
ChannelPipeline p = ch.pipeline();

// 打印日志
p.addLast(new LoggingHandler(LogLevel.INFO));

// 空闲检测 ********
p.addLast(new ClientIdleCheckHandler());

p.addLast(new HttpClientCodec());
p.addLast(new HttpObjectAggregator(8192));
p.addLast(WebSocketClientCompressionHandler.INSTANCE);
p.addLast(handler);

// 心跳 ********
p.addLast(new PingPongHandler());

// websocket编解码器
p.addLast(new BinaryWebSocketFrameDecoder());
p.addLast(new BinaryWebSocketFrameEncoder());

// 二次编解码器
p.addLast(new MahjongProtocolDecoder());
p.addLast(new MahjongProtocolEncoder());
// 处理器
p.addLast(new MahjongClientHandler());
```

细心的同学可能会发现，怎么没看到服务端处理 Ping 请求呢？

其实，Ping 请求的处理是在 WebSocketServerProtocolHandler 中，它接收到是 Ping 请求，直接返回了一个 Pong 响应：

```
protected void decode(ChannelHandlerContext ctx, WebSocketFrame frame, List<Object> out) throws Exception {
    if (frame instanceof PingWebSocketFrame) {
        frame.content().retain();
        ctx.channel().writeAndFlush(new PongWebSocketFrame(frame.content()));
        readIfNeeded(ctx);
        return;
    }
    if (frame instanceof PongWebSocketFrame && dropPongFrames) {
        readIfNeeded(ctx);
        return;
    }

    out.add(frame.retain());
}
```

> 细心的同学会看到，它这里也使用了 retain () 方法，跟我们上一节用的 `ReferenceCountUtil.retain(msg);` 是一样的。

到这里，空闲检测 + 心跳 就完成了，分别启动服务端和客户端，查看日志：

```
19:28:09 [nioEventLoopGroup-2-1] PingPongHandler: idle check, send ping request
19:28:09 [nioEventLoopGroup-2-1] PingPongHandler: client received pong response
```

一切正常，不过，试着在牌局中停顿一会，会发现，连接被无情地断开了，这是因为客户端有一个等待用户输入的操作，这个操作是阻塞的，导致客户端的心跳没有发送出去，要解决这个问题其实也很简单，在客户端添加一个业务线程池，专门处理业务逻辑：

```
// 添加业务线程池
p.addLast(new DefaultEventLoopGroup(), new MahjongClientHandler());
```

再次运行程序，一切都正常了，我就不演示了。

好了，到这里，空闲检测 + 心跳就介绍完毕了，你 Get 到了吗？

连接合法性

其实，上面介绍的空闲检测 + 心跳的方式，在一定程度上，也包含了连接合法性的要求，你可能已经发现了，PingWebSocketFrame 是可以设置一个参数的，这个参数相当于是密钥，我们可以利用这个参数提高一定的安全性，如果是非法的客户端，它可能都不一定知道要心跳，即使知道要心跳，也不一定知道密钥是多少。

那么，我们这里说的连接合法性是什么呢？

一般是指黑白名单。

所谓黑名单，是被服务端禁止访问的用户，反之，白名单表示只有在白名单中的用户才可以访问服务端。

最简单的黑白名单实现方式，就是通过 IP 地址进行判断。

Netty 支持黑白名单吗？

天然支持。

在 Netty 中，定义了两种 IP 策略，一种是 UniqueIpFilter，一种是 RuleBasedIpFilter，前者表示一个 IP 地址只能建立一条连接，后者表示基于 IP 的过滤器，你可以设置一些规则，常用的规则是根据子网判断。

我以 RuleBasedIpFilter 为例添加一个黑名单过滤器：

```
ChannelPipeline p = ch.pipeline();
// 打印日志
p.addLast(new LoggingHandler(LogLevel.INFO));

// 黑名单过滤器 ******
IpFilterRule ipFilterRule = new IpSubnetFilterRule("192.168.175.1", 8, IpFilterRuleType.REJECT);
p.addLast(new RuleBasedIpFilter(ipFilterRule));

// 空闲检测
p.addLast(new ServerIdleCheckHandler());

// ...省略其他
```

这里的 IpSubnetFilterRule 使用的是 CIDR 表示法，即经常看到的 `130.39.37.100/24` 表示法，有兴趣的同学可以去了解下，当然，你也可以实现 IpFilterRule 接口定义自己的规则。

此时，重启服务端，启动客户端会被无情的断开连接：

```
21:02:45 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x11a6fb5a, L:/192.168.175.1:54759 - R:0.0.0.0/0.0.0.0:8080] CLOSE
21:02:45 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x11a6fb5a, L:/192.168.175.1:54759 ! R:0.0.0.0/0.0.0.0:8080] USER_EVENT: io.netty.channel.socket.ChannelInputShutdownReadComplete@41034074
21:02:45 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x11a6fb5a, L:/192.168.175.1:54759 ! R:0.0.0.0/0.0.0.0:8080] INACTIVE
21:02:45 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x11a6fb5a, L:/192.168.175.1:54759 ! R:0.0.0.0/0.0.0.0:8080] UNREGISTERED
```

好了，黑白名单我们就介绍到这里，是不是超级简单，想不想了解源码怎么实现的？自己看去～～

数据安全性

到目前为止，前后端传输的数据还是明文的，怎么做到加密传输呢？

> 虽然已经使用 Protobuf 序列化成字节数组了，但是仍然视为明文，别人只要拿到这个数据和结构体很容易就分析出来的。

这就不得不提鼎鼎大名的 SSL/TLS 了，有兴趣的同学可以去了解下它加密传输的过程，非常有意思。

那么，Netty 支持 SSL 吗？

天然支持，一个 Handler 的事，没有什么是 Handler 不能解决的问题。

好，我们来看 Netty 如何支持 SSL，先看服务端：

```
public class MahjongServer {

    private static final String WEBSOCKET_PATH = "/websocket";
    static final int PORT = Integer.parseInt(System.getProperty("port", "8443"));

    public static void main(String[] args) throws Exception {
        // ssl，使用自己生成的证书
        SelfSignedCertificate ssc = new SelfSignedCertificate();
        SslContext sslCtx = SslContextBuilder.forServer(ssc.certificate(), ssc.privateKey())
                .build();
        // 查看证书生成的位置
        System.out.println(ssc.certificate().getPath());

        // ...省略其他代码

        serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
                // ...省略其他代码

                // 空闲检测
                p.addLast(new ServerIdleCheckHandler());

                // ssl，创建一个handler
                p.addLast(sslCtx.newHandler(ch.alloc()));
            }
        }
    }
}
```

使用自己生成的证书，并通过 SslContext 生成一个 Handler 添加到 Pipeline 中即可，就是这么简单。

我们再来看看客户端的实现：

```
public class MahjongClient {

    // 记得修改协议为wss
    static final String URL = System.getProperty("url", "wss://127.0.0.1:8443/websocket");

    public static void main(String[] args) throws Exception {
        // 创建sslContext
        SslContext sslCtx = SslContextBuilder.forClient().build();

        // 工作线程池
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(workerGroup);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    // ...省略其他代码

                    // 空闲检测
                    p.addLast(new ClientIdleCheckHandler());

                    // ssl，创建handler
                    p.addLast(sslCtx.newHandler(ch.alloc()));
                }
            }
    }
}
```

客户端的实现基本上类似，是不是很简单？让我们运行起来看看，报错了，提示找不到证书，这是怎么回事呢？

那是因为我们的证书是自己生成的，不是从权威机构花钱买的，所以，不受信任，此时，需要把导入证书，客户端才能正常访问服务器，导入命令如下：

```
// 导入证书
keytool.exe -import -alias netty-core -keystore "D:\Program Files\Java\jdk1.8.0_202\jre\lib\security\cacerts" -file "C:\Users\alan\AppData\Local\Temp\keyutil_example.com_8575751454001666416.crt" -storepass changeit

// 移除证书
keytool.exe -delete -alias netty-core -keystore "D:\Program Files\Java\jdk1.8.0_202\jre\lib\security\cacerts" -storepass changeit
```

注意，保存的位置是你的 IDEA 使用的 JRE 的位置下面的 `lib\security\cacerts`，我的 IDEA 一般配置的是 JDK 的目录，那就用 JDK 目录下面的 JRE 下面对应的路径，别弄错了。证书的位置在服务端启动的时候会打印出来路径。

OK，再次重启客户端，一切正常了，我就不演示了。

到此，我们也通过 SSL 的方式实现了数据安全了。

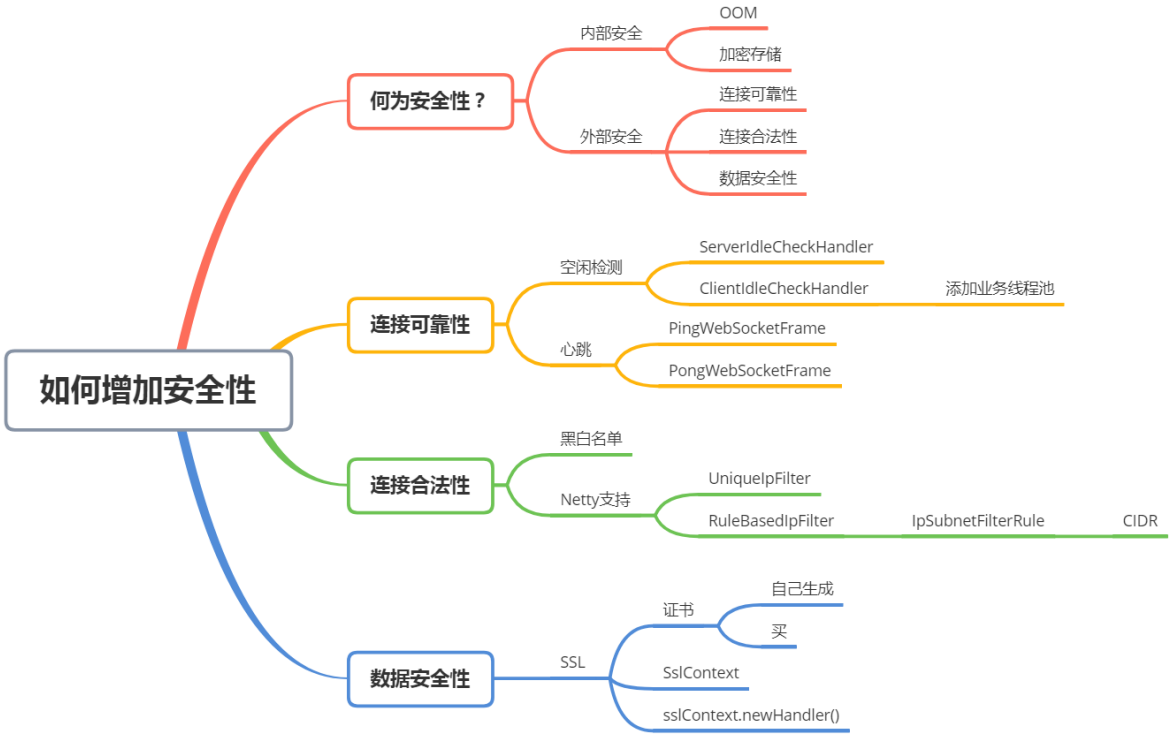
后记

本节，我们从空闲检测（心跳）、黑白名单、SSL 三个方面分别对我们的实战项目做了安全增强，在 Netty 中，使用这些技术都变得非常简单，没有什么是一个 Handler 不能搞定的事，如果真的有，那就用两个，可见，Netty 的扩展性真的是无话可说。

到这里，Netty 还可以再调优吗？

下一节，我们将从参数的角度来对实战项目进行调优，敬请期待。

思维导图



}

35 如何支持WebSocket

37 如何调优参数