



涵盖高并发开发、
大厂面试的核心难题

Java

卷1

高并发核心编程

NIO、Netty、Redis、ZooKeeper



从设计模式和基础知识入手，抽丝剥茧，
将高深莫测的Java高并发知识讲解得浅显易懂

尼恩 编著



机械工业出版社
China Machine Press

更多IT书籍请关注 t.cn/blogs.cn

Java高并发核心编程. 卷1, NIO、Netty、Redis、ZooKeeper

尼恩 编著

ISBN: 978-7-111-67758-1

本书纸版由机械工业出版社于2021年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目 录

[作者简介](#)

[前言](#)

[自序](#)

[第1章 高并发时代的必备技能](#)

[1.1 Netty为何这么火](#)

[1.1.1 Netty火热的程度](#)

[1.1.2 Netty是面试的必杀器](#)

[1.2 高并发利器Redis](#)

[1.2.1 什么是Redis](#)

[1.2.2 Redis成为缓存事实标准的原因](#)

[1.3 分布式利器ZooKeeper](#)

[1.3.1 什么是ZooKeeper](#)

[1.3.2 ZooKeeper的优势](#)

[1.4 高性能HTTP通信技术](#)

[1.4.1 十万级以上高并发场景中的高并发HTTP通信技术](#)

[1.4.2 微服务之间的高并发RPC技术](#)

[1.5 高并发IM的综合实战](#)

[1.5.1 高并发IM的学习价值](#)

[1.5.2 庞大的应用场景](#)

[第2章 高并发IO的底层原理](#)

[2.1 IO读写的基本原理](#)

[2.1.1 内核缓冲区与进程缓冲区](#)

[2.1.2 典型的系统调用流程](#)

[2.2 四种主要的IO模型](#)

[2.2.1 同步阻塞IO](#)

[2.2.2 同步非阻塞IO](#)

[2.2.3 IO多路复用](#)

[2.2.4 异步IO](#)

[2.3 通过合理配置来支持百万级并发连接](#)

[第3章 Java NIO核心详解](#)

[3.1 Java NIO简介](#)

[3.1.1 NIO和OIO的对比](#)

[3.1.2 通道](#)

[3.1.3 选择器](#)

[3.1.4 缓冲区](#)

[3.2 详解NIO Buffer类及其属性](#)

[3.2.1 Buffer类](#)

[3.2.2 Buffer类的重要属性](#)

[3.3 详解NIO Buffer类的重要方法](#)

[3.3.1 allocate\(\)](#)

[3.3.2 put\(\)](#)

[3.3.3 flip\(\)](#)

[3.3.4 get\(\)](#)

[3.3.5 rewind\(\)](#)

[3.3.6 mark\(\)和reset\(\)](#)

[3.3.7 clear\(\)](#)

[3.3.8 使用Buffer类的基本步骤](#)

[3.4 详解NIO Channel类](#)

[3.4.1 FileChannel](#)

[3.4.2 使用FileChannel完成文件复制的实战案例](#)

[3.4.3 SocketChannel](#)

[3.4.4 使用SocketChannel发送文件的实战案例](#)

[3.4.5 DatagramChannel](#)

[3.4.6 使用DatagramChannel发送数据的实战案例](#)

[3.5 详解NIO Selector](#)

[3.5.1 选择器与注册](#)

[3.5.2 SelectableChannel](#)

[3.5.3 SelectionKey](#)

[3.5.4 选择器使用流程](#)

[3.5.5 使用NIO实现Discard服务器的实战案例](#)

[3.5.6 使用SocketChannel在服务端接收文件的实战案例](#)

[第4章 鼎鼎大名的Reactor模式](#)

[4.1 Reactor模式的重要性](#)

[4.1.1 为什么首先学习Reactor模式](#)

[4.1.2 Reactor模式简介](#)

[4.1.3 多线程IO的致命缺陷](#)

[4.2 单线程Reactor模式](#)

[4.2.1 什么是单线程Reactor](#)

[4.2.2 单线程Reactor的参考代码](#)

[4.2.3 单线程Reactor模式的EchoServer的实战案例](#)

[4.2.4 单线程Reactor模式的缺点](#)

[4.3 多线程Reactor模式](#)

[4.3.1 多线程版本的Reactor模式演进](#)

[4.3.2 多线程版本Reactor的实战案例](#)

[4.3.3 多线程版本Handler的实战案例](#)

[4.4 Reactor模式的优缺点](#)

[第5章 Netty核心原理与基础实战](#)

[5.1 第一个Netty实战案例DiscardServer](#)

- 5.1.1 创建第一个Netty项目
- 5.1.2 第一个Netty服务端程序
- 5.1.3 业务处理器NettyDiscardHandler
- 5.1.4 运行NettyDiscardServer
- 5.2 解密Netty中的Reactor模式
 - 5.2.1 回顾Reactor模式中IO事件的处理流程
 - 5.2.2 Netty中的Channel
 - 5.2.3 Netty中的Reactor
 - 5.2.4 Netty中的Handler
 - 5.2.5 Netty中的Pipeline
- 5.3 详解Bootstrap
 - 5.3.1 父子通道
 - 5.3.2 EventLoopGroup
 - 5.3.3 Bootstrap启动流程
 - 5.3.4 ChannelOption
- 5.4 详解Channel
 - 5.4.1 Channel的主要成员和方法
 - 5.4.2 EmbeddedChannel
- 5.5 详解Handler
 - 5.5.1 ChannelInboundHandler入站处理器
 - 5.5.2 ChannelOutboundHandler出站处理器
 - 5.5.3 ChannelInitializer通道初始化处理器
 - 5.5.4 ChannelInboundHandler的生命周期的实战案例
- 5.6 详解Pipeline
 - 5.6.1 Pipeline入站处理流程
 - 5.6.2 Pipeline出站处理流程
 - 5.6.3 ChannelHandlerContext

- [5.6.4 HeadContext与TailContext](#)
- [5.6.5 Pipeline入站和出站的双向链接操作](#)
- [5.6.6 截断流水线的入站处理传播过程](#)
- [5.6.7 在流水线上热插拔Handler](#)
- [5.7 详解ByteBuf](#)
 - [5.7.1 ByteBuf的优势](#)
 - [5.7.2 ByteBuf的组成部分](#)
 - [5.7.3 ByteBuf的重要属性](#)
 - [5.7.4 ByteBuf的方法](#)
 - [5.7.5 ByteBuf基本使用的实战案例](#)
 - [5.7.6 ByteBuf的引用计数](#)
 - [5.7.7 ByteBuf的分配器](#)
 - [5.7.8 ByteBuf缓冲区的类型](#)
 - [5.7.9 两类ByteBuf使用的实战案例](#)
 - [5.7.10 ByteBuf的自动创建与自动释放](#)
 - [5.7.11 ByteBuf浅层复制的高级使用方式](#)
- [5.8 Netty的零拷贝](#)
 - [5.8.1 通过CompositeByteBuf实现零拷贝](#)
 - [5.8.2 通过wrap操作实现零拷贝](#)
- [5.9 EchoServer的实战案例](#)
 - [5.9.1 NettyEchoServer](#)
 - [5.9.2 NettyEchoServerHandler](#)
 - [5.9.3 NettyEchoClient](#)
 - [5.9.4 NettyEchoClientHandler](#)
- [第6章 Decoder与Encoder核心组件](#)
 - [6.1 Decoder原理与实战](#)
 - [6.1.1 ByteToMessageDecoder解码器处理流程](#)

- 6.1.2 自定义Byte2IntegerDecoder整数解码器
 - 6.1.3 ReplayingDecoder解码器
 - 6.1.4 整数的分包解码器的实战案例
 - 6.1.5 字符串的分包解码器的实战案例
 - 6.1.6 MessageToMessageDecoder解码器
 - 6.2 常用的内置Decoder
 - 6.2.1 LineBasedFrameDecoder解码器
 - 6.2.2 DelimiterBasedFrameDecoder解码器
 - 6.2.3 LengthFieldBasedFrameDecoder解码器
 - 6.2.4 多字段Head-Content协议数据包解析的实战案例
 - 6.3 Encoder原理与实战
 - 6.3.1 MessageToByteEncoder编码器
 - 6.3.2 MessageToMessageEncoder编码器
 - 6.4 解码器和编码器的结合
 - 6.4.1 ByteToMessageCodec编解码器
 - 6.4.2 CombinedChannelDuplexHandler组合器
- 第7章 序列化与反序列化：JSON和Protobuf
- 7.1 详解粘包和拆包
 - 7.1.1 半包问题的实战案例
 - 7.1.2 什么是半包问题
 - 7.1.3 半包问题的根因分析
 - 7.2 使用JSON协议通信
 - 7.2.1 JSON的核心优势
 - 7.2.2 JSON序列化与反序列化开源库
 - 7.2.3 JSON序列化与反序列化的实战案例
 - 7.2.4 JSON传输的编码器和解码器
 - 7.2.5 JSON传输的服务端的实战案例

- 7.2.6 JSON传输的客户端的实战案例
- 7.3 使用Protobuf协议通信
 - 7.3.1 一个简单的proto文件的实战案例
 - 7.3.2 通过控制台命令生成POJO和Builder
 - 7.3.3 通过Maven插件生成POJO和Builder
 - 7.3.4 Protobuf序列化与反序列化的实战案例
- 7.4 Protobuf编解码的实战案例
 - 7.4.1 Netty内置的Protobuf基础编码器/解码器
 - 7.4.2 Protobuf传输的服务端的实战案例
 - 7.4.3 Protobuf传输的客户端的实战案例
- 7.5 详解Protobuf协议语法
 - 7.5.1 proto文件的头部声明
 - 7.5.2 Protobuf的消息结构体与消息字段
 - 7.5.3 Protobuf字段的数据类型
 - 7.5.4 proto文件的其他语法规规范
- 第8章 基于Netty单体IM系统的开发实战
 - 8.1 自定义Protobuf编解码器
 - 8.1.1 自定义Protobuf编码器
 - 8.1.2 自定义Protobuf解码器
 - 8.1.3 IM系统中Protobuf消息格式的设计
 - 8.2 IM的登录流程
 - 8.2.1 图解登录/响应流程的环节
 - 8.2.2 客户端涉及的主要模块
 - 8.2.3 服务端涉及的主要模块
 - 8.3 客户端的登录处理的实战案例
 - 8.3.1 LoginConsoleCommand和User POJO
 - 8.3.2 LoginSender

- 8.3.3 ClientSession
- 8.3.4 LoginResponseHandler
- 8.3.5 客户端流水线的装配
- 8.4 服务端的登录响应的实战案例
 - 8.4.1 服务端流水线的装配
 - 8.4.2 LoginRequestHandler
 - 8.4.3 LoginProcesser
 - 8.4.4 EventLoop线程和业务线程相互隔离
- 8.5 详解Session服务器会话
 - 8.5.1 通道的容器属性
 - 8.5.2 ServerSession服务端会话类
 - 8.5.3 SessionMap会话管理器
- 8.6 点对点单聊的实战案例
 - 8.6.1 单聊的端到端流程
 - 8.6.2 客户端的ChatConsoleCommand收集聊天内容
 - 8.6.3 客户端的CommandController发送POJO
 - 8.6.4 服务端的ChatRedirectHandler进行消息转发
 - 8.6.5 服务端的ChatRedirectProcesser进行异步消息转发
 - 8.6.6 客户端的ChatMsgHandler聊天消息处理器
- 8.7 详解心跳检测
 - 8.7.1 网络连接的假死现象
 - 8.7.2 服务端的空闲检测
 - 8.7.3 客户端的心跳发送

第9章 HTTP原理与Web服务器实战

- 9.1 高性能Web应用架构
 - 9.1.1 十万级并发的Web应用架构
 - 9.1.2 千万级高并发的Web应用架构

9.2 详解HTTP应用层协议

9.2.1 HTTP简介

9.2.2 HTTP的请求URL

9.2.3 HTTP的请求报文

9.2.4 HTTP的响应报文

9.2.5 HTTP中GET和POST的区别

9.3 HTTP的演进

9.3.1 HTTP的1.0版本

9.3.2 HTTP的1.1版本

9.3.3 HTTP的2.0版本

9.4 基于Netty实现简单的Web服务器

9.4.1 基于Netty的HTTP服务器演示实例

9.4.2 基于Netty的HTTP请求的处理流程

9.4.3 Netty内置的HTTP报文解码流程

9.4.4 基于Netty的HTTP响应编码流程

9.4.5 HttpEchoHandler回显业务处理器的实战案例

9.4.6 使用Postman发送多种类型的请求体

第10章 高并发HTTP通信的核心原理

10.1 需要进行HTTP连接复用的高并发场景

10.1.1 反向代理Nginx与Java Web应用服务之间的HTTP高并发通信

10.1.2 微服务网关与微服务Provider实例之间的HTTP高并发通信

10.1.3 分布式微服务Provider实例之间的RPC的HTTP高并发通信

10.1.4 Java通过HTTP客户端访问REST接口服务的HTTP高并发通信

10.2 详解传输层TCP

10.2.1 TCP/IP的分层模型

10.2.2 HTTP报文传输原理

10.2.3 TCP的报文格式

- 10. 2. 4 TCP的三次握手
 - 10. 2. 5 TCP的四次挥手
 - 10. 2. 6 三次握手、四次挥手的常见面试题
 - 10. 3 TCP连接状态的原理与实验
 - 10. 3. 1 TCP/IP连接的11种状态
 - 10. 3. 2 通过netstat指令查看连接状态
 - 10. 4 HTTP长连接原理
 - 10. 4. 1 HTTP长连接和短连接
 - 10. 4. 2 不同HTTP版本中的长连接选项
 - 10. 5 服务端HTTP长连接技术
 - 10. 5. 1 应用服务器Tomcat的长连接配置
 - 10. 5. 2 Nginx承担服务端角色时的长连接配置
 - 10. 5. 3 服务端长连接设置的注意事项
 - 10. 6 客户端HTTP长连接技术原理与实验
 - 10. 6. 1 HttpURLConnection短连接技术
 - 10. 6. 2 HTTP短连接的通信实验
 - 10. 6. 3 Apache HttpClient客户端的HTTP长连接技术
 - 10. 6. 4 Apache HttpClient客户端长连接实验
 - 10. 6. 5 Nginx承担客户端角色时的长连接技术
- 第11章 WebSocket原理与实战
- 11. 1 WebSocket协议简介
 - 11. 1. 1 Ajax短轮询和Long Poll长轮询的原理
 - 11. 1. 2 WebSocket与HTTP之间的关系
 - 11. 2 WebSocket回显演示程序开发
 - 11. 2. 1 WebSocket回显程序的客户端代码
 - 11. 2. 2 WebSocket相关的Netty内置处理类
 - 11. 2. 3 WebSocket的回显服务器

- [11. 2. 4 WebSocket的业务处理器](#)
- [11. 3 WebSocket协议通信的原理](#)
- [11. 3. 1 抓取WebSocket协议的本机数据包](#)
- [11. 3. 2 WebSocket握手过程](#)
- [11. 3. 3 WebSocket通信报文格式](#)
- [第12章 SSL/TLS核心原理与实战](#)
- [12. 1 什么是SSL/TLS](#)
- [12. 1. 1 SSL/TLS协议的版本演进](#)
- [12. 1. 2 SSL/TLS协议的分层结构](#)
- [12. 2 加密算法原理与实战](#)
- [12. 2. 1 哈希单向加密算法原理与实战](#)
- [12. 2. 2 对称加密算法原理与实战](#)
- [12. 2. 3 非对称加密算法原理与实战](#)
- [12. 2. 4 数字签名原理与实战](#)
- [12. 3 SSL/TLS运行过程](#)
- [12. 3. 1 SSL/TLS第一阶段握手](#)
- [12. 3. 2 SSL/TLS第二阶段握手](#)
- [12. 3. 3 SSL/TLS第三阶段握手](#)
- [12. 3. 4 SSL/TLS第四阶段握手](#)
- [12. 4 详解Keytool工具](#)
- [12. 4. 1 数字证书与身份识别](#)
- [12. 4. 2 存储密钥与证书文件格式](#)
- [12. 4. 3 使用Keytool工具管理密钥和证书](#)
- [12. 5 使用Java程序管理密钥与证书](#)
- [12. 5. 1 使用Java操作数据证书所涉及的核心类](#)
- [12. 5. 2 使用Java程序创建密钥与仓库](#)
- [12. 5. 3 使用Java程序导出证书文件](#)

[12.5.4 使用Java程序将数字证书导入信任仓库](#)

[12.6 OI0通信中的SSL/TLS使用实战](#)

[12.6.1 JSSE安全套接字扩展核心类](#)

[12.6.2 JSSE安全套接字的创建过程](#)

[12.6.3 OI0安全通信的Echo服务端实战](#)

[12.6.4 OI0安全通信的Echo客户端实战](#)

[12.7 单向认证与双向认证](#)

[12.7.1 SSL/TLS单向认证](#)

[12.7.2 使用证书信任管理器](#)

[12.7.3 SSL/TLS双向认证](#)

[12.8 Netty通信中的SSL/TLS使用实战](#)

[12.8.1 Netty安全通信演示实例](#)

[12.8.2 Netty内置SSLEngine处理器详解](#)

[12.8.3 Netty的简单安全聊天器服务端程序](#)

[12.9 HTTPS协议安全通信实战](#)

[12.9.1 使用Netty实现HTTPS回显服务端程序](#)

[12.9.2 通过HttpsURLConnection发送HTTPS请求](#)

[12.9.3 HTTPS服务端与客户端的测试用例](#)

[第13章 ZooKeeper分布式协调](#)

[13.1 ZooKeeper伪集群安装和配置](#)

[13.1.1 创建数据目录和日志目录](#)

[13.1.2 创建myid文本文件](#)

[13.1.3 创建和修改配置文件](#)

[13.1.4 配置文件示例](#)

[13.1.5 启动ZooKeeper伪集群](#)

[13.2 使用ZooKeeper进行分布式存储](#)

[13.2.1 详解ZooKeeper存储模型](#)

- [13. 2. 2 zkCli客户端指令清单](#)
- [13. 3 ZooKeeper应用开发实战](#)
- [13. 3. 1 ZkClient开源客户端](#)
- [13. 3. 2 Curator开源客户端](#)
- [13. 3. 3 准备Curator开发环境](#)
- [13. 3. 4 创建Curator客户端实例](#)
- [13. 3. 5 通过Curator创建节点](#)
- [13. 3. 6 通过Curator读取节点](#)
- [13. 3. 7 通过Curator更新节点](#)
- [13. 3. 8 通过Curator删除节点](#)
- [13. 4 分布式命名服务实战](#)
- [13. 4. 1 ID生成器](#)
- [13. 4. 2 ZooKeeper分布式ID生成器的实战案例](#)
- [13. 4. 3 集群节点的命名服务的实战案例](#)
- [13. 4. 4 结合ZooKeeper实现SnowFlake ID算法](#)
- [13. 5 分布式事件监听的重点](#)
- [13. 5. 1 Watcher标准的事件处理器](#)
- [13. 5. 2 NodeCache节点缓存的监听](#)
- [13. 5. 3 PathCache子节点监听](#)
- [13. 5. 4 TreeCache节点树缓存](#)
- [13. 6 分布式锁原理与实战](#)
- [13. 6. 1 公平锁和可重入锁的原理](#)
- [13. 6. 2 ZooKeeper分布式锁的原理](#)
- [13. 6. 3 分布式锁的基本流程](#)
- [13. 6. 4 加锁的实现](#)
- [13. 6. 5 释放锁的实现](#)
- [13. 6. 6 分布式锁的使用](#)

[13. 6. 7 Curator的InterProcessMutex可重入锁](#)

[13. 6. 8 ZooKeeper分布式锁的优缺点](#)

[第14章 分布式缓存Redis实战](#)

[14. 1 Redis入门](#)

[14. 1. 1 Redis的安装和配置](#)

[14. 1. 2 Redis客户端命令](#)

[14. 1. 3 Redis键的命名规范](#)

[14. 2 Redis数据类型](#)

[14. 2. 1 String](#)

[14. 2. 2 List](#)

[14. 2. 3 Hash](#)

[14. 2. 4 Set](#)

[14. 2. 5 ZSet](#)

[14. 3 Jedis基础编程的实战案例](#)

[14. 3. 1 Jedis操作String](#)

[14. 3. 2 Jedis操作List](#)

[14. 3. 3 Jedis操作Hash](#)

[14. 3. 4 Jedis操作Set](#)

[14. 3. 5 Jedis操作ZSet](#)

[14. 4 JedisPool连接池的实战案例](#)

[14. 4. 1 JedisPool的配置](#)

[14. 4. 2 JedisPool的创建和预热](#)

[14. 4. 3 JedisPool的使用](#)

[14. 5 使用spring-data-redis完成CRUD的实战案例](#)

[14. 5. 1 CRUD中应用缓存的场景](#)

[14. 5. 2 配置spring-redis.xml](#)

[14. 5. 3 RedisTemplate模板API](#)

[14.5.4 使用RedisTemplate模板API完成CRUD的实战案例](#)

[14.5.5 使用RedisCallback回调完成CRUD的实战案例](#)

[14.6 Spring的Redis缓存注解](#)

[14.6.1 使用Spring缓存注解完成CRUD的实战案例](#)

[14.6.2 spring-redis.xml中配置的调整](#)

[14.6.3 @CachePut和@Cacheable注解](#)

[14.6.4 @CacheEvict注解](#)

[14.6.5 @Caching组合注解](#)

[14.7 详解SpEL](#)

[14.7.1 SpEL运算符](#)

[14.7.2 缓存注解中的SpEL表达式](#)

[第15章 亿级高并发IM架构与实战](#)

[15.1 支撑亿级流量的高并发IM架构的理论基础](#)

[15.1.1 亿级流量的系统架构的开发实战](#)

[15.1.2 高并发架构的技术选型](#)

[15.1.3 详解IM消息的序列化协议选型](#)

[15.1.4 详解长连接和短连接](#)

[15.2 分布式IM的命名服务的实战案例](#)

[15.2.1 IM节点的POJO类](#)

[15.2.2 IM节点的ImWorker类](#)

[15.3 Worker集群的负载均衡的实战案例](#)

[15.3.1 ImLoadBalance负载均衡器](#)

[15.3.2 与WebGate的整合](#)

[15.4 即时通信消息的路由和转发的实战案例](#)

[15.4.1 IM路由器WorkerRouter](#)

[15.4.2 IM转发器PeerSender](#)

[15.5 在线用户统计的实战案例](#)

[15.5.1 Curator的分布式计数器](#)

[15.5.2 用户上线和下线的统计](#)

本书从操作系统底层的I/O原理入手讲解Java高并发核心编程知识，同时提供高性能开发的实战案例，是一本Java高并发编程的基础原理和实战图书。

本书共分为15章。第1~4章为高并发基础，浅显易懂地剖析高并发I/O的底层原理，图文并茂地介绍Java异步回调模式，细致地讲解Reactor高性能模式。这些原理方面的基础知识非常重要，会为读者打下坚实的基础，也是日常开发Java后台应用时解决实际问题的金钥匙。第5~8章为Netty原理和实战，是本书的重中之重，主要介绍高性能通信框架Netty、Netty的重要组件、单体IM的实战设计和模块实现。第9~12章从TCP、HTTP入手，介绍客户端与服务端、服务端与服务端之间的高性能HTTP通信和WebSocket通信。第13~15章对ZooKeeper、Curator API、Redis、Jedis API的使用进行详尽的说明，以提升读者设计和开发高并发、可扩展系统的能力。

本书兼具基础知识和实战案例，既可作为对Java NIO、高性能I/O、高并发编程感兴趣的大专院校学生以及初/中级Java工程师的自学图书，也可作为在生产项目中需要用到Netty、Redis、ZooKeeper三大框架的架构师或项目人员的参考书。

作者简介

尼恩，中南大学硕士，架构师，先后在华为、神州数码从事技术研发工作，专注于高性能Web平台、高性能通信、高性能搜索、数据挖掘等领域的架构设计和分析工作。

前言

移动时代、5G时代、物联网时代的大幕已经开启，新时代提升了对Java应用的高性能、高并发的要求，也抬升了Java工程师的技术台阶和面试门槛。

很多公司的面试题从某个侧面反映了生产场景的技术要求。之前只有BAT等大公司才有高并发技术相关的面试题，现在与Java项目相关的整个行业基本都涉及此类面试题。Java NIO、Reactor模式、高性能通信框架Netty、分布式锁、分布式ID、分布式缓存、高并发架构、JUC、JMM、高并发设计模式、线程池、微服务框架（如Spring Cloud、Nginx反向代理）等高并发方面的面试题，从以前的加分题变成现在的基础题。本书着重介绍Java NIO、Reactor模式、高性能通信框架Netty、ZooKeeper分布式锁、分布式ID、Redis分布式缓存、分布式IM方面的内容，以帮助大家快速掌握Java高并发的底层通信知识和分布式架构知识。

本书内容

本书实际上是《Netty、Redis、ZooKeeper高并发实战》[\[1\]](#)一书的升级版，同时也是三卷本《Java高并发核心编程》的第1卷，旨在帮助大家掌握Netty、Redis、ZooKeeper、TCP、HTTP、分布式IM的原理，为大家打下Java高并发技术的知识基础。

第1~4章从操作系统的底层原理开始，浅显易懂地揭秘高并发IO的底层原理，并介绍如何让单体Java应用支持百万级的高并发；从传统的阻塞式IO开始，细致地解析Reactor高性能模式，介绍高性能网络开发的基础知识。这些非常底层的原理知识和基础知识非常重要，是开发过程中解决Java实际问题必不可少的。

第5~8章重点讲解Netty。目前Netty是高性能通信框架皇冠上当之无愧的明珠，是支撑其他众多著名的高并发、分布式、大数据框架底层的框架。这几章从Reactor模式入手，以“四两拨千斤”的方式为大家介绍Netty原理。同时，还将介绍如何通过Netty来解决网络编程中的重点难题，如Protobuf序列化问题、半包问题等。

第9~12章从TCP、HTTP入手，介绍客户端与服务端、服务端与服务端之间的高性能HTTP通信和WebSocket通信。这几章深入浅出地介绍TCP、HTTP、WebSocket三大常用的协议，以及如何基于Netty实现HTTP、WebSocket高性能通信。

第13章对ZooKeeper进行详细的介绍。除了全面地介绍Curator API之外，还从实战的角度出发介绍如何使用ZooKeeper设计分布式ID生成器，并对重要的SnowFlake算法进行详细的介绍。另外，还结合小故事以图文并茂的方式浅显易懂地介绍分布式锁的基本原理。

第14章从实战开发层面对Redis进行介绍，详细介绍Redis的5种数据类型、客户端操作指令、Jedis Java API。另外，还通过spring-data-redis来完成数据分布式缓存的实战案例，详尽地介绍Spring的缓存注解以及涉及的SpEL表达式语言。

第15章通过CrazyIM项目为大家介绍一个亿级流量的高并发IM系统模型，这个高并发架构的系统模型不仅仅限于IM系统，通过简单的调整和适配就可以应用于当前主流的Java后台系统。

读者对象

- 对Java NIO、高性能IO、高并发编程感兴趣的大专院校学生。
- 需要学习Java高并发技术和高并发架构的初、中级Java工程师。
- 生产项目中需要用到Netty、Redis、ZooKeeper三大框架的架构师或者项目人员。

本书源代码下载

本书的源代码可以从
https://gitee.com/crazymaker/netty_redis_zookeeper_source_code.git 下载。另外，还可以登录机械工业出版社华章公司网站 (www.hzbook.com) 下载，方法是：先搜索到本书，然后在页面上的“资料下载”模块下载。如果下载有问题，请发送电子邮件至 booksaga@126.com，邮件主题为“求Java高并发核心编程 卷1下载资源”。

勘误和支持

由于笔者水平和能力有限，书中不妥之处在所难免，希望读者提出宝贵意见和建议。本系列书的读者QQ群为104131248，目前群中已经包含了不少高质量的面试题以及开发技术难题，欢迎读者入群进行交流。

致谢

首先感谢卞诚君老师，没有他的指导和帮助，就不会有《Netty、Redis、ZooKeeper高并发实战》一书的面世，更不会有后续的本书。

然后感谢《Netty、Redis、ZooKeeper高并发实战》的读者，是他们对该书的高度评价，激励我对图书的内容不断进行升级和完善，并最终成为三卷本《Java高并发核心编程》的第1卷。

最后感谢“疯狂创客圈”社群中的开发小伙伴们，他们中有很多非常有前途的技术狂人，他们对Java高并发技术的高涨热情和孜孜以求让笔者惊叹不已。欢迎大家进入“疯狂创客圈”社群（QQ群为104131248）积极“砸”问题，虽然有的技术难题笔者不一定能给出最佳的解决方案，但坦诚、纯粹的技术交流，能让大家相互启发，产生技术灵感，拓展技术视野，并最终提升技术水平。

尼恩

2020年11月29日

[1] 该书已由机械工业出版社出版，书号为978-7-111-63290-0。

——编辑注

自序

身边常常有小伙伴问我怎样提高Java技术水平。下面给两个简单的例子：

- 小伙伴A（6年经验）说：尼恩，使用Java编程时，我在思路和速度上都赶不上小伙伴B（5年经验），尤其是在解决复杂问题的时候，我该怎么办？
- 小伙伴C（12年经验）说：尼恩，我司刚刚引进了一位高薪的Java核心架构师，他的薪酬挺令人心动的，如何才能提高我的Java技术水平，成为核心架构师呢？

遇到这类问题，我一概回答：“多读书。就目前看来，这是一条快捷、经济、有效地提高Java水平的途径。”

为什么这么说呢？首先，以我本人为例，身为核心架构师，我在技术能力方面早已得到团队认可，在团队内长期居于Bug排除榜前列，专门负责解决复杂、困难的技术问题。实际上，方法很简单，就是多阅读专业图书，我家里的技术书都可以用汗牛充栋来形容了。其次，给大家简单地分析一下具体原因。目前学习技术的途径大致有三种：

（1）阅读博文；（2）观看视频；（3）阅读书籍。通过途径1（阅读博文）获得的知识，往往过于碎片化，难成体系。这种途径更适用于了解技术趋势、解决问题时进行资料查阅。通过途径2（观看视频）获取知识时，需要耗费大量的时间，而且很多视频是填鸭式的知识灌输。所以，途径2更适用于初学者。对于有经验、能动性高的Java工程师来说，途径2的效率太低，需要大量的时间成本。通过途径3（阅读

书籍) 获取知识有一个显著的优势：书籍能以很小的体积承载巨量知识，而且所承载的是系统化、层次化的知识。

上述三种途径各有优劣，鉴于Java高并发所涉及的核心技术比较多，包括Spring Cloud、Nginx、JUC、JMM、Kafka、ElasticSearch等，我将结合博文、视频、书籍三种形式，为大家提供一个立体的、全方位的Java高并发核心编程知识仓库。在“疯狂创客圈”（我发起的Java高并发交流社群，QQ群为104131248）中，我将已出版的、在写的、规划中的图书整合成一个高并发核心编程的图书系列，大致清单如下：

(1) 《Netty、Redis、ZooKeeper高并发实战》：从操作系统底层的IO原理、Reactor高并发模式入手，介绍Java分布式、高并发通信原理，并指导大家进行高并发IM实战。

此书已于2019年8月出版，由于内容略微单薄，特进行内容的完善和升级，升级版进行了书名的变更，新书名为《Java高并发核心编程 卷1：NIO、Netty、Redis、ZooKeeper》。

(2) 《Spring Cloud、Nginx高并发核心编程》[\[1\]](#)：涵盖Spring Cloud、Nginx的核心原理和编程知识，并指导大家编写一个高并发的秒杀实战程序。此书已于2020年10月出版。

(3) 《Java高并发核心编程 卷1：NIO、Netty、Redis、ZooKeeper》：介绍Reactor模式、Netty、ZooKeeper、Redis、TCP、HTTP、WebSocket、NIO等Java高性能通信的核心原理和编程知识，并指导大家编写一个高并发的分布式IM实战程序——CrazyIM。

此卷即为本书，作为《Netty、Redis、ZooKeeper高并发实战》一书的升级版，对上一版本的内容进行了大量的优化和扩充。和上一版本相比，此卷知识量更大，所以学习价值也更高。

(4) 《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》：聚焦Java高并发基础知识，内容包括多线程、线程池、JMM内存模型、JUC并发包、AQS同步器、高并发容器类、高并发设计模式等。

(5) 《Java高并发核心编程 卷3》（最终书名待定）：覆盖Kafka、RocketMQ、ElasticSearch等重要的高并发中间件的核心原理和编程知识。

此卷仍在规划中，相关的内容后续可能还会调整。

Java高并发系列图书的初衷是为大家奉上一系列有关Java高并发方面的“原理级”“思想级”的图书，帮助大家轻松、切实、快捷地获取Java高并发核心知识，从而扎稳自己的知识底盘，提升自己的开发内功。

很多“疯狂创客圈”社群小伙伴已经通过阅读有了深切的体会：读书才是一条快捷、经济、有效地提高Java水平的途径。在这里，祝贺这些小伙伴！

尼恩

2020年10月23日

[1]该书已由机械工业出版社出版，书号为978-7-111-66557-1。

——编辑注

第1章 高并发时代的必备技能

随着5G应用、多终端应用、物联网应用、工业互联应用、大数据应用、人工智能应用的飞速发展，高并发开发时代已然到来，能够驾驭高并发和大数据的物联网架构师、高并发架构师、大数据架构师、Java高级工程师在人才市场也随之成为“香饽饽”，Netty、Redis、ZooKeeper、高性能HTTP服务器组件（如Nginx）、高并发Java组件（JUC包）等则成为广大Java工程师所必须掌握的开发技能。

1.1 Netty为何这么火

Netty是JBoss提供的一个Java开源框架，是基于NIO的客户端/服务器编程框架，既能快速开发高并发、高可用、高可靠的网络服务器程序，也能开发高可用、高可靠的客户端程序。

说明

这里的NIO是指非阻塞输入输出（Non-Blocking I/O），也称非阻塞I/O。另外，本书为了行文上的一致性，把输入输出的英文缩写统一为I/O，而不用I/O。

1.1.1 Netty火热的程度

Netty已经有了成百上千的分布式中间件、各种开源项目以及各种商业项目的应用。例如，火爆的Kafka和RocketMQ等消息中间件、火热的ElasticSearch开源搜索引擎、大数据处理Hadoop的RPC框架Avro、分布式通信框架Dubbo，都使用了Netty。总之，使用Netty开发的项目，已经有点数不过来了。

Netty之所以受青睐，是因为它提供了异步的、事件驱动的网络应用程序框架和工具。作为一个异步框架，Netty的所有I/O操作都是异步非阻塞的，通过Future-Listener机制，用户可以方便地主动获取或者通过通知机制获得I/O操作结果。

与JDK原生NIO相比，Netty提供了十分简单易用的API，因而非常适合网络编程。Netty主要是基于NIO来实现的，在Netty中也可以提供阻塞IO的服务。

Netty之所以这么火，与它的巨大优点是密不可分的，大致可以总结如下：

- API使用简单，开发门槛低。
- 功能强大，预置了多种编解码功能，支持多种主流协议。
- 定制能力强，可以通过**ChannelHandler**对通信框架进行灵活扩展。
- 性能高，与其他业界主流的NIO框架相比，Netty的综合性能最优。
- 成熟、稳定，Netty修复了在JDK NIO中所有已发现的Bug，业务开发人员不需要再为NIO的Bug而烦恼。
- 社区活跃，版本迭代周期短，发现的Bug可以被及时修复。

1.1.2 Netty是面试的必杀器

Netty是互联网中间件领域使用最广泛、最核心的网络通信框架，几乎所有Java互联网中间件或者大数据中间件的高性能通信与传输均离不开Netty。所以，掌握Netty是一名初、中级工程师迈向高级工程师的重要技能之一。

目前，主要的互联网公司，例如阿里、腾讯、美团、新浪、淘宝等，在高级工程师的面试过程中经常会问一些高性能通信框架方面的

问题，还会问“你有没有读过什么著名框架的源代码？”之类的问题。

如果掌握了Netty相关的技术，更进一步说，如果你能全面地阅读和掌握Netty源代码，相信到大公司面试时，一定会底气十足，成功在握。

1.2 高并发利器Redis

任何高并发的系统不可或缺的就是缓存。Redis缓存目前已经成为缓存的事实标准。

1.2.1 什么是Redis

Redis是Remote Dictionary Server（远程字典服务器）的缩写，最初是作为数据库的工具来使用的，是目前使用广泛、高效的开源缓存。Redis使用C语言开发，将数据保存在内存中，可以看成是一款纯内存的数据库，所以它的数据存取速度非常快。一些经常用并且创建时间较长的内容可以缓存到Redis中，而应用程序能以极快的速度存取这些内容。举例来说，如果某个页面会经常被访问到，而创建页面时需要多次访问数据库，造成网页内容的生成时间较长，那么就可以使用Redis将这个页面缓存起来，从而减轻网站的负担，降低网站的延迟。

Redis通过键-值对（Key-Value Pair）的形式来存储数据，类似于Java中的Map（映射）。Redis的Key（键）只能是String（字符串）类型，Value（值）则可以是String类型、Map类型、List（列表）类型、Set（集合）类型、SortedSet（有序集合）类型。

Redis的主要应用场景是缓存（数据查询、短连接、新闻内容、商品内容等）、分布式会话（Session）、聊天室的在线好友列表、任务队列（秒杀、抢购、12306等）、应用排行榜、访问统计、数据过期处理（可以精确到毫秒）。

1.2.2 Redis成为缓存事实标准的原因

相对于其他的键-值对内存数据库（如Memcached），Redis具有如下特点：

- (1) 速度快。不需要等待磁盘的IO，而是在内存之间进行数据存储和查询，速度非常快。当然，缓存的数据总量不能太大，因为受到物理内存空间大小的限制。
- (2) 丰富的数据结构，有String、List、Hash、Set、SortedSet五种类型。
- (3) 单线程，避免了线程切换和锁机制的性能消耗。
- (4) 可持久化。支持RDB与AOF两种方式，将内存中的数据写入外部的物理存储设备。
- (5) 支持发布/订阅。
- (6) 支持Lua脚本。
- (7) 支持分布式锁。在分布式系统中，不同的节点需要访问同一个资源时，往往需要通过互斥机制来防止彼此干扰，并且保证数据的一致性。在这种情况下，需要用到分布式锁。分布式锁和Java的锁用于实现不同线程之间的同步访问，原理上是类似的。
- (8) 支持原子操作和事务。Redis事务是一组命令的集合。一个事务中的命令要么都执行，要么都不执行。如果命令在运行期间出现错误，不会自动回滚。

(9) 支持主-从（Master-Slave）复制与高可用（Redis Sentinel）集群（3.0版本以上）。

(10) 支持管道。Redis管道是指客户端可以将多个命令一次性发送到服务器，然后由服务器一次性返回所有结果。管道技术的优点是，在批量执行命令的应用场景中，可以大大减少网络传输的开销，提高性能。

1.3 分布式利器ZooKeeper

单体应用在达到性能瓶颈之后，就必须靠分布式集群解决高并发问题，而集群的分布式架构和集群节点之间的交互一定少不了可靠的分布式协调工具，ZooKeeper就是目前极为重要的分布式协调工具。

1.3.1 什么是ZooKeeper

ZooKeeper最早起源于雅虎公司研究院的一个研究小组。当时，研究人员发现，在雅虎内部很多大型的系统需要依赖一个类似的系统进行分布式协调，但是这些系统往往存在分布式单点问题，所以雅虎的开发人员就试图开发一个通用的无单点问题的分布式协调框架。

此框架的命名过程也是非常有趣的。在项目初期给这个项目命名时，准备和很多项目一样，按照雅虎公司的惯例使用动物的名字来命名（例如著名的Pig项目）。在探讨取什么名字的时候，研究院的首席科学家Raghu Ramakrishnan开玩笑说：“再这样下去，我们这儿就变成动物园了。”此话一出，大家纷纷表示新框架就叫动物园管理员吧，于是ZooKeeper（动物园管理员）诞生了。而ZooKeeper正好是用来协调分布式环境的不同节点的，形象地说，可以理解为协调各个以动物命名的分布式组件，所以ZooKeeper也就“名副其实”了。

1.3.2 ZooKeeper的优势

ZooKeeper的核心优势是实现了分布式环境的数据一致性，简单地说：每时每刻我们访问ZooKeeper的树结构时，不同的节点返回的数据

都是一致的。也就是说，对ZooKeeper进行数据访问时，无论是什么时间，都不会引起“脏读”“幻读”“不可重复读”问题。

“脏读”“幻读”“不可重复读”是数据库事务的概念，当然，ZooKeeper也可以被理解为一种简单的分布式数据库。“脏读”是指一个事务中访问到了另外一个事务未提交的数据。“不可重复读”是指在一个事务内根据同一个条件对数据进行多次查询，但是结果却不一样，原因是其他事务对该数据进行了修改。“幻读”是指当两个完全相同的查询执行时，第二次查询所返回的结果集和第一次查询所返回的结果集不相同，原因也是另外一个事务新增、删除了第一个事务结果集中的数据。

说明

“不可重复读”和“幻读”的区别是：“不可重复读”关注的重点在于记录的更新操作，对同样的记录，再次读取后发现返回的数据值不一样了；“幻读”关注的重点在于记录新增或者删除操作（数据条数发生了变化），同样的条件第一次和第二次查询出来的记录数不一样。

ZooKeeper对不同系统环境的支持都很好，在绝大多数主流的操作系统上都能够正常运行，如GNU/Linux、Sun Solaris、Win32以及MacOS等。但是，ZooKeeper官方文档中特别强调，由于FreeBSD系统的JVM实现对Java的NIO Selector（选择器）支持得不是很好，因此不建议在FreeBSD系统上部署ZooKeeper生产服务器。

可以说，ZooKeeper提供的是分布式系统中非常底层且必不可少的基本功能，如果开发者自己来实现这些功能且达到高吞吐、低延迟，同时还要保持一致性和可用性，实际上是非常困难的。借助ZooKeeper提供的这些功能，开发者就可以轻松地在ZooKeeper之上构建自己的各种分布式系统。

1.4 高性能HTTP通信技术

和传统的Web应用有所不同，高并发的5G应用、物联网应用、工业互联应用、大数据应用、人工智能应用基本上都是大流量应用，QPS（Query Per Second，每秒查询率）在十万级甚至上千万级，在这些高并发应用中，如何使用高并发HTTP通信技术去提升内部各个节点的通信性能，对于提升分布式系统整体的吞吐量有着非常重大的作用。

1.4.1 十万级以上高并发场景中的高并发HTTP通信技术

QPS在十万级的Web应用架构大致如图1-1所示。

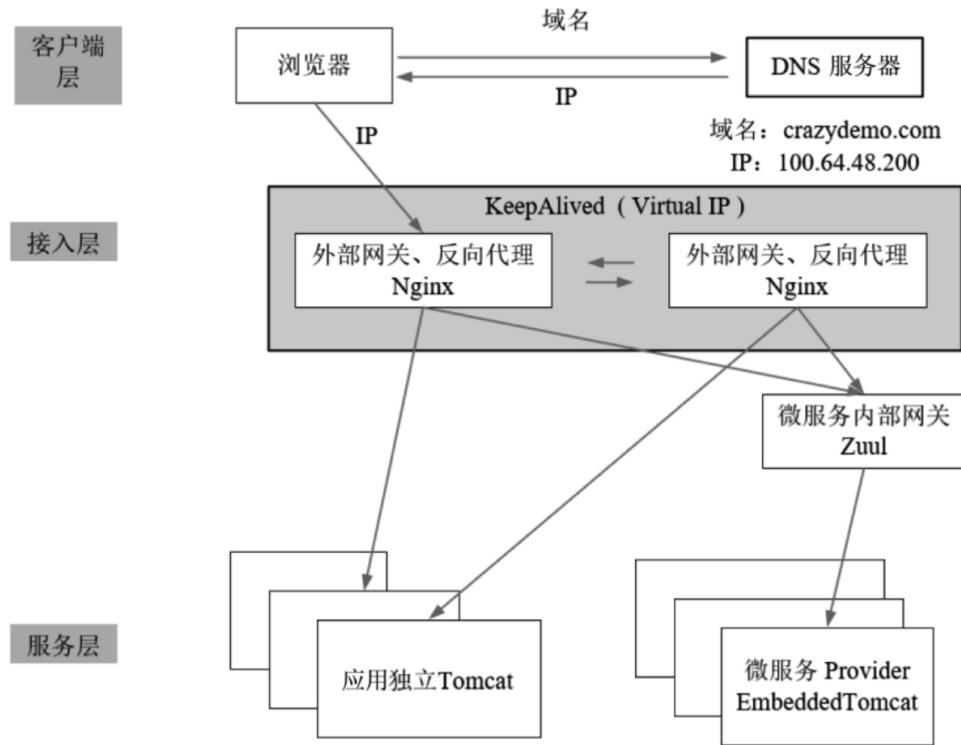


图1-1 十万级QPS的Web应用架构图

对于十万级流量的系统应用而言，其架构一般可以分为三层：服务层、接入层、客户端层。

服务层一般执行的是Java应用程序，可以细分为传统的单体应用和目前主流的Spring Cloud分布式应用。传统的单体Java应用执行在Tomcat服务器上，目前主流的Spring Cloud微服务应用执行在内嵌的Tomcat服务器上。

接入层主要完成鉴权、限流、反向代理和负载均衡等功能。由于在静态资源、登录验证等简单逻辑的处理性能上Nginx和Tomcat不可同日而语（一般在10倍以上），因此接入层基本上都是使用Nginx + Lua扩展作为接入服务器。另外，为了保证Nginx接入服务器的高可用，会

搭建有冗余的接入服务器，然后使用KeepAlive中间件进行高可用监控管理并且虚拟出外部IP，供外部访问。

说明

Nginx是一个强大的Web服务器软件，用于处理高并发的HTTP请求和作为反向代理服务器进行负载均衡，具有高性能、轻量级、内存消耗少、强大的负载均衡能力等优势。有关Nginx的原理知识，请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

对于十万级QPS流量的Web应用，如果流量增长到百万级，可以对接入层Nginx进行横向扩展，甚至可以引入LVS进行负载均衡。

QPS在千万级的Web应用架构大致如图1-2所示。

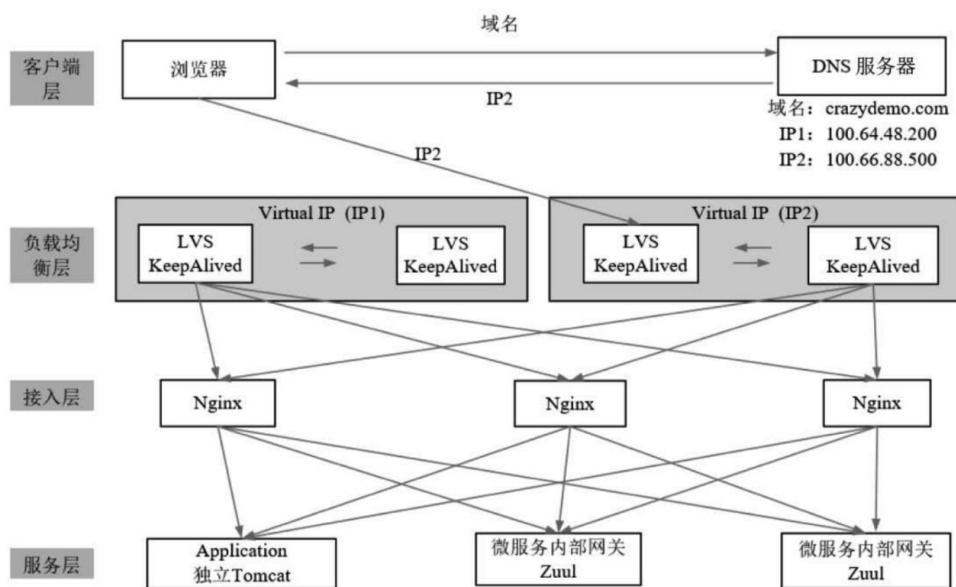


图1-2 千万级QPS的Web应用架构图

对于千万级QPS的Web应用，除了服务层的独立Tomcat或者Spring Cloud微服务节点需要进行不断的横向扩展之外，还需要进行以下两大增强：

- (1) 引入LVS负载均衡层，进行请求分发和接入层的负载均衡。
- (2) 引入DNS服务器的负载均衡，可以在域名下面添加多个IP，由DNS服务器进行多个IP之间的负载均衡，甚至可以按照就近原则为用户返回最近的服务器IP地址。

总之，如何抵抗十万级甚至千万级QPS访问洪峰，涉及大量的开发知识、运维知识。对于开发人员来说，并不一定需要掌握太多的操作系统层面（如LVS）运维知识，主要原因是术业有专攻，一般会有专业的运维人员去解决系统的运行问题。但是对千万级QPS系统中所涉及的高并发方面的开发知识，则是开发人员必须掌握的。

在十万级甚至千万级QPS的Web应用架构中，如何提高平台内部的接入层Nginx和服务层Tomcat（或者其他Java容器）之间的HTTP通信能力，涉及高并发HTTP通信这个核心技术问题，这是本书后面章节会从TCP、HTTP层面出发所重点剖析和解读的问题。

1.4.2 微服务之间的高并发RPC技术

在基于Spring Cloud技术架构的分布式Web应用中，微服务Provider（服务节点）之间存在着大量的RPC，具体如图1-3所示。

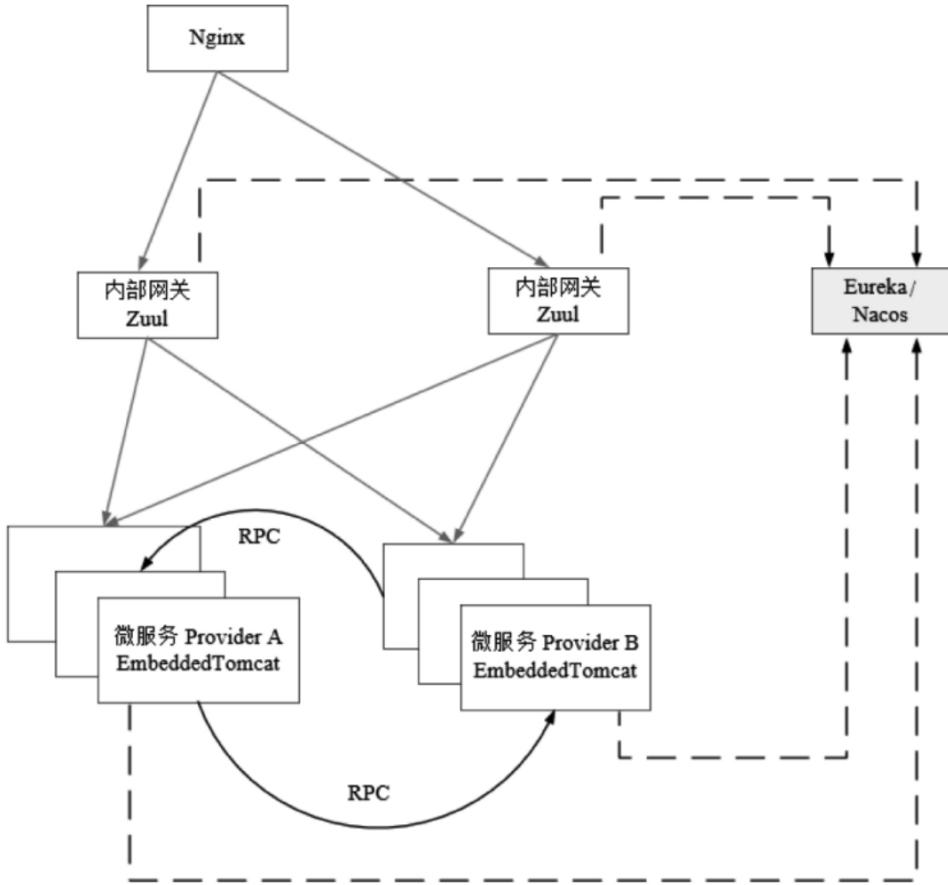


图1-3 微服务Provider（服务节点）之间的RPC调用示意图

微服务Provider实例之间的RPC在Spring Cloud全家桶技术体系中是由Feign基于Ribbon完成的，并由Hystrix组件提供RPC的熔断、回退、限流等保护。

说明

分布式微服务架构目前已经成为Java应用的主流架构，在接入层同样会与Nginx结合，所以常常都是Nginx + Spring Cloud架构

(有关该架构的原理知识，请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》)。

Spring Cloud并没有提供高性能RPC通信（HTTP通信）的技术方案，通过配置可以借助Apache HttpClient或者OkHttp等通信组件实现HTTP高性能通信。由于HTTP高性能通信涉及底层socket连接（TCP连接）的复用管理，甚至涉及TCP、HTTP等一系列非常基础、原理的知识，因此在《Spring Cloud、Nginx高并发核心编程》一书中并没有对高并发HTTP通信进行介绍，而是将这些知识放在本书中。

1.5 高并发IM的综合实战

为了方便交流和学习，笔者组织了一群高性能、高并发的发烧友，成立了一个高性能社群——“疯狂创客圈”。同时，牵头组织社群的小伙伴们应用Netty、Redis、ZooKeeper持续迭代一个高并发学习项目——CrazyIM。

1.5.1 高并发IM的学习价值

为什么在成为Java高级工程师甚至架构师的学习路上建议大家完成一个高并发IM（即时通信）项目呢？

首先，通过实战完成一个分布式、高并发的IM系统具有相当大的技术挑战性。对于传统的企业级Web开发者来说，这相当于进入了一片全新的天地。企业级Web的QPS峰值可能在1000以内，甚至100以内，没有多少技术挑战性和含金量，属于重复性的CRUD（Create，创建；Retrieve，查询；Update，更新；Delete，删除）的体力活。一个分布式、高并发的IM系统面临的QPS峰值可能在十万、百万、千万甚至上亿级别。层次化、递进的高并发需求将无极限地考验着系统的性能，从通信协议到系统的架构需要不断地进行优化，这对技术能力是一种非常极致的考验和训练。

其次，就具有不同QPS峰值规模的IM系统而言，它们所处的用户需求环境是不一样的。这就造成了不同用户规模的IM系统各自具有一定的市场需求和实际需要，因而它们不一定都需要上亿级的高并发。但是，作为一个顶级的架构师，应该具备全栈式的架构能力，对不同用

户规模、差异化的应用场景构建出相匹配的高并发IM系统。也就是说，IM系统的综合性相对较强，相关的技术涉及满足各种不同应用场景的网络传输、分布式协调、分布式缓存、服务化架构等方面。

接下来具体看看高并发IM的应用场景。

1.5.2 庞大的应用场景

可以说，大部分高并发实时通信、消息推送的应用场景都需要高并发IM。随着移动互联网、AI的飞速发展，高性能、高并发IM有着非常广泛的应用场景。

高并发IM典型的应用场景有私信、聊天、大规模推送、弹幕、实时定位、在线教育、智能家居、互动游戏、抽奖等，如图1-4所示。

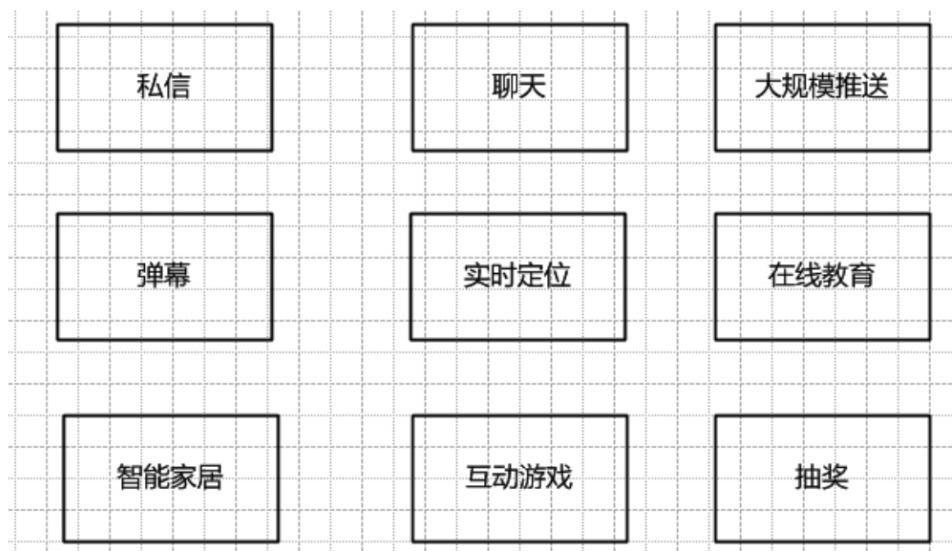


图1-4 高并发IM典型的应用场景

尤其是对于从事APP开发的小伙伴们来说，IM已经成为大多数APP的标配。在移动互联网时代，推送（Push）服务成为APP不可或缺的重要组成部分，可以提升用户的活跃度和留存率。我们的手机每天接收到各种各样的广告和提示消息等，其中大多数都是通过推送服务实现的。

随着5G时代物联网的发展，未来所有接入物联网的智能设备都将是IM系统的客户端，这就意味着推送服务会在未来面临海量的设备和终端接入。为了支持这些千万级、上亿级的终端，一定需要强悍的后台系统。

有这么多的应用场景，对于想成为Java高手的小伙伴们，高并发IM是一个绕不开的技术难题。对于想在后台有所成就的小伙伴们，高并发IM实战更是在成为顶级工程师甚至架构师的道路上所必须经历的。

第2章 高并发IO的底层原理

本书的原则是：从基础讲起。IO底层原理是隐藏在Java编程知识之下的基础知识，是开发人员必须掌握的基本原理，可以说是基础的基础，更是大公司面试通关的必备知识。

本章从操作系统的底层原理入手，通过图文并茂的方式为大家深入剖析高并发IO的底层原理，并介绍如何通过设置来让操作系统支持高并发。

2.1 IO读写的基本原理

为了避免用户进程直接操作内核，保证内核安全，操作系统将内存（虚拟内存）划分为两部分：一部分是内核空间（Kernel-Space），另一部分是用户空间（User-Space）。在Linux系统中，内核模块运行在内核空间，对应的进程处于内核态；用户程序运行在用户空间，对应的进程处于用户态。

操作系统的内核是内核程序，它独立于普通的应用程序，既有权限访问受保护的内核空间，也有权限访问硬件设备，而普通的应用程序并没有这样的权限。内核空间总是驻留在内存中，是为操作系统的内核保留的。应用程序不允许直接在内核空间区域进行读写，也不允许直接调用内核代码定义的函数。每个应用程序进程都有一个单独的用户空间，对应的进程处于用户态，用户态进程不能访问内核空间中的数据，也不能直接调用内核函数，因此需要将进程切换到内核态才能进行系统调用。

内核态进程可以执行任意命令，调用系统的一切资源，而用户态进程只能执行简单的运算，不能直接调用系统资源，那么问题来了：用户态进程如何执行系统调用呢？答案是：用户态进程必须通过系统调用（System Call）向内核发出指令，完成调用系统资源之类的操作。

说明

如果没有特别声明，本书后文所提到的内核是指操作系统的内核。

用户程序进行IO的读写依赖于底层的IO读写，基本上会用到底层的read和write两大系统调用。虽然在不同的操作系统中read和write两大系统调用的名称和形式可能不完全一样，但是它们的基本功能是一样的。

操作系统层面的read系统调用并不是直接从物理设备把数据读取到应用的内存中，write系统调用也不是直接把数据写入物理设备。上层应用无论是调用操作系统的read还是调用操作系统的write，都会涉及缓冲区。具体来说，上层应用通过操作系统的read系统调用把数据从内核缓冲区复制到应用程序的进程缓冲区，通过操作系统的write系统调用把数据从应用程序的进程缓冲区复制到操作系统的内核缓冲区。

简单来说，应用程序的IO操作实际上不是物理设备级别的读写，而是缓存的复制。read和write两大系统调用都不负责数据在内核缓冲区和物理设备（如磁盘、网卡等）之间的交换。这个底层的读写交换操作是由操作系统内核（Kernel）来完成的。所以，在应用程序中，无论是对socket的IO操作还是对文件的IO操作，都属于上层应用的开发，它们在输入（Input）和输出（Output）维度上的执行流程是类似的，都是在内核缓冲区和进程缓冲区之间进行数据交换。

2.1.1 内核缓冲区与进程缓冲区

为什么设置那么多的缓冲区，导致读写过程那么麻烦呢？

缓冲区的目的是减少与设备之间的频繁物理交换。计算机的外部物理设备与内存和CPU相比，有着非常大的差距，外部设备的直接读写涉及操作系统的中断。发生系统中断时，需要保存之前的进程数据和状态等信息，结束中断之后，还需要恢复之前的进程数据和状态等信息。为了减少底层系统的频繁中断所导致的时间损耗、性能损耗，出现了内核缓冲区。

操作系统会对内核缓冲区进行监控，等待缓冲区达到一定数量的时候，再进行I/O设备的中断处理，集中执行物理设备的实际I/O操作，通过这种机制来提升系统的性能。至于具体什么时候执行系统中断（包括读中断、写中断）则由操作系统的内核来决定，应用程序不需要关心。

上层应用使用read系统调用时，仅仅把数据从内核缓冲区复制到应用的缓冲区（进程缓冲区）；上层应用使用write系统调用时，仅仅把数据从应用的缓冲区复制到内核缓冲区。

内核缓冲区与应用缓冲区在数量上也不同。在Linux系统中，操作系统内核只有一个内核缓冲区。每个用户程序（进程）都有自己独立的缓冲区，叫作用户缓冲区或者进程缓冲区。在大多数情况下，Linux系统中用户程序的I/O读写程序并没有进行实际的I/O操作，而是在用户缓冲区和内核缓冲区之间直接进行数据的交换。

2.1.2 典型的系统调用流程

用户程序所使用的系统调用read和write并不是使数据在内核缓冲区和物理设备之间交换：read调用把数据从内核缓冲区复制到应用的用户缓冲区，write调用把数据从应用的用户缓冲区复制到内核缓冲区。两个系统调用的大致流程如图2-1所示。

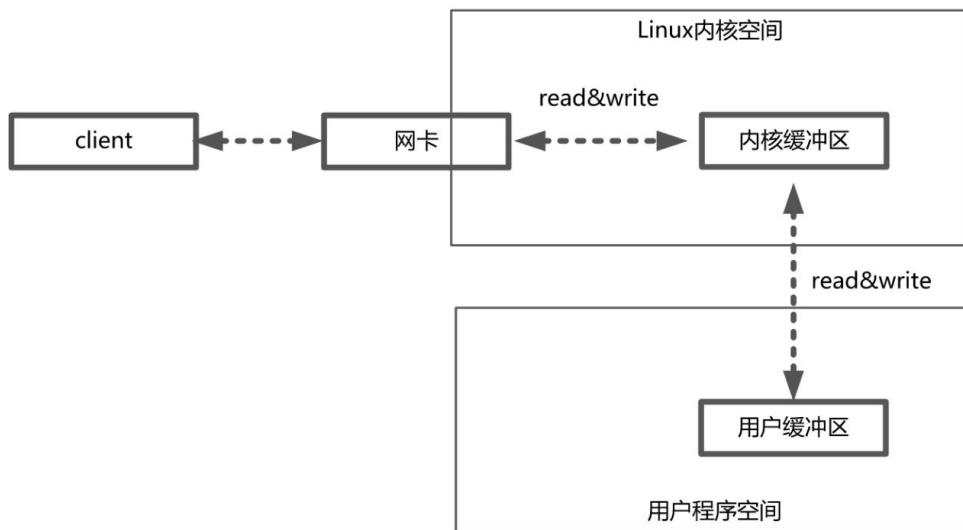


图2-1 系统调用read和write的流程

这里以read系统调用为例，看一下一个完整输入流程的两个阶段：

- 应用程序等待数据准备好。
- 从内核缓冲区向用户缓冲区复制数据。

如果是读取一个socket（套接字），那么以上两个阶段的具体处理流程如下：

- 第一个阶段，应用程序等待数据通过网络到达网卡，当所等待的分组到达时，数据被操作系统复制到内核缓冲区中。这个工作由操作系统自动完成，用户程序无感知。

- 第二个阶段，内核将数据从内核缓冲区复制到应用的用户缓冲区。

再具体一点，如果是在Java客户端和服务端之间完成一次socket请求和响应（包括read和write）的数据交换，其完整的流程如下：

- 客户端发送请求：Java客户端程序通过**write**系统调用将数据复制到内核缓冲区，Linux将内核缓冲区的请求数据通过客户端机器的网卡发送出去。在服务端，这份请求数据会从接收网卡中读取到服务端机器的内核缓冲区。
- 服务端获取请求：Java服务端程序通过**read**系统调用从Linux内核缓冲区读取数据，再送入Java进程缓冲区。
- 服务端业务处理：Java服务器在自己的用户空间中完成客户端的请求所对应的业务处理。
- 服务端返回数据：Java服务器完成处理后，构建好的响应数据将从用户缓冲区写入内核缓冲区，这里用到的是**write**系统调用，操作系统会负责将内核缓冲区的数据发送出去。
- 发送给客户端：服务端Linux系统将内核缓冲区中的数据写入网卡，网卡通过底层的通信协议将数据发送给目标客户端。

由于生产环境的Java高并发应用基本都运行在Linux操作系统上，因此以上案例中的操作系统以Linux作为实例。

2.2 四种主要的IO模型

服务端高并发IO编程往往要求的性能都非常高，一般情况下需要选用高性能的IO模型。另外，对于Java工程师来说，有关IO模型的知识也是通过大公司面试的必备知识。本章从最为基础的模型开始为大家揭秘IO模型的核心原理。

常见的IO模型有四种。

1. 同步阻塞IO

首先，解释一下阻塞与非阻塞。阻塞IO指的是需要内核IO操作彻底完成后才返回到用户空间执行用户程序的操作指令。“阻塞”指的是用户程序（发起IO请求的进程或者线程）的执行状态。可以说传统的IO模型都是阻塞IO模型，并且在Java中默认创建的socket都属于阻塞IO模型。

其次，解释一下同步与异步。简单来说，可以将同步与异步看成发起IO请求的两种方式。同步IO是指用户空间（进程或者线程）是主动发起IO请求的一方，系统内核是被动接收方。异步IO则反过来，系统内核是主动发起IO请求的一方，用户空间是被动接收方。

同步阻塞IO（Blocking IO）指的是用户空间（或者线程）主动发起，需要等待内核IO操作彻底完成后才返回到用户空间的IO操作。在IO操作过程中，发起IO请求的用户进程（或者线程）处于阻塞状态。

2. 同步非阻塞IO

非阻塞IO（Non-Blocking IO, NIO）指的是用户空间的程序不需要等待内核IO操作彻底完成，可以立即返回用户空间去执行后续的指令，即发起IO请求的用户进程（或者线程）处于非阻塞状态，与此同时，内核会立即返回给用户一个IO状态值。

阻塞和非阻塞的区别是什么呢？阻塞是指用户进程（或者线程）一直在等待，而不能做别的事情；非阻塞是指用户进程（或者线程）获得内核返回的状态值就返回自己的空间，可以去做别的事情。在Java中，非阻塞IO的socket被设置为NONBLOCK模式。

说明

同步非阻塞IO也可以简称为NIO，但是它不是Java编程中的NIO。Java编程中的NIO（New IO）类库组件所归属的不是基础IO模型中的NIO模型，而是IO多路复用模型。

同步非阻塞IO指的是用户进程主动发起，不需要等待内核IO操作彻底完成就能立即返回用户空间的IO操作。在IO操作过程中，发起IO请求的用户进程（或者线程）处于非阻塞状态。

3. IO多路复用

为了提高性能，操作系统引入了一种新的系统调用，专门用于查询IO文件描述符（含socket连接）的就绪状态。在Linux系统中，新的系统调用为select/epoll系统调用。通过该系统调用，一个用户进程（或者线程）可以监视多个文件描述符，一旦某个描述符就绪（一般

是内核缓冲区可读/可写），内核就能够将文件描述符的就绪状态返回给用户进程（或者线程），用户空间可以根据文件描述符的就绪状态进行相应的I0系统调用。

I0多路复用（I0 Multiplexing）属于一种经典的Reactor模式实现，有时也称为异步阻塞I0，Java中的Selector属于这种模型。

4. 异步I0

异步I0（Asynchronous I0， AI0）指的是用户空间的线程变成被动接收者，而内核空间成为主动调用者。在异步I0模型中，当用户线程收到通知时，数据已经被内核读取完毕并放在了用户缓冲区内，内核在I0完成后通知用户线程直接使用即可。

异步I0类似于Java中典型的回调模式，用户进程（或者线程）向内核空间注册了各种I0事件的回调函数，由内核去主动调用。

接下来对以上4种常见的I0模型进行详细的介绍。

2.2.1 同步阻塞I0

默认情况下，在Java应用程序进程中所有对socket连接进行的I0操作都是同步阻塞I0。

在阻塞式I0模型中，从Java应用程序发起I0系统调用开始，一直到系统调用返回，这段时间内发起I0请求的Java进程（或者线程）是阻塞的。直到返回成功后，应用进程才能开始处理用户空间的缓冲区数据。

同步阻塞IO的具体流程如图2-2所示。

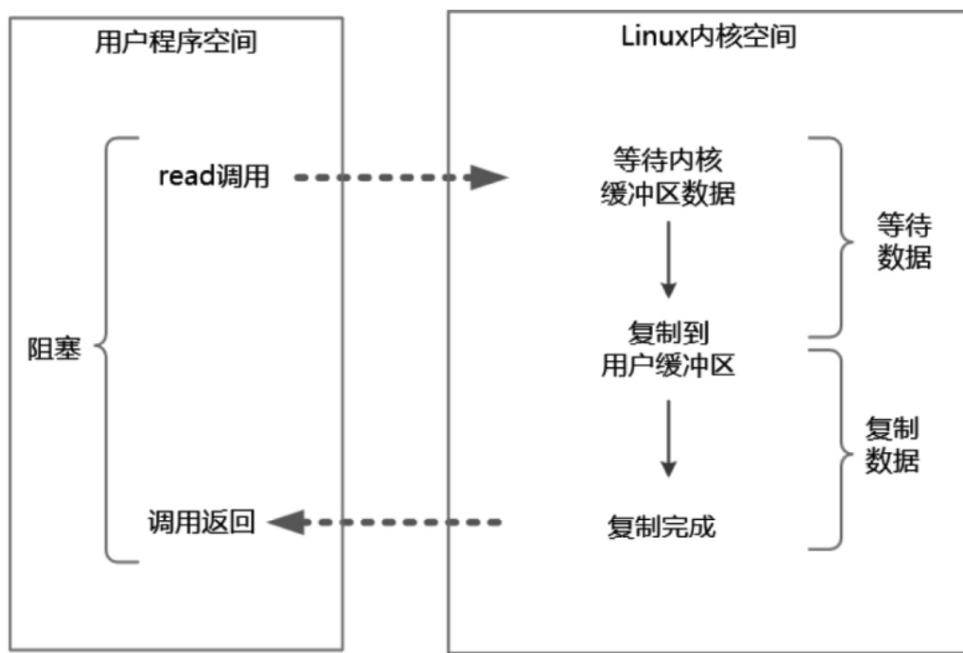


图2-2 同步阻塞I/O的流程

举个例子，在Java中发起一个socket的read操作的系统调用，流程大致如下：

- (1) 从Java进行I/O读后发起read系统调用开始，用户线程（或者线程）就进入阻塞状态。
- (2) 当系统内核收到read系统调用后就开始准备数据。一开始，数据可能还没有到达内核缓冲区（例如，还没有收到一个完整的socket数据包），这时内核就要等待。
- (3) 内核一直等到完整的数据到达，就会将数据从内核缓冲区复制到用户缓冲区（用户空间的内存），然后内核返回结果（例如返回复制到用户缓冲区中的字节数）。

(4) 直到内核返回后用户线程才会解除阻塞的状态，重新运行起来。

阻塞IO的特点是在内核执行IO操作的两个阶段，发起IO请求的用户进程（或者线程）被阻塞了。

阻塞IO的优点是：应用程序开发非常简单；在阻塞等待数据期间，用户线程挂起，基本不会占用CPU资源。

阻塞IO的缺点是：一般情况下会为每个连接配备一个独立的线程，一个线程维护一个连接的IO操作。在并发量小的情况下，这样做没有什么问题。在高并发的应用场景下，阻塞IO模型需要大量的线程来维护大量的网络连接，内存、线程切换开销会非常巨大，性能很低，基本上是不可用的。

2.2.2 同步非阻塞IO

在Linux系统下，socket连接默认是阻塞模式，可以将socket设置成非阻塞模式。在NIO模型中，应用程序一旦开始IO系统调用，就会出现以下两种情况：

(1) 在内核缓冲区中没有数据的情况下，系统调用会立即返回一个调用失败的信息。

(2) 在内核缓冲区中有数据的情况下，在数据的复制过程中系统调用是阻塞的，直到完成数据从内核缓冲区复制到用户缓冲区。复制完成后，系统调用返回成功，用户进程（或者线程）可以开始处理用户空间的缓冲区数据。

同步非阻塞IO的流程如图2-3所示。

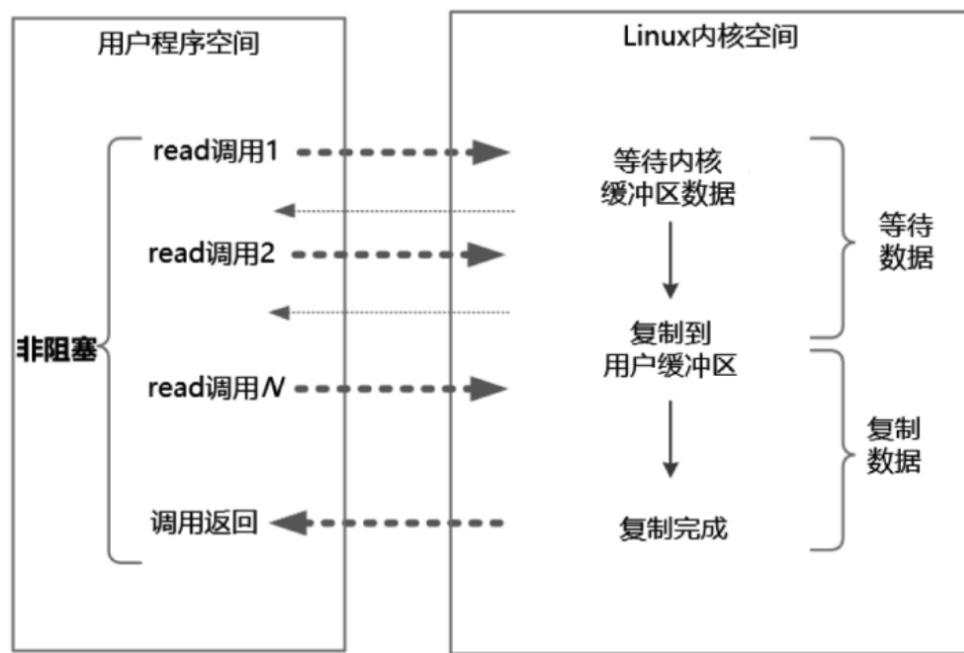


图2-3 同步非阻塞I/O的流程

举个例子，发起一个非阻塞socket的read操作的系统调用，流程如下：

(1) 在内核数据没有准备好的阶段，用户线程发起I/O请求时立即返回。所以，为了读取最终的数据，用户进程（或者线程）需要不断地发起I/O系统调用。

(2) 内核数据到达后，用户进程（或者线程）发起系统调用，用户进程（或者线程）阻塞。内核开始复制数据，它会将数据从内核缓冲区复制到用户缓冲区，然后内核返回结果（例如返回复制到的用户缓冲区的字节数）。

(3) 用户进程（或者线程）读到数据后，才会解除阻塞状态，重新运行起来。也就是说，用户空间需要经过多次尝试才能保证最终真

正读到数据，而后继续执行。

同步非阻塞IO的特点是应用程序的线程需要不断地进行IO系统调用，轮询数据是否已经准备好，如果没有准备好就继续轮询，直到完成IO系统调用为止。

同步非阻塞IO的优点是每次发起的IO系统调用在内核等待数据过程中可以立即返回，用户线程不会阻塞，实时性较好。

同步非阻塞IO的缺点是不断地轮询内核，这将占用大量的CPU时间，效率低下。

总体来说，在高并发应用场景中，同步非阻塞IO是性能很低的，也是基本不可用的，一般Web服务器都不使用这种IO模型。在Java的实际开发中，不会涉及这种IO模型，但是此模型还是有价值的，其作用在于其他IO模型中可以使用非阻塞IO模型作为基础，以实现其高性能。

2.2.3 IO多路复用

如何避免同步非阻塞IO模型中轮询等待的问题呢？答案是采用IO多路复用模型。

目前支持IO多路复用的系统调用有select、epoll等。几乎所有的操作系统都支持select系统调用，它具有良好的跨平台特性。epoll是在Linux 2.6内核中提出的，是select系统调用的Linux增强版本。

在IO多路复用模型中通过select epoll系统调用，单个应用程序的线程可以不断地轮询成百上千的socket连接的就绪状态，当某个或

者某些socket网络连接有I0就绪状态时就返回这些就绪的状态（或者说就绪事件）。

举个例子来说明I0多路复用模型的流程。发起一个多路复用I0的read操作的系统调用，流程如下：

(1) 选择器注册。首先，将需要read操作的目标文件描述符(socket连接)提前注册到Linux的select/epoll选择器中，在Java中所对应的选择器类是Selector类。然后，开启整个I0多路复用模型的轮询流程。

(2) 就绪状态的轮询。通过选择器的查询方法，查询所有提前注册过的文件描述符(socket连接)的I0就绪状态。通过查询的系统调用，内核会返回一个就绪的socket列表。当任何一个注册过的socket中的数据准备好或者就绪了就说明内核缓冲区有数据了，内核将该socket加入就绪的列表中，并且返回就绪事件。

(3) 用户线程获得了就绪状态的列表后，根据其中的socket连接发起read系统调用，用户线程阻塞。内核开始复制数据，将数据从内核缓冲区复制到用户缓冲区。

(4) 复制完成后，内核返回结果，用户线程才会解除阻塞的状态，用户线程读取到了数据，继续执行。

说明

在用户进程进行IO就绪事件的轮询时，需要调用选择器的select查询方法，发起查询的用户进程或者线程是阻塞的。当然，如果使用了查询方法的非阻塞的重载版本，发起查询的用户进程或者线程也不会阻塞，重载版本会立即返回。

IO多路复用模型的read系统调用流程如图2-4所示。

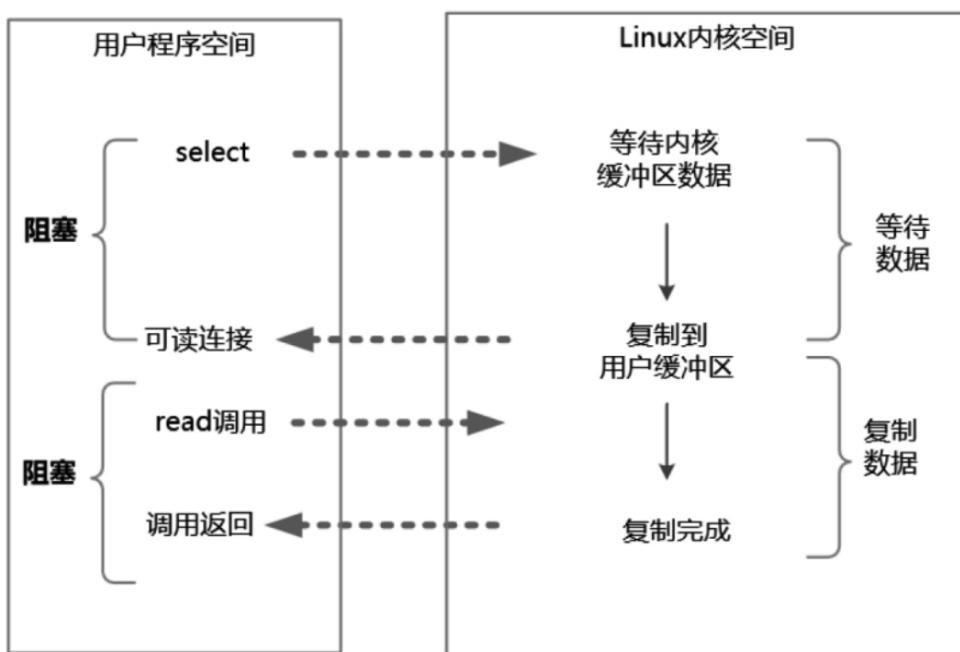


图2-4 IO多路复用模型的read系统调用流程

IO多路复用模型的特点是：IO多路复用模型的IO涉及两种系统调用，一种是IO操作的系统调用，另一种是select/epoll就绪查询系统调用。IO多路复用模型建立在操作系统的基础设施之上，即操作系统的内核必须能够提供多路分离的系统调用select/epoll。

和NIO模型相似，多路复用IO也需要轮询。负责select/epoll状态查询调用的线程，需要不断地进行select/epoll轮询，以找出达到IO

操作就绪的socket连接。

I0多路复用模型与同步非阻塞I0模型是有密切关系的，具体来说，注册在选择器上的每一个可以查询的socket连接一般都设置成同步非阻塞模型，只是这一点对于用户程序而言是无感知的。

I0多路复用模型的优点是一个选择器查询线程可以同时处理成千上万的网络连接，所以用户程序不必创建大量的线程，也不必维护这些线程，从而大大减少了系统的开销。与一个线程维护一个连接的阻塞I0模式相比，这一点是I0多路复用模型的最大优势。

通过JDK的源码可以看出，Java语言的NIO组件在Linux系统上是使用epoll系统调用实现的。所以，Java语言的NIO组件所使用的就是I0多路复用模型。

I0多路复用模型的缺点是，本质上select/epoll系统调用是阻塞式的，属于同步I0，需要在读写事件就绪后由系统调用本身负责读写，也就是说这个读写过程是阻塞的。要彻底地解除线程的阻塞，就必须使用异步I0模型。

2.2.4 异步I0

异步I0模型的基本流程是：用户线程通过系统调用向内核注册某个I0操作。内核在整个I0操作（包括数据准备、数据复制）完成后通知用户程序，用户执行后续的业务操作。

在异步I0模型中，在整个内核的数据处理过程（包括内核将数据从网络物理设备（网卡）读取到内核缓冲区、将内核缓冲区的数据复

制到用户缓冲区)中，用户程序都不需要阻塞。

异步I/O模型的流程如图2-5所示。

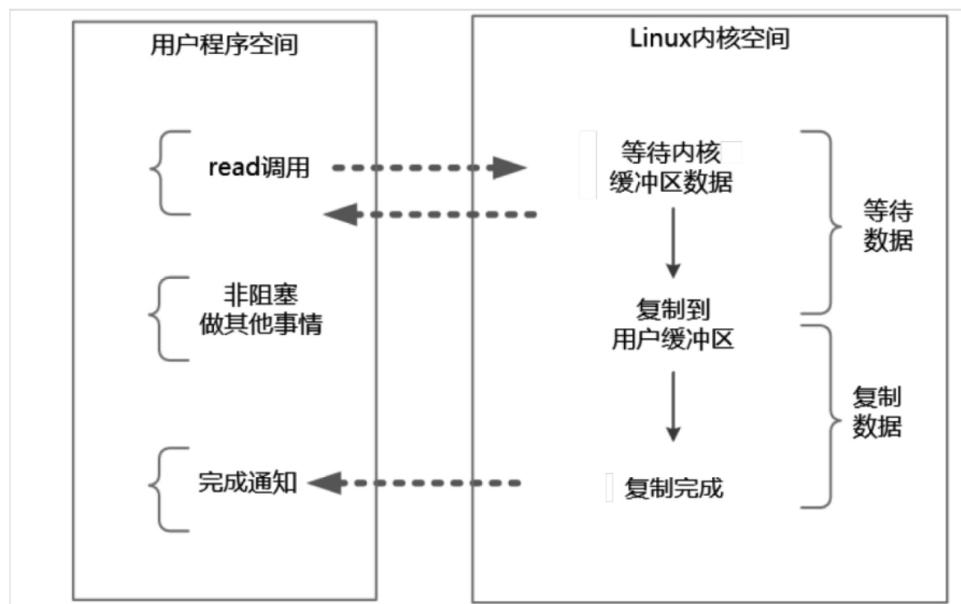


图2-5 异步I/O模型的流程

举个例子，发起一个异步I/O的read操作的系统调用，流程如下：

- (1) 当用户线程发起了read系统调用后，立刻就可以去做其他的事，用户线程不阻塞。
- (2) 内核开始I/O的第一个阶段：准备数据。准备好数据，内核就会将数据从内核缓冲区复制到用户缓冲区。
- (3) 内核会给用户线程发送一个信号（Signal），或者回调用户线程注册的回调方法，告诉用户线程read系统调用已经完成，数据已经读入用户缓冲区。
- (4) 用户线程读取用户缓冲区的数据，完成后续的业务操作。

异步I0模型的特点是在内核等待数据和复制数据的两个阶段，用户线程都不是阻塞的。用户线程需要接收内核的I0操作完成的事件，或者用户线程需要注册一个I0操作完成的回调函数。正因为如此，异步I0有的时候也被称为信号驱动I0。

异步I0模型的缺点是应用程序仅需要进行事件的注册与接收，其余的工作都留给了操作系统，也就是说需要底层内核提供支持。

理论上来说，异步I0是真正的异步输入输出，它的吞吐量高于I0多路复用模型的吞吐量。就目前而言，Windows系统下通过IOCP实现了真正的异步I0。在Linux系统下，异步I0模型在2.6版本才引入，JDK对它的支持目前并不完善，因此异步I0在性能上没有明显的优势。

大多数高并发服务端的程序都是基于Linux系统的。因而，目前这类高并发网络应用程序的开发大多采用I0多路复用模型。大名鼎鼎的Netty框架使用的就是I0多路复用模型，而不是异步I0模型。

2.3 通过合理配置来支持百万级并发连接

本章所聚焦的主题是高并发IO的底层原理。前面已经深入浅出地介绍了高并发IO的模型，但是即使采用了最先进的模型，如果不进行合理的操作系统配置，也没有办法支撑百万级的并发网络连接。在生产环境中，大家都使用Linux系统，所以后续内容如果没有特别说明，所指的操作系统都是Linux系统。

这里所涉及的配置就是Linux操作系统中文件句柄数的限制。在生产环境Linux系统中，基本上都需要解除文件句柄数的限制。原因是Linux系统的默认值为1024，也就是说，一个进程最多可以接受1024个socket连接，这是远远不够的。

文件句柄也叫文件描述符。在Linux系统中，文件可分为普通文件、目录文件、链接文件和设备文件。文件描述符（File Descriptor）是内核为了高效管理已被打开的文件所创建的索引，是一个非负整数（通常是小整数），用于指代被打开的文件。所有的IO系统调用（包括socket的读写调用）都是通过文件描述符完成的。

在Linux下，通过调用ulimit命令可以看到一个进程能够打开的最大文件句柄数量。这个命令的具体使用方法是：

```
ulimit -n
```

ulimit命令是用来显示和修改当前用户进程的基础限制命令，-n选项用于引用或设置当前的文件句柄数量的限制值，Linux系统的默认值为1024。

理论上，1024个文件描述符对绝大多数应用（例如Apache、桌面应用程序）来说已经足够，对于一些用户基数很大的高并发应用则是远远不够的。一个高并发的应用面临的并发连接数往往是十万级、百万级、千万级，甚至像腾讯QQ一样的上亿级。

文件句柄数不够，会导致什么后果呢？当单个进程打开的文件句柄数量超过了系统配置的上限值时会发出“Socket/File:Can't open so many files”的错误提示。

所以，对于高并发、高负载的应用，必须调整这个系统参数，以适应并发处理大量连接的应用场景。可以通过ulimit来设置这两个参数，方法如下：

```
ulimit -n 1000000
```

在上面的命令中，n的值设置越大，可以打开的文件句柄数量越大。建议以root用户来执行此命令。

使用ulimit命令有一个缺陷，即该命令只能修改当前用户环境的一些基础限制，仅在当前用户环境有效。也就是说，在当前的终端工具连接当前shell期间，修改是有效的，一旦断开用户会话，或者说用户退出Linux，它的数值就又变回系统默认的1024了。并且，系统重启后，句柄数量会恢复为默认值。

ulimit命令只能用于临时修改，如果想永久地把最大文件描述符数量值保存下来，可以编辑/etc/rc.local开机启动文件，在文件中添加如下内容：

```
ulimit -SHn 1000000
```

以上示例增加了-S和-H两个命令选项。选项-S表示软性极限值，-H表示硬性极限值。硬性极限值是实际的限制，就是最大可以是100万，不能再多了。软性极限值则是系统发出警告（Warning）的极限值，超过这个极限值，内核会发出警告。

普通用户通过ulimit命令可将软性极限值更改到硬性极限值的最大设置值。如果要更改硬性极限值，必须拥有root用户权限。

要彻底解除Linux系统的最大文件打开数量的限制，可以通过编辑Linux的极限配置文件/etc/security/limits.conf来做到。修改此文件，加入如下内容：

```
soft    nofile  1000000  
hard    nofile  1000000
```

soft nofile表示软性极限，hard nofile表示硬性极限。

举个实际例子，在使用和安装目前非常流行的分布式搜索引擎ElasticSearch时，必须修改这个文件，以增加最大的文件描述符的极限值。当然，在生产环境运行Netty时，也需要修改/etc/security/limits.conf文件来增加文件描述符数量的极限值。

第3章 Java NIO核心详解

高性能的Java通信绝对离不开Java NIO组件，现在主流的技术框架或中间件服务器都使用了Java NIO组件，譬如Tomcat、Jetty、Netty。学习和掌握Java NIO组件已经不是一项加分技能，而是一项必备技能。

不管是面试还是实际开发，作为Java工程师，都必须掌握NIO的原理和开发实战技能。

3.1 Java NIO简介

在1.4版本之前，Java IO类库是阻塞IO；从1.4版本开始，引进了新的异步IO库，被称为Java New IO类库，简称为Java NIO。New IO类库的目标就是要让Java支持非阻塞IO，基于此，更多的人喜欢称Java NIO为非阻塞IO（Non-Blocking IO），称“老的”阻塞式Java IO为OIO（Old IO）。总体上说，NIO弥补了原来面向流的OIO同步阻塞的不足，为标准Java代码提供了高速、面向缓冲区的IO。

Java NIO类库包含以下三个核心组件：

- Channel（通道）
- Buffer（缓冲区）
- Selector（选择器）

理解了第2章的四种IO模型，大家一眼就能识别出来Java NIO属于第三种模型——IO多路复用模型。只不过，Java NIO组件提供了统一的API，为大家屏蔽了底层的操作系统的差异。

在后面的章节中，我们会对以上三个Java NIO的核心组件展开详细介绍。先来看看Java的NIO和OIO的简单对比。

3.1.1 NIO和OIO的对比

在Java中，NIO和OIO的区别主要体现在三个方面：

(1) OI0是面向流（Stream Oriented）的， NI0是面向缓冲区（Buffer Oriented）的。

在一般的OI0操作中，面向字节流或字符流的I0操作总是以流式的方式顺序地从一个流（Stream）中读取一个或多个字节，因此，我们不能随意改变读取指针的位置。在NI0操作中则不同， NI0中引入了Channel和Buffer的概念。面向缓冲区的读取和写入只需要从通道读取数据到缓冲区中，或将数据从缓冲区写入通道中。 NI0不像OI0那样是顺序操作，它可以随意读取Buffer中任意位置的数据。

(2) OI0的操作是阻塞的，而NI0的操作是非阻塞的。

OI0操作都是阻塞的。例如，我们调用一个read方法读取一个文件的内容，调用read的线程就会被阻塞，直到read操作完成。

在NI0模式中，当我们调用read方法时，如果此时有数据，则read读取数据并返回；如果此时没有数据，则read也会直接返回，而不会阻塞当前线程。

NI0的非阻塞是如何做到的呢？其实在上一章已经揭晓答案，即NI0使用了通道和通道的多路复用技术。

(3) OI0没有选择器（Selector）的概念，而NI0有选择器的概念。

NI0的实现是基于底层选择器的系统调用的，所以NI0需要底层操作系统提供支持；而OI0不需要用到选择器。

3.1.2 通道

在OIO中，同一个网络连接会关联到两个流：一个是输入流（Input Stream），另一个是输出流（Output Stream）。Java应用程序通过这两个流不断地进行输入和输出的操作。

在NIO中，一个网络连接使用一个通道表示，所有NIO的IO操作都是通过连接通道完成的。一个通道类似于OIO中两个流的结合体，既可以读取数据，也可以向通道写入数据。

3.1.3 选择器

首先回顾一下前面介绍的基础知识——I0多路复用指的是一个进程/线程可以同时监视多个文件描述符（含socket连接），一旦其中的一个或者多个文件描述符可读或者可写，该监听进程/线程就能够进行I0就绪事件的查询。

在Java应用层面，如何实现对多个文件描述符的监视呢？需要用到一个非常重要的Java NIO组件——选择器。选择器可以理解为一个I0事件的监听与查询器。通过选择器，一个线程可以查询多个通道的I0事件的就绪状态。

从编程实现维度来说，I0多路复用编程的第一步是把通道注册到选择器中，第二步是通过选择器所提供的事件查询（select）方法来查询这些注册的通道是否有已经就绪的I0事件（例如可读、可写、网络连接完成等）。

由于一个选择器只需要一个线程进行监控，因此我们可以很简单地使用一个线程，通过选择器去管理多个连接通道。

与OIO相比，NIO使用选择器的最大优势是系统开销小。系统不必为每一个网络连接（文件描述符）创建进程/线程，从而大大减少了系统的开销。总之，一个线程负责多个连接通道的I/O处理是非常高效的，这种高效来自Java的选择器组件Selector及其底层的操作系统I/O多路复用技术的支持。

3.1.4 缓冲区

应用程序与通道的交互主要是进行数据的读取和写入。为了完成NIO的非阻塞读写操作，NIO为大家准备了第三个重要的组件——Buffer。所谓通道的读取，就是将数据从通道读取到缓冲区中；所谓通道的写入，就是将数据从缓冲区写入通道中。缓冲区的使用是面向流进行读写操作的OIO所没有的，也是NIO非阻塞的重要前提和基础之一。

接下来笔者从缓冲区开始为大家详细介绍NIO的三大核心组件。

3.2 详解NIO Buffer类及其属性

NIO的Buffer本质上是一个内存块，既可以写入数据，也可以从中读取数据。Java NIO中代表缓冲区的Buffer类是一个抽象类，位于java.nio包中。

NIO的Buffer内部是一个内存块（数组），与普通的内存块（Java数组）不同的是：NIO Buffer对象提供了一组比较有效的方法，用来进行写入和读取的交替访问。

说明

Buffer类是一个非线程安全类。

3.2.1 Buffer类

Buffer类是一个抽象类，对应于Java的主要数据类型。在NIO中，有8种缓冲区类，分别是ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer、ShortBuffer、MappedByteBuffer。前7种Buffer类型覆盖了能在I0中传输的所有Java基本数据类型，第8种类型是一种专门用于内存映射的ByteBuffer类型。不同的Buffer子类可以操作的数据类型能够通过名称进行判断，比如IntBuffer只能操作Integer类型的对象。

实际上，使用最多的是ByteBuffer（二进制字节缓冲区）类型，后面的章节会看到它的具体使用。

3.2.2 Buffer类的重要属性

Buffer的子类会拥有一块内存，作为数据的读写缓冲区，但是读写缓冲区并没有定义在Buffer基类中，而是定义在具体的子类中。例如，ByteBuffer子类就拥有一个byte[]类型的数组成员final byte[] hb，可以作为自己的读写缓冲区，数组的元素类型与Buffer子类的操作类型相对应。

说明

在本书的上一个版本中，这里的内容为：Buffer内部有一个byte[]类型的数组作为数据的读写缓冲区。乍看上去没有什么错误，实际上那个结论是错误的。具体原因是作为读写缓冲区的数组，并没有定义在Buffer类中，而是定义在各具体子类中。感谢社群小伙伴@炬，是他发现了这个比较隐蔽的编写错误。

为了记录读写的状态和位置，Buffer类额外提供了一些重要的属性，其中有三个重要的成员属性：capacity（容量）、position（读写位置）和limit（读写的限制）。接下来对这三个成员属性进行比较详细的介绍。

1. capacity属性

Buffer类的capacity属性表示内部容量的大小。一旦写入的对象数量超过了capacity，缓冲区就满了，不能再写入了。

Buffer类的capacity属性一旦初始化，就不能再改变。原因是什么呢？Buffer类的对象在初始化时会按照capacity分配内部数组的内存，在数组内存分配好之后，它的大小就不能改变了。

前面讲到，Buffer类是一个抽象类，Java不能直接用来新建对象。在具体使用的时候，必须使用Buffer的某个子类，例如DoubleBuffer子类，该子类能写入的数据类型是double，如果在创建实例时其capacity是100，那么我们最多可以写入100个double类型的数据。

说明

capacity并不是指内部的内存块byte[]数组的字节数量，而是指能写入的数据对象的最大限制数量。

2. position属性

Buffer类的position属性表示当前的位置。position属性的值与缓冲区的读写模式有关。在不同的模式下，position属性值的含义是不同的，在缓冲区进行读写的模式改变时，position值会进行相应的调整。

在写模式下，position值的变化规则如下：

(1) 在刚进入写模式时，position值为0，表示当前的写入位置为从头开始。

(2) 每当一个数据写到缓冲区之后，position会向后移动到下一个可写的位置。

(3) 初始的position值为0，最大可写值为limit-1。当position值达到limit时，缓冲区就已经无空间可写了。

在读模式下，position值的变化规则如下：

(1) 当缓冲区刚开始进入读模式时，position会被重置为0。

(2) 当从缓冲区读取时，也是从position位置开始读。读取数据后，position向前移动到下一个可读的位置。

(3) 在读模式下，limit表示可读数据的上限。position的最大值为最大可读上限limit，当position达到limit时表明缓冲区已经无数据可读。

Buffer的读写模式具体如何切换呢？当新建了一个缓冲区实例时，缓冲区处于写模式，这时是可以写数据的。在数据写入完成后，如果要从缓冲区读取数据，就要进行模式的切换，可以调用flip()方法将缓冲区变成读模式，flip为翻转的意思。

在从写模式到读模式的翻转过程中，position和limit属性值会进行调整，具体的规则是：

(1) limit属性被设置成写模式时的position值，表示可以读取的最大数据位置。

(2) position由原来的写入位置变成新的可读位置，也就是0，表示可以从头开始读。

3. limit属性

Buffer类的limit属性表示可以写入或者读取的数据最大上限，其属性值的具体含义也与缓冲区的读写模式有关。在不同的模式下，limit值的含义是不同的，具体分为以下两种情况：

(1) 在写模式下，limit属性值的含义为可以写入的数据最大上限。在刚进入写模式时，limit的值会被设置成缓冲区的capacity值，表示可以一直将缓冲区的容量写满。

(2) 在读模式下，limit值的含义为最多能从缓冲区读取多少数据。

一般来说，在进行缓冲区操作时是先写入再读取的。当缓冲区写入完成后，就可以开始从Buffer读取数据，调用flip()方法（翻转），这时limit的值也会进行调整。具体如何调整呢？将写模式下的position值设置成读模式下的limit值，也就是说，将之前写入的最大数量作为可以读取数据的上限值。

Buffer在翻转时的属性值调整主要涉及position、limit两个属性，但是这种调整比较微妙，不是太好理解，下面举一个简单的例子：

首先，创建缓冲区。新创建的缓冲区处于写模式，其position值为0，limit值为最大容量capacity。

然后，向缓冲区写数据。每写入一个数据，position向后面移动一个位置，也就是position的值加1。这里假定写入了5个数，当写入完成后，position的值为5。

最后，使用flip方法将缓冲区切换到读模式。limit的值会先被设置成写模式时的position值，所以新的limit值是5，表示可以读取数据的最大上限是5。之后调整position值，新的position会被重置为0，表示可以从0开始读。

缓冲区切换到读模式后就可以从缓冲区读取数据了，一直到缓冲区的数据读取完毕。

除了以上capacity、position、limit三个重要的成员属性之外，Buffer还有一个比较重要的标记属性：mark（标记）属性。该属性的大致作用为：在缓冲区操作过程当中，可以将当前的position值临时存入mark属性中；需要的时候，再从mark中取出暂存的标记值，恢复到position属性中，重新从position位置开始处理。

下面用表3-1总结一下Buffer类的四个重要属性。

表3-1 Buffer类的四个重要属性说明

属性	说 明
capacity	容量，即可以容纳的最大数据量，在缓冲区创建时设置并且不能改变
limit	读写的限制，缓冲区中当前的数据量
position	读写位置，缓冲区中下一个要被读或写的元素的索引
mark	调用 mark()方法来设置 mark=position，再调用 reset()让 position 恢复到 mark 标记的位置，即 position=mark

3.3 详解NIO Buffer类的重要方法

本节将详细介绍Buffer类的几个常用方法，包含Buffer实例的创建、写入、读取、重复读、标记和重置等。

3.3.1 allocate()

在使用Buffer实例之前，我们首先需要获取Buffer子类的实例对象，并且分配内存空间。需要获取一个Buffer实例对象时，并不是使用子类的构造器来创建，而是调用子类的allocate()方法。

下面的程序片段演示如何获取一个整型的Buffer实例对象：

```
package com.crazymakercircle.bufferDemo;  
import com.crazymakercircle.util.Logger;  
import java.nio.IntBuffer;  
  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    public static void allocateTest()  
    {  
        //创建一个intBuffer实例对象  
        intBuffer = IntBuffer.allocate(20);  
        Logger.debug("-----after allocate-----");  
    }  
}
```

```
-----");
        Logger.debug("position=" +
intBuffer.position());
        Logger.debug("limit=" + intBuffer.limit());
        Logger.debug("capacity=" +
intBuffer.capacity());
    }
    //省略其他代码
}
```

本例中，IntBuffer是具体的Buffer子类，通过调用IntBuffer.allocate(20)创建了一个intBuffer实例对象，并且分配了 20×4 字节的内存空间。运行程序之后，通过程序的输出结果，我们可以查看一个新建缓冲区实例对象的主要属性值，如下所示：

```
allocatTest |> -----after allocate-----
allocatTest |> position=0
allocatTest |> limit=20
allocatTest |> capacity=20
```

从上面的运行结果可以看出：一个缓冲区在新建后处于写模式，position属性（代表写入位置）的值为0，缓冲区的capacity值是初始化时allocate方法的参数值（这里是20），而limit最大可写上限值也为allocate方法的初始化参数值。

3.3.2 put()

在调用allocate()方法分配内存、返回了实例对象后，缓冲区实例对象处于写模式，可以写入对象，如果要把对象写入缓冲区，就需要调用put()方法。put()方法很简单，只有一个参数，即需要写入的对象，只不过要求写入的数据类型与缓冲区的类型保持一致。

接着前面的例子向刚刚创建的intBuffer缓存实例对象写入5个整数，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import  
  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    //省略了创建缓冲区的代码，具体查看前面小节的内容和随书源码  
    public static void putTest()  
    {  
        for (int i = 0; i < 5; i++)  
        {  
            //写入一个整数到缓冲区  
            intBuffer.put(i);  
        }  
  
        //输出缓冲区的主要属性值  
        Logger.debug("-----after putTest-----  
--");  
        Logger.debug("position=" + intBuffer.position());
```

```
    Logger.debug("limit=" + intBuffer.limit());
    Logger.debug("capacity=" + intBuffer.capacity());
}
//省略其他代码
}
```

写入5个元素后，同样输出缓冲区的主要属性值，输出的结果如下：

```
putTest |> -----after putTest-----
putTest |> position=5
putTest |> limit=20
putTest |> capacity=20
```

从结果可以看到，写入了5个元素之后，缓冲区的position属性值变成了5，所以指向了第6个（从0开始的）可以进行写入的元素位置。limit最大可写上限、capacity最大容量两个属性的值都没有发生变化。

3.3.3 flip()

向缓冲区写入数据之后，是否可以直接从缓冲区读取数据呢？不能！这时缓冲区还处于写模式，如果需要读取数据，要将缓冲区转换成读模式。flip()翻转方法是Buffer类提供的一个模式转变的重要方法，作用是将写模式翻转成读模式。

接着前面的例子演示一下flip()方法的使用：

```
package com.crazymakercircle.bufferDemo;  
//省略import  
  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的创建、写入数据的代码，具体查看前面小节的内容和随书  
源码  
  
    public static void flipTest()  
    {  
        //翻转缓冲区，从写模式翻转成读模式  
        intBuffer.flip();  
        //输出缓冲区的主要属性值  
        Logger.info("-----after flip -----");  
        Logger.info("position=" +  
intBuffer.position());  
        Logger.info("limit=" +  
intBuffer.limit());  
        Logger.info("capacity=" +  
intBuffer.capacity());  
    }  
    //省略其他代码  
}
```

在调用`flip()`方法进行缓冲区的模式翻转之后，通过程序的输出内容可以看到缓冲区的属性有了奇妙的变化，具体如下：

```
flipTest |> -----after flipTest -----
flipTest |> position=0
flipTest |> limit=5
flipTest |> capacity=20
```

调用`flip()`方法后，新模式下可读上限`limit`的值变成了之前写模式下的`position`属性值，也就是5；而新的读模式下的`position`值简单粗暴地变成了0，表示从头开始读取。

对`flip()`方法从写入到读取转换的规则，再一次详细介绍如下：

首先，设置可读上限`limit`的属性值。将写模式下的缓冲区中内容的最后写入位置`position`值作为读模式下的`limit`上限值。

其次，把读的起始位置`position`的值设为0，表示从头开始读。

最后，清除之前的`mark`标记，因为`mark`保存的是写模式下的临时位置，发生模式翻转后，如果继续使用旧的`mark`标记，就会造成位置混乱。

上面三步其实可以查看`Buffer.flip()`方法的源代码，具体如下：

```
public final Buffer flip() {
    limit = position;      //设置可读上限limit，设置为写模式下的
    position = 0;          //把读的起始位置position的值设为0，表示从头开
    mark = UNSET_MARK;     //清除之前的mark标记
}
```

```
        return this;  
    }  
  
    new Problem("缓冲区读写模式的转换") {  
        @Override  
        public void solve() {  
            // 1. 创建一个写模式的缓冲区  
            ByteBuffer buffer = ByteBuffer.allocate(10);  
            // 2. 将缓冲区切换为读模式  
            buffer.flip();  
            // 3. 从缓冲区读取数据  
            int value = buffer.get();  
            System.out.println("读取到的数据是：" + value);  
        }  
    }.execute();  
}
```

新的问题来了：在读取完成后，如何再一次将缓冲区切换成写模式呢？答案是：可以调用Buffer.clear()清空或者Buffer.compact()压缩方法，它们可以将缓冲区转换为写模式。总体的Buffer模式转换大致如图3-1所示。

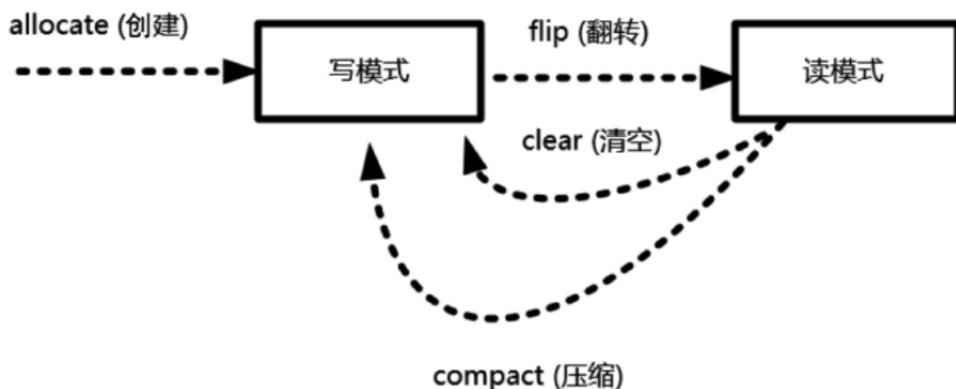


图3-1 缓冲区读写模式的转换

3.3.4 get()

调用flip()方法将缓冲区切换成读模式之后，就可以开始从缓冲区读取数据了。读取数据的方法很简单，可以调用get()方法每次从position的位置读取一个数据，并且进行相应的缓冲区属性的调整。

接着前面调用flip()方法的实例，演示一下缓冲区的读取操作，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import
```

```
public class UseBuffer
{
    //一个整型的Buffer静态变量
    static IntBuffer intBuffer = null;

    //省略了缓冲区的创建、写入、翻转的代码，具体查看前面小节的内容和随
    书源码
```

```
public static void getTest()
{
    //先读2个数据
    for (int i = 0; i < 2; i++)
    {
        int j = intBuffer.get();
        Logger.info("j = " + j);
    }

    //输出缓冲区的主要属性值
    Logger.info("-----after get 2 int -----");
    Logger.info("position=" +
    intBuffer.position());
    Logger.info("limit=" + intBuffer.limit());
    Logger.info("capacity=" +
    intBuffer.capacity());

    //再读3个数据
    for (int i = 0; i < 3; i++)
```

```

    {
        int j = intBuffer.get();
        Logger.info("j = " + j);
    }
    //输出缓冲区的主要属性值
    Logger.info("-----after get 3 int -----");
    Logger.info("position=" +
intBuffer.position());
    Logger.info("limit=" + intBuffer.limit());
    Logger.info("capacity=" +
intBuffer.capacity());
}
//...
}

//省略其他代码
}

```

以上代码调用get方法从缓冲实例中先读取2个元素，再读取3个元素，运行后输出的结果如下：

```

getTest |> -----after get 2 int -----
getTest |> position=2
getTest |> limit=5
getTest |> capacity=20
getTest |> -----after get 3 int -----

```

```
getTest |> position=5  
getTest |> limit=5  
getTest |> capacity=20
```

从程序的输出结果可以看到，读取操作会改变可读位置position的属性值，而可读上限limit值并不会改变。在position值和limit值相等时，表示所有数据读取完成，position指向了一个没有数据的元素位置，已经不能再读了，此时再读就会抛出BufferUnderflowException异常。

这里强调一下，在读完之后是否可以立即对缓冲区进行数据写入呢？答案是不能。现在还处于读模式，我们必须调用Buffer.clear()或Buffer.compact()方法，即清空或者压缩缓冲区，将缓冲区切换成写模式，让其重新可写。

此外还有一个问题：缓冲区是不是可以重复读呢？答案是可以的，既可以通过倒带方法rewind()去完成，也可以通过mark()和reset()两个方法组合实现。

3.3.5 rewind()

已经读完的数据，如果需要再读一遍，可以调用rewind()方法。rewind()也叫倒带，就像播放磁带一样倒回去，再重新播放。

接着前面的示例代码，继续rewind方法使用的演示，示例代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import  
  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的写入和读取等代码，具体查看前面小节的内容和随书源码  
    public static void rewindTest() {  
        //倒带  
        intBuffer.rewind();  
        //输出缓冲区属性  
        Logger.info("-----after rewind -----  
-----");  
        Logger.info("position=" +  
intBuffer.position());  
        Logger.info("limit=" + intBuffer.limit());  
        Logger.info("capacity=" +  
intBuffer.capacity());  
    }  
  
    //省略其他代码  
}
```

这个范例程序的执行结果如下：

```
rewindTest |> -----after rewind -----  
rewindTest |> position=0
```

```
rewindTest |> limit=5  
rewindTest |> capacity=20
```

rewind ()方法主要是调整了缓冲区的position属性与mark属性，具体的调整规则如下：

- (1) position重置为0，所以可以重读缓冲区中的所有数据。
- (2) limit保持不变，数据量还是一样的，仍然表示能从缓冲区中读取的元素数量。
- (3) mark被清理，表示之前的临时位置不能再用了。

从JDK中可以查阅Buffer. rewind()方法的源代码，具体如下：

```
public final Buffer rewind() {  
    position = 0; //重置为0，所以可以重读缓冲区中的所有数据  
    mark = -1; //mark被清理，表示之前的临时位置不能再用了  
    return this;  
}
```

通过源代码，我们可以看到rewind()方法与flip()方法很相似，区别在于：倒带方法rewind()不会影响limit属性值；而翻转方法flip()会重设limit属性值。

在rewind()倒带之后，就可以再一次读取，重复读取的示例代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import
```

```
public class UseBuffer
{
    //一个整型的Buffer静态变量
    static IntBuffer intBuffer = null;
    //省略了缓冲区的写入和读取、倒带等代码，具体查看前面小节的内容和随
    书源码
```

```
public static void reRead() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) {
            //临时保存，标记一下第3个位置
            intBuffer.mark();
        }
        //读取元素
        int j = intBuffer.get();
        Logger.info("j = " + j);
    }
    //输出缓冲区的属性值
    Logger.info("-----after reRead-----");
    Logger.info("position=" +
    intBuffer.position());
    Logger.info("limit=" + intBuffer.limit());
    Logger.info("capacity=" +
    intBuffer.capacity());
}
```

```
//省略其他代码  
}
```

这段代码与前面的读取示例代码基本相同，只是增加了一个mark调用。大家可以通过随书源码工程执行以上代码并观察输出结果，具体的输出与前面的类似，这里不再赘述。

3.3.6 mark()和reset()

mark()和reset()两个方法是配套使用的：Buffer.mark()方法将当前position的值保存起来放在mark属性中，让mark属性记住这个临时位置；然后可以调用Buffer.reset()方法将mark的值恢复到position中。

说明

Buffer.mark()和Buffer.reset()两个方法都涉及mark属性的使用。mark()方法与mark属性的名字虽然相同，但是一个是Buffer类的成员方法，一个是Buffer类的成员属性，不能混淆。

例如，在前面重复读取的示例代码中，在读到第三个元素（i为2时）时，可以调用mark()方法，把当前位置position的值保存到mark属性中，这时mark属性的值为2。

接下来可以调用reset()方法将mark属性的值恢复到position中，这样就可以从位置2（第三个元素）开始重复读取了。

接着前面重复读取的代码，进行mark()方法和reset()方法的示例演示，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的倒带、重复读取等代码，具体查看前面小节的内容和随书  
源码
```

```
//演示前提：  
//在前面的reRead()演示方法中，已经通过mark()方法暂存了  
position值
```

```
public static void afterReset() {  
    Logger.info("-----after reset-----  
-----");  
    //把前面保存在mark中的值恢复到position中  
    intBuffer.reset();  
    //输出缓冲区的属性值  
    Logger.info("position=" +  
    intBuffer.position());
```

```
        Logger.info("limit=" + intBuffer.limit());
        Logger.info("capacity=" +
intBuffer.capacity());
        //读取并且输出元素
        for (int i = 2; i < 5; i++) {
            int j = intBuffer.get();
            Logger.info("j = " + j);
        }
    }
    //省略其他代码
}
```

在上面的代码中，首先调用reset()把mark中的值恢复到position中，因此读取的位置position就是2，表示可以再次开始从第三个元素开始读取数据。上面的程序代码的输出结果是：

```
afterReset |> -----
afterReset |> position=2
afterReset |> limit=5
afterReset |> capacity=20
afterReset |> j = 2
afterReset |> j = 3
afterReset |> j = 4
```

调用reset()方法之后，position的值为2，此时去读取缓冲区，输出后面的三个元素2、3、4。

3.3.7 clear()

在读模式下，调用clear()方法将缓冲区切换为写模式。此方法的作用是：

- (1) 将position清零。
- (2) limit设置为capacity最大容量值，可以一直写入，直到缓冲区写满。

接着上面的实例演示一下clear()方法的使用，大致的代码如下：

```
package com.crazymakercircle.bufferDemo;  
//省略import  
public class UseBuffer  
{  
    //一个整型的Buffer静态变量  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的创建、写入、读取等代码，具体查看前面小节的内容和随书  
    源码  
  
    public static void clearDemo() {  
        Logger.info("-----after clear-----  
");  
        //清空缓冲区，进入写模式  
        intBuffer.clear();  
        //输出缓冲区的属性值  
        Logger.info("position=" + intBuffer.position());
```

```
    Logger.info("limit=" + intBuffer.limit());  
    Logger.info("capacity=" + intBuffer.capacity());  
}  
  
//省略其他代码  
}
```

这个示例程序运行之后，结果如下：

```
main |>清空  
clearDemo |> -----after clear-----  
clearDemo |> position=0  
clearDemo |> limit=20  
clearDemo |> capacity=20
```

在缓冲区处于读模式时，调用clear()，缓冲区会被切换成写模式。调用clear()之后，我们可以看到清空了position（写入的起始位置）的值，其值被设置为0，并且limit值（写入的上限）为最大容量。

3.3.8 使用Buffer类的基本步骤

总体来说，使用Java NIO Buffer类的基本步骤如下：

- (1) 使用创建子类实例对象的allocate()方法创建一个Buffer类的实例对象。
- (2) 调用put()方法将数据写入缓冲区中。

(3) 写入完成后，在开始读取数据前调用Buffer.flip()方法，将缓冲区转换为读模式。

(4) 调用get()方法，可以从缓冲区中读取数据。

(5) 读取完成后，调用Buffer.clear()方法或Buffer.compact()方法，将缓冲区转换为写模式，可以继续写入。

3.4 详解NIO Channel类

前面提到，Java NIO中一个socket连接使用一个Channel来表示。从更广泛的层面来说，一个通道可以表示一个底层的文件描述符，例如硬件设备、文件、网络连接等。然而，远不止如此，Java NIO的通道可以更加细化。例如，不同的网络传输协议类型，在Java中都有不同的NIO Channel实现。

这里不对Java NIO的全部通道类型进行过多的描述，仅着重介绍其中最为重要的四种Channel实现：FileChannel、SocketChannel、ServerSocketChannel、DatagramChannel。

对于以上四种通道，说明如下：

- (1) FileChannel：文件通道，用于文件的数据读写。
- (2) SocketChannel：套接字通道，用于套接字TCP连接的数据读写。
- (3) ServerSocketChannel：服务器套接字通道（或服务器监听通道），允许我们监听TCP连接请求，为每个监听到的请求创建一个SocketChannel通道。
- (4) DatagramChannel：数据报通道，用于UDP的数据读写。

这四种通道涵盖了文件I/O、TCP网络、UDP I/O三类基础I/O读写操作。下面从通道的获取、读取、写入、关闭这四个重要的操作入手，对它们进行简单的介绍。

3.4.1 FileChannel

FileChannel（文件通道）是专门操作文件的通道。通过FileChannel，既可以从一个文件中读取数据，也可以将数据写入文件中。特别申明一下，FileChannel为阻塞模式，不能设置为非阻塞模式。

下面分别介绍FileChannel的获取、读取、写入、关闭这四个操作。

1. 获取FileChannel

可以通过文件的输入流、输出流获取FileChannel，示例如下：

```
//创建一个文件输入流  
FileInputStream fis = new FileInputStream(srcFile);  
//获取文件流的通道  
FileChannel inChannel = fis.getChannel();  
  
//创建一个文件输出流  
FileOutputStream fos = new FileOutputStream(destFile);  
//获取文件流的通道  
FileChannel outchannel = fos.getChannel();
```

也可以通过RandomAccessFile（文件随机访问）类来获取FileChannel实例，代码如下：

```
//创建RandomAccessFile随机访问对象  
RandomAccessFile rFile = new
```

```
RandomAccessFile("filename.txt", "rw");  
//获取文件流的通道（可读可写）  
FileChannel channel = rFile.getChannel();
```

2. 读取FileChannel

在大部分应用场景中，从通道读取数据都会调用通道的int `read(ByteBuffer buf)` 方法，它把从通道读取的数据写入 `ByteBuffer` 缓冲区，并且返回读取的数据量。

```
RandomAccessFileaFile = new RandomAccessFile(fileName, "rw");  
//获取通道（可读可写）  
FileChannel channel=aFile.getChannel();  
//获取一个字节缓冲区  
ByteBuffer buf = ByteBuffer.allocate(CAPACITY);  
int length = -1;  
//调用通道的read()方法，读取数据并写入字节类型的缓冲区  
while ((length = channel.read(buf)) != -1) {  
//省略buf中的数据处理  
}
```

说明

以上代码中 `channel.read(buf)` 读取通道的数据时，对于通道来说是读模式，对于 `ByteBuffer` 缓冲区来说是写入数据，这时 `ByteBuffer` 缓冲区处于写模式。

3. 写入FileChannel

把数据写入通道，在大部分应用场景中都会调用通道的 write(ByteBuffer) 方法，此方法的参数是一个 ByteBuffer 缓冲区实例，是待写数据的来源。

write(ByteBuffer) 方法的作用是从 ByteBuffer 缓冲区中读取数据，然后写入通道自身，而返回值是写入成功的字节数。

```
//如果buf处于写模式（如刚写完数据），需要翻转buf，使其变成读模式  
buf.flip();  
  
int outlength = 0;  
  
//调用write()方法，将buf的数据写入通道  
  
while ((outlength = outchannel.write(buf)) != 0) {  
    System.out.println("写入的字节数：" + outlength);  
}
```

在以上的 outchannel.write(buf) 调用中，对于入参 buf 实例来说，需要从其中读取数据写入 outchannel 通道中，所以入参 buf 必须处于读模式，不能处于写模式。

4. 关闭通道

当通道使用完成后，必须将其关闭。关闭非常简单，调用 close() 方法即可。

```
//关闭通道  
channel.close();
```

5. 强制刷新到磁盘

在将缓冲区写入通道时，出于性能的原因，操作系统不可能每次都实时地将写入数据落地（或刷新）到磁盘，完成最终的数据保存。

在将缓冲区数据写入通道时，要保证数据能写入磁盘，可以在写入后调用一下FileChannel的force()方法。

```
//强制刷新到磁盘  
channel.force(true);
```

3.4.2 使用FileChannel完成文件复制的实战案例

下面是一个简单的实战案例：使用FileChannel复制文件。具体的功能是使用FileChannel将原文件复制一份，把原文件中的数据都复制到目标文件中。完整代码如下：

```
package com.crazymakercircle.iodemo.fileDemos;  
  
//省略import，具体请参见源代码工程  
  
public class FileNIOCopyDemo {  
  
    public static void main(String[] args) {  
  
        //演示复制资源文件  
  
       nioCopyResouceFile();  
  
    }  
  
    /**  
     * 复制两个资源目录下的文件  
     */  
  
    public static void nioCopyResouceFile() {  
  
        //源
```

```

        String sourcePath =
NioDemoConfig.FILE_RESOURCE_SRC_PATH;
        String srcPath = IOUtil.getResourcePath(sourcePath);
        Logger.info("srcPath=" + srcPath);

        //目标
        String destPath =
NioDemoConfig.FILE_RESOURCE_DEST_PATH;
        String destDecodePath =
IOUtil.builderResourcePath(destPath);
        Logger.info("destDecodePath=" + destDecodePath);

        //复制文件
       nioCopyFile(srcDecodePath, destDecodePath);
    }

    /**
     * NIO方式复制文件
     * @param srcPath 源路径
     * @param destPath 目标路径
     */
    public static void nioCopyFile(String srcPath, String
destPath) {
        File srcFile = new File(srcPath);
        File destFile = new File(destPath);
        try {
            //如果目标文件不存在，则新建
            if (!destFile.exists()) {

```

```
        destFile.createNewFile();

    }

    long startTime = System.currentTimeMillis();

    FileInputStream fis = null;

    FileOutputStream fos = null;

    FileChannel inChannel = null; //输入通道

    FileChannel outchannel = null; //输出通道

    try {

        fis = new FileInputStream(srcFile);

        fos = new FileOutputStream(destFile);

        inChannel = fis.getChannel();

        outchannel = fos.getChannel();

        int length = -1;

        //新建buf，处于写模式

        ByteBufferbuf = ByteBuffer.allocate(1024);

        //从输入通道读取到buf

        while ((length = inChannel.read(buf)) != -1) {

            //buf第一次模式切换：翻转buf，从写模式变成读模式

            buf.flip();

            int outlength = 0;

            //将buf写入输出的通道

            while ((outlength = outchannel.write(buf)) !=

0) {

                System.out.println("写入的字节数：" +

outlength);

            }

            //buf第二次模式切换：清除buf，变成写模式
```

```
        buf.clear();

    }

    //强制刷新到磁盘

    outchannel.force(true);

} finally {

    //关闭所有的可关闭对象

    IOUtil.closeQuietly(outchannel);

    IOUtil.closeQuietly(fos);

    IOUtil.closeQuietly(inChannel);

    IOUtil.closeQuietly(fis);

}

long endTime = System.currentTimeMillis();

Logger.info("base复制毫秒数: " + (endTime - startTime));

} catch (IOException e) {

    e.printStackTrace();

}

}
```

除了FileChannel的通道操作外，还需要注意代码执行过程中隐藏的ByteBuffer的模式切换。新建的ByteBuffer在写模式时才可作为inChannel.read(ByteBuffer)方法的参数，inChannel.read()方法将从通道inChannel读到的数据写入ByteBuffer。然后，调用缓冲区的flip方法，将ByteBuffer从写模式切换成读模式才能作为outchannel.write(ByteBuffer)方法的参数，以便从ByteBuffer读取数据，最终写入outchannel（输出通道）。

完成一次复制之后，在进入下一次复制前还要进行一次缓冲区的模式切换。此时，需要通过clear方法将Buffer切换成写模式才能进入下一次的复制。所以，在示例代码中，每一轮外层的while循环都需要两次ByteBuffer模式切换：第一次模式切换时翻转buf，变成读模式；第二次模式切换时清除buf，变成写模式。

上面示例代码的主要目的在于演示文件通道以及字节缓冲区的使用。作为文件复制的程序来说，以上实战代码的效率不是最高的。更高效的文件复制可以调用文件通道的transferFrom()方法。具体的代码可以参见源代码工程中的FileNIOFastCopyDemo类，完整源文件的路径为

com.crazymakercircle.iodemo.fileDemos.FileNIOFastCopyDemo。

请大家在随书源码工程中自行运行和学习以上代码，这里不再赘述。

3.4.3 SocketChannel

在NIO中，涉及网络连接的通道有两个：一个是SocketChannel，负责连接的数据传输；另一个是ServerSocketChannel，负责连接的监听。其中，NIO中的SocketChannel传输通道与OIO中的Socket类对应，NIO中的ServerSocketChannel监听通道对应于OIO中的ServerSocket类。

ServerSocketChannel仅应用于服务端，而SocketChannel同时处于服务端和客户端。所以，对于一个连接，两端都有一个负责传输的SocketChannel。

无论是ServerSocketChannel还是SocketChannel，都支持阻塞和非阻塞两种模式。如何进行模式的设置呢？调用configureBlocking()方法，具体如下：

(1) socketChannel.configureBlocking(false) 设置为非阻塞模式。

(2) socketChannel.configureBlocking(true) 设置为阻塞模式。

在阻塞模式下，SocketChannel的连接、读、写操作都是同步阻塞式的，在效率上与Java IO面向流的阻塞式读写操作相同。因此，在这里不介绍阻塞模式下通道的具体操作。在非阻塞模式下，通道的操作是异步、高效的，这也是相对于传统IO的优势所在。下面详细介绍在非阻塞模式下通道的获取、读写和关闭等操作。

1. 获取SocketChannel传输通道

在客户端，先通过SocketChannel静态方法open()获得一个套接字传输通道，然后将socket设置为非阻塞模式，最后通过connect()实例方法对服务器的IP和端口发起连接。

```
//获得一个套接字传输通道
SocketChannel socketChannel = SocketChannel.open();
//设置为非阻塞模式
socketChannel.configureBlocking(false);
//对服务器的IP和端口发起连接
socketChannel.connect(new InetSocketAddress("127.0.0.1", 80));
```

在非阻塞情况下，与服务器的连接可能还没有真正建立，socketChannel.connect()方法就返回了，因此需要不断地自旋，检查当前是否连接到了主机：

```
while(! socketChannel.finishConnect()) {  
    //不断地自旋、等待，或者做一些其他的事情  
}
```

在服务端，如何获取与客户端对应的传输套接字呢？

在连接建立的事件到来时，服务端的ServerSocketChannel能成功地查询出这个新连接事件，并且通过调用服务端ServerSocketChannel监听套接字的accept()方法来获取新连接的套接字通道：

```
//新连接事件到来，首先通过事件获取服务器监听通道  
ServerSocketChannel server = (ServerSocketChannel)  
key.channel();  
//获取新连接的套接字通道  
SocketChannel socketChannel = server.accept();  
//设置为非阻塞模式  
socketChannel.configureBlocking(false);
```

说明

NIO套接字通道主要用于非阻塞的传输场景。所以，基本上都需要调用通道的configureBlocking(false)方法，将通道从阻塞模式切换为非阻塞模式。

2. 读取SocketChannel传输通道

当SocketChannel传输通道可读时，可以从SocketChannel读取数据，具体方法与前面的文件通道读取方法是相同的。调用read()方法，将数据读入缓冲区ByteBuffer。

```
ByteBufferbuf = ByteBuffer.allocate(1024);  
int bytesRead = socketChannel.read(buf);
```

在读取时，因为是异步的，所以我们必须检查read()的返回值，以便判断当前是否读取到了数据。read()方法的返回值是读取的字节数，如果是-1，那么表示读取到对方的输出结束标志，即对方已经输出结束，准备关闭连接。实际上，通过read()方法读数据本身是很简单的，比较困难的是在非阻塞模式下如何知道通道何时是可读的。这需要用到NIO的新组件——Selector通道选择器，稍后介绍它。

3. 写入SocketChannel传输通道

和前面把数据写入FileChannel一样，大部分应用场景都会调用通道的int write(ByteBufferbuf)方法。

```
//写入前需要读取缓冲区，要求ByteBuffer是读模式  
buffer.flip();  
socketChannel.write(buffer);
```

4. 关闭SocketChannel传输通道

在关闭SocketChannel传输通道前，如果传输通道用来写入数据，则建议调用一次shutdownOutput()终止输出方法，向对方发送一个输出的结束标志（-1）。然后调用socketChannel.close()方法，关闭套接字连接。

```
//调用终止输出方法，向对方发送一个输出的结束标志  
socketChannel.shutdownOutput();  
//关闭套接字连接  
IOUtil.closeQuietly(socketChannel);
```

3.4.4 使用SocketChannel发送文件的实战案例

下面的实战案例是使用FileChannel读取本地文件内容，然后在客户端使用SocketChannel把文件信息和文件内容发送到服务器。客户端的完整代码如下：

```
package com.crazymakercircle.iodemo.socketDemos;  
//...  
public class NioSendClient {  
    private Charset charset = Charset.forName("UTF-8");  
    /**  
     * 向服务端传输文件  
     */  
    public void sendFile()  
    {  
        try  
        {
```

```
String sourcePath = NioDemoConfig.SOCKET_SEND_FILE;

String srcPath =
IOUtil.getResourcePath(sourcePath);

Logger.debug("srcPath=" + srcPath);

String destFile =
NioDemoConfig.SOCKET_RECEIVE_FILE;

Logger.debug("destFile=" + destFile);

File file = new File(srcPath);

if (!file.exists())

{

    Logger.debug("文件不存在");

    return;

}

FileChannel fileChannel =
new FileInputStream(file).getChannel();

SocketChannel socketChannel =
SocketChannel.open();

socketChannel.socket().connect(
    new InetSocketAddress("127.0.0.1",18899));
socketChannel.configureBlocking(false);
Logger.debug("Client 成功连接服务端");

while (!socketChannel.finishConnect())
{
```

```
//不断地自旋、等待，或者做一些其他的事情
}

//发送文件名称和长度
ByteBuffer buffer =
    sendFileNameAndLength(destFile,
file, socketChannel);

//发送文件内容
int length =
    sendContent(file, fileChannel,
socketChannel, buffer);

if (length == -1)
{
    IOUtil.closeQuietly(fileChannel);
    socketChannel.shutdownOutput();
    IOUtil.closeQuietly(socketChannel);
}

Logger.debug("===== 文件传输成功 =====");

} catch (Exception e)
{
    e.printStackTrace();
}

}

//方法：发送文件内容
public int sendContent(File file, FileChannel fileChannel,
```

```
        SocketChannel socketChannel,
        ByteBuffer buffer) throws
IOException
{
    //发送文件内容
    Logger.debug("开始传输文件");
    int length = 0;
    long progress = 0;
    while ((length = fileChannel.read(buffer)) > 0)
    {
        buffer.flip();
        socketChannel.write(buffer);
        buffer.clear();
        progress += length;
        Logger.debug("| " + (100 * progress / file.length()) + "%");
    }
    return length;
}

//方法：发送文件名称和长度
public ByteBuffer sendFileNameAndLength(String destFile,
                                         File
file,
                                         SocketChannel
socketChannel) throws IOException
{
```

```
//发送文件名称
    ByteBuffer fileNameByteBuffer =
charset.encode(destFile);

    ByteBuffer buffer =
ByteBuffer.allocate(NioDemoConfig.SEND_BUFFER_SIZE);
//发送文件名称长度
    int fileNameLen = fileNameByteBuffer.capacity();
    buffer.putInt(fileNameLen);
    buffer.flip();
    socketChannel.write(buffer);
    buffer.clear();
    Logger.info("Client 文件名称长度发送完成:", fileNameLen);

//发送文件名称
    socketChannel.write(fileNameByteBuffer);
    Logger.info("Client 文件名称发送完成:", destFile);
//发送文件长度
    buffer.putLong(file.length());
    buffer.flip();
    socketChannel.write(buffer);
    buffer.clear();
    Logger.info("Client 文件长度发送完成:", file.length());
    return buffer;
}
```

以上代码中，文件发送过程是：首先发送文件名称（不带路径）和文件长度，然后发送文件内容。代码中的配置项（如服务器的IP、服务端口、待发送的源文件名称（带路径）、远程的目标文件名称等配置信息）都是从system.properties配置文件中读取的，通过自定义的NioDemoConfig配置类来完成配置。

在运行以上客户端的程序之前，需要先运行服务端的程序。服务端的类与客户端的源代码在同一个包下，类名为NioReceiveServer，具体参见源代码工程，我们稍后再详细介绍这个类。

3.4.5 DatagramChannel

在Java中使用UDP传输数据比TCP更加简单。和socket的TCP不同，UDP不是面向连接的协议。使用UDP时，只要知道服务器的IP和端口就可以直接向对方发送数据。在Java NIO中，使用DatagramChannel来处理UDP的数据传输。

1. 获取DatagramChannel

获取数据报通道的方式很简单，调用DatagramChannel类的open()静态方法即可。然后调用configureBlocking(false)方法，设置成非阻塞模式。

```
//获取DatagramChannel  
DatagramChannel channel = DatagramChannel.open();  
//设置为非阻塞模式  
datagramChannel.configureBlocking(false);
```

如果需要接收数据，还需要调用bind()方法绑定一个数据报的监听端口，具体如下：

```
//调用bind()方法绑定一个数据报的监听端口  
channel.socket().bind(new InetSocketAddress(18080));
```

2. 从DatagramChannel读取数据

当DatagramChannel通道可读时，可以从DatagramChannel读取数据。和前面的SocketChannel读取方式不同，这里不调用read()方法，而是调用receive(ByteBufferbuf)方法将数据从DatagramChannel读入，再写入ByteBuffer缓冲区中。

```
//创建缓冲区  
ByteBuffer buf = ByteBuffer.allocate(1024);  
//从DatagramChannel读入，再写入ByteBuffer缓冲区  
SocketAddress clientAddr= datagramChannel.receive(buf);
```

通道读取receive(ByteBufferbuf)方法虽然读取了数据到buf缓冲区，但是其返回值是SocketAddress类型，表示返回发送端的连接地址（包括IP和端口）。通过receive方法读取数据非常简单，但是在非阻塞模式下如何知道DatagramChannel通道何时是可读的呢？和SocketChannel一样，同样需要用到NIO的新组件——Selector通道选择器。

3. 写入DatagramChannel

向DatagramChannel发送数据，和向SocketChannel通道发送数据的方法是不同的。这里不是调用write()方法，而是调用send()方法。

示例代码如下：

```
//把缓冲区翻转为读模式  
buffer.flip();  
//调用send()方法，把数据发送到目标IP+端口  
dChannel.send(buffer, new  
InetSocketAddress("127.0.0.1",18899));  
//清空缓冲区，切换到写模式  
buffer.clear();
```

由于UDP是面向非连接的协议，因此在调用send()方法发送数据时需要指定接收方的地址（IP和端口）。

4. 关闭DatagramChannel

这个比较简单，直接调用close()方法即可关闭数据报通道。

```
//简单关闭即可  
dChannel.close();
```

3.4.6 使用DatagramChannel发送数据的实战案例

下面是一个使用DatagramChannel发送数据的客户端示例程序，功能是获取用户的输入数据，通过DatagramChannel将数据发送到远程的服务器。客户端的完整程序代码如下：

```
package com.crazymakercircle.iodemo.udpDemos;  
//...  
public class UDPClient {
```

```
public void send() throws IOException {
    //获取DatagramChannel
    DatagramChannel dChannel = DatagramChannel.open();
    //设置为非阻塞
    dChannel.configureBlocking(false);
    ByteBuffer buffer =
        ByteBuffer.allocate(NioDemoConfig.SEND_BUFFER_SIZE);

    Scanner scanner = new Scanner(System.in);
    Print.tcf("UDP客户端启动成功！");
    Print.tcf("请输入发送内容:");
    while (scanner.hasNext()) {
        String next = scanner.next();
        buffer.put((Dateutil.getNow() + " >> " +
next).getBytes());
        buffer.flip();
        //通过DatagramChannel发送数据
        dChannel.send(buffer, new
InetSocketAddress("127.0.0.1",18899));
        buffer.clear();
    }
    //关闭DatagramChannel
    dChannel.close();
}

public static void main(String[] args) throws IOException {
    new UDPClient().send();
}
```

```
    }  
}
```

从示例程序可以看出，在客户端使用DatagramChannel发送数据比在客户端使用SocketChannel发送数据要简单得多。

接下来看看在服务端应该如何使用DatagramChannel接收数据。

下面给出服务端通过DatagramChannel接收数据的程序代码。大家目前不一定看得懂，因为代码中用到了Selector。

服务端是通过DatagramChannel绑定一个服务器地址（IP+端口），接收客户端发送过来的UDP数据报。服务端的完整代码如下：

```
package com.crazymakercircle.iodemo.udpDemos;  
//...  
public class UDPServer {  
    public void receive() throws IOException {  
        //获取DatagramChannel  
        DatagramChannel datagramChannel =  
        DatagramChannel.open();  
        //设置为非阻塞模式  
        datagramChannel.configureBlocking(false);  
        //绑定监听地址  
        datagramChannel.bind(new  
        InetSocketAddress("127.0.0.1", 18899));  
        Print.tcf("UDP服务器启动成功！");  
        //开启一个通道选择器
```

```
Selector selector = Selector.open();
//将通道注册到选择器
datagramChannel.register(selector,
SelectionKey.OP_READ);
//通过选择器查询IO事件
while (selector.select() > 0) {
    Iterator<SelectionKey> iterator =
        selector.selectedKeys().iterator();
    ByteBuffer buffer =
        ByteBuffer.allocate(NioDemoConfig.SEND_BUFFER_SIZE);

    //迭代IO事件
    while (iterator.hasNext()) {
        SelectionKey selectionKey = iterator.next();
        //可读事件，有数据到来
        if (selectionKey.isReadable()) {
            //读取DatagramChannel数据
            SocketAddress client =
                datagramChannel.receive(buffer);
            buffer.flip();
            Print.tcf0(new String(buffer.array(), 0,
buffer.limit()));
            buffer.clear();
        }
    }
    iterator.remove();
}
```

```
        }

        //关闭选择器和通道

        selector.close();

        datagramChannel.close();

    }

    public static void main(String[] args) throws IOException {

        new UDPServer().receive();

    }

}
```

在服务端，首先调用了bind()方法绑定DatagramChannel的监听端口。当数据到来时调用了receive()方法，从DatagramChannel接收数据后写入ByteBuffer缓冲区中。

在服务端代码中，为了监控数据的到来，使用了Selector。什么是Selector？如何使用Selector呢？请看下一节。

3.5 详解NIO Selector

Java NIO的三大核心组件是Channel（通道）、Buffer（缓冲区）、Selector（选择器）。其中，通道和缓冲区的联系比较密切：数据总是从通道读到缓冲区内，或者从缓冲区写入通道中。

前面两个组件已经介绍完毕，下面迎来最后一个非常重要的角色——选择器。

3.5.1 选择器与注册

选择器是什么？选择器和通道的关系又是什么？

简单地说，选择器的使命是完成I/O的多路复用，其主要工作是通道的注册、监听、事件查询。一个通道代表一条连接通路，通过选择器可以同时监控多个通道的I/O（输入输出）状况。选择器和通道的关系是监控和被监控的关系。

选择器提供了独特的API方法，能够选出（select）所监控的通道已经发生了哪些I/O事件，包括读写就绪的I/O操作事件。

在NIO编程中，一般是一个单线程处理一个选择器，一个选择器可以监控很多通道。所以，通过选择器，一个单线程可以处理数百、数千、数万甚至更多的通道。在极端情况下（数万个连接），只用一个线程就可以处理所有的通道，这样会大量地减少线程之间上下文切换的开销。

通道和选择器之间的关联通过register（注册）的方式完成。调用通道的Channel.register(Selector sel, int ops)方法，可以将通道实例注册到一个选择器中。register方法有两个参数：第一个参数指定通道注册到的选择器实例；第二个参数指定选择器要监控的IO事件类型。

可供选择器监控的通道IO事件类型包括以下四种：

- (1) 可读：SelectionKey.OP_READ。
- (2) 可写：SelectionKey.OP_WRITE。
- (3) 连接：SelectionKey.OP_CONNECT。
- (4) 接收：SelectionKey.OP_ACCEPT。

以上事件类型常量定义在SelectionKey类中。如果选择器要监控通道的多种事件，可以用“按位或”运算符来实现。例如，同时监控可读和可写IO事件：

```
//监控通道的多种事件，用“按位或”运算符来实现  
int key = SelectionKey.OP_READ | SelectionKey.OP_WRITE ;
```

什么是IO事件？

这个概念容易混淆，这里特别说明一下。这里的IO事件不是对通道的IO操作，而是通道处于某个IO操作的就绪状态，表示通道具备执行某个IO操作的条件。例如，某个SocketChannel传输通道如果完成了和对端的三次握手过程，就会发生“连接就绪”（OP_CONNECT）事件；某个ServerSocketChannel服务器连接监听通道，在监听到一个新

连接到来时，则会发生“接收就绪”（OP_ACCEPT）事件；一个SocketChannel通道有数据可读，就会发生“读就绪”（OP_READ）事件；一个SocketChannel通道等待数据写入，就会发生“写就绪”（OP_WRITE）事件。

说明

socket连接事件的核心原理和TCP连接的建立过程有关。关于TCP的核心原理和连接建立时的三次握手、四次挥手知识，请参阅本书后面有关TCP原理的内容。

3.5.2 SelectableChannel

并不是所有的通道都是可以被选择器监控或选择的。例如，FileChannel就不能被选择器复用。判断一个通道能否被选择器监控或选择有一个前提：判断它是否继承了抽象类SelectableChannel（可选择通道），如果是，就可以被选择，否则不能被选择。

简单地说，一个通道若能被选择，则必须继承SelectableChannel类。

SelectableChannel类是何方神圣呢？它提供了实现通道可选择性所需要的公共方法。Java NIO中所有网络连接socket通道都继承了SelectableChannel类，都是可选择的。FileChannel并没有继承SelectableChannel，因此不是可选择通道。

3.5.3 SelectionKey

通道和选择器的监控关系注册成功后就可以选择就绪事件，具体的选择工作可调用Selector的select()方法来完成。通过select()方法，选择器可以不断地选择通道中所发生操作的就绪状态，返回注册过的那些感兴趣的IO事件。换句话说，一旦在通道中发生了某些IO事件（就绪状态达成），并且是在选择器中注册过的IO事件，就会被选择器选中，并放入SelectionKey（选择键）的集合中。

SelectionKey是什么呢？简单地说，SelectionKey就是那些被选择器选中的IO事件。前面讲到，一个IO事件发生（就绪状态达成）后，如果之前在选择器中注册过，就会被选择器选中，并放入SelectionKey中；如果之前没有注册过，那么即使发生了IO事件，也不会被选择器选中。SelectionKey和IO的关系可以简单地理解为SelectionKey就是被选中了的IO事件。

在实际编程时，SelectionKey的功能是很强大的。通过SelectionKey，不仅可以获得通道的IO事件类型（比如SelectionKey.OP_READ），还可以获得发生IO事件所在的通道。另外，还可以获得选择器实例。

3.5.4 选择器使用流程

选择器的使用主要有以下三步：

(1) 获取选择器实例。选择器实例是通过调用静态工厂方法open()来获取的，具体如下：

```
//调用静态工厂方法open()来获取Selector实例  
Selector selector = Selector.open();
```

Selector的类方法open()的内部是向选择器SPI发出请求，通过默认的SelectorProvider（选择器提供者）对象获取一个新的选择器实例。Java中的SPI（Service Provider Interface，服务提供者接口）是一种可以扩展的服务提供者发现机制。Java通过SPI的方式提供选择器的默认实现版本。也就是说，其他的提供者可以通过SPI的方式提供定制化版本的选择器的动态替换或者扩展。

(2) 将通道注册到选择器实例。要实现选择器管理通道，需要将通道注册到相应的选择器上，简单的示例代码如下：

```
//获取通道  
ServerSocketChannel serverSocketChannel =  
ServerSocketChannel.open();  
  
//设置为非阻塞  
serverSocketChannel.configureBlocking(false);  
  
//绑定连接  
serverSocketChannel.bind(new InetSocketAddress(18899));  
  
//将通道注册到选择器上，并指定监听事件为“接收连接”  
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

上面通过调用通道的register()方法将ServerSocketChannel注册到了一个选择器上。当然，在注册之前，需要准备好通道。

这里需要注意：注册到选择器的通道必须处于非阻塞模式下，否则将抛出IllegalBlockingModeException异常。这意味着，

FileChannel不能与选择器一起使用，因为FileChannel只有阻塞模式，不能切换到非阻塞模式；而socket相关的所有通道都可以。其次，一个通道并不一定支持所有的四种IO事件。例如，服务器监听通道ServerSocketChannel仅支持Accept（接收到新连接）IO事件，而传输通道SocketChannel则不同，它不支持Accept类型的IO事件。

如何判断通道支持哪些事件呢？可以在注册之前通过通道的validOps()方法来获取该通道支持的所有IO事件集合。

(3) 选出感兴趣的IO就绪事件（选择键集合）。通过Selector的select()方法，选出已经注册的、已经就绪的IO事件，并且保存到SelectionKey集合中。SelectionKey集合保存在选择器实例内部，其元素为SelectionKey类型实例。调用选择器的selectedKeys()方法，可以取得选择键集合。

接下来，迭代集合的每一个选择键，根据具体IO事件类型执行对应的业务操作。大致的处理流程如下：

```
//轮询，选择感兴趣的IO就绪事件（选择键集合）
while (selector.select() > 0) {
    Set selectedKeys = selector.selectedKeys();
    Iterator keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        //根据具体的IO事件类型执行对应的业务操作
        if(key.isAcceptable()) {
            //IO事件：ServerSocketChannel服务器监听通道有新连接
        }
    }
}
```

```
        } else if (key.isConnectable()) {  
            //IO事件：传输通道连接成功  
        } else if (key.isReadable()) {  
            //IO事件：传输通道可读  
        } else if (key.isWritable()) {  
            //IO事件：传输通道可写  
        }  
        //处理完成后，移除选择键  
        keyIterator.remove();  
    }  
}
```

处理完成后，需要将选择键从SelectionKey集合中移除，以防止下一次循环时被重复处理。SelectionKey集合不能添加元素，如果试图向SelectionKey中添加元素，则将抛出java.lang.UnsupportedOperationException异常。

用于选择就绪的IO事件的select()方法有多个重载的实现版本，具体如下：

- (1) select(): 阻塞调用，直到至少有一个通道发生了注册的IO事件。
- (2) select(long timeout): 和select()一样，但最长阻塞时间为timeout指定的毫秒数。
- (3) selectNow(): 非阻塞，不管有没有IO事件都会立刻返回。

`select()`方法的返回值是整数类型（int），表示发生了I/O事件的数量，即从上一次`select`到这一次`select`之间有多少通道发生了I/O事件，更加准确地说是发生了选择器感兴趣（注册过）的I/O事件数。

3.5.5 使用NIO实现Discard服务器的实战案例

Discard服务器的功能很简单：仅读取客户端通道的输入数据，读取完成后直接关闭客户端通道，并且直接抛弃掉（Discard）读取到的数据。Discard服务器足够简单明了，作为第一个学习NIO的通信实例比较有参考价值。

下面的Discard服务器代码将选择器使用步骤进行了进一步细化：

```
package com.crazymakercircle.iodemo.NioDiscard;  
//...  
public class NioDiscardServer {  
    public static void startServer() throws IOException {  
        //1.获取选择器  
        Selector selector = Selector.open();  
        //2.获取通道  
        ServerSocketChannel serverSocketChannel =  
        ServerSocketChannel.open();  
        //3.设置为非阻塞  
        serverSocketChannel.configureBlocking(false);  
        //4.绑定连接  
        serverSocketChannel.bind(new InetSocketAddress(18899));  
        Logger.info("服务器启动成功");
```

```
//5.将通道注册的“接收新连接”IO事件注册到选择器上
serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);

//6.轮询感兴趣的IO就绪事件（选择键集合）
while (selector.select() > 0) {

    //7.获取选择键集合
    Iterator<SelectionKey> selectedKeys =
        selector.selectedKeys().iterator();

    while (selectedKeys.hasNext()) {
        //8.获取单个的选择键，并处理
        SelectionKey selectedKey = selectedKeys.next();
        //9.判断key是具体的什么事件
        if (selectedKey.isAcceptable()) {

            //10.若选择键的IO事件是“连接就绪”，就获取客户端连接
            SocketChannel socketChannel =
                serverSocketChannel.accept();
            //11.将新连接切换为非阻塞模式
            socketChannel.configureBlocking(false);
            //12.将新连接的通道的可读事件注册到选择器上
            socketChannel.register(selector,
SelectionKey.OP_READ);

        } else if (selectedKey.isReadable()) {
```

```
//13.若选择键的IO事件是“可读”，则读取数据
SocketChannel socketChannel =
        (SocketChannel)
selectedKey.channel();

//14.读取数据，然后丢弃
ByteBuffer byteBuffer =
        ByteBuffer.allocate(1024);
int length = 0;
while ((length =
socketChannel.read(byteBuffer)) >0)
{
    byteBuffer.flip();
    Logger.info(new String(byteBuffer.array(), 0,
length));
    byteBuffer.clear();
}
socketChannel.close();
}

//15.移除选择键
selectedKeys.remove();
}

}

//16.关闭连接
serverSocketChannel.close();
}

public static void main(String[] args) throws IOException {
```

```
        startServer();  
    }  
}
```

实现DiscardServer共分为16步，其中第7~15步是循环执行的，不断查询，将感兴趣的IO事件选择到选择键集合中，然后通过selector.selectedKeys()获取该选择键集合，并且进行迭代处理。在事件处理过程中，对于新建立的socketChannel1客户端传输通道，也要注册到同一个选择器上，这样就能使用同一个选择线程不断地对所有的注册通道进行选择键的查询。

在DiscardServer程序中，涉及两次选择器注册：一次是注册serverChannel1（服务器通道）；另一次是注册接收到的socketChannel1客户端传输通道。serverChannel1所注册的是新连接的IO事件SelectionKey.OP_ACCEPT，socketChannel1所注册的是可读IO事件SelectionKey.OP_READ。

注册完成后如果有事件发生，则DiscardServer在对选择键进行处理时先判断类型，然后进行相应的处理：

(1) 如果是SelectionKey.OP_ACCEPT新连接事件类型，代表serverChannel1接收到新的客户端连接，发生了新连接事件，则通过服务器通道的accept方法获取新的socketChannel1传输通道，并且将新通道注册到选择器。

(2) 如果是SelectionKey.OP_READ可读事件类型，代表某个客户端通道有数据可读，则读取选择键中socketChannel1传输通道的数据，进行业务处理，这里是直接丢弃数据。

客户端首先建立到服务器的连接，发送一些简单的数据，然后直接关闭连接。客户端的DiscardClient代码更加简单，代码如下：

```
package com.crazymakercircle.iodemo.NioDiscard;  
//...  
  
public class NioDiscardClient {  
  
    public static void startClient() throws IOException {  
  
        InetSocketAddress address = new  
InetSocketAddress("127.0.0.1", 18899);  
  
        //1. 获取通道  
  
        SocketChannel socketChannel =  
SocketChannel.open(address);  
  
        //2. 切换成非阻塞模式  
  
        socketChannel.configureBlocking(false);  
  
        //不断地自旋、等待连接完成，或者做一些其他的事情  
  
        while (!socketChannel.finishConnect()) {  
  
        }  
  
        Logger.info("客户端连接成功");  
  
        //3. 分配指定大小的缓冲区  
  
        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  
  
        byteBuffer.put("hello world".getBytes());  
  
        byteBuffer.flip();  
  
        //发送到服务器  
  
        socketChannel.write(byteBuffer);  
  
        socketChannel.shutdownOutput();  
  
        socketChannel.close();  
    }  
}
```

```
public static void main(String[] args) throws IOException {  
    startClient();  
}  
}
```

说明

如果需要执行整个Discard演示程序，首先要执行前面的**NioDiscardServer**服务端程序，然后才能执行本客户端程序。

通过Discard服务器的开发实战，大家应该对NIO Selector的使用流程了解得非常清楚了。下面来看一个稍微复杂一点的案例：在服务端接收文件和内容。

3.5.6 使用SocketChannel在服务端接收文件的实战案例

本示例演示文件的接收，是服务端的程序，和前面介绍的发送文件的SocketChannel客户端程序是相互配合使用的。由于在服务端需要用到选择器，因此直到完成了选择器的知识介绍之后才开始介绍NIO文件传输的socket服务端程序。服务端接收文件的示例代码如下：

```
package com.crazymakercircle.iodemo.socketDemos;  
//省略import
```

```
/***
 * 文件传输Server端
 * Created by 尼恩@ 疯狂创客圈
 */
public class NioReceiveServer
{
    //接收文件路径
    private static final String RECEIVE_PATH =
NioDemoConfig.SOCKET_RECEIVE_PATH;

    private Charset charset = Charset.forName("UTF-8");

    /**
     * 服务端保存的客户端对象，对应一个客户端文件
     */
    static class Client
    {
        //文件名称
        String fileName;
        //长度
        long fileLength;
        //开始传输的时间
        long startTime;
        //客户端的地址
    }
}
```

```
InetSocketAddress remoteAddress;

//输出的文件通道
FileChannel outChannel;

//接收长度
long receiveLength;

public boolean isFinished()
{
    return receiveLength >= fileLength;
}

private ByteBuffer buffer
=
ByteBuffer.allocate(NioDemoConfig.SERVER_BUFFER_SIZE);

//使用Map保存每个客户端传输
//当OP_READ通道可读时，根据Channel找到对应的对象
Map<SelectableChannel, Client> clientMap =
new HashMap<SelectableChannel, Client>();

public void startServer() throws IOException
{
    //1.获取选择器
```

```
Selector selector = Selector.open();

//2.获取通道
ServerSocketChannel serverChannel =
ServerSocketChannel.open();

ServerSocket serverSocket = serverChannel.socket();

//3.设置为非阻塞
serverChannel.configureBlocking(false);

//4.绑定连接
InetSocketAddress address
= new InetSocketAddress(18899);

serverSocket.bind(address);

//5.将通道注册到选择器上，并且注册的IO事件为“接收新连接”
serverChannel.register(selector,
SelectionKey.OP_ACCEPT);

Print.tcf("serverChannel is listening...");

//6.轮询感兴趣的I/O就绪事件（选择键集合）
while (selector.select() > 0)

{
    //7.获取选择键集合
    Iterator<SelectionKey> it =
selector.selectedKeys().iterator();

    while (it.hasNext())

    {
        //8.获取单个的选择键，并处理
        SelectionKey key = it.next();
```

```
//9.判断key是具体的什么事件，是否为新连接事件
if (key.isAcceptable())
{
    //10.若接收的事件是“新连接”，则获取客户端新连接
    ServerSocketChannel server =
        (ServerSocketChannel) key.channel();
    SocketChannel socketChannel =
        server.accept();
    if (socketChannel == null) continue;
    //11.客户端新连接，切换为非阻塞模式
    socketChannel.configureBlocking(false);
    //12.将客户端新连接通道注册到Selector上
    SelectionKey selectionKey
    =socketChannel.register(selector, SelectionKey.OP_READ);
    //余下为业务处理
    Client client = new Client();
    client.remoteAddress=(InetSocketAddress)
    socketChannel.
    getRemoteAddress();
    clientMap.put(socketChannel, client);
    Logger.debug(socketChannel.getRemoteAddress() + "连接成
功...");
}

} else if (key.isReadable())
{
    processData(key);
```

```
        }

        //NIO的特点是只会累加，已选择键的集合不会删除
        //如果不删除，下一次又会被select()函数选中
        it.remove();

    }

}

}

/**
 * 处理客户端传输过来的数据
 */
private void processData(SelectionKey key) throws
IOException
{
    Client client = clientMap.get(key.channel());

    SocketChannel socketChannel = (SocketChannel)
key.channel();

    int num = 0;

    try
    {
        buffer.clear();

        while ((num = socketChannel.read(buffer)) > 0)

        {

            buffer.flip();

            //客户端发送过来的，首先处理文件名

            if (null == client.fileName)
```

```
{  
  
    if (buffer.capacity() < 4)  
    {  
        continue;  
    }  
  
    int fileNameLen = buffer.getInt();  
    byte[] fileNameBytes = new  
byte[fileNameLen];  
  
    buffer.get(fileNameBytes);  
  
    //文件名  
    String fileName = new String(fileNameBytes,  
charset);  
  
    File directory = new File(RECEIVE_PATH);  
    if (!directory.exists())  
    {  
        directory.mkdir();  
    }  
    Logger.info("NIO 传输目标dir: ", directory);  
  
    client.fileName = fileName;  
    String fullName =  
directory.getAbsolutePath() + File.  
separatorChar + fileName;  
    Logger.info("NIO 传输目标文件: ", fullName);
```

```
File file = new File(fullName.trim());  
  
if (!file.exists())  
{  
    file.createNewFile();  
}  
  
FileChannel fileChannel = new  
FileOutputStream(file).  
getChannel();  
client.outChannel = fileChannel;  
  
if (buffer.capacity() < 8)  
{  
    continue;  
}  
  
//文件长度  
long fileLength = buffer.getLong();  
client.fileLength = fileLength;  
client.startTime =  
System.currentTimeMillis();  
Logger.debug("NIO 传输开始: ");  
  
client.receiveLength += buffer.capacity();  
if (buffer.capacity() > 0)  
{  
    //写入文件
```

```
        client.outChannel.write(buffer);

    }

    if (client.isFinished())
    {

        finished(key, client);

    }

    buffer.clear();

}

//客户端发送过来的，最后是文件内容

else

{

    client.receiveLength += buffer.capacity();

    //写入文件

    client.outChannel.write(buffer);

    if (client.isFinished())
    {

        finished(key, client);

    }

    buffer.clear();

}

key.cancel();

} catch (IOException e)

{

    key.cancel();

    e.printStackTrace();

}
```

```
        return;

    }

    //调用close为-1， 到达末尾

    if (num == -1)

    {

        finished(key, client);

        buffer.clear();

    }

}

private void finished(SelectionKey key, Client client)

{

    IOUtil.closeQuietly(client.outChannel);

    Logger.info("上传完毕");

    key.cancel();

    Logger.debug("文件接收成功,File Name: " + client.fileName);

    Logger.debug(" Size: " + IOUtil.getFormatFileSize(client.fileLength));

    long endTime = System.currentTimeMillis();

    Logger.debug("NIO IO 传输毫秒数: " + (endTime - client.startTime));

}

/***
 * 入口

```

```
*/  
public static void main(String[] args) throws Exception  
{  
    NioReceiveServer server = new NioReceiveServer();  
    server.startServer();  
}  
}
```

客户端每次传输文件都会分为多次传输：首先传入文件名称，其次是文件大小，然后是文件内容。

对于每一个客户端socketChannel，创建一个客户端对象，用于保存客户端状态，分别保存文件名、文件大小和写入的目标文件通道outChannel。

socketChannel和Client对象之间是一对一的对应关系：建立连接时，在Map中以键-值对的形式保存Client实例，其中socketChannel作为键（Key），Client对象作为值（Value）。当socketChannel传输通道有数据可读时，通过选择键key.channel()方法取出IO事件所在的socketChannel通道，然后通过socketChannel通道从map中获取对应的Client对象。

接收到数据时，如果文件名为空，就先处理文件名称，并把文件名保存到Client对象，同时创建服务器上的目标文件；接下来读取到数据，说明接收到了文件大小，把文件大小保存到Client对象；接下来接收到数据，说明是文件内容，写入Client对象的outChannel文件通道中，直到数据读取完毕。

运行方式是先启动NioReceiveServer服务器程序，再启动前面介绍的客户端程序NioSendClient，完成文件的传输。

由于NIO传输是非阻塞、异步的，因此在传输过程中会出现“粘包”和“半包”问题。正因如此，无论是前面NIO文件传输实例还是Discard服务器程序，都会在传输过程中出现异常现象（偶现）。由于以上实例在生产过程中不会使用，仅仅是为了大家学习NIO的知识，所以没有为了解决“粘包”和“半包”问题而将代码编写得很复杂。

说明

很多小伙伴在“疯狂创客圈”社群的交流群反馈：在执行以上实例时，传输过程中会出现异常现象——部分内容传输出错。其实并不是程序问题，而是传输过程中发生了“粘包”和“半包”问题。后面的章节会专门介绍“粘包”和“半包”问题及其根本性的解决方案。

第4章 鼎鼎大名的Reactor模式

本书的原则是从基础讲起，而Reactor（反应器）模式是高性能网络编程在设计和架构层面的基础模式，算是基础的原理性知识。只有彻底了解反应器的原理，才能真正构建好高性能的网络应用、轻松地学习和掌握高并发通信服务器与框架（如Netty框架、Nginx服务器）。

正因为Reactor模式是高并发的重要基础原理，所以该模式也是BAT级别大公司必不可少的面试题。

4.1 Reactor模式的重要性

在详细介绍什么是Reactor模式之前，首先说明一下它的重要性。

到目前为止，高性能网络编程都绕不开Reactor模式。很多著名的服务器软件或者中间件都是基于Reactor模式实现的。例如，“全宇宙有名的、高性能”的Web服务器Nginx就是基于Reactor模式的；如雷贯耳的Redis，作为高性能的缓存服务器之一，也是基于Reactor模式的；目前热门的在开源项目中应用极为广泛的高性能通信中间件Netty，还是基于Reactor模式的。

从开发的角度来说，要完成和胜任高性能的服务器开发，Reactor模式是必须学会和掌握的。从学习的角度来说，Reactor模式相当于高性能、高并发的一项非常重要的基础知识，只有掌握了它，才能真正理解和掌握Nginx、Redis、Netty等这些大名鼎鼎的中间件技术。正因为如此，在大的互联网公司（如阿里、腾讯、京东）的面试过程中，Reactor模式相关的问题是经常出现的面试问题。

总之，Reactor模式是高性能网络编程必知、必会的模式。

4.1.1 为什么首先学习Reactor模式

本书的目标是学习基于Netty的开发高性能通信服务器。为什么在学习Netty之前首先要学习Reactor模式呢？

资深程序员都知道，Java程序不是按照顺序执行的逻辑来组织的。代码中所用到的设计模式在一定程度上已经演变成代码的组织方

式。越是高水平的Java代码，抽象的层次越高，到处都是高度抽象和面向接口的调用，大量用到继承、多态、设计模式。

在阅读别人的源代码时，如果不了解代码所使用的设计模式，往往会觉得晕头转向，不知身在何处，对代码跟踪和阅读都很成问题。反过来，如果先掌握到代码的设计模式，再去阅读代码，其过程就会变得很轻松，代码也就不会那么难懂了。

当然，在编写代码时，如果不能熟练地掌握设计模式，也很难写出高水平的Java代码。

本书的重要使命之一就是帮助大家学习和掌握高并发通信（包括Netty框架）。Netty本身很抽象，大量应用了设计模式。所以，学习像Netty这样的“精品中的精品”框架也是需要先从设计模式入手的，而Netty的整体架构是基于Reactor模式的。

所以，学习和掌握Reactor模式，对于开始学习高并发通信（包括Netty框架）的人来说，一定是磨刀不误砍柴工。

4.1.2 Reactor模式简介

本书站在巨人的肩膀上，引用一下Doug Lea大师在文章“Scalable IO in Java”中对Reactor模式的定义：

Reactor模式由Reactor线程、Handlers处理器两大角色组成，两大角色的职责分别如下：

(1) Reactor线程的职责：负责响应IO事件，并且分发到Handlers处理器。

(2) `Handlers`处理器的职责：非阻塞的执行业务处理逻辑。

说明

Doug Lea是一位让人无限景仰的大师，是Java中`Concurrent`并发包（简称JUC包）的作者。`Concurrent`并发包的原理和使用是一个Java工程师必备的基础知识，有关其具体内容请参阅《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》。

从上面的Reactor模式定义中看不出这种模式有什么神奇的地方。当然，从简单到复杂，Reactor模式也有很多版本，前面的定义仅仅是最为简单的一个版本。如果需要彻底了解Reactor模式，还得从最原始的OIO编程开始讲起。

4.1.3 多线程OIO的致命缺陷

在Java的OIO编程中，原始的网络服务器程序一般使用一个`while`循环不断地监听端口是否有新的连接。如果有，就调用一个处理函数来完成传输处理。示例代码如下：

```
while(true){  
    socket = accept(); //阻塞，接收连接  
    handle(socket); //读取数据、业务处理、写入结果  
}
```

这种方法的最大问题是：如果前一个网络连接的handle(socket)没有处理完，那么后面的新连接无法被服务端接收，于是后面的请求就会被阻塞，导致服务器的吞吐量太低。这对于服务器来说是一个严重的问题。

为了解决这个严重的连接阻塞问题，出现了一个极为经典的模式：Connection Per Thread（一个线程处理一个连接）模式。示例代码如下：

```
package com.crazymakercircle.iodemo.OIO;  
//省略import导入的Java类  
  
class ConnectionPerThread implements Runnable {  
  
    public void run() {  
  
        try {  
  
            //服务器监听socket  
  
            ServerSocket serverSocket =  
                new  
                    ServerSocket(NioDemoConfig.SOCKET_SERVER_PORT);  
  
            while (!Thread.interrupted()) {  
  
                Socket socket = serverSocket.accept();  
  
                //接收一个连接后，为socket连接，新建一个专属的处理器对  
                //象  
  
                Handler handler = new Handler(socket);  
  
                //创建新线程，专门负责一个连接的处理  
  
                new Thread(handler).start();  
            }  
        }  
    }  
}
```

```
        } catch (IOException ex) { /* 处理异常 */ }

    }

//处理器，这里将内容回显到客户端

static class Handler implements Runnable {

    final Socket socket;

    Handler(Socket s) {

        socket = s;

    }

    public void run() {

        while (true) {

            try {

                byte[] input = new byte[1024];

                /* 读取数据 */

                socket.getInputStream().read(input);

                /* 处理业务逻辑，获取处理结果 */

                byte[] output = null;

                /* 写入结果 */

                socket.getOutputStream().write(output);

            } catch (IOException ex) { /*处理异常*/ }

        }

    }

}

}
```

以上示例代码中，对于每一个新的网络连接都分配给一个线程。每个线程都独自处理自己负责的socket连接的输入和输出。当然，服务器的监听线程也是独立的，任何socket连接的输入和输出处理都不

会阻塞到后面新socket连接的监听和建立，这样服务器的吞吐量就得到了提升。早期版本的Tomcat服务器就是这样实现的。

Connection Per Thread模式（一个线程处理一个连接）的优点是解决了前面的新连接被严重阻塞的问题，在一定程度上较大地提高了服务器的吞吐量。

Connection Per Thread模式的缺点是对应于大量的连接，需要耗费大量的线程资源，对线程资源要求太高。在系统中，线程是比较昂贵的系统资源。如果线程的数量太多，系统将无法承受。而且，线程的反复创建、销毁、切换也需要代价。因此，在高并发的应用场景下，多线程OIO的缺陷是致命的。

新的问题来了：如何减少线程数量？比如说让一个线程同时负责处理多个socket连接的输入和输出，行不行？看上去没有什么不可以，实际上作用不大。因为在传统OIO编程中每一次socket传输的I/O读写处理都是阻塞的。在同一时刻，一个线程里只能处理一个socket的读写操作，前一个socket操作被阻塞了，其他连接的I/O操作同样无法被并行处理。所以，在OIO中，即使是一个线程同时负责处理多个socket连接的输入和输出，同一时刻该线程也只能处理一个连接的I/O操作。

如何解决Connection Per Thread模式的巨大缺陷呢？一个有效途径是使用Reactor模式。用Reactor模式对线程的数量进行控制，做到一个线程处理大量的连接。那么它是如何做到的呢？首先来看一个简单的版本——单线程的Reactor模式。

4.2 单线程Reactor模式

总体来说，Reactor模式有点类似事件驱动模式。在事件驱动模式中，当有事件触发时，事件源会将事件分发到Handler（处理器），由Handler负责事件处理。Reactor模式中的反应器角色类似于事件驱动模式中的事件分发器（Dispatcher）角色。

具体来说，在Reactor模式中有Reactor和Handler两个重要的组件：

(1) Reactor：负责查询I/O事件，当检测到一个I/O事件时将其发送给相应的Handler处理器去处理。这里的I/O事件就是NIO中选择器查询出来的通道I/O事件。

(2) Handler：与I/O事件（或者选择键）绑定，负责I/O事件的处理，完成真正的连接建立、通道的读取、处理业务逻辑、负责将结果写到通道等。

4.2.1 什么是单线程Reactor

什么是单线程版本的Reactor模式呢？简单地说，Reactor和Handlers处于一个线程中执行。这是最简单的Reactor模型，如图4-1所示。

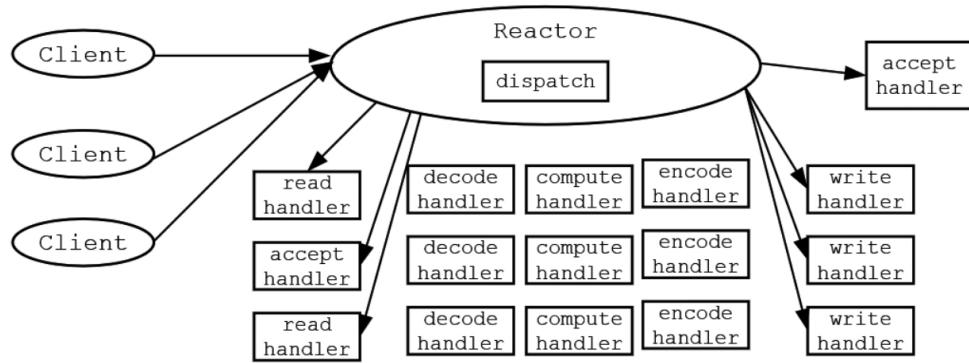


图4-1 单线程Reactor模式

基于Java NIO如何实现简单的单线程版本的Reactor模式呢？需要用到SelectionKey（选择键）的几个重要的成员方法：

(1) void attach(Object o): 将对象附加到选择键。

此方法可以将任何Java POJO对象作为附件添加到SelectionKey实例。此方法非常重要，因为在单线程版本的Reactor模式实现中可以将Handler实例作为附件添加到SelectionKey实例。

(2) Object attachment(): 从选择键获取附加对象。

此方法与attach(Object o)是配套使用的，其作用是取出之前通过attach(Object o)方法添加到SelectionKey实例的附加对象。这个方法同样非常重要，当I0事件发生时，选择键将被select方法查询出来，可以直接将选择键的附件对象取出。

在Reactor模式实现中，通过attachment()方法所取出的是之前通过attach(Object o)方法绑定的Handler实例，然后通过该Handler实例完成相应的传输处理。

总之，在Reactor模式中，需要将attach和attachment结合使用：在选择键注册完成之后调用attach()方法，将Handler实例绑定到选择键；当IO事件发生时调用attachment()方法，可以从选择键取出Handler实例，将事件分发到Handler处理器中完成业务处理。

4.2.2 单线程Reactor的参考代码

Doug Lea在“Scalable I/O in Java”一文中实现了一个单线程Reactor模式的参考代码。这里，我们站在巨人的肩膀上，借鉴Doug Lea的实现，对Reactor模式进行介绍。为了方便说明，本书对Doug Lea的参考代码进行一些适当的修改。具体的参考代码如下：

```
package com.crazymakercircle.ReactorModel;  
//省略import  
//单线程Reactor  
class EchoServerReactor implements Runnable {  
    Selector selector;  
    ServerSocketChannel serverSocket;  
    //构造函数  
    EchoServerReactor() throws IOException {  
        //省略：打开选择器、serverSocket连接监听通道  
        //注册serverSocket的accept新连接接收事件  
        SelectionKey sk =serverSocket.register(selector,  
        SelectionKey.OP_ACCEPT);  
        //将新连接处理器作为附件，绑定到sk选择键  
        sk.attach(new AcceptorHandler());  
    }  
}
```

```

public void run() {
    //选择器轮询
    try {
        while (!Thread.interrupted()) {
            selector.select();
            Set selected = selector.selectedKeys();
            Iterator it = selected.iterator();
            while (it.hasNext()) {
                //反应器负责dispatch收到的事件
                SelectionKey sk=it.next();
                dispatch(sk);
            }
            selected.clear();
        }
    } catch (IOException ex) { ex.printStackTrace(); }
}

//反应器的分发事件
void dispatch(SelectionKey k) {
    Runnable handler = (Runnable) (k.attachment());
    //调用之前绑定到选择键的handler对象
    if (handler != null) {
        handler.run();
    }
}

//处理器：处理新连接
class AcceptorHandler implements Runnable {

```

```

public void run() {
    //接受新连接
    SocketChannel channel = serverSocket.accept();
    //需要为新连接创建一个输入输出的handler
    if (channel != null)
        new EchoHandler(selector, channel);
}
}

//...
}

```

在上面的代码中设计了一个Handler，叫作AcceptorHandler处理器，它是一个内部类。在注册serverSocket服务监听连接的接受事件之后，创建一个AcceptorHandler新连接处理器的实例作为附件，被附加（attach）到SelectionKey中。

```

//为serverSocket注册新连接接受(accept)事件
SelectionKey sk = serverSocket.register(selector,
SelectionKey.OP_ACCEPT);
//将新连接处理器作为附件，绑定到sk选择键
sk.attach(new AcceptorHandler());

```

当新连接事件发生后，取出之前附加到SelectionKey中的Handler业务处理器进行socket的各种IO处理。

```

void dispatch(SelectionKey k) {
    Runnable r = (Runnable) (k.attachment());
    //调用之前绑定到选择键的处理器对象
}

```

```
    if (r != null) {  
        r.run();  
    }  
}
```

处理器AcceptorHandler的两大职责是完成新连接的接收工作、为新连接创建一个负责数据传输的Handler（称之为IOHandler）。

```
//新连接处理器  
  
class AcceptorHandler implements Runnable {  
  
    public void run() {  
        //接受新连接  
  
        SocketChannel channel = serverSocket.accept();  
  
        //需要为新连接创建一个输入输出的Handler  
  
        if (channel != null) new EchoHandler(selector,  
channel);  
    }  
}
```

顾名思义，IOHandler就是负责socket连接的数据输入、业务处理、结果输出。该处理器的示例代码大致如下：

```
package com.crazymakercircle.ReactorModel;  
  
//负责数据传输的Handler  
  
class IOHandler implements Runnable {  
  
    final SocketChannel channel;  
    final SelectionKey sk;  
  
    IOHandler (Selector selector, SocketChannel c) {  
        channel = c;  
    }
```

```
c.configureBlocking(false);

//与之前的注册方式不同，先仅仅取得选择键，之后再单独设置感兴趣的
IO事件

sk = channel.register(selector, 0); //仅仅取得选择键
//将Handler处理器作为选择键的附件
sk.attach(this);
//注册读写就绪事件

sk.interestOps(SelectionKey.OP_READ|SelectionKey.OP_WRITE);

}

public void run() {
//...处理输入和输出
}

}
```

在传输处理器IOHandler的构造器中，有两点比较重要：

(1) 将新的SocketChannel传输通道注册到Reactor类的同一个选择器中。这样保证了Reactor在查询IO事件时能查询到Handler注册到选择器的IO事件（数据传输事件）。

(2) Channel传输通道注册完成后，将IOHandler实例自身作为附件附加到选择键中。这样，在Reactor类分发事件（选择键）时，能执行到IOHandler的run()方法，完成数据传输处理。

如果由于上面的示例代码过于复杂而导致不能被快速理解，可以参考下面的EchoServer回显服务器实例，自己动手开发一个可以执行的单线程反应器实例。

4.2.3 单线程Reactor模式的EchoServer的实战案例

EchoServer的功能很简单：读取客户端的输入并回显到客户端，所以也叫回显服务器。基于Reactor模式来实现，设计三个重要的类：

- (1) 设计一个反应器类：EchoServerReactor类。
- (2) 设计两个处理器类：AcceptorHandler新连接处理器、EchoHandler回显处理器。

反应器类EchoServerReactor的实现思路和前面的示例代码基本上相同，具体如下：

```
package com.crazymakercircle.ReactorModel;  
//省略import  
//反应器  
class EchoServerReactor implements Runnable {  
    Selector selector;  
    ServerSocketChannel serverSocket;  
    //构造器  
    EchoServerReactor() throws IOException {  
        //省略：获取选择器、开启serverSocket服务监听通道  
        //省略：绑定AcceptorHandler新连接处理器到selectKey  
    }  
    //轮询和分发事件  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {
```

```

        selector.select();

        Set<SelectionKey> selected =
    selector.selectedKeys();

        Iterator<SelectionKey> it =
selected.iterator();

        while (it.hasNext()) {

            //反应器负责dispatch收到的事件

            SelectionKey sk = it.next();
            dispatch(sk);

        }

        selected.clear();
    }

} catch (IOException ex) {
    ex.printStackTrace();
}

}

//反应器的事件分发

void dispatch(SelectionKey sk) {
    Runnable handler = (Runnable) sk.attachment();
    //调用之前，附加绑定到选择键的handler对象
    if (handler != null) {
        handler.run();
    }
}

//Handler之一：新连接处理器

class AcceptorHandler implements Runnable {

```

```

        public void run() {
            try {
                SocketChannel channel = serverSocket.accept();
                if (channel != null)
                    new EchoHandler(selector, channel);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws IOException {
        new Thread(new EchoServerReactor()).start();
    }
}

```

第二个处理器为EchoHandler回显处理器，也是一个传输处理器，主要是完成客户端的内容读取和回显，具体如下：

```

import com.crazymakercircle.util.Logger;
//...
class EchoHandler implements Runnable {
    final SocketChannel channel;
    final SelectionKey sk;
    final ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
    //处理器实例的状态：发送和接收，一个连接对应一个处理器实例
    static final int RECIEVING = 0, SENDING = 1;
}

```

```
int state = RECEIVING;

//构造器

EchoHandler(Selector selector, SocketChannel c) {

    channel = c;

    c.configureBlocking(false);

    //取得选择键，再设置感兴趣的IO事件

    sk = channel.register(selector, 0);

    //将Handler自身作为选择键的附件，一个连接对应一个处理器实例

    sk.attach(this);

    //注册Read就绪事件

    sk.interestOps(SelectionKey.OP_READ);

    selector.wakeup();

}

public void run() {

    try {

        if (state == SENDING) {

            //发送状态，把数据写入连接通道

            channel.write(byteBuffer);

            //byteBuffer切换成写模式，写完后，就准备开始从通道读

            byteBuffer.clear();

            //注册read就绪事件，开始接收客户端数据

            sk.interestOps(SelectionKey.OP_READ);

            //修改状态，进入接收状态

            state = RECEIVING;

        } else if (state == RECEIVING) {

            //接收状态，从通道读取数据

        }

    }

}
```

```

        int length = 0;
        while ((length = channel.read(byteBuffer)) > 0)
    {
        Logger.info(new String(byteBuffer.array(), 0,
length));
    }

        //读完后，翻转byteBuffer的读写模式
        byteBuffer.flip();
        //准备写数据到通道，注册write就绪事件
        sk.interestOps(SelectionKey.OP_WRITE);
        //注册完成后，进入发送状态
        state = SENDING;
    }

        //处理结束了，这里不能关闭select key，需要重复使用
        //sk.cancel();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}

```

以上代码是一个基于Reactor模式的EchoServer回显服务器的完整实现。它是一个单线程版本的Reactor模式，Reactor和所有的Handler实例都在同一条线程中执行。

运行EchoServerReactor类中的main()方法，可以启动回显服务器。如果要看到具体的回显输出，还需要启动客户端程序。客户端的

代码在同一个包下，其类名为EchoClient，它的主要职责为数据的发送。打开源代码工程，直接运行即可。由于篇幅原因，这里不再贴出客户端的代码。

4.2.4 单线程Reactor模式的缺点

单线程Reactor模式是基于Java的NIO实现的。相对于传统的多线程IO，Reactor模式不再需要启动成千上万条线程，避免了线程上下文的频繁切换，服务端的效率自然是大大提升了。

在单线程Reactor模式中，Reactor和Handler都在同一条线程中执行。这样，带来了一个问题：当其中某个Handler阻塞时，会导致其他所有的Handler都得不到执行。在这种场景下，被阻塞的Handler不仅仅负责输入和输出处理的传输处理器，还包括负责新连接监听的AcceptorHandler处理器，可能导致服务器无响应。这是一个非常严重的缺陷，导致单线程反应器模型在生产场景中使用得比较少。

除此之外，目前的服务器都是多核的，单线程Reactor模式模型不能充分利用多核资源。总之，在高性能服务器应用场景中，单线程Reactor模式实际使用的很少。

4.3 多线程Reactor模式

Reactor和Handler挤在单个线程中会造成非常严重的性能缺陷，可以使用多线程来对基础的Reactor模式进行改造和演进。

4.3.1 多线程版本的Reactor模式演进

多线程Reactor的演进分为两个方面：

(1) 升级Handler。既要使用多线程，又要尽可能高效率，则可以考虑使用线程池。

(2) 升级Reactor。可以考虑引入多个Selector（选择器），提升选择大量通道的能力。

总体来说，多线程版本的Reactor模式大致如下：

(1) 将负责数据传输处理的IOHandler处理器的执行放入独立的线程池中。这样，业务处理线程与负责新连接监听的反应器线程就能相互隔离，避免服务器的连接监听受到阻塞。

(2) 如果服务器为多核的CPU，可以将反应器线程拆分为多个子反应器（SubReactor）线程；同时，引入多个选择器，并且为每一个SubReactor引入一个线程，一个线程负责一个选择器的事件轮询。这样充分释放了系统资源的能力，也大大提升了反应器管理大量连接或者监听大量传输通道的能力。

4.3.2 多线程版本Reactor的实战案例

在前面的“回显服务器”（EchoServerReactor）的基础上完成多线程反应器的升级。多线程反应器的实战案例设计如下：

- (1) 引入多个选择器。
- (2) 设计一个新的子反应器（SubReactor）类，子反应器负责查询一个选择器。
- (3) 开启多个处理线程，一个处理线程负责执行一个子反应器。

为了提升效率，可以让SubReactor的数量和选择器的数量一致，避免多个线程负责一个选择器，导致需要进行线程同步，引起效率降低。

多线程版本反应器MultiThreadEchoServerReactor的逻辑模型如图4-2所示。

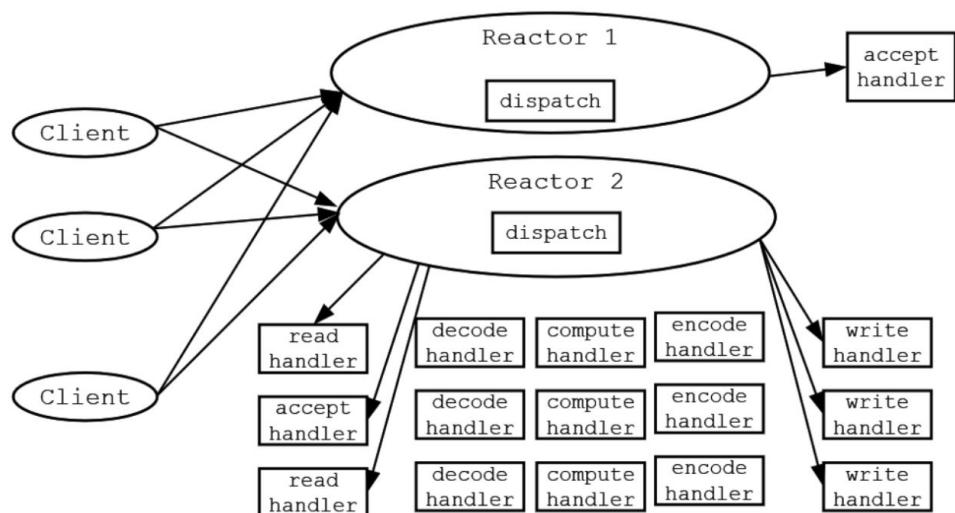


图4-2 多线程版本反应器MultiThreadEchoServerReactor的逻辑模型

多线程版本反应器MultiThreadEchoServerReactor的参考代码大致如下：

```
package com.crazymakercircle.ReactorModel;  
//...  
//多线程版本反应器  
class MultiThreadEchoServerReactor {  
    ServerSocketChannel serverSocket;  
    AtomicInteger next = new AtomicInteger(0);  
    //选择器集合，引入多个选择器  
    Selector[] selectors = new Selector[2];  
    //引入多个子反应器  
    SubReactor[] subReactors = null;  
    MultiThreadEchoServerReactor() throws IOException {  
        //初始化多个选择器  
        selectors[0] = Selector.open(); //用于监听新连接事件  
        selectors[1] = Selector.open(); //用于监听传输事件  
        serverSocket = ServerSocketChannel.open();  
        InetSocketAddress address =  
            new InetSocketAddress("127.0.0.1", 18899);  
        serverSocket.socket().bind(address);  
        //非阻塞  
        serverSocket.configureBlocking(false);  
        //第一个选择器，负责监控新连接事件  
        SelectionKey sk =  
            serverSocket.register(selectors[0], OP_ACCEPT);  
        //绑定Handler：新连接监控Handler绑定到SelectionKey（选择键）
```

```

        sk.attach(new AcceptorHandler());
        //第一个子反应器，负责第一个选择器的新连接事件分发（而不处理）
        SubReactor subReactor1 = new SubReactor(selectors[0]);
        //第二个子反应器，负责第二个选择器的传输事件的分发（而不处理）
        SubReactor subReactor2 = new SubReactor(selectors[1]);
        subReactors = new SubReactor[]{subReactor1,
        subReactor2};
    }

private void startService() {
    //一个子反应器对应一个线程
    new Thread(subReactors[0]).start();
    new Thread(subReactors[1]).start();
}

//子反应器，负责事件分发，但是不负责事件处理
class SubReactor implements Runnable {
    //每个线程负责一个选择器的查询和选择
    final Selector selector;
    public SubReactor(Selector selector) {
        this.selector = selector;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                selector.select();
                Set<SelectionKey>keySet =

```

```

        selector.selectedKeys();

                Iterator<SelectionKey> it =
keySet.iterator();

                while (it.hasNext()) {
                        //dispatch所查询的事件
                        SelectionKey sk = it.next();
                        dispatch(sk);
                }

                keySet.clear();
}

} catch (IOException ex) {
        ex.printStackTrace();
}

}

void dispatch(SelectionKey sk) {
        Runnable handler = (Runnable) sk.attachment();
        //获取之前attach绑定到选择键的handler处理器对象，执行事件
处理

        if (handler != null) {
                handler.run();
        }
}

}

//Handler:新连接处理器

class AcceptorHandler implements Runnable {
        public void run() {

```

```
try {
    SocketChannel channel = serverSocket.accept();
    //创建传输处理器，并且将传输通道注册到选择器2
    if (channel != null)
        new MultiThreadEchoHandler(selectors[1],
channel);
} catch (IOException e) {
    e.printStackTrace();
}
}

public static void main(String[] args) throws IOException {
    MultiThreadEchoServerReactor server =
        new MultiThreadEchoServerReactor();
    server.startService();
}
}
```

上面是反应器的多线程版本演进代码，创建了两个子反应器，负责查询和分发两个选择器的事件。

总共有两个选择器：第一个选择器专门负责查询和分发新连接事件，第二个选择器专门负责查询和分发IO传输事件。

总共有两条事件轮询线程：第一条线程为新连接事件轮询线程，专门轮询第一个选择器；第二条线程为IO事件轮询线程，专门轮询第二个选择器。

服务端的监听通道注册到第一个选择器，而所有的Socket传输通道都注册到第二个选择器，从而实现了新连接监听和IO读写事件监听的线程分离。

接下来为大家演示一下Handler的多线程演进。

4.3.3 多线程版本Handler的实战案例

仍然基于前面的单线程Reactor模式的回显处理器的程序代码加以改进，新的回显处理器为MultiThreadEchoHandler，主要的升级是引入了一个线程池（ThreadPool），使得数据传输和业务处理的代码执行在独立的线程池中，彻底地做到IO处理以及业务处理线程和反应器IO事件轮询线程的完全隔离。这个实战案例的代码如下：

```
class MultiThreadEchoHandler implements Runnable
{
    final SocketChannel channel;
    final SelectionKey sk;
    final ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
    static final int RECIEVING = 0, SENDING = 1;
    int state = RECIEVING;
    //引入线程池
    static ExecutorService pool =
        Executors.newFixedThreadPool(4);
    //构造器
    MultiThreadEchoHandler(Selector selector, SocketChannel c)
    {
    }
```

```
channel = c;
c.configureBlocking(false);
//先取得选择键，再设置感兴趣的IO事件
sk = channel.register(selector, 0);
//将本Handler作为sk选择键的附件，方便事件dispatch
sk.attach(this);
//向sk选择键设置Read就绪事件
sk.interestOps(SelectionKey.OP_READ);
selector.wakeup();
}

//此run方法在IO事件轮询线程中被调用
public void run()
{
    //提交数据传输任务到线程池
    //使得IO处理不在IO事件轮询线程中执行，而是在独立的线程池中执行
    pool.submit(() -> asyncRun());
}

//数据传输与业务处理任务，不在IO事件轮询线程中执行，而是在独立的线程池
//中执行
public synchronized void asyncRun()
{
    try
    {
        if (state == SENDING)
        {
            //写入通道
        }
    }
}
```

```
        channel.write(byteBuffer);
        //byteBuffer切换成写模式，写完后准备开始从通道读
        byteBuffer.clear();
        //写完后，注册read就绪事件
        sk.interestOps(SelectionKey.OP_READ);
        //进入接收的状态
        state = RECEIVING;
    } else if (state == RECEIVING)
    {
        //从通道读
        int length = 0;
        while ((length = channel.read(byteBuffer)) > 0)
        {
            Logger.info(new String(byteBuffer.array(), 0,
length));
        }
        //读完后，翻转byteBuffer的读写模式
        byteBuffer.flip();
        //注册write就绪事件
        sk.interestOps(SelectionKey.OP_WRITE);
        //进入发送的状态
        state = SENDING;
    }
    //处理结束了，这里不能关闭select key，需要重复使用
    //sk.cancel();
} catch (IOException ex)
{
}
```

```
    ex.printStackTrace();  
}  
}  
}
```

以上代码中，IO操作和业务处理被提交到线程池中异步执行，为了避免发送和读取的状态混乱，需要进行线程安全处理，这里在asyncRun()方法的前面加上synchronized同步修饰符。

至此，多线程版本的Reactor模式实战案例的代码介绍完毕，可以开始执行新版本的多线程MultiThreadEchoServerReactor服务器。当然，也可以执行之前的EchoClient客户端程序，完成整个回显的通信演示。

由于演示程序的输出结果与前面单线程版本的EchoServer运行输出是一模一样的，因此这里不再贴出程序的执行结果。

4.4 Reactor模式的优缺点

在总结Reactor模式的优点和缺点之前，先看看Reactor模式和其他模式的对比，加强一下对它的理解。

(1) Reactor模式和生产者消费者模式对比

二者的相似之处：在一定程度上，Reactor模式有点类似生产者消费者模式。在生产者消费者模式中，一个或多个生产者将事件加入一个队列中，一个或多个消费者主动从这个队列中拉取（Pull）事件来处理。

二者的不同之处：Reactor模式是基于查询的，没有专门的队列去缓冲存储IO事件，查询到IO事件之后，反应器会根据不同IO选择键（事件）将其分发给对应的Handler来处理。

(2) Reactor模式和观察者模式对比

二者的相似之处：在Reactor模式中，当查询到IO事件后，服务处理程序使用单路/多路分发（Dispatch）策略，同步分发这些IO事件。观察者模式（Observer Pattern）也被称作发布/订阅模式，它定义了一种依赖关系，让多个观察者同时监听某一个主题（Topic）。这个主题对象在状态发生变化时会通知所有观察者，它们能够执行相应的处理。

二者的不同之处：在Reactor模式中，Handler实例和IO事件（选择键）的订阅关系基本上是一个事件绑定到一个Handler，每一个IO事件（选择键）被查询后，反应器会将事件分发给所绑定的Handler，也

就是一个事件只能被一个Handler处理；在观察者模式中，同一时刻、同一主题可以被订阅过的多个观察者处理。

最后，总结一下Reactor模式的优点和缺点。作为高性能的IO模式，Reactor模式的优点如下：

- 响应快，虽然同一反应器线程本身是同步的，但是不会被单个连接的IO操作所阻塞。
- 编程相对简单，最大限度避免了复杂的多线程同步，也避免了多线程各个进程之间切换的开销。
- 可扩展，可以方便地通过增加反应器线程的个数来充分利用CPU资源。

Reactor模式的缺点如下：

- Reactor模式增加了一定的复杂性，因而有一定的门槛，并且不容易调试。
- Reactor模式依赖于操作系统底层的IO多路复用系统调用的支持，如Linux中的epoll系统调用。如果操作系统的底层不支持IO多路复用，Reactor模式不会那么高效。
- 在同一个Handler业务线程中，如果出现一个长时间的数据读写，就会影响这个反应器中其他通道的IO处理。例如，在大文件传输时，IO操作就会影响其他客户端的响应时间。对于这种操作，还需要进一步对Reactor模式进行改进。

第5章 Netty核心原理与基础实战

Netty是一个Java NIO客户端/服务器框架，是一个为了快速开发可维护的高性能、高可扩展的网络服务器和客户端程序而提供的异步事件驱动基础框架和工具。基于Netty，可以快速轻松地开发网络服务器和客户端的应用程序。与直接使用Java NIO相比，Netty给大家造出了一个非常优美的轮子，它可以大大简化网络编程流程。例如，Netty极大地简化了TCP、UDP套接字和HTTP Web服务程序的开发。

Netty的目标之一是使通信开发可以做到“快速和轻松”。使用Netty，除了能“快速和轻松”地开发TCP/UDP等自定义协议的通信程序之外，还可以做到“快速和轻松”地开发应用层协议的通信程序，如FTP、SMTP、HTTP以及其他的传统应用层协议。

Netty的目标之二是要做到高性能、高可扩展性。基于Java的NIO，Netty设计了一套优秀的、高性能的Reactor模式实现，并且基于Netty的Reactor模式实现中的Channel（通道）、Handler（处理器）等基础类库能进行快速扩展，以支持不同协议通信、完成不同业务处理的大量应用类。

5.1 第一个Netty实战案例DiscardServer

在开始介绍Netty核心原理之前，首先为大家介绍一个非常简单的入门实战案例，这是一个丢弃服务器（DiscardServer）的简单通信案例，其作用类似于学习Java基础编程时的“Hello World”程序。

在开始编写实战案例之前，需要准备Netty的版本，并且配置好开发环境。

5.1.1 创建第一个Netty项目

首先我们需要创建项目（或者模块），这里取名为NettyDemos。第一个Netty的实战案例DiscardServer就在这个项目中进行实战开发。DiscardServer功能很简单：读取客户端的输入数据，直接丢弃，不给客户端任何回复。

在使用Netty前，需要考虑一下JDK的版本，Netty官方建议使用JDK 1.6以上，本书使用的是JDK 1.8。然后是Netty自己的版本，建议使用Netty 4.0以上的版本，本书使用的Netty版本是4.1.6。

使用maven导入Netty的依赖坐标到工程（或项目），Netty的依赖坐标如下：

```
<dependency>
<groupId>io.netty</groupId>
<artifactId>netty-all</artifactId>
```

```
<version>4.1.6.Final</version>  
</dependency>
```

说明

Netty版本在不断升级，但是4.0以上的版本使用比较广泛。Netty曾经升级到5.0，不过出现了一些问题，版本又回退了。另外，很多的大数据开源框架使用的还是3.0版本的Netty。从学习的角度来看，关键是学习其核心原理和编程技巧。理解原理之后，在实际开发过程中，根据具体的版本，看看其源码或者API手册即可。

准备好项目工程之后，就可以正式开始编写第一个Netty程序了。

5.1.2 第一个Netty服务端程序

创建一个服务端类NettyDiscardServer，用以实现消息的Discard（丢弃）功能，源代码如下：

```
package com.crazymakercircle.netty.basic;  
//...  
public class NettyDiscardServer {  
    private final int serverPort;  
    ServerBootstrap b = new ServerBootstrap();  
    public NettyDiscardServer(int port) {  
        this.serverPort = port;
```

```
}

public void runServer() {
    //创建反应器轮询组

    EventLoopGroup bossLoopGroup = new
        NioEventLoopGroup(1);

    EventLoopGroup workerLoopGroup = new
        NioEventLoopGroup();

    try {
        //1. 设置反应器轮询组

        b.group(bossLoopGroup, workerLoopGroup);

        //2. 设置nio类型的通道

        b.channel(NioServerSocketChannel.class);

        //3. 设置监听端口

        b.localAddress(serverPort);

        //4. 设置通道的参数

        b.option(ChannelOption.SO_KEEPALIVE, true);

        //5. 装配子通道流水线

        b.childHandler(new ChannelInitializer<SocketChannel>
        () {
            //有连接到达时会创建一个通道

            protected void initChannel(SocketChannel ch) {
                //流水线的职责：负责管理通道中的处理器

                //向“子通道”（传输通道）流水线添加一个处理器

                ch.pipeline().addLast(new
                    NettyDiscardHandler()));
            }
        });
    }
}
```

```
//6. 开始绑定服务器
//通过调用sync同步方法阻塞直到绑定成功
ChannelFuture channelFuture = b.bind().sync();
Logger.info(" 服务器启动成功，监听端口：" +
channelFuture.channel().localAddress());
//7. 等待通道关闭的异步任务结束
//服务监听通道会一直等待通道关闭的异步任务结束
ChannelFuture closeFuture =
channelFuture.channel().closeFuture();
closeFuture.sync();
} catch (Exception e) {
e.printStackTrace();
} finally {
//8. 优雅关闭EventLoopGroup
//释放掉所有资源，包括创建的线程
workerLoopGroup.shutdownGracefully();
bossLoopGroup.shutdownGracefully();
}
}
public static void main(String[] args) {
int port = NettyDemoConfig.SOCKET_SERVER_PORT;
new NettyDiscardServer(port).runServer();
}
}
```

如果是第一次看Netty应用程序的代码，那么上面的代码应用是晦涩难懂的，因为代码中涉及很多Netty专用组件。不过不要紧，因为Netty是基于Reactor模式实现的。通过前面的章节学习，大家已经非常深入地了解了Reactor模式，所以现在只需要顺藤摸瓜理清楚Netty的Reactor模式对应的组件，Netty的核心组件结构就相对简单了。

首先要说的是Reactor模式中的Reactor组件。前面讲到，反应器组件的作用是进行IO事件的查询和分发。Netty中对应的反应器组件有多种，不同应用通信场景用到的反应器组件各不相同。一般来说，对应于多线程的Java NIO通信的应用场景，Netty对应的反应器组件为NioEventLoopGroup。

在上面的例子中，使用了两个NioEventLoopGroup反应器组件实例：第一个负责服务器通道新连接的IO事件的监听，可以形象地理解为“包工头”角色；第二个主要负责传输通道的IO事件的处理和数据传输，可以形象地理解为“工人”角色。

其次要说的是Reactor模式中的Handler（处理器）角色组件。Handler的作用是对应到IO事件，完成IO事件的业务处理。Handler需要为业务做专门开发，下一小节将对上面的NettyDiscardHandler自定义处理器进行介绍。

再次，在上面的例子中还用到了Netty的服务引导类ServerBootstrap。服务引导类是一个组装和集成器，职责是将不同的Netty组件组装在一起。此外，ServerBootstrap能够按照应用场景的需要为组件设置好基础性的参数，最后帮助快速实现Netty服务器的监听和启动。服务引导类ServerBootstrap也是本章重点之一，后面将对其进行详细的介绍。

5.1.3 业务处理器NettyDiscardHandler

在Reactor模式中，所有的业务处理都在Handler中完成，业务处理一般需要自己编写，这里编写一个新类：NettyDiscardHandler。这里的业务处理很简单：把收到的任何内容直接丢弃，也不会回复任何消息。

NettyDiscardHandler的代码如下：

```
package com.crazymakercircle.netty.basic;  
//...  
NettyDiscardHandler extends ChannelInboundHandlerAdapter {  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object msg) {  
        ByteBuf in = (ByteBuf) msg;  
        try {  
            Logger.info("收到消息，丢弃如下：");  
            while (in.isReadable()) {  
                System.out.print((char) in.readByte());  
            }  
            System.out.println(); //换行  
        } finally {  
            ReferenceCountUtil.release(msg);  
        }  
    }  
}
```

Netty的Handler需要处理多种IO事件（如读就绪、写就绪），对应于不同的IO事件，Netty提供了一些基础方法。这些方法都已经提前封装好，应用程序直接继承或者实现即可。比如说，对于处理入站的IO事件，其对应的接口为ChannelInboundHandler，并且Netty提供了ChannelInboundHandlerAdapter适配器作为入站处理器的默认实现。

说明

这里将引入一组新的概念：入站和出站。简单理解，入站指的是输入，出站指的是输出。后面也会有详细介绍。Netty中的出/入站与Java NIO中的出/入站有些微妙的不同，Netty的出站可以理解为从Handler传递到Channel的操作，比如说write写通道、read读通道数据；Netty的入站可以理解为从Channel传递到Handler的操作，比如说Channel数据过来之后，会触发Handler的channelRead()入站处理方法。

如果要实现自己的入站处理器，可以简单地继承ChannelInboundHandlerAdapter入站处理器适配器，再写入自己的入站处理的业务逻辑。也就是说，重写通道读取方法channelRead()即可。

在上面例子中的channelRead()方法将Netty缓冲区ByteBuf的输入数据打印到服务端控制台后，直接丢弃不管了，而且不给客户端任何回复。

Netty的ByteBuf缓冲区组件（后面会单独对其进行详细的介绍）可以对应到前面介绍的Java NIO类库的数据缓冲区Buffer组件。只不过相对而言，Netty的ByteBuf缓冲区性能更好，使用也更加方便。

5.1.4 运行NettyDiscardServer

在上面的例子中出现了Netty中的各种组件：服务器引导类、缓冲区、反应器、业务处理器、Future异步回调、数据传输通道等。这些Netty组件都是需要掌握的，也都是我们在后面需要进行专项学习的。

说明

Future异步回调或者同步阻塞是高并发开发频繁使用到的技术，所以有关Future异步回调或者同步阻塞的原理和知识是非常重要的，具体请参阅《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》的相关内容。

如果看不懂以上NettyDiscardServer程序，没有关系。此程序的目的只是为大家展示一下Netty开发中会涉及什么内容，给大家留一个初步的印象。接下来，大家可以启动NettyDiscardServer服务器来体验一下Netty程序的运行。

在源代码工程中找到消息丢弃服务器类NettyDiscardServer，启动它的main()方法，就启动了这个服务器应用。

如果想看到最终的“消息丢弃”执行效果，不能仅仅启动服务器，还需要启动客户端，需要从客户端向服务器发送消息。这里的客户端只要能通过TCP与服务器建立Socket连接即可，不一定是使用Netty编写的客户端程序，可以是Java IO或者NIO客户端。因此，直接使用前面章节中的EchoClient程序作为客户端程序即可，因为所使用的TCP通信端口是一致的。

在源代码工程中，我们可以找到发送消息到服务器的客户端类：EchoClient。通过启动它的main()方法，就可以启动这个客户端程序。然后在客户端的标准化输入窗口不断输入要发送的消息，发送到服务器即可，在服务端可以看到所打印的丢弃了的消息。

虽然EchoClient客户端是使用Java NIO编写的，而NettyDiscardServer服务端是使用Netty编写的，但是不影响它们之间的相互通信。不仅仅是因为底层Netty框架也是使用Java NIO开发的，更加核心的原因是都使用了TCP通信协议。

5.2 解密Netty中的Reactor模式

在前面的章节中，已经反复说明：设计模式是Java代码或者程序的重要组织方式，如果不了解设计模式，学习和阅读Java程序代码往往找不到头绪，上下求索而不得其法。故而，在学习Netty组件之前，我们必须了解Netty中的Reactor模式是如何实现的。

这里，先回顾一下Java NIO中I0事件的处理流程和Reactor模式的基础内容。

5.2.1 回顾Reactor模式中I0事件的处理流程

一个I0事件从操作系统底层产生后，在Reactor模式中的处理流程如图5-1所示。

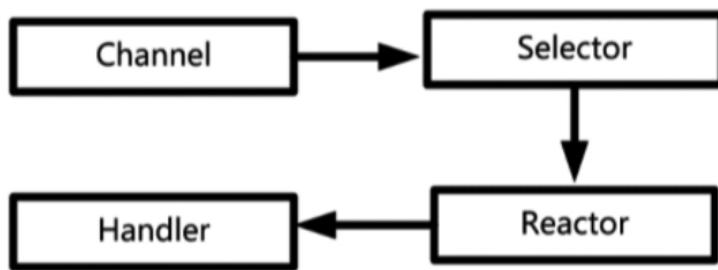


图5-1 Java Reactor模式中I0事件的处理流程

Reactor模式中I0事件的处理流程大致分为4步，具体如下：

第1步：通道注册。I0事件源于通道（Channel），I0是和通道（对于底层连接而言）强相关的。一个I0事件一定属于某个通道。如果要查询通道的事件，首先就要将通道注册到选择器。

第2步：查询事件。在Reactor模式中，一个线程会负责一个反应器（或者SubReactor子反应器），不断地轮询，查询选择器中的I/O事件（选择键）。

第3步：事件分发。如果查询到I/O事件，则分发给与I/O事件有绑定关系的Handler业务处理器。

第4步：完成真正的I/O操作和业务处理，这一步由Handler业务处理器负责。

以上4步就是整个Reactor模式的I/O处理器流程。其中，第1步和第2步其实是Java NIO的功能，Reactor模式仅仅是利用了Java NIO的优势而已。

说明

Reactor模式的I/O事件处理流程比较重要，是学习Netty的基础性和铺垫性知识。如果这里看不懂，就先回到前面有关Reactor模式详细介绍的部分内容，回头再学习一下Reactor模式原理。

5.2.2 Netty中的Channel

Channel组件是Netty中非常重要的组件，为什么首先要说的是Channel组件呢？原因是：Reactor模式和通道紧密相关，反应器的查询和分发的I/O事件都来自Channel组件。

Netty中不直接使用Java NIO的Channel组件，对Channel组件进行了自己的封装。Netty实现了一系列的Channel组件，为了支持多种通信协议，换句话说，对于每一种通信连接协议，Netty都实现了自己的通道。除了Java的NIO，Netty还提供了Java面向流的OIO处理通道。

总结起来，对应到不同的协议，Netty实现了对应的通道，每一种协议基本上都有NIO和OIO两个版本。

对应于不同的协议，Netty中常见的通道类型如下：

- **NioSocketChannel**: 异步非阻塞TCP Socket传输通道。
- **NioServerSocketChannel**: 异步非阻塞TCP Socket服务端监听通道。
- **NioDatagramChannel**: 异步非阻塞的UDP传输通道。
- **NioSctpChannel**: 异步非阻塞Sctp传输通道。
- **NioSctpServerChannel**: 异步非阻塞Sctp服务端监听通道。
- **OioSocketChannel**: 同步阻塞式TCP Socket传输通道。
- **OioServerSocketChannel**: 同步阻塞式TCP Socket服务端监听通道。
- **OioDatagramChannel**: 同步阻塞式UDP传输通道。
- **OioSctpChannel**: 同步阻塞式Sctp传输通道。
- **OioSctpServerChannel**: 同步阻塞式Sctp服务端监听通道。

一般来说，服务端编程用到最多的通信协议还是TCP，对应的Netty传输通道类型为NioSocketChannel类、Netty服务器监听通道类型为NioServerSocketChannel。不论是哪种通道类型，在主要的API和使用方式上和NioSocketChannel类基本都是相同的，更多是底层的传

输协议不同，而Netty帮大家极大地屏蔽了传输差异。如果没有特殊情况，本书的很多案例都将以NioSocketChannel通道为主。

在Netty的NioSocketChannel内部封装了一个Java NIO的SelectableChannel成员，通过对该内部的Java NIO通道的封装，对Netty的NioSocketChannel通道上的所有IO操作最终都会落地到Java NIO的SelectableChannel底层通道。NioSocketChannel的继承关系图如图5-2所示。

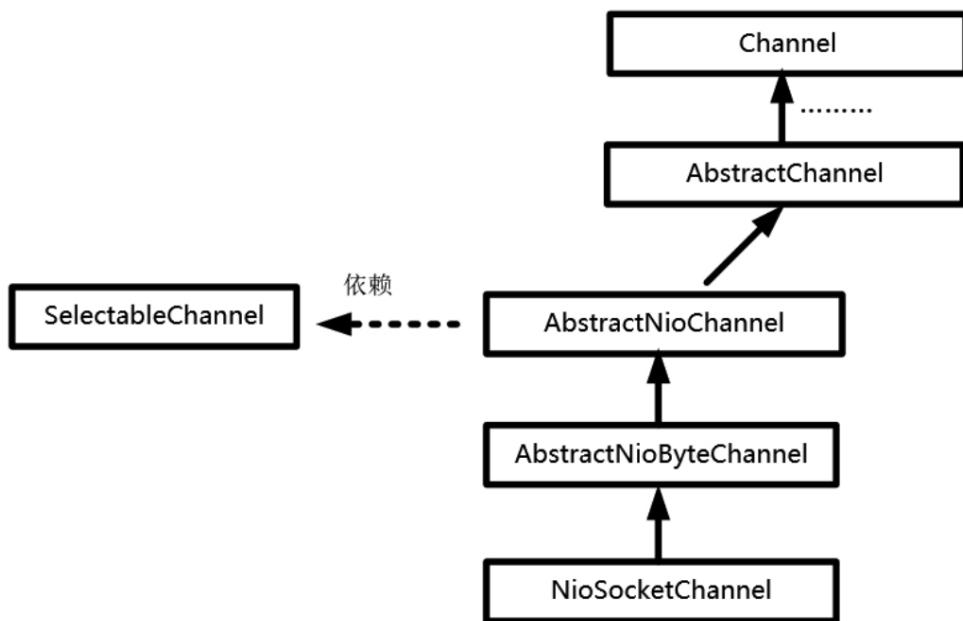


图5-2 NioSocketChannel的继承关系图

5.2.3 Netty中的Reactor

在Reactor模式中，一个反应器（或者SubReactor子反应器）会由一个事件处理线程负责事件查询和分发。该线程不断进行轮询，通过Selector选择器不断查询注册过的IO事件（选择键）。如果查询到IO事件，就分发给Handler业务处理器。

首先为大家介绍一下Netty中的反应器组件。Netty中的反应器组件有多个实现类，这些实现类与其通道类型相互匹配。对于NioSocketChannel通道，Netty的反应器类为NioEventLoop（NIO事件轮询）。

NioEventLoop类有两个重要的成员属性：一个是Thread线程类的成员，一个是Java NIO选择器的成员属性。NioEventLoop的继承关系和主要成员属性如图5-3所示。

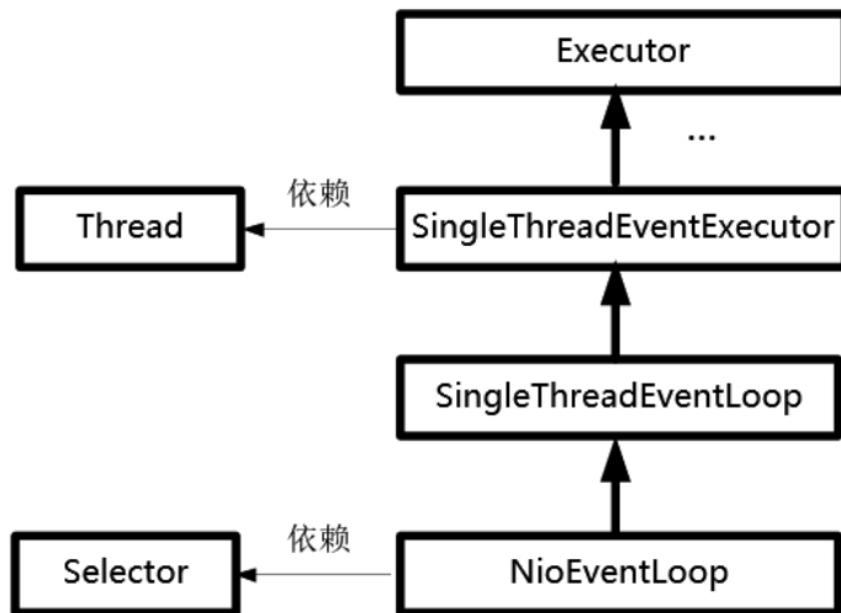


图5-3 NioEventLoop的继承关系和主要成员属性

通过这个关系图可以看出：NioEventLoop和前面章节讲的反应器实现在思路上是一致的：一个NioEventLoop拥有一个线程，负责一个Java NIO选择器的IO事件轮询。

在Netty中，EventLoop反应器和Channel的关系是什么呢？理论上来说，一个EventLoop反应器和NettyChannel通道是一对多的关系：一

一个反应器可以注册成千上万的通道，如图5-4所示。

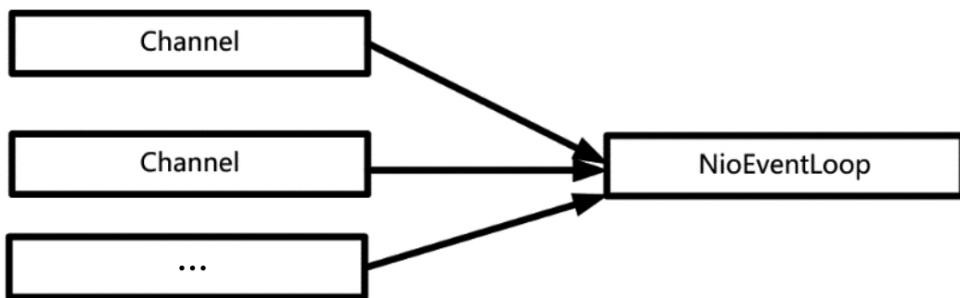


图5-4 EventLoop反应器和Channel的关系

5.2.4 Netty中的Handler

在前面的章节介绍Java NIO的IO事件类型时讲到，可供选择器监控的通道IO事件类型包括以下4种：

- 可读：SelectionKey.OP_READ。
- 可写：SelectionKey.OP_WRITE。
- 连接：SelectionKey.OP_CONNECT。
- 接收：SelectionKey.OP_ACCEPT。

在Netty中，EventLoop反应器内部有一个线程负责Java NIO选择器的事件的轮询，然后进行对应的事件分发。事件分发（Dispatch）的目标就是Netty的Handler（含用户定义的业务处理器）。

Netty的Handler分为两大类：第一类是ChannelInboundHandler入站处理器；第二类是ChannelOutboundHandler出站处理器，二者都继承了ChannelHandler处理器接口。有关Handler的接口与继承关系如图5-5所示。

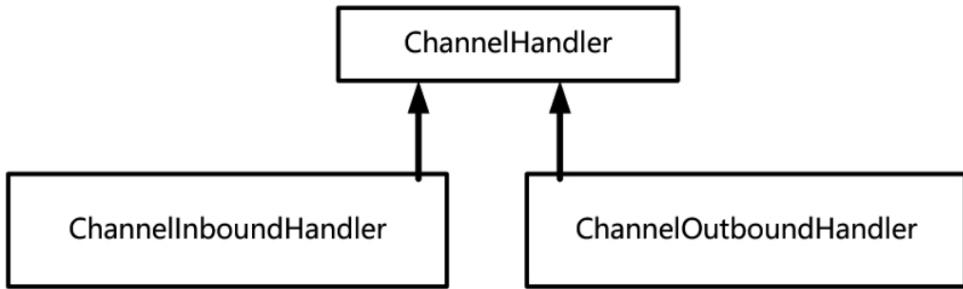


图5-5 Netty中的Handler的接口与继承关系

Netty入站处理的流程是什么呢？以底层的Java NIO中的OP_READ输入事件为例：在通道中发生了OP_READ事件后，会被EventLoop查询到，然后分发给ChannelInboundHandler入站处理器，调用对应的入站处理的read()方法。在ChannelInboundHandler入站处理器内部的read()方法具体实现中，可以从通道中读取数据。

Netty中的入站处理触发的方向为从通道触发，ChannelInboundHandler入站处理器负责接收（或者执行）。Netty中的入站处理不仅仅是OP_READ输入事件的处理，还包括从通道底层触发，由Netty通过层层传递，调用ChannelInboundHandler入站处理器进行的其他某个处理。

Netty中的出站处理指的是从ChannelOutboundHandler出站处理器到通道的某次IO操作。例如，在应用程序完成业务处理后，可以通过ChannelOutboundHandler出站处理器将处理的结果写入底层通道。最常用的一个方法就是write()方法，即把数据写入通道。

Netty中的出站处理不仅仅包括Java NIO的OP_WRITE可写事件，还包括Netty自身从处理器到通道方向的其他操作。OP_WRITE可写事件是Java NIO的概念，和Netty的出站处理在概念上不是一个维度，Netty的出站处理是应用层维度的。

无论是入站还是出站，Netty都提供了各自的默认适配器实现：
ChannelInboundHandler的默认实现为
ChannelInboundHandlerAdapter（入站处理适配器）。
ChannelOutboundHandler的默认实现为
ChannelOutboundHandlerAdapter（出站处理适配器）。这两个默认的通道处理适配器分别实现了基本的入站操作和出站操作功能。如果要实现自己的业务处理器，不需要从零开始去实现处理器的接口，只需要继承通道处理适配器即可。

5.2.5 Netty中的Pipeline

在介绍Netty的Pipeline事件处理流水线之前，先梳理一下Netty的Reactor模式实现中各个组件之间的关系：

(1) 反应器（或者SubReactor子反应器）和通道之间是一对多的关系：一个反应器可以查询很多个通道的IO事件。

(2) 通道和Handler处理器实例之间是多对多的关系：一个通道的IO事件可以被多个Handler实例处理；一个Handler处理器实例也能绑定到很多通道，处理多个通道的IO事件。

问题是：通道和Handler处理器实例之间的绑定关系，Netty是如何组织的呢？

Netty设计了一个特殊的组件，叫作ChannelPipeline（通道流水线）。它像一条管道，将绑定到一个通道的多个Handler处理器实例串联在一起，形成一条流水线。ChannelPipeline的默认实现实际上被设

计成一个双向链表。所有的Handler处理器实例被包装成双向链表的节点，被加入到ChannelPipeline中。

说明

一个Netty通道拥有一个ChannelPipeline类型的成员属性，该属性的名称叫作pipeline。

以入站处理为例，每一个来自通道的IO事件都会进入一次ChannelPipeline。在进入第一个Handler处理器后，这个IO事件将按照既定的从前往后次序，在流水线上不断地向后流动，流向下一个Handler处理器。

在向后流动的过程中，会出现3种情况：

(1) 如果后面还有其他Handler入站处理器，那么IO事件可以交给下一个Handler处理器向后流动。

(2) 如果后面没有其他的入站处理器，就意味着这个IO事件在此次流水线中的处理结束了。

(3) 如果在中间需要终止流动，可以选择不将IO事件交给下一个Handler处理器，流水线的执行也被终止了。

Netty的通道流水线与普通的流水线不同，Netty的流水线不是单向的，而是双向的，而普通的流水线基本都是单向的。Netty是这样规定的：入站处理器的执行次序是从前到后，出站处理器的执行次序是

从后到前。总之，IO事件在流水线上的执行次序与IO事件的类型是有关系的，如图5-6所示。

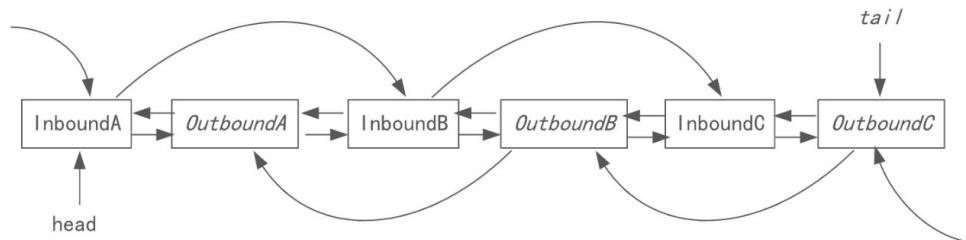


图5-6 流水线上入站处理器和出站处理器的执行次序

除了流动的方向与IO操作类型有关之外，流动过程中所经过的处理器类型也是与IO操作的类型有关的。入站的IO操作只能从Inbound入站处理器类型的Handler流过；出站的IO操作只能从Outbound出站处理器类型的Handler流过。

至此，在了解完流水线之后，大家应该对Netty中的通道、EventLoop反应器、处理器，以及三者之间的协作关系，有了一个清晰的认知和了解，基本可以动手开发简单的Netty程序了。为了方便开发者，Netty提供了一系列辅助类，用于把上面的三个组件快速组装起来完成一个Netty应用，这个系列的类叫作引导类。服务端的引导类叫作ServerBootstrap类，客户端的引导类叫作Bootstrap类。

接下来，为大家详细介绍一下这些能提升开发效率的Bootstrap。

5.3 详解Bootstrap

Bootstrap类是Netty提供的一个便利的工厂类，可以通过它来完成Netty的客户端或服务端的Netty组件的组装，以及Netty程序的初始化和启动执行。Netty的官方解释是，完全可以不用这个Bootstrap类，可以一点点去手动创建通道、完成各种设置和启动注册到EventLoop反应器，然后开始事件的轮询和处理，但是这个过程会非常麻烦。通常情况下，使用这个便利的Bootstrap工具类的效率会更高。

在Netty中有两个引导类，分别用于服务器和客户端，如图5-7所示。

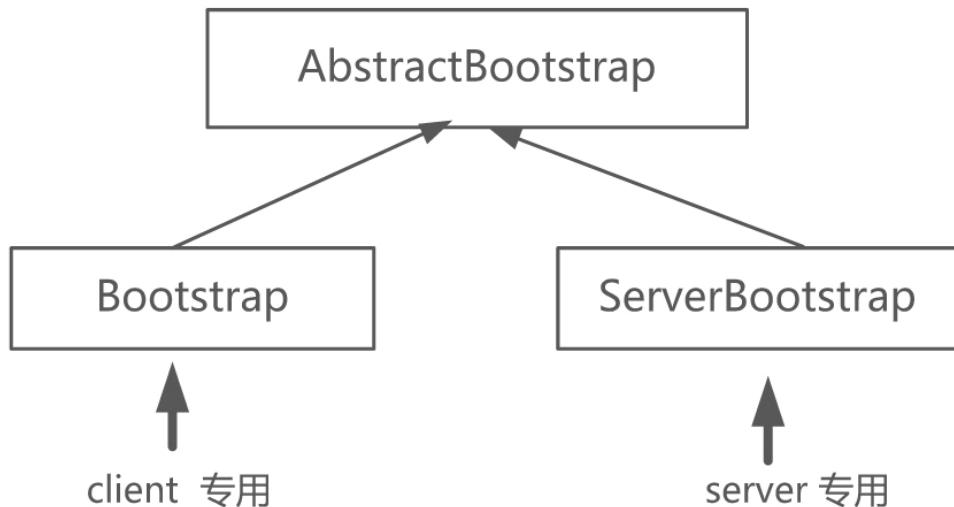


图5-7 Netty中的两个引导类

这两个引导类仅是使用的地方不同，它们大致的配置和使用方法都是相同的。下面以ServerBootstrap类作为重点介绍对象。

在介绍ServerBootstrap的服务器启动流程之前，首先介绍一下涉及的两个基础概念：父子通道、EventLoopGroup（事件轮询线程组）。

5.3.1 父子通道

在Netty中，每一个NioSocketChannel通道所封装的都是Java NIO通道，再往下就对应到了操作系统底层的socket文件描述符。理论上来说，操作系统底层的socket文件描述符分为两类：

- 连接监听类型。连接监听类型的socket描述符处于服务端，负责接收客户端的套接字连接；在服务端，一个“连接监听类型”的socket描述符可以接受（Accept）成千上万的传输类的socket文件描述符。
- 数据传输类型。数据传输类的socket描述符负责传输数据。同一条TCP的Socket传输链路在服务器和客户端都分别会有一个与之相对应的数据传输类型的socket文件描述符。

在Netty中，异步非阻塞的服务端监听通道NioServerSocketChannel所封装的Linux底层的文件描述符是“连接监听类型”的socket描述符；异步非阻塞的传输通道NioSocketChannel所封装的Linux的文件描述符是“数据传输类型”的socket描述符。

在Netty中，将有接收关系的监听通道和传输通道叫作父子通道。其中，负责服务器连接监听和接收的监听通道（如NioServerSocketChannel）也叫父通道（Parent Channel），对应于

每一个接收到的传输类通道（如NioSocketChannel）也叫子通道（Child Channel）。

5.3.2 EventLoopGroup

在前面介绍Reactor模式的具体实现时，分为单线程实现版本和多线程实现版本。Netty中的Reactor模式实现不是单线程版本的，而是多线程版本的。

实际上，在Netty中一个EventLoop相当于一个子反应器（SubReactor），一个NioEventLoop子反应器拥有了一个事件轮询线程，同时拥有一个Java NIO选择器。

Netty是如何完成多线程版本的Reactor模式实现的呢？答案是使用EventLoopGroup（事件轮询组）。多个EventLoop线程放在一起，可以组成一个EventLoopGroup。反过来说，EventLoopGroup就是一个多线程版本的反应器，其中的单个EventLoop线程对应于一个子反应器（SubReactor）。

Netty的程序开发不会直接使用单个EventLoop（事件轮询器），而是使用EventLoopGroup。EventLoopGroup的构造函数有一个参数，用于指定内部的线程数。在构造器初始化时，会按照传入的线程数量在内部构造多个线程和多个EventLoop子反应器（一个线程对应一个EventLoop子反应器），进行多线程的I/O事件查询和分发。

如果使用EventLoopGroup的无参数构造函数，没有传入线程数量或者传入的数量为0，那么EventLoopGroup内部的线程数量到底是多少呢？默认的EventLoopGroup内部线程数量为最大可用的CPU处理器数量

的2倍。假设电脑使用的是4核的CPU，那么在内部会启动8个EventLoop线程，相当于8个子反应器实例。

从前文可知，为了及时接收新连接，在服务端，一般有两个独立的反应器，一个负责新连接的监听和接收，另一个负责I/O事件轮询和分发，并且两个反应器相互隔离。对应到Netty服务器程序中，则需要设置两个EventLoopGroup，一个组负责新连接的监听和接受，另外一组负责I/O传输事件的轮询与分发，两个轮询组的职责具体如下：

(1) 负责新连接的监听和接收的EventLoopGroup中的反应器完成查询通道的新连接I/O事件查询。这些反应器有点像负责招工的包工头，因此，该轮询组可以形象地称为“包工头”(Boss)轮询组。

(2) 负责I/O事件轮询和分发的反应器完成查询所有子通道的I/O事件，并且执行对应的Handler处理器完成I/O处理——例如数据的输入和输出(有点儿像搬砖)，这个轮询组可以形象地称为“工人”(Worker)轮询组。

Netty的EventLoopGroup与EventLoop之间、EventLoop与Channel之间的关系如图5-8所示。

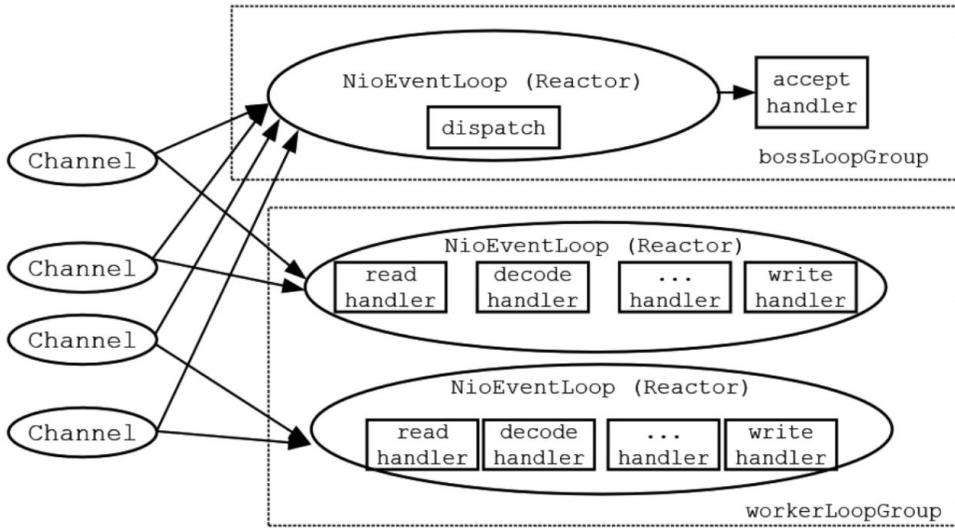


图5-8 Netty中的Reactor模式示意图

至此，介绍完了两个重要的基础概念：父子通道与EventLoopGroup。有了这些基础知识作为铺垫，接下来可以正式介绍ServerBootstrap的启动流程了。

5.3.3 Bootstrap启动流程

Bootstrap的启动流程也就是Netty组件的组装、配置，以及Netty服务器或者客户端的启动流程。在本节中对启动流程进行了梳理，大致分成8个步骤。本书仅仅演示的是服务端引导类的使用，用到的引导类为ServerBootstrap。正式使用前，首先创建一个服务端的引导类实例。

```
//创建一个服务端的引导类
ServerBootstrap b = new ServerBootstrap();
```

接下来，结合前面的NettyDiscardServer服务器的程序代码，给大家详细介绍一下Bootstrap启动流程中精彩的8个步骤。

第1步：创建反应器轮询组，并设置到ServerBootstrap引导类实例，大致的代码如下：

```
//创建反应器轮询组  
//boss轮询组  
EventLoopGroup bossLoopGroup = new NioEventLoopGroup(1);  
//worker轮询组  
EventLoopGroup workerLoopGroup = new NioEventLoopGroup();  
//...  
//step1: 为引导类实例设置反应器轮询组  
b.group(bossLoopGroup, workerLoopGroup);
```

在设置反应器轮询组之前，创建了两个NioEventLoopGroup，一个负责处理连接监听IO事件，名为bossLoopGroup；另一个负责数据传输事件和处理，名为workerLoopGroup。在两个轮询组创建完成后，就可以配置给引导类实例，它一次性地给引导类配置了两大轮询组。

如果不需要分开监听新连接事件和输出事件，就不一定非得配置两个轮询组，可以仅配置一个EventLoopGroup反应器轮询组。具体的配置方法是调用b.group(workerGroup)。在这种模式下，新连接监听IO事件和数据传输IO事件可能被挤在了同一个线程中处理。这样会带来一个风险：新连接的接收被更加耗时的数据传输或者业务处理所阻塞。所以，在服务端，建议设置成两个轮询组的工作模式。

第2步：设置通道的IO类型。Netty不止支持Java NIO，也支持阻塞式的OIO。下面配置的是Java NIO类型的通道类型：

```
//step2: 设置传输通道的类型为NIO类型  
b.channel(NioServerSocketChannel.class);
```

如果确实指定Bootstrap的IO模型为BIO类型，可以配置为
OioServerSocketChannel.class类。NIO的优势巨大，因此通常不会在
Netty中使用BIO。

第3步：设置监听端口，代码大致如下：

```
//step3: 设置监听端口  
b.localAddress(new InetSocketAddress(port));
```

这是最为简单的一部操作，主要是设置服务器的监听地址。

第4步：设置传输通道的配置选项，代码大致如下：

```
//step4: 设置通道的参数  
b.option(ChannelOption.SO_KEEPALIVE, true);  
b.option(ChannelOption.ALLOCATOR,  
PooledByteBufAllocator.DEFAULT);
```

这里调用了Bootstrap的option()选项设置方法。对于服务器的
Bootstrap而言，这个方法的作用是：给父通道（Parent Channel）设
置一些与传输协议相关的选项。如果要给子通道（Child Channel）设
置一些通道选项，则需要调用childOption()设置方法。

可以设置哪些通道选项（ChannelOption）呢？在上面的代码中，设置了一个底层TCP相关的选项ChannelOption.SO_KEEPALIVE。该选项表示是否开启TCP底层心跳机制，true为开启，false为关闭。其他的通道设置选项，参见下一小节。

第5步：装配子通道的Pipeline。每一个通道都用一条ChannelPipeline流水线，它的内部有一个双向的链表。装配流水线的方式是：将业务处理器ChannelHandler实例包装之后加入双向链表中。

如何装配Pipeline流水线呢？装配子通道的Handler流水线调用引导类的childHandler()方法，该方法需要传入一个ChannelInitializer通道初始化类的实例作为参数。每当父通道成功接收到一个连接并创建成功一个子通道后，就会初始化子通道，此时这里配置的ChannelInitializer实例就会被调用。

在ChannelInitializer通道初始化类的实例中，有一个initChannel初始化方法，在子通道创建后会被执行，向子通道流水线增加业务处理器。

装配子通道的Pipeline流水线的大致代码如下：

```
//step5: 装配子通道流水线
b.childHandler(new ChannelInitializer<SocketChannel>() {
    //有连接到达时会创建一个通道的子通道，并初始化
    protected void initChannel(SocketChannel ch) ...{
        //这里可以管理子通道中的Handler业务处理器
        //向子通道流水线添加一个Handler业务处理器
    }
})
```

```
        ch.pipeline().addLast(new NettyDiscardHandler());  
    }  
});
```

为什么仅装配子通道的流水线，而不需要装配父通道的流水线呢？原因是：父通道（NioServerSocketChannel）的内部业务处理是固定的：接收新连接后，创建子通道，然后初始化子通道，所以不需要特别的配置，由Netty自行进行装配。如果需要完成特殊的父通道业务处理，可以类似地调用ServerBootstrap的handler(ChannelHandler handler)方法，为父通道设置ChannelInitializer初始化器。

在装配流水线时需要注意的是，ChannelInitializer处理器有一个泛型参数SocketChannel，它代表需要初始化的通道类型，这个类型需要和前面的引导类中设置的传输通道类型一一对应起来。

第6步：开始绑定服务器新连接的监听端口，代码大致如下：

```
//step6: 开始绑定端口，通过调用sync()同步方法阻塞直到绑定成功  
ChannelFuture channelFuture = b.bind().sync();  
Logger.info(" 服务器启动成功，监听端口：" +  
channelFuture.channel().localAddress());
```

这个也很简单。b.bind()方法的功能是返回一个端口绑定Netty的异步任务channelFuture。在这里，并没有给channelFuture异步任务增加回调监听器，而是阻塞channelFuture异步任务，直到端口绑定任务执行完成。

在Netty中，所有的IO操作都是异步执行的，这就意味着任何一个IO操作都会立即返回，返回时异步任务还没有真正执行。什么时候执

行完成呢？Netty中的IO操作都会返回异步任务实例（如channelFuture实例）。通过该异步任务实例，既可以实现同步阻塞一直到channelFuture异步任务执行完成，也可以通过为其增加事件监听器的方式注册异步回调逻辑，以获得Netty中的IO操作的真正结果。上面所使用的是同步阻塞一直到channelFuture异步任务执行完成的处理方式。

至此，服务器正式启动。

说明

Future异步回调或者同步阻塞，涉及高并发的核心模式——异步回调模式，是高并发开发非常重要的基础性知识，具体请参阅《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》的相关内容。

第7步：自我阻塞，直到监听通道关闭，代码大致如下：

```
//step7: 自我阻塞，直到通道关闭的异步任务结束
ChannelFuture closeFuture =
channelFuture.channel().closeFuture();
closeFuture.sync();
```

如果要阻塞当前线程直到通道关闭，可以调用通道的closeFuture()方法，以获取通道关闭的异步任务。当通道被关闭时，closeFuture实例的sync()方法会返回。

第8步：关闭EventLoopGroup，代码大致如下：

```
//step8: 释放掉所有资源，包括创建的反应器线程  
workerLoopGroup.shutdownGracefully();  
bossLoopGroup.shutdownGracefully();
```

关闭反应器轮询组，同时会关闭内部的子反应器线程，也会关闭内部的选择器、内部的轮询线程以及负责查询的所有子通道。在子通道关闭后，会释放掉底层的资源，如Socket文件描述符等。

5.3.4 ChannelOption

无论是对于NioServerSocketChannel父通道类型还是对于NioSocketChannel子通道类型，都可以设置一系列的ChannelOption（通道选项）。ChannelOption类中定义了一系列选项，下面介绍一些常见的选项。

1. SO_RCVBUF和SO_SNDBUF

这两个为TCP传输选项，每个TCP socket（套接字）在内核中都有一个发送缓冲区和一个接收缓冲区，这两个选项就是用来设置TCP连接的两个缓冲区大小的。TCP的全双工工作模式以及TCP的滑动窗口对两个独立的缓冲区都有依赖。

2. TCP_NODELAY

此为TCP传输选项，如果设置为true就表示立即发送数据。TCP_NODELAY用于开启或关闭Nagle算法。如果要求高实时性，有数据发送时就马上发送，就将该选项设置为true（关闭Nagle算法）；如果

要减少发送次数、减少网络交互，就设置为false（开启Nagle算法），等累积一定大小的数据后再发送。关于TCP_NODELAY的值，Netty默认为true，而操作系统默认为false。

Nagle算法将小的碎片数据连接成更大的报文（或数据包）来最小化所发送报文的数量，如果需要发送一些较小的报文，则需要禁用该算法。

Netty默认禁用Nagle算法，报文会立即发送出去，从而最小化报文传输的延时。

说明

TCP_NODELAY的值设置为true表示关闭延迟，设置为false表示开启延迟。其值与是否开启Nagle算法是相反的。

3. SO_KEEPALIVE

此为TCP传输选项，表示是否开启TCP的心跳机制。true为连接保持心跳，默认值为false。启用该功能时，TCP会主动探测空闲连接的有效性。需要注意的是：默认的心跳间隔是7200秒，即2小时。Netty默认关闭该功能。

4. SO_REUSEADDR

此为TCP传输选项，为true时表示地址复用，默认值为false。有四种情况需要用到这个参数设置：

- 当有一个地址和端口相同的连接socket1处于TIME_WAIT状态时，而又希望启动一个新的连接socket2要占用该地址和端口。
- 有多块网卡或用IP Alias技术的机器在同一端口启动多个进程，但每个进程绑定的本地IP地址不能相同。
- 同一进程绑定相同的端口到多个socket（套接字）上，但每个socket绑定的IP地址不同。
- 完全相同的地址和端口的重复绑定，但这只用于UDP的多播，不用于TCP。

说明

Socket连接状态（如TIME_WAIT）和连接建立时三次握手以及断开时四次挥手有关，请参阅本书后面有关TCP协议原理的内容。

5. SO_LINGER

此为TCP传输选项，可以用来控制socket.close()方法被调用后的行为，包括延迟关闭时间。如果此选项设置为-1，就表示socket.close()方法在调用后立即返回，但操作系统底层会将发送缓冲区的数据全部发送到对端；如果此选项设置为0，就表示socket.close()方法在调用后会立即返回，但是操作系统会放弃发送缓冲区数据，直接向对端发送RST包，对端将收到复位错误；如果此选项设置为非0整数值，就表示调用socket.close()方法的线程被阻塞，直到延迟时间到来，发送缓冲区中的数据发送完毕，若超时，则对端会收到复位错误。

SO_LINGER的默认值为-1，表示禁用该功能。

6. SO_BACKLOG

此为TCP传输选项，表示服务端接收连接的队列长度，如果队列已满，客户端连接将被拒绝。服务端在处理客户端新连接请求时（三次握手）是顺序处理的，所以同一时间只能处理一个客户端连接，多个客户端到来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，队列的大小通过SO_BACKLOG指定。

具体来说，服务端对完成第二次握手的连接放在一个队列（暂时称A队列），如果进一步完成第三次握手，再把连接从A队列移动到新队列（暂时称B队列），接下来应用程序会通过调用accept()方法取出握手成功的连接，而系统则会将该连接从B队列移除。A和B队列的长度之和是SO_BACKLOG指定的值，当A和B队列的长度之和大于SO_BACKLOG值时，新连接将会被TCP内核拒绝。所以，如果SO_BACKLOG过小，accept速度可能会跟不上，A和B队列全满，导致新客户端无法连接。

说明

SO_BACKLOG对程序支持的连接数并无影响，影响的只是还没有被accept取出的连接数，也就是三次握手的排队连接数。

如果连接建立频繁，服务器处理新连接较慢，那么可以适当调大这个参数。

7. SO_BROADCAST

此为TCP传输选项，表示设置为广播模式。

5.4 详解Channel

本节首先为大家介绍一下Channel（通道）的主要成员和方法，然后为大家介绍一下Netty所提供的一个专门的单元测试通道——EmbeddedChannel（嵌入式通道）。

5.4.1 Channel的主要成员和方法

通道是Netty的核心概念之一，代表网络连接，由它负责同对端进行网络通信，既可以写入数据到对端，也可以从对端读取数据。

Netty通道的抽象类AbstractChannel的构造函数如下：

```
protected AbstractChannel(Channel parent) {  
    this.parent = parent; //父通道  
    id = newId();  
    unsafe = newUnsafe(); //新建一个底层的NIO 通道，完成实际的IO  
操作  
    pipeline = newChannelPipeline(); //新建一条通道流水线  
}
```

AbstractChannel内部有一个pipeline属性，表示处理器的流水线。Netty在对通道进行初始化的时候，将pipeline属性初始化为DefaultChannelPipeline的实例。以上代码表明每个通道都拥有一条ChannelPipeline处理器流水线。

AbstractChannel内部有一个parent父通道属性，保持通道的父通道。对于连接监听通道（如NioServerSocketChannel）来说，其parent属性的值为null；对于传输通道（如NioSocketChannel）来说，其parent属性的值为接收到该连接的监听通道。

几乎所有的Netty通道实现类都继承了AbstractChannel抽象类，都拥有上面的parent和pipeline两个属性成员。

接下来，介绍一下通道接口中所定义的几个重要方法。

(1) ChannelFuture connect(SocketAddress address)

此方法的作用为连接远程服务器。方法的参数为远程服务器的地址，调用后会立即返回，其返回值为执行连接操作的异步任务ChannelFuture。此方法在客户端的传输通道使用。

(2) ChannelFuture bind(SocketAddress address)

此方法的作用为绑定监听地址，开始监听新的客户端连接。此方法在服务器的新连接监听和接收通道时调用。

(3) ChannelFuture close()

此方法的作用为关闭通道连接，返回连接关闭的ChannelFuture异步任务。如果需要在连接正式关闭后执行其他操作，则需要为异步任务设置回调方法；或者调用ChannelFuture异步任务的sync()方法来阻塞当前线程，一直等到通道关闭的异步任务执行完毕。

(4) Channel read()

此方法的作用为读取通道数据，并且启动入站处理。具体来说，从内部的Java NIO Channel通道读取数据，然后启动内部的Pipeline流水线，开启数据读取的入站处理。此方法的返回通道自身用于链式调用。

(5) ChannelFuture write (Object o)

此方法的作用为启程出站流水处理，把处理后的最终数据写到底层通道（如Java NIO通道）。此方法的返回值为出站处理的异步处理任务。

(6) Channel flush()

此方法的作用为将缓冲区中的数据立即写出到对端。调用前面的write()出站处理时，并不能将数据直接写出到对端，write操作的作用在大部分情况下仅仅是写入操作系统的缓冲区，操作系统会根据缓冲区的情况决定什么时候把数据写到对端。执行flush()方法会立即将缓冲区的数据写到对端。

上面的6种方法仅仅是常见的通道方法。在Channel接口中以及各种通道的实现类中还定义了大量的通道操作方法。在一般的日常开发中，如果需要用到，请直接查阅Netty API文档或者Netty源代码。

5.4.2 EmbeddedChannel

在Netty的实际开发中，底层通信传输的基础工作Netty已经替大家完成。实际上，更多的工作是设计和开发ChannelHandler业务处理

器。处理器开发完成后，需要投入单元测试。一般单元测试的大致流程是：先将Handler业务处理器加入到通道的Pipeline流水线中，接下来先后启动Netty服务器、客户端程序，相互发送消息，测试业务处理器的效果。这些复杂的工序存在一个问题：如果每开发一个业务处理器都进行服务器和客户端的重复启动，那么整个的过程是非常烦琐和浪费时间的。如何解决这种徒劳、低效的重复工作呢？Netty提供了一个专用通道，即EmbeddedChannel（嵌入式通道）。

EmbeddedChannel仅仅是模拟入站与出站的操作，底层不进行实际传输，不需要启动Netty服务器和客户端。除了不进行传输之外，EmbeddedChannel的其他事件机制和处理流程和真正的传输通道是一模一样的。因此，使用EmbeddedChannel，开发人员可以在单元测试用例中方便、快速地进行ChannelHandler业务处理器的单元测试。

为了模拟数据的发送和接收，EmbeddedChannel提供了一组专门的方法，具体如表5-1所示。

表5-1 EmbeddedChannel单元测试的辅助方法

名 称	说 明
writeInbound()	向通道写入入站数据，模拟真实通道收到数据的场景。也就是说，这些写入的数据会被流水线上的入站处理器所处理到
readInbound()	从 EmbeddedChannel 中读取入站数据，返回经过流水线最后一个入站处理器处理完成之后的入站数据。如果没有数据，则返回 null
writeOutbound()	向通道写入出站数据，模拟真实通道发送数据。也就是说，这些写入的数据会被流水线上的出站处理器处理
readOutbound()	从 EmbeddedChannel 中读取出站数据，返回经过流水线最后一个出站处理器处理之后的出站数据。如果没有数据，则返回 null
finish()	结束 EmbeddedChannel，它会调用通道的 close()方法

最为重要的两个方法为writeInbound()和writeOutbound()方法。

(1) writeInbound()

它的使用场景是测试入站处理器。在测试入站处理器时（例如测试一个解码器），需要读取入站（Inbound）数据。可以调用writeInbound()方法，向EmbeddedChannel写入一个入站数据（如二进制ByteBuf数据包），模拟底层的入站包，从而被入站处理器处理到，达到测试的目的。

(2) writeOutbound()

它的使用场景是测试出站处理器。在测试出站处理器时（例如测试一个编码器），需要有出站（Outbound）的数据进入流水线。可以调用writeOutbound()方法，向模拟通道写入一个出站数据（如二进制ByteBuf数据包），该包将进入处理器流水线，被待测试的出站处理器所处理。

总之，EmbeddedChannel类既拥有通道的通用接口和方法，又增加了一些单元测试的辅助方法，在开发时是非常有用的。有关它的具体用法，后面还会结合其他Netty组件的实例反复提到。

5.5 详解Handler

在Reactor经典模型中，反应器查询到IO事件后会分发到Handler业务处理器，由Handler完成IO操作和业务处理。

整个IO处理操作环节大致包括从通道读数据包、数据包解码、业务处理、目标数据编码、把数据包写到通道，然后由通道发送到对端，如图5-9所示。

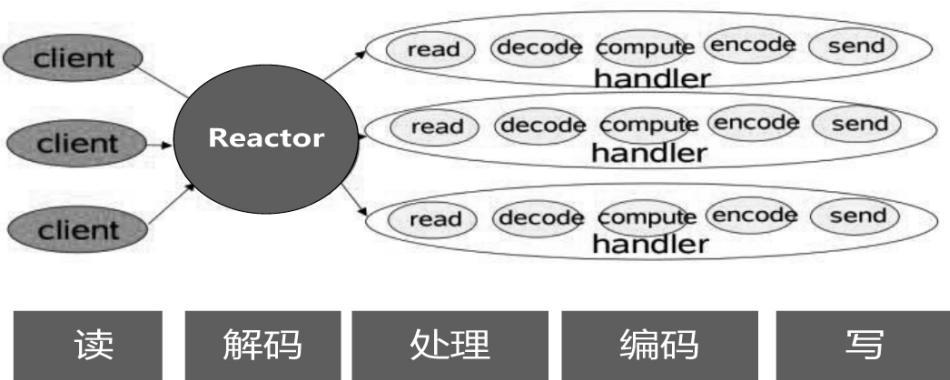


图5-9 整个的IO处理操作环节

整个的IO处理操作环节的前后两个环节（包括从通道读数据包和由通道发送到对端），由Netty的底层负责完成，不需要用户程序负责。

用户程序主要涉及的Handler环节为数据包解码、业务处理、目标数据编码、把数据包写到通道中。

前面已经介绍过，从应用程序开发人员的角度来看有入站和出站两种类型操作。

- 入站处理触发的方向为自底向上，从Netty的内部（如通道）到ChannelInboundHandler入站处理器。
- 出站处理触发的方向为自顶向下，从ChannelOutboundHandler出站处理器到Netty的内部（如通道）。

按照这种触发方向来区分，IO处理操作环节前面的数据包解码、业务处理两个环节属于入站处理器的工作；后面目标数据编码、把数据包写到通道中两个环节属于出站处理器的工作。

5.5.1 ChannelInboundHandler入站处理器

当对端数据入站到Netty通道时，Netty将触发ChannelInboundHandler入站处理器所对应的入站API，进行入站操作处理。ChannelInboundHandler的主要操作如图5-10所示。

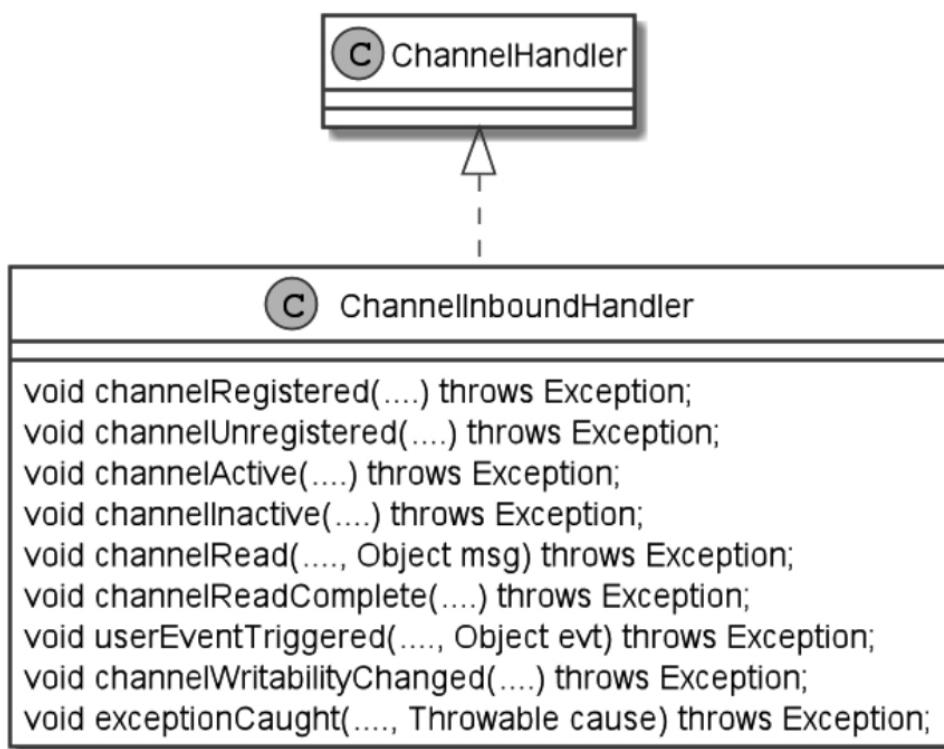


图5-10 ChannelInboundHandler的主要操作

对于ChannelInboundHandler的核心方法，大致介绍如下：

1. channelRegistered()

当通道注册完成后，Netty会调用fireChannelRegistered()方法，触发通道注册事件，而在通道流水线注册过的入站处理器的channelRegistered()回调方法会被调用。

2. channelActive()

当通道激活完成后，Netty会调用fireChannelActive()方法，触发通道激活事件，而在通道流水线注册过的入站处理器的channelActive()回调方法会被调用。

3. channelRead()

当通道缓冲区可读时，Netty会调用fireChannelRead()方法，触发通道可读事件，而在通道流水线注册过的入站处理器的channelRead()回调方法会被调用，以便完成入站数据的读取和处理。

4. channelReadComplete()

当通道缓冲区读完时，Netty会调用fireChannelReadComplete()方法，触发通道缓冲区读完事件，而在通道流水线注册过的入站处理器的channelReadComplete()回调方法会被调用。

5. channelInactive()

当连接被断开或者不可用时，Netty会调用fireChannelInactive()方法，触发连接不可用事件，而在通道流水线注册过的入站处理器的channelInactive()回调方法会被调用。

6. exceptionCaught()

当通道处理过程发生异常时，Netty会调用fireExceptionCaught()方法，触发异常捕获事件，而在通道流水线注册过的入站处理器的exceptionCaught()方法会被调用。注意，这个方法是在ChannelHandler中定义的方法，入站处理器、出站处理器接口都继承了该方法。

上面介绍的并不是ChannelInboundHandler的全部方法，仅仅介绍了其中几种比较重要的方法。在Netty中，入站处理器的默认实现为ChannelInboundHandlerAdapter，在实际开发中只需要继承ChannelInboundHandlerAdapter默认实现，重写自己需要的回调方法即可。

5.5.2 ChannelOutboundHandler出站处理器

当业务处理完成后，需要操作Java NIO底层通道时，通过一系列的ChannelOutboundHandler出站处理器完成Netty通道到底层通道的操作，比如建立底层连接、断开底层连接、写入底层Java NIO通道等。ChannelOutboundHandler接口定义了大部分的出站操作，如图5-11所示。

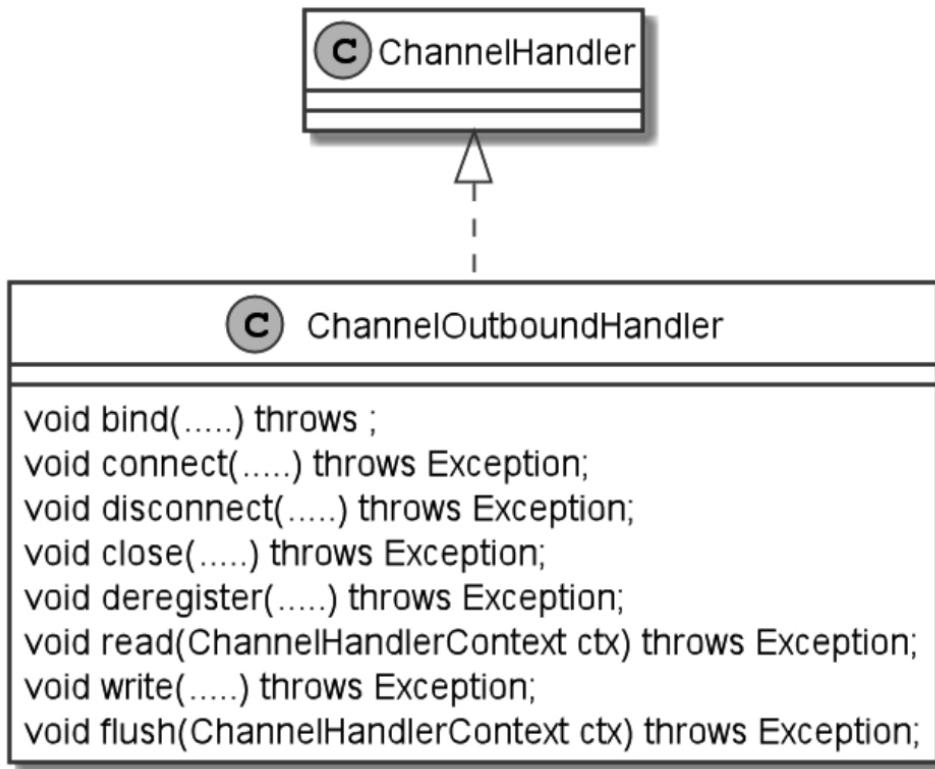


图5-11 ChannelOutboundHandler的主要操作

再强调一下，Netty出站处理的方向是通过上层Netty通道去操作底层Java I/O通道，主要出站（Outbound）的操作如下：

(1) bind()

监听地址（IP+端口）绑定：完成底层Java I/O通道的IP地址绑定。如果使用TCP传输协议，这个方法用于服务端。

(2) connect()

连接服务端：完成底层Java I/O通道的服务端的连接操作。如果使用TCP传输协议，那么这个方法将用于客户端。

(3) write()

写数据到底层：完成Netty通道向底层Java IO通道的数据写入操作。此方法仅仅是触发一下操作，并不是完成实际的数据写入操作。

(4) flush()

将底层缓存区的数据腾空，立即写出到对端。

(5) read ()

从底层读数据：完成Netty通道从Java IO通道的数据读取。

(6) disConnect()

断开服务器连接：断开底层Java IO通道的socket连接。如果使用TCP传输协议，此方法主要用于客户端。

(7) close()

主动关闭通道：关闭底层的通道，例如服务端的新连接监听通道。

上面介绍的并不是ChannelOutboundHandler的全部方法，仅仅介绍了其中几个比较重要的方法。在Netty中，它的默认实现为ChannelOutboundHandlerAdapter。在实际开发中，只需要继承ChannelOutboundHandlerAdapter默认实现，重写自己需要的方法即可。

5.5.3 ChannelInitializer通道初始化处理器

在前面已经讲到，Channel和Handler业务处理器的关系是：一条Netty的通道拥有一条Handler业务处理器流水线，负责装配自己的Handler业务处理器。装配Handler的工作发生在通道开始工作之前。现在的问题是：如果向流水线中装配业务处理器呢？这就得借助通道的初始化处理器——ChannelInitializer。

首先回顾一下NettyDiscardServer丢弃服务端的代码，在给接收到的新连接装配Handler业务处理器时，调用childHandler()方法设置了一个ChannelInitializer实例：

```
//step5: 装配子通道流水线
b.childHandler(new ChannelInitializer<SocketChannel>() {
    //有连接到达时会创建一个通道的子通道，并初始化
    protected void initChannel(SocketChannel ch) ...{
        //这里可以管理子通道中的Handler业务处理器
        //向子通道流水线添加一个Handler业务处理器
        ch.pipeline().addLast(new NettyDiscardHandler());
    }
});
```

上面的ChannelInitializer也是通道初始化器，属于入站处理器的类型。在示例代码中，使用了ChannelInitializer的initChannel()方法。initChannel()方法是ChannelInitializer定义的一个抽象方法，这个抽象方法需要开发人员自己实现。

在通道初始化时，会调用提前注册的初始化处理器的initChannel()方法。比如，在父通道接收到新连接并且要初始化其子

通道时，会调用初始化器的initChannel()方法，并且会将新接收的通道作为参数，传递给此方法。

一般来说，initChannel()方法的大致业务代码是：拿到新连接通道作为实际参数，往它的流水线中装配Handler业务处理器。

5.5.4 ChannelInboundHandler的生命周期的实战案例

为了弄清Handler业务处理器的各个方法的执行顺序和生命周期，这里定义一个简单的入站Handler处理器——InHandlerDemo。这个类继承于ChannelInboundHandlerAdapter适配器，实现了基类的大部分入站处理方法，并在每一个方法的实现代码中都加上必要的输出信息，以便于观察方法是否被执行到。

InHandlerDemo的代码如下：

```
package com.crazymakercircle.netty.handler;  
//...  
public class InHandlerDemo extends ChannelInboundHandlerAdapter  
{  
    @Override  
    public void handlerAdded(ChannelHandlerContext ctx)...{  
        Logger.info("被调用: handlerAdded()");  
        super.handlerAdded(ctx);  
    }  
    @Override
```

```
public void channelRegistered(ChannelHandlerContext ctx)...{
    Logger.info("被调用: channelRegistered()");
    super.channelRegistered(ctx);
}

@Override
public void channelActive(ChannelHandlerContext ctx)...{
    Logger.info("被调用: channelActive()");
    super.channelActive(ctx);
}

@Override
public void channelRead(ChannelHandlerContext ctx, Object msg)...{
    Logger.info("被调用: channelRead()");
    super.channelRead(ctx, msg);
}

@Override
public void channelReadComplete(ChannelHandlerContext ctx)...
{
    Logger.info("被调用: channelReadComplete()");
    super.channelReadComplete(ctx);
}

@Override
public void channelInactive(ChannelHandlerContext ctx)...{
    Logger.info("被调用: channelInactive()");
    super.channelInactive(ctx);
}

@Override
```

```

public void channelUnregistered(ChannelHandlerContext ctx) ...
{
    Logger.info("被调用: channelUnregistered()");
    super.channelUnregistered(ctx);
}

@Override
public void handlerRemoved(ChannelHandlerContext ctx)...{
    Logger.info("被调用: handlerRemoved()");
    super.handlerRemoved(ctx);
}
}

```

为了演示这个入站处理器，需要编写一个单元测试代码：将上面的Inhandler入站处理器加入一个EmbeddedChannel嵌入式通道的流水线中。接着，通过writeInbound()方法写入ByteBuf数据包。InHandlerDemo作为一个入站处理器，会处理到流水线上的入站报文。单元测试的代码如下：

```

package com.crazymakercircle.netty.handler;
//省略import

public class InHandlerDemoTester {
    @Test
    public void testInHandlerLifeCircle() {
        final InHandler DemoInHandler = new InHandlerDemo();
        //初始化处理器
        ChannelInitializer i =
new ChannelInitializer<EmbeddedChannel>()

```

```

{
    protected void initChannel(EmbeddedChannel ch) {
        ch.pipeline().addLast(inHandler);
    }
};

//创建嵌入式通道
EmbeddedChannel channel = new EmbeddedChannel(i);
ByteBuf buf = Unpooled.buffer();
buf.writeInt(1);
//模拟入站，向嵌入式通道写一个入站数据包
channel.writeInbound(buf);
channel.flush();
//模拟入站，再写一个入站数据包
channel.writeInbound(buf);
channel.flush();
//通道关闭
channel.close();
//...
}
}

```

运行上面的测试用例，主要的输出结果具体如下：

```

[main|handlerAdded]: 被调用: handlerAdded()
[main|channelRegistered]: 被调用: channelRegistered()
[main|channelActive]: 被调用: channelActive()
[main|channelRead]: 被调用: channelRead()

```

```
[main|channelReadComplete]: 被调用: channelReadComplete()  
[main|channelRead]: 被调用: channelRead()  
[main|channelReadComplete]: 被调用: channelReadComplete()  
[main|channelInactive]: 被调用: channelInactive()  
[main|channelUnregistered]: 被调用: channelUnregistered()  
[main|handlerRemoved]: 被调用: handlerRemoved()
```

在讲解上面的方法之前，首先对处理器的方法进行分类：是生命周期方法还是数据入站回调方法。上面的几个方法中，channelRead、channelReadComplete是入站处理方法；而其他的6个方法是入站处理器的周期方法。从输出的结果可以看到，ChannelHandler中回调方法的执行顺序为：

handlerAdded() → channelRegistered() → channelActive() → 数据传输的入站回调 → channelInactive() → channelUnregistered() → handlerRemoved()

其中，数据传输的入站回调过程为：

channelRead() → channelReadComplete()

读数据的入站回调过程会根据入站数据的数量被重复调用，每一次有ByteBuf数据包入站都会调用到。

除了两个入站回调方法外，其余的6个方法都和ChannelHandler的生命周期有关，具体的介绍如下：

(1) handlerAdded(): 当业务处理器被加入到流水线后，此方法将被回调。也就是在完成ch.pipeline().addLast(handler)语句之后会回调handlerAdded()。

(2) channelRegistered(): 当通道成功绑定一个NioEventLoop反应器后，此方法将被回调。

(3) channelActive(): 当通道激活成功后，此方法将被回调。通道激活成功指的是所有的业务处理器添加、注册的异步任务完成，并且与NioEventLoop反应器绑定的异步任务完成。

(4) channelInactive(): 当通道的底层连接已经不是ESTABLISH状态或者底层连接已经关闭时，会首先回调所有业务处理器的channelInactive()方法。

(5) channelUnregistered(): 通道和NioEventLoop反应器解除绑定，移除掉对这条通道的事件处理之后，回调所有业务处理器的channelUnregistered()方法。

(6) handlerRemoved(): Netty会移除掉通道上所有的业务处理器，并且回调所有业务处理器的handlerRemoved()方法。

在上面的6个生命周期方法中，前面3个在通道创建和绑定时被先后回调，后面3个在通道关闭时会先后被回调。

除了生命周期的回调，还有数据传输的入站回调方法。对于Inhandler入站处理器，有两个很重要的回调方法：

(1) channelRead(): 有数据包入站，通道可读。流水线会启动入站处理流程，从前向后，入站处理器的channelRead()方法会被依次回调到。

(2) channelReadComplete(): 流水线完成入站处理后，会从前向后依次回调每个入站处理器的channelReadComplete()方法，表示数

据读取完毕。

至此，大家对ChannelInboundHandler的生命周期和入站业务处理应该有了一个非常清楚的了解。

上面的入站处理器实战案例InHandlerDemo演示的是入站处理器的工作流程。对于出站处理器ChannelOutboundHandler的生命周期以及回调的顺序，与入站处理器的顺序是大致相同的。不同的是，出站处理器的业务处理方法略微不同。在随书源代码工程中，有一个关于出站处理器的实战案例——OutHandlerDemo。它的代码、包名和上面的类似，大家可以自己去运行和学习，这里不再赘述。

5.6 详解Pipeline

前面讲到，一条Netty通道需要很多业务处理器来处理业务。每条通道内部都有一条流水线（Pipeline）将Handler装配起来。Netty的业务处理器流水线ChannelPipeline是基于责任链设计模式（Chain of Responsibility）来设计的，内部是一个双向链表结构，能够支持动态地添加和删除业务处理器。

5.6.1 Pipeline入站处理流程

为了完整地演示Pipeline入站处理流程，将新建三个极为简单的入站处理器：SimpleInHandlerA、SimpleInHandlerB、SimpleInHandlerC。在ChannelInitializer处理器的initChannel方法中，把它们加入到流水线中。添加的顺序为A→B→C。实战的代码如下：

```
package com.crazymakercircle.netty.pipeline;  
//...  
  
public class InPipeline {  
    //内部类：第一个入站处理器  
    static class SimpleInHandlerA extends  
        ChannelInboundHandlerAdapter {  
        @Override  
        public void channelRead(ChannelHandlerContext ctx, Object  
msg) {...  
        Logger.info("入站处理器 A: 被回调 ");  
    }  
}
```

```
        super.channelRead(ctx, msg);

    }

}

//内部类：第二个入站处理器

static class SimpleInHandlerB extends
ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object
msg)...{

        Logger.info("入站处理器 B: 被回调 ");

        super.channelRead(ctx, msg);

    }

}

//内部类：第三个入站处理器

static class SimpleInHandlerC extends
ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
...{

        Logger.info("入站处理器 C: 被回调 ");

        super.channelRead(ctx, msg);

    }

}

@Test

public void testPipelineInBound() {

    ChannelInitializer i =

new ChannelInitializer<EmbeddedChannel>() {
```

```
protected void initChannel(EmbeddedChannel ch) {  
    ch.pipeline().addLast(new SimpleInHandlerA());  
    ch.pipeline().addLast(new SimpleInHandlerB());  
    ch.pipeline().addLast(new SimpleInHandlerC());  
}  
};  
  
EmbeddedChannel channel = new EmbeddedChannel(i);  
ByteBuf buf = Unpooled.buffer();  
buf.writeInt(1);  
//向通道写一个入站报文（数据包）  
channel.writeInbound(buf);  
//省略不相关代码  
}  
}
```

在以上三个内部入站处理器的channelRead()方法中，我们打印当前Handler业务处理器的信息，然后调用父类的channelRead()方法，而父类的channelRead()方法的主要作用是把当前入站处理器中处理完毕的结果传递到下一个入站处理器。只是在示例程序中传递的对象都是同一个数据（也就是程序中的msg实例）。

运行实战案例的代码，输出的结果如下：

```
[main|InPipeline$SimpleInHandlerA:channelRead]: 入站处理器 A: 被回调  
[main|InPipeline$SimpleInHandlerB:channelRead]: 入站处理器 B: 被回调
```

[main|InPipeline\$SimpleInHandlerC:channelRead]: 入站处理器 C: 被回调

我们可以看到，入站处理器的流动次序是从前到后，如图5-12所示。

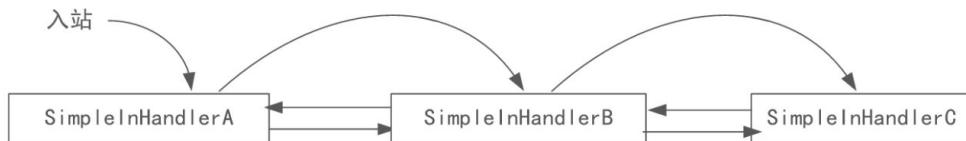


图5-12 入站处理器的执行次序

疑问：在入站处理器的channelRead()方法中，如果不调用父类的channelRead()方法，结果会如何呢？大家可以自行尝试。

5.6.2 Pipeline出站处理流程

为了完整地演示Pipeline出站处理流程，将新建三个极为简单的出站处理器：SimpleOutHandlerA、SimpleOutHandlerB、SimpleOutHandlerC。在ChannelInitializer处理器的initChannel()方法中，把它们加入到流水线中，添加的顺序为A→B→C。实战案例的代码如下：

```
package com.crazymakercircle.netty.pipeline;  
//...  
public class OutPipeline {  
    //内部类：第一个出站处理器  
    public class SimpleOutHandlerA extends  
        ChannelOutboundHandlerAdapter {
```

```
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)...
    {
        Logger.info("出站处理器 A: 被回调" );
        super.write(ctx, msg, promise);
    }
}

//内部类: 第二个出站处理器
public class SimpleOutHandlerB extends
ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)...
    {
        Logger.info("出站处理器 B: 被回调" );
        super.write(ctx, msg, promise);
    }
}

//内部类: 第三个出站处理器
public class SimpleOutHandlerC extends
ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)...
    {
        Logger.info("出站处理器 C: 被回调" );
        super.write(ctx, msg, promise);
    }
}
```

```
}

@Test
public void testPipelineOutBound() {
    ChannelInitializer i =
    new ChannelInitializer<EmbeddedChannel>() {
        protected void initChannel(EmbeddedChannel ch) {
            ch.pipeline().addLast(new SimpleOutHandlerA());
            ch.pipeline().addLast(new SimpleOutHandlerB());
            ch.pipeline().addLast(new SimpleOutHandlerC());
        }
    };
    EmbeddedChannel channel = new EmbeddedChannel(i);
    ByteBuf buf = Unpooled.buffer();
    buf.writeInt(1);
    //向通道写入一个出站报文(或数据包)
    channel.writeOutbound(buf);
    //省略不相关代码
}
}
```

在以上出站处理器的write()方法中，打印当前Handler业务处理器的信息，然后调用父类的write()方法，而这里父类的write()方法会将出站数据通过通道流水线发送到下一个出站处理器。运行上面的实战案例程序，控制台的输出如下：

```
[main|OutPipeline$SimpleOutHandlerC:write]: 出站处理器 C: 被回调
[main|OutPipeline$SimpleOutHandlerB:write]: 出站处理器 B: 被回调
```

[main|OutPipeline\$SimpleOutHandlerA:write]: 出站处理器 A: 被回调

在代码中，通过pipeline.addLast()方法添加OutBoundHandler出站处理器的顺序为A→B→C。从结果可以看出，出站流水处理次序为从后向前（C→B→A），最后加入的出站处理器反而执行在最前面如图5-13所示。这一点和Inbound入站处理的次序是恰好相反的。

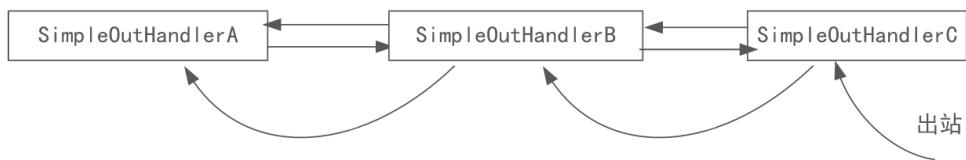


图5-13 出站处理器的执行次序

疑问：在出站处理器的write()方法中，如果不调用父类的write()方法，结果会如何呢？大家可以自行尝试和体验。

5.6.3 ChannelHandlerContext

在Netty的设计中Handler是无状态的，不保存和Channel有关的信息。Handler的目标是将自己的处理逻辑做得很通用，可以给不同的Channel使用。与Handler不同的是，Pipeline是有状态的，保存了Channel的关系。于是，Handler和Pipeline之间需要一个中间角色将它们联系起来。这个中间角色是谁呢？ChannelHandlerContext（通道处理器上下文）！

不管我们定义的是哪种类型的业务处理器，最终它们都是以双向链表的方式保存在流水线中。这里流水线的节点类型并不是前面的业务处理器基类，而是其包装类型ChannelHandlerContext类。当业务处理器被添加到流水线中时会为其专门创建一个ChannelHandlerContext

实例，主要封装了ChannelHandler（通道处理器）和ChannelPipeline（通道流水线）之间的关联关系。所以，流水线ChannelPipeline中的双向链接实质是一个由ChannelHandlerContext组成的双向链表。作为Context的成员，无状态的Handler关联在ChannelHandlerContext中。

ChannelPipeline流水线的示意图大致如图5-14所示。

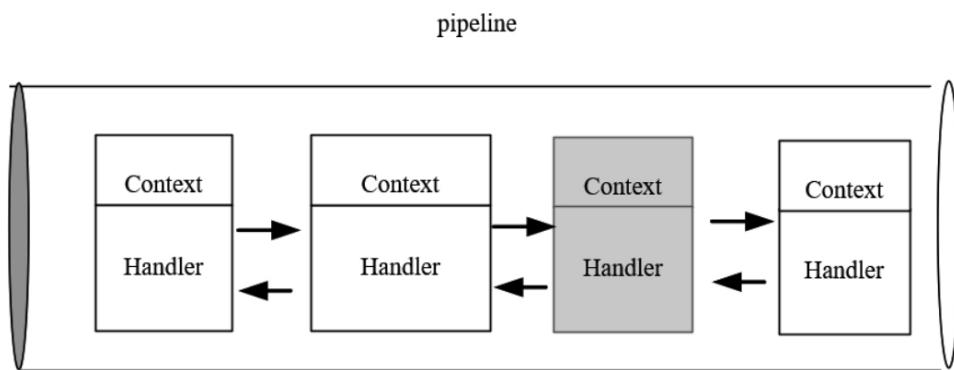


图5-14 ChannelPipeline流水线的示意图

ChannelHandlerContext中包含了许多方法，主要可以分为两类：第一类是获取上下文所关联的Netty组件实例，如所关联的通道、所关联的流水线、上下文内部Handler业务处理器实例等；第二类是入站和出站处理方法。

在Channel、ChannelPipeline、ChannelHandlerContext三个类中，都存在同样的出站和入站处理方法，这些出现在不同的类中的相同方法，功能有何不同呢？

如果通过Channel或ChannelPipeline的实例来调用这些出站和入站处理方法，它们就会在整条流水线中传播。如果是通过ChannelHandlerContext调用出站和入站处理方法，就只会从当前的节

点开始往同类型的下一站处理器传播，而不是在整条流水线从头至尾进行完整的传播。

总结一下Channel、Handler、ChannelHandlerContext三者的关系：Channel拥有一条ChannelPipeline，每一个流水线节点为一个ChannelHandlerContext上下文对象，每一个上下文中包裹了一个ChannelHandler。在ChannelHandler的入站/出站处理方法中，Netty会传递一个Context实例作为实际参数。处理器中的回调代码可以通过Context实参，在业务处理过程中去获取ChannelPipeline实例或者Channel实例。

5.6.4 HeadContext与TailContext

通道流水线在没有加入任何处理器之前装配了两个默认的处理器上下文：一个头部上下文HeadContext，一个尾部上下文TailContext。pipeline的创建、初始化除了保存一些必要的属性外，核心就在于创建了HeadContext头节点和TailContext尾节点。

每个流水线中双向链表结构从一开始就存在了HeadContext和TailContext两个节点，后面添加的处理器上下文节点都添加在HeadContext实例和TailContext实例之间。在添加了一些必要的解码器、业务处理器、编码器之后，一条流水线的结构大致如图5-15所示。

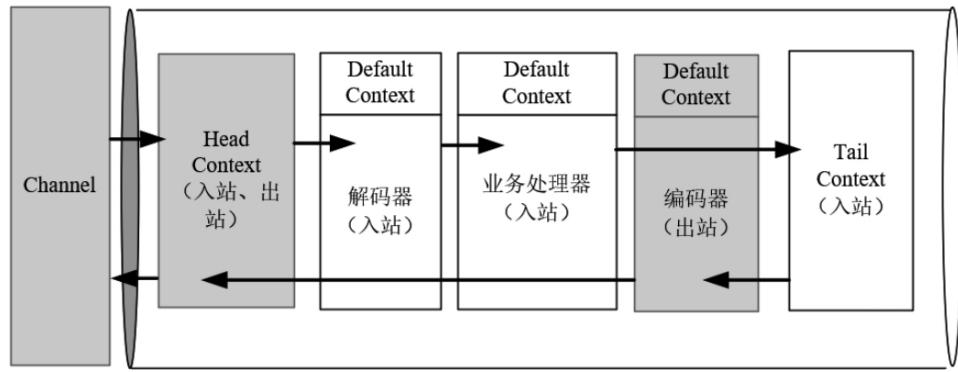


图5-15 一条流水线的结构大致示意图

流水线尾部的TailContext不仅仅是一个上下文类，还是一个入站处理器类，实现了所有入站处理回调方法，这些回调实现的主要工作基本上都是有关收尾处理的，如释放缓冲区对象、完成异常处理等。

TailContext是流水线默认实现类DefaultChannelPipeline的一个内部类，代码大致如下：

```
//流水线默认实现类（来自Netty4.1.49版本）

public class DefaultChannelPipeline implements ChannelPipeline
{
    ...

    //内部类：尾部处理器和尾部上下文是同一个类
    final class TailContext extends AbstractChannelHandlerContext
        implements
    ChannelInboundHandler {

        //入站处理方法：读取通道
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object
msg) {
```

```
//释放缓冲区  
...  
}  
//省略TailContext 其他的入站处理方法  
}  
...  
}
```

流水线头部的HeadContext比TailContext复杂得多，既是一个出站处理器，也是一个入站处理器，还保存了一个unsafe（完成实际通道传输的类）实例，也就是HeadContext还需要负责最终的通道传输工作。

HeadContext也是流水线默认实现类DefaultChannelPipeline的一个内部类，代码大致如下：

```
//流水线默认实现类（来自Netty4.1.49版本）  
public class DefaultChannelPipeline implements ChannelPipeline  
{  
    ...  
    //内部类：头部处理器和头部上下文是同一个类  
    //并且头部处理器既是出站处理器也是入站处理器  
    final class HeadContext extends AbstractChannelHandlerContext  
        implements ChannelOutboundHandler,  
    ChannelInboundHandler {  
  
        //传输操作类实例：完成通道最终的输入、输出等操作  
        //此类专供Netty内部使用，应用程序不能使用，所以取名unsafe
```

```
private final Unsafe unsafe;

//入站处理举例：入站（从Channel到Handler）读操作
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ctx.fireChannelRead(msg);
}

//出站处理举例：出站（从Handler到Channel）读取传输数据
@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}

//出站处理举例：出站（从Handler到Channel）写操作
@Override
public void write(ChannelHandlerContext ctx,
Object msg, ChannelPromise promise) {
    unsafe.write(msg, promise);
}

//省略HeadContext其他的处理方法
}

...
}
```

5.6.5 Pipeline入站和出站的双向链接操作

在理解了HeadContext与TailContext两个重要的节点之后，再来梳理一下Pipeline的出站和入站处理流程中的双向链接操作。下面摘取流水线的一个入站（读）操作和一个出站（写）操作，源码大致如下：

```
final class DefaultChannelPipeline implements ChannelPipeline {  
    final AbstractChannelHandlerContext head; //HeadContext  
    final AbstractChannelHandlerContext tail; //TailContext  
  
    //出站：启动流水线的出站写  
    @Override  
    public ChannelFuture write(Object msg) {  
        return tail.write(msg); //从后往前传递  
    }  
  
    //入站：启动流水线的入站读  
    @Override  
    public ChannelPipeline fireChannelRead(Object msg) {  
        head.fireChannelRead(msg); //从头往后传递  
        return this;  
    }  
    ...  
}
```

完整的出站和入站处理流转过程都是通过调用流水线实例的相应出/入站方法开启的。先看看入站处理的流转过程，以流水线的入站读

的启动过程为例，从以上源码可以看出，流水线的入站流程是从fireXXX()方法开始的（XXX表示具体入站操作，入站读的操作为ChannelRead）。在fireChannelRead的源码中，从流水线的头节点Head开始，将入站的msg数据沿着流水线上的入站处理器逐个向后传递，如图5-16所示。

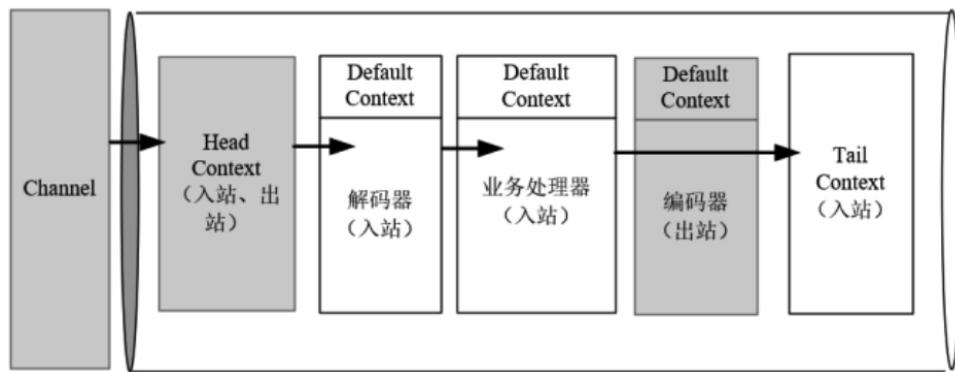


图5-16 流水线的入站处理流程大致示意图

如果所有的入站处理过程都没有截断流水线的处理，则该入站数据msg（如ByteBuffer缓冲区）将一直传递到流水线的末尾，也就是TailContext处理器。

从源码可以看出，流水线的出站流程是从流水线的尾部节点Tail开始的，将出站的msg数据沿着流水线上的出站处理器逐个向前传递，如图5-17所示。

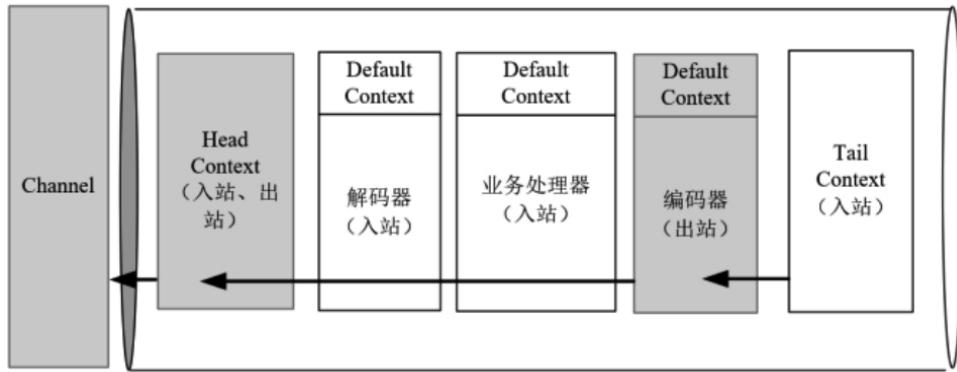


图5-17 流水线的出站处理流程大致示意图

出站msg数据在经过所有出站处理器之后，将一直传递到流水线的头部，也就是HeadContext处理器，并且通过unsafe传输实例将二进制数据写入底层传输通道，完成整个传输处理过程。

出站和入站被流水线启动之后，其传播的中间过程具体如何呢？这里需要了解一下流水线链表的节点实现，其默认的实现类为AbstractChannelHandlerContext抽象类，此类也是HeadContext与TailContext的父类。pipeline内部的双向链表的指针维护以及节点前驱和后继的计算方法都在这个类中实现。

AbstractChannelHandlerContext的核心成员如下：

```
abstract class AbstractChannelHandlerContext
...implements ChannelHandlerContext {
    //双向链表的指针：指向后继
    volatile AbstractChannelHandlerContext next;
    //双向链表的指针：指向前驱
    volatile AbstractChannelHandlerContext prev;
    private final boolean inbound; //标志：是否为入站
```

节点

```
private final boolean outbound; //标志：是否为出站
```

节点

```
private final AbstractChannel channel; //上下文节点所关联
```

的通道

```
private final DefaultChannelPipeline pipeline; //所属流
```

水线

```
private final String name; //上下文节点名称，可以在加入流水
```

线时指定

```
//节点的执行线程，如果没有特别设置，则为通道的IO线程
```

```
final EventExecutor executor;
```

```
//...
```

```
}
```

AbstractChannelHandlerContext的成员属性不止这些，以上成员仅仅是与Pipeline入站和出站的双向链接操作有关的核心成员属性。

Pipeline如何通过上下文实例进行出入站的传播呢？

首先介绍入站操作的传播。以入站读ChannelRead操作为例，下面是fireChannelRead()方法（传播入站读）的源码：

```
abstract class AbstractChannelHandlerContext  
...implements ChannelHandlerContext {  
//...  
@Override  
public ChannelHandlerContext fireChannelRead(final  
Object msg) {
```

```

    if (msg == null) {
        throw new NullPointerException("msg");
    }

    //在双向链表中向后查找，找到下一个入站节点（同类的后继）
    final AbstractChannelHandlerContext next =
        findContextInbound();

    EventExecutor executor = next.executor(); //获取
    后继的处理线程

    if (executor.inEventLoop()) {
        //如果当前线程为后继的处理线程
        //执行后继上下文所包装的处理器
        next.invokeChannelRead(msg);
    } else {
        //如果当前处理线程不是后继的处理线程，则提交到
        后继处理线程去排队
        //保障该节点的处理器被设置的线程调用，避免发生
        线程安全问题

        executor.execute(new OneTimeTask() {
            @Override
            public void run() {
                //提交到后继处理线程
                next.invokeChannelRead(msg);
            }
        });
    }
}

```

```
        return this;  
    }  
  
    ...  
}
```

Pipeline的入站和出站的传播方向是相反的，入站是顺着双向链表向后传播，出站是顺着双向链表向前传播。所以，在fireChannelRead()方法中，调用findContextInbound()方法，找到下一个入站节点（后继的入站节点），该方法的源码如下：

```
//在双向链表中向后查找，找到下一个入站节点
```

```
private AbstractChannelHandlerContext findContextInbound() {  
    AbstractChannelHandlerContext ctx = this;  
    do {  
        ctx = ctx.next; //向后查找，一直到末尾或者找到入站类型节点为止  
    } while (!ctx.inbound);  
    return ctx;  
}
```

在fireChannelRead()方法中通过findContextInbound()方法找到下一棒入站Context之后，准备开始执行下一站所包装的处理器，只不过这里需要确保执行的线程是该Context实例的executor成员线程以保证线程安全。执行下一站的处理器的方法如下：

```
//执行下一棒入站Context所包装的处理器  
private void invokeChannelRead(Object msg) {
```

```

try {
    ((ChannelInboundHandler)
handler()).channelRead(this, msg);
} catch (Throwable t) {
    notifyHandlerException(t);
}
}

```

以上为入站处理的传播过程。Pipeline的出站传播除了方向是相反的，其余的地方与入站传播大致相同，其查找一下出站处理的方法之源码如下：

```

//在双向链表中向前查找，找到前一个出站节点
private AbstractChannelHandlerContext findContextOutbound() {
    AbstractChannelHandlerContext ctx = this;
    do {
        //向前查找，直到头部或者找到一个出站Context为止
        ctx = ctx.prev;
    } while (!ctx.outbound);
    return ctx;
}

```

5.6.6 截断流水线的入站处理传播过程

在入站/出站的过程中，如果由于业务条件不满足而需要截断流水线的处理，不让处理传播到下一站，那么该怎么办呢？

这里以channelRead入站读的处理流程为例，看看如何截断入站处理流程。这里采用的办法是在处理器的channelRead()方法中不再调用父处理器的channelRead()入站方法。代码如下：

```
package com.crazymakercircle.netty.pipeline;
//...
public class InPipeline {
    //省略SimpleInHandlerA、SimpleInHandlerC

    //定义 SimpleInHandlerB2，替换掉SimpleInHandlerB
    static class SimpleInHandlerB2 extends
        ChannelInboundHandlerAdapter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object
msg) ...{
            Logger.info("入站处理器 B: 被回调 ");
            //不调用基类的channelRead，终止流水线的执行
            //super.channelRead(ctx, msg);
        }
    }

    @Test
    public void testPipelineCutting() {
        ChannelInitializer i =
new ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannel ch) {
                ch.pipeline().addLast(new SimpleInHandlerA());
            }
        }
    }
}
```

```

        ch.pipeline().addLast(new SimpleInHandlerB());
        ch.pipeline().addLast(new SimpleInHandlerC());
    }
};

EmbeddedChannel channel = new EmbeddedChannel(i);
ByteBuf buf = Unpooled.buffer();
buf.writeInt(1);
//向通道写一个入站报文（或数据包），启动入站处理器流程
channel.writeInbound(buf);
//...
}
}

```

以上代码同样定义了3个业务处理器，只是中间的业务处理器 SimpleInHandlerB 没有调用父类的 super.channelRead() 方法。运行的结果如下：

```
[T:main|F:channelRead] |>入站处理器 A: 被回调
[T:main|F:channelRead] |>入站处理器 B: 被回调
```

从运行的结果可以看出，入站处理器C没有执行到，说明处理流水线被成功地截断了，如图5-18所示。

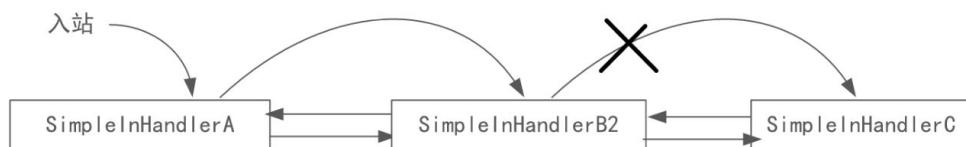


图5-18 处理流水线的截断

在以上代码中，通过不调用基类的channelRead()方法截断流水线的执行。在channelRead()方法中，将入站处理结果发送到一站还有一种方法：调用Context上下文的ctx.fireChannelRead(msg)方法。如果要截断流水线的处理，显然不能调用ctx.fireChannelRead(msg)方法。

上面演示的是channelRead读操作入站流程的截断，仅仅是一个示例，如果要截断其他的入站处理的流水线操作（使用Xxx指代），也可以同样处理：

- (1) 不调用super.channelXxx(ChannelHandlerContext)。
- (2) 不调用ctx.fireChannelXxx()。

大家在编写入站处理器的代码时一般会继承ChannelInboundHandlerAdapter适配器，而该适配器的默认入站实现主要是进行入站操作的流水线传播，并且是通过上下文Context实例完成的，大致的源码如下：

```
//入站处理适配器
public class ChannelInboundHandlerAdapter
    extends ChannelHandlerAdapter implements
ChannelInboundHandler {

    //入站方法举例：入站读
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object
msg) ...{
```

```
//通过上下文进行入站读操作的流水线传播  
ctx.fireChannelRead(msg);  
}  
//...其他的入站方法的源码类似，故省略  
}
```

至此，入站处理传播流程的截断技巧和背后的原理介绍完了。

流水线的出站处理传播流程如何截断呢？结论是：出站处理流程只要开始执行，就不能被截断，强行截断的话Netty会抛出异常。如果业务条件不满足，可以不启动出站处理。大家可以运行示例工程中的testPipelineOutBoundCutting()测试方法，查看出站处理截断后抛出的异常，这里不再赘述。

5.6.7 在流水线上热插拔Handler

Netty中的处理器流水线是一个双向链表。在程序执行过程中，可以动态进行业务处理器的热插拔：动态地增加、删除流水线上的业务处理器。主要的Handler热拔插方法声明在ChannelPipeline接口中，具体如下：

```
package io.netty.channel;  
//...  
public interface ChannelPipeline  
    extends Iterable<Entry<String,  
    ChannelHandler>>  
{
```

```
//...
//在流水线头部增加一个业务处理器，名字由name指定
ChannelPipeline addFirst(String name, ChannelHandler
handler);
//在流水线尾部增加一个业务处理器，名字由name指定
ChannelPipeline addLast(String name, ChannelHandler
handler);
//在baseName处理器的前面增加一个业务处理器，名字由name指定
ChannelPipeline addBefore(String baseName, String name,
ChannelHandler handler);

//在baseName处理器的后面增加一个业务处理器，名字由name指定
ChannelPipeline addAfter(String baseName, String name,
ChannelHandler handler);

//删除一个业务处理器实例
ChannelPipeline remove(ChannelHandler handler);
//删除一个处理器实例
ChannelHandler remove(String handler);
//删除第一个业务处理器
ChannelHandler removeFirst();
//删除最后一个业务处理器
ChannelHandler removeLast();
//...
}
```

如果需要动态地增加、删除流水线上的业务处理器，调用以上 ChannelPipeline 的某个方法即可。下面是一个简单的示例：调用流水

线实例的remove(ChannelHandler)方法，从流水线动态地删除一个Handler。

```
package com.crazymakercircle.netty.pipeline;
//...
public class PipelineHotOperateTester {
    static class SimpleInHandlerA extends
ChannelInboundHandlerAdapter {
        public void channelRead(ChannelHandlerContext ctx, Object
msg) ...{
            Logger.info("入站处理器 A: 被回调 ");
            super.channelRead(ctx, msg);
            //从流水线删除当前业务处理器
            ctx.pipeline().remove(this);
        }
    }
    //省略SimpleInHandlerB、SimpleInHandlerC的定义
    //测试业务处理器的热拔插
    @Test
    public void testPipelineHotOperating() {
        ChannelInitializer i = new
ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannel ch) {
                ch.pipeline().addLast(new SimpleInHandlerA());
                ch.pipeline().addLast(new SimpleInHandlerB());
            }
        }
    }
}
```

```
        ch.pipeline().addLast(new SimpleInHandlerC());  
    }  
};  
  
EmbeddedChannel channel = new EmbeddedChannel(i);  
ByteBuf buf = Unpooled.buffer();  
buf.writeInt(1);  
//第一次向通道写入站报文（或数据包）  
channel.writeInbound(buf);  
//第二次向通道写入站报文（或数据包）  
channel.writeInbound(buf);  
//第三次向通道写入站报文（或数据包）  
channel.writeInbound(buf);  
//省略其他代码  
}
```

运行示例代码，结果节选如下：

```
[...A|F:channelRead] |>入站处理器 A: 被回调  
[...B|F:channelRead] |>入站处理器 B: 被回调  
[...C|F:channelRead] |>入站处理器 C: 被回调  
[...B|F:channelRead] |>入站处理器 B: 被回调  
[...C|F:channelRead] |>入站处理器 C: 被回调  
[...B|F:channelRead] |>入站处理器 B: 被回调  
[...C|F:channelRead] |>入站处理器 C: 被回调
```

从运行结果中可以看出，在SimpleInHandlerA从流水线中删除后，在后面的入站流水处理中（第二次和第三次入站处理流程），SimpleInHandlerA已经不再被调用了。

这里为大家分析一下通道初始化处理器ChannelInitializer没有被重复调用的原因。通过翻看源码可以知道，在注册完成channelRegistered回调方法中调用ctx.pipeline().remove(this)将自己从流水线中删除了，所以该处理器仅仅被执行了一次。有关ChannelInitializer的源代码，节选如下：

```
package io.netty.channel;  
//省略不相关代码  
public abstract class ChannelInitializer extends  
    ChannelInboundHandlerAdapter {  
    //...  
    //通道初始化，抽象方法，需要子类实现  
    protected abstract void initChannel(Channel var1)  
throws Exception;  
    //回调方法：加入通道（注册完成）后触发  
    public final void  
channelRegistered(ChannelHandlerContext ctx) {  
    //调用通道初始化实现  
    this.initChannel(ctx.channel());  
    //删除通道初始化处理器  
    ctx.pipeline().remove(this);  
    //发送注册消息到下一站  
    ctx.fireChannelRegistered();  
}  
//...  
}
```

ChannelInitializer在完成了通道的初始化之后，为什么要将自己从流水线中删除呢？原因很简单，就是一条通道流水线只需要做一次装配工作。

5.7 详解ByteBuf

Netty提供了ByteBuf缓冲区组件来替代Java NIO的ByteBuffer缓冲区组件，以便更加快捷和高效地操纵内存缓冲区。

5.7.1 ByteBuf的优势

与Java NIO的ByteBuffer相比，ByteBuf的优势如下：

- Pooling（池化），减少了内存复制和GC，提升了效率。
- 复合缓冲区类型，支持零复制。
- 不需要调用`flip()`方法去切换读/写模式。
- 可扩展性好。
- 可以自定义缓冲区类型。
- 读取和写入索引分开。
- 方法的链式调用。
- 可以进行引用计数，方便重复使用。

5.7.2 ByteBuf的组成部分

ByteBuf是一个字节容器，内部是一个字节数组。从逻辑上来分，字节容器内部可以分为四个部分，具体如图5-19所示。

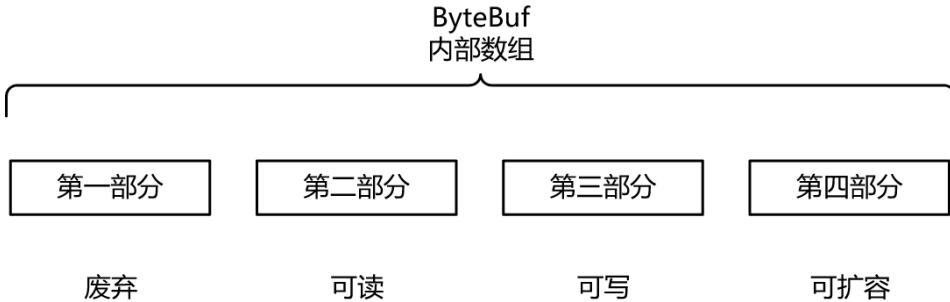


图5-19 ByteBuf的内部字节数组

第一部分是已用字节，表示已经使用完的废弃的无效字节；第二部分是可读字节，这部分数据是ByteBuf保存的有效数据，从ByteBuf中读取的数据都来自这一部分；第三部分是可写字节，写入ByteBuf的数据都会写到这一部分中；第四部分是可扩容字节，表示的是该ByteBuf最多还能扩容的大小。

5.7.3 ByteBuf的重要属性

ByteBuf通过三个整数类型的属性有效地区分可读数据和可写数据的索引，使得读写之间相互没有冲突。这三个属性定义在AbstractByteBuf抽象类中，分别是：

- `readerIndex`（读指针）：指示读取的起始位置。每读取一个字节，`readerIndex`自动增加1。一旦`readerIndex`与`writerIndex`相等，则表示ByteBuf不可读了。
- `writerIndex`（写指针）：指示写入的起始位置。每写一个字节，`writerIndex`自动增加1。一旦增加到`writerIndex`与`capacity()`容量相等，则表示ByteBuf不可写了。注意，`capacity()`是一个成员方法，不是一个成员属性，表示ByteBuf中可以写入的容量，而且它的值不一定是最大容量值。

- `maxCapacity`（最大容量）：表示`ByteBuf`可以扩容的最大容量。当向`ByteBuf`写数据的时候，如果容量不足，可以进行扩容。扩容的最大限度由`maxCapacity`来设定，超过`maxCapacity`就会报错。

`ByteBuf`的这三个重要属性的含义如图5-20所示。

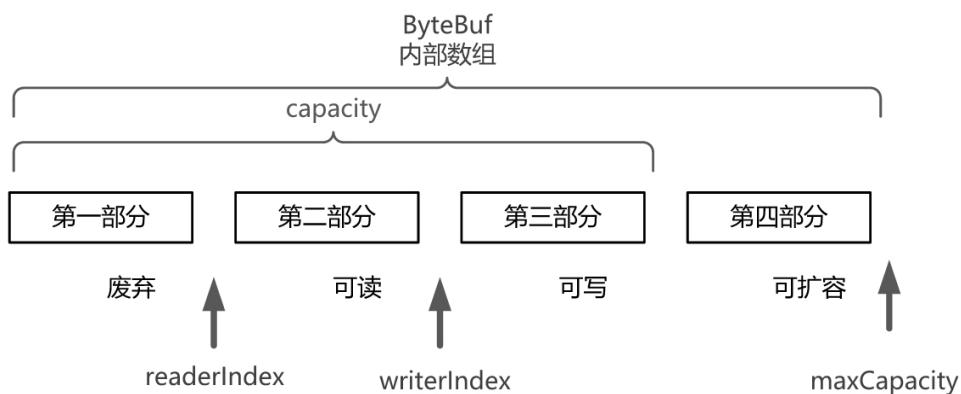


图5-20 `ByteBuf`内部的三个重要属性的含义

5.7.4 `ByteBuf`的方法

`ByteBuf`的方法大致可以分为三组。

第一组：容量系列

- `capacity()`：表示`ByteBuf`的容量，是废弃的字节数、可读字节数和可写字节数之和。
- `maxCapacity()`：表示`ByteBuf`能够容纳的最大字节数。当向`ByteBuf`中写数据的时候，如果发现容量不足，则进行扩容，直至扩容到`maxCapacity`设定的上限。

第二组：写入系列

- `isWritable()`: 表示ByteBuf是否可写。如果`capacity()`容量大于`writerIndex`指针的位置，则表示可写，否则为不可写。注意：`isWritable()`返回`false`并不代表不能再往ByteBuf中写数据了。如果Netty发现往ByteBuf中写数据写不进去，就会自动扩容ByteBuf。
- `writableBytes()`: 取得可写入的字节数，它的值等于容量`capacity()`减去`writerIndex`。
- `maxWritableBytes()`: 取得最大的可写字节数，它的值等于最大容量`maxCapacity`减去`writerIndex`。
- `writeBytes(byte[] src)`: 把入参src字节数组中的数据全部写到ByteBuf。这是最为常用的一个方法。
- `writeTYPE(TYPE value)`: 写入基础数据类型的数据。TYPE表示基础数据类型，这里包含了八种大基础数据类型：
`writeByte()`、`writeBoolean()`、`writeChar()`、
`writeShort()`、`writeInt()`、`writeLong()`、`writeFloat()`、
`writeDouble()`。
- `setTYPE(TYPE value)`: 基础数据类型的设置，不改变`writerIndex`指针值。TYPE表示基础数据类型这里包含了八大基础数据类型的设置，即`setByte()`、`setBoolean()`、
`setChar()`、`setShort()`、`setInt()`、`setLong()`、
`setFloat()`、`setDouble()`。`setTYPE`系列与`writeTYPE`系列的不同点是`setTYPE`系列不改变写指针`writerIndex`的值，`writeTYPE`系列会改变写指针`writerIndex`的值。
- `markWriterIndex()`与`resetWriterIndex()`: 前一个方法表示把当前的写指针`writerIndex`属性的值保存在`markedWriterIndex`

标记属性中；后一个方法表示把之前保存的markedWriterIndex的值恢复到写指针writerIndex属性中。这两个方法都用到了标记属性markedWriterIndex，相当于一个写指针的暂存属性。

第三组：读取系列

- `isReadable()`：返回ByteBuf是否可读。如果writerIndex指针的值大于readerIndex指针的值，则表示可读，否则为不可读。
- `readableBytes()`：返回表示ByteBuf当前可读取的字节数，它的值等于writerIndex减去readerIndex。
- `readBytes(byte[] dst)`：将数据从ByteBuf读取到dst目标字节数组中，这里dst字节数组的大小通常等于`readableBytes()`可读字节数。这个方法也是最为常用的方法之一。
- `readTYPE()`：读取基础数据类型。可以读取八大基础数据类型：`readByte()`、`readBoolean()`、`readChar()`、`readShort()`、`readInt()`、`readLong()`、`readFloat()`、`readDouble()`。
- `getTYPE()`：读取基础数据类型，并且不改变readerIndex读指针的值，具体为`getByte()`、`getBoolean()`、`getChar()`、`getShort()`、`getInt()`、`getLong()`、`getFloat()`、`getDouble()`。`getTYPE`系列与`readTYPE`系列的不同点是`getTYPE`系列不会改变读指针readerIndex的值，`readTYPE`系列会改变读指针readerIndex的值。
- `markReaderIndex()`与`resetReaderIndex()`：前一种方法表示把当前的读指针readerIndex保存在markedReaderIndex属性中；后一种方法表示把保存在markedReaderIndex属性的值恢复到读指针readerIndex中。`markedReaderIndex`属性定义在

AbstractByteBuf抽象基类中，是一个标记属性，相当于一个读指针的暂存属性。

5.7.5 ByteBuf基本使用的实战案例

ByteBuf的基本使用分为三部分：

(1) 分配一个ByteBuf实例。

(2) 向ByteBuf写数据。

(3) 从ByteBuf读数据。

这里使用默认的分配器分配了一个初始容量为9、最大限制为100个字节的缓冲区。关于ByteBuf实例的分配器，后面章节会详细介绍。

实战代码很简单，具体如下：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
public class WriteReadTest {  
    @Test  
    public void testWriteRead() {  
        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(9,  
100);  
        print("动作：分配ByteBuf(9, 100)", buffer);  
        buffer.writeBytes(new byte[]{1, 2, 3, 4});  
        print("动作：写入4个字节 (1,2,3,4)", buffer);  
        Logger.info("start=====:get=====");  
    }  
}
```

```

        getByteBuf(buffer);
        print("动作： 取数据ByteBuf", buffer);
        Logger.info("start=====:read=====");
        readByteBuf(buffer);
        print("动作： 读完ByteBuf", buffer);
    }

    //取字节

    private void readByteBuf(ByteBuf buffer) {
        while (buffer.isReadable()) {
            Logger.info("取一个字节：" + buffer.readByte());
        }
    }

    //读字节， 不改变指针

    private void getByteBuf(ByteBuf buffer) {
        for (int i = 0; i<buffer.readableBytes(); i++) {
            Logger.info("读一个字节：" + buffer.getByte(i));
        }
    }

}

```

有关运行的结果，节选如下：

```

[main|...:print]: after =====动作： 分配ByteBuf(9,
100)=====
[main|...:print]: 1.0 isReadable(): false
[main|...:print]: 1.1 readerIndex(): 0
[main|...:print]: 1.2 readableBytes(): 0

```

```
[main|...:print]: 2.0 isWritable(): true
[main|...:print]: 2.1 writerIndex(): 0
[main|...:print]: 2.2 writableBytes(): 9
[main|...:print]: 3.0 capacity(): 9
[main|...:print]: 3.1 maxCapacity(): 100
[main|...:print]: 3.2 maxWritableBytes(): 100
//...
[main|...:print]: after =====动作: 写入4个字节
(1,2,3,4)=====
[main|...:print]: 1.0 isReadable(): true
[main|...:print]: 1.1 readerIndex(): 0
[main|...:print]: 1.2 readableBytes(): 4
[main|...:print]: 2.0 isWritable(): true
[main|...:print]: 2.1 writerIndex(): 4
[main|...:print]: 2.2 writableBytes(): 5
[main|...:print]: 3.0 capacity(): 9
[main|...:print]: 3.1 maxCapacity(): 100
[main|...:print]: 3.2 maxWritableBytes(): 96
//...
[main|...:print]: after =====动作: 取数据ByteBuf=====
[main|...:print]: 1.0 isReadable(): true
[main|...:print]: 1.1 readerIndex(): 0
[main|...:print]: 1.2 readableBytes(): 4
[main|...:print]: 2.0 isWritable(): true
[main|...:print]: 2.1 writerIndex(): 4
[main|...:print]: 2.2 writableBytes(): 5
[main|...:print]: 3.0 capacity(): 9
```

```
[main|...:print]: 3.1 maxCapacity(): 100
[main|...:print]: 3.2 maxWritableBytes(): 96
//...
[main|...:print]: after =====动作: 读完ByteBuf=====
[main|...:print]: 1.0 isReadable(): false
[main|...:print]: 1.1 readerIndex(): 4
[main|...:print]: 1.2 readableBytes(): 0
[main|...:print]: 2.0 isWritable(): true
[main|...:print]: 2.1 writerIndex(): 4
[main|...:print]: 2.2 writableBytes(): 5
[main|...:print]: 3.0 capacity(): 9
[main|...:print]: 3.1 maxCapacity(): 100
[main|...:print]: 3.2 maxWritableBytes(): 96
```

可以看到，使用get取数据是不会影响ByteBuf指针属性值的。由于篇幅原因，这里不仅省略了很多输出结果，还省略了print()方法的源代码，它的作用是打印ByteBuf的属性值。建议打开源代码工程，查看和运行本案例的代码。

5.7.6 ByteBuf的引用计数

JVM中使用“计数器”（一种GC算法）来标记对象是否“不可达”进而收回，Netty也使用了这种手段来对ByteBuf的引用进行计数。

（注：GC是Garbage Collection的缩写，即Java中的垃圾回收机制。）Netty的ByteBuf的内存回收工作是通过引用计数方式管理的。

Netty之所以采用“计数器”来追踪ByteBuf的生命周期，一是能对Pooled ByteBuf进行支持，二是能够尽快“发现”那些可以回收的ByteBuf（非Pooled），以便提升ByteBuf的分配和销毁的效率。

说明

什么是池化（Pooled）的ByteBuf缓冲区呢？从Netty 4版本开始，新增了ByteBuf的池化机制，即创建一个缓冲区对象池，将没有被引用的ByteBuf对象放入对象缓存池中，需要时重新从对象缓存池中取出，而不需要重新创建。

在通信程序的数据传输过程中，Buffer缓冲区实例会被频繁创建、使用、释放，从而频繁创建对象、内存分配、释放内存，这样会导致系统的开销大、性能低。如何提升性能、提高Buffer实例的使用率呢？池化ByteBuf是一种非常有效的方式。

ByteBuf引用计数的大致规则如下：在默认情况下，当创建完一个ByteBuf时，引用计数为1；每次调用retain()方法，引用计数加1；每次调用release()方法，引用计数减1；如果引用为0，再次访问这个ByteBuf对象，将会抛出异常；如果引用为0，表示这个ByteBuf没有哪个进程引用，它占用的内存需要回收。

在下面的例子中，多次调用了ByteBuf的retain()和release()方法，运行后可以看效果：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
  
public class ReferenceTest {  
  
    @Test  
  
    public void testRef()  
  
    {  
  
        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer();  
        Logger.info("after create:" + buffer.refCnt());  
  
        buffer.retain(); //增加一次引用计数  
        Logger.info("after retain:" + buffer.refCnt());  
  
        buffer.release(); //减少一次引用计数  
        Logger.info("after release:" + buffer.refCnt());  
  
        buffer.release(); //减少一次引用计数  
        Logger.info("after release:" + buffer.refCnt());  
  
        //错误:refCnt: 0, 不能再retain  
        buffer.retain(); //增加一次引用计数  
        Logger.info("after retain:" + buffer.refCnt());  
    }  
}
```

运行程序，结果如下：

```
[main|ReferenceTest.testRef] |> after create:1
[main|ReferenceTest.testRef] |> after retain:2
[main|ReferenceTest.testRef] |> after release:1
[main|ReferenceTest.testRef] |> after release:0
... (省略不相关的输出)
io.netty.util.IllegalReferenceCountException: refCnt: 0,
increment: 1
... (省略异常信息)
```

运行后我们会发现：最后一次retain()方法抛出了IllegalReferenceCountException异常。原因是：在此之前，缓冲区buffer的引用计数已经为0，不能再retain了。也就是说：在Netty中，引用计数为0的缓冲区不能再继续使用。

为了确保引用计数不会混乱，在Netty的业务处理器开发过程中应该坚持一个原则：retain()和release()方法应该结对使用。对缓冲区调用了一次retain()，就应该调用一次release()。大致的参考代码如下：

```
public void handleMethodA(ByteBuf byteBuf) {
    byteBuf.retain();
    try {
        handleMethodB(byteBuf);
    } finally {
        byteBuf.release();
    }
}
```

如果retain()和release()这两个方法一次都不调用呢？Netty在缓冲区使用完成后会调用一次release()，就是释放一次。例如，在Netty流水线上，中间所有的业务处理器处理完ByteBuf之后会直接传递给下一个，由最后一个Handler负责调用其release()方法来释放缓冲区的内存空间。

当ByteBuf的引用计数已经为0时，Netty会进行ByteBuf的回收，分为以下两种场景：

(1) 如果属于池化的ByteBuf内存，回收方法是：放入可以重新分配的ByteBuf池，等待下一次分配。

(2) 如果属于未池化的ByteBuf缓冲区，需要细分为两种情况：如果是堆(Heap)结构缓冲，会被JVM的垃圾回收机制回收；如果是直接(Direct)内存类型，则会调用本地方法释放外部内存(unsafe.freeMemory)。

除了通过ByteBuf成员方法retain()和release()管理引用计数之外，Netty还提供了一组用于增加和减少引用计数的通用静态方法：

(1) ReferenceCountUtil.retain(Object)：增加一次缓冲区引用计数的静态方法，从而防止该缓冲区被释放。

(2) ReferenceCountUtil.release(Object)：减少一次缓冲区引用计数的静态方法，如果引用计数为0，缓冲区将被释放。

5.7.7 ByteBuf的分配器

Netty通过ByteBufAllocator分配器来创建缓冲区和分配内存空间。Netty提供了两种分配器实现：PoolByteBufAllocator和UnpooledByteBufAllocator。

PoolByteBufAllocator（池化的ByteBuf分配器）将ByteBuf实例放入池中，提高了性能，将内存碎片减少到最小；池化分配器采用了jemalloc高效内存分配的策略，该策略被好几种现代操作系统所采用。

UnpooledByteBufAllocator是普通的未池化ByteBuf分配器，没有把ByteBuf放入池中，每次被调用时，返回一个新的ByteBuf实例；使用完之后，通过Java的垃圾回收机制回收或者直接释放（对于直接内存而言）。

为了验证两者的性能，大家可以做一下对比试验：

（1）使用UnpooledByteBufAllocator方式分配ByteBuf缓冲区，开启10000个长连接，每秒所有的连接发一条消息，再看看服务器的内存使用量情况。

实验的参考结果：在较短时间内，就可以看到程序占到10GB多的内存空间，随着系统的运行，内存空间会不断增长，直到整个系统内存被占满而导致内存溢出，最终宕机。

（2）把UnpooledByteBufAllocator换成PooledByteBufAllocator，再进行试验，看看服务器的内存使用量情况。

实验的参考结果：内存使用量基本能维持在一个连接占用1MB左右的内存空间，内存使用量保持在10GB左右，经过长时间的运行测试，我们会发现内存使用量能维持在这个数量附近，系统不会因为内存被耗尽而崩溃。

在Netty中，默认的分配器为ByteBufAllocator.DEFAULT。该默认的分配器可以通过系统参数（System Property）选项 io.netty.allocator.type 进行配置，配置时使用字符串值：“unpooled”，“pooled”。

不同的Netty版本，对于分配器的默认使用策略是不一样的。在 Netty 4.0 版本中，默认的分配器为 UnpooledByteBufAllocator（非池化内存分配器）。在 Netty 4.1 版本中，默认的分配器为 PooledByteBufAllocator（池化内存分配器），初始化代码在 ByteBufUtil 类中的静态代码中，具体如下：

```
public final class ByteBufUtil {  
    ...  
    static {  
        //Android系统默认为unpooled，其他系统默认为pooled  
        //除非通过系统属性io.netty.allocator.type 做专门配  
        //置  
        String allocType = SystemPropertyUtil.get(  
            "io.netty.allocator.type",  
            PlatformDependent.isAndroid() ?  
                "unpooled" : "pooled");  
        ByteBufAllocator alloc;  
        if ("unpooled".equals(allocType)) {
```

```
        alloc =
UnpooledByteBufAllocator.DEFAULT;

        ...
} else if ("pooled".equals(allocType)) {
    alloc = PooledByteBufAllocator.DEFAULT;
    ...
} else {
    alloc = PooledByteBufAllocator.DEFAULT;
    ...
}
DEFAULT_ALLOCATOR = alloc;
...
}

}
```

现在PooledByteBufAllocator已经广泛使用了一段时间，并且有了增强的缓冲区泄漏追踪机制。因此，也可以在Netty程序中设置引导类Bootstrap装配的时候将PooledByteBufAllocator设置为默认的分配器。

```
ServerBootstrap b = new ServerBootstrap()
//设置通道的参数
b.option(ChannelOption.SO_KEEPALIVE, true);
//设置父通道的缓冲区分配器
b.option(ChannelOption.ALLOCATOR,
PooledByteBufAllocator.DEFAULT);
//设置子通道的缓冲区分配器
```

```
b.childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
```

Netty的内存管理策略可以灵活调整，这是使用Netty所带来的又一个好处：只需一行简单的配置就能获得到池化缓冲区带来的好处。在底层，Netty为我们干了所有“脏活、累活”！

使用缓冲区分配器创建ByteBuf的方法有多种，下面列出几种主要的：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
public class AllocatorTest {  
    @Test  
    public void showAlloc() {  
        ByteBuf buffer = null;  
  
        //方法1：通过默认分配器分配  
        //初始容量为9、最大容量为100的缓冲区  
        buffer = ByteBufAllocator.DEFAULT.buffer(9, 100);  
  
        //方法2：通过默认分配器分配  
        //初始容量为256、最大容量为Integer.MAX_VALUE的缓冲区  
        buffer = ByteBufAllocator.DEFAULT.buffer();  
  
        //方法3：非池化分配器，分配Java的堆（Heap）结构内存缓冲区  
        buffer = UnpooledByteBufAllocator.DEFAULT.heapBuffer();
```

```

    //方法4：池化分配器，分配由操作系统管理的直接内存缓冲区
    buffer = PooledByteBufAllocator.DEFAULT.directBuffer();
    //其他方法
}
}

```

Netty中缓冲区分配的方法很多，可以根据实际需要进行选择。

5.7.8 ByteBuf缓冲区的类型

介绍完了分配器的类型，再来说一下缓冲区的类型（见表5-2）。根据内存的管理方不同，缓冲区分为堆缓冲区和直接缓冲区，也就是 Heap ByteBuf和Direct ByteBuf。另外，为了方便缓冲区进行组合，还提供了一种组合缓存区。

表5-2 ByteBuf缓冲区的类型

类 型	说 明	优 点	不 足
Heap ByteBuf	内部数据为一个 Java 数组，存储在 JVM 的堆空间中，可以通过 hasArray 方法来判断是不是堆缓冲区	未使用池化的情况下，能提供快速的分配和释放	写入底层传输通道之前，都会复制到直接缓冲区
Direct ByteBuf	内部数据存储在操作系统的物理内存中	能获取超过 JVM 堆限制大小的内存空间；写入传输通道比堆缓冲区更快	释放和分配空间昂贵（使用了操作系统的方 法）；在 Java 中读取数据时，需要复制一次到堆上
CompositeBuffer	多个缓冲区的组合表示	方便一次操作多个缓冲区实例	

上面三种缓冲区都可以通过池化（Pooled）、非池化（Unpooled）两种分配器来创建和分配内存空间。

下面介绍一下Direct Memory（直接内存）：

- Direct Memory不属于Java堆内存，所分配的内存其实是调用操作系统malloc()函数来获得的，由Netty的本地Native堆进行管理。
- Direct Memory容量可通过-XX:MaxDirectMemorySize来指定，如果不指定，则默认与Java堆的最大值（-Xmx指定）一样。注意：并不是强制要求，有的JVM默认Direct Memory与-Xmx值无直接关系。
- Direct Memory的使用避免了Java堆和Native堆之间来回复制数据。在某些应用场景中提高了性能。
- 在需要频繁创建缓冲区的场合，由于创建和销毁Direct Buffer（直接缓冲区）的代价比较高昂，因此不宜使用Direct Buffer。也就是说，Direct Buffer尽量在池化分配器中分配和回收。如果能将Direct Buffer进行复用，在读写频繁的情况下就可以大幅度改善性能。
- 对Direct Buffer的读写比Heap Buffer快，但是它的创建和销毁比普通Heap Buffer慢。
- 在Java的垃圾回收机制回收Java堆时，Netty框架也会释放不再使用的Direct Buffer缓冲区，因为它的内存为堆外内存，所以清理的工作不会为Java虚拟机（JVM）带来压力。注意一下垃圾回收的应用场景：①垃圾回收仅在Java堆被填满，以至于无法为新的堆分配请求提供服务时发生；②在Java应用程序中调用System.gc()函数来释放内存。

5.7.9 两类ByteBuf使用的实战案例

首先对比介绍一下Heap ByteBuf和Direct ByteBuf两类缓冲区的使用，它们有以下几点不同：

- Heap ByteBuf通过调用分配器的buffer()方法来创建；Direct ByteBuf通过调用分配器的directBuffer()方法来创建。
- Heap ByteBuf缓冲区可以直接通过array()方法读取内部数组；Direct ByteBuf缓冲区不能读取内部数组。
- 可以调用hasArray()方法来判断是否为Heap ByteBuf类型的缓冲区；如果hasArray()返回值为true，则表示是堆缓冲，否则为直接内存缓冲区。
- 从Direct ByteBuf读取缓冲数据进行Java程序处理时，相对比较麻烦，需要通过getBytes/readBytes等方法先将数据复制到Java的堆内存，然后进行其他的计算。

在实战案例中对比Heap ByteBuf和Direct ByteBuf这两类缓冲区的使用，大致的代码如下：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
  
public class BufferTypeTest {  
  
    final static Charset UTF_8 = Charset.forName("UTF-8");  
  
    //堆缓冲区测试用例  
  
    @Test  
  
    public void testHeapBuffer() {  
  
        //取得堆内存  
  
        ByteBuf heapBuf =  
ByteBufAllocator.DEFAULT.heapBuffer();  
  
        heapBuf.writeBytes("疯狂创客圈：高性能学习社
```

```
群".getBytes(UTF_8));

    if (heapBuf.hasArray()) {
        //取得内部数组
        byte[] array = heapBuf.array();
        int offset = heapBuf.arrayOffset() +
        heapBuf.readerIndex();
        int length = heapBuf.readableBytes();
        Logger.info(new String(array, offset, length,
        UTF_8));
    }
    heapBuf.release();
}

//直接缓冲区测试用例
@Test
public void testDirectBuffer() {
    ByteBuf directBuf =
ByteBufAllocator.DEFAULT.directBuffer();
    directBuf.writeBytes("疯狂创客圈:高性能学习社
群".getBytes(UTF_8));

    if (!directBuf.hasArray()) {
        int length = directBuf.readableBytes();
        byte[] array = new byte[length];
        //把数据读取到堆内存array中，再进行Java处理
        directBuf.getBytes(directBuf.readerIndex(), array);
        Logger.info(new String(array, UTF_8));
    }
}
```

```
    directBuf.release();  
}  
}
```

Direct ByteBuf的hasArray()会返回false；反过来，如果hasArray()返回false，不一定代表缓冲区一定就是Direct ByteBuf，也有可能是CompositeByteBuf。CompositeByteBuf缓冲区是Netty为了减少内存复制而提供的组合缓冲区，有关其具体的知识请查阅后面的Netty零拷贝章节。

为了快速创建ByteBuffer，Netty提供了一个非常方便的获取缓冲区的类——Unpooled，用它可以创建和使用非池化的缓冲区。Unpooled的使用也很容易，下面给出三个例子：

```
//创建堆缓冲区  
ByteBuf heapBuf = Unpooled.buffer(8);  
  
//创建直接缓冲区  
ByteBuf directBuf = Unpooled.directBuffer(16);  
  
//创建复合缓冲区  
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
```

Unpooled提供了很多方法，主要的方法大致如表5-3所示。

表5-3 Unpooled提供的主要方法

方法名称	说明
buffer()	
buffer(int initialCapacity)	返回 heap ByteBuf
buffer(int initialCapacity, int maxCapacity)	
directBuffer()	
directBuffer(int initialCapacity)	返回 direct ByteBuf
directBuffer(int initialCapacity, int maxCapacity)	
compositeBuffer()	返回 CompositeByteBuf
copiedBuffer()	返回 copied ByteBuf

除了在Netty开发中使用之外，Unpooled类的应用场景还包括不需要其他Netty组件（除了缓冲区之外）甚至无网络操作的场景，从而使得Java程序可以使用Netty的高性能、可扩展的缓冲区技术。Unpooled类可用于在Netty应用之外的其他程序中独立使用ByteBuf缓冲区。

在处理器的开发过程中（这个为Netty应用开发的主要工作），推荐大家通过调用Context.alloc()方法来获取通道的缓冲区分配器来创建ByteBuf。下面给出一个例子，演示如何通过Context上下文来获取ByteBuf：

```
public class AllocatorTest
{
    ...
    //辅助的方法：输出ByteBuf是否为直接内存，以及内存分配器
    public static void printByteBuf(String action, ByteBuf
b)
    {
        ...
    }
}
```

```

        Logger.info(" =====" + action +
"=====");
        //true表示缓冲区为Java堆内存（组合缓冲例外）
        //false表示缓冲区为操作系统管理的内存（组合缓冲例外）
        Logger.info("b.hasArray: " + b.hasArray());

        //输出内存分配器
        Logger.info("b.ByteBufAllocator: " +
b.alloc());
    }

    //处理器类：演示使用Context来获取ByteBuf
    static class AllocDemoHandler extends
ChannelInboundHandlerAdapter
{
    @Override
    public void channelRead(ChannelHandlerContext
ctx, Object msg) throws Exception
    {
        printByteBuf("入站的ByteBuf", (ByteBuf)
msg);
        ByteBuf buf = ctx.alloc().buffer();
        buf.writeInt(100);
        //向模拟通道写入一个出站包，模拟数据出站，需要
刷新通道才能获取到输出
        ctx.channel().writeAndFlush(buf);
    }
}

```

```
}

//测试用例入口

@Test

public void testByteBufAlloc()

{

    ChannelInitializer<EmbeddedChannel> i = new

    ChannelInitializer<EmbeddedChannel>()

    {

        protected void

        initChannel(EmbeddedChannel ch)

        {

            ch.pipeline().addLast(new AllocDemoHandler());

        }

    };

    EmbeddedChannel channel = new

    EmbeddedChannel(i);

    //配置通道的缓冲区分配器，这里设置一个池化的分

配器

    channel.config().setAllocator(PooledByteBufAllocator.DEFAULT);

    ByteBuf buf = Unpooled.buffer();

    buf.writeInt(1);

    //向模拟通道写入一个入站包，模拟数据入站

    channel.writeInbound(buf);

    //获取通道的出站包
```

```

        ByteBuf outBuf = (ByteBuf)
channel.readOutbound();
printByteBuf("出站的ByteBuf", (ByteBuf)
outBuf);
//省略不相关代码
}
}

```

运行测试用例入口方法testByteBufAlloc()，输出大致如下：

```

[main] |> =====入站的ByteBuf=====
[main] |> b.hasArray: true
[main] |> b.ByteBufAllocator:
UnpooledByteBufAllocator(directByDefault: true)
[main] |> =====出站的ByteBuf=====
[main] |> b.hasArray: false
[main] |> b.ByteBufAllocator:
PooledByteBufAllocator(directByDefault: true)

```

以上代码的AllocDemoHandler处理器调用ctx.alloc().buffer()
方法获取ByteBuf，有关ctx.alloc()方法的源码如下：

```

abstract class AbstractChannelHandlerContext...{

...
//获取通道的缓冲区分配器
@Override
public ByteBufAllocator alloc() {
    return channel().config().getAllocator();
}

```

```
    }  
}
```

通过源码可以看出，`ctx.alloc()`方法所获取的分配器是通道的缓冲区分配器。该分配器可以通过Bootstrap引导类为通道进行配置，也可以直接通过`channel.config().setAllocator()`为通道设置一个缓冲区分配器。

5.7.10 ByteBuf的自动创建与自动释放

1. ByteBuf的自动创建

首先来看一个问题：在入站处理时，Netty是何时自动创建入站的ByteBuf缓冲区的呢？

查看Netty源代码，我们可以看到，Netty的Reactor线程会通过底层的Java NIO通道读数据。发生NIO读取的方法为
`AbstractNioByteChannel.NioByteUnsafe.read()`，其代码如下：

```
public void read() {  
    ...  
    //channel的config信息  
    final ChannelConfig config = config();  
    //获取通道的缓冲区分配器  
    final ByteBufAllocator allocator =  
        config.getAllocator();  
    //channel的pipeline  
    final ChannelPipeline pipeline = pipeline();
```

```
//缓冲区分配时的大小推测与计算组件

final RcvByteBufAllocator.Handle allocHandle =
unsafe().recvBufAllocHandle();

//输入缓冲变量

ByteBuf byteBuf = null;
Throwable exception = null;

try {

    ...
    do {

        ...
        //使用缓冲区分配器、大小计算组件一起
        //由分配器按照计算好的大小分配的一个缓冲区
        byteBuf =
allocHandle.allocate(allocator);

        ...
        //读取数据到缓冲区
        int localReadAmount =
doReadBytes(byteBuf);

        ...
        //发送数据到流水线，进行入站处理
        pipeline.fireChannelRead(byteBuf);

        ...
    } while (++ messages < maxMessagesPerRead);

    ...
} catch (Throwable t) {
    handleReadException(pipeline, byteBuf, t,
close);
```

```
    }  
    ...  
}
```

分配缓冲区的时候，为什么要计算大小呢？从通道里读取数据时是不知道接收到数据的具体大小的，那么申请的缓冲区究竟要多大呢？首先，不能太大，太大了浪费；其次，也不能太小，太小了又不够，就需要进行缓冲区的扩容，会影响性能。所以，需要推测要申请的缓冲区大小。Netty设计了一个RecvByteBufAllocator大小推测接口和一系列的大小推测实现类，以帮助进行缓冲区大小的计算和推测。默认的缓冲区大小推测实现类为AdaptiveRecvByteBufAllocator，其特点是能够根据上一次接收数据的大小来自动调整下一次缓冲区创建时分配的空间大小，从而避免内存浪费。

再来看一个问题：在入站处理完成时，入站的ByteBuf是如何自动释放的呢？

方式一：TailContext自动释放

Netty默认会在ChannelPipeline的最后添加一个TailContext（尾部上下文，也是一个入站处理器）。它实现了默认的入站处理方法，在这些方法中会帮助完成ByteBuf内存释放的工作，具体如图5-21所示。

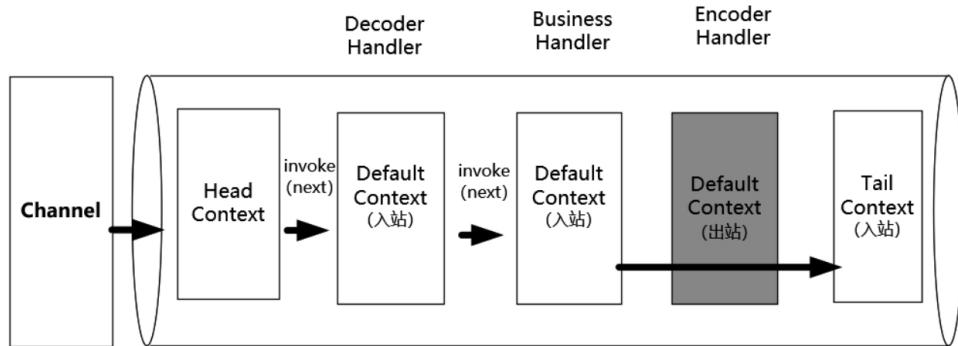


图5-21 TailContext帮助释放缓冲区

所以，只要最初的ByteBuf数据包一路向后传递，进入流水线的末端，TailContext（末尾处理器）就会自动释放掉入站的ByteBuf实例。其源码大致如下：

```
//流水线实现类

public class DefaultChannelPipeline implements ChannelPipeline
{

    //内部类：尾部处理器和尾部上下文是同一个类
    final class TailContext extends AbstractChannelHandlerContext
        implements ChannelInboundHandler {

        //入站处理方法：读取通道
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) {
            onUnhandledInboundMessage(ctx, msg);
        }
    }

    ...
}
```

```
}

...
//入站消息没有被处理，或者说来到了流水线末尾，释放缓冲区
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(...);
    } finally {
        //释放缓冲区
        ReferenceCountUtil.release(msg);
    }
}

...
}
```

说明

以上的TailContext源码来自Netty的4.1.49版本，其他版本的源码可能会有微小的区别，比如说4.0.33版本的源码就有所不同。虽然代码不同，但是干的活都是类似的，就是需要进行缓冲区的释放。

如何让ByteBuf数据包通过流水线一路向后传递，到达末尾的TailContext呢？如果自定义的InboundHandler（入站处理器）继承自ChannelInboundHandlerAdapter适配器，那么可以在入站处理方法中调用基类的入站处理方法，演示代码如下：

```
public class DemoHandler extends ChannelInboundHandlerAdapter {  
    /**  
     * 出站处理方法  
     * @param ctx 上下文  
     * @param msg 入站数据包  
     * @throws Exception 可能抛出的异常  
     */  
  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) {...  
    ByteBuf byteBuf = (ByteBuf) msg;  
    //省略ByteBuf的业务处理  
    //调用父类的入站方法，默认的动作是将msg向下一站传递，一直到末端  
    super.channelRead(ctx, msg);  
    //方式二：手动释放ByteBuf  
    //byteBuf.release();  
}  
}
```

当然，如果没有调用父类的入站处理方法将ByteBuf缓存区向后传递，则需要手动进行释放。

如果Handler业务处理器需要截断流水线的处理流程，不将ByteBuf数据包送入流水线末端的TailContext入站处理器，并且也不愿意手动释放ByteBuf缓冲区实例，那么该怎么办呢？继承SimpleChannelInboundHandler，利用它的自动释放功能来完成。

方式二：SimpleChannelInboundHandler自动释放

以入站读数据为例，Handler业务处理器可以继承自 SimpleChannelInboundHandler基类，此时必须将业务处理代码移动到重写的channelRead0(ctx, msg)方法中。

SimpleChannelInboundHandle类的入站处理方法（如channelRead等）会在调用完实际的channelRead0()方法后帮忙释放ByteBuf实例。如果想看看SimpleChannelInboundHandler是如何释放ByteBuf的，那么可以看看Netty源代码。截取的部分代码如下：

```
public abstract class SimpleChannelInboundHandler<I>
extends ChannelInboundHandlerAdapter
{
    //基类的入站方法
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object
msg) ...{
        boolean release = true;
        try {
            if (acceptInboundMessage(msg)) {
                @SuppressWarnings("unchecked")
                I imsg = (I) msg;
                //调用实际的业务代码，必须由子类提供实现
                channelRead0(ctx, imsg);
            } else {
                release = false;
                ctx.fireChannelRead(msg);
            }
        }
    }
}
```

```
        } finally {
            if (autoRelease&& release) {
                //释放ByteBuf
                ReferenceCountUtil.release(msg);
            }
        }
    ...
}
```

在Netty的SimpleChannelInboundHandler类的源代码中，执行完子类的channelRead0()业务处理后，在finally语句代码段中ByteBuf被释放了一次，如果ByteBuf计数器为零，就将被彻底释放掉。

2. 出站处理时的自动释放

出站缓冲区的自动释放方式是HandlerContext自动释放。出站处理用到的ByteBuf缓冲区一般是要发送的消息，通常是由Handler业务处理器所申请分配的。例如，通过write()方法写入流水线时，调用ctx.writeAndFlush(ByteBuf msg)，就会让ByteBuf缓冲区进入流水线的出站处理流程。在每一个出站Handler业务处理器中的处理完成后，数据包（或消息）会来到出站处理的最后一棒HandlerContext，在完成数据输出到通道之后，ByteBuf会被释放一次，如果计数器为零，就将被彻底释放掉，如图5-22所示。

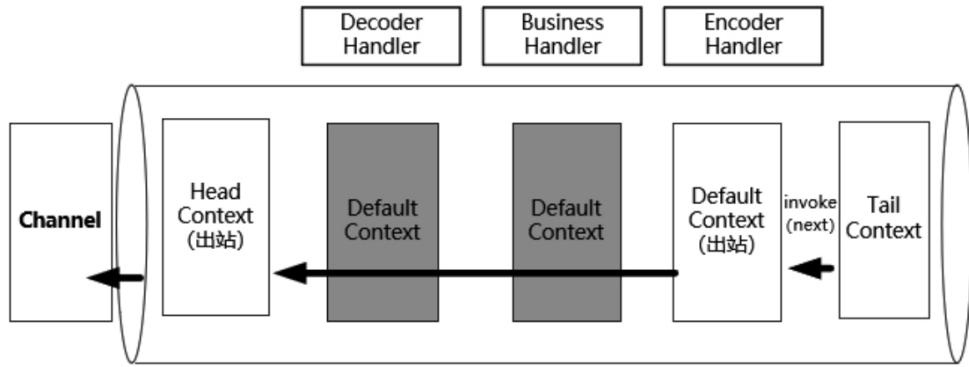


图5-22 HeadContext头部处理器帮助释放ByteBuffer缓冲区

在出站处理的流水处理过程中，在最终进行写入刷新的时候，HeadContext要通过通道实现类自身实现的doWrite()方法将ByteBuf缓冲区的字节数据发送出去（比如复制到内部的Java NIO通道），发送完成后，doWrite()方法就会减少ByteBuf缓冲区的引用计数，代码大致如下：

```

public abstract class AbstractNioByteChannel
extends AbstractNioChannel {

    //执行二进制字节内容的写入，写入Java NIO通道
    @Override
    protected void doWrite(ChannelOutboundBuffer in) ...{
        int writeSpinCount = -1;
        boolean setOpWrite = false;
        //死循环：发送缓冲区的数据，直到缓冲区发送完毕
        for (;;) {
            Object msg = in.current();
            ...
            if (msg instanceof ByteBuf) {
                ByteBuf buf = (ByteBuf) msg;

```

```
    int readableBytes = buf.readableBytes();  
  
    //发送完毕  
    if (readableBytes == 0) {  
  
        //remove()里边包含释放msg的引用减少代码  
        //具体为: ReferenceCountUtil.safeRelease(msg);  
        in.remove();  
        continue;  
    }  
  
    ...  
    //发送缓冲区的字节数据到Java NIO通道  
    int localFlushedAmount = doWriteBytes(buf);  
  
    ...  
} else if (msg instanceof FileRegion) {  
  
    ...  
} else {  
    //Should not reach here.  
    throw new Error();  
}  
}  
  
...  
}  
  
//发送缓冲区的字节数据，将其复制到Java NIO通道  
@Override  
protected int doWriteBytes(ByteBuf buf) ...{
```

```
    final int expectedWrittenBytes = buf.readableBytes();
    //复制数据到Java NIO通道，相当于发送到Java NIO通道
    return buf.readBytes(javaChannel(),
expectedWrittenBytes);
}
}

...
}
```

总之，在Netty应用开发中，必须密切关注ByteBuf缓冲区的释放。如果释放不及时，就会造成Netty的内存泄漏（Memory Leak），最终导致内存耗尽。

5.7.11 ByteBuf浅层复制的高级使用方式

首先说明浅层复制是一种非常重要的操作，可以很大程度地避免内存复制。这一点对于大规模消息通信来说是非常重要的。ByteBuf的浅层复制分为两种：切片（slice）浅层复制和整体（duplicate）浅层复制。

1. 切片浅层复制

ByteBuf的slice()方法可以获取到一个ByteBuf的切片。一个ByteBuf可以进行多次切片浅层复制；多次切片后的ByteBuf对象可以共享一个存储区域。

slice()方法有两个重载版本：

- (1) public ByteBuf slice()
- (2) public ByteBuf slice(int index, int length)

第一个是不带参数的slice()方法，在内部调用了带参数的重载版本，调用大致方式为：

```
public abstract class AbstractByteBuf extends ByteBuf {  
    ...  
    @Override  
    public ByteBuf slice() {  
        return slice(readerIndex,  
readableBytes());  
    }  
}
```

也就是说，第一个无参数slice()方法的返回值是ByteBuf实例中可读部分的切片。带参数的slice(int index, int length)方法可以通过灵活地设置不同起始位置和长度来获取到ByteBuf不同区域的切片。

一个简单的slice的使用示例代码如下：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
public class SliceTest {  
    @Test  
    public void testSlice() {  
        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(9,
```

```
100);

    print("动作： 分配ByteBuf(9, 100)", buffer);

    buffer.writeBytes(new byte[]{1, 2, 3, 4});

    print("动作： 写入4个字节 (1,2,3,4)", buffer);

    ByteBuf slice = buffer.slice();

    print("动作： 切片 slice", slice);

}

}
```

在上面的代码中，输出了源ByteBuf和调用slice()方法后的切片ByteBuf的三组属性值，运行结果如下：

```
//省略了ByteBuf刚分配后的属性值输出

[main|...]: after =====动作： 写入4个字节 (1,2,3,4)=====

[main|...]: 1.0 isReadable(): true

[main|...]: 1.1 readerIndex(): 0

[main|...]: 1.2 readableBytes(): 4

[main|...]: 2.0 isWritable(): true

[main|...]: 2.1 writerIndex(): 4

[main|...]: 2.2 writableBytes(): 5

[main|...]: 3.0 capacity(): 9

[main|...]: 3.1 maxCapacity(): 100

[main|...]: 3.2 maxWritableBytes(): 96

[main|...]: after =====动作： 切片 slice=====

[main|...]: 1.0 isReadable(): true

[main|...]: 1.1 readerIndex(): 0

[main|...]: 1.2 readableBytes(): 4
```

```
[main|...]: 2.0 isWritable(): false  
[main|...]: 2.1 writerIndex(): 4  
[main|...]: 2.2 writableBytes(): 0  
[main|...]: 3.0 capacity(): 4  
[main|...]: 3.1 maxCapacity(): 4  
[main|...]: 3.2 maxWritableBytes(): 0
```

调用slice()方法后，返回的切片是一个新的ByteBuf对象，该对象的几个重要属性值大致如下：

- `readerIndex`（读指针）值为0。
- `writerIndex`（写指针）值为源ByteBuf的`readableBytes()`可读字节数。
- `maxCapacity`（最大容量）值为源ByteBuf的`readableBytes()`可读字节数。

切片后的新ByteBuf有两个特点：

- 切片不可以写入，原因是：`maxCapacity`与`writerIndex`值相同。
- 切片和源ByteBuf的可读字节数相同，原因是：切片后的可读字节数为自己的属性`writerIndex - readerIndex`，也就是源ByteBuf的`readableBytes() - 0`。

切片后的新ByteBuf和源ByteBuf的关联性如下：

- 切片不会复制源ByteBuf的底层数据，底层数组和源ByteBuf的底层数组是同一个。
- 切片不会改变源ByteBuf的引用计数。

从根本上说，slice()无参数方法所生成的切片就是源ByteBuf可读部分的浅层复制。

2. 整体浅层复制

和slice切片不同，duplicate()方法返回的是源ByteBuf的整个对象的一个浅层复制，包括如下内容：

- Duplicate()的读写指针、最大容量值，与源ByteBuf的读写指针相同。
- duplicate()不会改变源ByteBuf的引用计数。
- duplicate()不会复制源ByteBuf的底层数据。

duplicate()和slice()方法都是浅层复制。不同的是，slice()方法是切取一段的浅层复制，而duplicate()是整体的浅层复制。

3. 浅层复制的问题

浅层复制方法不会实际去复制数据，也不会改变ByteBuf的引用计数，会导致一个问题：在源ByteBuf调用release()方法之后，一旦引用计数为零，就变得不能访问了；在这种场景下，源ByteBuf的所有浅层复制实例也不能进行读写了；如果强行对浅层复制实例进行读写，则会报错。

因此，在调用浅层复制实例时，可以通过调用一次retain()方法来增加引用，表示它们对应的底层内存多了一次引用，引用计数为2。在浅层复制实例用完后，需要调用两次release()方法，将引用计数减1，这样就不会影响源ByteBuf的内存释放了。

5.8 Netty的零拷贝

大部分场景下，在Netty接收和发送ByteBuffer的过程中会使用直接内存进行Socket通道读写，使用JVM的堆内存进行业务处理，会涉及直接内存、堆内存之间的数据复制。内存的数据复制其实是效率非常低的，Netty提供了多种方法，以帮助应用程序减少内存的复制。

Netty的零拷贝（Zero-Copy）主要体现在五个方面：

(1) Netty提供CompositeByteBuf组合缓冲区类，可以将多个ByteBuf合并为一个逻辑上的ByteBuf，避免了各个ByteBuf之间的拷贝。

(2) Netty提供了ByteBuf的浅层复制操作（slice、duplicate），可以将ByteBuf分解为多个共享同一个存储区域的ByteBuf，避免内存的拷贝。

(3) 在使用Netty进行文件传输时，可以调用FileRegion包装的transferTo()方法直接将文件缓冲区的数据发送到目标通道，避免普通的循环读取文件数据和写入通道所导致的内存拷贝问题。

(4) 在将一个byte数组转换为一个ByteBuf对象的场景下，Netty提供了一系列的包装类，避免了转换过程中的内存拷贝。

(5) 如果通道接收和发送ByteBuf都使用直接内存进行Socket读写，就不需要进行缓冲区的二次拷贝。如果使用JVM的堆内存进行Socket读写，那么JVM会先将堆内存Buffer拷贝一份到直接内存再写入Socket中，相比于使用直接内存，这种情况在发送过程中会多出一次

缓冲区的内存拷贝。所以，在发送ByteBuffer到Socket时，尽量使用直接内存而不是JVM堆内存。

说明

Netty中的零拷贝和操作系统层面上的零拷贝是有区别的，不能混淆，我们所说的Netty零拷贝完全是基于Java层面或者说用户空间的，它更多的是偏向于应用中的数据操作优化，而不是系统层面的操作优化。

5.8.1 通过CompositeByteBuf实现零拷贝

CompositeByteBuf可以把需要合并的多个ByteBuf组合起来，对外提供统一的readIndex和writerIndex。CompositeByteBuf只是在逻辑上是一个整体，在CompositeByteBuf内部，合并的多个ByteBuf都是单独存在的。CompositeByteBuf里面有一个Component数组，聚合的ByteBuf都放在Component数组里面，最小容量为16。

在很多通信编程场景下，需要多个ByteBuf组成一个完整的消息。例如，HTTP协议传输时消息总是由Header（消息头）和Body（消息体）组成。如果传输的内容很长，就会分成多个消息包进行发送，消息中的Header就需要重用，而不是每次发送都创建新的Header缓冲区。这时可以使用CompositeByteBuf缓冲区进行ByteBuf组合，避免内存拷贝。

假设有一份协议数据，它由头部和消息体组成，而头部和消息体是分别存放在两个ByteBuf中的，为了方便后续处理，要将两个ByteBuf进行合并，具体如图5-23所示。

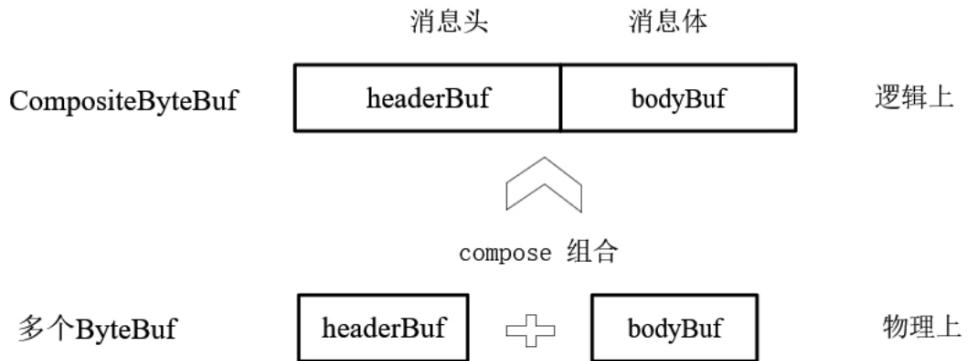


图5-23 `CompositeByteBuf`实现合并`ByteBuf`

使用`CompositeByteBuf`合并多个`ByteBuf`，大致的代码如下：

```
ByteBuf headerBuf = ...  
ByteBuf bodyBuf = ...  
CompositeByteBuf compositeByteBuf = Unpooled.compositeBuffer();  
cbuf.addComponents(headerBuf, bodyBuf);
```

不使用`CompositeByteBuf`，将header和body合并为一个`ByteBuf`的代码大致如下：

```
ByteBuf headerBuf = ...  
ByteBuf bodyBuf = ...  
long length=headerBuf.readableBytes() +  
bodyBuf.readableBytes();  
ByteBuf allBuf = Unpooled.buffer(length);
```

```
allBuf.writeBytes(headerBuf) ; //拷贝header数据  
allBuf.writeBytes(body) ; //拷贝body数据
```

上述过程将header和body都拷贝到了新的allBuf中，这增加了两次额外的数据拷贝操作。所以，使用CompositeByteBuf合并ByteBuf可以减少两次额外的数据拷贝操作。

下面是一段通过CompositeByteBuf来复用header的比较完整的演示代码：

```
package com.crazymakercircle.netty.bytebuf;  
  
//...  
  
public class CompositeBufferTest {  
  
    static Charset utf8 = Charset.forName("UTF-8");  
  
    @Test  
  
    public void byteBufComposite() {  
  
        CompositeByteBuf cbuf =  
ByteBufAllocator.DEFAULT.compositeBuffer();  
  
        //消息头  
  
        ByteBuf headerBuf = Unpooled.copiedBuffer("疯狂创客圈:",  
utf8);  
  
        //消息体1  
  
        ByteBuf bodyBuf = Unpooled.copiedBuffer("高性能Netty",  
utf8);  
  
        cbuf.addComponents(headerBuf, bodyBuf);  
  
        sendMsg(cbuf);  
  
        //在refCnt为0前，retain  
  
        headerBuf.retain();
```

```

        cbuf.release();

        cbuf = ByteBufAllocator.DEFAULT.compositeBuffer();
        //消息体2
        bodyBuf = Unpooled.copiedBuffer("高性能学习社群", utf8);
        cbuf.addComponents(headerBuf, bodyBuf);
        sendMsg(cbuf);
        cbuf.release();
    }

private void sendMsg(CompositeByteBuf cbuf) {
    //处理整个消息
    for (ByteBuf b : cbuf) {
        int length = b.readableBytes();
        byte[] array = new byte[length];
        //将CompositeByteBuf中的数据统一复制到数组中
        b.getBytes(b.readerIndex(), array);
        //处理一下数组中的数据
        System.out.print(new String(array, utf8));
    }
    System.out.println();
}
}

```

在上面的程序中，调用CompositeByteBuf的addComponents()方法向自身增加了ByteBuf对象实例。对于所添加的ByteBuf，Heap ByteBuf、Direct ByteBuf均可。

如果CompositeByteBuf内部只存在一个ByteBuf，则调用其hasArray()方法，返回的是这个唯一实例hasArray()方法的值；如果有多个ByteBuf，则其hasArray()方法会返回false。

另外，调用CompositeByteBuf的nioBuffer()方法可以将CompositeByteBuf实例合并成一个新的NIO ByteBuffer缓冲区（注意：不是Netty的ByteBuf缓冲区）。演示代码如下：

```
package com.crazymakercircle.netty.bytebuf;  
//...  
  
public class CompositeBufferTest {  
  
    @Test  
  
    public void intCompositeBufComposite() {  
  
        CompositeByteBuf cbuf = Unpooled.compositeBuffer(3);  
  
        cbuf.addComponent(Unpooled.wrappedBuffer(new byte[] {1,  
2, 3}));  
  
        cbuf.addComponent(Unpooled.wrappedBuffer(new byte[]  
{4}));  
  
        cbuf.addComponent(Unpooled.wrappedBuffer(new byte[] {5,  
6}));  
  
        //合并成一个的Java NIO缓冲区  
  
        ByteBuffer nioBuffer = cbuf.nioBuffer(0, 6);  
  
        byte[] bytes = nioBuffer.array();  
  
        System.out.print("bytes = ");  
  
        for (byte b : bytes) {  
  
            System.out.print(b);  
  
        }  
    }  
}
```

```
    cbuf.release();  
}  
}
```

5.8.2 通过wrap操作实现零拷贝

Unpooled提供了一系列的wrap包装方法，可以帮助大家方便、快速地包装出CompositeByteBuf实例或者ByteBuf实例，而不用进行内存拷贝。

Unpooled包装CompositeByteBuf的操作使用起来更加方便。例如，上一小节的header与body的组合可以调用Unpooled.wrappedBuffer()方法。大致的代码如下：

```
ByteBuf headerBuf = ...  
ByteBuf bodyBuf = ...  
ByteBuf allByteBuf = Unpooled.wrappedBuffer(headerBuf, bodyBuf  
);
```

Unpooled类提供了很多重载的wrappedBuffer()方法，将多个ByteBuf包装为CompositeByteBuf实例，从而实现零拷贝。这些重载方法大致如下：

```
public static ByteBuf wrappedBuffer(ByteBuffer buffer)  
public static ByteBuf wrappedBuffer(ByteBuf buffer)  
public static ByteBuf wrappedBuffer(ByteBuf... buffers)  
public static ByteBuf wrappedBuffer(ByteBuffer... buffers)
```

除了通过Unpooled包装CompositeByteBuf之外，还可以将byte数组包装成ByteBuf。如果将一个byte数组转换为一个ByteBuf对象，大致的代码如下：

```
byte[] bytes = ...  
ByteBuf byteBuf = Unpooled.wrappedBuffer(bytes);
```

通过调用Unpooled.wrappedBuffer()方法将bytes包装为一个UnpooledHeapByteBuf对象，在包装的过程中不会有拷贝操作，所得到的ByteBuf对象和bytes数组共用同一个存储空间，对bytes的修改也是对ByteBuf对象的修改。

如果不是调用Unpooled.wrappedBuffer()包装方法，那么传统的做法是将此byte数组的内容拷贝到ByteBuf中，大致的代码如下：

```
byte[] bytes = ...  
ByteBuf byteBuf = Unpooled.buffer();  
byteBuf.writeBytes(bytes);
```

显然，传统的转换方式是有额外的内存申请和拷贝操作的，既浪费了内存空间，又需要耗费内存复制的时间。相对而言，Unpooled提供的wrap操作既复用了空间，又节省了时间。

Unpooled提供了多个包装字节数组的重载方法，大致如下：

```
public static ByteBuf wrappedBuffer(byte[] array)  
public static ByteBuf wrappedBuffer(byte[] array, int offset,  
int length)  
public static ByteBuf wrappedBuffer(byte[]... arrays)
```

Unpooled类还提供了一些其他的避免零拷贝的方法，具体可以参见其源码，这里不再赘述。

5.9 EchoServer的实战案例

前面实现过Java NIO版本的EchoServer，在学习了Netty的原理和基本使用后，这里为大家设计和实现一个Netty版本的EchoServer。

5.9.1 NettyEchoServer

首先回顾一下NettyEchoServer的功能，很简单：服务端读取客户端输入的数据，然后将数据直接回显到Console控制台。此实战案例的目标是帮助大家掌握以下知识：

- 服务端**ServerBootstrap**的装配和使用。
- 服务端**NettyEchoServerHandler**入站处理器的**channelRead**入站处理方法的编写。
- **Netty**的**ByteBuf**缓冲区的读取、写入，以及**ByteBuf**引用计数的查看。

首先是服务端的**ServerBootstrap**装配和启动过程，代码如下：

```
package com.crazymakercircle.netty.echoServer;  
//...  
  
public class NettyEchoServer {  
    //...  
  
    public void runServer() {  
        //创建反应器轮询组  
        EventLoopGroup bossLoopGroup = new
```

```

NioEventLoopGroup(1);

        EventLoopGroup workerLoopGroup = new
NioEventLoopGroup();

        //省略设置：1 反应器轮询组/2 通道类型/4 通道选项等
        //5 装配子通道流水线

        b.childHandler(new

ChannelInitializer<SocketChannel>() {
            //有连接到达时会创建一个通道

            protected void
initChannel(SocketChannel ch) ...{
                //管理子通道中的Handler

                //向子通道流水线添加一个Handler

ch.pipeline().addLast(NettyEchoServerHandler.INSTANCE);

            }
        });

        //省略启动、等待、优雅关闭等
    }

//省略 main()方法
}

```

5.9.2 NettyEchoServerHandler

Netty EchoServerHandler入站处理器继承自 ChannelInboundHandlerAdapter，实现了channelRead()入站读方法（在可读IO事件到来时将被流水线回调）。

回显服务器处理器的逻辑分为两步：

(1) 第一步，读取从对端输入的数据。channelRead()方法的msg参数的形参类型不是ByteBuf，而是Object，这是由流水线的上一站决定的。一般而言，入站处理的流程是：Netty读取底层的二进制数据，填充到msg时，msg是ByteBuf类型，然后经过流水线，传入第一个入站处理器；每一个节点处理完后，将自己的处理结果（类型不一定是ByteBuf）作为msg参数不断向后传递。因此，msg参数的形参类型只能是Object类型。第一个入站处理器的channelRead()方法的msg类型绝对是ByteBuf类型，因为它是Netty读取到的ByteBuf数据包。在本实例中，NettyEchoServerHandler就是第一个业务处理器，虽然msg的实参类型是Object，但是实际类型就是ByteBuf，所以可以强制转成ByteBuf类型。

另外，从Netty 4.1开始，ByteBuf的默认类型是Direct ByteBuf。注意，Java不能直接访问Direct ByteBuf内部的数据，必须通过调用getBytes()、readBytes()等方法将数据读入Java数组中才能继续进行处理。

(2) 第二步，将数据写回客户端。这一步很简单，直接复用前面的msg实例即可。不过要注意，如果上一步调用的是readBytes()方法，那么这一步就不能直接将msg写回了，因为数据已经被readBytes()方法读完了。幸好，上一步调用的读数据方法是getBytes()，它不影响ByteBuf的数据指针，因此可以继续使用。这里除了调用ctx.writeAndFlush()方法把msg数据写回客户端之外，也可调用通道的ctx.channel().writeAndFlush()方法发送数据。这两种方法在这里的效果是一样的，因为这个流水线上没有任何出站处理器。

服务端的入站处理器NettyEchoServerHandler的代码如下：

```
package com.crazymakercircle.netty.echoServer;  
//...  
@ChannelHandler.Sharable  
public class NettyEchoServerHandler  
    extends  
    ChannelInboundHandlerAdapter {  
    public static final NettyEchoServerHandler INSTANCE  
        = new  
        NettyEchoServerHandler();  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) {...  
    ByteBuf in = (ByteBuf) msg;  
    Logger.info("msg type: " + (in.hasArray()?"堆内存":"直接  
内存"));  
    int len = in.readableBytes();  
    byte[] arr = new byte[len];  
    in.getBytes(0, arr);  
    Logger.info("server received: " + new String(arr, "UTF-  
8"));  
  
    Logger.info("写回前, msg.refCnt:" + ((ByteBuf)  
msg).refCnt());  
    //写回数据, 异步任务  
    ChannelFuture f = ctx.writeAndFlush(msg);
```

```
f.addListener((ChannelFuture futureListener) -> {
    Logger.info("写回后, msg.refCnt:" + ((ByteBuf)
msg).refCnt());
}) ;
}
```

NettyEchoServerHandler加了一个特殊的Netty注解：
@ChannelHandler.Sharable。这个注解的作用是标注一个Handler实例可以被多个通道安全地共享（多个通道的流水线可以加入同一个Handler实例）。这种共享操作，Netty默认是不允许的。

很多应用场景都需要Handler实例能共享。例如，一个服务器处理十万以上的通道，如果每一个通道都新建很多重复的Handler实例，就会浪费很多宝贵的空间，降低了服务器的性能。所以，如果在Handler实例中没有与特定通道强相关的数据或者状态，建议设计成共享模式。

如果没有加@ChannelHandler.Sharable注解，试图将同一个Handler实例添加到多个ChannelPipeline时，Netty将会抛出异常。

如何判断一个Handler是否为@Sharable呢？
ChannelHandlerAdapter提供了实用方法——isSharable()。如果其对应的实现加上了@Sharable注解，那么这个方法将返回true，表示它可以被添加到多个ChannelPipeline中。

NettyEchoServerHandler没有保存与任何通道连接相关的数据，也没有内部的其他数据需要保存。所以，该处理器不仅仅可以用来共

享，而且不需要做任何同步控制。这里为它加上了@Sharable注解，表示可以共享。更进一步，这里还设计了一个通用的INSTANCE静态实例，所有的通道直接使用这个实例即可。

5.9.3 NettyEchoClient

客户端的实战案例可以帮助大家掌握以下知识：

- 客户端Bootstrap的装配和使用。
- 在客户端NettyEchoClientHandler入站处理器中接收回写的数据，并且释放内存。
- 有多种方式可以用于释放ByteBuf，包括自动释放和手动释放。

客户端Bootstrap的装配和使用代码如下：

```
package com.crazymakercircle.netty.echoServer;  
//...  
  
public class NettyEchoClient {  
  
    private int serverPort;  
    private String serverIp;  
    Bootstrap b = new Bootstrap();  
  
    public NettyEchoClient(String ip, int port) {  
        this.serverPort = port;  
        this.serverIp = ip;  
    }  
}
```

```
public void runClient() {  
    //创建反应器轮询组  
  
    EventLoopGroup workerLoopGroup = new  
    NioEventLoopGroup();  
  
    try {  
        //1 设置反应器轮询组  
        b.group(workerLoopGroup);  
        //2 设置nio类型的通道  
        b.channel(NioSocketChannel.class);  
        //3 设置监听端口  
        b.remoteAddress(serverIp, serverPort);  
        //4 设置通道的参数  
        b.option(ChannelOption.ALLOCATOR,  
                 PooledByteBufAllocator.DEFAULT);  
  
        //5 装配子通道流水线  
        b.handler(new ChannelInitializer<SocketChannel>() {  
            //有连接到达时会创建一个通道  
            protected void initChannel(SocketChannel ch)...{  
                //管理子通道中的Handler  
                //向子通道流水线添加一个Handler  
  
                ch.pipeline().addLast(NettyEchoClientHandler.INSTANCE);  
            }  
        });  
    }  
}
```

```
ChannelFuture f = b.connect();
f.addListener((ChannelFutureListener) ->
{
    if (futureListener.isSuccess()) {
        Logger.info("EchoClient客户端连接成功!");
    } else {
        Logger.info("EchoClient客户端连接失败!");
    }
}) ;

//阻塞，直到连接成功
f.sync();

Channel channel = f.channel();
Scanner scanner = new Scanner(System.in);
Print.tcf("请输入发送内容:");
while (scanner.hasNext()) {
    //获取输入的内容
    String next = scanner.next();
    byte[] bytes = (Dateutil.getNow() + " >>" +
                    + next).getBytes("UTF-8");
    //发送ByteBuf
    ByteBuf buffer = channel.alloc().buffer();
    buffer.writeBytes(bytes);
    channel.writeAndFlush(buffer);
    Print.tcf("请输入发送内容:");
}
} catch (Exception e) {
```

```
        e.printStackTrace();

    } finally {
        //优雅关闭EventLoopGroup,
        //释放掉所有资源，包括创建的线程
        workerLoopGroup.shutdownGracefully();
    }
}

//省略 main()方法
}
```

在上面的代码中，客户端在成功连接到服务端后不断循环获取控制台的输入，通过与服务端之间的连接通道发送到服务器。

5.9.4 NettyEchoClientHandler

客户端接收服务器回显的数据包，显示在Console控制台上，所以客户端的处理器流水线不是空的，还需要装配一个回显处理器。该处理的功能很简单，代码如下：

```
package com.crazymakercircle.netty.echoServer;
//省略import

@ChannelHandler.Sharable
public class NettyEchoClientHandler extends
        ChannelInboundHandlerAdapter {

    public static final NettyEchoClientHandler INSTANCE
            = new NettyEchoClientHandler();

    //入站处理方法
}
```

```
@Override  
public void channelRead(ChannelHandlerContext ctx, Object  
msg) {...  
    ByteBuf byteBuf = (ByteBuf) msg;  
    int len = byteBuf.readableBytes();  
    byte[] arr = new byte[len];  
    byteBuf.getBytes(0, arr);  
    Logger.info("client received: " + new String(arr, "UTF-  
8"));  
  
    //释放ByteBuf的两种方法  
    //方法一：手动释放ByteBuf  
    byteBuf.release();  
  
    //方法二：调用父类的入站方法，将msg向后传递  
    //super.channelRead(ctx, msg);  
}
```

通过代码可以看到，从服务端发送过来的ByteBuf被手动方式强制释放了。当然，也可以使用前面介绍的自动释放方式来释放ByteBuf。

第6章 Decoder与Encoder核心组件

Netty从底层Java通道读取ByteBuf二进制数据，传入Netty通道的流水线，随后开始入站处理。在入站处理过程中，需要将ByteBuf二进制类型解码成Java POJO对象。这个解码过程可以通过Netty的Decoder（解码器）去完成。

在出站处理过程中，业务处理后的结果（出站数据）需要从某个Java POJO对象编码为最终的ByteBuf二进制数据，然后通过底层Java通道发送到对端。在编码过程中，需要用到Netty的Encoder（编码器）去完成数据的编码工作。

本章专门为家解读Netty非常核心的组件：编码器和解码器。

6.1 Decoder原理与实战

什么是Netty的解码器呢？

首先，它是一个InBound入站处理器，负责处理“入站数据”。

其次，它能将上一站Inbound入站处理器传过来的输入（Input）数据进行解码或者格式转换，然后发送到下一站Inbound入站处理器。

一个标准的解码器的职责为：将输入类型为ByteBuf的数据进行解码，输出一个一个的Java POJO对象。Netty内置了ByteToMessageDecoder解码器。

Netty中的解码器都是Inbound入站处理器类型，都直接或者间接地实现了入站处理的超级接口ChannelInboundHandler。

6.1.1 ByteToMessageDecoder解码器处理流程

ByteToMessageDecoder是一个非常重要的解码器基类，是一个抽象类，实现了解码处理的基础逻辑和流程。ByteToMessageDecoder继承自ChannelInboundHandlerAdapter适配器，是一个入站处理器，用于完成从ByteBuf到Java POJO对象的解码功能。

ByteToMessageDecoder解码的流程大致如图6-1所示。首先，它将上一站传过来的输入到ByteBuf中的数据进行解码，解码出一个List<Object>对象列表；然后，迭代List<Object>列表，逐个将Java POJO对象传入下一站Inbound入站处理器。

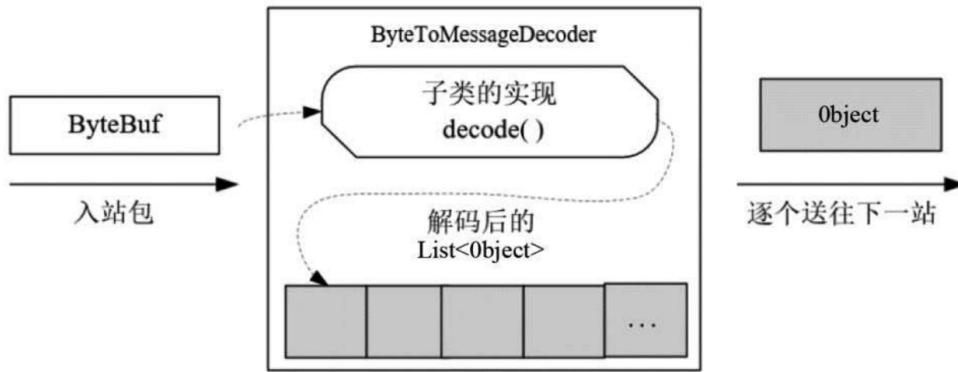


图6-1 ByteToMessageDecoder解码的流程

ByteToMessageDecoder是抽象类，不能以实例化方式创建对象。也就是说，直接通过ByteToMessageDecoder类并不能完成ByteBuf字节码到具体Java类型的解码，还得依赖于它的具体实现。

ByteToMessageDecoder的解码方法为`decode()`，是一个抽象方法。也就是说，对于`decode()`方法的具体解码过程，ByteToMessageDecoder没有具体的实现，如何将ByteBuf中的字节数据变成什么样的Object实例（包含多少个Object实例）需要子类去完成。所以，作为解码器的父类，ByteToMessageDecoder仅仅提供了一个整体框架：它会调用子类的`decode()`方法，完成具体的二进制字节解码，然后会获取子类解码之后的Object结果，放入自己内部的结果列表`List<Object>`中，最终父类负责将`List<Object>`中的元素一个一个地传递给下一站。从这个角度来说，ByteToMessageDecoder在设计上使用了模板模式（Template Pattern）。

ByteToMessageDecoder的子类要做的是将从入站ByteBuf解码出来的所有Object实例加入父类的`List<Object>`列表中。

实现一个解码器，首先要继承ByteToMessageDecoder抽象类，然后实现其基类的decode()抽象方法。总体来说，流程大致如下：

- (1) 继承ByteToMessageDecoder抽象类。
- (2) 实现基类的decode()抽象方法，将ByteBuf到目标POJO的解码逻辑写入此方法，以将ByteBuf中的二进制数据解码成一个一个的Java POJO对象。
- (3) 解码完成后，需要将解码后的Java POJO对象放入decode()方法的List<Object>实参中，此实参是父类所传入的解码结果收集容器。

余下的工作都由父类ByteToMessageDecoder自动完成。在流水线的处理过程中，父类在执行完子类的解码后，会将List<Object>收集到的结果一个一个地传递到下一个Inbound入站处理器。

6.1.2 自定义Byte2IntegerDecoder整数解码器

下面是一个小小的ByteToMessageDecoder子类的实战案例：整数解码器。其功能是将ByteBuf中的字节解码成整数类型。

Byte2IntegerDecoder整数解码器的代码很简单，具体如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class Byte2IntegerDecoder extends ByteToMessageDecoder {  
    @Override  
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
```

```
        List<Object> out) {  
            while (in.readableBytes() >= 4) {  
                int i = in.readInt();  
                Logger.info("解码出一个整数: " + i);  
                out.add(i);  
            }  
        }  
    }  
}
```

上面实战案例程序中，decode()方法的逻辑大致如下：

首先，Byte2IntegerDecoder解码器继承自ByteToMessageDecoder。

其次，在decode()方法中，通过ByteBuf的readInt()实例方法从输入缓冲区读取整数，其作用是将二进制数据解码成一个一个的整数。

再次，将解码后的整数增加到decode()方法的List<Object>列表参数中。

最后，decode()方法不断地循环解码，并且不断地添加到List<Object>结果容器中。

前面反复讲到，decode()方法处理完成后，基类会继续后面的传递处理：将List<Object>结果列表中所得到的整数一个一个地传递到下一个Inbound入站处理器。

至此，一个简单的解码器就完成了。

如何使用这个自定义的Byte2IntegerDecoder解码器呢？首先，需要将其加入通道流水线中；其次，由于解码器的功能仅仅是完成ByteBuf的解码，不做其他业务处理，所以还需要编写一个业务处理器，用于在读取解码后的Java POJO对象之后完成具体的业务处理。

这里编写一个简单的配套处理器IntegerProcessHandler，用于处理Byte2IntegerDecoder解码之后的整数。其功能是：读取上一站的入站数据，把它转换成整数，并且输出到Console（控制台）上。配套处理器的代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class IntegerProcessHandler  
    extends ChannelInboundHandlerAdapter  
{  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) ...{  
        Integer integer = (Integer) msg;  
        Logger.info("打印出一个整数：" + integer);  
    }  
}
```

至此，已经编写了解码处理器Byte2IntegerDecoder和配套处理器IntegerProcessHandler这两个自己的入站处理器：一个负责解码，一个负责模拟处理解码结果。

最终如何测试这两个入站处理器呢？使用EmbeddedChannel（嵌入式通道）编写一个测试用例，代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
  
public class Byte2IntegerDecoderTester {  
  
    /**  
     * 整数解码器的使用实例  
     */  
  
    @Test  
  
    public void testByteToIntegerDecoder() {  
  
        ChannelInitializer i= new  
        ChannelInitializer<EmbeddedChannel>(){  
  
            protected void initChannel(EmbeddedChannel ch) {  
  
                ch.pipeline().addLast(new  
                Byte2IntegerDecoder());  
  
                ch.pipeline().addLast(new  
                IntegerProcessHandler());  
  
            }  
        };  
  
        EmbeddedChannel channel = new EmbeddedChannel(i);  
  
        for (int j = 0; j < 100; j++) {  
  
            ByteBuf buf = Unpooled.buffer();  
  
            buf.writeInt(j);  
  
            channel.writeInbound(buf);  
        }  
        //...  
    }  
}
```

```
    }  
}
```

在测试用例中，新建了一个EmbeddedChannel实例，将Byte2IntegerDecoder和IntegerProcessHandler加入通道的流水线上。

这里请注意先后次序：Byte2IntegerDecoder解码器在前，IntegerProcessHandler处理器在后。为什么呢？因为入站处理的次序为从前到后。

为了测试入站处理器，需要确保通道能接收到ByteBuf入站数据。这里调用writeInbound()方法，模拟入站数据的写入，向EmbeddedChannel写入100次ByteBuf入站缓冲区，每一次写入仅仅包含一个整数。模拟入站数据会被流水线上的两个入站处理器所接收和处理。接着，这些入站的二进制字节被解码成一个一个的整数，逐个输出到控制台上。运行测试用例，部分输出结果如下：

```
//省略部分输出  
[main|Byte2IntegerDecoder:decode]: 解码出一个整数: 0  
[main|IntegerProcessHandler:channelRead]: 打印出一个整数: 0  
[main|Byte2IntegerDecoder:decode]: 解码出一个整数: 1  
[main|IntegerProcessHandler:channelRead]: 打印出一个整数: 1  
[main|Byte2IntegerDecoder:decode]: 解码出一个整数: 2  
[main|IntegerProcessHandler:channelRead]: 打印出一个整数: 2  
[main|Byte2IntegerDecoder:decode]: 解码出一个整数: 3  
[main|IntegerProcessHandler:channelRead]: 打印出一个整数: 3
```

通过这个实例，大家对ByteToMessageDecoder基类以及如何动手去实现一个解码器应该比较了解了，甚至还可以仿照这个例子实现了除了整数之外的Java基本数据类型（Short、Char、Long、Float、Double等）的解码器。

最后说明一下：ByteToMessageDecoder传递给下一站的是解码之后的Java POJO对象，不是ByteBuf缓冲区。那么问题来了，ByteBuf缓冲区并没有发送到流水线的TailContext（尾部处理器），将由谁负责释放引用计数呢？其实，基类ByteToMessageDecoder会完成ByteBuf释放工作，它会调用ReferenceCountUtil.release(in)方法将之前的ByteBuf缓冲区的引用计数减1。

这个ByteBuf先被释放了，如果在后面还需要用到，怎么办？可以在子类的decode()方法中调用一次ReferenceCountUtil.retain(in)来增加一次引用计数，不过在使用完成后要及时将增加的这次计数减去。

6.1.3 ReplayingDecoder解码器

使用上面的Byte2IntegerDecoder整数解码器会面临一个问题：需要对ByteBuf的长度进行检查，有足够的字节才能进行整数的读取。这种长度的判断是否可以由Netty来帮忙完成呢？答案是可以的，可以使用Netty的ReplayingDecoder类省去长度的判断。

ReplayingDecoder类是ByteToMessageDecoder的子类，作用是：

- 在读取ByteBuf缓冲区的数据之前，需要检查缓冲区是否有足够的字节。

- 若ByteBuf中有足够的字节，则会正常读取；反之，则会停止解码。

使用ReplayingDecoder基类改写上一个整数解码器，可以不进行长度检测。创建一个新的整数解码器，类名为Byte2IntegerReplayDecoder，代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class Byte2IntegerReplayDecoder extends ReplayingDecoder  
{  
    @Override  
    public void decode(ChannelHandlerContext ctx,  
                       ByteBuf in, List<Object>  
                       out) {  
        int i = in.readInt();  
        Logger.info("解码出一个整数: " + i);  
        out.add(i);  
    }  
}
```

通过这个示例程序，我们可以看到：继承ReplayingDecoder实现一个解码器，就不用编写长度判断的代码。ReplayingDecoder进行长度判断的原理很简单：内部定义一个新的二进制缓冲区类（类名为ReplayingDecoderBuffer），又对ByteBuf缓冲区进行装饰。该装饰器的特点是，在缓冲区真正读数据之前先进行长度的判断：如果长度合格，就读取数据；否则，抛出ReplayError。ReplayingDecoder捕获到ReplayError后会留着数据，等待下一次IO事件到来时再读取。

简单来讲，ReplayingDecoder对输入的ByteBuf进行了“偷梁换柱”，在将外部传入的ByteBuf缓冲区传给子类之前，换成了自己装饰过的ReplayingDecoderBuffer缓冲区。也就是说，在示例程序中，Byte2IntegerReplayDecoder中的decode()方法所得到的实参in的直接类型并不是原始的ByteBuf类型，而是ReplayingDecoderBuffer类型。

ReplayingDecoderBuffer类型首先是一个内部类，其次是继承了ByteBuf类型，包装了ByteBuf类型的部分读取方法。ReplayingDecoderBuffer对ByteBuf类型的读取方法做了什么样的功能增强呢？主要是进行二进制数据长度的判断，如果长度不足，就抛出异常。这个异常会反过来被ReplayingDecoder基类所捕获，将解码工作停掉。

实质上，ReplayingDecoder的作用远远不止于进行长度判断，它更重要的作用是用于分包传输的应用场景。

6.1.4 整数的分包解码器的实战案例

前面讲到，底层通信协议是分包传输的，一份数据可能分几个数据包到达对端。发送端出去的包在传输过程中会进行多次拆分和组装。接收端收到的包和发送端所发送的包不是一模一样的（见图6-2）：在发送端发出4个字符串，Netty或者NIO接收端可能只接收到了3个ByteBuf数据缓冲。

在Java IO流式传输中，程序若读不到完整的信息则会一直阻塞，而不会继续执行，也就不会出现图6-2所示的问题了。在Java的

NIO（具有非阻塞性）中，保证一次性读取到完整的数据则成了一个大问题。

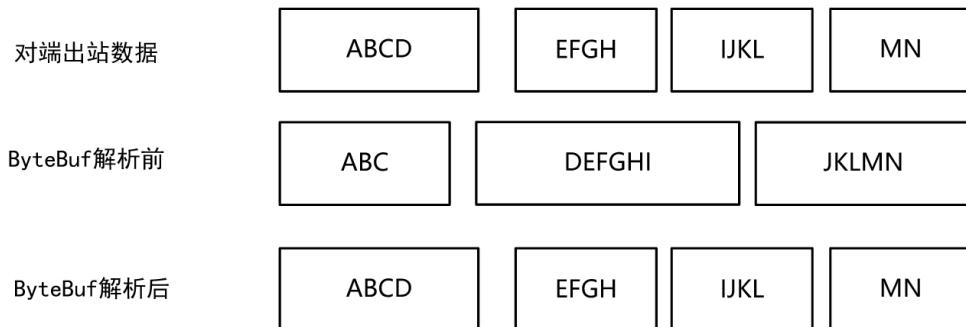


图6-2 通道接收到的ByteBuf数据包和发送端发送的数据包不完全一致

那么，Netty通过什么样的解码器对图6-2中接收端的3个ByteBuf缓冲数据进行解码，而后得到和发送端一模一样的4个字符串呢？理论上可以使用ReplayingDecoder来解决。在进行数据解析时，如果发现当前ByteBuf中所有可读的数据不够，那么ReplayingDecoder会一直等待，直到可读数据是足够的。这一切都是在ReplayingDecoder内部，通过与缓冲区装饰器ReplayingDecoderBuffer相互配合完成的。所以，图6-2展示的字符串错乱问题完全可以通过继承ReplayingDecoder基类实现自己的解码器来解决。

图6-2中的问题是字符串传输过程中出现的，并且实现字符串的解码和纠正相对比较复杂。为了好懂，这里先介绍一个简单点的例子——整数序列解码，并且将它们两两一组进行相加，重点是，解码过程中需要保持发送时的次序。

要完成上述例子，需要用到ReplayingDecoder的一个很重要的属性——state成员属性。该成员属性的作用是保存当前解码器在解码过程中所处的阶段。在Netty源代码中，该属性的定义如下：

```
public abstract class ReplayingDecoder<S>
    extends ByteToMessageDecoder
{
    //省略不相关的代码

    //缓冲区装饰器

    private final ReplayingDecoderByteBuf replayable =
        new
    ReplayingDecoderByteBuf();

    //重要的成员属性，表示解码过程中所处的阶段，类型为泛型，默认为Object
    private S state;

    //默认的构造器，state值为空，没有用到该属性
    protected ReplayingDecoder() {
        this((Object)null);
    }

    //重载的构造器
    protected ReplayingDecoder(S initialState) {
        //初始化内部的ByteBuf缓冲区装饰器类
        this.replayable = new ReplayingDecoderByteBuf();

        //读指针检查点，默认为-1
        this.checkpoint = -1;

        //状态state的默认值为null
        this.state = initialState;
    }
}
```

```
//省略不相关的方法  
}
```

在上一小节定义的整数解码实例中，使用的是默认的无参数构造器，该构造器初始化state成员的值为null，也就是没有用到state属性。本小节将用到state成员属性。为什么呢？因为整数序列的解码工作不可能一次完成，要完成两个整数的提取并相加需要解码两次，每一次解码只能解码出一个整数，只有在第二个整数提取之后才能求和，整个解码工作才算完成。这里存在两个阶段，具体的阶段需要使用state来记录。

具体来说，完成两个整数的提取并求和的过程可以从业务上分成两个阶段。使用state属性来保存目前所处的阶段：如果是第一个阶段，则仅仅提取第一个整数，完成后进入第二个阶段；如果是第二个阶段，则不仅要提取第二个整数，提取后还需要计算相加的结果，并将相加的和作为解码结果输出。只有两个阶段全部完成才表示一次解码工作完成。

下面先基于ReplayingDecoder基础解码器编写一个整数相加的解码器：解码两个整数，并把这两个数据之和作为解码的结果。代码如下：

```
package com.crazymakercircle.netty.decoder;  
//省略import  
public class IntegerAddDecoder  
    extends ReplayingDecoder<IntegerAddDecoder.PHASE>  
{  
    //自定义的状态枚举值，代表两个阶段
```

```

enum PHASE
{
    PHASE_1, //第一个阶段，仅仅提取第一个整数，完成后进入
    第二个阶段
    PHASE_2 //第二个阶段，提取第二个整数后，还需要计算相
    加的结果并输出
}

private int first;
private int second;
public IntegerAddDecoder()
{
    //在构造函数中，初始化父类的state属性为
    PHASE_1，表示第一个阶段
    super(PHASE.PHASE_1);
}

@Override
protected void decode(ChannelHandlerContext
ctx,
ByteBuf in, List<Object> out) throws Exception{
    switch (state()) //判断当前的状态
    {
        //第一个阶段，仅仅提取第一个整数，完成后进
        入第二个阶段
        case PHASE_1:
            //从装饰器ByteBuf 中读取数
            //据
            first = in.readInt();
    }
}

```

//第一步解析成功，进入第二步，设置“state”为第二个阶段

```
checkpoint(PHASE.PHASE_2);
```

```
break;
```

//提取第二个整数后还需要计算相加的结果

//并将和作为解码的结果输出

```
case PHASE_2:
```

```
second = in.readInt();
```

```
Integer sum = first +
```

```
second;
```

```
out.add(sum);
```

//进入下一轮解码的第一步，

设置“state”为第一个阶段

```
checkpoint(PHASE.PHASE_1);
```

```
break;
```

```
default:
```

```
break;
```

```
}
```

```
}
```

```
}
```

IntegerAddDecoder类继承了
ReplayingDecoder<IntegerAddDecoder.PHASE>，其后面的泛型实参为

IntegerAddDecoder.PHASE自定义的状态类型，是一个枚举类型，用来作为泛型变量state的实际类型。该枚举值有两个常量：

- (1) PHASE_1：表示第一个阶段，读取第一个整数。
- (2) PHASE_2：表示第二个阶段，读取第二个整数，然后求和。

父类的成员变量state的值可能为PHASE_1或者PHASE_2，代表当前的阶段。state值需要在构造函数中进行初始化，在这里的子类构造函数中调用super(Status.PHASE_1)将state初始化为第一个阶段。

在IntegerAddDecoder类中，每一次decode()方法中的解码都有两个阶段：

- (1) 第一个阶段，解码出前一个整数。
- (2) 第二个阶段，解码出后一个整数，然后求和。

每一个阶段一完成就通过checkpoint(PHASE)方法（类似于state属性的setter()方法）把当前的state状态设置为新的PHASE枚举值。checkpoint()方法有两个作用：

- (1) 设置state属性的值，更新一下当前的状态。
- (2) 设置“读指针检查点”。

什么是ReplayingDecoder的“读指针”呢？就是ReplayingDecoder提取二进制数据的ByteBuf缓冲区的readerIndex。“读指针检查点”是ReplayingDecoder类的一个重要成员，用于暂存内部ReplayingDecoderBuffer装饰器缓冲区的readerIndex，有点类似

于mark。当读数据时，一旦缓冲区可读的二进制数据不够，缓冲区装饰器ReplayingDecoderBuffer在抛出ReplayError异常之前就会把readerIndex的值还原到之前通过checkpoint()方法设置的“读指针检查点”。在ReplayingDecoder下一次重新读取时，将会从“读指针检查点”开始读取。

回到IntegerAddDecoder的decode()方法，该方法的逻辑大致如下：

- (1) 判断当前解码器的state阶段是Status.PHASE_1还是Status.PHASE_2，根据对应的阶段进行读取处理。
- (2) 每一次读取完成之后要切换阶段并保持当前“读指针检查点”，以便于在可读数据不足之后进行读指针恢复。

通过上面的分析可以看出，IntegerAddDecoder与前面自定义的整数解码器不同，该解码器是有状态的，不能在不同的通道之间进行简单的共享。更进一步说，ReplayingDecoder类型及其所有的子类都需要保存状态信息，都不适合在不同的通道之间进行简单的共享。

至此，IntegerAddDecoder基本介绍完了。那么，如何使用IntegerAddDecoder解码器呢？具体的测试用例和前面Byte2IntegerDecoder的大致相同，由于篇幅的限制，这里不再赘述。大家可以在源代码包中执行对应的Byte2IntegerReplayDecoderTester测试用例。

6.1.5 字符串的分包解码器的实战案例

通过前面的整数分包传输，大家应该对ReplayingDecoder的分阶段解码有了完整的了解。现在来看一下字符串的分包传输。在原理上，字符串分包解码和整数分包解码是一样的，所不同的是：整数的长度是固定的，目前在Java中是4字节；字符串的长度是不固定的，是可变的。

如何获取字符串的长度信息呢？这是一个小小的难题，和程序所使用的具体传输协议是强相关的。一般来说，在Netty中进行字符串的传输可以采用普通的Head-Content内容传输协议。该协议的规则很简单：

- (1) 在协议的Head部分放置字符串的字节长度，可以用一个整数类型来描述。
- (2) 在协议的Content部分，放置字符串的字节数组。

在实际的传输过程中，一个Head-Content内容包在发送端会被编码成一个ByteBuf内容发送包，当到达接收端后可能被分成很多ByteBuf接收包。对于这些参差不齐的接收包，如何解码成最初的ByteBuf内容发送包来获得Head-Content内容呢？采用ReplayingDecoder解码器即可解决。

下面就是基于ReplayingDecoder实现自定义的字符串分包解码器的示例程序：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class StringReplayDecoder  
    extends ReplayingDecoder<StringReplayDecoder.PHASE> {
```

```
enum PHASE
{
    PHASE_1, //第一个阶段：解码出字符串的长度
    PHASE_2 //第二个阶段：按照第一个阶段的字符串长度解码出字符串的
内容
}

private int length;
private byte[] inBytes;
public StringReplayDecoder()
{
    //在构造函数中，需要初始化父类的state属性为PHASE_1阶段
    super(PHASE.PHASE_1);
}
@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf
in,
                      List<Object> out) throws Exception
{
    switch (state())
    {
        case PHASE_1:
            //第一步，从装饰器ByteBuf中读取字符串的长度
            length = in.readInt();
            inBytes = new byte[length];
            //进入第二步，读取内容
            //并设置“读指针检查点”为当前的readerIndex位置
    }
}
```

```
        checkpoint(PHASE.PHASE_2);

        break;

    case PHASE_2:
        //第二步，从装饰器ByteBuf 中读取字符串的内容数组
        in.readBytes(inBytes, 0, length);
        out.add(new String(inBytes, "UTF-8"));
        //第二步解析成功，进入下一个字符串的解析
        //并设置“读指针检查点”为当前的readerIndex位置
        checkpoint(PHASE.PHASE_1);
        break;

    default:
        break;
    }
}

}
```

在StringReplayDecoder类中，每一次字符串解码分为两个步骤：

- 第一步，解码出一个字符串的长度。
- 第二步，按照第一个阶段的字符串长度解码出字符串的内容。

在decode()方法中，每个阶段完成后都会通过checkpoint(Status)方法把当前的状态设置为新的Status值。

为了处理StringReplayDecoder解码后的字符串，这里编写一个简单的辅助性质的业务处理器。其功能是读取上一站的入站数据，把它转换成字符串，并输出到控制台上。新业务处理器的名称为StringProcessHandler，具体代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class StringProcessHandler  
    extends ChannelInboundHandlerAdapter {  
  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) ...{  
        String s = (String) msg;  
        Logger.info("打印出一个字符串: " + s);  
    }  
}
```

至此，已经编写了StringReplayDecoder和StringProcessHandler两个入站处理器：一个负责字符串解码，一个负责字符串输出。如何使用这两个入站处理器呢？编写一个测试用例，代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class StringReplayDecoderTester {  
    static String content= "疯狂创客圈：高性能学习社群！";  
    @Test  
    public void testStringReplayDecoder() {  
        ChannelInitializer i = new  
ChannelInitializer<EmbeddedChannel>() {  
            protected void initChannel(EmbeddedChannel ch) {  
                ch.pipeline().addLast(new  
StringReplayDecoder());  
            }  
        };  
        EmbeddedChannel channel = new EmbeddedChannel(i);  
        channel.writeInbound(content);  
        String result = channel.readInbound();  
        assertEquals(content, result);  
    }  
}
```

```

        ch.pipeline().addLast(new
StringProcessHandler());
    }

};

EmbeddedChannel channel = new EmbeddedChannel(i);
//待发送字符串content的字节数组
byte[] bytes = content.getBytes(Charset.forName("utf-
8"));
//循环发送100轮，每一轮可以理解为发送一个Head-Content报文
for (int j = 0; j < 100; j++) { //发送100个包
    //每个包为随机1~3个 "疯狂创客圈：高性能学习社群！"
    int random = RandomUtil.randInMod(3);
    ByteBuf buf = Unpooled.buffer();
    //发送长度：字节数组长度*重复次数
    buf.writeInt(bytes.length * random);
    //重复拷贝content的字节数据到发送缓冲区
    for (int k = 0; k < random; k++) {
        buf.writeBytes(bytes);
    }
    //发送内容：发送buf缓冲区
    channel.writeInbound(buf);
}
}
}

```

在测试用例中，新建一个EmbeddedChannel实例，将StringReplayDecoder和StringProcessHandler加入通道的流水线中。

为了测试入站处理器，调用writeInbound()方法向EmbeddedChannel（嵌入式通道）写入100次ByteBuf入站缓冲区，每个ByteBuf缓冲区包含一个字符串（为了进行区分，对content随机重複，最多3次）。

EmbeddedChannel接收到入站数据后，流水线上的两个入站处理器就能不断地处理这些入站数据：将接收到的二进制字节解码成一个一个的字符串，然后逐个输出到控制台上。

```
//部分输出省略
```

```
打印：疯狂创客圈：高性能学习社群！
```

```
打印：疯狂创客圈：高性能学习社群！
```

```
打印：疯狂创客圈：高性能学习社群！疯狂创客圈：高性能学习社群！
```

```
打印：疯狂创客圈：高性能学习社群！疯狂创客圈：高性能学习社群！
```

```
打印：疯狂创客圈：高性能学习社群！
```

```
打印：疯狂创客圈：高性能学习社群！疯狂创客圈：高性能学习社群！
```

通过ReplayingDecoder解码器，可以正确地解码分包后的ByteBuf数据包。但是，在实际开发中不建议继承这个类，原因如下：

(1) 不是所有的ByteBuf操作都被ReplayingDecoderBuffer装饰器类支持，可能有些ByteBuf方法在ReplayingDecoder的decode()方法中会抛出ReplayError异常。

(2) 在数据解码逻辑复杂的应用场景下，ReplayingDecoder在解码速度上相对较差。因为在ByteBuf长度不够时，ReplayingDecoder会捕获一个ReplayError异常，并会把ByteBuf中的读指针还原到之前的读指针检查点（checkpoint），然后结束这次解析操作，等待下一次

I/O读事件。在网络条件比较糟糕时，一个数据包的解析逻辑会被反复执行多次，此时解析过程是一个消耗CPU的操作，解码速度上相对较差。所以，ReplayingDecoder更多地应用于数据解析逻辑简单的场景。

在数据解析复杂的应用场景下，建议使用前文介绍的解码器 ByteToMessageDecoder或者其子类（后文介绍）。这里继承 ByteToMessageDecoder基类，实现一个定制的Head-Content协议字符串内容解码器，代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class StringIntegerHeaderDecoder extends  
ByteToMessageDecoder {  
    @Override  
    protected void decode(ChannelHandlerContext  
channelHandlerContext,  
        ByteBuf buf, List<Object> out) ...{  
        //可读字节小于4，消息头还没读满，返回  
        if (buf.readableBytes() < 4) {  
            return;  
        }  
        //消息头已经完整  
        //在真正开始从缓冲区读取数据之前，调用markReaderIndex()设置  
        mark  
        buf.markReaderIndex();  
        int length = buf.readInt();
```

```
//从缓冲区读出消息头的大小，这会导致readIndex读指针变化
//如果剩余长度不够消息体的大小，则需要重置读指针，下一次从相同的位置处理

    if (buf.readableBytes() < length) {

        //读指针重置到消息头的readIndex位置处
        buf.resetReaderIndex();

        return;
    }

    //读取数据，编码成字符串
    byte[] inBytes = new byte[length];
    buf.readBytes(inBytes, 0, length);
    out.add(new String(inBytes, "UTF-8"));
}

}
```

在上面的示例程序中，在读取数据之前，需要调用
buf.markReaderIndex()方法标记当前的位置指针，当可读内容不够
(buf.readableBytes() < length) 时，需要调用
buf.resetReaderIndex()方法将readerIndex读指针恢复到标记位置。

表面上ByteToMessageDecoder基类是无状态的，不像
ReplayingDecoder那样需要使用状态位来保存当前的读取阶段，实际上ByteToMessageDecoder也是有状态的。其内部有一个二进制字节累积器cumulation，用来保存没有解析完的二进制内容。所以，ByteToMessageDecoder及其子类都是有状态的，其实例不能在通道之间共享。在每次初始化通道的流水线时，都要重新创建一个ByteToMessageDecoder或者它的子类的实例。

6.1.6 MessageToMessageDecoder解码器

前面的解码器都是将ByteBuf缓冲区中的二进制数据解码成Java的普通POJO对象，那么是否存在一些解码器可以将一种POJO对象解码成另外一种POJO对象呢？答案是存在。与前面不同的是，解码器需要继承一个新的Netty解码器基类MessageToMessageDecoder<I>。在继承它的时候，需要明确的泛型实参<I>，用于指定入站消息的Java POJO类型。

为什么继承MessageToMessageDecoder<I>时需要指定入站数据的类型，而在前面继承ByteToMessageDecoder解码ByteBuf时不需要指定泛型实参呢？原因很简单：ByteToMessageDecoder的入站消息类型是十分明确的，就是二进制缓冲区ByteBuf类型；MessageToMessageDecoder<I>的入站消息类型是不明确的，可以是任何POJO类型，所以需要指定。

MessageToMessageDecoder同样使用了模板模式，也有一个decode()抽象方法，其具体解码的逻辑需要子类去实现。下面通过实现一个整数到字符串转换的解码器演示一下MessageToMessageDecoder的使用。代码很简单，如下所示：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class IntegerToStringDecoder extends  
    MessageToMessageDecoder<Integer> {  
    @Override  
    public void decode(ChannelHandlerContext ctx, Integer msg,
```

```
        List<Object> out) ...{  
    out.add(String.valueOf(msg));  
}  
}
```

这里定义的Integer2StringDecoder新类继承了MessageToMessageDecoder基类。基类泛型实参为Integer，表明子类解码器的入站数据类型为Integer。在decode()方法中，将整数转成字符串，再加入一个List输出容器（由父类在调用时传递过来的）中即可。在子类decode()方法处理完成后，父类会将List容器中的所有元素进行迭代，逐个发送给下一站Inbound入站处理器。

Integer2StringDecoder的使用与前面的解码器一样，其具体的测试用例和前面的StringReplayDecoder实例的也大致相同，由于篇幅的限制，这里不再赘述。大家可以在源代码包中查看，其测试用例的具体类名为Integer2StringDecoderTester。

6.2 常用的内置Decoder

Netty提供了不少开箱即用的Decoder（解码器），能够满足很多编解码应用场景的需求。下面将几个比较基础的解码器梳理一下。

（1）固定长度数据包解码器——FixedLengthFrameDecoder

适用场景：每个接收到的数据包的长度都是固定的，例如100字节。在这种场景下，把FixedLengthFrameDecoder解码器加到流水线中，它就会把入站ByteBuf数据包拆分成一个个长度为100的数据包，然后发往下一个channelHandler入站处理器。

说明

这里所指的数据包在Netty中是一个ByteBuf实例。

（2）行分割数据包解码器——LineBasedFrameDecoder

适用场景：每个ByteBuf数据包使用换行符（或者回车换行符）作为边界分隔符。在这种场景下，把LineBasedFrameDecoder解码器加到流水线中，Netty就会使用换行分隔符把ByteBuf数据包分割成一个一个完整的应用层ByteBuf数据包再发送到下一站。

（3）自定义分隔符数据包解码器—— DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder是LineBasedFrameDecoder按照行分割的通用版本，不同之处在于这个解码器更加灵活，可以自定义分隔符，而不是局限于换行符。如果使用这个解码器，那么所接收到的数据包末尾必须带上对应的分隔符。

（4）自定义长度数据包解码器—— LengthFieldBasedFrameDecoder

这是一种基于灵活长度的解码器，在ByteBuf数据包中加了一个长度字段，保存了原始数据包的长度，解码时会按照原始数据包长度进行提取。此解码器在所有开箱即用解码器中是最为复杂的一种，后面会重点介绍。

6.2.1 LineBasedFrameDecoder解码器

在前面字符串分包解码器中，内容是按照Head-Content协议进行传输的。如果不使用Head-Content协议，而是在发送端通过换行符（"\n"或者"\r\n"）来分割每一次发送的字符串，接收端是否可以正确地解析呢？答案是肯定的。

在Netty中，提供了一个开箱即用、使用换行符分割字符串的解码器——LineBasedFrameDecoder，它是一个最为基础的Netty内置解码器。这个解码器的工作原理很简单，依次遍历ByteBuf数据包中的可读字节，判断在二进制字节流中是否存在换行符"\n"或者"\r\n"的字节码。如果有，就以此位置为结束位置，把从可读索引到结束位置之间的字节作为解码成功后的ByteBuf数据包。

LineBasedFrameDecoder支持配置一个最大长度值，表示解码出来的ByteBuf能包含的最大字节数。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常。

下面演示一下LineBasedFrameDecoder的使用，代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
  
public class NettyOpenBoxDecoder {  
  
    static String spliter = "\r\n";  
  
    static String content = "疯狂创客圈：高性能学习社群！";  
  
    @Test  
  
    public void testLineBasedFrameDecoder() {  
  
        ChannelInitializer i =  
                new  
        ChannelInitializer<EmbeddedChannel>() {  
  
            protected void initChannel(EmbeddedChannel ch)  
            {  
  
                ch.pipeline().addLast(new  
                    LineBasedFrameDecoder(1024));  
  
                ch.pipeline().addLast(new StringDecoder());  
  
                ch.pipeline().addLast(new  
                    StringProcessHandler());  
  
            }  
  
        };  
  
        EmbeddedChannel channel = new EmbeddedChannel(i);  
        for (int j = 0; j < 100; j++) { //发送100个包
```

```
//每个包为随机1~3个 "疯狂创客圈：高性能学习社群!"  
int random = RandomUtil.randInMod(3);  
ByteBuf buf = Unpooled.buffer();  
for (int k = 0; k < random; k++) {  
    buf.writeBytes(content.getBytes("UTF-8"));  
}  
//发送"\r\n"回车换行符作为包结束符  
buf.writeBytes(splitter.getBytes("UTF-8"));  
channel.writeInbound(buf);  
}  
}  
}
```

在这个示例程序中，向通道写入100个入站数据包，每个入站包都以”\r\n”回车换行符结束。模拟通道的LineBasedFrameDecoder解码器会将”\r\n”作为分隔符，分隔出一个一个的入站ByteBuf，然后发送给StringDecoder，将这些ByteBuf二进制数据转成字符串，再发送到StringProcessHandler业务处理器，由它负责将字符串展示出来。

至此，LineBasedFrameDecoder演示完毕，仅仅是Netty中非常简单的数据包解码器。

6.2.2 DelimiterBasedFrameDecoder解码器

DelimiterBasedFrameDecoder解码器不仅可以使用换行符，还可以使用其他特殊字符作为数据包的分隔符，例如制表符”\t”。其构造方法如下：

```
public DelimiterBasedFrameDecoder(  
    int maxFrameLength,          //解码的数据包的最大长度  
    Boolean stripDelimiter,     //解码后的数据包是否去掉分隔符，一般  
选择是  
    ByteBuf delimiter)          //分隔符  
{  
    //省略构造器的源代码  
}
```

DelimiterBasedFrameDecoder解码器的使用方法与LineBasedFrameDecoder是一样的，只是在构造参数上有一点点不同。下面是一个实战案例。

```
package com.crazymakercircle.netty.decoder;  
//...  
public class NettyOpenBoxDecoder {  
    static String spliter2 = "\t";  
    static String content = "疯狂创客圈：高性能学习社群！";  
    /**  
     * LengthFieldBasedFrameDecoder使用实例  
     */  
    @Test  
    public void testDelimiterBasedFrameDecoder() {  
        try {  
            final ByteBuf delimiter =
```

```

        Unpooled.copiedBuffer(splitter2.getBytes("UTF-8"));

        ChannelInitializer i
            = new

    ChannelInitializer<EmbeddedChannel>() {
        protected void initChannel(EmbeddedChannel ch)
        {

            ch.pipeline().addLast(
                new DelimiterBasedFrameDecoder(1024, true,
delimiter));
                ch.pipeline().addLast(new StringDecoder());
                ch.pipeline().addLast(new
StringProcessHandler());
            }
        };

        //省略与前一个实例相同的重复代码
    }
}
}

```

以上实例中，通过DelimiterBasedFrameDecoder构造了一个以制表符作为分隔符的字符串分包器。向模拟通道发送字符串的代码，由于与前一小节的发送代码基本相同，这里省略。需要注意的是，发送一个包后，要发送一个制表符作为结束。

6.2.3 LengthFieldBasedFrameDecoder解码器

在Netty的开箱即用解码器中，最为复杂的是解码器为LengthFieldBasedFrameDecoder自定义长度数据包。它的难点在于参数比较多，也比较难以理解，但同时它又比较常用，因而下面对它进行重点介绍。

LengthFieldBasedFrameDecoder可以翻译为“长度字段数据包解码器”。传输内容中的Length（长度）字段的值是指存放在数据包中要传输内容的字节数。普通的基于Head-Content协议的内容传输尽量用内置的LengthFieldBasedFrameDecoder来解码。

一个简单的LengthFieldBasedFrameDecoder使用示例如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
  
public class NettyOpenBoxDecoder {  
  
    public static final int VERSION = 100;  
  
    static String content = "疯狂创客圈：高性能学习社群！";  
  
    /**  
     * LengthFieldBasedFrameDecoder使用示例 1  
     */  
  
    @Test  
  
    public void testLengthFieldBasedFrameDecoder1() {  
  
        try {  
  
            final LengthFieldBasedFrameDecoder spliter =  
                new LengthFieldBasedFrameDecoder(1024, 0,  
                    4, 0, 4);  
  
            ChannelInitializer i =  
                new
```

```
ChannelInitializer<EmbeddedChannel>() {
    protected void initChannel(EmbeddedChannel ch)
    {
        ch.pipeline().addLast(splitter);
        ch.pipeline().addLast(new
StringDecoder(Charset.forName("UTF-8")));
        ch.pipeline().addLast(new
StringProcessHandler());
    }
};

EmbeddedChannel channel = new EmbeddedChannel(i);

for (int j = 1; j <= 100; j++) {
    ByteBuf buf = Unpooled.buffer();
    String s = j + "次发送->" + content;
    byte[] bytes = s.getBytes("UTF-8");
    buf.writeInt(bytes.length);
    buf.writeBytes(bytes);
    channel.writeInbound(buf);
}

Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
}
```

```
    }  
}
```

上面的示例程序中用到了一个LengthFieldBasedFrameDecoder构造器，具体如下：

```
public LengthFieldBasedFrameDecoder(  
    int maxFrameLength,          //发送的数据包的最大长度  
    int lengthFieldOffset,       //长度字段偏移量  
    int lengthFieldLength,       //长度字段本身占用的字节数  
    int lengthAdjustment,        //长度字段的偏移量矫正  
    int initialBytesToStrip)    //丢弃的起始字节数  
{  
    //...  
}
```

在前面的示例程序中涉及5个参数和值，分别解读如下：

(1) `maxFrameLength`: 发送的数据包的最大长度。示例程序中该值为1024，表示一个数据包最多可发送1024字节。

(2) `lengthFieldOffset`: 长度字段偏移量，指的是长度字段位于整个数据包内部字节数组中的下标索引值。

(3) `lengthFieldLength`: 长度字段所占的字节数。如果长度字段是一个int整数，则为4；如果长度字段是一个short整数，则为2。

(4) `lengthAdjustment`: 长度的调整值。这个参数最为难懂。在传输协议比较复杂的情况下，例如协议包含了长度字段、协议版本

号、魔数等，那么解码时就需要进行长度调整。长度调整值的计算公式为：内容字段偏移量-长度字段偏移量-长度字段的字节数。这个公式一看就比较复杂，下一小节会有详细的举例说明。

(5) initialBytesToStrip: 丢弃的起始字节数。在有效数据字段Content前面，如果还有一些其他字段的字节，作为最终的解析结果可以丢弃。例如，在上面的示例程序中，前面有4字节的长度字段，它起辅助的作用，最终的结果中不需要这个长度，所以丢弃的字节数为4。

在前面的示例程序中，自定义长度解码器的构造参数值如下：

```
LengthFieldBasedFrameDecoder spliter = new  
LengthFieldBasedFrameDecoder(1024, 0, 4, 0, 4);
```

第一个参数maxFrameLength设置为1024，表示数据包的最大长度为1024字节。

第2个参数lengthFieldOffset设置为0，表示长度字段的偏移量为0，也就是长度字段放在了最前面，处于数据包的起始位置。

第3个参数lengthFieldLength设置为4，表示长度字段的长度为4字节，即内容长度的值占用数据包的4字节。

第4个参数lengthAdjustment设置为0。长度调整值的计算公式为：内容字段偏移量-长度字段偏移量-长度字段的字节数，在上面示例程序中的实际值为4-0-4=0。

第5个参数initialBytesToStrip为4，表示获取最终内容Content的字节数组时抛弃最前面的4字节的数据。

运行上面的示例程序，输出结果节选如下：

```
//...  
打印：1次发送->疯狂创客圈：高性能学习社群！  
打印：2次发送->疯狂创客圈：高性能学习社群！  
打印：3次发送->疯狂创客圈：高性能学习社群！  
打印：4次发送->疯狂创客圈：高性能学习社群！  
打印：5次发送->疯狂创客圈：高性能学习社群！  
打印：6次发送->疯狂创客圈：高性能学习社群！
```

如果对这些传输没有直观的了解，下面对第一个传输的数据包给出一个简单的字节图（见图6-3）：长度字段为4字节，内容字段为52字节，整个数据包为56字节。

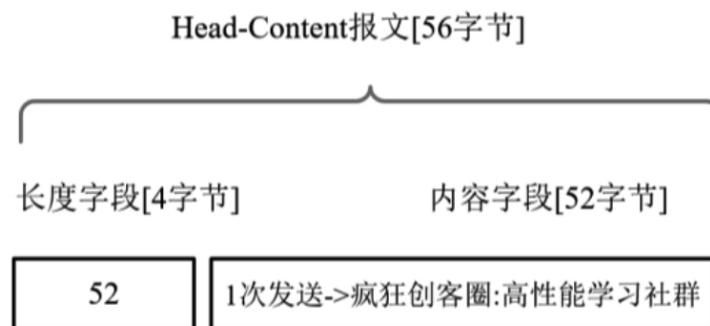


图6-3 Head-Content协议的示意图

6.2.4 多字段Head-Content协议数据包解析的实战案例

Head-Content协议是最为简单的内容传输协议。在实际使用过程中则没有那么简单，除了长度和内容，在数据包中还可能包含其他字

段，例如协议版本号，如图6-4所示。

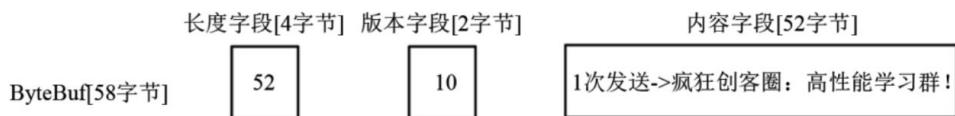


图6-4 包含协议版本号的Head-Content协议的示意图

使用LengthFieldBasedFrameDecoder解码器解析以上带有版本号的Head-Content协议的数据包，该如何进行构造器参数的计算呢？

第1个参数maxFrameLength可以为1024，表示数据包的最大长度为1024字节。

第2个参数lengthFieldOffset为0，表示长度字段处于数据包的起始位置。

第3个参数lengthFieldLength的值为4，表示长度字段的长度为4字节。

第4个参数lengthAdjustment为2，长度调整值的计算方法为：内容字段偏移量-长度字段偏移量-长度字段的长度=6-0-4=2。换句话说，在这个例子中，lengthAdjustment就是夹在内容字段和长度字段中的部分——版本号的长度。

第5个参数initialBytesToStrip为6，表示获取最终内容的字节数组时抛弃最前面的6字节数据。换句话说，长度字段、版本字段的值被抛弃。

实战案例的代码如下：

```
package com.crazymakercircle.netty.decoder;

//...

public class NettyOpenBoxDecoder {

    public static final int VERSION = 100;

    static String content = "疯狂创客圈：高性能学习社群！";

    /**
     * LengthFieldBasedFrameDecoder使用示例 2
     */

    @Test
    public void testLengthFieldBasedFrameDecoder2() {
        try {
            final LengthFieldBasedFrameDecoder spliter =
                new LengthFieldBasedFrameDecoder(1024, 0,
                    4, 2, 6);

            ChannelInitializer<EmbeddedChannel> i =
                new
            ChannelInitializer<EmbeddedChannel>() {
                protected void initChannel(EmbeddedChannel ch)
                {
                    ch.pipeline().addLast(spliter);
                    ch.pipeline().addLast(new
                        StringDecoder(Charset.forName("UTF-8")));
                    ch.pipeline().addLast(new
                        StringProcessHandler());
                }
            };
        }
    }
}
```

```
EmbeddedChannel channel = new EmbeddedChannel(i);

    for (int j = 1; j <= 100; j++) {
        ByteBuf buf = Unpooled.buffer();
        String s = j + "次发送->" + content;
        byte[] bytes = s.getBytes("UTF-8");
        buf.writeInt(bytes.length);
        buf.writeChar(VERSION);
        buf.writeBytes(bytes);
        channel.writeInbound(buf);
    }

    Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
}
```

运行实战案例，大家可以发现运行的结果和前一个实例一样，表明参数设置是正确的，LengthFieldBasedFrameDecoder解码器可以正确地解析内容。

将协议设计得再复杂一点：将2字节的协议版本放在最前面，在长度字段前面加上2字节的版本字段，在长度字段后面加上4字节的魔

数，魔数用来对数据包做一些安全的认证。协议的数据包如图6-5所示。

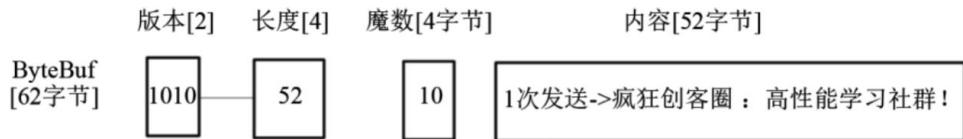


图6-5 包含协议版本号、魔数的Head-Content协议的示意图

使用LengthFieldBasedFrameDecoder解码器解码图6-5中的Head-Content协议，构造器的参数该如何计算呢？参数的设置大致如下：

第1个参数maxFrameLength可以设置为1024，表示数据包的最大长度为1024字节。

第2个参数lengthFieldOffset可以设置为2，表示长度字段处于版本号的后面。

第3个参数lengthFieldLength可以设置为4，表示长度字段为4字节。

第4个参数lengthAdjustment可以设置为4。长度调整值的计算方法为：内容字段偏移量-长度字段偏移量-长度字段的长度=10-2-4=4。在这个例子中，lengthAdjustment就是夹在内容字段和长度字段中的部分——魔数字段的长度。

第5个参数initialBytesToStrip可以设置为10，表示获取最终Content内容的字节数组时抛弃最前面的10字节数据。换句话说，长度字段、版本字段、魔数字段的值被抛弃。

实战案例的代码如下：

```
package com.crazymakercircle.netty.decoder;

//...

@Test

public void testLengthFieldBasedFrameDecoder3() {

    try {

        final LengthFieldBasedFrameDecoder spliter =
            new LengthFieldBasedFrameDecoder(1024, 2, 4, 4,
                10);

        ChannelInitializer i = new
ChannelInitializer<EmbeddedChannel>() {

            protected void initChannel(EmbeddedChannel ch) {
                ch.pipeline().addLast(spliter);
                ch.pipeline().addLast(
                    new
StringDecoder(Charset.forName("UTF-8")));
                ch.pipeline().addLast(new
StringProcessHandler());
            }
        };

        EmbeddedChannel channel = new EmbeddedChannel(i);
        for (int j = 1; j <= 100; j++) {
            ByteBuf buf = Unpooled.buffer();
            String s = j + "次发送->" + content;
            byte[] bytes = s.getBytes("UTF-8");
            buf.writeChar(VERSION);
            buf.writeInt(bytes.length);
            buf.writeInt(MAGICCODE);
        }
    }
}
```

```
        buf.writeBytes (bytes) ;  
        channel.writeInbound (buf) ;  
    }  
  
    Thread.sleep (Integer.MAX_VALUE) ;  
} catch (InterruptedException e) {  
    e.printStackTrace () ;  
} catch (UnsupportedEncodingException e) {  
    e.printStackTrace () ;  
}  
}  
}  
}
```

运行实战案例，可以发现运行的结果和前一个实例一样。这说明参数设置是正确的，LengthFieldBasedFrameDecoder解码器可以正确地解析内容。

6.3 Encoder原理与实战

在Netty的业务处理完成后，业务处理的结果往往是某个Java POJO对象需要编码成最终的ByteBuf二进制类型，通过流水线写入底层的Java通道，这就需要用到Encoder（编码器）。

在Netty中，什么叫编码器？首先，编码器是一个Outbound出站处理器，负责处理“出站”数据；其次，编码器将上一站Outbound出站处理器传过来的输入（Input）数据进行编码或者格式转换，然后传递到下一站ChannelOutboundHandler出站处理器。

编码器与解码器相呼应，Netty中的编码器负责将“出站”的某种Java POJO对象编码成二进制ByteBuf，或者转换成另一种Java POJO对象。

编码器是ChannelOutboundHandler的具体实现类。一个编码器将出站对象编码之后，数据将被传递到下一个ChannelOutboundHandler出站处理器进行后面的出站处理。

由于最后只有ByteBuf才能写入通道中，因此可以肯定通道流水线上装配的第一个编码器一定是把数据编码成了ByteBuf类型。为什么编码成的最终ByteBuf类型数据包的编码器是在流水线的头部，而不是在流水线的尾部呢？原因很简单：出站处理的顺序是从后向前的。

6.3.1 MessageToByteEncoder编码器

MessageToByteEncoder是一个非常重要的编码器基类，位于Netty的io.netty.handler.codec包中。MessageToByteEncoder的功能是将一个Java POJO对象编码成一个ByteBuf数据包。它是一个抽象类，仅仅实现了编码的基础流程，在编码过程中通过调用encode()抽象方法来完成。它的encode()编码方法是一个抽象方法，没有具体的编码逻辑实现，实现encode()抽象方法的工作需要子类去完成。

如果要实现一个自己的编码器，则需要继承自MessageToByteEncoder基类，实现它的encode()抽象方法。作为演示，下面实现一个整数编码器。其功能是将Java整数编码成二进制ByteBuf数据包。这个示例程序的代码如下：

```
package com.crazymakercircle.netty.encoder;  
//...  
public class Integer2ByteEncoder  
        extends  
        MessageToByteEncoder<Integer> {  
    @Override  
    public void encode(ChannelHandlerContext ctx,  
                      Integer msg, ByteBuf out) ...  
    {  
        out.writeInt(msg);  
        Logger.info("encoder Integer = " + msg);  
    }  
}
```

在继承MessageToByteEncoder时，需要带上泛型实参，具体为编码之前的Java POJO原类型（输入类型）。在这个示例程序中，编码之

前的类型是Java Integer。

上面的encode()方法实现很简单：将入站数据Integer类型对象msg写入Out实参（基类传入的ByteBuf实例）。编码完成后，基类MessageToByteEncoder会将输出的ByteBuf数据包发送到下一站。

编码器Integer2ByteEncoder已经完成，如何使用呢？这里编写了一个测试用例，代码如下：

```
package com.crazymakercircle.netty.encoder;  
//...  
  
public class Integer2ByteEncoderTester {  
  
    @Test  
    public void testIntegerToByteDecoder() {  
        ChannelInitializer i = new  
        ChannelInitializer<EmbeddedChannel>() {  
            protected void  
            initChannel(EmbeddedChannel ch) {  
  
                ch.pipeline().addLast(new Integer2ByteEncoder());
            }
        };
        EmbeddedChannel channel = new
        EmbeddedChannel(i);
        for (int j = 0; j < 100; j++) {
            channel.write(j); //向通道写入整数
        }
        channel.flush();
    }
}
```

```
//取得通道的出站数据包  
ByteBuf buf = (ByteBuf) channel.readOutbound();  
while (null != buf) {  
    System.out.println("o = " +  
buf.readInt());  
    buf = (ByteBuf)  
channel.readOutbound();  
}  
//...  
}  
}
```

在上面的实例中，首先将Integer2ByteEncoder加入嵌入式通道，然后调用write()方法向通道写入100个数字。写完之后，调用channel.readOutbound()方法从通道中读取模拟的出站数据包，并且不断地循环，将数据包中的数字打印出来。

此编码器的运行比较简单，运行的结果就不在书中给出了。建议读者参考源代码工程，自行设计和实现一个整数编码器，以便加深理解。

6.3.2 MessageToMessageEncoder编码器

上一小节的示例程序是将POJO对象编码成ByteBuf二进制对象，那么是否能够通过Netty的编码器将某种POJO对象编码成另外一种POJO对象呢？答案是肯定的。需要继承另外一个Netty的重要编码器——MessageToMessageEncoder编码器，并实现它的encode()抽象方法。在

子类的encode()方法实现中，完成原POJO类型到目标POJO类型的转换逻辑。在encode()实现方法中，编码完成后，将解码后的目标对象加入encode()方法中的实参list输出容器即可。

下面是一个从字符串（String）到整数（Integer）的编码器，演示一下MessageToMessageEncoder的使用。此编码器的具体功能是将字符串中的所有数字提取出来，然后输出到下一站。代码很简单，具体如下：

```
package com.crazymakercircle.netty.encoder;  
//...  
  
public class String2IntegerEncoder  
    extends MessageToMessageEncoder<String>  
{  
    @Override  
    protected void encode(  
        ChannelHandlerContext c, String s, List<Object> list)...{  
        char[] array = s.toCharArray();  
        for (char a : array) {  
            //48 是0的编码，57 是9 的编码  
            if (a >= 48 && a <= 57) {  
                list.add(new Integer(a));  
            }  
        }  
    }  
}
```

这里定义的String2IntegerEncoder类继承了MessageToMessageEncoder基类，并且明确了入站的数据类型为String。在encode()方法中，将字符串中的数字（编码在48和57之间）提取出来之后，放入list输出容器中，如果遇到数字之外的其他字符则直接略过。

在子类的encode()方法处理完成之后，基类会对这个list输出容器中的所有元素进行迭代，将列表的元素逐个发送给下一站。

编码器String2IntegerEncoder已经完成，下面编写一个测试用例，代码如下：

```
package com.crazymakercircle.netty.encoder;
//...
public class String2IntegerEncoderTester {
    /**
     * 测试字符串到整数的编码器
     */
    @Test
    public void testStringToIntengerDecoder() {
        ChannelInitializer<EmbeddedChannel> i = new
        ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannel ch) {
                ch.pipeline().addLast(new
                    Integer2ByteEncoder());
                ch.pipeline().addLast(new
                    String2IntegerEncoder());
            }
        }
    }
}
```

```
};

EmbeddedChannel channel = new EmbeddedChannel(i);

for (int j = 0; j < 100; j++) {

    String s = "i am " + j;

    channel.write(s); //向通道写入含有数字的字符串

}

channel.flush();

ByteBuf buf = (ByteBuf) channel.readOutbound();

while (null != buf) {

    System.out.println("o = " + buf.readInt()); //打印数

    buf = (ByteBuf) channel.readOutbound(); //读取数

}

}

}
```

测试用例中除了需要使用String2IntegerEncoder编码器外，还需要用到Integer2ByteEncoder编码器。String2IntegerEncoder仅仅是编码的第一棒，负责将字符串编码成整数；Integer2ByteEncoder是编码的第二棒，将整数进一步变成ByteBuf数据包后才能最终写入通道。由于出站处理的过程是从后向前的次序，因此Integer2ByteEncoder先加入流水线，String2IntegerEncoder后加入流水线。

此编码器的运行比较简单，运行结果就不在书中给出了。建议读者参考源代码工程，查看运行结果，以便加深理解。

6.4 解码器和编码器的结合

在实际的开发中，由于数据的入站和出站关系紧密，因此编码器和解码器的关系很紧密。编码和解码更是一种紧密的、相互配套的关系。在流水线处理时，数据的流动往往一进一出，进来时解码，出去时编码。所以，在同一个流水线上，加了某种编码逻辑，常常需要加上一个相对应的解码逻辑。

前面讲到编码器和解码器是分开实现的。例如，通过继承 ByteToMessageDecoder 基类或者其子类，完成 ByteBuf 数据包到 POJO 的解码工作；通过继承基类 MessageToByteEncoder 或者其子类，完成 POJO 到 ByteBuf 数据包的编码工作。总之，具有相反逻辑的编码器和解码器分开实现在两个不同的类中，导致的一个结果是相互配套的编码器和解码器在加入通道的流水线时常常需要分两次添加。

现在的问题是：具有相互配套逻辑的编码器和解码器能否放在同一个类中呢？答案是肯定的，这需要用到 Netty 的新类型—— Codec（编解码器）。

6.4.1 ByteToMessageCodec 编解码器

完成 POJO 到 ByteBuf 数据包的编解码器基类为 ByteToMessageCodec<I>，它是一个抽象类。从功能上说，继承 ByteToMessageCodec<I> 就等同于继承了 ByteToMessageDecoder 和 MessageToByteEncoder 这两个基类。

编解码器ByteToMessageCodec同时包含了编码encode()和解码decode()两个抽象方法，这两个方法都需要我们自己实现：

(1) 编码方法——encode(ChannelHandlerContext, I, ByteBuf)。

(2) 解码方法——decode(ChannelHandlerContext, ByteBuf, List<Object>)。

下面是一个整数到字节、字节到整数的编解码器，代码如下：

```
package com.crazymakercircle.netty.codec;  
//...  
  
public class Byte2IntegerCodec extends  
ByteToMessageCodec<Integer> {  
  
    @Override  
  
    public void encode(ChannelHandlerContext ctx,  
                       Integer msg, ByteBuf out) ...{  
  
        out.writeInt(msg);  
  
        System.out.println("write Integer = " + msg);  
    }  
  
    @Override  
  
    public void decode(ChannelHandlerContext ctx,  
                      ByteBuf in, List<Object> out) ...{  
  
        if (in.readableBytes() >= 4) {  
  
            int i = in.readInt();  
  
            System.out.println("Decoder i= " + i);  
  
            out.add(i);  
        }  
    }  
}
```

```
    }  
}  
}
```

这是编码器和解码器的结合，简单地通过继承的方式将前面编码器的encode()方法和解码器的decode()方法放在了同一个自定义类中，这样在逻辑上更加紧密。在使用时，加入流水线时也只需要加入一次。

从上面的示例程序可以看出，ByteToMessageCodec编解码器和前面的编码器与解码器分开来实现相比仅仅是少写了一个类，少加入了一次流水线，在技术、功能上和分开实现、添加到流水线没有任何区别。

对于POJO之间进行转换的编码和解码，Netty将MessageToMessageEncoder编码器和MessageToMessageDecoder解码器进行了简单的整合，整合出一个新的编解码器基类——MessageToMessageCodec。这个基类同时包含了encode()和decode()两个抽象方法，用于完成POJO-T0-POJO的双向转换。仅仅是使用形式变得简化了，在技术上并没有增加太多的难度，所以本书不再展开介绍。

6.4.2 CombinedChannelDuplexHandler组合器

前面的编码器和解码器相结合是通过继承完成的。继承的不足之处在于：将编码器和解码器的逻辑强制性地放在同一个类中，在只需要编码或者解码单边操作的流水线上，逻辑上不大合适。

编码器和解码器如果要结合起来，除了继承的方法之外，还可以通过组合的方式实现。与继承相比，组合会带来更大的灵活性：编码器和解码器可以捆绑使用，也可以单独使用。

如何把单独实现的编码器和解码器组合起来呢？

Netty提供了一个新的组合器——CombinedChannelDuplexHandler基类。其用法也很简单，下面通过示例程序来演示如何将前面的整数解码器IntegerFromByteDecoder和对应的整数编码器IntegerToByteEncoder组合起来。代码如下：

```
package com.crazymakercircle.netty.codec;  
//...  
public class IntegerDuplexHandler extends  
CombinedChannelDuplexHandler<  
    Byte2IntegerDecoder, Integer2ByteEncoder>  
{  
    public IntegerDuplexHandler() {  
        super(new Byte2IntegerDecoder(), new  
Integer2ByteEncoder());  
    }  
}
```

只需要继承CombinedChannelDuplexHandler，而不需要像ByteToMessageCodec那样把编码逻辑和解码逻辑都挤在同一个类中，还是复用原来分开的编码器和解码器实现代码。

总之，使用CombinedChannelDuplexHandler可以保证有了相反逻辑关系的encoder编码器和decoder解码器既可以结合使用，又可以分开使用，十分方便。

第7章 序列化与反序列化：JSON和Protobuf

我们在开发一些远程过程调用（RPC）的程序时通常会涉及对象的序列化/反序列化问题，例如一个Person对象从客户端通过TCP方式发送到服务端。由于TCP（或者UDP等类似低层协议）只能发送字节流，因此需要应用层将Java POJO对象“序列化”成字节流，发送过去之后，数据接收端再将字节流“反序列化”成Java POJO对象即可。

“序列化”和“反序列化”一定会涉及POJO的编码和格式化（Encoding & Format），目前我们可选择的编码方式有：

- 使用**JSON**。将Java POJO对象转换成JSON结构化字符串。基于HTTP，在Web应用、移动开发方面等，这种是常用的编码方式，因为JSON的可读性较强。这种方式的缺点是它的性能稍差。
- 基于**XML**。和JSON一样，数据在序列化成字节流之前需要转换成字符串。这种方式的可读性强，性能差，异构系统、Open API类型的应用中常用。
- 使用**Java**内置的编码和序列化机制，可移植性强，性能稍差，无法跨平台（语言）。
- 开源的二进制的序列化/反序列化框架，例如**Apache Avro**、**Apache Thrift**、**Protobuf**等。前面的两个框架和Protobuf相比，性能非常接近，而且设计原理如出一辙。其中，Avro在大数据存储（RPC数据交换、本地存储）时比较常用；Thrift的亮

点在于内置了RPC机制，所以在开发一些RPC交互式应用时，客户端和服务端的开发与部署都非常简单。

如何选择序列化/反序列化框架呢？

评价一个序列化框架的优缺点大概从两方面着手：

(1) 结果数据大小：原则上说，序列化后的数据尺寸越小，传输效率越高。

(2) 结构复杂度：会影响序列化/反序列化的效率，结构越复杂越耗时。

理论上来说，对于对性能要求不是太高的服务器程序，可以选择JSON文本格式的序列化框架；对于性能要求比较高的服务器程序，应该选择传输效率更高的二进制序列化框架，建议是Protobuf。

Protobuf是一个高性能、易扩展的序列化框架，性能比较高，其性能的有关数据可以参看官方文档。Protobuf本身非常简单，易于开发，而且结合Netty框架，可以非常便捷地实现一个通信应用程序。反过来，Netty也提供了相应的编解码器，为Protobuf解决了有关Socket通信中“半包、粘包”等问题。

无论是使用JSON、Protobuf还是其他的传输协议，我们必须保证在数据包的反序列化之前，接收端的ByteBuf二进制数据包一定是一个完整的应用层二进制包，不能是一个半包或者粘包，这就涉及通信过程中的拆包技术。

7.1 详解粘包和拆包

什么是粘包和半包？先从数据包的发送和接收开始讲起。大家知道，Netty发送和读取数据的“场所”是ByteBuf缓冲区。对于发送端，每一次发送就是向通道写入一个ByteBuf，发送数据时先填好ByteBuf，然后通过通道发送出去。对于接收端，每一次读取就是通过业务处理器的入站方法从通道读到一个ByteBuf。读取数据的方法如下：

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
{
    ByteBuf byteBuf = (ByteBuf) msg;
    //省略入站处理
}
```

最为理想的情况是：发送端每发送一个ByteBuf缓冲区，接收端就能接收到一个ByteBuf，并且发送端和接收端的ByteBuf内容一模一样。然而，在实际的通信过程中并没有大家预料的那么完美。下面给大家看一个实例，看看实际通信过程中所遇到的诡异情况。

7.1.1 半包问题的实战案例

改造一下前面的NettyEchoClient实例，通过循环的方式向NettyEchoServer回显服务器写入大量的ByteBuf，然后看看实际的服务器响应结果。注意：服务器类不需要改造，直接使用之前的回显服务器即可。

改造好的客户端类——叫NettyDumpSendClient。在客户端建立连接成功之后，使用一个for循环不断通过通道向服务端发送ByteBuf，一直写到1000次，这些ByteBuf的内容相同，都是字符串的内容：“疯狂创客圈：高性能学习者社群！”。代码如下：

```
package com.crazymakercircle.netty.echoServer;  
//...  
  
public class NettyDumpSendClient {  
  
    private int serverPort;  
  
    private String serverIp;  
  
    Bootstrap b = new Bootstrap();  
  
    public NettyDumpSendClient(String ip, int port) {  
  
        this.serverPort = port;  
  
        this.serverIp = ip;  
  
    }  
  
  
    public void runClient() {  
  
        //创建反应器线程组  
  
        //省略启动客户端Bootstrap引导类配置和启动  
  
        //阻塞，直到连接完成  
  
        f.sync();  
  
        Channel channel = f.channel();  
  
  
        //发送大量的文字  
  
        String content= "疯狂创客圈：高性能学习者社群！";  
  
        byte[] bytes  
=content.getBytes(Charset.forName("utf-8"));
```

```
        for (int i = 0; i < 1000; i++) {
            //发送ByteBuf
            ByteBuf buffer = channel.alloc().buffer();
            buffer.writeBytes(bytes);
            channel.writeAndFlush(buffer);
        }
    //省略优雅关闭客户端
}
public static void main(String[] args) throws
InterruptedException {
    int port = NettyDemoConfig.SOCKET_SERVER_PORT;
    String ip = NettyDemoConfig.SOCKET_SERVER_IP;
    new NettyDumpSendClient(ip, port).runClient();
}
}
```

运行程序查看结果之前，首先要启动前面介绍过的NettyEchoServer回显服务器，然后启动新编写的NettyDumpSendClient客户端程序，连接成功后客户端会向服务器发送1000个ByteBuf内容缓冲区，服务器NettyEchoServer收到后会输出到控制台，然后回写给客户端。服务器的输出如图7-1所示。

```
:channelRead]: 写回前, msg.refCnt:1  
:lambda$channelRead$0]: 写回后, msg.refCnt:0  
:channelRead]: msg type: 直接内存  
:channelRead]: server received: 者社群!疯狂创客圈: 高性能学习者社群!疯狂创客圈  
:channelRead]: 写回前, msg.refCnt:1  
:lambda$channelRead$0]: 写回后, msg.refCnt:0  
:channelRead]: msg type: 直接内存  
:channelRead]: server received: ◆◆学习者社群!疯狂创客圈: 高性能学习者社群!疯  
:channelRead]: 写回前, msg.refCnt:1  
:lambda$channelRead$0]: 写回后, msg.refCnt:0  
:channelRead]: msg type: 直接内存  
:channelRead]: server received: ◆高性能学习者社群!疯狂创客圈: 高性能学习者社
```

图7-1 NettyEchoServer的控制台输出

仔细观察服务端的控制台输出，可以看出存在三种类型的输出：

- (1) 读到一个完整的客户端输入ByteBuf。
- (2) 读到多个客户端的ByteBuf输入，但是“粘”在了一起。
- (3) 读到部分ByteBuf的内容，并且有乱码。

除了观察服务端的输出之外，再仔细观察客户端的输出，可以看到客户端也存在以上三种类型的输出。

对应于第1种情况接收到的完整的ByteBuf，这里称为“全包”。
对应于第2种情况，多个发送端的输入ByteBuf“粘”在了一起，这里称为“粘包”。
对应于第3种情况，一个发送过来的ByteBuf被“拆开”接收，接收端读取到一个破碎的包，这里称为“半包”。

为了简单起见，也可以将“粘包”的情况看成特殊的“半包”。
“粘包”和“半包”可以统称为传输的“半包问题”。

7.1.2 什么是半包问题

半包问题包含了“粘包”和“半包”两种情况：

(1) 粘包：接收端（Receiver）收到一个ByteBuf，包含了发送端（Sender）的多个ByteBuf，发送端的多个ByteBuf在接收端“粘”在了一起。

(2) 半包：Receiver将Sender的一个ByteBuf“拆”开了收，收到了多个破碎的包。换句话说，Receiver收到了Sender的一个ByteBuf的一小部分。

无论是粘包还是半包都不是一次正常的ByteBuf缓存区接收，具体如图7-2所示。

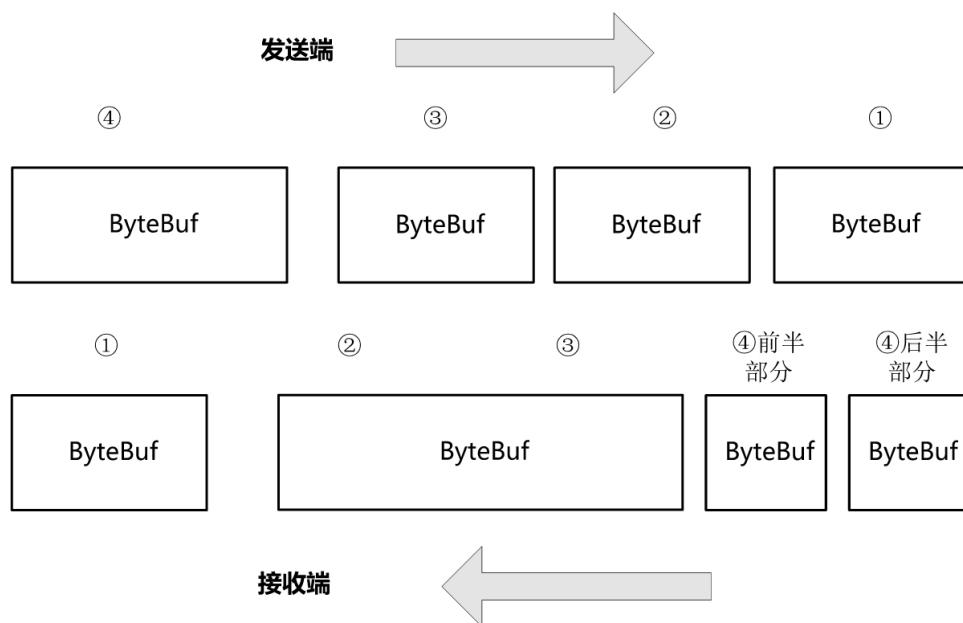


图7-2 粘包和半包现象 (②和③为粘包, ④为半包)

7.1.3 半包问题的根因分析

粘包和半包的来源得从操作系统底层说起。

大家都知道，底层网络是以二进制字节报文的形式来传输数据的。读数据的过程大致为：当I/O可读时，Netty会从底层网络将二进制数据读到ByteBuf缓冲区中，再交给Netty程序转成Java POJO对象。写数据的过程大致为：编码器将一个Java类型的数据转换成底层能够传输的二进制ByteBuf缓冲数据。

在发送端Netty的应用层进程缓冲区中，程序以ByteBuf为单位来发送数据，但是到了底层操作系统内核缓冲区，底层会按照协议的规范对数据包进行二次封装，封装成传输层的协议报文，再进行发送。在接收端收到传输层的二进制包后，首先复制到内核缓冲区，Netty读取ByteBuf时才复制到应用的用户缓冲区。

在接收端，当Netty程序将数据从内核缓冲区复制到用户缓冲区的ByteBuf时，问题来了：

(1) 每次读取底层缓冲的数据容量是有限制的，当TCP内核缓冲区的数据包比较大时，可能会将一个底层包分成多次ByteBuf进行复制，进而造成用户缓冲区读到的是半包。

(2) 当TCP内核缓冲区的数据包比较小时，一次复制的是不止一个内核缓冲区包，进而会造成用户缓冲区读到粘包。

如何解决呢？基本思路是，在接收端，Netty程序需要根据自定义协议将读取到的进程缓冲区ByteBuf在应用层进行二次组装，重新组装

应用层的数据包。接收端的这个过程通常也称为分包或者拆包。

在Netty中分包的方法主要有以下两种：

(1) 可以自定义解码器分包器：基于ByteToMessageDecoder或者ReplayingDecoder，定义自己的用户缓冲区分包器。

(2) 使用Netty内置的解码器。例如，可以使用Netty内置的LengthFieldBasedFrameDecoder自定义长度数据包解码器对用户缓冲区ByteBuf进行正确的分包。

在本章后面会用到这两种方法。

7.2 使用JSON协议通信

JSON（JavaScript Object Notation，JS对象简谱）是一种轻量级的数据交换格式。它是基于ECMAScript（欧洲计算机协会制定的JS规范）的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得JSON成为理想的数据交换语言。

JSON协议是一种文本协议，易于人阅读和编写，同时也易于机器解析和生成，并能有效地提升网络传输效率。

7.2.1 JSON的核心优势

XML是一种常用的文本协议，和JSON一样都使用结构化方法来标记数据。和XML相比，JSON作为数据包格式传输的时候具有更高的效率。这是因为JSON不像XML那样需要有严格的闭合标签，让有效数据量与总数据包比大大提升，从而在同等数据流量的情况下减少了网络的传输压力。

下面来做一个简单的比较。

(1) 部分省市数据用XML表示如下：

```
<?xml version="1.0" encoding="utf-8"?>
<country>
    <name>中国</name>
    <province>
        <name>广东</name>
```

```
<cities>
    <city>广州</city>
    <city>深圳</city>
</cities>
</province>
<province>
    <name>新疆</name>
    <cities>
        <city>乌鲁木齐</city>
    </cities>
</province>
</country>
```

(2) 以上部分省市数据用JSON表示如下：

```
{
    "name": "中国",
    "province": [
        {
            "name": "广东",
            "cities": {
                "city": ["广州", "深圳"]
            }
        },
        {
            "name": "新疆",
            "cities": {
                "city": ["乌鲁木齐"]
            }
        }
    ]
}
```

```
    }  
}  
}]  
}
```

可以看到，JSON的语法格式和清晰的层次结构非常简单，明显要比XML容易阅读，并且在数据交换方面JSON所使用的字符要比XML少得多，可以大大节约传输数据所占用的带宽。

7.2.2 JSON序列化与反序列化开源库

Java处理JSON数据有三个比较流行的开源类库：阿里巴巴的FastJson、谷歌的Gson和开源社区的Jackson。

Jackson是一个简单的、基于Java的JSON开源库。使用Jackson开源库，可以轻松地将Java POJO对象转换成JSON、XML格式字符串；同样也可以方便地将JSON、XML字符串转换成Java POJO对象。Jackson开源库的优点是：所依赖的Jar包较少、简单易用、性能也不错。另外，Jackson社区相对比较活跃。Jackson开源库的缺点是：对于复杂的POJO类型以及复杂的集合Map、List的转换结果，不是标准的JSON格式，或者会出现一些问题。

谷歌的Gson开源库是一个功能齐全的JSON解析库，起源于谷歌公司内部需求而由谷歌自行研发而来，在2008年5月公开发布第一版之后已被许多公司或用户应用。Gson可以完成复杂类型的POJO和JSON字符串的相互转换，转换能力非常强。

阿里巴巴的FastJson是一个高性能的JSON库。顾名思义，FastJson库采用独创的快速算法，将JSON转成POJO的速度提升到极

致，从性能上说，序列化速度超过其他JSON开源库。

在实际开发中，目前主流的策略是Gson和FastJson结合使用。在POJO序列化成JSON字符串的应用场景下，使用谷歌的Gson库；在JSON字符串反序列化成POJO的应用场景下，使用阿里巴巴的FastJson库。

下面将JSON的序列化和反序列化功能放在一个通用类JsonUtil中，方便后面统一使用。代码如下：

```
package com.crazymakercircle.util;  
//省略import  
public class JsonUtil {  
  
    //谷歌GsonBuilder构造器  
    static GsonBuilder gb = new GsonBuilder();  
    static {  
        //不需要html escape  
        gb.disableHtmlEscaping();  
    }  
  
    //序列化：使用Gson将 POJO 转成字符串  
    public static String pojoToJson(java.lang.Object obj) {  
        String json = gb.create().toJson(obj);  
        return json;  
    }  
  
    //反序列化：使用Fastjson将字符串转成 POJO对象  
    public static <T> T jsonToPojo(String json, Class<T>tClass)
```

```
{  
    T t = JSONObject.parseObject(json, tClass);  
    return t;  
}  
}
```

7.2.3 JSON序列化与反序列化的实战案例

下面通过一个小实例演示一下POJO对象的JSON协议的序列化和反序列化。

首先定义一个POJO类，名称为JsonMsg，包含id和content两个属性，然后使用lombok开源库的@Data注解为属性加上getter()和setter()方法。POJO类的源码如下：

```
package com.crazymakercircle.netty.protocol;  
//省略import  
  
@Data  
  
public class JsonMsg {  
    private int id; //id Field(字段)  
    private String content; //content Field(字段)  
    //序列化：调用通用方法，使用Gson转成字符串  
  
    public String convertToJson() {  
        return JsonUtil.pojoToJson(this);  
    }  
  
    //反序列化：使用FastJson转成Java POJO对象
```

```
public static JsonMsg parseFromJson(String json) {  
    return JsonUtil.jsonToPojo(json, JsonMsg.class);  
}  
}
```

在POJO类JsonMsg中，首先加上了一个JSON序列化方法convertToJson()，它调用通用类定义的JsonUtil.pojoToJson(Object)方法将对象自身序列化成JSON字符串。另外，JsonMsg加上了一个JSON反序列化方法parseFromJson(String)。它是一个静态方法，调用通用类定义的JsonUtil.jsonToPojo(String, Class)方法将JSON字符串反序列化成JsonMsg实例。

使用POJO类JsonMsg的序列化、反序列化的实战案例代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
public class JsonMsgDemo {  
    //构建JSON对象  
    public JsonMsg buildMsg() {  
        JsonMsg user = new JsonMsg();  
        user.setId(1000);  
        user.setContent("疯狂创客圈:高性能学习社群");  
        return user;  
    }  
  
    //测试用例: serialization & Deserialization  
    @Test
```

```
public void serAndDesr() throws IOException {  
    JsonMsg message = buildMsg();  
    //将POJO对象序列化成字符串  
    String json = message.convertToJson();  
    //可以用于网络传输，保存到内存或外存  
    Logger.info("json:=" + json);  
  
    //将JSON 字符串反序列化成POJO对象  
    JsonMsg Msg = JsonMsg.parseFromJson(json);  
    Logger.info("id:=" + inMsg.getId());  
    Logger.info("content:=" + inMsg.getContent());  
}  
}
```

7.2.4 JSON传输的编码器和解码器

从本质上来说，JSON格式仅仅是字符串的一种组织形式。所以，传输JSON所用到的协议与传输普通文本所使用的协议没有什么不同。下面使用常用的Head-Content协议来介绍一下JSON传输。

Head-Content数据包的解码过程（见图7-3）是：首先，使用Netty内置的LengthFieldBasedFrameDecoder解码Head-Content二进制数据包，解码出Content字段的二进制内容；然后，使用StringDecoder字符串解码器（Netty内置的解码器）将二进制内容解码成JSON字符串；最后，使用自定义业务解码器JsonMsgDecoder将JSON字符串解码成自定义的POJO业务对象。

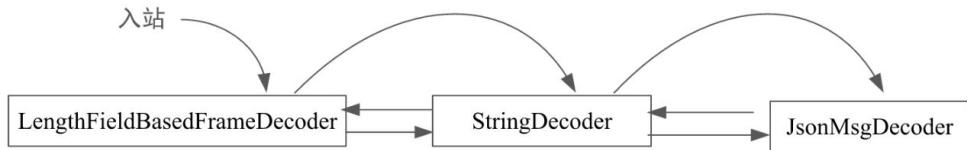


图7-3 JSON格式Head-Content数据包的解码过程

Head-Content数据包的编码过程（见图7-4）是：首先，使用Netty内置StringEncoder编码器将JSON字符串编码成二进制字节数组；然后，使用Netty内置LengthFieldPrepender编码器将二进制字节数组编码成Head-Content二进制数据包。

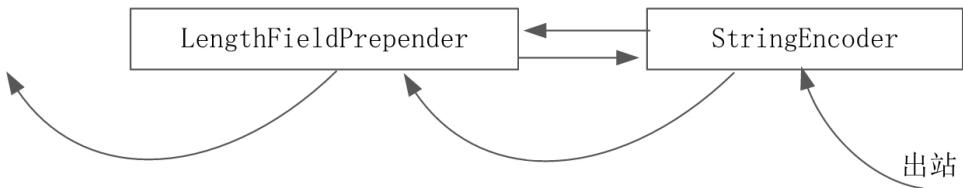


图7-4 JSON格式Head-Content数据包的编码过程

Netty内置LengthFieldPrepender编码器的作用是在数据包的前面加上内容的二进制字节数组的长度。这个编码器和LengthFieldBasedFrameDecoder解码器是天生的一对，常常配套使用。这组“天仙配”属于Netty所提供的一组非常重要的编码器和解码器，常用于Head-Content数据包的传输。

LengthFieldPrepender编码器有两个常用的构造器：

```

//构造器一
public LengthFieldPrepender(int lengthFieldLength) {
    this(lengthFieldLength, false);
}

```

```
//构造器二

public LengthFieldPrepender(int lengthFieldLength,
                             Boolean lengthIncludesLengthFieldLength)
{
    this(lengthFieldLength, 0,
         lengthIncludesLengthFieldLength);
}

//省略其他的构造器
```

在上面的构造器中，第一个参数lengthFieldLength表示Head长度字段所占用的字节数，第二个参数lengthIncludesLengthFieldLength表示Head字段的总长度值是否包含长度字段自身的字节数，如果该参数的值为true，表示长度字段的值（总长度）包含了用自己的字节数。如果该参数的值为false，表示长度值只包含内容的二进制数据的长度。lengthIncludesLengthFieldLength值一般设置为false。

7.2.5 JSON传输的服务端的实战案例

为了清晰地演示JSON传输，下面设计一个简单的客户端/服务端传输程序：服务器接收客户端的数据包，并解码成JSON，再转换成POJO；客户端将POJO转换成JSON字符串，编码后发送到服务端。

为了简化流程，此服务端的代码仅仅包含Inbound入站处理的流程，不包含OutBound出站处理的流程，是一个“丢弃”服务器。也就是说，服务端的程序仅仅读取客户端数据包并完成解码，服务端的程

序没有写出任何输出数据包到对端（客户端）。服务端实战案例的程序代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
  
public class JsonServer {  
    //省略成员属性、构造器  
  
    public void runServer() {  
        //创建反应器线程组  
  
        EventLoopGroup bossLoopGroup = new  
        NioEventLoopGroup(1);  
  
        EventLoopGroup workerLoopGroup = new  
        NioEventLoopGroup();  
  
        try {  
            //省略引导类的反应器线程、设置配置项等  
  
            //5 装配子通道流水线  
  
            b.childHandler(new  
ChannelInitializer<SocketChannel>() {  
                //有连接到达时会创建一个通道  
  
                protected void initChannel(SocketChannel ch)...{  
                    //管理子通道中的Handler  
  
                    //向子通道流水线添加3个Handler  
                    ch.pipeline().addLast(  
                        new LengthFieldBasedFrameDecoder(1024, 0,  
                        4, 0, 4));  
  
                    ch.pipeline().addLast(new
```

```
StringDecoder(CharsetUtil.UTF_8)) ;  
        ch.pipeline().addLast(new  
JsonMsgDecoder());  
    }  
}  
  
//省略端口绑定、服务监听、优雅关闭  
}  
  
//服务端业务处理器  
static class JsonMsgDecoder extends  
ChannelInboundHandlerAdapter {  
    @Override  
    public void channelRead(ChannelHandlerContext ctx,  
Object msg) ...{  
        String json = (String) msg;  
        JsonMsg jsonMsg = JsonMsg.parseFromJson(json);  
        Logger.info("收到一个 Json 数据包 =>>" + jsonMsg);  
    }  
}  
  
public static void main(String[] args) throws  
InterruptedException {  
    int port = NettyDemoConfig.SOCKET_SERVER_PORT;  
    new JsonServer(port).runServer();  
}  
}
```

7.2.6 JSON传输的客户端的实战案例

为了简化流程，客户端的代码仅仅包含Outbound出站处理的流程，不包含Inbound入站处理的流程。也就是说，客户端的程序仅仅进行数据的编码，然后把数据包写到服务端。客户端的程序并没有去处理从对端（服务端）过来的输入数据包。客户端的编码流程大致如下：

- (1) 通过谷歌的Gson框架，将POJO序列化成JSON字符串。
- (2) 使用StringEncoder编码器（Netty内置）将JSON字符串编码成二进制字节数组。
- (3) 使用LengthFieldPrepender编码器（Netty内置）将二进制字节数组编码成Head-Content格式的二进制数据包。

客户端实战案例的程序代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
  
public class JsonSendClient {  
    static String content = "疯狂创客圈：高性能学习社群！";  
    //省略成员属性、构造器  
  
    public void runClient() {  
        //创建反应器线程组  
  
        EventLoopGroup workerLoopGroup = new  
NioEventLoopGroup();  
  
        try {
```

```
//省略引导类的反应器线程、设置配置项等

//5 装配通道流水线

b.handler(new ChannelInitializer<SocketChannel>() {
    //初始化客户端通道

    protected void initChannel(SocketChannel ch) ...{
        //客户端通道流水线添加2个Handler

        ch.pipeline().addLast(new
LengthFieldPrepender(4));

        ch.pipeline().addLast(new

StringEncoder(CharsetUtil.UTF_8));

    }

}) ;

ChannelFuture f = b.connect();

//...

//阻塞，直到连接完成

f.sync();

Channel channel = f.channel();

//发送 JSON 字符串对象

for (int i = 0; i < 1000; i++) {

    JsonMsg user = build(i, i + "->" + content);

    channel.writeAndFlush(user.convertToJson());

    Logger.info("发送报文: " + user.convertToJson());

}

channel.flush();

//7 等待通道关闭的异步任务结束
```

```
//服务监听通道会一直等待通道关闭的异步任务结束

    ChannelFuture closeFuture = channel.closeFuture();

    closeFuture.sync();

} catch (Exception e) {

    e.printStackTrace();

} finally {

    //省略优雅关闭

}

//构建JSON对象

public JsonMsg build(int id, String content) {

    JsonMsg user = new JsonMsg();

    user.setId(id);

    user.setContent(content);

    return user;

}

//省略main()方法

}
```

整体执行次序是先启动服务端，然后启动客户端。启动后，客户端会向服务器发送1000个POJO转换成JSON后的字符串。如果能从服务器的控制台看到输出的JSON格式的字符串，说明程序运行是正确的。

7.3 使用Protobuf协议通信

Protobuf（Protocol Buffer）是Google提出的一种数据交换格式，是一套类似JSON或者XML的数据传输格式和规范，用于不同应用或进程之间的通信。Protobuf具有以下特点：

(1) 语言无关，平台无关

Protobuf支持Java、C++、Python、JavaScript等多种语言，支持跨多个平台。

(2) 高效

比XML更小（3~10倍）、更快（20~100倍）、更为简单。

(3) 扩展性、兼容性好

可以更新数据结构，而不影响和破坏原有的旧程序。

Protobuf既独立于语言又独立于平台。Google官方提供了多种语言的实现：Java、C#、C++、GO、JavaScript和Python。Protobuf的编码过程为：使用预先定义的Message数据结构将实际的传输数据进行打包，然后编码成二进制的码流进行传输或者存储。Protobuf的解码过程刚好与编码过程相反：将二进制码流解码成Protobuf自己定义的Message结构的POJO实例。

与JSON、XML相比，Protobuf算是后起之秀，只是Protobuf更加适合于高性能、快速响应的数据传输应用场景。Protobuf数据包是一种二进制格式，相对于文本格式的数据交换（JSON、XML）来说，速度要

快很多。Protobuf优异的性能使得它更加适用于分布式应用场景下的数据通信或者异构环境下的数据交换。

JSON、XML是文本格式，数据具有可读性；Protobuf是二进制数据格式，数据本身不具有可读性，只有反序列化之后才能得到真正可读的数据。正因为Protobuf是二进制数据格式，所以数据序列化之后体积相比JSON和XML要小，更加适合网络传输。

总体来说，在一个需要大量数据传输的应用场景中，数据量很大，选择Protobuf可以明显地减少传输的数据量和提升网络IO的速度。对于打造一款高性能的通信服务器来说，Protobuf传输协议是最高性能的传输协议之一。微信的消息传输就采用了Protobuf协议。

7.3.1 一个简单的proto文件的实战案例

Protobuf使用proto文件来预先定义的消息格式。数据包按照proto文件所定义的消息格式完成二进制码流的编码和解码。proto文件简单地说就是一个消息的协议文件，这个协议文件的后缀文件名为“.proto”。

作为演示，下面介绍一个非常简单的proto文件：仅仅定义一个消息结构体，并且该消息结构体也非常简单，仅包含两个字段。实例如下：

```
// [开始头部声明]  
syntax = "proto3";  
  
package com.crazymakercircle.netty.protocol;  
// [结束头部声明]
```

```
//[开始 Java选项配置]  
option java_package = "com.crazymakercircle.netty.protocol";  
option java_outer_classname = "MsgProtos";  
//[结束 Java选项配置]  
  
//[开始消息定义]  
message Msg {  
    uint32 id = 1;           //消息ID  
    string content = 2;     //消息内容  
}  
//[结束消息定义]
```

在.proto文件的头部声明中，需要声明一下所使用的Protobuf协议版本，示例中使用的是“proto3”版本。也可以使用旧一点的“proto2”版本，两个版本的消息格式有一些细微的不同，默认的协议版本为“proto2”。

Protobuf支持很多语言，所以它为不同的语言提供了一些可选的配置选项，使用option关键字。option java_package选项的作用为：在生成proto文件中消息的POJO类和Builder（构造者）的Java代码时，将生成的Java代码放入该选项所指定的package类路径中。option java_outer_classname选项的作用为：在生成proto文件所对应的Java代码时，生成的Java外部类使用配置的名称。

在proto文件中，使用message关键字来定义消息的结构体。在生成proto对应的Java代码时，每个具体的消息结构体将对应于一个最终的Java POJO类。结构体的字段（Field）对应到POJO类的属性

(Attribute)。也就是说，每定义一个message结构体相当于声明一个Java中的类。proto文件的message可以内嵌message，就像Java的内部类一样。

每个消息结构体可以有多个字段。定义一个字段的格式为“类型名称 = 编号”。例如，“string content = 2;”表示该字段是String类型，字段名为content，编号为2。字段编号表示在Protobuf数据包的序列化、反序列化时该字段的具体排序。

在一个proto文件中可以声明多个message，大部分情况下会把存在依赖关系或者包含关系的message结构体写入一个proto文件，将那些没有关系、相互独立的message结构体分别写入不同的文件，这样便于管理。

7.3.2 通过控制台命令生成POJO和Builder

完成“.proto”文件定义后，下一步是生成消息的POJO类和Builder（构造者）类。生成Java类有两种方式：一种是通过控制台命令；另一种是使用Maven插件。

先看第一种方式：通过控制台命令生成消息的POJO类和Builder构造者。

首先从

<https://github.com/protocolbuffers/protobuf/releases>下载Protobuf的安装包，可以选择不同的版本，这里下载的是3.6.1的Java版本。在Windows下解压后执行安装。（备注：这里以Windows平台为例子，对于Linux或者Mac平台，大家可自行尝试。）

生成构造者代码需要用到安装文件中的protoc.exe可执行文件。安装完成后，设置一下path环境变量，将proto的安装目录加入path环境变量中。

下面开始使用protoc.exe文件生成Java的Builder（构造者），生成的命令如下：

```
protoc.exe --java_out=./src/main/java/ ./Msg.proto
```

在上面的命令中，使用的proto文件的名称为./Msg.proto，所生成的POJO类和构造者类的输出文件夹为./src/main/java/。

使用命令行生成Java类的操作比较烦琐，另一种更加方便的方式是使用protobuf-maven-plugin插件生成Java类。

7.3.3 通过Maven插件生成POJO和Builder

使用protobuf-maven-plugin插件可以非常方便地生成消息的POJO类和Builder（构造者）类的Java代码。在Maven的pom文件中增加此插件的配置项，具体如下：

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.5.0</version>
  <extensions>true</extensions>
  <configuration>
    <!--proto文件路径-->
```

```
<protoSourceRoot>
    ${project.basedir}/protobuf</protoSourceRoot>
    <!-- 目标路径-->

<outputDirectory>${project.build.sourceDirectory}
</outputDirectory>
    <!-- 设置是否在生成Java文件之前清空outputDirectory的文件-->
    <!-->

<clearOutputDirectory>false</clearOutputDirectory>
    <!--临时目录-->
    <temporaryProtoFileDirectory>
        ${project.build.directory}/protoc-
temp
    </temporaryProtoFileDirectory>
    <!--protoc可执行文件路径-->
    <protocExecutable>
        ${project.basedir}/protobuf/protoc3.6.1.exe
    </protocExecutable>
</configuration>
<executions>
    <execution>
        <goals>
            <goal>compile</goal>
            <goal>test-compile</goal>
        </goals>
    </execution>
</executions>
```

```
</execution>  
</executions>  
</plugin>
```

protobuf-maven-plugin插件的配置项介绍如下：

- **protoSourceRoot**: proto消息结构体所在文件的路径。
- **outputDirectory**: 生成的POJO类和Builder类的目标路径。
- **protocExecutable**: protobuf的Java代码生成工具的 protoc3.6.1.exe 可执行文件的路径。

配置好之后，执行插件的compile命令，Java代码就生成了；在 Maven的项目编译时，POJO类和Builder类也会自动生成。

7.3.4 Protobuf序列化与反序列化的实战案例

在Maven的pom.xml文件中加上protobuf的Java运行包的依赖，代码如下：

```
<dependency>  
    <groupId>com.google.protobuf</groupId>  
    <artifactId>protobuf-java</artifactId>  
    <version>${protobuf.version}</version>  
</dependency>
```

这里的protobuf.version版本号为3.6.1。需要注意的是：Java运行时的Protobuf依赖坐标的版本，.proto消息结构体文件中的syntax 配置项值（Protobuf协议的版本号），以及通过proto文件生成POJO和

Builder类的protoc3.6.1.exe可执行文件的版本，这三个版本需要配套一致。

1. 使用Builder构造POJO消息对象

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtobufDemo {
    public static MsgProtos.Msg buildMsg() {
        MsgProtos.Msg.Builder personBuilder =
                MsgProtos.Msg.newBuilder();
        personBuilder.setId(1000);
        personBuilder.setContent("疯狂创客圈:高性能学习社群");
        MsgProtos.Msg message = personBuilder.build();
        return message;
    }
    //...
}
```

Protobuf为每个message结构体生成的Java类中包含了一个POJO类、一个Builder类。构造POJO消息，首先使用POJO类的newBuilder静态方法获得一个Builder，其次POJO每一个字段的值需要通过Builder的setter()方法去设置。字段值设置完成之后，使用构造者的build()方法构造出POJO消息对象。

2. 序列化与反序列化的方式一

获得消息POJO的实例之后，可以通过多种方法将POJO对象序列化成二进制字节或者反序列化。方式一为调用Protobuf POJO对象的toByteArray()方法将POJO对象序列化成字节数组，具体的代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
public class ProtobufDemo {  
  
    //第1种方式：序列化与反序列化  
  
    @Test  
    public void serAndDesr1() throws IOException {  
        MsgProtos.Msg message = buildMsg();  
        //将Protobuf对象序列化成二进制字节数组  
        byte[] data = message.toByteArray();  
        //可以用于网络传输，保存到内存或外存  
        ByteArrayOutputStream outputStream = new  
        ByteArrayOutputStream();  
        outputStream.write(data);  
        data = outputStream.toByteArray();  
        //二进制字节数组反序列化成Protobuf对象  
        MsgProtos.Msg inMsg = MsgProtos.Msg.parseFrom(data);  
        Logger.info("id:=" + inMsg.getId());  
        Logger.info("content:=" + inMsg.getContent());  
    }  
    //...  
}
```

这种方式首先通过调用Protobuf POJO对象的toByteArray()方法将POJO对象序列化成字节数组，然后通过调用Protobuf POJO类的parseFrom(byte[] data)静态方法从字节数组中重新反序列化得到POJO新的实例。

这种方式类似于普通Java对象的序列化，适用于很多将Protobuf的POJO序列化到内存或者外存（如物理硬盘）的应用场景。

3. 序列化与反序列化的方式二

这种方式通过调用Protobuf生成的POJO对象的writeTo(OutputStream)方法将POJO对象的二进制字节写出到输出流。通过调用Protobuf生成的POJO对象的parseFrom(InputStream)方法，Protobuf从输入流中读取二进制码然后反序列化，得到POJO新的实例。具体的代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
public class ProtobufDemo {  
    //...  
    //第2种方式：序列化与反序列化  
    @Test  
    public void serAndDesr2() throws IOException {  
        MsgProtos.Msg message = buildMsg();  
        //序列化到二进制码流  
        ByteArrayOutputStream outputStream = new  
        ByteArrayOutputStream();  
        message.writeTo(outputStream);
```

```
        ByteArrayInputStream inputStream =
        new ByteArrayInputStream(outputStream.toByteArray());

        //从二进码流反序列化成Protobuf对象
        MsgProtos.Msg inMsg =
        MsgProtos.Msg.parseFrom(inputStream);
        Logger.info("id:=" + inMsg.getId());
        Logger.info("content:=" + inMsg.getContent());
    }
}
```

以上代码调用POJO对象的writeTo(OutputStream)方法将自己的二进制字节写出到输出流，然后调用静态类的parseFrom(InputStream)方法，Protobuf从输入流中读取二进制码重新反序列化，得到POJO新的实例。

在阻塞式的二进制码流传输应用场景中，这种序列化和反序列化的方式是没有问题的。例如，可以将二进制码流写入阻塞式的Java IO套接字或者输出到文件。但是，这种方式在异步操作的NIO应用场景中存在粘包/半包的问题。

4. 序列化与反序列化方式三

这种方式通过调用Protobuf生成的POJO对象的writeDelimitedTo(OutputStream)方法在序列化的字节码之前添加了字节数组的长度。这一点类似于前面介绍的Head-Content协议，只不过Protobuf做了优化，长度的类型不是固定长度的int类型，而是可变长度varint32类型。具体实例如下：

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtobufDemo {
//...
//第3种方式：序列化与反序列化
//带字节长度：[字节长度][字节数据]，用于解决粘包/半包问题
@Test
public void serAndDesr3() throws IOException {
    MsgProtos.Msg message = buildMsg();
    //序列化到二进制码流
    ByteArrayOutputStream outputStream =
        new
    ByteArrayOutputStream();
    message.writeDelimitedTo(outputStream);
    ByteArrayInputStream inputStream =
        new
    ByteArrayInputStream(outputStream.toByteArray());
    //从二进制码字节流反序列化成Protobuf对象
    MsgProtos.Msg inMsg =
        MsgProtos.Msg.parseDelimitedFrom(inputStream);
    Logger.info("id:=" + inMsg.getId());
    Logger.info("content:=" + inMsg.getContent());
}
}
```

反序列化时，调用Protobuf生成的POJO类的
parseDelimitedFrom(InputStream)静态方法，从输入流中先读取
varint32类型的长度值，然后根据长度值读取此消息的二进制字节，
再反序列化得到POJO新的实例。

这种方式用于异步操作的NIO应用场景中，解决了粘包/半包的问
题。

7.4 Protobuf编解码的实战案例

Netty默认支持Protobuf的编码与解码，内置了一套基础的Protobuf编码和解码器。

7.4.1 Netty内置的Protobuf基础编码器/解码器

Netty内置的基础Protobuf编码器、解码器为ProtobufEncoder、ProtobufDecoder。此外，还提供了一组简单的解决半包问题的编码器和解码器。

1. ProtobufEncoder编码器

翻开Netty源代码，我们发现ProtobufEncoder的实现逻辑非常简单，直接调用了Protobuf POJO实例的toByteArray()方法将自身编码成二进制字节，然后放入Netty的ByteBuf缓冲区中，接着会被发送到下一站编码器。其源码如下：

```
package io.netty.handler.codec.protobuf;

...
@Sharable
public class ProtobufEncoder extends

MessageToMessageEncoder<MessageLiteOrBuilder> {

    @Override
    protected void encode(ChannelHandlerContext ctx,
```

```
        MessageLiteOrBuilder msg,  
List<Object> out)  
    throws Exception {  
    if (msg instanceof MessageLite) {  
        out.add(Unpooled.wrappedBuffer(  
            ((MessageLite) msg).toByteArray()));  
        return;  
    }  
    if (msg instanceof MessageLite.Builder) {  
        out.add(Unpooled.wrappedBuffer((  
            MessageLite.Builder)  
msg).build().toByteArray()));  
    }  
}  
}
```

2. ProtobufDecoder解码器

ProtobufDecoder和ProtobufEncoder相互对应，只不过在使用的时候ProtobufDecoder解码器需要指定一个Protobuf POJO实例作为解码的参考原型（prototype），解码时会根据原型实例找到对应的Parser解析器，将二进制的字节解码为Protobuf POJO实例。

```
new ProtobufDecoder(MsgProtos.Msg.getDefaultInstance())
```

在Java NIO通信中，仅仅使用以上这组编码器和解码器，传输过程中会存在粘包/半包的问题。Netty也提供了配套的Head-Content类

型的Protobuf编码器和解码器，在二进制码流之前加上二进制字节数组的长度。

3. ProtobufVarint32LengthFieldPrepender长度编码器

这个编码器的作用是在ProtobufEncoder生成的字节数组之前前置一个varint32数字，表示序列化的二进制字节数量或者长度。

4. ProtobufVarint32FrameDecoder长度解码器

ProtobufVarint32FrameDecoder和ProtobufVarint32LengthFieldPrepender相互对应，其作用是根据数据包中长度域（varint32类型）中的长度值解码一个足额的字节数组，然后将字节数组交给下一站的解码器ProtobufDecoder。

什么是varint32类型的长度？Protobuf为什么不用int这种固定类型的长度？

varint32是一种紧凑的表示数字的方法，不是一种固定长度（如32位）的数字类型。varint32用一个或多个字节来表示一个数字，值越小，使用的字节数越少，值越大使用的字节数越多。varint32根据值的大小自动进行收缩，能够减少用于保存长度的字节数。也就是说，varint32与int类型的最大区别是：varint32用一个或多个字节来表示一个数字，int是固定长度的数字。varint32不是固定长度，所以为了更好地减少通信过程中的传输量，消息头中的长度尽量采用varint格式。

至此，Netty内置的Protobuf编码器和解码器已经初步介绍完，可以通过这两组编码器/解码器完成Head-Content（Length + Protobuf

Data) 协议的数据传输。但是，在更加复杂的传输应用场景下，Netty 的内置编码器和解码器是不够用的。例如，在Head部分需要加上魔数字段进行安全验证或者需要对Protobuf字节内容进行加密和解密，或者在其他复杂的传输应用场景下，需要定制属于自己的Protobuf编码器和解码器。

7.4.2 Protobuf传输的服务端的实战案例

为了清晰地演示Protobuf传输，下面设计一个简单的客户端/服务器传输程序：服务器接收客户端的数据包，并解码成Protobuf的POJO；客户端将Protobuf的POJO编码成二进制数据包，再发送到服务端。

在服务端，Protobuf协议的解码过程如下：

首先，使用Netty内置的ProtobufVarint32FrameDecoder，根据varint32格式的可变长度值，从入站数据包中解码出二进制Protobuf字节码。然后，使用Netty内置的ProtobufDecoder解码器将字节码解码成Protobuf POJO对象。最后，自定义一个ProtobufBusinessDecoder解码器来处理Protobuf POJO对象。

服务端的实战案例程序代码如下：

```
package com.crazymakercircle.netty.protocol;  
//...  
public class ProtoBufServer  
{  
    //省略成员属性、构造器
```

```
public void runServer()
{
    //创建反应器线程组

    EventLoopGroup bossLoopGroup = new
    NioEventLoopGroup(1);

    EventLoopGroup workerLoopGroup = new
    NioEventLoopGroup();

    try
    {
        //省略引导类的反应器线程、设置配置项

        //5 装配子通道流水线

        b.childHandler(new
        ChannelInitializer<SocketChannel>()

        {
            //有连接到达时会创建一个通道

            protected void initChannel(SocketChannel ch) ...
            {

                //流水线管理子通道中的Handler业务处理器

                //向子通道流水线添加3个Handler业务处理器

                ch.pipeline().addLast(
                    new
                    ProtobufVarint32FrameDecoder());
                    ch.pipeline().addLast(
                    new
                    ProtobufDecoder(MsgProtos.Msg.getDefaultInstance()));
                    ch.pipeline().addLast(new
                    ProtobufBusinessDecoder());
    
```

```
        }

    });

    //省略端口绑定、服务监听、优雅关闭
}

//服务端的Protobuf业务处理器
static class ProtobufBussinessDecoder
    extends ChannelInboundHandlerAdapter
{
    @Override
    public void channelRead(
        ChannelHandlerContext ctx, Object
msg) ... {

    MsgProtos.Msg protoMsg = (MsgProtos.Msg) msg;
    //经过流水线的各个解码器取得了POJO实例
    Logger.info("收到一个Protobuf POJO =>>");
    Logger.info("protoMsg.getId() :=" +
protoMsg.getId());
    Logger.info("protoMsg.getContent() :=" +
protoMsg.getContent());
}

}

public static void main(String[] args) throws
InterruptedException
```

```

{
    int port = NettyDemoConfig.SOCKET_SERVER_PORT;
    new ProtoBufServer(port).runServer();
}
}

```

7.4.3 Protobuf传输的客户端的实战案例

在客户端开始出站之前，需要提前构造好Protobuf的POJO对象，然后可以使用通道的write/writeAndFlush方法启动出站处理的流水线执行工作。

客户端的出站处理流程中，Protobuf协议的编码过程（见图7-5），如下：

- (1) 使用Netty内置的ProtobufEncoder将Protobuf POJO对象编码成二进制的字节数组。
- (2) 使用Netty内置的ProtobufVarint32LengthFieldPrepender编码器，加上varint32格式的可变长度。Netty会将完成了编码后的Length+Content格式的二进制字节码发送到服务端。

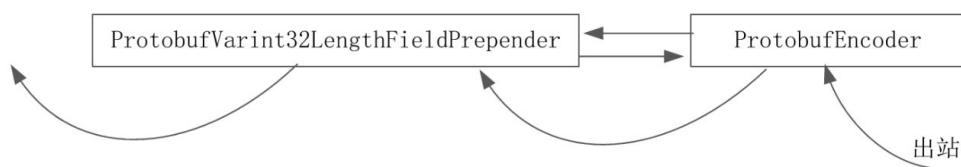


图7-5 Protobuf协议的编码过程

一个简单的Protobuf传输的客户端的案例代码如下：

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtoBufSendClient {
    static String content = "疯狂创客圈：高性能学习社群!";
    //省略成员属性、构造器
    public void runClient() {
        //创建反应器线程组
        EventLoopGroup workerLoopGroup = new
NioEventLoopGroup();
        try {
            //省略反应器组、IO通道、通道参数等设置
            //5 装配通道流水线
            b.handler(new ChannelInitializer<SocketChannel>() {
                //初始化客户端通道
                protected void initChannel(SocketChannel ch) ...{
                    //客户端流水线添加2个Handler业务处理器
                    ch.pipeline().addLast(
                        new
                        ProtobufVarint32LengthFieldPrepender());
                    ch.pipeline().addLast(new
ProtobufEncoder());
                }
            });
            ChannelFuture f = b.connect();
            //...
            //阻塞，直到连接完成
            f.sync();
        }
```

```
    Channel channel = f.channel();

    //发送Protobuf对象
    for (int i = 0; i < 1000; i++) {
        MsgProtos.Msg user = build(i, i + "->" +
content);
        channel.writeAndFlush(user);
        Logger.info("发送报文数: " + i);
    }
    channel.flush();
    //省略关闭等待、优雅关闭
}

//构建ProtoBuf对象
public MsgProtos.Msg build(int id, String content) {
    MsgProtos.Msg.Builder builder =
MsgProtos.Msg.newBuilder();
    builder.setId(id);
    builder.setContent(content);
    return builder.build();
}

public static void main(String[] args) throws
InterruptedException {
    int port = NettyDemoConfig.SOCKET_SERVER_PORT;
    String ip = NettyDemoConfig.SOCKET_SERVER_IP;
    new ProtoBufSendClient(ip, port).runClient();
}
```

```
    }  
}
```

服务端和客户端整体的执行次序是：先启动服务端，再启动客户端。启动后，客户端会向服务器发送构造好的1000个Protobuf POJO实例。如果能从服务器的控制台看到输出的POJO实例的属性值，就说明程序运行是正确的。

7.5 详解Protobuf协议语法

在Protobuf中，通信协议的格式是通过proto文件定义的。一个proto文件有两大组成部分：头部声明、消息结构体的定义。头部声明部分主要包含了协议的版本、包名、特定语言的选项设置等；消息结构体部分可以定义一个或者多个消息结构体。

在Java中，当用Protobuf编译器（如protoc3.6.1.exe）来编译.proto文件时，编译器将生成Java语言的POJO消息类和Builder构造者类。通过POJO消息类和Builder构造者，Java程序可以很容易地操作在proto文件中定义的消息和字段，包括获取、设置字段值，将消息序列化到一个输出流中（序列化），以及从一个输入流中解析消息（反序列化）。

7.5.1 proto文件的头部声明

前面介绍了一个简单的proto文件，其头部声明如下：

```
// [开始声明]
syntax = "proto3";
// 定义Protobuf的包名称空间
package com.crazymakercircle.netty.protocol;
// [结束声明]

// [开始 Java 选项配置]
option java_package = "com.crazymakercircle.netty.protocol";
```

```
option java_outer_classname = "MsgProtos";  
//[结束 Java 选项配置]
```

对其中用到的主要配置选项做一下简单的介绍：

1. syntax版本号

对于一个proto文件而言，文件第一个非空、非注释的行必须注明Protobuf的语法版本，这里为syntax = "proto3"，如果没有声明，则默认版本是"proto2"。

2. package包

和Java语言类似，通过package指定包名，用来避免消息名字相冲突。如果两个消息的名称相同，但是package包名不同，那么它们是可以共同存在的。

通过package，还可以实现消息的引用。例如，假设第一个proto文件定义了一个Msg结构体，package包名如下：

```
package com.crazymakercircle.netty.protocol;  
message Msg{ ... }
```

假设另一个proto文件也定义了一个相同名字的消息，package包名如下：

```
package com.other.netty.protocol;  
message Msg{  
//...  
com.crazymakercircle.netty.protocol.Msg crazyMsg = 1;
```

```
//...  
}
```

我们可以看到，在第二个proto文件中，可以用“包名+消息名称”（全限定名）来引用第一个proto文件中的Msg结构体，而且不同包中的结构体可以同名。这一点和Java中package的使用方法是一样的。

另外， package指定包名后会对应到生成的消息POJO代码和Builder代码。在Java语言中，会以package指定的包名作为生成的POJO类的包名。

3. option配置选项

不是所有的option配置选项都会生效，option选项是否生效与proto文件使用的一些特定语言场景有关。在Java语言中，以“java_”打头的option选项会生效。

选项option java_package表示Protobuf编译器在生成Java POJO消息类时，生成在此选项所配置的Java包名下。如果没有该选项，则会以头部声明中的package作为Java包名。

选项option java_multiple_files表示在生成Java类时的打包方式，具体来说有以下两种方式：

方式1：一个消息对应一个独立的Java类。

方式2：所有的消息都作为内部类，打包到一个外部类中。

此选项的值默认为false，即方式2，表示使用外部类打包的方式。如果设置option java_multiple_files= true，则使用第一种方式生成Java类，则一个消息对应一个POJO Java类，多个消息结构体会对应到多个类。

选项option java_outer_classname表示Protobuf编译器在生成Java POJO消息类时，如果采用的是上面的方式2（全部POJO类都作为内部类打包在同一个外部类中），就以此选项所配置的值作为唯一外部类的类名。

7.5.2 Protobuf的消息结构体与消息字段

定义一个Protobuf消息结构体的关键字为message。一个消息结构体由一个或者多个消息字段组合而成。下面是一个简单的例子：

```
// [开始消息定义]  
message Msg {  
    uint32 id = 1;           // 消息ID  
    string content = 2;     // 消息内容  
}  
// [结束消息定义]
```

Protobuf消息字段的格式为：

限定修饰符① | 数据类型② | 字段名称③ | = | 分配标识号④

对以上格式中的4个部分介绍如下：

(1) 消息字段的限定修饰符

repeated限定修饰符：表示该字段可以包含 $0^{\sim}N$ 个元素值，相当于Java中的List（列表数据类型）。

singular限定修饰符：表示该字段可以包含 $0^{\sim}1$ 个元素值。
singular限定修饰符是默认的字段修饰符。

reserved限定修饰符：指定保留字段名称（Field Name）和分配标识号（Assigning Tags），用于将来的扩展。下面是一个简单的**reserved**限定修饰符使用的例子：

```
message MsgFoo{  
    //...  
    reserved 12, 15, 9 to 11;      //预留将来使用的分配标识号  
    (Assigning Tags),  
    reserved "foo", "bar";        //预留将来使用的字段名（field  
    name)  
}
```

（2）消息字段的数据类型

类似于Java中的数据类型，详见下一节。

（3）消息字段的字段名称

字段名称的命名与Java语言的成员变量命名方式几乎是相同的。Protobuf建议字段的命名以下划线分隔（例如first_name），而不是驼峰式（例如firstName）。

（4）消息字段的分配标识号

在消息定义中，每个字段都有唯一的一个数字标识符，可以理解为字段编码值，叫作分配标识号（Assigning Tags）。通过该值，通信双方才能互相识别对方的字段。当然，相同的编码值，它的限定修饰符和数据类型必须相同。分配标识号是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。

分配标识号的取值范围为 $1 \sim 2^{32}$ （4 294 967 296）。其中，编号[1, 15]之内的分配标识号，时间和空间效率都是最高的。因为[1, 15]之内的标识号在编码的时候只会占用一个字节，[16, 2047]之内的标识号要占用两个字节。所以，那些频繁出现的消息字段应该使用[1, 15]之内的标识号。切记：要为将来有可能添加的、频繁出现的字段预留一些标识号。另外，[1900, 2000]之内的标识号为Protobuf内部保留值，建议不要在自己的项目中使用。

标识号的特点是：一个消息结构体中的标识号是可以不连续的；在同一个消息结构体中，不同的字段不能使用相同的标识号。

7.5.3 Protobuf字段的数据类型

Protobuf定义了一套基本数据类型，具体如表7-1所示，但是这些数据类型几乎都可以对应到C++/Java等语言的基本数据类型。

表7-1 Protobuf定义的基本数据类型

.proto Type	说 明	对应的 Java Type
double	双精度浮点类型	double
float	单精度浮点类型	float
int32	使用变长编码，对于负值的效率很低，如果字段有可能有负值，就使用 sint64 替代	int
uint32	使用变长编码的 32 位整数类型	int
uint64	使用变长编码的 64 位整数类型	long
sint32	使用变长编码，有符号的 32 位整数类型值。这些编码在负值时比 int32 高效得多	int
sint64	使用变长编码，有符号的 64 位整数类型值。编码时比通常的 int64 高效	long
fixed32	4 个字节，如果数值总是比 2^{28} 大，那么这个类型会比 uint32 高效	int
fixed64	8 个字节，如果数值总是比 2^{56} 大，那么这个类型会比 uint64 高效	long
sfixed32	4 个字节	int
sfixed64	8 个字节	long
bool	布尔类型	boolean
string	一个字符串必须是 UTF-8 编码或者 7-bit ASCII 编码的文本	string
bytes	可能包含任意顺序的字节数据	bytestring

变长编码的类型（如int32）表示打包的字节并不是固定的，而是根据数据的大小或者长度来定的。例如int32，如果数值比较小，在0~127时，就使用一个字节打包。

定长编码（如fixed32）和变长编码（如int32）的区别是：fixed32的打包效率比int32的效率高，但是使用的空间一般比int32多。因此，定长编码时间效率高，变长编码空间效率高，可以根据项目的实际情况选择。一般情况下可以选择fixed32，但是遇到对传输效率要求比较苛刻的环境时，可以选择int32。

7.5.4 proto文件的其他语法规范

1. 声明

在需要多个消息结构体时，proto文件可以像Java语言的类文件一样按照模块进行分开设计，所以一个项目可能有多个proto文件，一个

文件在需要依赖其他proto文件时可以通过import导入。导入的操作，这和Java的import操作大致相同。

2. 嵌套消息

proto文件支持嵌套消息。消息中既可以包含另一个消息实例作为其字段，也可以在消息中定义一个新的消息。

```
message Outer {          //Level 0
    message MiddleA{      //Level 1
        message Inner {    //Level 2
            int64 ival = 1;
            bool booly = 2;
        }
    }
    message MiddleB{      //Level 1
        message Inner {    //Level 2
            int32 ival = 1;
            bool booly = 2;
        }
    }
}
```

如果想在父消息类型的外部重复使用这些内部的消息类型，那么可以使用Parent.Type的形式来引用，例如：

```
message SomeOtherMessage {
    Outer.MiddleA.Inner ref = 1;
```

}

3. 枚举

枚举的定义和Java相同，但是有一些限制：枚举值必须是大于等于0的整数。另外，需要使用分号（;）分隔枚举变量，而不是Java语言中的逗号“，”。

```
enum VoipProtocol
{
    H323 = 1;
    SIP = 2;
    MGCP = 3;
    H248 = 4;
}
```

第8章 基于Netty单体IM系统的开发实战

本章是Netty应用的综合实战篇，将综合使用前面学到的编码器、解码器、业务处理器等知识完成一个单体聊天（IM）系统的设计和实现。

说明

疯狂创客圈社群在不断地进行聊天器的交流和讨论，Netty单体IM系统代码也在不断地优化，为了方便代码管理和团队协助，本系统的源码托管在码云Git仓库中，大家可以去拉取最新代码，其地址为<https://gitee.com/crazymaker/SimpleCrayIM.git>。

下面介绍单体IM系统中使用的自定义Protobuf编码器和解码器。

8.1 自定义Protobuf编解码器

Netty内置了一组Protobuf编解码器——ProtobufDecoder解码器和ProtobufEncoder编码器，它们负责Protobuf生成的POJO实例和二进制字节之间的编码和解码。除此之外，Netty还自带了一组配套的半包处理器：ProtobufVarint32FrameDecoder、ProtobufVarint32LengthFieldPrepender拆包解码器和编码器，它们为二进制ByteBuf加上varint32格式的可变长度，解决了Protobuf传输过程中的粘包/半包问题。

使用Netty内置的Protobuf系列编解码器，虽然可以解决简单的Protobuf协议的传输问题，但是对复杂Head-Content协议（例如数据包头部存在魔数、版本号字段，具体如图8-1所示）的解析，内置Protobuf系列编解码器就显得无能为力了，这种情况下需要自定义Protobuf编码器和解码器。

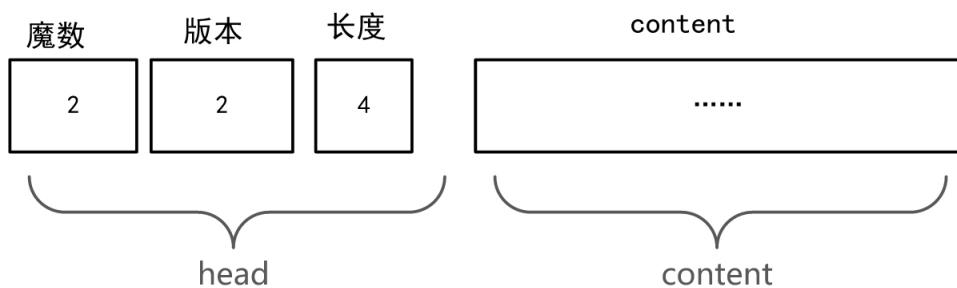


图8-1 复杂Head-Content协议的数据包

数据包中魔数的作用是什么？魔数可以理解为通信的口令。例如，在电影《智取威虎山》中土匪内部使用暗号接头，魔数和接头暗号在原理上是一样的。无论是服务端还是客户端，通信之前首先要对口令，如果口令不对，就不是安全的数据包，不符合自定义协议规

范。通过魔数校验，服务端能够在第一时间识别出不符合规范的数据包，当收到非法包时，为了安全考虑，可以直接关闭连接。

数据包中版本号的作用是什么？如果在程序中有通信协议升级的需求，又需要同时兼顾新旧版本的协议，就会用这个版本号。例如，APP协议升级后，旧版本APP还需要使用。

8.1.1 自定义Protobuf编码器

自定义Protobuf编码器，通过继承Netty中基础的MessageToByteEncoder编码器类，实现其抽象的编码方法encode()，在该方法中把以下内容写入目标ByteBuf：

- (1) 写入待发送的Protobuf POJO实例的二进制字节长度。
- (2) 写入其他的字段，如魔数、版本号。
- (3) 写入Protobuf POJO实例的二进制字节码内容。

按照上面的步骤自定义一个ProtobufEncoder编码器，大致代码如下：

```
package com.crazymakercircle.im.common.codec;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class ProtobufEncoder extends  
    MessageToByteEncoder<ProtoMsg.Message>  
{  
    @Override  
    protected void encode(ChannelHandlerContext ctx, ProtoMsg.Message msg, ByteBuf out)  
    {  
        // 将消息对象序列化为字节  
        byte[] bytes = msg.toByteArray();  
        out.writeInt(bytes.length);  
        out.writeBytes(bytes);  
    }  
}
```

```
protected void encode(ChannelHandlerContext ctx,
                      ProtoMsg.Message msg, ByteBuf out) ...{
    byte[] bytes = msg.toByteArray();           //将对象转换为字节
    int length = bytes.length;                  //读取消息的长度
    //将消息长度写入，这里只用两个字节，最大为32767
    out.writeShort(length);
    //省略魔数、版本号的写入，写入的方式、写入长度是类似的
    //消息体中包含我们要发送的数据
    out.writeBytes(msg.toByteArray());
}
```

说明

这里写入的消息长度调用了`writeShort(length)`方法，长度仅仅是两个字节，表明了数据包最大的净负荷长度为32 767个字节（有符号的短整数类型）。如果数据包的长度较大，需要传输更多的内容，可以调用`writeInt(length)`方法写入长度。

8.1.2 自定义Protobuf解码器

自定义Protobuf解码器，通过继承Netty中基础的`ByteToMessageDecoder`解码器类实现，在其继承的`decode()`方法中，将`ByteBuf`字节码解码成Protobuf的POJO实例，大致过程如下：

- (1) 读取长度，如果长度位数不够，就终止读取。
- (2) 读取魔数、版本号等其他字段。
- (3) 按照净长度读取内容。如果内容的字节数不够，则恢复到之前的起始位置（也就是长度的位置），然后终止读取。

自定义Protobuf解码器的核心代码如下所示：

```
package com.crazymakercircle.im.common.codec;  
//...  
@Slf4j  
public class ProtobufDecoder extends ByteToMessageDecoder  
{  
    @Override  
    protected void decode(ChannelHandlerContext ctx, ByteBuf  
    in,  
    List<Object> out) throws Exception  
    {  
        //标记一下当前的读指针readIndex的位置  
        in.markReaderIndex();  
        //判断包头的长度  
        if (in.readableBytes() < 2)          //不够包头中的长度  
        {  
            return;  
        }  
  
        int length = in.readShort();      //读取传送过来的消息的长度
```

```
if (length < 0) //长度如果小于0
{
    ctx.close(); //非法数据，关闭连接
}

if (length >in.readableBytes()) //可读字节少于预期消息长度
{
    in.resetReaderIndex(); //重置读取位置
    return;
}

//省略：读取魔数、版本号等其他数据
//省略：读取内容

byte[] array ;
if (in.hasArray()) //堆缓冲
{
    ByteBuf slice=in.slice();
    array=slice.array();
}
else
{
    array = new byte[length]; //直接缓冲
    in.readBytes( array, 0, length);
}

//字节转成Protobuf的POJO对象

ProtoMsg.Message outmsg =
ProtoMsg.Message.parseFrom(array);

if (outmsg != null)
```

```
{  
    out.add(outmsg); //将Protobuf的POJO实例加入出站List容器  
}  
}  
}
```

在自定义的解码过程中，如果需要进行版本号或者魔数的校验也是非常简单的：只需读取相应的字节数进行合理的校验即可。

8.1.3 IM系统中Protobuf消息格式的设计

一般来说，IM系统所涉及消息的格式不管是直接使用二进制承载还是使用XML、JSON等字符串承载，一般可以分为三大消息类型：请求消息，应答消息，命令消息。每个往来的消息报文基本上会包含一个序列号和一个类型定义，序列号用来唯一区分一个消息，类型用来决定消息的处理方式。

IM系统的Protobuf消息格式大致有以下几个可供参考的原则。

1. 原则一：消息类型使用enum定义

在proto协议文件中，可以定义一个HeadType枚举类型，包含系统用到的所有消息类型，具体例子如下：

```
enum HeadType {  
    LOGIN_REQUEST = 0; //登录请求  
    LOGIN_RESPONSE = 1; //登录响应  
    LOGOUT_REQUEST = 2; //登出请求
```

```
LOGOUT_RESPONSE = 3;           // 登出响应
KEEPALIVE_REQUEST = 4;          // 心跳请求
KEEPALIVE_RESPONSE = 5;         // 心跳响应
MESSAGE_REQUEST = 6;           // 聊天消息请求
MESSAGE_RESPONSE = 7;          // 聊天消息响应
MESSAGE_NOTIFICATION = 8;       // 服务器通知
}

}
```

2. 原则二：使用一个Protobuf消息结构定义一类消息

例如，对应于登录请求（LOGIN_REQUEST）类型的消息，其消息结构如下：

```
/*登录请求信息*/
message LoginRequest {
    string uid = 1;           // 用户唯一ID
    string deviceId = 2;      // 设备ID
    string token = 3;          // 用户token
    uint32 platform = 4;       // 客户端平台 Windows、MAC、
                               // Android、IOS、Web
    string appVersion = 5;     // APP版本号
}

}
```

3. 原则三：建议给应答消息加上成功标记和应答序号

应答消息并非总是成功的，因此建议在应答消息中加上两个字段：成功标记和应答序号。成功标记是一个用于描述应答是否成功的标记，建议使用bool类型，true表示发送成功，false表示发送失败。

另外，建议设置info字段的类型为字符串，用于放置失败时的提示信息。

应答序号的作用是什么？如果一个请求有多个响应，则发送端可以设计为每个响应消息可以包含一个应答的序号，最后一个响应消息包含一个结束标记。接收端在处理时，根据应答序号和结束标记可以合并所有的响应消息。

对应于聊天响应（MESSAGE_RESPONSE）类型的消息，其消息结构可以设计如下：

```
/*聊天响应*/  
  
message MessageResponse {  
    bool result = 1;           //true表示发送成功, false表示发送失败  
    uint32 code = 2;           //错误码  
    string info = 3;           //错误描述  
    uint32 expose = 4;         //错误描述是否提示给用户:1 提示; 0 不提示  
    bool lastBlock = 5;         //是否为最后的应答  
    fixed32 blockIndex = 6;     //应答的序号  
}
```

4. 原则四：编解码从顶层消息开始

建议定义一个外层的消息，把所有的消息类型全部封装在一起。在通信时可以从外层消息开始编码或者解码。对于聊天器中的外层消息，外层的消息结构可以定义如下：

```
/*外层消息*/  
  
message Message {  
    HeadType type = 1;                                //消息类型  
    uint64 sequence = 2;                               //序列号  
    string sessionId = 3;                             //会话ID  
    LoginRequest loginRequest = 4;                   //登录请求  
    LoginResponse loginResponse = 5;                 //登录响应  
    MessageRequest messageRequest = 6;                //聊天请求  
    MessageResponse messageResponse = 7;              //聊天响应  
    MessageNotification notification = 8;            //通知消息  
}
```

序列号主要用于请求数据包和响应数据包的配套，响应包中的序列号必须和请求包的序列号相同，使得发送端可以进行“请求-响应”的匹配处理。

完整的聊天器的proto协议文件，在源代码工程中所处的路径为chatcommon\proto\protoConfig\ProtoMsg.proto。

大家可以打开源码工程，自行阅读以上proto通信协议文件，并且可以使用Maven插件尝试生成对应的Protobuf Builder和POJO类，以供后续使用。

8.2 IM的登录流程

单体IM系统首先需要登录。从端到端（End to End）的角度来说，登录的流程包括以下环节：

- (1) 客户端发送登录数据包。
- (2) 服务端进行用户信息验证。
- (3) 服务端创建会话。
- (4) 服务端返回登录结果的信息给客户端，包括成功标志、Session ID等。

整个端到端（End to End）的登录流程涉及4次编码/解码：

- (1) 客户端编码：客户端对登录请求的Protobuf数据包进行编码。
- (2) 服务端解码：服务端对登录请求的Protobuf数据包进行解码。
- (3) 服务端编码：服务端对编码登录响应的Protobuf数据包进行编码。
- (4) 客户端解码：客户端对登录响应的Protobuf数据包进行解码。

8.2.1 图解登录/响应流程的环节

从细分的角度来说，整个登录/响应的流程大概包含9个环节，如图8-2所示。

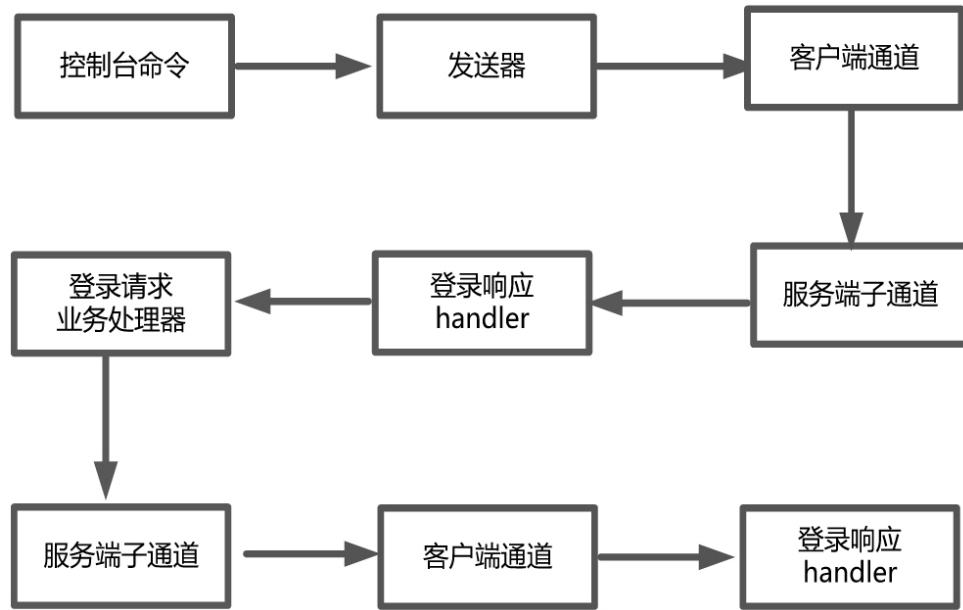


图8-2 登录/响应的流程

从客户端到服务端再到客户端，9个环节的相关介绍如下：

- (1) 客户端收集用户ID和密码，需要使用LoginConsoleCommand控制台命令类。
- (2) 客户端发送Protobuf数据包到客户端通道，需要通过LoginSender发送器组装Protobuf数据包。
- (3) 客户端通道将Protobuf数据包发送到对端，需要通过Netty底层来完成。
- (4) 服务器子通道收到Protobuf数据包，需要通过Netty底层来完成。

(5) 服务端UserLoginHandler入站处理器收到登录消息，交给业务处理器LoginMsgProcesser处理异步的业务逻辑。

(6) 服务端LoginMsgProcesser处理完异步的业务逻辑，将处理结果写入用户绑定的子通道。

(7) 服务器子通道将登录响应Protobuf数据帧发送到客户端，需要通过Netty底层来完成。

(8) 客户端通道收到Protobuf登录响应数据包，需要通过Netty底层来完成。

(9) 客户端LoginResponseHandler业务处理器处理登录响应，例如设置登录的状态、保存会话的Session ID等。

8.2.2 客户端涉及的主要模块

在IM登录的整体执行流程中，客户端所涉及的主要模块大致如下：

(1) ClientCommand模块：控制台命令收集器。

(2) ProtobufBuilder模块：Protobuf数据包构造者。

(3) Sender模块：数据包发送器。

(4) Handler模块：服务器响应处理器。

上面的这些模块都有一个或者多个专门的POJO Java类来完成对应的工作：

(1) LoginConsoleCommand类：属于ClientCommand模块，负责收集用户在控制台输入的用户ID和密码。

(2) CommandController类：属于ClientCommand模块，负责收集用户在控制台输入的命令类型，根据相应的类型调用相应的命令处理器，然后收集相应的信息。例如，如果用户输入的命令类型为登录，则调用LoginConsoleCommand命令处理器将收集到的用户ID和密码封装成User类，然后启动登录处理。

(3) LoginMsgBuilder类：属于ProtobufBuilder模块，负责将User类组装成Protobuf登录请求数据包。

(4) LoginSender类：属于Sender模块，负责将组装好的Protobuf登录数据包发送到服务端。

(5) LoginResponseHandler类：属于Handler模块，负责处理服务端的登录响应。

8.2.3 服务端涉及的主要模块

在IM登录的整体执行流程中，服务端涉及的主要模块如下：

(1) Handler模块：客户端请求的处理。

(2) Processer模块：以异步方式完成请求的业务逻辑处理。

(3) Session模块：管理用户与通道的绑定关系。

在具体的服务器登录流程中，上面的这些模块都有一个或者多个专门的Java类来完成对应的工作，大致的类为：

(1) UserLoginRequestHandler类：属于Handler模块，负责处理收到的Protobuf登录请求包，然后使用LoginProcesser类以异步方式进行用户校验。

(2) LoginProcesser类：属于Processer模块，完成服务端的用户校验，再将校验的结果组装成一个登录响应Protobuf数据包写回到客户端。

(3) ServerSession类：属于Session模块，如果校验成功，设置相应的会话状态；然后，将会话加入服务端的SessionMap映射中，这样该用户就可以接受其他用户发送的聊天消息了。

问题：为什么在服务端登录处理需要分成两个模块（一个模块是Handler业务处理器，另一个模块是Processer以异步方式完成请求的业务逻辑处理器），而不是像客户端一样在Netty的Handler入站处理器模块中统一完成业务的处理逻辑呢？具体答案，在8.4.4小节揭晓。

8.3 客户端的登录处理的实战案例

在输入登录信息之前，用户所选择的菜单是登录的选项。最开始的时候，客户端通过ClientCommandMenu菜单展示类展示出一个命令菜单，以供用户选择。效果如下：

```
//...  
INFO (NettyClient.java:102) - 客户端开始连接 [疯狂创客圈IM]  
INFO (CommandController.java:95) - 疯狂创客圈 IM 服务器连接成功!
```

请输入某个操作指令：

```
[menu] 0->查看所有命令 | 1->登录 | 2->聊天 | 10->退出 |
```

从上面的输出可以看出，ClientCommandMenu菜单展示类打印了4个选项：

(1) 登录， (2) 聊天， (3) 退出， (4) 查看全部命令

每个菜单选项都对应到一个信息的收集类：

(1) 聊天命令的信息收集类：ChatConsoleCommand。

(2) 登录命令的信息收集类：LoginConsoleCommand。

(3) 退出命令的信息收集类：LogoutConsoleCommand。

(4) 命令的类型收集类：ClientCommandMenu。

以上4个客户端命令的收集类都组合在CommandClient类中，CommandClient类代表了整个客户端。当用户输入的命令为“1”（表示登录）时，CommandClient类会找到与命令“1”对应的登录命令收集类LoginConsoleCommand去完成用户ID和密码的收集。

8.3.1 LoginConsoleCommand和User POJO

登录命令收集类LoginConsoleCommand负责从控制台收集客户端输入的用户ID和密码，代码如下：

```
package com.crazymakercircle.imClient.clientCommand;  
//...  
  
public class LoginConsoleCommand implements BaseCommand {  
  
    public static final String KEY = "1";  
  
    private String userName; //简单起见，假设用户名和id一致  
    private String password; //登录密码  
  
    @Override  
  
    public void exec(Scanner scanner) {  
  
        System.out.println("请输入用户信息(id:password) ");  
  
        String[] info = null;  
  
        while (true) {  
  
            String input = scanner.next();  
  
            info = input.split(":");  
  
            if (info.length != 2) {  
  
                System.out.println("请按照格式输入  
(id:password):");  
  
            } else {  
  
                //处理逻辑  
            }  
        }  
    }  
}
```

```
        break;  
    }  
}  
  
userName=info[0];  
password = info[1];  
  
}  
  
//...  
}
```

成功获取到用户密码和ID获取后，客户端CommandClient将这些内容组装成User POJO用户对象，然后通过客户端登录消息发送器loginSender开始向服务端发送登录请求，主要代码如下：

```
package com.crazymakercircle.imClient.client;  
  
//...  
  
@Service("CommandClient")  
  
public class CommandClient {  
  
    //...  
  
    //命令收集线程  
  
    public void startCommandThread() throws  
InterruptedException {  
  
        Thread.currentThread().setName("主线程");  
  
        while (true) {  
  
            //建立连接  
  
            while (connectFlag == false) {  
  
                //开始连接  
  
                startConnectServer();
```

```
        waitCommandThread();

    }

    //处理命令

    while (null != session && session.isConnected()) {

        Scanner scanner = new Scanner(System.in);

        clientCommandMenu.exec(scanner);

        String key =

clientCommandMenu.getCommandInput();

        //取到命令收集类POJO

        BaseCommand command = commandMap.get(key);

        switch (key) {

            //登录的命令

            case LoginConsoleCommand.KEY:

                command.exec(scanner); //收集用户名和

password

                startLogin((LoginConsoleCommand)

command);

                break;

            case... //省略其他的命令收集代码

        }

    }

}

//开始发送登录请求

private void startLogin(LoginConsoleCommand command) {

    //...
```

```
        User user = new User();
        user.setUid(command.getUserName());
        user.setToken(command.getPassword());
        user.setDevId("1111");
        loginSender.setUser(user);
        loginSender.setSession(session);
        loginSender.sendLoginMsg();
    }
//...
}
```

8.3.2 LoginSender

LoginSender消息发送器的sendLoginMsg()方法主要有两步：第一步生成Protobuf登录数据包，第二步调用BaseSender基类的sendMsg()方法来发送数据包。

```
package com.crazymakercircle.imClient.sender;
//...
@Slf4j
@Service("LoginSender")
public class LoginSender extends BaseSender {
    public void sendLoginMsg() {
        log.info("生成登录消息");
        ProtoMsg.Message message =
            LoginMsgBuilder.buildLoginMsg(getUser(),
                getSession());
    }
}
```

```
    log.info ("发送登录消息");
    super.sendMsg(message);
}
}
```

在以上代码中，使用LoginMsgBuilder构造者构造一个登录请求的Protobuf消息。这一步比较简单，大家直接看源代码即可。然后调用基类的sendMsg()方法来发送登录消息，BaseSender基类的代码如下：

```
package com.crazymakercircle.imClient.sender;
//...

public abstract class BaseSender {
    private User user;
    private ClientSession session;
    //...
    public void sendMsg(ProtoMsg.Message message) {
        if (null == getSession() || !isConnected()) {
            log.info("连接还没成功");
            return;
        }
        Channel channel=getSession().getChannel();
        ChannelFuture f = channel.writeAndFlush(message);
        f.addListener(new GenericFutureListener<Future<? super
Void>>() {
            @Override
            public void operationComplete(Future<? super Void>
future)
```

```
        ...
        //回调

        if (future.isSuccess()) {
            sendSucceed(message);
        } else {
            sendFailed(message);
        }
    }

} );
//...
}

protected void sendSucceed(ProtoMsg.Message message) {
    log.info("发送成功");
}

protected void sendFailed(ProtoMsg.Message message) {
    log.info("发送失败");
}

}
```

一般来说，在Netty中会调用write(pkg)或者writeAndFlush(pkg)方法来发送数据包，前面多次反复讲到，发送方法调用后会立即返回，返回的类型是一个ChannelFuture异步任务实例。问题是：发送方法返回时，数据包是否已经发送到对端？答案是没有，比如在write(pkg)方法返回时，真正的TCP写入的操作其实还没有执行。这和Netty中在同一个通道上的同一个处理器的出入站操作的串行执行特点有关。

在Netty中，无论是入站操作还是出站操作，都有两大特点：

(1) 同一条通道的同一个Handler处理器的所有出/入站处理都是串行的，而不是并行的。Netty是如何保障这一点的呢？在某个出/入站开启时，Netty会对当前的执行线程进行判断：如果当前线程不是Handler的执行线程，则处理暂时不执行，Netty会为当前处理建立一个新的异步可执行任务，加入Handler的执行线程任务队列中。

在处理加入通道时，可以为处理器设置一个单独的处理器线程，大致代码如下：

```
//创建一个独立的线程池，假定有32条线程
EventExecutorGroup threadGroup = new
DefaultEventExecutorGroup(32);

final OutHandlerDemo handlerA = new OutHandlerDemo(); //创建处
理器

ChannelInitializer i = new
ChannelInitializer<EmbeddedChannel>() {
    protected void initChannel(EmbeddedChannel ch) {
        //handlerA的执行，从threadGroup池中绑定一条线程
        ch.pipeline().addLast(threadGroup, handler);
    }
};
```

在处理器加入通道时，如果为处理器设置独立的线程组(EventExecutorGroup) A，那么处理器会被绑定到该组的一个特定线程Executor B，并且Netty会保证后续该通道有其他处理器也使用线程组A时，这些处理器会绑定到同一个特定线程Executor A上。

如果通道上所有的处理器都没有设置线程组，则所有的出入站处理任务都在通道的反应器线程上执行，这些任务都一个接一个地被串行处理。Netty的线程（Executor）维护了一个任务队列，对所有的处理任务进行排队。

Netty Executor线程的任务队列是一个MPSC队列（多生产者单消费者队列）。MPSC队列的特点是：只有EventLoop线程自己是唯一的消费者，它将遍历任务队列，逐个执行任务；其他线程只能作为生产者，它们的出/入站操作都会作为异步任务加入任务队列。

通过MPSC队列，EventLoop线程能做到确保同一个通道上所有的出/入站处理都是串行的，这样不同的Handler业务处理器之间不需要进行线程的同步，能大大提升IO的性能。如果在通道加入处理器时为处理器配置了专用的线程组，则可以保证属于同组的所有出/入站处理都是串行的。

(2) Netty的出/入站操作不是单个Handler业务处理器操作，而是流水线上一系列的出/入站处理流程。只有整个流程都处理完，出/入站操作才真正处理完成。

基于以上两点，大家可以简单地推断，在调用完channel.writeAndFlush(pkg)后，真正的出站操作肯定是没有执行完成的，可能还需要在EventLoop的任务队列中排队等待。

如何才能判断writeAndFlush()执行完毕了呢？writeAndFlush()方法会返回一个ChannelFuture异步任务实例，可以通过为其增加GenericFutureListener监听器的方式来判断writeAndFlush()是否已经执行完毕。当监听器的operationComplete方法被回调时，表示

`writeAndFlush()`方法已经执行完毕了。具体的回调业务逻辑可以放在`operationComplete`回调方法中。

在上面的代码中，设计了两个`sendSucceed()` / `sendfailed()`业务回调方法，在发送操作被真正执行完成后，回调方法将被执行，并且将`sendSucceed()`和`sendfailed()`方法封装在发送器的`BaseSender`基类中，方便子类发送器进行继承。子类发送器需要改变默认的回调处理逻辑时，可以重写`sendSucceed()`和`sendfailed()`方法。

在上面的代码中，为获取客户端的通道，使用了`ClientSession`客户端会话，那么什么是会话、会话的作用是什么、什么时候创建会话呢？下一小节为大家解答。

8.3.3 ClientSession

`ClientSession`是一个很重要的胶水类，包含两个成员：一个是`user`，代表用户；另一个是`channel`，代表连接的通道。在实际开发中，这两个成员的作用是：

- (1) 通过`user`，`ClientSession`可以获得当前的用户信息。
- (2) 通过`channel`，`ClientSession`可以向服务端发送消息。

`ClientSession`会话“左拥右抱”，左手“拥有”用户消息，右手“抱有”服务端的连接。其通过`user`成员可以获取当前的用户信息，借助`Channel`（通道）可以写入`Protobuf`数据包到对端，或者关闭`Netty`连接。

客户端会话`ClientSession`保存着当前的状态：

(1) 是否成功连接isConnected。

(2) 是否成功登录isLogin。

ClientSession绑定在Channel上，因而可以在入站处理时通过Channel反向取得绑定的ClientSession，从而对应到user信息。这一点非常重要！在疯狂创客圈社群中，总是有人问，“如何将Channel与用户对应呢？”其答案就在于ClientSession与Channel的双向绑定关系上，通过Channel可以找到绑定的ClientSession，进一步找到要对应的用户，从而实现Channel与用户的对应关系。

ClientSession客户端会话的主要代码如下：

```
package com.crazymakercircle.imClient.client;  
//...  
  
public class ClientSession {  
  
    public static final AttributeKey<ClientSession> SESSION_KEY  
    =  
        AttributeKey.valueOf("SESSION_KEY");  
  
  
    private Channel channel;  
    private User user;  
    private String sessionId; //保存登录后的服务端sessionid  
    private Boolean isConnected = false;  
    private Boolean isLogin = false;  
  
    //绑定通道  
    public ClientSession(Channel channel) {
```

```
        this.channel = channel;
        this.sessionId = String.valueOf(-1);
            //重要：ClientSession绑定到Channel上
        channel.attr(ClientSession.SESSION_KEY).set(this);
    }

//登录成功之后，设置sessionId
public static void loginSuccess(
    ChannelHandlerContext ctx, ProtoMsg.Message pkg) {
    Channel channel = ctx.channel();
    ClientSession session =
        channel.attr(ClientSession.SESSION_KEY).get();
    session.setSessionId(pkg.getSessionId());
    session.setLogin(true);
    log.info("登录成功");
}

//获取通道
public static ClientSession
getSession(ChannelHandlerContext ctx) {
    Channel channel = ctx.channel();
    ClientSession session =
        channel.attr(ClientSession.SESSION_KEY).get();
    return session;
}
```

```
//把Protobuf数据包写入通道

public ChannelFuture writeAndFlush(Object pkg) {
    ChannelFuture f = channel.writeAndFlush(pkg);
    return f;
}

//...
}
```

什么时候创建客户端会话呢？在Netty客户端发起连接请求之后，增加一个连接建立完成的异步回调任务，代码如下：

```
package com.crazymakercircle.imClient.client;
//...

public class CommandController {
    //...
    GenericFutureListener<ChannelFuture> connectedListener =
        (ChannelFuture f) -> {
        final EventLoop eventLoop = f.channel().eventLoop();
        if (!f.isSuccess()) {
            log.info("连接失败！在10秒之后准备尝试重连！");
            eventLoop.schedule(() -> nettyClient.doConnect(),
                10, TimeUnit.SECONDS);
        }
        connectFlag = false;
    } else {
        connectFlag = true;
        log.info("疯狂创客圈 IM 服务器连接成功！");
    }
}
```

```
        channel = f.channel();
        //创建会话
        session= new ClientSession(channel);
        channel.closeFuture().addListener(closeListener);
        //唤醒用户线程
        notifyCommandThread();
    }
};

//...
}
```

8.3.4 LoginResponseHandler

LoginResponseHandler登录响应处理器对消息类型进行判断：

(1) 如果消息类型是请求响应消息并且登录成功，则取出绑定的会话（Session），再设置登录成功的状态。完成登录成功处理之后，进行其他的客户端业务处理。

(2) 如果消息类型不是请求响应消息，则调用父类默认的super.channelRead()入站处理方法，将数据包交给流水线的下一站Handler业务处理器去处理。

```
package com.crazymakercircle.imClient.handler;
//...
public class LoginResponseHandler
    extends ChannelInboundHandlerAdapter {
```

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object
msg)
{
    ...
    //判断消息实例
    if (null == msg || !(msg instanceof ProtoMsg.Message)) {
        super.channelRead(ctx, msg);
        return;
    }

    //判断类型
    ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
    ProtoMsg.HeadType headType = ((ProtoMsg.Message)
msg).getType();
    if (!headType.equals(ProtoMsg.HeadType.LOGIN_RESPONSE))
    {
        super.channelRead(ctx, msg);
        return;
    }

    //判断返回是否成功
    ProtoMsg.LoginResponse info = pkg.getLoginResponse();
    ProtoInstant.ResultCodeEnum result =
        ProtoInstant.ResultCodeEnum.values()
[info.getCode()];
    if
(!result.equals(ProtoInstant.ResultCodeEnum.SUCCESS)) {
        //登录失败
    }
}
```

```
        log.info(result.getDesc());  
    } else {  
        //登录成功  
        ClientSession.loginSuccess(ctx, pkg);  
        ChannelPipeline p = ctx.pipeline();  
        //移除登录响应处理器  
        p.remove(this);  
  
        //在编码器后面动态插入心跳处理器  
        p.addAfter("encoder", "heartbeat",  
                   new  
HeartBeatClientHandler());  
    }  
}
```

在登录成功之后，需要将LoginResponseHandler登录响应处理器实例从流水线上移除，因为不需要再处理登录响应了。同时，需要在客户端和服务端（即服务器器）之间开启定时的心跳处理。心跳是一个比较复杂的议题，后面会专门详细介绍客户端和服务器之间的心跳。

8.3.5 客户端流水线的装配

在客户端的业务处理器流水线（Pipeline）上，首先需要装配一个ProtobufDecoder解码器和一个ProtobufEncoder编码器，编码器和

解码器一般都是装配在最前面。然后需要装配业务处理器——LoginResponseHandler（登录响应处理器）。

一般来说，在流水线最后还需要装配一个异常处理器（ExceptionHandler）。它也是一个入站处理器，用来实现Netty异常的处理以及在连接异常中断后进行重连。

```
package com.crazymakercircle.imClient.client;  
//省略部分代码  
  
public class NettyClient {  
  
    @Autowired  
    private ChatMsgHandler chatMsgHandler;  
    //聊天消息处理器  
  
    @Autowired  
    private LoginResponseHandler loginResponseHandler;  
    //登录响应处理器  
    //连接异步监听  
    private GenericFutureListener<ChannelFuture>  
connectedListener;  
  
    private Bootstrap b;  
    private EventLoopGroup g;  
    //省略部分代码  
  
    public void doConnect() {  
        try {  
            b = new Bootstrap();
```

```
//省略设置通道初始化参数

b.handler(new ChannelInitializer<SocketChannel>() {
    public void initChannel(SocketChannel ch) {
        ch.pipeline().addLast("decoder",
            new
        ProtobufDecoder());
        ch.pipeline().addLast("encoder",
            new
        ProtobufEncoder());
        ch.pipeline().addLast(loginResponseHandler);
        ch.pipeline().addLast(chatMsgHandler);
        ch.pipeline().addLast("exception",
            new
        ExceptionHandler());
    }
}) ;

log.info("客户端开始连接 [疯狂创客圈IM]");

ChannelFuture f = b.connect();
f.addListener(connectedListener);

} catch (Exception e) {
    log.info("客户端连接失败!" + e.getMessage());
}

//...
}
```

处理器装配次序说明：登录响应处理器必须装配在
ProtobufDecoder解码器之后。其具体的原因是：Netty客户端读到二
进制ByteBuf数据包之后，需要通过ProtobufDecoder完成解码操作。
解码后组装好Protobuf消息POJO，再进入LoginResponseHandler。

8.4 服务端的登录响应的实战案例

服务端的登录处理流程是：

(1) ProtobufDecoder解码器把请求ByteBuf数据包解码成Protobuf数据包。

(2) UserLoginRequestHandler登录处理器负责处理Protobuf数据包，进行一些必要的判断和预处理后，启动LoginProcesser登录业务处理器，以异步方式进行登录验证处理。

(3) LoginProcesser通过数据库或者远程接口完成用户验证，根据验证处理的结果生成登录成功/失败的登录响应报文，并发送给客户端。

8.4.1 服务端流水线的装配

与客户端类似，服务端流水线首先需要装配一个ProtobufDecoder解码器和一个ProtobufEncoder编码器，然后需要装配loginRequestHandler登录业务处理器实例。最后，在流水线上加入一个serverExceptionHandler异常处理器实例。

```
package com.crazymakercircle.imServer.server;  
//...  
public class ChatServer {  
    //...  
    @Autowired
```

```
private LoginRequestHandler loginRequestHandler;
//登录请求处理器

@Autowired

private ServerExceptionHandler serverExceptionHandler;
//服务器异常处理器

public void run() {
    try {
        //省略Bootstrap的配置选项
        //5 装配流水线
        b.childHandler(new
ChannelInitializer<SocketChannel>() {
        //有连接到达时会创建一个子通道
        protected void
initChannel(SocketChannel ch) ...{
        //装配子通道流水线中的
Handler业务处理器

ch.pipeline().addLast(new ProtobufDecoder()); //解码器

ch.pipeline().addLast(new ProtobufEncoder()); //编码器
        //在流水线中添加登录处理
器，登录后删除

ch.pipeline().addLast(loginRequestHandler);

ch.pipeline().addLast(serverExceptionHandler); //异常处理器
    }
}
```

```
        });

        //省略启动Bootstrap

    } catch (Exception e) {

        e.printStackTrace();

    } finally {

        //8 优雅关闭EventLoopGroup,
        //释放掉所有资源，包括创建的线程

        wg.shutdownGracefully();

        bg.shutdownGracefully();

    }

}

}
```

在服务端的登录处理流程中，ProtobufDecoder解码器把登录请求的二进制ByteBuf数据包解码成Protobuf数据包，然后发送给下一站LoginRequestHandler，由该处理器异步发起实际的登录处理。

8.4.2 LoginRequestHandler

这是一个入站处理器，继承自ChannelInboundHandlerAdapter入站适配器，重写了适配器的channelRead()方法，主要工作如下：

(1) 对消息进行必要的判断：判断是否为登录请求Protobuf数据包。如果不是，通过super.channelRead(ctx, msg)将消息交给流水线的下一个入站处理器。

(2) 如果是登录请求Protobuf数据包，准备进行登录处理，提前为客户建立一个服务端的会话ServerSession。

(3) 使用自定义的CallbackTaskScheduler异步任务调度器提交一个异步任务，启动LoginProcesser执行登录用户验证逻辑。

```
package com.crazymakercircle.imServer.handler;  
//...  
@Slf4j  
@Service("LoginRequestHandler")  
@ChannelHandler.Sharable  
public class LoginRequestHandler extends  
ChannelInboundHandlerAdapter {  
    @Autowired  
    LoginProcesser loginProcesser;  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) ...{  
        if (null == msg || !(msg instanceof ProtoMsg.Message)) {  
            super.channelRead(ctx, msg);  
            return;  
        }  
        ProtoMsg.Message pkg = (ProtoMsg.Message) msg;  
        //取得请求类型  
        ProtoMsg.HeadType headType = pkg.getType();  
        if (!headType.equals(loginProcesser.type())) {  
            super.channelRead(ctx, msg);  
            return;  
        }  
        loginProcesser.login(pkg);  
    }  
}
```

```
}

ServerSession session = new
ServerSession(ctx.channel());
    //异步任务，处理登录的逻辑
CallbackTaskScheduler.add(new CallbackTask<Boolean>() {
    @Override
    public Boolean execute()...{
        boolean r = loginProcesser.action(session,
pkg);
        return r;
    }
    //异步任务返回
    @Override
    public void onBack(Boolean r) {
        if (r) {
ctx.pipeline().remove(LoginRequestHandler.this);
            log.info("登录成功:" + session.getUser());
        } else {
            ServerSession.closeSession(ctx);
            log.info("登录失败:" + session.getUser());
        }
    }
    //异步任务异常
    @Override
    public void onException(Throwable t) {
        ServerSession.closeSession(ctx);
    }
}
```

```
        log.info("登录失败：" + session.getUser());
    }
}
}
}
```

8.4.3 LoginProcesser

LoginProcesser用户验证逻辑主要包括密码验证、将验证的结果写入通道。如果登录验证成功，还需要实现通道与服务端会话的双向绑定，并且将服务端会话加入到在线用户列表中。

```
package com.crazymakercircle.imServer.processor;
//...
@Slf4j
@Service("LoginProcesser")
public class LoginProcesser extends AbstractServerProcessor {
    @Autowired
    LoginResponseBuilder loginResponseBuilder;
    @Override
    public ProtoMsg.HeadType type() {
        return ProtoMsg.HeadType.LOGIN_REQUEST;
    }
    @Override
    public boolean action(ServerSession
session, ProtoMsg.Message proto) {
```

```
//取出token验证
ProtoMsg.LoginRequest info = proto.getLoginRequest();
long seqNo = proto.getSequence();

User user = User.fromMsg(info);
//检查用户
booleanisValidUser = checkUser(user);
if (!isValidUser) {
    ProtoInstant.ResultCodeEnum resultcode =
        ProtoInstant.ResultCodeEnum.NO_TOKEN;
    //生成登录失败的报文
    ProtoMsg.Message response =
        loginResponseBuilder.loginResponse(resultcode,
seqNo, "-1");
    //发送登录失败的报文
    session.writeAndFlush(response);
    return false;
}

session.setUser(user);
session.bind();

//登录成功
ProtoInstant.ResultCodeEnum resultcode =
    ProtoInstant.ResultCodeEnum.SUCCESS;
//生成登录成功的报文
ProtoMsg.Message response = loginResponseBuilder.
```

```

        loginResponse(resultcode, seqNo,
session.getSessionId());
//发送登录成功的报文
session.writeAndFlush(response);

return true;
}

private boolean checkUser(User user) {
if (SessionMap.inst().hasLogin(user)) {
return false;
}
//验证用户，比较耗时的操作，需要200毫秒以上的时间甚至更多
//方法1：调用远程用户RESTful校验服务
//方法2：调用数据库接口校验
return true;
}
}

```

用户密码验证的逻辑在checkUser()方法中完成。在实际的生产场景中，LoginProcesser进行用户登录验证的方式比较多：

- 通过**RESTful**接口验证用户。
- 通过数据库去验证用户。
- 通过认证（Auth）服务器去验证用户。

总之，验证用户涉及RPC等耗时操作，为了尽量简化流程，示例程序代码省去了通过账号和密码验证的过程，checkUser()方法直接返回true，也就是默认所有的登录都是成功的。

服务端校验通过之后，可以完成服务端会话（ServerSession）的绑定工作。服务端的ServerSession会话与客户端的ClientSession会话类似，也是一个胶水类。每个ServerSession拥有一个Channel成员实例、一个User成员实例。Channel成员代表与客户端连接的子通道；User成员代表用户信息。稍后，会对ServerSession进行详细介绍。

在用户校验成功后，服务端就需要向客户端发送登录响应，具体的方法是：调用登录响应的Protobuf消息构造器loginResponseBuilder，构造一个登录响应POJO实例，设置好校验成功的标志位，调用会话（Session）的writeAndFlush()方法把数据写到客户端。

8.4.4 EventLoop线程和业务线程相互隔离

在前面的章节中已经埋下了一个疑问：为什么在服务端的登录处理需要分成两个模块，而不是像客户端一样在InboundHandler入站处理器中统一完成处理呢？答案是在服务端需要隔离EventLoop（Reactor）线程和业务线程。基本方法是使用独立、异步的业务线程去执行用户验证的逻辑，而不是在EventLoop线程中去执行用户验证的逻辑。

实际上，Reactor线程和业务线程相互隔离，在服务端非常重要。为什么呢？

首先，以读通道channelRead为例，普通的登录入站处理的基本步骤是：

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
    throws Exception {
    //1判断消息是否需要处理
    //2 取得消息并判断类型
    //3 耗时的业务处理操作
    //4 把结果写入连接通道
}
```

其中的第三步通常会涉及一些比较耗时的业务处理操作，例如：

- (1) 如果是数据库操作，一般查询的耗时在100ms以上，百毫秒级。
- (2) 如果是远程接口调用，一般耗时在200ms以上，百毫秒级，稍微慢点的耗时在500ms以上。

再看Netty内部的IO读写操作，通常都是毫秒级。也就是说，Netty内部的IO操作和业务处理操作在时间上不在一个数量级。

问题来了：在大量（成千上万）的子通道复用一个EventLoop反应器线程的应用场景中，一旦某个耗时的业务处理操作在执行，就会导致子通道上的其他IO操作发生严重的阻塞问题。这样会导致严重的性能问题，为什么呢？在默认情况下，Netty的一个EventLoopGroup反应器组会开启2倍CPU核数的内部线程。通常情况下，一个Netty服务端会有几万或者几十万的连接通道。也就是说，一个EventLoop组内线程会负责处理几万或者上十万个通道连接的IO处理。

在一个EventLoop内部线程上的任务是串行的。如果一个Handler业务处理器中的channelRead()入站处理方法执行1000ms（即1秒）或

者几秒，最终的结果是阻塞了EventLoop内部线程其他几十万个通道的出站和入站处理，阻塞时长为1秒或者几秒。耗时的入站/出站处理越多，越会拖慢整个线程的其他IO处理，最终导致严重的性能问题。

如果出现了严重的性能问题，解决方法是：业务操作和EventLoop线程相隔离。具体来说，就是专门开辟一个独立的线程池，负责一个独立的异步任务处理。对于耗时的业务操作封装成异步任务，并放入独立的线程池中去处理。这样的话服务端的性能会提升很多，避免了对IO操作的阻塞。

有两种办法使用独立的线程池：一是使用Netty的EventLoopGroup线程池，二是使用自己创建的Java线程池。

方法1：创建Netty的EventLoopGroup线程池，专用于处理耗时任务。

此方法有一个特点，在同一通道上的所有出入站处理（未设置的除外）都会绑定在线程池中的同一线程上，保障这些处理是串行执行的，不需要进行同步控制，使用的示例如下：

```
//创建一个独立的线程池，假定有32条线程

EventExecutorGroup threadGroup = new
DefaultEventExecutorGroup(32);

final OutHandlerDemo handlerA = new OutHandlerDemo(); //创建处
理器

ChannelInitializer i = new
ChannelInitializer<EmbeddedChannel>() {
```

```
protected void initChannel(EmbeddedChannel ch) {  
    //处理器加入通道时，从专用threadGroup池中绑定一条线程  
    ch.pipeline().addLast(threadGroup, handler);  
}  
};
```

Netty Executor线程的任务队列是一个MPSC队列（多生产者单消费者队列）。MPSC队列的特点是：只有EventLoop线程自己是唯一的消费者，它将遍历任务队列，逐个执行任务；其他线程只能作为生产者，它们的出/入站操作都会作为异步任务加入到任务队列。通过MPSC队列，EventLoop线程能做到确保同一个通道上所有的出/入站处理都是串行的，不是并行的，这样不同的Handler业务处理器之间不需要进行线程的同步，这点也能节省线程之间同步的时间。

方法2：创建一个专门的Java线程池，专用于处理耗时任务。

可以写一个专门的辅助类，帮助线程池的创建和任务的提交，大致代码如下：

```
package com.crazymakercircle.concurrent;  
//...  
public class FutureTaskScheduler extends Thread  
{  
    //方法二是使用自建的线程池时专用于处理耗时操作  
    private static final ExecutorService POOL=  
        Executors.newFixedThreadPool(10);  
    //添加耗时任务
```

```
public static void add(Runnable executeTask)
{
    POOL.submit(executeTask);
}

}
```

提交任务时，使用辅助类的静态方法add(Runnable executeTask)添加耗时操作即可。不过以上的add()方法所添加的是没有回调处理的任务，如果需要添加有回调处理的任务，则可以自己增加一个类似的辅助函数。

说明

出于降低学习难度的目的，以上辅助类调用Executors.newFixedThreadPool(10)快捷方式创建了一个容量为10的固定大小线程池。注意，生产环境是不允许使用Executors快捷方式来创建线程池的，具体的原理请参阅本系列的《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》。这也是为什么这里使用辅助类，而不直接使用线程池的原因。如果需要修改和升级，那么优化一下辅助类FutureTaskScheduler即可，不需要去修改那些提交异步任务的代码，可以说升级的工作量很小。

8.5 详解Session服务器会话

无论是客户端还是服务端，为了让通道（Channel）连接和用户（User）状态的管理和使用变得方便，都使用了一个非常重要的概念——会话（Session）。有点儿类似Tomcat的服务器会话，只是在实现上比较简单。

由于客户端和服务器分别都有各自的通道，并且相关的参数有些也不一致，因此这里使用了两个会话类型：客户端会话ClientSession与服务端会话ServerSession。

会话和通道之间存在两个方向的导航关系：一个是正向导航，可以通过会话导航到通道，主要用于出站处理的场景，例如需要将数据包写出到通道；另一个是反向导航，可以通过通道导航到会话，主要用于入站处理的场景，入站时可以从通道获取绑定的会话，以便进一步进行业务处理。

如果进行反向导航呢？需要用到通道的容器属性。

8.5.1 通道的容器属性

Netty中的Channel通道类有类似于Map的容器功能，可以通过键-值对（Key-Value Pair）的形式来保存任何Java Object实例。一般来说，可以用于存放一些与通道相关联的属性，比如说会话实例。另外，除了Channel通道实例，Netty中的HandlerContext处理器上下文

实例也具备了类似的容器功能，可以绑定键-值对。那么，Channel和HandlerContext的容器功能具体是如何实现的呢？

Netty没有实现Map接口，而是定义了一个类似的接口，叫作AttributeMap（原理见图8-3）。它有且只有一个方法“`<T> Attribute<T> attr(AttributeKey<T> key);`”，此方法接收一个AttributeKey类型的键（Key），返回一个Attribute类型的值（Value），特点如下：

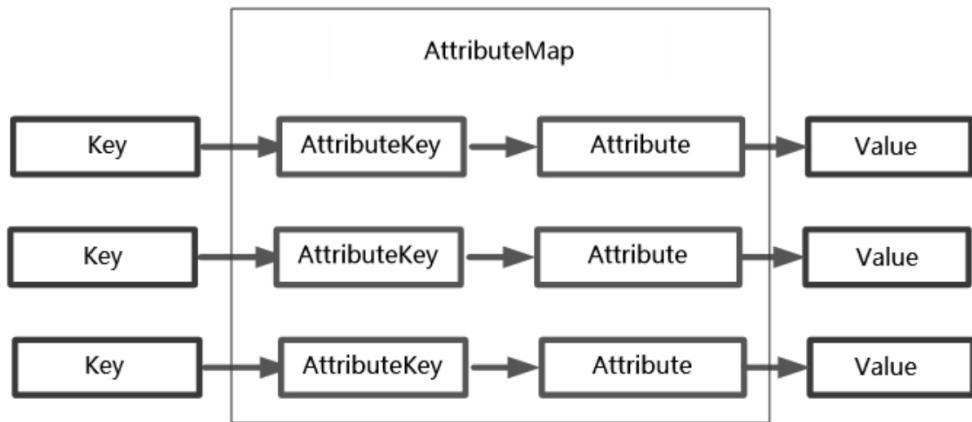


图8-3 AttributeMap原理图

(1) AttributeKey不是原始的键（如Map中的键），而是一个键的包装类。AttributeKey确保了键的唯一性，在单个Netty应用中，AttributeKey必须唯一。

(2) 这里的Attribute值不是原始的值（如Map中的值），也是值的包装类。原始的值就放置在Attribute包装实例中，可以通过Attribute包装类实现值的读取（get）和设置（set）。

在Netty中，接口AttributeMap的源代码如下：

```
package io.netty.util;  
  
public interface AttributeMap {  
  
    <T> Attribute<T> attr(AttributeKey<T> key);  
  
}
```

AttributeMap只是一个接口，Netty提供了默认的实现。AttributeMap的实现要求是线程安全的。可以先通过通道的attr()方法根据AttributeKey实例取得Attribute类型的value实例；然后通过Attribute类型的value实例完成最终的两个重要操作：设值（set）、取值（get）。

1. Attribute的设值

Attribute设值的方法举例如下：

```
//定义键  
  
public static final AttributeKey<ServerSession> SESSION_KEY =  
  
AttributeKey.valueOf("SESSION_KEY");  
  
//...  
  
//通过设置将会话绑定到通道  
  
channel.attr(SESSION_KEY).set(session);
```

AttributeKey的创建需要用到静态方法

AttributeKey.valueOf(String)方法。该方法的返回值为一个AttributeKey实例，其泛型参数为实际键-值对中值的实际类型。如果实际的值是ServerSession类型，则定义键的泛型参数为

ServerSession，整个AttributeKey定义为
AttributeKey<ServerSession>。

```
//键的泛型形参是设置的值类型  
public static final AttributeKey<ServerSession> SESSION_KEY =  
  
AttributeKey.valueOf("SESSION_KEY");
```

创建完AttributeKey后，就可以通过通道完成键-值对的设值（set）、取值（get）了。常常使用链式调用，首先通过通道的attr(AttributeKey)方法取得value的包装类Attribute实例，然后通过Attribute的set()方法设置真正的值。在例子中，值是一个会话（Session）实例。

```
//通过设置将会话绑定到通道  
channel.attr(SESSION_KEY).set(session);
```

说明

这里的AttributeKey一般定义为一个常量，需要提前定义；它的泛型参数是最终的Attribute的包装值value的数据类型。

2. Attribute取值

取值调用Attribute实例的get()方法。具体来说，首先通过通道的attr(AttributeKey)方法取得键（Key）所对应的Attribute包装实

例，然后通过Attribute的get()方法设置真正的值（Value）。举例如下：

```
//取得Attribute实例  
Attribute<ServerSession> attribute =  
ctx.channel().attr(SESSION_KEY);  
ServerSession session=attribute.get();
```

还可以使用链式调用，代码如下：

```
ServerSession session = ctx.channel().attr(SESSION_KEY).get();
```

8.5.2 ServerSession服务端会话类

在登录成功之后，服务端会为每一个新连接通道创建一个ServerSession实例，用于保持用户与服务端的会话信息。每个ServerSession实例都拥有一个唯一标识，为SessionId。注意SessionId不一定是UserId。主要是因为同一个用户可能从网页端、手机端、电脑桌面同时登录IM服务端，就像微信、QQ那样，此时同一个用户的消息需要在手机端、网页端、桌面端进行同步，各个终端需要能同时接收消息、同时发送消息。

```
package com.crazymakercircle.imServer.server;  
//...  
public class ServerSession {  
    public static final AttributeKey<ServerSession> SESSION_KEY  
    =  
        AttributeKey.valueOf("SESSION_KEY");
```

```
private Channel channel; //通道
private User user; //用户
private final String sessionId;//会话唯一标识

private boolean isLogin = false;//登录状态

public ServerSession(Channel channel) {
    this.channel = channel;
    this.sessionId = buildNewSessionId();
}

//反向导航

public static ServerSession
getSession(ChannelHandlerContext ctx) {
    Channel channel = ctx.channel();
    return channel.attr(ServerSession.SESSION_KEY).get();
}

//和通道实现双向绑定

public ServerSession bind() {
    log.info(" ServerSession绑定会话 " +
channel.remoteAddress());
    channel.attr(ServerSession.SESSION_KEY).set(this);
    SessionMap.inst().addSession(getSessionId(), this);
    isLogin = true;
    return this;
}

//构造session id

private static String buildNewSessionId() {
    String uuid = UUID.randomUUID().toString();
```

```
        return uuid.replaceAll("-", "");  
    }  
    //省略不太重要的方法  
}
```

从功能上说，ServerSession与ClientSession类似，也是一个很重要的胶水类：

(1) 通过ServerSession实例可以导航到Channel（通道）实例，以便发送消息。

(2) 在通道收到消息时，从通道能反向导航到ServerSession实例和用户，以便完成业务逻辑处理。

8.5.3 SessionMap会话管理器

一台服务器需要接受几万/几十万的客户端连接，每一条连接都对应到一个ServerSession实例，服务器需要对大量的ServerSession实例进行管理。这里使用一个会话容器SessionMap，负责管理服务端所有的ServerSession，其内部使用一个线程安全的ConcurrentHashMap类型的映射成员，保持sessionId到服务端ServerSession的映射。

```
package com.crazymakercircle.imServer.server;  
//...  
public final class SessionMap {  
    private ConcurrentHashMap<String, ServerSession> map =  
        new ConcurrentHashMap<String, ServerSession>();  
    //增加会话对象
```

```
public void addSession(String sessionId, ServerSession s) {  
    map.put(sessionId, s);  
    log.info("用户登录:id= " + s.getUser().getUid()  
            + " 在线总数: " + map.size());  
}  
  
//获取会话对象  
public ServerSession getSession(String sessionId) {  
    if (map.containsKey(sessionId)) {  
        return map.get(sessionId);  
    } else {  
        return null;  
    }  
}  
  
//省略不太重要的方法  
}
```

通过SessionMap可以实现在线用户的统计。除此之外，当用户与用户之间进行单聊时，服务端消息需要在不同的用户之间进行转发，这时也需要用到SessionMap。

8.6 点对点单聊的实战案例

单聊的业务非常简单，就像微信的文字聊天功能，主要业务流程大致如下：

(1) 当用户A登录成功之后，按照单聊的消息格式发送所要的消息。

为了简单，这里的消息格式简化为userId:content。其中，userId是消息接收方目标用户B的userId，content表示聊天的内容。

(2) 服务端收到消息后，根据目标userId进行消息的转发，发送到用户B所在的客户端。

(3) 客户端用户B收到用户A发来的消息，在自己的控制台显示出来。

这里有一个问题，为什么服务端的路由转发不是根据sessionID，而是根据userID呢？前面讲到，用户B可能登录了多个会话（桌面会话、移动端会话、网页端会话），这时发给用户B的聊天消息必须转发到多个会话，所以需要根据userID进行转发。

8.6.1 单聊的端到端流程

从大的角度来说，单聊的端到端流程包括以下环节（见图8-4）：

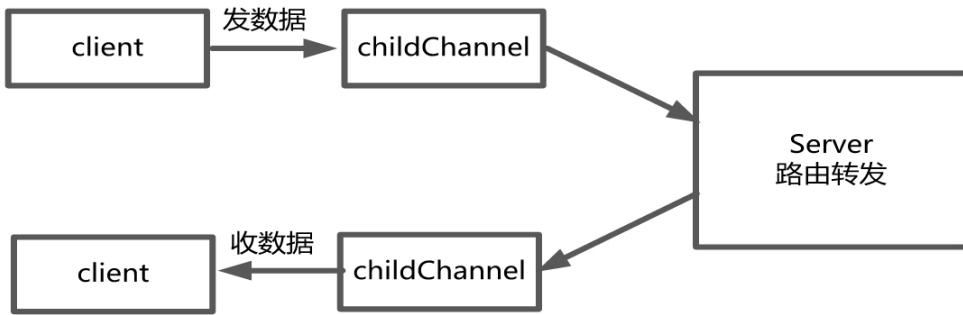


图8-4 单聊的端到端流程

- (1) 用户A发送单聊Protobuf数据包到服务端。
- (2) 服务端接收到用户A的单聊数据包。
- (3) 服务端转发单聊数据包到用户B。
- (4) 用户B接收到来自用户A的单聊数据包。

8.6.2 客户端的ChatConsoleCommand收集聊天内容

聊天消息收集类ChatConsoleCommand负责从控制台Scanner实例收集用户输入的聊天消息（格式为id:message），代码如下：

```

package com.crazymakercircle.imClient.command;

//...

@Data
@Service("ChatConsoleCommand")

public class ChatConsoleCommand implements BaseCommand {

    private String toUserId;      //目标用户id（这里为登录的用户名）

```

```
private String message;      //聊天内容
public static final String KEY = "2";

@Override
public void exec(Scanner scanner) {
    System.out.print("请输入聊天的消息(id:message): ");
    String[] info = null;
    while (true) {
        String input = scanner.next();
        info = input.split(":");
        if (info.length != 2) {
            System.out.println("请输入聊天的消息
(id:message):");
        } else {
            break;
        }
    }
    toUserId = info[0];
    message = info[1];
}
//...
}
```

8.6.3 客户端的CommandController发送POJO

ChatConsoleCommand的调用者是CommandController命令控制类，该控制类在收集完成聊天内容和目标用户后，在自己的startOneChat()方法中调用ChatSender发送实例，将聊天消息组装成Protobuf数据包，通过客户端的通道发往服务端。

```
package com.crazymakercircle.imClient.client;  
//...  
  
public class CommandController {  
  
    @Autowired  
    ChatConsoleCommand chatConsoleCommand; //聊天命令收集器实  
例  
  
    //省略其他成员  
  
    public void startCommandThread() throws  
InterruptedException {  
  
        Thread.currentThread().setName("命令线程");  
  
        while (true) {  
  
            //建立连接  
  
            while (connectFlag == false) {  
  
                //开始连接  
  
                startConnectServer();  
  
                waitCommandThread();  
  
            }  
  
            //处理命令  
  
            while (null != session) {  
  
                Scanner scanner = new  
Scanner(System.in);  
  
                clientCommandMenu.exec(scanner);  
            }  
        }  
    }  
}
```

```
        String key =  
clientCommandMenu.getCommandInput();  
        BaseCommand command =  
commandMap.get(key);  
        //...  
        switch (key) {  
            case ChatConsoleCommand.KEY:  
                command.exec(scanner);  
  
startOneChat((ChatConsoleCommand) command);  
            break;  
            //省略其他命令  
        }  
    }  
}  
//发送单聊消息  
private void startOneChat(ChatConsoleCommand c) {  
    chatSender.setSession(session);  
    chatSender.setUser(user);  
    chatSender.sendChatMsg(c.getToUserId(),  
c.getMessage());  
}  
//省略其他的命令处理  
}
```

8.6.4 服务端的ChatRedirectHandler进行消息转发

服务端收到聊天消息后会进行消息的转发，主要由消息转发处理器ChatRedirectHandler负责，其大致的工作如下：

- (1) 对消息类型进行判断：判断是否为聊天请求Protobuf数据包。如果不是，通过调用super.channelRead(ctx, msg)将消息交给流水线的下一站。
- (2) 对消息发送方用户登录进行判断：如果没有登录，则不能发送消息。
- (3) 开启异步的消息转发，由其ChatRedirectProcesser实例负责完成消息转发。

```
package com.crazymakercircle.imServer.handler;  
//...  
  
public class ChatRedirectHandler extends  
ChannelInboundHandlerAdapter {  
  
    @Autowired  
    ChatRedirectProcesser chatRedirectProcesser;  
  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) ...{  
        //判断消息实例  
        if (null == msg || !(msg instanceof ProtoMsg.Message)) {  
            super.channelRead(ctx, msg);  
            return;  
        }  
    }  
}
```

```

//判断消息类型

ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
ProtoMsg.HeadType headType = ((ProtoMsg.Message)
msg).getType();

if (!headType.equals(chatRedirectProcesser.type())) {
    super.channelRead(ctx, msg);
    return;
}

//判断是否登录

ServerSession session = ServerSession.getSession(ctx);
if (null == session || !session.isLogin()) {
    log.error("用户尚未登录，不能发送消息");
    return;
}

//异步处理IM消息转发的逻辑

FutureTaskScheduler.add(() ->
{
    chatRedirectProcesser.action(session, pkg);
}) ;
}

}

```

8.6.5 服务端的ChatRedirectProcesser进行异步消息转发

ChatRedirectProcesser异步消息转发类负责将消息发送到目标用户，这是一个异步执行的任务，其大致功能如下：

- (1) 根据目标用户ID找出所有的服务端的会话列表。
- (2) 为每一个会话转发一份消息。

大致的代码如下：

```
package com.crazymakercircle.imServer.processor;  
//...  
public class ChatRedirectProcessor extends  
AbstractServerProcessor {  
  
    @Override  
    public ProtoMsg.HeadType type() {  
        return ProtoMsg.HeadType.MESSAGE_REQUEST;  
    }  
  
    @Override  
    public boolean action(ServerSession fromSession,  
                         ProtoMsg.Message  
proto) {  
        //聊天处理  
        ProtoMsg.MessageRequest msg =  
proto.getMessageRequest();  
        //获取接收方的chatID  
        String to = msg.getTo();  
        List<ServerSession> toSessions =
```

```

SessionMap.inst().getSessionsBy(to);

    if (toSessions == null) {
        //接收方离线，这里一般会做离线消息处理
        Print.tcfo("[" + to + "] 不在线，发送失败!");
    } else {
        toSessions.forEach((session) -> {
            //将IM消息发送到每一个接收方的通道
            session.writeAndFlush(proto);
        });
    }
    return true;
}
}

```

由于一个用户可能有多个会话，因此需要通过调用SessionMap会话管理器的SessionMap.inst().getSessionsBy(uid)方法来取得这个用户的所有会话。

```

package com.crazymakercircle.imServer.server;
//...
@Slf4j
@Data
public final class SessionMap {
    //全部的会话映射 "uid->session"
    private ConcurrentHashMap<String, ServerSession> map =
        new ConcurrentHashMap<String, ServerSession>();
}

```

```
//根据用户id获取会话集合

public List<ServerSession>getSessionsBy(String userId) {
    List<ServerSession> list = map.values()
        .stream()
        .filter(s ->
>s.getUser().getUid().equals(userId))
        .collect(Collectors.toList());
    return list;
}

//...
}
```

8.6.6 客户端的ChatMsgHandler聊天消息处理器

客户端的ChatMsgHandler聊天消息处理器很简单，主要工作如下：

- (1) 对消息类型进行判断，判断是否为聊天请求Protobuf数据包。如果不是，通过super.channelRead(ctx, msg)将消息交给流水线的下一站。
- (2) 如果是聊天消息，就将聊天消息显示在控制台。

```
package com.crazymakercircle.imClient.handler;

public class ChatMsgHandler extends
ChannelInboundHandlerAdapter {
```

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object
msg) ...{
    //判断类型
    ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
    ProtoMsg.HeadType headType = pkg.getType();
    if
        (!headType.equals(ProtoMsg.HeadType.MESSAGE_REQUEST)) {
            super.channelRead(ctx, msg);
            return; //不是聊天消息
        }
    ProtoMsg.MessageRequest req = pkg.getMessageRequest();
    String content = req.getContent();
    String uid = req.getFrom();
    System.out.println(" 收到消息 from uid:" + uid + " -> "
+ content);
}
}
```

8.7 详解心跳检测

通信过程中的心跳发送与心跳检测对于任何长连接的应用来说都是一个非常基础的功能。要理解心跳的重要性，首先需要从网络连接假死的现象开始。

8.7.1 网络连接的假死现象

什么是连接假死呢？如果底层的TCP连接（socket连接）已经断开，但是服务端并没有正常关闭套接字，服务端认为这条TCP连接仍然是存在的，则该连接处于“假死”状态。连接假死的具体表现如下：

- (1) 在服务端，会有一些处于TCP_ESTABLISHED状态的“正常”连接。
- (2) 在客户端，TCP客户端显示连接已经断开。
- (3) 虽然客户端可以进行断线重连操作，但是上一次的连接状态依然被服务端认为有效，并且服务端的资源得不到正确释放，包括套接字上下文以及接收/发送缓冲区。

说明

Socket连接状态（如TCP_ESTABLISHED）和连接建立时三次握手以及断开时四次挥手有关，请参阅本书后面有关TCP协议原理的内容。



连接假死的情况虽然不多见，但是确实存在。服务端长时间运行后会面临大量假死连接得不到正常释放的情况。由于每个连接都会耗费CPU和内存资源，因此大量假死的连接会逐渐耗光服务器的资源，使得服务器越来越慢，IO处理效率越来越低，最终导致服务器崩溃。

连接假死通常是由以下多个原因造成的，例如：

- (1) 应用程序出现线程堵塞，无法进行数据的读写。
- (2) 网络相关的设备出现故障，例如网卡、机房故障。
- (3) 网络丢包。公网环境非常容易出现丢包和网络抖动等现象。

解决假死的有效手段是客户端定时进行心跳检测、服务端定时进行空闲检测。

8.7.2 服务端的空闲检测

空闲检测就是每隔一段时间检测子通道是否有数据读写，如果有，则子通道是正常的；如果没有，则子通道被判定为假死，关掉子通道。

服务端如何实现空闲检测呢？使用Netty自带的IdleStateHandler空闲状态处理器就可以实现这个功能。下面的示例程序继承自IdleStateHandler，定义一个假死处理类：

```
package com.crazymakercircle.imServer.handler;
//...
public class HeartBeatServerHandler extends IdleStateHandler {
    private static final int READ_IDLE_GAP = 150; //最大空闲，单位秒
    public HeartBeatServerHandler() {
        super(READ_IDLE_GAP, 0, 0, TimeUnit.SECONDS);
    }
    @Override
    protected void channelIdle(ChannelHandlerContext ctx,
                               IdleStateEvent evt) ...
{
    System.out.println(READ_IDLE_GAP + "秒内未读到数据，关闭连接");
    ServerSession.closeSession(ctx);
}
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    //...
    ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
    //判断和处理心跳数据包
    ProtoMsg.HeartType headType = pkg.getType();
    if (headType.equals(ProtoMsg.HeartType.HEART_BEAT)) {
        //异步处理，将心跳数据包直接回复给客户端
        FutureTaskScheduler.add(() -> {
            if (ctx.channel().isActive()) {
```

```
        ctx.writeAndFlush(msg);

    }

}

super.channelRead(ctx, msg);

}

}
```

在HeartBeatServerHandler的构造函数中，调用了基类IdleStateHandler的构造函数，传递了四个参数：

```
public HeartBeatServerHandler() {

    super(READ_IDLE_GAP, 0, 0, TimeUnit.SECONDS);

}
```

其中，第一个参数表示入站（Inbound）空闲时长，指的是一段时间内如果没有数据入站，就判定连接假死；第二个参数是出站（Outbound）空闲时长，指的是一段时间内如果没有数据出站，就判定连接假死；第三个参数是出/入站检测时长，表示在一段时间内如果没有出站或者入站，就判定连接假死；最后一个参数表示时间单位，`TimeUnit.SECONDS`表示秒。

假死被判定之后，IdleStateHandler类会回调自己的`channelIdle()`方法。在这个子类的重写版本中，重写了空闲回调方法，手动关闭连接。

```
@Override
protected void channelIdle(ChannelHandlerContext ctx,
```

```
    IdleStateEvent evt) ...{  
        System.out.println(READ_IDLE_GAP + "秒内未读到数据，关闭连接");  
        ServerSession.closeSession(ctx);  
    }  
}
```

HeartBeatServerHandler实现的主要功能是空闲检测。在客户端，为了避免被误判，需要定时发送心跳数据包进行配合，而且客户端发送心跳数据包的时间间隔需要远远小于服务端的空闲检测时间间隔。

HeartBeatServerHandler收到客户端的心跳数据包之后，可以直接回复到客户端，其目的是让客户端也能进行类似的空闲检测。由于IdleStateHandler本身是一个入站处理器，因此只需重写这个子类HeartBeatServerHandler的channelRead()方法，然后将心跳数据包直接回复客户端即可。

说明

如果HeartBeatServerHandler要重写channelRead()方法（一般都会），那么一定要记得调用基类的“super.channelRead(ctx, msg);”，不然IdleStateHandler的入站空闲检测会无效。

8.7.3 客户端的心跳发送

与服务端的空闲检测相配合，客户端需要定期发送数据包到服务端，通常这个数据包称为心跳数据包。接下来，定义一个Handler业务处理器定期发送心跳数据包给服务端。

```
package com.crazymakercircle.imClient.handler;  
//...  
public class HeartBeatClientHandler  
        extends ChannelInboundHandlerAdapter {  
    //心跳的时间间隔，单位为秒  
    private static final int HEARTBEAT_INTERVAL = 50;  
    //在Handler业务处理器被加入到流水线时，开始发送心跳数据包  
    @Override  
    public void handlerAdded(ChannelHandlerContext ctx) ...{  
        ClientSession session = ClientSession.getSession(ctx);  
        User user = session.getUser();  
        HeartBeatMsgBuilder builder =  
            new HeartBeatMsgBuilder(user, session);  
        ProtoMsg.Message message = builder.buildMsg();  
        //发送心跳数据包  
        heartBeat(ctx, message);  
    }  
    //使用定时器，定期发送心跳数据包  
    public void heartBeat(ChannelHandlerContext ctx,  
                          ProtoMsg.Message heartbeatMsg) {  
        //提交一个一次性的定时任务  
        ctx.executor().schedule(() -> {  
            if (ctx.channel().isActive()) {
```

```

        log.info(" 发送HEART_BEAT 消息to server");
        ctx.writeAndFlush(heartbeatMsg);
        //递归调用：提交下一个一次性的定时任务，发送下一次的心跳
        heartBeat(ctx, heartbeatMsg);
    }

}, HEARTBEAT_INTERVAL, TimeUnit.SECONDS);

}

//接收到服务器的心跳回写
@Override
public void channelRead(ChannelHandlerContext ctx, Object
msg) {
    //判断类型
    ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
    ProtoMsg.HeadType headType = pkg.getType();
    if (headType.equals(ProtoMsg.HeadType.HEART_BEAT)) {
        log.info(" 收到回写的HEART_BEAT 消息 from server");
        return;
    } else {
        super.channelRead(ctx, msg);
    }
}
}

```

在HeartBeatClientHandler实例被加入到流水线时，它重写的handlerAdded()方法被回调。在handlerAdded()方法中，开始调用heartBeat()方法，发送心跳数据包。heartBeat()是一个不断递归调用的方法，方式比较特别：调用ctx.executor()获取当前通道绑定的

反应器NIO线程，然后通过NIO线程的schedule()定时调度方法，隔一段时间（50秒）执行一次回调，向服务端发送一个心跳数据包，并递归设置下一次心跳发送任务。

客户端的心跳发送间隔要比服务端的空闲检测时间间隔短，一般来说要比服务端监测间隔的一半短一些，可以直接定义为空闲检测时间间隔的1/3。这样做的目的就是防止公网偶发的秒级抖动。

HeartBeatClientHandler实例并不是一开始就装配到了流水线中的，它装配的时机是在登录成功之后。登录处理器LoginResponseHandler的相关代码如下：

```
package com.crazymakercircle.imClient.clientHandler;  
//...  
  
public class LoginResponseHandler  
    extends ChannelInboundHandlerAdapter {  
  
    @Override  
    public void channelRead(ChannelHandlerContext ctx,  
                           Object msg) {...  
        //省略登录数据包的预处理  
        if  
            (!result.equals(ProtoInstant.ResultCodeEnum.SUCCESS)) {  
                //登录失败  
                log.info(result.getDesc());  
            } else {  
                //登录成功  
                //省略其他处理  
                //在编码器后面动态插入心跳处理器
```

```
        ChannelPipeline p=ctx.pipeline();
        p.addAfter("encoder", "heartbeat", new
HeartBeatClientHandler());
    }
}
```

在登录成功之后，在ChannelPipeline上，
HeartBeatClientHandler实例被动态插入到encoder解码器之后。

服务端的空闲检测处理器在收到客户端的心跳数据包之后，会进行回写。在HeartBeatClientHandler的channelRead()方法中，对回写的数据包进行简单的处理。

这个地方可以设置另外一个机关——HeartBeatClientHandler可以继承IdleStateHandler类，使其在完成心跳处理的同时还能和服务器的空闲检测处理器一样在客户端进行空闲检测。这样，客户端可以对服务器进行假死判定，在服务端假死的情况下，客户端可以发起重连。客户端的空闲检测实战则留给大家自行去实验。

第9章 HTTP原理与Web服务器实战

高性能的IM（即时通信）应用还需要高性能的Web应用先配合。高并发、大流量的Web应用，QPS在十万每秒甚至上千万每秒，如何使用高并发HTTP通信技术去提升内部各个节点的通信性能对于提升分布式系统整体的吞吐量有着非常重大的作用。

说明

本章介绍一个小小的HTTP服务器程序——HTTP Echo回显服务器。如果能够顺利掌握此程序，可以进入下一个阶段实战练习：疯狂创客圈的spring-boot-netty-server开源项目实战。该项目的功能是在Spring Boot、Spring Cloud应用中使用Netty来替换Tomcat、Jetty、Undertow等传统的Web容器，通过该项目可以练习比较复杂的Netty服务端编程，该项目的地址为
<https://gitee.com/crazymaker/spring-boot-netty-server>。

9.1 高性能Web应用架构

本节按照流量规模分别对十万级、千万级高并发的Web应用架构进行简单介绍。

9.1.1 十万级并发的Web应用架构

QPS在十万每秒的Web应用，其架构大致如图9-1所示。

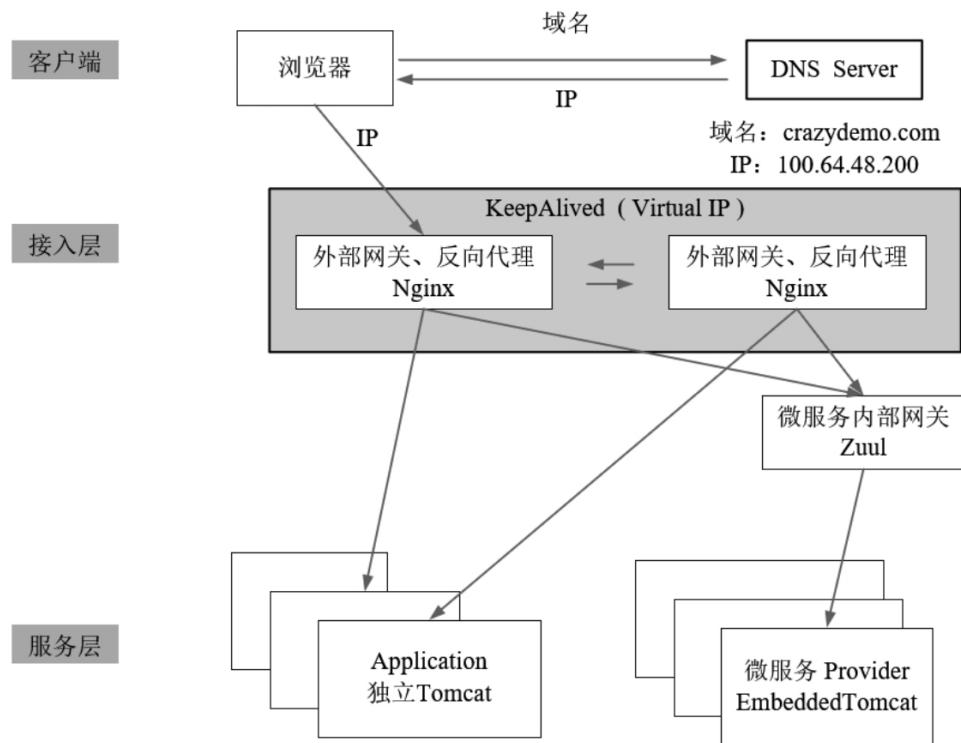


图9-1 十万级QPS的Web应用架构图

十万级QPS的Web应用架构主要包括客户端层、接入层、服务层，重点是接入层和服务层。

首先看服务层，在Spring Cloud微服务技术流程之前，服务层主要是通过Tomcat集群部署的向外提供服务的独立Java应用；在微服务技术成为主流之后，服务层主要是微服务Provider实例，并通过内部网关（如Zuul）向外提供统一的访问服务。

其次看接入层，接入层可以理解为客户端层与服务层之间的一个反向代理层，利用高性能的Nginx来做反向代理：

(1) Nginx将客户端请求分发给上游的多个Web服务；Nginx向外暴露一个外网IP，Nginx和内部Web服务（如Tomcat、Zuul）之间使用内网访问。

(2) Nginx需要保障负载均衡，并且通过Lua脚本可以具备动态伸缩、动态增加Web服务节点的能力。

(3) Nginx需要保障系统的高可用（High Availability），任何一台Web服务节点挂了，Nginx都可以将流量迁移到其他Web服务节点上。

Nginx的原理与Netty很像，也是应用了Reactor模式。Nginx执行过程中主要包括一个Master进程和 n ($n \geq 1$) 个Worker进程，所有的进程都是单线程（只有一个主线程）的。Nginx使用了多路复用和事件通知。其中，Master进程用于接收来自外界的信号，并给Worker进程发送信号，同时监控Worker进程的工作状态。Worker进程则是外部请求真正的处理者，每个Worker请求相互独立且平等的竞争来自客户端的请求。

因为Nginx应用了Reactor模式，所以在处理高并发请求时内存消耗非常小。在30000并发连接下，开启的10个Nginx进程才消耗

150 ($15 \times 10 = 150$) MB内存。

说明

有关Nginx的原理知识和具体的使用配置，请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

与Nginx类似、同样比较有名的Web服务器为Apache HTTP Server（纯Java实现）。该服务器在处理并发连接时会为每个连接建立一个单独的进程或线程，并且在网络输入/输出操作时阻塞。该阻塞式的IO将导致内存和CPU被大量消耗，因为新起一个单独的进程或线程需要准备新的运行时环境，包括堆内存和栈内存的分配，以及新的执行上下文，这些操作也会导致多余的CPU开销。最终会由于过多的上下文切换而导致服务器性能变差。因此，接入层的反向代理服务器原则上需要使用高性能的Nginx而不是Apache HTTP Server。

尽管单体的Nginx比较稳定，在长时间运行的情况下，还是存在有可能崩溃的情况。如何保障接入层的Nginx高可用呢？可以使用Nginx + KeepAlived组合模式，具体如下：

(1) 使用两台（或以上）Nginx组成一个集群，分别部署上KeepAlived，设置成相同的虚IP供下游访问，从而保证Nginx的高可用。

(2) 当一台Nginx挂了，KeepAlived能够探测到，并会将流量自动迁移到另一台Nginx上，整个过程对下游调用方透明。

如果流量不断增长，两台Nginx的集群模式不够，就可以使用LVS + KeepAlived组合模式实现Nginx的可扩展，并且在架构上进行升级，具体请看千万级流量的Web应用架构。

9.1.2 千万级高并发的Web应用架构

QPS在百万级甚至千万级的Web应用架构大致如图9-2所示。

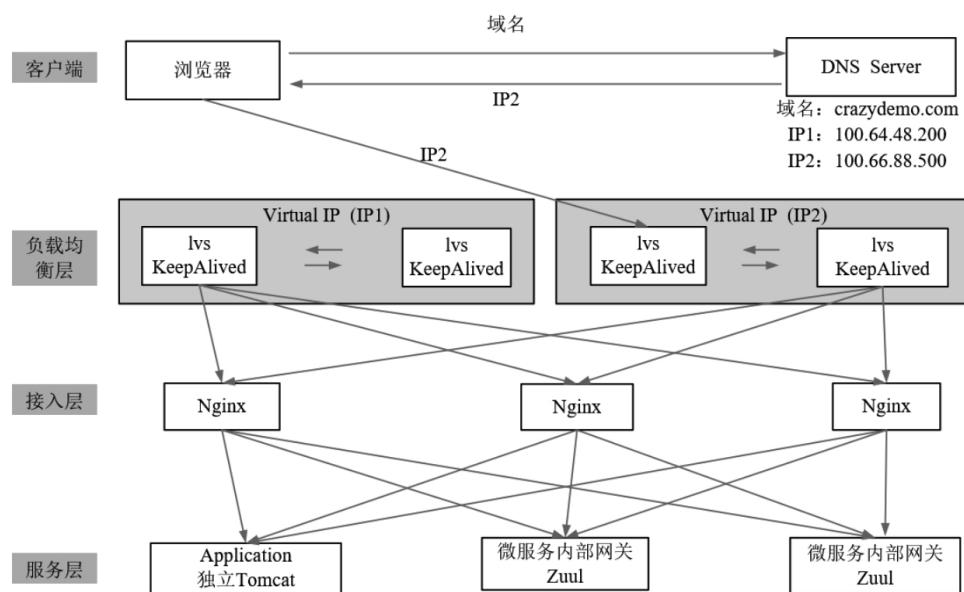


图9-2 百万级以上QPS的Web应用架构图

QPS在百万级甚至千万级的Web应用架构主要包括客户端层、负载均衡层、接入层、服务层，重点是客户端层和负载均衡层。

在客户端层，需要在DNS服务器上使用负载均衡的机制。DNS负载均衡的技术很简单，属于运维层面的技术，具体来说就是在DNS服务器中配置多个A记录，如表9-1所示。

表9-1 在DNS服务器中配置多个A记录示例

DNS 服务器	示 例
www.crazydemo.com IN A	114.100.80.1
www.crazydemo.com IN A	114.100.80.2
www.crazydemo.com IN A	114.100.80.3

通过在DNS服务器中配置多个A记录的方式可以在一个域名下面添加多个IP，由DNS域名服务器进行多个IP之间的负载均衡，甚至DNS服务器可以按照就近原则为用户返回最近的服务器IP地址。

DNS负载均衡虽然简单高效，但是也有不少缺点，具体如下：

(1) 通常无法动态调整主机地址权重（也有支持权重配置的DNS服务器），如果多台主机性能差异较大，则不能很好地均衡负载。

(2) DNS服务器通常会缓存查询响应，以便更迅速地向用户提供查询服务。在某台主机宕机的情况下，即使第一时间移除服务器IP也无济于事。

由于DNS负载均衡无法满足高可用性要求，因此通常仅仅被用于客户端层的简单复杂均衡。为了应对百万级、千万级高并发流量，需要在客户端与接入层之间引入一个专门的负载均衡层，该层通过LVS + KeepAlived组合模式达到高可用和负载均衡的目的。负载均衡层中的LVS (Linux Virtual Server, Linux虚拟服务器) 是一个虚拟的服务器集群系统，该项目在1998年5月由章文嵩博士成立，是国内最早出现的自由软件项目之一。

QPS在千万级的Web应用的高可用负载均衡层使用LVS + KeepAlived组合模式实现，具体的方案如下：

(1) 使用两台（或以上）LVS组成一个集群，分别部署上KeepAlive，设置成相同的虚IP（VIP）供下游访问。KeepAlive对LVS负载调度器实现健康监控、热备切换，具体来说，对服务器池中的各个节点进行健康检查，自动移除失效节点，恢复后再重新加入，从而保证LVS高可用。

(2) 在LVS系统上，可以配置多个接入层Nginx服务器集群，由LVS完成高速的请求分发和接入层的负载均衡。

LVS常常使用直接路由方式（DR）进行负载均衡，数据在分发过程中不修改IP地址，只修改MAC地址，由于实际处理请求的真实物理IP地址和数据请求目的IP地址一致，因此响应数据包可以不需要通过LVS负载均衡服务器进行地址转换，而是直接返回给用户浏览器，避免LVS负载均衡服务器网卡带宽成为瓶颈。此种方式又称作三角传输模式，具体如图9-3所示。

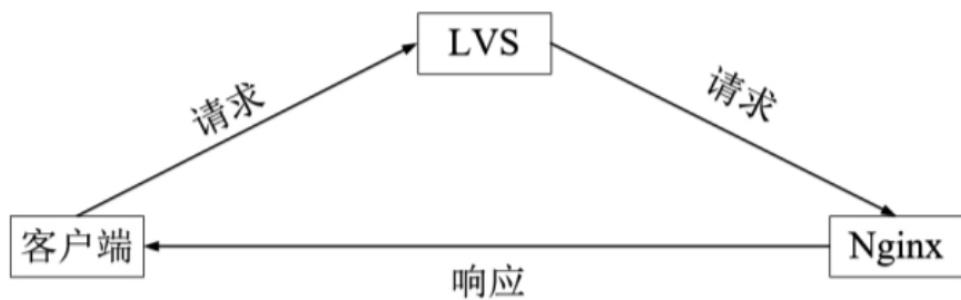


图9-3 三角传输模式

使用三角传输模式的链路层负载均衡是目前大型网站使用最广泛的一种负载均衡手段。目前，LVS是Linux平台上最好的三角传输模式软件负载均衡开源产品。当然，除了软件产品之外，还可以使用性能更好的专用硬件产品（如F5），但是其动辄几十万的昂贵价格并不是所有Web服务提供商所能承受的。

LVS目前已经是Linux标准内核的一部分，从Linux 2.4内核以后，无须专门给内核打任何补丁，可以直接使用LVS提供的各种功能。

说明

LVS和Nginx都具备负载均衡的能力，它们的区别是Nginx主要用于四层、七层的负载均衡，大家平时使用Nginx进行的Web Server负载均衡就属于七层负载均衡；LVS主要用于二层、四层的负载均衡，但是出于性能的原因，LVS更多用于二层（数据链路层）负载均衡。

什么是二层、四层、七层负载均衡？

(1) 二层（OSI模型的数据链路层）负载均衡：主要根据报文中的链路层内容（如MAC地址等）在多个上游服务器之间选择一个RS（Real Server，真实服务器），然后进行报文的处理和转发，从而实现负载均衡。

(2) 四层（OSI模型的传输层）负载均衡：主要通过修改报文中的目标IP地址和端口在多个上游TCP/UDP服务器之间选择一个RS，然后进行报文转发，从而实现负载均衡。

(3) 七层（OSI模型的应用层）负载均衡：主要根据报文中的应用层内容（如HTTP协议URI、Cookie信息、虚拟主机Host名称等）在多个上游应用层服务器（如HTTP Web服务器）之间选择一个RS，然后进行报文转发，从而实现负载均衡。

说明

上面所指的二层、四层、七层属于OSI模型的层次概念，不属于TCP/IP协议的层次概念，具体请参考后面章节有关TCP/IP协议的具体知识。

Nginx不具备二层的负载均衡能力，LVS不具备七层（应用层）的负载均衡能力，如果需要完成七层负载均衡的工作（如URL解析等），则使用LVS无法完成。

LVS的转发分为NAT模式（属于四层负载均衡）和DR模式（属于二层负载均衡），具体的介绍如下：

（1）LVS的NAT模式（属于四层负载均衡）

NAT（Network Address Translation）是一种外网和内网地址映射的技术，是一种网络地址转换技术。在NAT模式下，网络数据报的进出都要经过LVS的处理。LVS需要作为RS（真实服务器）的网关。

NAT包括目标地址转换（DNAT）和源地址转换（SNAT）。当包到达LVS时，LVS需要做目标地址转换（DNAT）：将目标IP改为RS的IP，RS在接收到数据包以后，仿佛是客户端直接发给它的一样；RS处理完返回响应时，源IP是RS的IP，目标IP是客户端的IP，这时LVS需要做源地址转换（SNAT），将包的源地址改为VIP（对外的IP），这样这个包对客户端看起来就仿佛是LVS直接返回给它的。

（2）LVS的DR模式（属于二层负载均衡）

DR模式也叫直接路由、三角传输模式。DR模式下需要LVS和RS集群绑定在同一个VIP上，与NAT的不同点在于：请求由LVS接收，处理后由RS直接返回给用户，响应返回的时候不经过LVS，所以也被形象地称为三角传输模式。

一个请求过来时，LVS只需要将网络帧的MAC地址修改为某一台RS的MAC，该包就会被转发到相应的RS处理，注意此时的源IP和目标IP都没变，RS收到LVS转发来的包时，链路层发现MAC是自己的，到上面的网络层，发现IP也是自己的，于是这个包被合法地接收，RS感知不到前面有LVS的存在。当RS返回响应时，只要直接向源IP（客户端的IP）返回即可，不再经过LVS转发。这里有一个系统运维的要点：RS的Loopback口需要和LVS设备上存在相同的VIP地址，这样响应才能直接返回到客户端。

在DR负载均衡模式下，数据在分发过程中不修改IP地址，只修改MAC地址，由于实际处理请求的真实物理IP地址和数据请求目的IP地址一致，因此不需要通过负载均衡服务器进行地址转换，其最大的优势为：可将响应数据包直接返回给用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈，因此DR模式具有较好的性能，是目前大型网站使用最广泛的一种负载均衡手段。

术业有专攻，LVS、KeepAlived的具体配置和运维更多的属于运维人员的工作，对于开发人员来说只要清楚其工作原理即可。

总之，如何抵抗十万级甚至千万级QPS访问洪峰，涉及大量的开发知识、运维知识，对于开发人员来说，并不一定需要掌握太多的操作系统层面（如LVS）的运维知识，主要原因是企业一般都会有专业的运

维人员去解决系统的运行问题，对千万级QPS系统中所涉及的高并发方面的开发知识则是必须掌握的。

在十万级甚至千万级QPS的Web应用架构过程中，如何提高平台内部的接入层Nginx到服务层Tomcat（或者其他Java容器）之间的HTTP通信能力涉及高并发HTTP通信以及TCP、HTTP等基础的知识。接下来，本书从HTTP应用层协议开始为大家解读这些作为Java核心工程师、架构师所必备的基础知识。

9.2 详解HTTP应用层协议

HTTP (Hyper Text Transfer Protocol, 超文本传输协议)，是一个基于请求与响应、无状态的应用层的协议，是互联网上应用最为广泛的一种网络协议，所有的WWW文件都必须遵守这个标准。设计HTTP的初衷是为了提供一种发布和接收HTML页面的方法。

关于TCP/IP和HTTP协议的关系，大致可以描述为在传输数据时，应用程序之间可以只使用TCP/IP（传输层）协议，如果没有应用层，应用程序便无法识别数据内容。如果想要使传输的数据有意义，则必须使用应用层协议。应用层协议有很多，比如HTTP、FTP、TELNET等，也可以自己定义应用层协议。

9.2.1 HTTP简介

HTTP是一个属于应用层的面向对象的协议，适用于分布式超媒体信息系统，是互联网上应用最为广泛的一种网络协议。所有的WWW文件都必须遵守这个标准。1960年美国人Ted Nelson构思了一种通过计算机处理文本信息的方法，并称之为超文本（Hyper Text），成为HTTP超文本传输协议标准架构的发展根基。最终，万维网协会（World Wide Web Consortium）和互联网工程工作小组（Internet Engineering Task Force）共同合作研究HTTP，最终发布了一系列的RFC文档，其中著名的RFC 2616定义了HTTP 1.1协议。

HTTP的主要特点可概括如下：

(1) 支持客户端/服务器模式。

(2) 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST，且每种方法规定了客户与服务器联系的类型不同。HTTP简单，使得HTTP服务器的程序规模较小，因此通信速度很快。

(3) 灵活：HTTP允许传输任意类型的数据对象，数据的类型由Content-Type加以标记。

(4) 无连接：每次连接只处理一个请求，服务器处理完客户的请求并收到客户的应答后即断开连接。

(5) 无状态：协议对于事务处理没有记忆能力。如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另外，在服务器不需要先前信息时它的应答较快。

总之，HTTP是请求-响应模式的协议，客户端发送一个HTTP请求，服务就响应此请求，大致如图9-4所示。

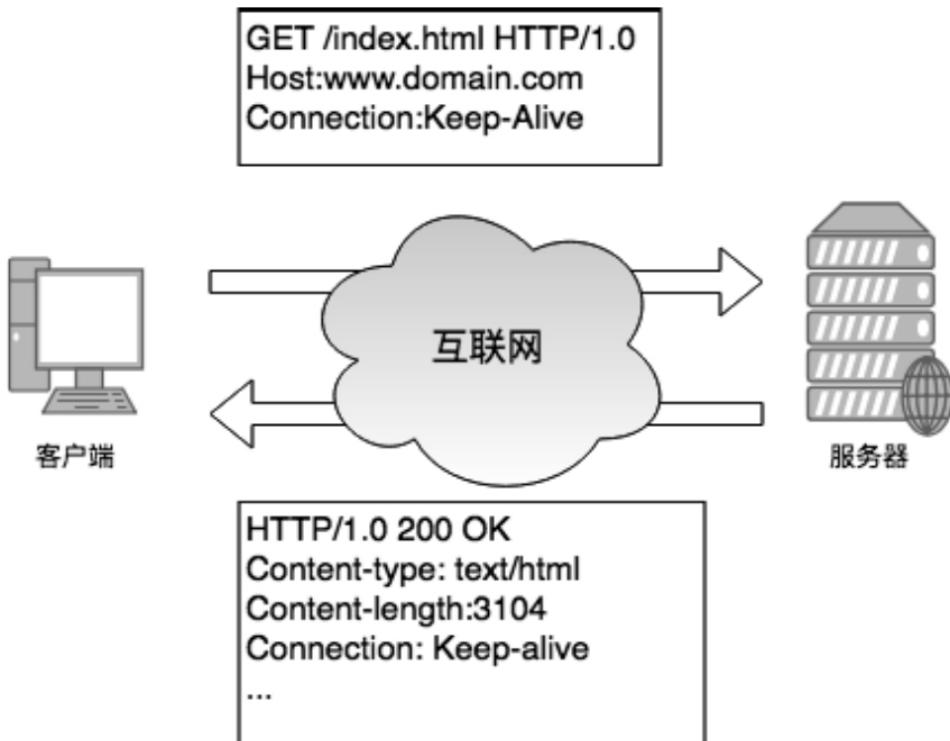


图9-4 HTTP的请求和响应示意

9.2.2 HTTP的请求URL

在HTTP通信中，客户端是终端用户，服务端一般是网站。通过使用Web浏览器、网络爬虫或者其他工具，客户端发起一个到服务器上指定端口（默认端口为80）的HTTP请求；对于该请求的应答，一般为服务器上存储的资源，比如HTML文件和图像。在客户端和源服务器中间可能存在多个中间层，比如代理、网关或者隧道（Tunnel）。

通常，由HTTP客户端发起一个请求，建立一个到服务器指定端口（默认是80端口）的TCP连接。HTTP服务器在那个端口监听客户端发送过来的请求。一旦收到请求，服务器（向客户端）就发回一个状态行（比如“HTTP/1.1 200 OK”）和消息响应。消息的消息体可能是请求的

文件、错误消息或者其他信息。为什么HTTP下层的传输层协议使用TCP而不是UDP呢？原因在于打开一个网页必须传送很多数据，而TCP提供传输控制，按顺序组织数据、纠正错误。

通过HTTP（或者HTTPS）协议请求的资源由统一资源标示符（Uniform Resource Identifier, URI）来标识。在Java编程中，更多的是URI的一个子类——URL（URL是一种特殊类型的URI，包含了用于查找某个资源的足够信息），其格式如下：

```
http://host[:port][abs_path]
```

在URL（统一资源定位符）中，http表示要通过HTTP来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用默认端口80；abs_path指定请求资源的URI；如果URL中没有给出abs_path，那么当请求URI时，就必须以“/”的形式给出，通常这个工作由浏览器自动帮我们完成。例如，通过浏览器地址栏输入“www.guet.edu.cn”，浏览器会自动转换成
http://www.guet.edu.cn/。

下面是一个URL的例子：

```
http://192.168.0.116:8080/index.jsp
```

9.2.3 HTTP的请求报文

HTTP请求由三部分组成，分别是请求行、请求头、请求体，一般也会将HTTP的请求行和请求头统一称为请求首部。

HTTP请求的请求行以一个方法（Method）符号开头，以空格分隔，后面跟着请求的URI和协议的版本，格式如下：

Method Request-URI HTTP-Version CRLF

其中的Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

为了能查看到HTTP请求报文的具体内容，这里使用Postman工具向一个特定的URI发送一个简单的POST请求，其请求体的内容如下：

```
{"msg": "Hello, World!. msg=sth"}
```

Postman工具的发送界面如图9-5所示。

The screenshot shows the Postman application window. At the top, it displays a POST request to 'http://crazydemo.com:9999/netty/echo'. The 'Body' tab is selected, showing the JSON content: {"msg": "Hello, World!. msg=sth"}. Below the body, the response status is shown as 200 OK with a time of 39 ms and a size of 435 B. The bottom section shows the raw JSON response:

```
1 "msg": "Hello, World!. msg=sth"
```

图9-5 使用Postman工具发送请求到http://crazydemo.com:9999/netty/echo

这里的/netty/echo服务是在本地开启的，那么为什么URL中使用的是crazydemo.com的域名而不是localhost呢？主要是为了能通过Fiddler工具进行报文抓取，localhost主机的报文该工具抓取不到。通过Fiddler抓取的请求报文大致如图9-6所示。

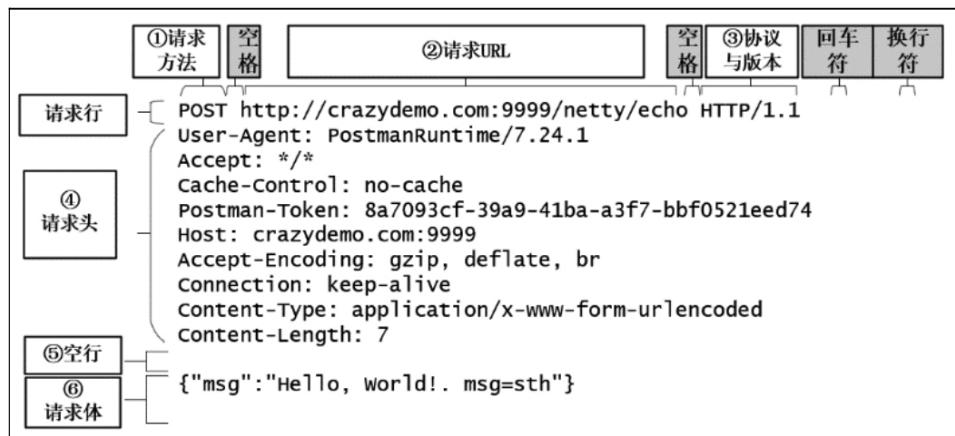


图9-6 Fiddler抓取的发送到http://crazydemo.com:9999/netty/echo的请求报文

说明

客户端请求地址“/netty/echo”所在的服务来自本章后面所开发的基于Netty的HTTP回显服务HttpEchoServer，在抓包之前，需要打开随书源码工程提前启动此服务。

HTTP请求报文由3部分组成（请求行+请求头+请求体）：

(1) Request Line (请求行)，包含请求方法、URL地址、协议名称和版本号。

(2) Request Header (请求头)，包含若干头部的字段。如有必要，客户程序还可以选择发送的请求头。大多数请求头并不是必需的，但Content-Length除外。对于POST请求来说，Content-Length必须出现。常见的请求头字段含义如下：

- ① Accept：客户端可接受的MIME类型。
- ② Accept-Charset：客户端可接受的字符集。
- ③ Accept-Encoding：客户端能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的客户端返回经gzip编码的HTML页面，许多情况下可以减少5~10倍的下载时间。
- ④ Accept-Language：客户端所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。
- ⑤ Authorization：用于设置用户身份信息，如果使用Authorization的方式进行认证，那么每次都要将认证的身份信息（如令牌）放到Authorization头部。
- ⑥ Content-Length：表示请求消息正文的长度。
- ⑦ Host：客户端通过这个头部信息告诉服务器想访问的主机名。Host头字段指定请求资源的主机和端口号，必须表示请求URL的原始服务器或网关的位置。HTTP/1.1请求必须包含主机头字段，否则系统会以400状态码返回。
- ⑧ If-Modified-Since：客户端通过这个头部信息告诉服务器资源的缓存时间。只有当所请求的内容在指定的时间后又经过修改才返回，否则返回304“Not Modified”应答。

⑨ Referer: 客户端通过这个头部字段告诉服务器它是从哪个资源来访问服务器的（防盗链）。Referer包含一个URL，表示用户从该URL代表的页面出发访问当前请求的页面。

⑩ User-Agent: 包含发出请求的用户信息。

⑪ Cookie: 客户端通过这个头部信息往服务器发数据，这是最重要的请求头信息之一。

⑫ Pragma: 值为“no-cache”，表示服务器必须返回一个刷新后的文档，如果服务器是代理服务器而且已经有了页面的本地缓存副本，则需要进行本地缓存副本的刷新。

⑬ From: 值为请求发送者的email地址，由一些特殊的Web客户程序使用，HTTP客户端不会用到。

⑭ Connection: 请求完成后是断开连接还是继续保持连接。如果值为“Keep-Alive”或者客户端使用的是HTTP 1.1（HTTP 1.1默认进行持久连接），它就可以利用持久连接的优点，当页面包含多个元素时（例如Applet，图片），会显著减少下载所需要的时间。当然，持久连接需要服务端进行配合，服务端需要在应答中发送一个Content-Length头，发送出响应内容的大小。

⑮ Range: 用于请求URL资源的部分内容，单位是byte（字节），并且从0开始。如果请求头携带了Range信息，就表示客户端需要进行分批下载或者分段传输。如果服务端支持分批下载，那么服务器会返回状态码206（Partial Content）以及该部分内容。如果服务器不支持分批下载，那么服务器会返回整个资源的大小以及状态码200。不同的请求范围对应的Range头部值如表9-2所示。

表9-2 Range头部值实例

Range 头部值	实 例
表示头 500 字节	bytes=0-499
表示第二个 500 字节	bytes=500-999
表示最后 500 字节	bytes=-500
表示 500 字节以后的范围	bytes=500-
第一个和最后一个字节	bytes=0-0, -1
同时指定几个范围	bytes=500-600, 601-999

⑯ UA-Pixels、UA-Color、UA-OS、UA-CPU：由某些版本的IE浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和CPU类型。

(3) Request Body（请求体），以文本或者其他形式组织的请求数据。若方法字段是GET，则请求体为空时表示没有请求体数据；若请求方法字段是POST，则通常来说此处放置的是要提交的数据。比如要使用POST方法提交一个表单，表单中user字段的数据为admin，password字段的数据为123456，那么这里的请求数据就是

“user=admin&password=123456”，HTTP协议会使用“&”符号连接各个字段。

对HTTP请求报文进一步细分，分为以下6个部分：

(1) HTTP Method（请求方法）。HTTP/1.1定义的请求方法有8种，即GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE，其中最常用的两种是GET和POST。如果是RESTful接口，一般会用到GET、POST、DELETE和PUT。

(2) HTTP报文的请求URL地址。它和报文头的Host属性组成完整的请求URL。URL可以传递请求参数，其方式类似于

“param1=value1¶m2=value2” 键-值对的字符串形式。

(3) 协议名称及版本号。

(4) HTTP报文的请求头。请求头包含若干个头部字段，每个字段的格式为“头部字段名:头部字段值”，服务端据此获取客户端的很多重要信息（如令牌）等。

(5) 空行。它的作用是通过一个空行告诉服务器请求头到此为止。

(6) HTTP报文的请求体：可以将一个页面表单中的组件值通过“param1=value1¶m2 =value2” 键-值对的形式编码成一个格式化串，从而用于承载多个请求参数。

总的来说，HTTP请求报文格式就如图9-7所示。

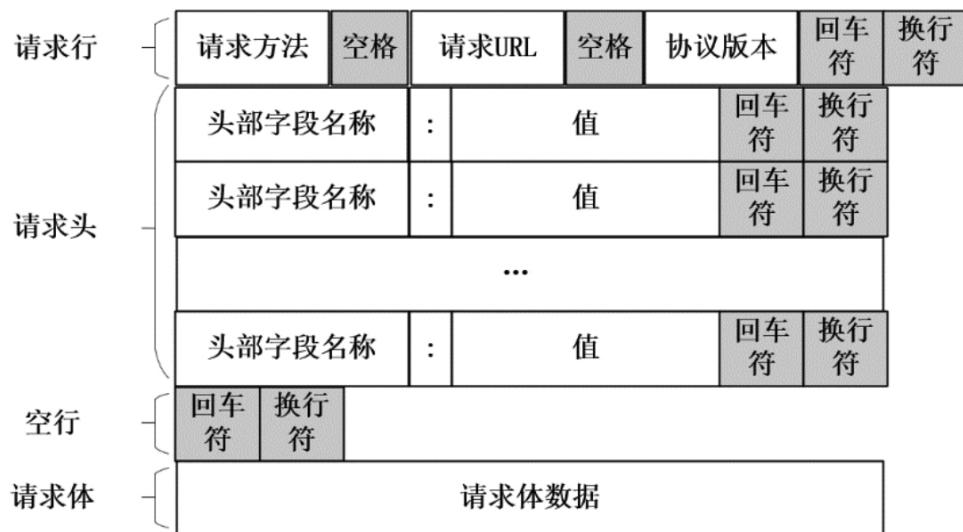


图9-7 HTTP请求报文格式

9.2.4 HTTP的响应报文

客户端向HTTP服务端发送请求之后，如果服务器能够正常处理并进行响应，就会向客户端发送HTTP响应。

在上一小节的示例中，使用Fiddler抓取的来自 `http://crazydemo.com:9999/netty/echo` 的响应报文大致如图9-8所示。



图9-8 Fiddler抓取的`http://crazydemo.com:9999/netty/echo`的响应报文

HTTP的响应报文也由3部分组成（响应行+响应头+响应体），具体示例如图9-9所示。

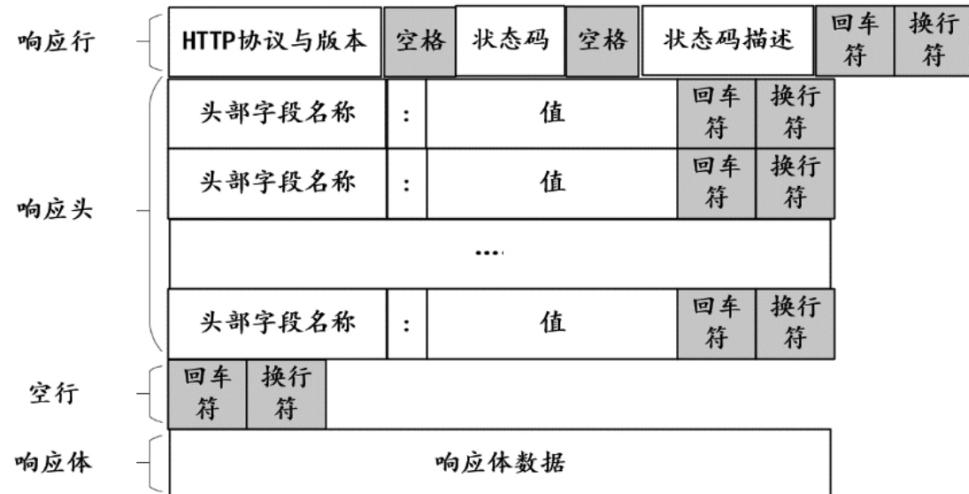


图9-9 HTTP的响应报文格式

(1) HTTP响应行：一般由协议版本、状态码及其描述组成，比如“HTTP/1.1 200 OK”。其中，协议版本为HTTP/1.1或者HTTP/1.0，200是状态码，OK为描述。常见状态码如表9-3所示。

表9-3 常见的HTTP响应行状态码

状态 码	说 明
100~199	表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程
200~299	表示成功接收请求并已完成整个处理过程，常用 200
300~399	为完成请求，客户需进一步细化请求
400~499	客户端的请求有错误，常用 404（请求的资源在 Web 服务器中没有）、403（服务器拒绝访问，如权限不够）
500~599	服务端出现错误，常用 500

(2) 响应头：用于描述服务器的基本信息和数据描述，服务器通过这些数据的描述信息可以通知客户端如何处理等一会它回送的数据。

设置HTTP响应头时往往和状态码结合起来。例如，有好几个表示“文档位置已经改变”的状态代码都伴随着一个Location头，而401状

态代码则必须伴随一个“WWW-authenticate”头部来表示未授权（Unauthorized）。响应头可以用来完成设置Cookie、指定修改日期、指示浏览器按照指定的间隔刷新页面、声明文档的长度以便利用持久HTTP连接等许多其他任务。

说明

响应码401（Unauthorized）和403（Forbidden）都是拒绝访问的意思，401和403的区别如下：

- 401表示服务端不知道客户端是谁。例如，Token失效、缺失甚至伪造，导致服务端无法识别客户端的身份，这时会返回401，客户端只能重试。
- 403表示服务端已经知道了客户端是谁，但是客户端没有权限去访问该数据资源。例如，客户端登录成功了，但却非要访问自己没有权限访问的内容，这时就会返回403。

常见的响应头字段大致如下：

- ① Allow：服务器支持哪些请求方法（如GET、POST等）。
- ② Content-Encoding：文档的编码（Encode）类型，如gzip压缩格式。客户端只有在解码之后才可以得到Content-Type头指定的内容类型。由于服务端返回gzip压缩文档能够显著地减少HTML文档的下载时间，因此服务端应该通过查看Accept-Encoding请求头检查客户端是

否支持gzip，为支持gzip的客户端返回经gzip压缩的HTML页面，而不支持gzip的其他客户端返回普通页面。

③ Content-Length：表示内容长度，只有当客户端使用持久HTTP连接时才需要这个数据。

④ Content-Type：表示后面的文档属于什么MIME类型。Servlet程序默认为text/plain，但通常需要显式地指定为text/html。由于经常要设置Content-Type，Servlet程序可以通过调用HttpServletResponse提供了一个专用的setContentType()方法去完成。

⑤ Date：当前的GMT时间，例如“Date:Mon, 31Dec200104:25:57GMT”。Date描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。可以用 setDateHeader来设置Date，以避免转换时间格式的麻烦。

⑥ Expires：告诉客户端把回送的资源缓存多长时间，-1或0表示不缓存。

⑦ Last-Modified：文档的最后改动时间。和客户端请求头配合使用，客户可以通过请求头If-Modified-Since提供一个起始时间，该请求头将被视为一个条件GET，只有改动时间迟于指定起始时间的文档才会返回，否则返回一个304（Not Modified）状态。

⑧ Location：配合302状态码使用，用于重定向接收者到一个新URI地址，表示客户应当到哪里去提取重定向文档。

⑨ Refresh：告诉客户端隔多久刷新一次，以秒计。

⑩ Server: 服务器通过这个头告诉客户端服务器的类型。Server响应头包含处理请求的原始服务器的软件信息。

⑪ Set-Cookie: 设置和页面关联的Cookie。

⑫ Transfer-Encoding: 告诉客户端数据的传送格式。

⑬ WWW-Authenticate: 告诉客户端应该在Authorization请求头中提供什么类型的授权信息。如果响应状态码为401 (Unauthorized)，则应答中这个头是必需的。

(3) 响应体: 响应的消息体，可以是文本内容或者二进制内容，比如JSON、HTML等都属于纯文本内容。

9.2.5 HTTP中GET和POST的区别

下面介绍GET和POST两种提交方式的区别。

1. 二者请求数据的放置位置不同

(1) 对于GET请求，请求的数据将会附在URL之后。具体来说，请求数据放置在HTTP请求行“Request-Line”的URL后面，以“?”分隔，多个参数之间用“&”连接，例如：

```
login.action?name=zhangsan&password=123456
```

如果数据是英文字母和数字，就会原样发送；如果数据是特殊字符中的空格，则转义为“+”；如果是中文或者其他字符，则直接把字符串用BASE64加密。

(2) 对于POST请求，提交的数据将被放置在HTTP请求报文的请求体中。

2. 二者所能传输数据的大小不同

虽然HTTP没有对传输的数据大小进行限制，协议规范也没有对URL长度进行限制，但是在实际开发中是存在限制的：

(1) 对于GET请求，特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。因此，GET提交时，传输数据会受到URL长度的限制。

(2) 对于POST请求，不是通过URL传值的，理论上数据不受限。实际上各个Web服务器会通过自定义设置对POST提交数据大小进行限制，Tomcat、Apache、IIS6都有各自的配置。

3. 二者传输数据的安全性不同

POST的安全性要比GET的安全性高。通过GET提交数据，用户名和密码将明文出现在URL上，通过查看浏览器的历史记录，就可以拿到其他用户的账号和密码了。

这里所说的安全性并不是指传输过程中的数据安全，也不是指传输过程中是否对数据进行加密保护，仅仅指的是数据可见性维度的浅层次数据安全。

9.3 HTTP的演进

HTTP在1.1版本之前具有无状态的特点，每次请求都需要通过TCP三次握手四次挥手与服务器重新建立连接。比如某个客户端在短时间内多次请求同一个资源，服务器并不能区别是否已经响应过用户的请求，所以每次需要重新响应请求、耗费不必要的文化和流量。为了节省资源消耗，HTTP也进行了发展和演进，通过持久连接的方法来进行连接复用。

HTTP是如今互联网的基石，其演进（见表9-4）从侧面反映了互联网技术的快速发展。

表9-4 HTTP的版本演进过程

版 本	产生时间	内 容	发展现状
HTTP/0.9	1991 年	不涉及数据包传输，规定客户端和服务器之间的通信格式，只能 GET 请求	没有作为正式的标准
HTTP/1.0	1996 年	传输内容格式不限制，增加 PUT、PATCH、HEAD、OPTIONS、DELETE 命令	正式作为标准
HTTP/1.1	1997 年	持久连接（长连接）、节约带宽、HOST 域、管道机制、分块传输编码	2015 年前使用广泛
HTTP/2.0	2015 年	多路复用、服务器推送、头部信息压缩、二进制协议等	逐渐覆盖市场

9.3.1 HTTP的1.0版本

第一个版本的HTTP是HTTP 0.9，组成极其简单，只允许客户端发送GET这一种请求，且不支持请求头。由于没有协议头，因此HTTP 0.9协议只支持一种内容，即纯文本。不过网页仍然支持用HTML语言格式化，同时无法插入图片。

HTTP的第二个版本为1.0版本，也是第一个在通信中指定版本号的HTTP版本，至今仍被广泛采用。相对于HTTP 0.9版本，HTTP 1.0版本增加了如下主要特性：

- (1) 请求与响应支持头部字段。
- (2) 响应对象以一个响应状态行开始。
- (3) 响应对象不只限于超文本。
- (4) 开始支持客户端通过POST方法向Web服务器提交数据，支持GET、HEAD、POST方法。
- (5) 支持长连接，但默认使用短连接，缓存机制，以及身份认证。
- (6) 请求行必须在尾部添加协议版本字段（HTTP 1.0），并且必须包含头部消息。

HTTP 1.0版本支持的请求方式为GET、POST和HEAD；请求访问的资源不再局限于上一个版本的HTML格式，可以根据Content-Type设置访问的格式；同时也开始支持Cache，当客户端在规定时间内访问同一URL资源时，直接访问Cache即可。

与HTTP 0.9版本相比，HTTP 1.0版本请求和回应的格式也变了。除了数据部分，每次通信都必须包括响应头信息（HTTP header），用来描述一些元数据。

HTTP 1.0版本使用Content-Type字段来表示客户端请求服务端的数据是什么格式，或者说客户端使用Content-Type来表示具体请求中

的媒体类型信息，服务端使用Content-Type来表示具体的响应体中的媒体类型信息。媒体类型（MediaType）的全称为互联网媒体类型（Internet Media Type），也叫作MIME（多用途互联网邮件扩展）类型。表9-5是一些常见的Content-Type字段的值。

表9-5 一些常见的Content-Type字段的值

Content-Type 字段值	说 明
text/html	HTML 格式
text/plain	纯文本格式
text/xml	XML 格式
image/gif	gif 图片格式
image/jpeg	jpg 图片格式
image/png	png 图片格式
application/xhtml+xml	XHTML 格式
application/xml	XML 数据格式
application/atom+xml	Atom XML 聚合格式
application/json	JSON 数据格式
application/pdf	pdf 格式
application/msword	Word 文档格式
application/octet-stream	二进制流数据（如常见的文件下载）
application/x-www-form-urlencoded	表单默认的提交数据的格式。表单 <form encType=""> 中默认的 encType 编码格式，form 表单数据默认被编码为 key=value 的键-值对格式发送到服务器
multipart/form-data	需要在表单中进行文件上传时使用该格式

MIME类型的每个值包括一级类型和二级类型，之间用斜杠分隔。除了预定义的类型，厂商也可以自定义类型，例如下面是一个自定义类型的例子：

application/vnd.debian.binary-package

上面的自定义MIME类型表明发送的是Debian系统的二进制数据包。

MIME类型值还可以在尾部使用分号、添加参数，下面是一个添加参数的例子：

```
Content-Type: text/html; charset=utf-8
```

上面的类型值表明HTTP报文中的内容是文本网页数据，并且文本的编码是UTF-8。

客户端在发送请求时可以使用Accept头部字段声明自己可以接受哪些数据格式。下面是一个Accept的例子：

```
Accept: */*
```

上面的Accept头部字段表明客户端声明自己可以接受来自服务端的任何格式的数据。

由于文本数据发送的时候往往可以通过压缩大大节省带宽，因此HTTP 1.0版本协议可以支持把数据压缩后再发送，其报文的Content-Encoding头部字段用于说明数据的压缩格式，具体如表9-6所示。

表9-6 用于Content-Encoding头部字段的压缩格式

头部字段的压缩格式	说 明
Content-Encoding: deflate	使用 RFC1950 说明的 zlib 格式进行数据压缩
Content-Encoding: gzip	使用 RFC1952 说明的 gzip 格式进行数据压缩
Content-Encoding: compress	使用 UNIX 的文件压缩程序对数据进行压缩

客户端在请求时可以使用Accept-Encoding字段说明自己可以接受哪些压缩方法，示例如下：

```
Accept-Encoding: gzip,deflate
```

上面的Accept-Encoding头部字段表明客户端声明自己可以接受来自服务端的zlib、gzip格式的压缩数据，但是不接受UNIX的文件压缩程序对数据进行压缩的数据。

除了Content-Type、Content-Encoding头部字段之外，HTTP 1.0版本其他的新增功能还包括响应状态码（Status Code）、多字符集支持、多部分发送（Multi-Part Type）、权限（Authorization）等。

除了上面的不同之外，HTTP 1.0版本与HTTP 0.9版本还有一个很重要的相同点：默认情况下HTTP 1.0版本的工作方式是每次发送一个请求需要一个TCP连接，当服务器响应后就会关闭这次连接，下一个请求需再次建立TCP连接，这点和HTTP 0.9版本的处理方式是一致的，具体如图9-10所示。

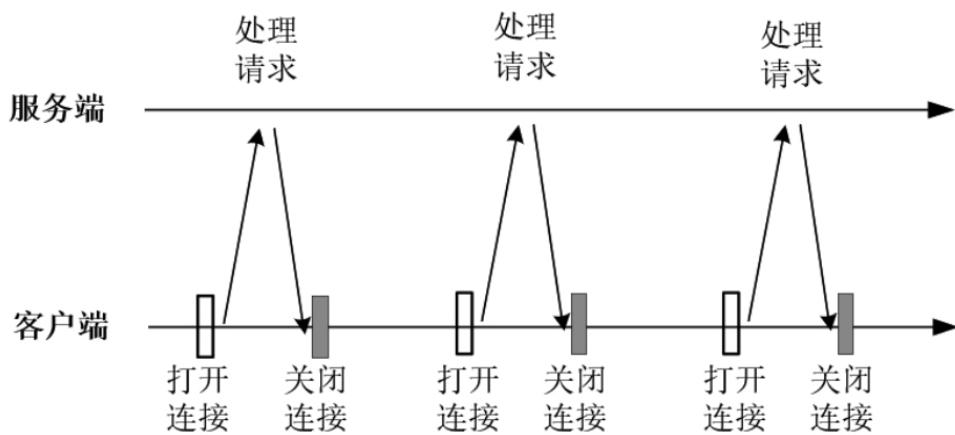


图9-10 HTTP 1.0版本与HTTP 0.9版本的请求处理方式

TCP连接的新建成本很高，因为建立连接时客户端和服务端三次握手，并且连接建立之初数据的发送速率较慢。所以，HTTP 1.0版本和HTTP 0.9版本一样，传输性能比较差，随着网页加载的外部资源越来越多，传输的性能问题就愈发突出了。

为了解决这个问题，有些浏览器在请求时对HTTP 1.0版本进行了扩展，增加了一个非标准的Connection头部字段，如果要对传输层的HTTP连接进行复用，Connection头部值如下：

```
Connection: keep-alive
```

这个头部字段要求服务器不要关闭TCP连接，以便其他HTTP请求复用，同样服务器需要回应这个字段。

```
Connection: keep-alive
```

如果连接的两端都有Connection: keep-alive头部，则一个可以复用的TCP连接就建立了，直到客户端或服务器主动关闭连接。但是，Connection不是标准字段，不同服务端实现的行为可能不一致，因此并不是提高传输性能的最终解决办法。

9.3.2 HTTP的1.1版本

HTTP的第三个版本是HTTP 1.1，是目前使用最广泛的协议版本，也是目前主流的HTTP版本。

HTTP 1.1版本引入了许多关键技术进行传输性能的优化，主要包括持久连接（Persistent Connection）、管道机制（Pipelining）、分块传输编码（Chunked Transfer Encoding, CTE）、字节范围（Range）请求等。

HTTP 1.1版本的最大变化就是引入了持久连接（Persistent Connection），即下层的TCP连接默认不关闭，可以被多个请求复用，而且报文不用声明“Connection: keep-alive”头部值。在HTTP 1.1

版本中，默认情况下一个TCP连接可以允许多个HTTP请求，具体如图9-11所示。

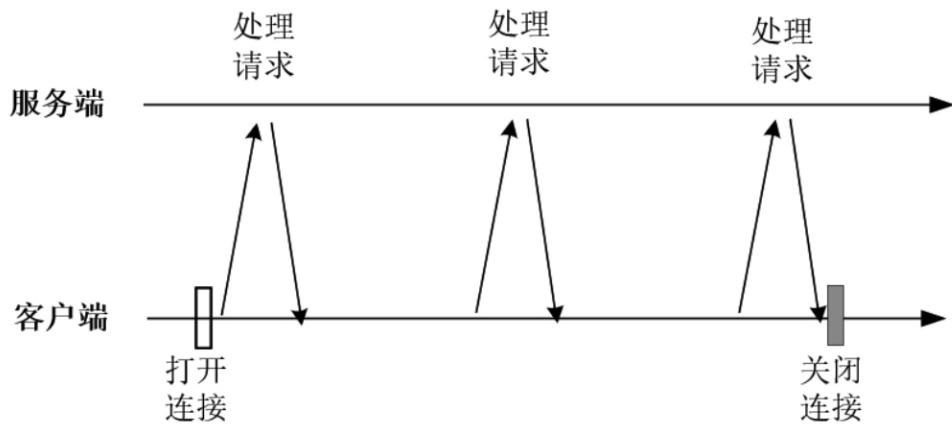


图9-11 HTTP 1.1版本的请求处理方式

TCP连接如何关闭呢？客户端和服务器都可以进行通信监测，如果发现对方在一段时间没有活动，就可以主动关闭TCP连接。不过，相对规范的做法是，客户端在最后一个请求时发送带“Connection：close”请求头的HTTP报文，明确要求服务器关闭TCP连接。

Connection: close

目前，对于同一个域名（带端口），大多数浏览器允许同时建立6个持久连接，这些持久连接在降低了传输延迟的同时也提高了带宽的利用率。

HTTP 1.1版本加入了管道机制，在同一个TCP连接里允许多个请求同时发送，增加了并发性，进一步改善了HTTP协议的效率。举例来说，客户端需要请求两个资源：以前的做法是，在同一个TCP连接里面先发送A请求，然后等待服务器做出回应，收到后再发出B请求；管道

机制则是允许浏览器同时发出A请求和B请求，但是服务器还是按照顺序先回应A请求，完成后再回应B请求，具体如图9-12所示。

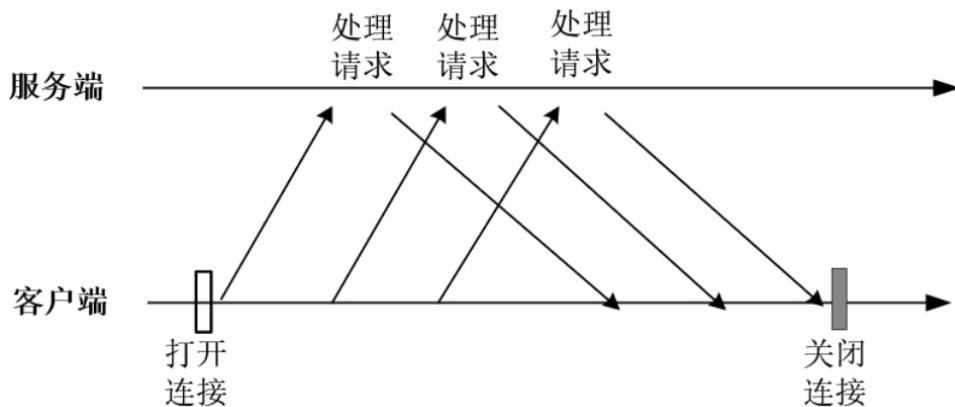


图9-12 HTTP 1.1版本加入了管道机制

在Method（请求方法）中，HTTP 1.1版本新增了PUT、PATCH、OPTIONS、DELETE等多种请求方法。

HTTP 1.1版本客户端请求的头部信息新增了Host字段，用来指定服务器的域名。在HTTP 1.0版本中，协议认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（Host Name）。随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-Homed Web Servers），并且它们共享一个IP地址甚至是端口号，为虚拟主机的兴起打下了基础。

有了Host字段，就可以将请求发往同一台服务器上的不同网站。通过Host字段，可以实现在一台Web服务器上的同一组IP地址和端口号上使用不同的主机名来创建多个虚拟Web站点，或者说，多个虚拟Server可以共享同一组IP地址和端口号。另外，在HTTP 1.1版本的请求消息中，如果没有Host头部字段，很多服务器会报告一个400（Bad Request）错误，Host头部的示例如下：

Host: www.example.com

HTTP 1.1版本加入了一个新的状态码100（Continue），服务端通过该响应码告知客户端继续发送后面的需求。例如，客户端事先发送一个只带令牌的Authorization头部字段而不带Body的请求，如果服务器因为权限拒绝了请求，就回送响应码401（Unauthorized）；如果服务器通过权限校验而接收此请求，就回送响应码100，客户端就可以继续发送带实体的完整请求了。

使用新的状态码100（Continue）可以允许客户端在发具有较大Body体积的消息之前用Request Header试探一下Server，看Server要不要接收Body，再决定是不是发Body。当Body的体积比较大时，在验证不能通过的情况下才能大大节约带宽，传输的性能优势非常明显。

HTTP 1.1版本加入了一些Cache的新特性。当缓存对象的Age超过Expire时，缓存对象变为Stale对象之后，HTTP 1.0版本会直接抛弃Stale对象，HTTP 1.1版本可以不直接抛弃Cache中的Stale对象，而是与源服务器进行重新验证（Revalidation）操作。

HTTP 1.1版本新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性地删除。

HTTP 1.1版本支持传送内容的一部分，也就是“字节范围请求”。当客户端已经拥有请求资源的一部分后，只需与服务器请求另外的部分资源即可。“字节范围请求”是支持文件断点续传的基础。

具体来说，“字节范围请求”是通过Range头部实现的，HTTP 1.0版本每次传送文件都是只能从文件头开始，即0字节处开始。在HTTP

1.1版本中，客户端通过“Range:bytes=XX”的请求头部值表示要求服务器从文件的“XX”字节处开始传送，也就是断点续传。其对应的部分内容的响应码不是200，而是使用专门的响应码206（Partial Content）。

HTTP 1.1版本支持分块（Chunked）传输编码。分块传输编码（Chunked Transfer Encoding, CTE）是一种新数据传输机制，允许服务端将数据分成多个部分发送到客户端。普通的服务端响应会将响应数据的长度通过Content-Length字段告诉客户端。

不过，使用Content-Length字段的前提条件是，服务器发送回应之前，必须知道回应的数据长度。对于一些很耗时的动态操作来说，这意味着服务器要等到所有操作完成才能发送数据，显然这样的效率不高。更好的处理方法是，产生一块数据就发送一块，采用“流模式（Stream）”发送取代“缓存模式（Buffer）”发送。

因此，HTTP 1.1版本规定请求或者响应报文可以不使用Content-Length字段告知长度，而使用分块传输编码（CTE）字段。只要请求或回应的头部有Transfer-Encoding字段，就表明数据将由数量未定的数据块组成。

Transfer-Encoding: chunked

每个分块报文的非空的数据块之前会有一个十六进制的数值，表示当前块的长度。最后是一个大小为0的块，表示本次回应的数据发送完了。

分块传输编码（CTE）的具体传输规则为：

(1) 在头部加入Transfer-Encoding: chunked之后，就代表这个报文采用了分块编码。这时报文中的实体需要改为用一系列分块来传输。

(2) 每个分块包含十六进制的长度值和数据，其中长度值独占一行。长度不包括分块长度后面结尾CRLF（\r\n）的长度，也不包括分块数据后面结尾CRLF（\r\n）的长度。

(3) 最后一个分块的长度值必须为0，对应的分块数据没有内容，表示所有的Body数据传输完成。

下面是一个例子。

HTTP/1.1 200 OK

Content-Type: text/plain

Transfer-Encoding: chunked

25

This is the data in the first chunk

1C

and this is the second one

3

con

8

sequence

注意，示例中的25、1C、3、8、0为十六进制的分片内容的净长度，并且不包括分片内容后面的\r\n的长度。

为什么在以上的示例报文中只有第一个分片报文有HTTP头部，后面的报文没有HTTP头部呢？因为HTTP1.1采用了持久的连接，也就是TCP的连接会进行复用，许多请求（或响应）分片（Chunked）在一个TCP的连接上发送，所以接收端通过最后一个长度为0分片（Chunked）标识当前的Body在这里结束即可。

9.3.3 HTTP的2.0版本

HTTP的2.0版本（或者说HTTP/2.0协议）是一个二进制协议。二进制更易于Frame（帧、数据包）的传输。HTTP 1.x版本在应用层以纯文本的形式进行通信，HTTP 2.0将所有的传输信息分割为更小的消息和数据帧，并对它们采用二进制格式编码。这样，客户端和服务端都需要引入新的二进制编码和解码的机制，就像本书前面编写的Protobuf聊天数据帧的编码器和解码器一样。

HTTP/2.0协议有10个不同Frame定义，其中两个最为基础的Frame是Data帧和Headers帧，其中HTTP/1.x报文的头部信息会被封装到HTTP/2.0报文的Headers帧中，而HTTP/1.x报文的请求体（Request Body）则被封装到HTTP/2.0报文的Data帧中，具体如图9-13所示。

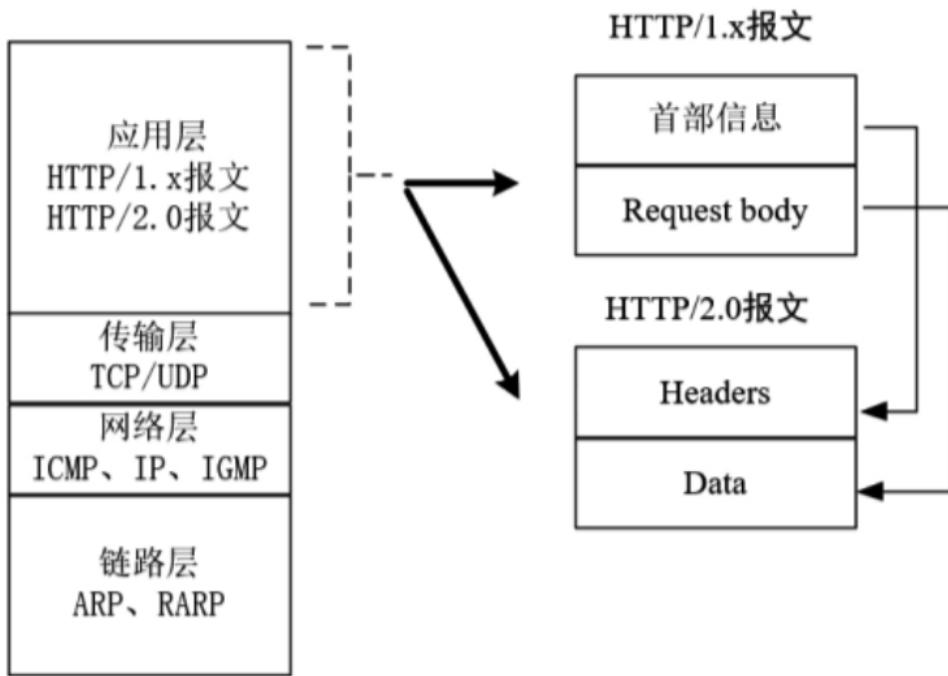


图9-13 HTTP/2.0协议与HTTP/1.x协议的报文对应关系

通过以上报文对应关系可以看出，HTTP/2.0协议没有改变HTTP/1.x协议的语义，只是在应用层使用二进制分帧方式传输。HTTP 2.0最大的特点是没有改动HTTP的语义，包括HTTP方法、状态码、URI 及请求头首部字段等。HTTP/1.x的核心概念在语义上一如往常，但是HTTP/2.0协议却改进了传输性能，实现了低延迟和高吞吐量。

HTTP/2.0协议引入了新的通信单位：帧、消息、流。分帧有什么好处？服务器单位时间接收到的请求数变多，可以提高并发数。最重要的是，为多路复用提供了底层支持。HTTP/2.0协议之所以叫HTTP/2.0版本而不是HTTP/1.2版本，关键在于新增的二进制分帧传输在传输的方式上发生了重大变化。

既然要保证HTTP的各种方法、首部都不受影响，又需要通过二进制进行传输，那就需要在应用层和传输层（TCP/UDP）之间增加一个二

进制分帧层，在该二进制分帧层上，HTTP 2.0版本会将所有传输的信息分割为更小的消息和数据帧，并对它们采用二进制格式的编码。然后，HTTP 2.0协议的通信都在一个连接上完成，这个连接可以承载任意数量的双向数据流。

HTTP/2协议的主要特点有首部压缩、多路复用、并行双向传输、服务端推送等。

1. 首部压缩

HTTP/2.0协议在客户端和服务端使用“首部（请求头）表”来跟踪和存储之前发送的请求头键-值对，对于相同的数据，不再通过每次请求和响应发送；通信期间几乎不会改变通用键-值对的值（如用户代理、可接受的MIME值等），所以请求头只需发送一次即可。事实上，如果请求中不包含首部（例如对同一资源的请求），那么首部开销就是零字节。此时所有首部都自动使用之前请求发送的首部。

如果请求的首部发生了变化，那么只需要在Headers帧里发送变化了的首部，将新增或修改的首部帧追加到“首部表”即可。首部表中的键-值对在HTTP/2.0协议的TCP连接存续期内始终存在，由客户端和服务器共同渐进地更新。

2. 多路复用

HTTP/2.0协议的多路复用指的是对多资源的请求可以在一个TCP连接上完成。HTTP/2.0协议把HTTP协议通信的基本单位缩小为一个一个的帧，这些帧对应着逻辑流中的消息，并行地在同一个TCP连接上双向交换消息。

实际上，HTTP性能的关键在于低延迟而不是带宽利用率低。大多数HTTP连接的时间都很短，数据传输是突发性的，但是TCP传输只有在长连接并且传输大块数据时其效率才是最高的。HTTP/2.0协议通过让所有数据流共用同一个连接，可以更有效地让TCP连接高带宽，也能真正地服务于HTTP的性能提升。

多资源单链接的多路复用方式在服务器和网络传输的层面都得到了好处：

- (1) 可以减少服务链接压力，内存占用少了，连接吞吐量大了。
- (2) 由于TCP连接减少而使网络拥塞状况得以改观。
- (3) TCP慢启动时间减少，拥塞和丢包恢复速度更快。

3. 并行双向传输

在HTTP/2.0协议中，客户端和服务器可以把HTTP消息分解为互不依赖的帧，然后乱序发送，最后在另一端把它们重新组合起来。注意，同一连接上有多个不同方向的数据流在传输。客户端可以一边乱序发送消息流，也可以一边接收服务器的响应流，而服务器那端同理。

把HTTP消息分解为独立的帧，双向交错发送，然后在另一端重新组装，是HTTP/2.0重要的一项增强。该机制会在整个Web技术栈中引发一系列连锁反应，从而带来巨大的性能提升，大致的原因是：

- (1) 可以并行交错地发送请求，请求之间互不影响。
- (2) 可以并行交错地发送响应，响应之间互不干扰。

(3) 只使用一个连接即可并行发送多个请求和响应。

(4) 消除不必要的延迟，从而减少页面加载的时间。

4. 服务端推送

在HTTP/2.0协议中，新增的一个强大的功能就是服务器可以对一个客户端请求发送多个响应。或者说，除了对最初请求的进行响应外，服务器还可以额外向客户端推送资源，而无须客户端明确地请求。

在服务端主动推送这一点上，HTTP/2.0协议和WebSocket协议有点类似。

那么，如何使用HTTP/2.0协议呢？前提是需要Web服务器和浏览器双方都支持，任何一端不匹配，都会回退到HTTP/1.1协议。

有数据表明，全球排名1000万个网站只有12%左右支持HTTP/2.0协议。目前所有新版本的浏览器包括Firefox、Safari、Chrome以及其他基于Blink核心的浏览器已完全支持HTTP/2.0协议。虽然目前HTTP/1.1协议还是主流，但是相信不久的将来HTTP/2.0协议会大行其道。

9.4 基于Netty实现简单的Web服务器

Netty天生是异步事件驱动的架构，无论是在性能上还是在可靠性上都表现优异，非常适合作为Web服务器使用，相比于传统的Tomcat、Jetty等Web容器，基于Netty的Web服务器具有更加轻量和小巧、灵活性和定制性更好的特点。

9.4.1 基于Netty的HTTP服务器演示实例

在学习基于Netty进行HTTP处理的相关知识之前，先介绍一下本节实现的演示服务器示例——HttpEchoServer，这是一个简单基于Netty的HTTP回显服务器。

HttpEchoServer的功能是：当通过HTTP客户端（如Postman工具、浏览器等）向演示服务器发起HTTP请求时，服务器会回显该HTTP请求的请求方法、请求参数、请求URI、请求头、请求体等内容。

使用Fiddler工具抓取的一个HttpEchoServer服务器的回显结果大致如图9-14所示。

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
{
  "request method": "GET",
  "paramsFromGet": {
    "param1": "value1",
    "param2": "value2"
  },
  "request uri": "/getrequest?param1=value1&param2=value2",
  "request header": {
    "content-length": "0",
    "Accept": "*/*",
    "Cache-Control": "no-cache",
    "foo": "bar",
    "User-Agent": "PostmanRuntime/7.22.0",
    "Connection": "keep-alive",
    "Postman-Token": "956a87c8-26b4-4bc0-bb7d-f5d24ebf6336",
    "Host": "crazydemo.com:18899",
    "Accept-Encoding": "gzip, deflate, br",
    "Content-Type": "application/x-www-form-urlencoded"
  }
}
```

图9-14 一个HttpEchoServer服务器的响应结果示意图

说明

在具体的调试过程中，为了能通过Fiddler（抓包工具）抓取到HTTP的往返报文，需要将本地地址（127.0.0.1）在操作系统（这里是Windows）的hosts文件中绑定到crazydemo.com主机名称上，只有这样，在调试过程中将向该主机名称发送请求才能成功抓取报文。

基于Netty的HTTP回显服务器的服务端Pipeline处理器流水线构成大致如图9-15所示。

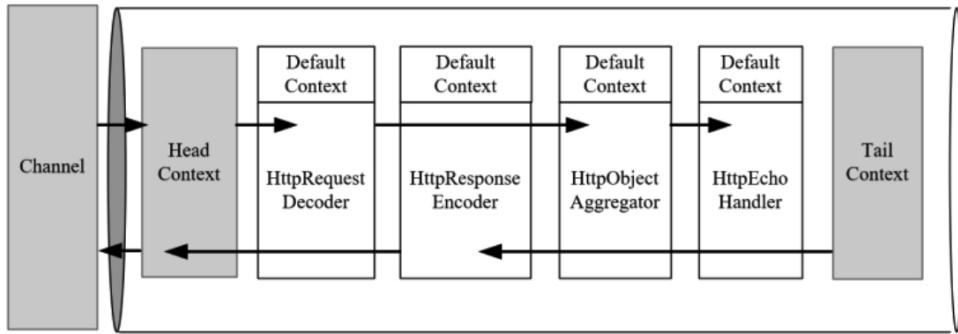


图9-15 基于Netty的HTTP回显服务器的处理器流水线

9.4.2 基于Netty的HTTP请求的处理流程

通常HTTP协议通信过程中，客户端和服务端的交互过程如下：

- (1) 客户端（如Postman工具、浏览器、Java程序等）向服务端发送HTTP请求。
- (2) 服务端对HTTP请求进行解析。
- (3) 服务端向客户端发送HTTP响应报文。
- (4) 客户端解析HTTP响应的应用层协议内容。

在以上交互过程中，服务端将涉及HTTP请求的解码处理和HTTP响应的编码处理。不过，Netty已经内置了这些解码和编码的处理器，大致如下：

- (1) HttpRequestDecoder：HTTP请求编码器，是一个入站处理器，间接地继承了ByteToMessageDecoder，将ByteBuf缓冲区解码成代表请求的HttpRequest首部实例和HttpContent内容实例，并且

HttpRequestDecoder在解码时会处理好分块（Chunked）类型和固定长度（Content-Length）类型的HTTP请求报文。

(2) HttpResponseEncoder: HTTP响应编码器，把代表响应的HttpResponse首部实例和HttpContent内容实例编码成ByteBuf字节流，是一个出站处理器。

(3) HttpServerCodec: HTTP的编解码器是HttpRequestDecoder解码器和HttpResponseEncoder编码器的结合体。

(4) HttpObjectAggregator: 是HttpObject实例聚合器，也是一个入站处理器。通过HttpObject实例聚合器，可以把HttpMessage首部实例和一个或多个HttpContent内容实例最终聚合成一个FullHttpRequest实例。上文中涉及的与HTTP相关的HttpMessage、HttpRequest、HttpContent、FullHttpRequest等类型都是HttpObject的子类。

(5) QueryStringDecoder: 把HTTP的请求URI分割成Path路径和Key-Value参数键-值对，同一次请求，该解码器仅能使用一次。

基于Netty的HTTP请求的处理流程大致如下：

(1) 二进制的HTTP数据包从Channel通道入站后，首先进入Pipeline流水线的是ByteBuf字节流。

(2) HttpRequestDecoder首先将ByteBuf缓冲区中的请求行(Request Line)和请求头Header解析成HttpRequest首部对象，传入到HttpObjectAggregator。然后将HTTP数据包的请求体Body解析出HttpContent对象（可能是多个），传入到HttpObjectAggregator聚合

器。解码完成之后，如果没有更多的请求体内容，
HttpRequestDecoder会传递一个LastHttpContent结束实例到聚合器
HttpObjectAggregator，表示HTTP请求数据已经解析完成。

(3) 当HttpObjectAggregator发现有入站包为LastHttpContent
实例入站时，代表HTTP请求数据协议解析完成，此时会将所收到的全
部HttpObject实例封装成一个FullHttpRequest整体请求实例发送给下
一站，这里的下一站基本上为业务处理器。

Netty的HTTP请求的处理流程大致如图9-16所示。

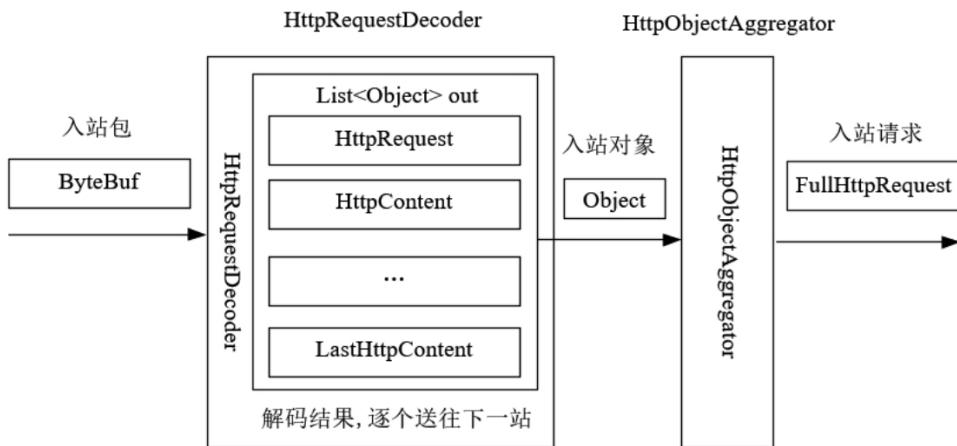


图9-16 Netty的HTTP请求的处理流程

在请求体Request Body处理过程中会涉及Content-Length和Trunked两种类型请求体，但是其处理差异被HttpRequestDecoder协议解码器所屏蔽，它们的最终出站对象是一致的，通过聚合器HttpObjectAggregator处理之后，输出的都是FullHttpRequest实例。HTTP服务端的业务处理器（如EchoHandler）可以通过该FullHttpRequest实例获取到所有与HTTP请求的内容。

总体来说，如果要进行HTTP请求报文的读取，只需要在Netty的流水线上配置好两个内置处理器HttpRequestDecoder和HttpObjectAggregator即可。

以本节的HttpEchoServer演示实例的服务端处理器为例，大致的流水线装配代码如下：

```
ChannelPipeline pipeline = ch.pipeline();
//请求的解码器
pipeline.addLast(new HttpRequestDecoder());
//请求聚合器
pipeline.addLast(new HttpObjectAggregator(65535));
//响应的编码器
pipeline.addLast(new HttpResponseEncoder());
//自定义的业务Handler
pipeline.addLast(new HttpEchoHandler());
```

9.4.3 Netty内置的HTTP报文解码流程

通过内置处理器HttpRequestDecoder和HttpObjectAggregator对HTTP请求报文进行解码之后，Netty会将HTTP请求封装成一个FullHttpRequest实例（具体见图9-17），然后发送给下一站。

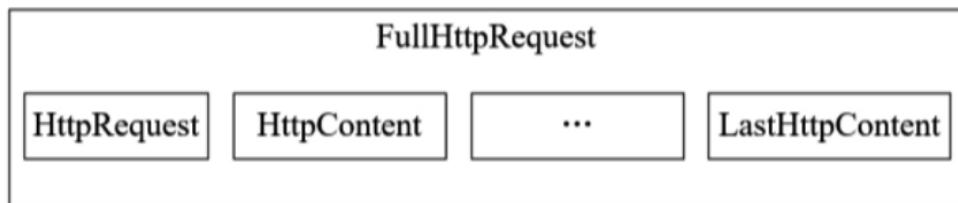


图9-17 FullHttpRequest结构图

Netty内置的与HTTP请求报文相对应的类大致有如下几个：

- (1) FullHttpRequest：包含整个HTTP请求的信息，包含对HttpRequest首部和HttpContent请求体的组合。
- (2) HttpRequest：请求首部，主要包含对HTTP请求行和请求头的组合。
- (3) HttpContent：对HTTP请求体进行封装，本质上就是一个ByteBuf缓冲区实例。如果ByteBuf的长度是固定的，则请求体过大，可能包含多个HttpContent。解码的时候，最后一个解码返回对象为LastHttpContent（空的HttpContent），表示对请求体的解码已经结束。
- (4) HttpMethod：主要是对HTTP请求方法的封装。
- (5) HttpVersion：对HTTP版本的封装，该类定义了HTTP/1.0和HTTP/1.1两个协议版本。
- (6) HttpHeaders：包含对HTTP报文请求头的封装及相关操作。

以上清单中的类与HTTP请求报文各部分的对应关系大致如图9-18所示。

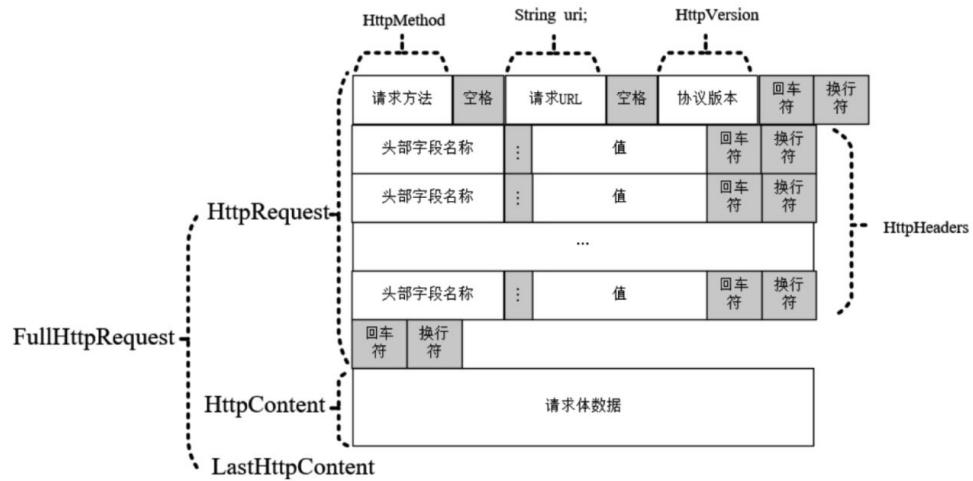


图9-18 HTTP报文各部分所对应的Netty类

Netty的HttpRequest首部类中有一个String uri成员，主要是对请求URI的封装，该成员包含了HTTP请求的Path路径与跟随在其后的请求参数。

有关请求参数的解析，不同的Web服务器所使用的解析策略有所不同。在Tomcat中，如果客户端提交的是application/x-www-form-urlencoded类型的表单Post请求，则Java请求参数实例除了包含跟随在URI后面的键-值对之外，请求参数还包含HTTP请求体Body中的键-值对。在Netty中，Java中请求参数实例仅仅包含跟在URI后面的键-值对。

接下来介绍本节的重点：Netty的HTTP报文拆包方案。

一般来说，服务端收到的HTTP字节流可能被分成多个ByteBuf包。Netty服务端如何处理HTTP报文的分包问题呢？大致有如下几种策略：

(1) 定长分包策略：接收端按照固定长度进行数据包分割，发送端按照固定长度发送数据包。

(2) 长度域分包策略：比如使用LengthFieldBasedFrameDecoder长度域解码器在接收端分包，而在发送端先发送4个字节表示消息的长度，紧接着发送消息的内容。

(3) 分隔符分割：比如使用LineBasedFrameDecoder解码器通过换行符进行分包，或者使用DelimiterBasedFrameDecoder通过特定的分隔符进行分包。

Netty结合使用上面第(2)种和第(3)种策略完成HTTP报文的拆包：对于请求头，应用了分隔符分包策略，以特定分隔符（"\r\n"）进行拆包；对于HTTP请求体，应用长度字段中的分包策略，按照请求头中的内容长度进行内容拆包。

Netty总体的HTTP拆包方案具体如下：

(1) 处理HTTP请求行，由于请求行的边界是CRLF（"\r\n"），如果读取到CRLF，则意味着请求行的信息已经读取完成。

(2) 开始处理请求头部分，由于Header的边界是CRLF，每遇到一个CRLF，则表示一个请求头读取完成；如果连续读取到两个CRLF，则意味着全部Header的信息读取完成。

(3) 请求体的长度一般由请求头Content-Length来进行确定。如果请求头中没有Content-Length头部，则属于“块编码”报文，具体的解析方式请参考Trunked协议。

为了减少内存复制，Netty使用了CompositeByteBuf。例如，Netty聚合各个HttpObject实例的FullHttpResponse实现类，内部就是一个CompositeByteBuf实例，该组合缓冲区会将HttpRequest内部的

ByteBuf、HttpContent内部的ByteBuf都组合在一起，作为最终的HTTP报文缓冲区，从而避免数据拷贝（也就是内存复制），具体如图9-19所示。

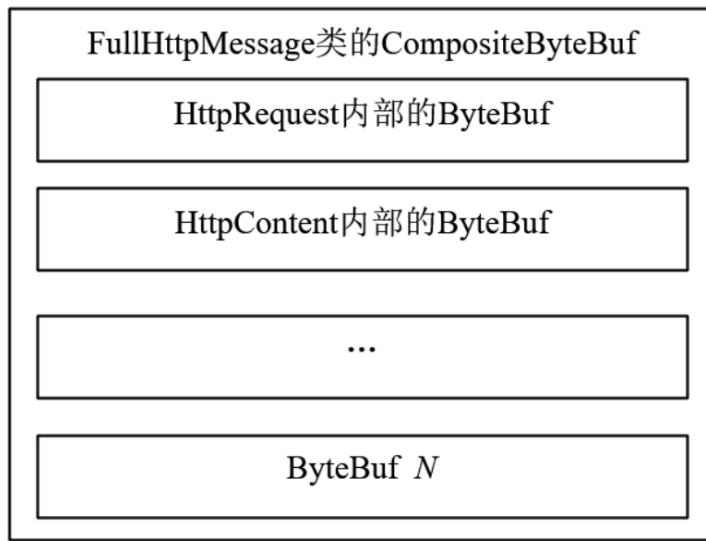


图9-19 FullHttpResponse实现类内部的CompositeByteBuf成员

9.4.4 基于Netty的HTTP响应编码流程

Netty的HTTP响应的处理流程只需在流水线装配 HttpResponseEncoder编码器即可。该编码器是一个出站处理器，有以下特点：

- (1) 该编码器输入的是FullHttpResponse响应实例，输出的是ByteBuf字节缓冲器。后面的处理器会将ByteBuf数据写入Channel，最终被发送到HTTP客户端。
- (2) 该编码器按照HTTP对入站FullHttpResponse实例的请求行、请求头、请求体进行序列化，通过请求头去判断是否含有Content-

Length头或者Trunked头，然后将请求体按照相应的长度规则对内容进行序列化。

Netty的HTTP响应的编码流程具体如图9-20所示。

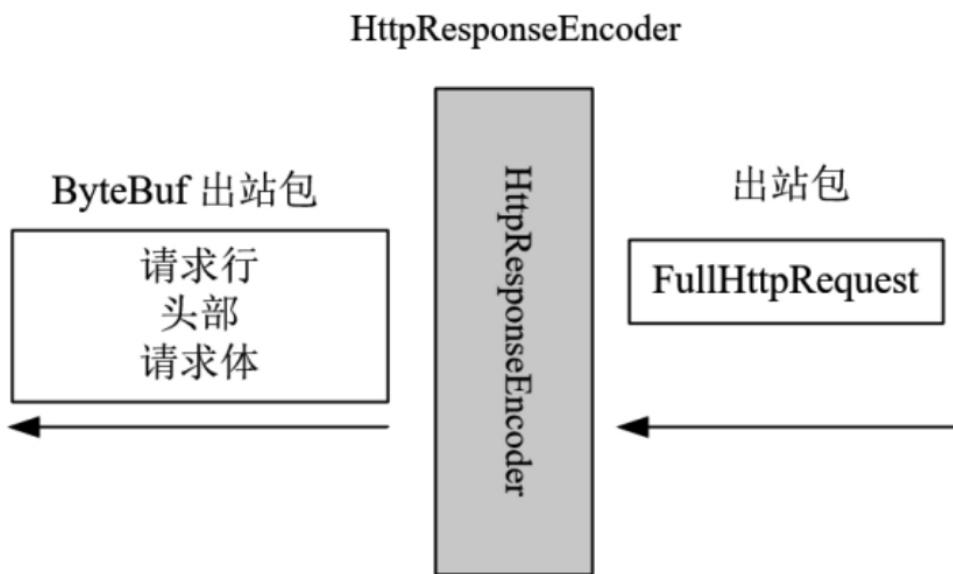


图9-20 Netty的HTTP响应的编码流程

如果只是发送简单的HTTP响应，就可以通过 DefaultFullHttpResponse默认响应实现类完成。通过该默认响应类既可以设置响应的内容，又可以进行响应头的设置。在本书的随书源码中编写了一个HttpProtocolHelper帮助类，通过该响应类进行HTTP响应的设置和发送，相关的部分代码如下：

```
package com.crazymakercircle.netty.util;  
...  
public class HttpProtocolHelper  
{  
    ...
```

```
 /**
 * 发送Json格式的响应
 * @param ctx 上下文
 * @param content 响应内容
 */
public static void sendJsonContent(
        ChannelHandlerContext ctx, String
content)
{
    HttpVersion version = getHttpVersion(ctx);
    /**
     * 构造一个默认的FullHttpResponse实例
     */
    FullHttpResponse response = new DefaultFullHttpResponse(
            version, OK, Unpooled.copiedBuffer(content,
CharsetUtil.UTF_8));
    /**
     * 设置响应头
     */
    response.headers().set(HeaderNames.CONTENT_TYPE,
            "application/json;
charset=UTF-8");
    /**
     * 发送FullHttpResponse响应内容
     */
    sendAndCleanupConnection(ctx, response);
}
```

```
/*
 * 发送FullHttpResponse响应
 */
public static void sendAndCleanupConnection(
    ChannelHandlerContext ctx, FullHttpResponse
response)
{
    final boolean keepAlive =
        ctx.channel().attr(KEEP_ALIVE_KEY).get();
    HttpUtil.setContentLength(
        response,
        response.content().readableBytes());
    if (!keepAlive)
    {
        //如果不是长连接，就设置connection:close头部
        response.headers().set(
            HttpHeadersNames.CONNECTION,
            HttpHeadersValues.CLOSE);
    } else if (isHTTP_1_0(ctx))
    {
        //如果是1.0版本的长连接，就设置connection:keep-alive头部
        response.headers().set(
            HttpHeadersNames.CONNECTION,
            HttpHeadersValues.KEEP_ALIVE);
    }
    //发送内容
}
```

```
        ChannelFuture flushPromise =
ctx.writeAndFlush(response);

        if (!keepAlive)
{
    //如果不是长连接，那么发送完成之后关闭连接
    flushPromise.addListener(ChannelFutureListener.CLOSE);
}

...
}
```

9.4.5 HttpEchoHandler回显业务处理器的实战案例

基于Netty的HttpEchoHandler回显业务处理器，将来自客户端的HTTP客户端的请求方法、URI请求参数、请求体数据、请求头字段回显到客户端（写回到客户端）。回显业务处理器主要对GET请求、Form表单POST请求、JSON类型的POST请求进行处理，所涉及的可以回显处理的客户端请求大致如图9-21所示。

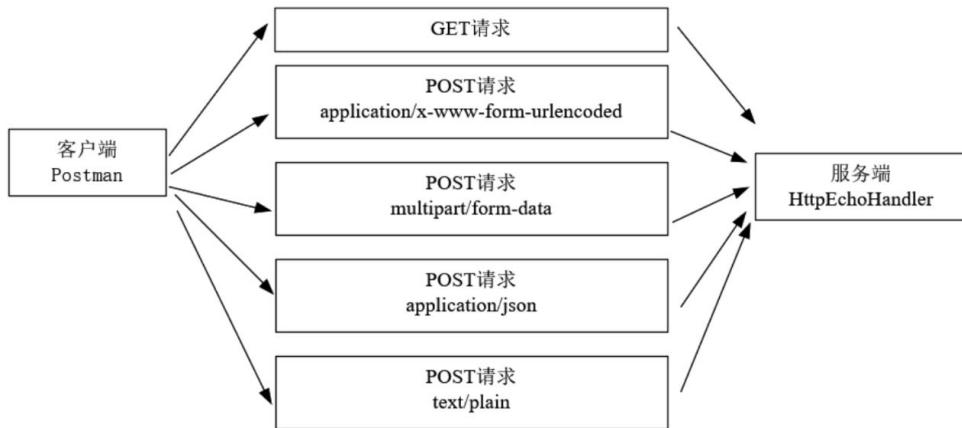


图9-21 HttpEchoHandler所涉及的请求类型

HttpEchoHandler回显处理器的主要实现代码大致如下：

```

package com.crazymakercircle.netty.http.echo;

...
@Slf4j
public class HttpEchoHandler extends
SimpleChannelInboundHandler<FullHttpRequest>
{

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
                           FullHttpRequest request) throws
Exception
    {
        if (!request.decoderResult().isSuccess())
        {
            HttpProtocolHelper.sendError(ctx, BAD_REQUEST);
        }
    }
}

```

```
        return;
    }
}

/**
 * 调用辅助类的方法，缓存HTTP协议的版本号
 */

HttpProtocolHelper.cacheHttpProtocol(ctx, request);

Map<String, Object> echo = new HashMap<String, Object>

();

//1.获取URI

String uri = request.uri();

echo.put("request uri", uri);

//2.获取请求方法

HttpMethod method = request.method();

echo.put("request method", method.toString());

//3.获取请求头

Map<String, Object> echoHeaders = new HashMap<String,

Object>();

HttpHeaders headers = request.headers();

//迭代请求头

Iterator<Map.Entry<String, String>> hit =


headers.entries().iterator();

while (hit.hasNext())

{

    Map.Entry<String, String> header = hit.next();

    echoHeaders.put(header.getKey(),

header.getValue());
}
```

```
}

echo.put("request header", echoHeaders);

/***
 * 获取URI请求参数
 */

Map<String, Object> uriDatas = paramsFromUri(request);
echo.put("paramsFromUri", uriDatas);

//处理POST请求

if (POST.equals(request.method()))
{
    /**
     * 获取请求体数据到 map
     */
    Map<String, Object> postData =
dataFromPost(request);

    echo.put("dataFromPost", postData);
}

/**
 * 回显内容转换成json字符串
 */
String sendContent = JsonUtil.pojoToJson(echo);

/***
 * 发送回显内容到客户端
*/
```

```

        */
        HttpProtocolHelper.sendJsonContent(ctx,
sendContent);
    }

/*
 * 从URI后面获取请求的参数
*/
private Map<String, Object> paramsFromUri(
    FullHttpRequest
fullHttpRequest)
{
    Map<String, Object> params = new HashMap<String,
Object>();
    //把URI后面的参数串分割成键-值对的形式
    QueryStringDecoder decoder =
        new
QueryStringDecoder(fullHttpRequest.uri());
    //提取键-值对形式的参数串
    Map<String, List<String>> paramList =
decoder.parameters();
    //迭代键-值对形式的参数串
    for (Map.Entry<String, List<String>>
entry : paramList.entrySet())
    {
        params.put(entry.getKey(),
entry.getValue().get(0));
    }
}

```

```
        }

        return params;
    }

/*
 * 获取POST方式传递的请求体数据
 */
private Map<String, Object> dataFromPost(
    FullHttpRequest
fullHttpRequest)
{
    Map<String, Object> postData = null;

    try
    {
        String contentType =
            fullHttpRequest.headers().get("Content-
Type").trim();
        //普通form表单数据，非multipart形式表单
        if (contentType.contains(
            "application/x-www-form-
urlencoded"))
        {
            postData = formBodyDecode(fullHttpRequest);
        }
        //multipart形式表单
        else if (contentType.contains("multipart/form-

```

```
data"))
{
    postData = formBodyDecode(fullHttpRequest);
}

//JSON形式的POST请求
else if (contentType.contains("application/json"))
{
    postData = jsonBodyDecode(fullHttpRequest);
}

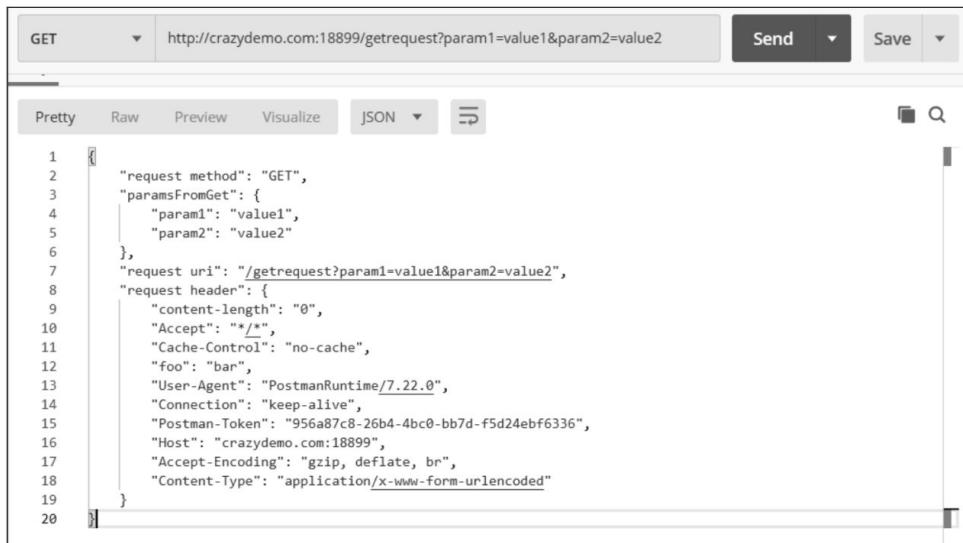
//普通文本形式的POST请求
else if (contentType.contains("text/plain"))
{
    ByteBuf content = fullHttpRequest.content();
    byte[] reqContent = new
byte[content.readableBytes()];
    content.readBytes(reqContent);
    String text = new String(reqContent, "UTF-8");
    postData = new HashMap<String, Object>();
    postData.put("text", text);
}

return postData;
} catch (UnsupportedEncodingException e)
{
    return null;
}
}
```

```
...  
}
```

为了节省篇幅，HttpEchoServer回显服务器主类（也是引导类）的代码在这里不再贴出，请大家通过随书源码工程查看。

运行HttpEchoServer的main()方法，正式启动HTTP回显服务。同时，为了抓取和查看报文，可以开启Fiddler抓包程序，在一切都准备妥当之后，可以在Postman中输入一个带参数的URI，去访问回显服务器。服务端所返回的回显结果大致如图9-22所示。



```
GET http://crazydemo.com:18899/getrequest?param1=value1&param2=value2  
Send Save  
Pretty Raw Preview Visualize JSON  
1 [ {  
2   "request method": "GET",  
3   "paramsFromGet": {  
4     "param1": "value1",  
5     "param2": "value2"  
6   },  
7   "request uri": "/getrequest?param1=value1&param2=value2",  
8   "request header": {  
9     "content-length": "0",  
10    "Accept": "*/*",  
11    "Cache-Control": "no-cache",  
12    "foo": "bar",  
13    "User-Agent": "PostmanRuntime/7.22.0",  
14    "Connection": "keep-alive",  
15    "Postman-Token": "956a87c8-26b4-4bc0-bb7d-f5d24ebf6336",  
16    "Host": "crazydemo.com:18899",  
17    "Accept-Encoding": "gzip, deflate, br",  
18    "Content-Type": "application/x-www-form-urlencoded"  
19  }  
20 }
```

图9-22 Postman提交GET请求到回显服务器后的返回结果

9.4.6 使用Postman发送多种类型的请求体

接下来的演示通过Postman发送多种类型的POST请求体。按照Content-Type进行划分，POST请求体有很多编码类型，以下为常见的几种：

(1) `text/plain`: 请求体以普通文本形式编码，其中不含任何控件或格式字符。

(2) `application/json`: 请求体以JSON格式编码。

(3) `application/x-www-form-urlencoded`: 请求体被编码为“名称=值”对相连的形式，这是标准的表单项编码格式。

(4) `multipart/form-data`: 请求体被编码为多部分，每个表单数据对应到消息中的一部分。

(5) `application/octet-stream`: 从字面意思得知，请求体只可以发送二进制数据，通常用来上传文件。因为没有键的名称，所以该类型请求体一次只能上传一个文件。

这里重点对`application/x-www-form-urlencoded`和`multipart/form-data`两种请求体内容类型进行使用的演示和介绍。

实验1：发送`application/x-www-form-urlencoded`编码类型的请求体

首先演示和介绍`application/x-www-form-urlencoded`类型的请求体编码形式。该类型的请求体会将表单的每个表单项名称和值转换为“名称=值”的形式，然后用“`&`”符号连在一起，最终将整个表单编码后的字符串作为POST请求的请求体发送出去。如果是GET请求，则将编码后的字符串追加到URI后面发送出去。在Postman提交该类型的POST请求到回显服务器，具体的请求URL如下：

`http://crazydemo.com:18899/postrequest`

服务端返回的回显结果大致如图9-23所示。

The screenshot shows the Postman interface for a POST request to `http://crazydemo.com:18899/postrequest`. The 'Body' tab is selected, showing the following data:

KEY	VALUE
<input checked="" type="checkbox"/> key1	value1
<input checked="" type="checkbox"/> key2	value1
Key	Value

Below the table, the JSON response is displayed:

```
1 {
2   "request method": "POST",
3   "paramsFromUri": {},
4   "request uri": "/postrequest",
5   "dataFromPost": {
6     "key1": "value1",
7     "key2": "value1"
8   },
9   "request header": {
10    "Accept": "*/*",
11    "Cache-Control": "no-cache",
12    "foo": "bar",
13    "User-Agent": "PostmanRuntime/7.26.3",
14    "Connection": "keep-alive",
15    "Postman-Token": "854907f7-e347-49d3-b6e9-8b27b95c26da",
}
```

图9-23 Postman提交application/x-www-form-urlencoded类型的POST请求体

通过Fiddler查看到以上POST请求的应用层HTTP协议数据包，具体如图9-24所示。

The screenshot shows the Fiddler tool interface with the raw request data highlighted:

```
POST http://crazydemo.com:18899/postrequest HTTP/1.1
foo: bar
Content-Type: application/x-www-form-urlencoded
User-Agent: PostmanRuntime/7.26.3
Accept: */*
Cache-Control: no-cache
Postman-Token: 854907f7-e347-49d3-b6e9-8b27b95c26da
Host: crazydemo.com:18899
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 23

key1=value1&key2=value1
```

图9-24 Fiddler查看multipart/form-data类型的POST请求数据包

实验2：发送multipart/form-data编码类型的请求体

这里介绍一下multipart/form-data类型的请求体编码形式。在Postman提交该类型的POST请求到回显服务器，具体如图9-25所示。

The screenshot shows the Postman interface for a POST request to `http://crazydemo.com:18899/postrequest`. The 'Body' tab is selected, showing the following form-data:

KEY	VALUE
<input checked="" type="checkbox"/> formkey1	value1
<input checked="" type="checkbox"/> formkey2	value2
Key	Value

Below the table, the JSON response is displayed:

```
1 {  
2   "request method": "POST",  
3   "paramsFromUri": {},  
4   "request uri": "/postrequest",  
5   "dataFromPost": {  
6     "formkey1": "value1",  
7     "formkey2": "value2"  
8   },|
```

图9-25 Postman提交multipart/form-data类型的POST请求体

浏览器对于multipart/form-data类型的POST请求的报文编码稍微有点复杂。它在将表单项编码成请求体时，会将每一个表单项分开进行编码。每个表单项都有一个Content-disposition来说明表单项的类型，表单Field字段的类型值为form-data（数据）、File字段的类型值为file（文件）。紧跟在Content-disposition属性的后面，每个表单项都有一个name属性，其值为表单项的名称。在name名字之后是两个“\r\n”，然后是表单项的值，如果是上传文件，则此处为文件的

内容；每个表单项的末尾都有一段boundary分隔字符串，隔开自己和下一个表单项。

为了演示方便，仅仅在Postman中将上一个请求的Content-Type改为multipart/form-data，填写两个表单项，然后发送POST请求到回显服务器，并通过Fiddler抓包查看请求体的内容，大致如图9-26所示。

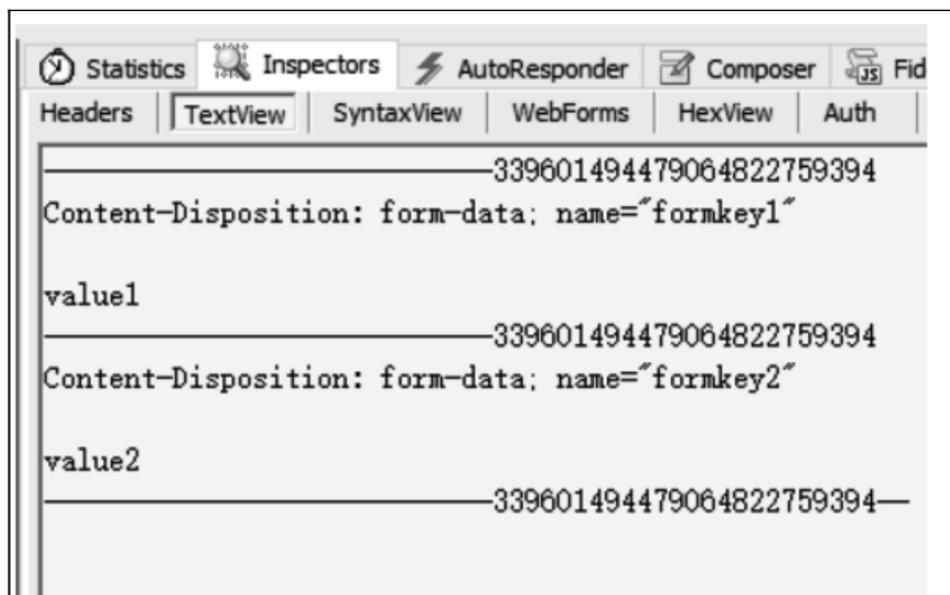


图9-26 Fiddler查看multipart/form-data类型的POST请求体

编码之后的Form表单项被同一个boundary分隔符分开，boundary分隔符的值则被包含在请求的Content-Type请求头的后半部分，处于“multipart/form-data”的后面，具体如图9-27所示。

The screenshot shows the Fiddler application interface with the 'Headers' tab selected. The 'Request Headers' section displays the following information:

- Cache**: Cache-Control: no-cache
- Client**: Accept: */*, Accept-Encoding: gzip, deflate, br, User-Agent: PostmanRuntime/7.25.0
- Entity**: Content-Length: 282
- Miscellaneous**: foo: bar, Postman-Token: 8c64e56c-7730-4c0f-9d7d-e5f497008ef9
- Transport**: Connection: keep-alive, Host: crazydemo.com:18899

Below the headers, the URL is listed as POST /getrequest?param1=value1¶m2=value2 HTTP/1.1.

图9-27 Fiddler查看处于Content-Type请求头的后半部分boundary的分隔符值

实验3：发送text/plain、application/json等编码类型的请求体

除了上面介绍的请求体的两种表单编码类型，使用Postman还可以提交Content-Type为text/plain、application/json等类型的POST请求体。在这种情况下，需要在其操作界面的Body类型选项中选择原始请求体类型，然后进一步选择Text、JSON或其他细分类型的原始内容类型，具体如图9-28所示。

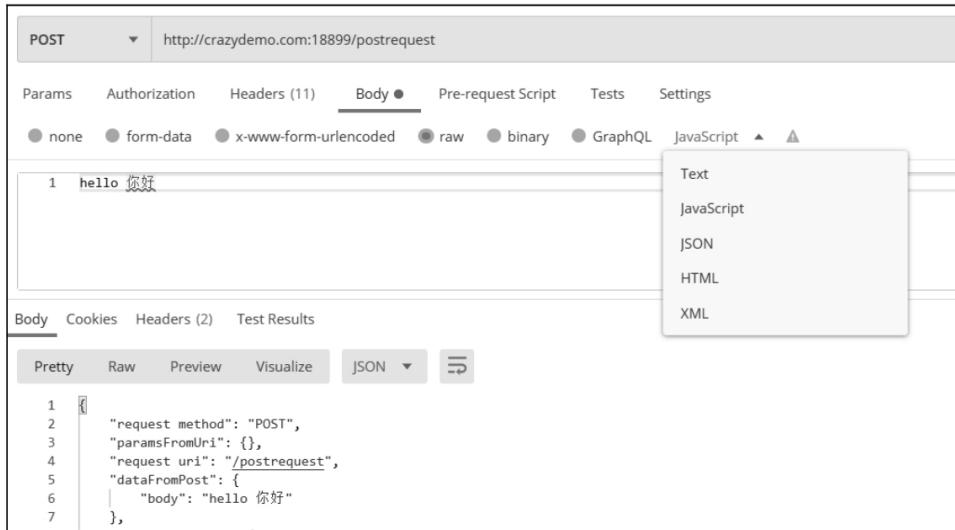


图9-28 使用Postman发送Body为Raw类型的POST请求体

大家可以自行通过Postman工具发送不同类型的请求体，并且可以通过抓包工具Fiddler去观察HTTP报文的变化。具体的实验过程，这里不再赘述。

说明

如果能够顺利掌握HTTP回显服务器程序，就可以开启下一个进阶实验：参考疯狂创客圈的spring-boot-netty-server开源项目，在Spring Boot、Spring Cloud应用中，使用Netty来替换Tomcat、Jetty、Undertow等传统的Web容器，练习比较复杂的基于Netty的服务端编程。有关该开源项目的实战交流，具体可以参见疯狂创客圈社群博客。

第10章 高并发HTTP通信的核心原理

HTTP是应用层协议，是建立在传输层TCP基础之上的。在通信过程中，TCP每一次连接的建立和拆除都会经历三次握手和四次挥手，性能和效率是比较低的。HTTP一个显著的特点是无状态的，并且最初的设计初衷是用于短连接场景，请求时建连接、请求完释放连接，以尽快将释放服务资源供其他客户端使用。这就导致每一次原始HTTP协议的传输都需要进行连接的建立和拆除，从而导致性能比较低。

10.1 需要进行HTTP连接复用的高并发场景

在客户端与服务器之间，使用HTTP短连接对用户体验和整体性能的影响是不大的，毕竟单个用户的请求频率不会太高。所以，HTTP短连接有自己的适用场景，在单个客户端与服务器通信不频繁的场景下，短连接的性能还是很高的。

随着微服务技术的发展，分布式应用的内部会存在大量、高频率的内部RPC调用或者HTTP通信。在这些场景下，如果频繁地进行传输层TCP连接的建立和拆除，就会减低整体的效率、拖慢整体的性能。要提高服务端之间的HTTP通信性能，就需要使用到HTTP连接复用技术。

在Java分布式应用的架构和实现过程中，涉及HTTP连接复用的高并发场景，大致有以下几种：

- (1) 反向代理Nginx到Java Web应用服务之间的HTTP高并发通信。
- (2) 微服务网关与微服务Provider实例之间的HTTP高并发通信。
- (3) 分布式微服务Provider实例与Provider实例之间RPC远程调用的HTTP高并发通信。
- (4) Java通过HTTP客户端访问REST接口服务的HTTP高并发通信。

10.1.1 反向代理Nginx与Java Web应用服务之间的HTTP高并发通信

传统的Nginx + Tomcat架构的Web应用一般使用Tomcat作为Web服务器，在并发访问量上升之后会引入Nginx作为接入层反向代理服务器，利用其负载均衡的能力，将请求代理分发到多个上游Web服务器。

一个较为简单的传统Nginx + Tomcat架构的示例如图10-1所示。该架构可以通过Web服务器的横向扩展甚至反向代理的分层扩展，从而使得系统具备高并发的能力。

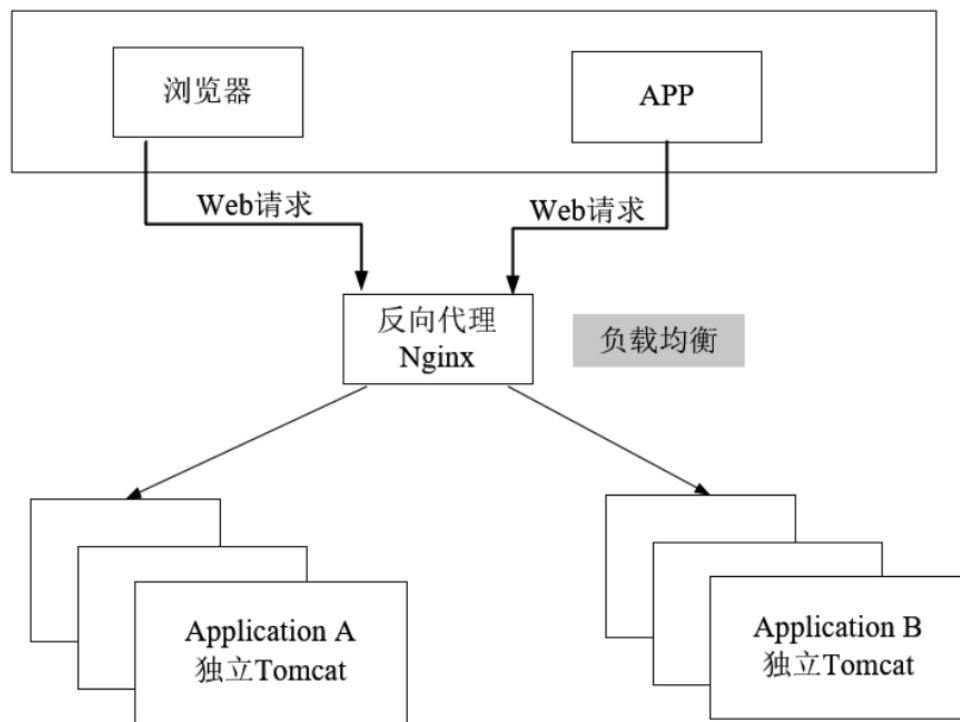


图10-1 一个经典的Nginx + Tomcat架构系统示例

传统的Nginx + Tomcat架构中，在Nginx和Tomcat之间进行方向代理请求转发时，对性能和速度的要求是很高的，此时需要HTTP下层的TCP连接通道具备可复用的能力，以提升响应效率和高并发能力。

10.1.2 微服务网关与微服务Provider实例之间的HTTP高并发通信

一个经典的分布式微服务应用架构如图10-2所示。

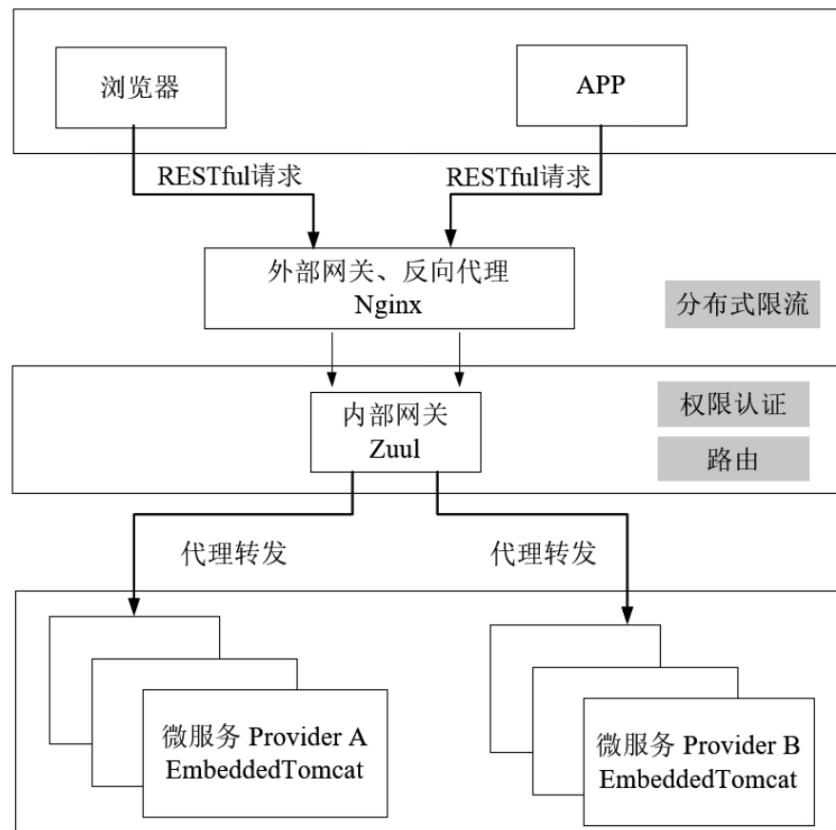


图10-2 一个经典的Nginx + Spring Cloud分布式架构示例

说明

分布式微服务架构已经成为Java应用的主流架构。一般来说，分布式微服务架构的接入层会引入Nginx反向代理，所以应用在整体

上常常是Nginx + Spring Cloud架构。有关该架构的原理知识，具体请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

在使用Nginx + Spring Cloud微服务架构的应用中，外部接入网关Nginx与内部网关Zuul（或Spring Cloud Gateway）之间，以及内部网关与微服务Provider实例之间，都存在着HTTP请求的反向代理（或者请求转发）的关系。以上不同的架构角色之间的HTTP通信和传输对性能要求都比较高，所以在微服务网关与微服务Provider之间的HTTP高并发通信的场景中需要HTTP传输层的连接通道具备可复用的能力，以提升高并发能力。

10.1.3 分布式微服务Provider实例之间的RPC的HTTP高并发通信

在微服务架构中，微服务Provider实例之间的RPC（具体见图10-3）也是通过HTTP完成的。RPC调用对性能和速度的要求是比较高的，需要HTTP下层TCP传输层的连接通道具备可复用的能力，以提升响应效率和高并发能力。

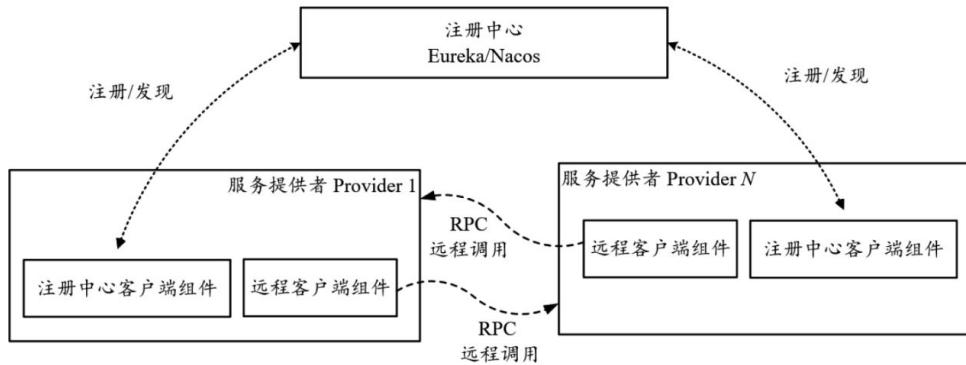


图10-3 Provider微服务实例之间的RPC远程调用

10.1.4 Java通过HTTP客户端访问REST接口服务的HTTP高并发通信

在实际开发中，Java应用通常会涉及对ESB企业服务总线注册的REST接口服务，或者Java应用会涉及对其他独立REST接口服务的访问。一般情况下，Java应用会使用本地HTTP客户端发起请求，从而获得REST访问结果。

在这种场景下，本地HTTP客户端和远程REST接口服务之间需要进行频繁的HTTP通信。显而易见，Java客户端与REST接口服务之间的HTTP通信时需要下层TCP传输层的连接通道具备可复用的能力，以提升请求效率和速度。

总结起来，实际需要复用HTTP连接的高并发通信场景远不止上面介绍的四种。客观地讲只要是在进行HTTP通信的两端之间通信和交互的频率高，就都需要具备连接复用的能力，都属于需要复用HTTP连接的场景。

HTTP连接复用实质上指的是承载HTTP报文的传输层TCP连接的复用。为什么要进行TCP连接的复用呢？原因是TCP建立连接和拆除连接的效率很低。那么，是什么原因导致TCP建立连接和拆除连接的效率较低呢？要彻底弄清楚这个问题，还是从传输层TCP协议的基础知识讲起。

10.2 详解传输层TCP

TCP/IP包含了一系列协议，也叫TCP/IP协议族（TCP/IP Protocol Suite，或TCP/IP Protocols），简称TCP/IP。TCP/IP协议族提供了点对点的连接机制，并且将传输数据帧的封装、寻址、传输、路由以及接收方式都予以标准化。

10.2.1 TCP/IP的分层模型

在展开介绍TCP/IP协议之前，首先介绍一下七层ISO模型。国际标准化组织ISO为了使网络应用更为普及，推出了OSI参考模型，即开放式系统互联（Open System Interconnect）模型，一般都叫OSI参考模型。OSI参考模型是ISO组织在1985年发布的网络互连模型，其含义就是为所有公司使用一个统一的规范来控制网络，这样所有公司遵循相同的通信规范，网络就能互联互通了。

OSI模型定义了网络互连的七层框架（物理层、数据链路层、网络层、传输层、会话层、表示层、应用层），每一层实现各自的功能和协议，并完成与相邻层的接口通信。OSI模型各层的通信协议，大致如表10-1所示。

表10-1 OSI模型各层的通信协议

七层框架		通信协议
应用层	HTTP、SMTP、SNMP、FTP、Telnet、SIP、SSH、NFS、RTSP、XMPP、Whois、ENRP 等	
表示层	XDR、ASN.1、SMB、AFP、NCP 等	
会话层	ASAP、SSH、RPC、NetBIOS、ASP、Winsock、BSD Sockets 等	
传输层	TCP、UDP、TLS、RTP、SCTP、SPX、ATP、IL 等	
网络层	IP、ICMP、IGMP、IPX、BGP、OSPF、RIP、IGRP、EIGRP、ARP、RARP、X.25 等	
数据链路层	以太网、令牌环、HDLC、帧中继、ISDN、ATM、IEEE 802.11、FDDI、PPP 等	
物理层	铜缆、网线、光缆、无线电等	

TCP/IP协议是互联网最基本的协议，在一定程度上参考了七层ISO模型，有些复杂，所以在TCP/IP协议中七层被简化为四个层次。TCP/IP模型中的各种协议依功能不同被归属到四层之中，常被视为简化后的七层OSI模型。TCP/IP协议与七层ISO模型的对应关系大致如图10-4所示。



图10-4 TCP/IP协议与七层ISO模型的对应关系

1. TCP/IP协议的应用层

应用层包括所有和应用程序协同工作并利用基础网络交换应用程序的业务数据的协议。一些特定的程序被认为运行在这个层上，该层协议所提供的服务能直接支持用户应用。应用层协议包括HTTP（万维网服务）、FTP（文件传输）、SMTP（电子邮件）、SSH（安全远程登录）、DNS（域名解析）以及许多其他协议。

2. TCP/IP协议的传输层

传输层的协议解决了端到端可靠性等问题，能确保数据可靠地到达目的地，甚至能保证数据按照正确的顺序到达目的地。传输层的主要功能大致如下：

- (1) 为端到端连接提供传输服务。
- (2) 这种传输服务分为可靠和不可靠的，其中TCP是典型的可靠传输、UDP是不可靠传输。
- (3) 为端到端连接提供流量控制、差错控制、服务质量 (Quality of Service, QoS) 等管理服务。

传输层主要有两个性质不同的协议：TCP（传输控制协议）和 UDP（用户数据报协议）。

TCP是一个面向连接的、可靠的传输协议，提供一种可靠的字节流，能保证数据完整、无损并且按序到达。TCP尽量连续不断地测试网络的负载并且控制发送数据的速度以避免网络过载。另外，TCP试图将数据按照规定的顺序发送。

UDP是一个无连接的数据报协议，是一个“尽力传递”和“不可靠”协议，不会对数据包是否已经到达目的地进行检查，并且不保证数据包按顺序到达。

总体来说，TCP传输效率低，但可靠性强；UDP传输效率高，但可靠性略低，适用于传输可靠性要求不高、体量小的数据（比如QQ聊天数据）。

3. TCP/IP协议的网络层

TCP/IP协议网络层的作用是在复杂的网络环境中为要发送的数据报找到一个合适的路径进行传输。简单来说，网络层负责将数据传输到目标地址，目标地址可以是多个网络通过路由器连接而成的某一个地址。另外，网络层负责寻找合适的路径到达对方计算机，并把数据帧传送给对方，网络层还可以实现拥塞控制、网际互联等功能。网络层协议的代表包括ICMP、IP、IGMP等。

4. TCP/IP协议的链路层

链路层有时也称作数据链路层或网络接口层，用来处理连接网络的硬件部分。该层既包括操作系统硬件的设备驱动、NIC（网卡）、光纤等物理可见部分，也包括连接器等一切传输媒介。在这一层中，数据的传输单位为比特。其主要协议有ARP、RARP等。

10.2.2 HTTP报文传输原理

利用TCP/IP进行网络通信时，数据包会按照分层顺序与对方进行通信。发送端从应用层往下走，接收端从链路层往上走。从客户端到

服务器的数据，每一帧的传输顺序都为应用层→传输层→网络层→链路层→链路层→网络层→传输层→应用层。

以一个HTTP请求的传输为例，请求从HTTP客户端（如浏览器）和HTTP服务端应用的传输过程大致如图10-5所示。

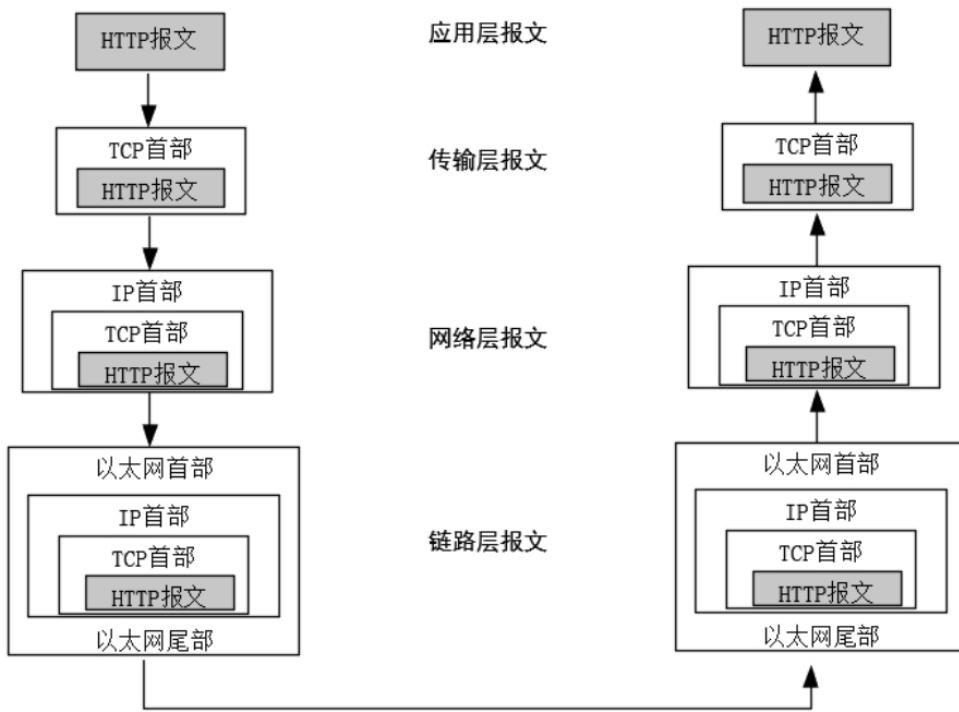


图10-5 HTTP请求报文的分层传输过程

接下来为大家介绍一下数据封装和分用。

数据通过互联网传输的时候不可能是光秃秃的不加标识（数据会乱），所以在发送数据的时候需要加上特定标识（数据封装），在使用数据的时候再去掉特定标识（数据分用）。TCP/IP协议的数据封装和分用过程大致如图10-6所示。

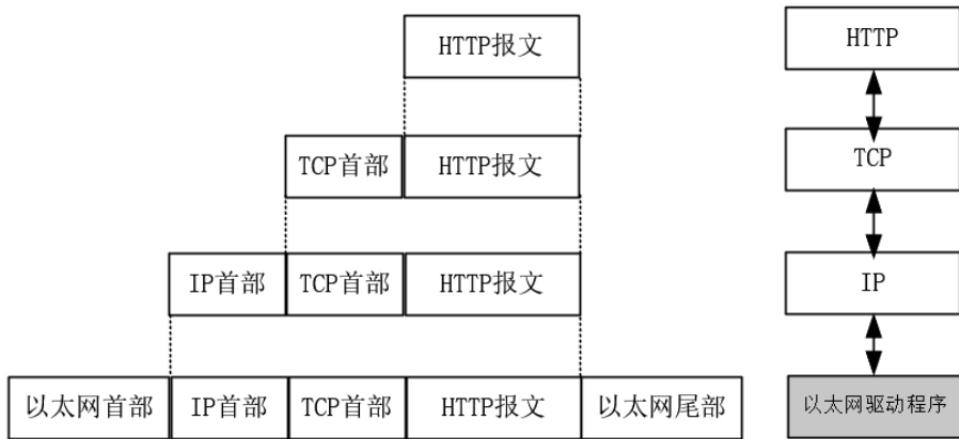


图10-6 TCP/IP协议的数据封装和分用过程

在数据封装时，数据经过每个层都会打上该层特定标识，添加上头部。

在传输层封装时，添加的报文首部要存入一个应用程序的标识符，无论是TCP还是UDP都用一个16位的端口号来表示不同的应用程序，并且都会将源端口和目的端口存入报文首部中。

在网络层封装时，IP首部会标识处理数据的协议类型，或者说标识出网络层数据帧所携带的上层数据类型，如TCP、UDP、ICMP、IP、IGMP等。具体来说，会在IP首部存入一个长度为8位的数值，称作协议域：1表示ICMP协议、2表示IGMP协议、6表示TCP协议、17表示UDP协议……IP首部还会标识发送方地址（源IP）和接收方地址（目标IP）。

在链路层封装时，网络接口分别要发送和接收IP、ARP和RARP等多种不同协议的报文，因此也必须在以太网的帧首部加入某种形式的标识，以指明所处理的协议类型。为此，以太网报文帧的首部也有一个

16位的类型字段，标识出以太网数据帧所携带的上层数据类型，如IPv4、ARP、IPV6、PPPoE等。

数据封装和分用的过程大致为：发送端每通过一层就增加该层的首部，接收端每通过一层就删除该层的首部。

总体来说，TCP/IP分层管理、数据封装和分用的好处是：分层之后若需改变相关设计，只需替换变动的层。各层之间的接口部分规划好之后，每个层次内部的设计就可以自由改动了。层次化之后，设计也变得相对简单：各个层只需考虑分派给自己的传输任务即可。

TCP/IP除了与OSI在分层模块上稍有区别外，更重要的区别是：OSI参考模型注重“通信协议必要的功能是什么”，而TCP/IP则更强调“在计算机上实现协议应该开发哪种程序”。

实际上，在传输过程中，数据报文会在不同的物理网络之间传递。还是以一个HTTP请求的传输为例，请求在不同物理网络之间的传输过程大致如图10-7所示。

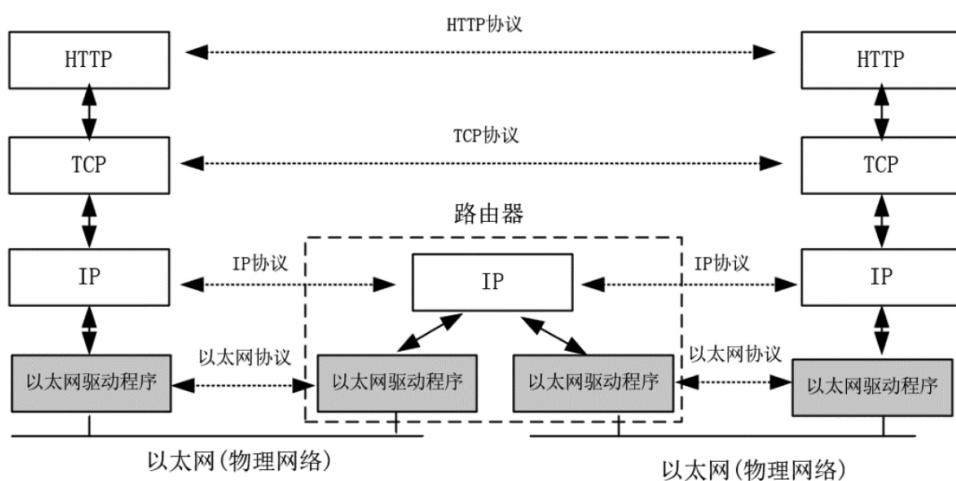


图10-7 HTTP请求在不同物理网络之间的传输过程

在不同物理网络之间的传输过程中，网络层会通过路由器对不同网络之间的数据包进行存储、分组转发处理。构造互联网最简单的方法是把两个或多个网络通过路由器进行连接。路由器可以简单理解为一种特殊的用于网络互连的硬件盒，其作用是为不同类型的物理网络提供连接：以太网、令牌环网、点对点的链接和FDDI（光纤分布式数据接口）等。

物理网络之间通过路由器进行互连，随着增加不同类型的物理网络，可能会有很多个路由器，但是对于应用层来说仍然是一样的，TCP协议栈为大家屏蔽了物理层的复杂性。总之，物理细节和差异性的隐藏使得互联网TCP/IP传输的功能变得非常强大。

接下来开始为大家介绍与传输性能有密切关系的内容：TCP传输层的三次握手建立连接，四次挥手释放连接。不过在此之前，还得先介绍一下TCP报文协议。

10.2.3 TCP的报文格式

在TCP/IP协议栈中，IP层只关心如何使数据能够跨越本地网络边界的问题，而不关心数据如何传输。TCP/IP协议栈配合起来解决数据如何通过许许多多个点对点通路顺利传输到目的地。一个点对点通路被称为一“跳”（hop），通过TCP/IP协议栈，网络成员能够在许多“跳”的基础上建立相互的数据通路。

传输层TCP提供了一种面向连接的、可靠的字节流服务，其数据帧格式大致如图10-8所示。

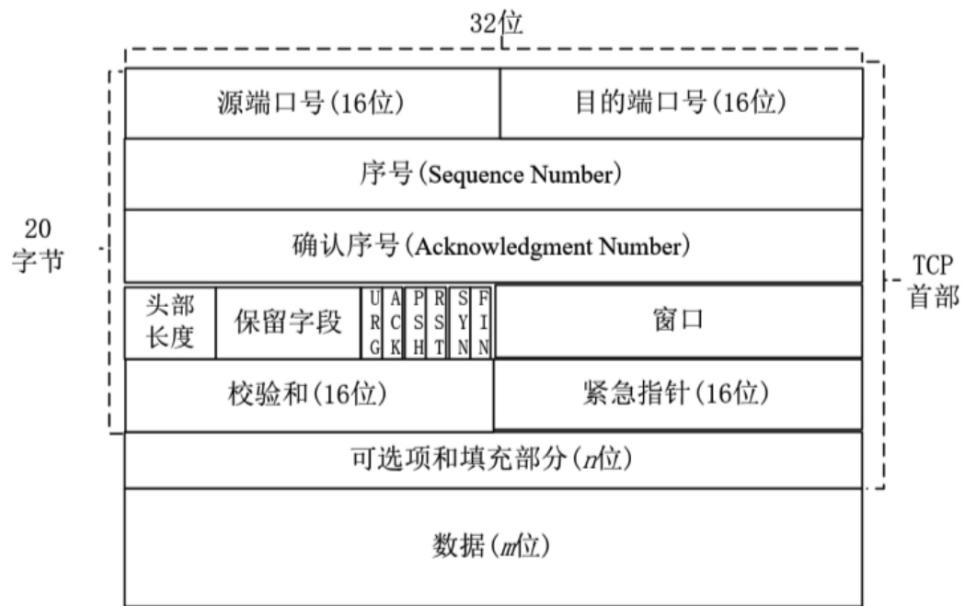


图10-8 传输层TCP的数据帧格式

一个传输层TCP的数据帧大致包含以下字段：

(1) 源端口号

源端口号表示报文的发送端口，占16位。源端口号和源IP地址组合起来，可以标识报文的发送地址。

(2) 目的端口号

目的端口号表示报文的接收端口，占16位。目的端口号和目的IP地址相结合，可以标识报文的接收地址。

TCP是在IP基础上传输的，TCP报文中的“源端口号+源IP”与“目的端口号+目的IP”组合起来唯一确定一条TCP连接。

(3) 序号

TCP传输过程中，在发送端发出的字节流中，传输报文中的数据部分的每一个字节都有它的编号。序号（Sequence Number）占32位，发起方发送数据时，都需要标记序号。

序号的语义与SYN控制标志（Control Bits）的值有关。根据控制标志（Control Bits）中的SYN来表达不同的序号含义：

- 当 $SYN = 1$ 时，当前为连接建立阶段，序号为初始序号ISN（Initial Sequence Number），通过算法来随机生成序号。
- 当 $SYN = 0$ 时，在数据传输正式开始时，第一个报文的序号为 $ISN + 1$ ，后面的报文序号为前一个报文的SN值 + TCP报文的净荷字节数（不包含TCP头）。如果发送端发送的一个TCP帧的净荷为12B，序号为5，则发送端接着发送下一个数据包时，序号的值应该设置为 $5 + 12 = 17$ 。

在数据传输过程中，TCP通过序号（Sequence Number）对上层提供有序的数据流。发送端可以用序号来跟踪发送的数据量；接收端可以用序号识别出重复接收到的TCP包，从而丢弃重复包；对于乱序的数据包，接收端也可以依靠序号对其进行排序。

（4）确认序号

确认序号（Acknowledgment Number）标识了报文接收端期望接收的字节序列。如果设置了ACK控制位，确认序号的值表示一个准备接收的包的序列码。注意，它所指向的是准备接收的包，也就是下一个期望接收的包的序列码。

举个例子，假设发送端（如Client）发送3个净荷为1000B、起始SN序号为1的数据包给服务端，服务端每收到一个包之后，需要回复一个ACK响应确认数据包给客户端。ACK响应数据包的ACK Number值为每个客户端包的SN+包净荷，除了表示服务端已经确认收到的字节数，还表示期望接收到的下一个客户端发送包的SN序号。具体的ACK值如图10-9左边的正常传输部分所示。

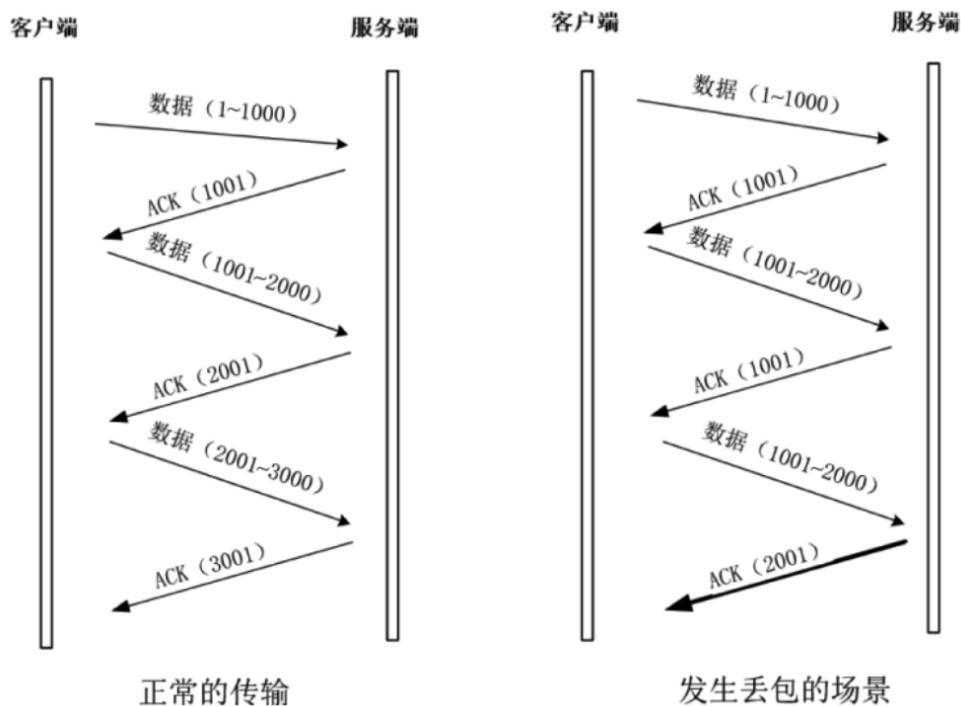


图10-9 传输过程的确认序号值示例图

在图10-9的左边部分，服务端第1个ACK包的ACK Number值为1001，是通过客户端第1个包的SN+包净荷（1+1000）计算得到，表示期望第2个客户端包的SN序号为1001；服务端第2个ACK包的ACK Number值为2001，为客户端第2个包的SN+包净荷（2000+1），表示期望第3个服务端包的SN为2001，以此类推。

如果发生错误，假设服务端在处理客户端的第二个发送包时发生异常，服务端仍然回复一个ACK Number值为1001的确认包，则客户端的第二个数据包需要重复发送，具体的ACK值如图10-9右边的正常传输部分所示。

只有控制标志的ACK标志为1时，数据帧中的确认序号ACK Number才有效。TCP规定，连接建立后，所有发送报文的ACK必须为1，即建立连接后所有报文的确认序号有效。如果是SYN类型的报文，其ACK标志为0，故没有确认序号。

(5) 头部长度

该字段占用4位，用来表示TCP报文首部的长度，单位是4位。其值所表示的并不是字节数，而是头部所含有的32位的数目（或者倍数），或者4字节的倍数，所以TCP头部最多可以有60字节 $(4 \times 15 = 60)$ 。没有任何选项字段的TCP头部长度为20字节，所以其头部长度为5，可以通过 $20/4=5$ 计算得到。

(6) 保留字段

头部长度后面预留的字段长度为6位，作为保留字段，暂时没有什么用处。

(7) 控制标志

控制标志（Control Bits）共6位，具体的标志位为URG、ACK、PSH、RST、SYN、FIN，如表10-2所示。

表10-2 TCP报文控制标志说明

标 志 位	说 明
URG	占 1 位，表示紧急指针字段有效。URG 位指示报文段里的上层实体（数据）标记为“紧急”数据。当 URG=1 时，其后的紧急指针指示紧急数据在当前数据段中的位置（相对于当前序列号的字节偏移量），TCP 接收方必须通知上层实体
ACK	占 1 位，置位 ACK=1 表示确认号字段有效。TCP 规定，连接建立后所有发送的报文的 ACK 必须为 1。当 ACK=0 时，表示该数据段不包含确认信息。当 ACK=1 时，表示该报文段包括一个对已被成功接收报文段的确认序号 Acknowledgment Number，该序号同时也是下一个报文的预期序号
PSH	占 1 位，表示当前报文需要请求推（push）操作；当 PSH=1 时，接收方在收到数据后立即将数据交给上层，而不是直到整个缓冲区满
RST	占 1 位，置位 RST=1 表示复位 TCP 连接；用于重置一个已经混乱的连接，也可用于拒绝一个无效的数据段或者拒绝一个连接请求。如果数据段被设置了 RST 位，就说明报文发送方有问题发生
SYN	占 1 位，在连接建立时用来同步序号。当 SYN=1 而 ACK=0 时，表明这是一个连接请求报文。对方若同意建立连接，则应在响应报文中设置 SYN=1、ACK=1。综合一下，SYN 置为 1 就表示这是一个连接请求或连接接受报文
FIN	占 1 位，用于在释放 TCP 连接时标识发送方比特流结束，用来释放一个连接。当 FIN = 1 时，表明此报文的发送方数据已经发送完毕，并要求释放连接

在连接建立的三次握手过程中，若只是单个SYN置位，则表示的是建立连接请求。如果SYN和ACK同时置位为1，就表示建立连接之后的响应。

（8）窗口

长度为16位，共2字节，用来进行流量控制。流量控制的单位为字节数，这个值是本端期望一次接收的字节数。

（9）校验和

长度为16位，共2字节。对整个TCP报文段，即TCP头部和TCP数据进行校验和计算，接收端用于对收到的数据包进行验证。

（10）紧急指针

长度为16位，共2字节，是一个偏移量，和SN序号值相加表示紧急数据最后一个字节的序号。

以上10项内容是TCP报文首部必需的字段，也称固有字段，长度为20字节。接下来是TCP报文的可选项和填充部分。

（11）可选项和填充部分

可选项和填充部分的长度为 $4n$ 字节（ n 是整数），是根据需要而增加的选项。如果不足 $4n$ 字节，则要加填充位，使得选项长度为32位（4字节）的整数倍，具体的做法是在这个字段中加入额外的零，以确保TCP头是32位（4字节）的整数倍。

常见的选项字段是MSS（Maximum Segment Size，最长报文大小），每个连接方通常都在通信的第一个报文段（SYN标志为1的那个段）中指明这个选项字段，表示当前连接方所能接受的最大报文段的长度。

由于可选项和填充部分不是必需的，因此TCP报文首部最小长度为20个字节。

至此，TCP报文首部的字段就全部介绍完了。TCP报文首部的后面接着的是数据部分，不过数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有TCP首部。如果一方没有数据要发送，就使用没有任何数据的首部来确认收到的数据，比如在处理超时的过程中，也会发送不带任何数据的报文段。

总体来说，TCP的可靠性主要通过以下几点来保障：

① 应用数据分割成TCP认为最适合发送的数据块。这部分是通过MSS（最大数据包长度）选项来控制的，通常被称为一种协商机制。MSS规定了TCP传往另一端的最大数据块的长度。值得注意的是，MSS只能出现在SYN报文段中，若一方不接收来自另一方的MSS值，则MSS就定为536字节。一般MSS值越大越好，以提高网络的利用率。

② 重传机制。设置定时器，等待确认包，如果定时器超时还没有收到确认包，则报文重传。

③ 对首部和数据进行校验。

④ 接收端对收到的数据进行排序，然后交给应用层。

⑤ 接收端丢弃重复的数据。

⑥ 提供流量控制，主要是通过滑动窗口来实现的。

至此，TCP的数据帧格式介绍完了。接下来开始为大家重点介绍TCP传输层的三次握手建立连接、四次挥手释放连接。

10.2.4 TCP的三次握手

TCP连接建立时，双方需要经过三次握手；断开连接时，双方需要经过四次挥手。

通常情况下，建立连接的双方由一端监听来自请求方的TCP（Socket）连接，当服务端监听开始时，必须准备好接受外来的连接，在Java中该操作通过创建一个ServerSocket服务监听套接字实例来完成。此操作会调用底层操作系统（如Linux）C代码中的三个函数

socket()、bind()、listen()来完成。开始监听之后，服务端就做好接受外来连接的准备，如果监听到建立新连接的请求，就会开启一个传输套接字，称之为被动打开（Passive Open）。

一段简单的服务端监听新连接请求，并且被动打开（Passive Open）传输套接字的Java示例代码，具体如下：

```
public class SocketServer {  
    public static void main(String[] args) {  
        try {  
            //创建服务端socket  
            ServerSocket serverSocket = new  
            ServerSocket(8080);  
  
            //循环监听等待客户端的连接  
            while(true) {  
                //监听到客户端连接，传输套  
                //接字被动开启  
                Socket socket =  
                serverSocket.accept();  
                //开启线程进行连接的IO处理  
                ServerThread thread =  
                new ServerThread(socket);  
                thread.start();  
                ...  
            }  
        } catch (Exception e) {
```

```
        //处理异常  
        e.printStackTrace();  
    }  
}  
}
```

客户端在发起连接建立时，Java代码通过创建Socket实例调用底层的connect()方法，主动打开（Active Open）Socket连接。套接字监听方在收到请求之后，和发起方（客户端）之间就会建立一条连接通道（由双方IP和双方端口唯一确定）。

一段简单的客户端连接主动打开的Java示例代码如下：

```
public class SocketClient {  
  
    public static void main(String[] args) throws  
InterruptedException {  
    try {  
        //和服务器创建连接  
        Socket socket = new  
        Socket("localhost", 8080);  
  
        //写入给监听方的输出流  
        OutputStream os =  
        socket.getOutputStream();  
        ...  
        //读取监听方的输入流  
        InputStream is =
```

```
socket.getInputStream();

...
} catch (Exception e) {
    e.printStackTrace();
}

}

}

}
```

TCP连接建立时，双方需要经过三次握手，具体过程如下：

(1) 第一次握手：Client进入SYN_SENT状态，发送一个SYN帧来主动打开传输通道，该帧的SYN标志位被设置为1，同时会带上Client分配好的SN序列号，该SN是根据时间产生的一个随机值，通常情况下每间隔4ms（毫秒）会加1。除此之外，SYN帧还会带一个MSS（最大报文段长度）可选项的值，表示客户端发送出去的最大数据块的长度。

(2) 第二次握手：Server在收到SYN帧之后，会进入SYN_RCVD状态，同时返回SYN+ACK帧给Client，主要目的在于通知Client“Server已经收到SYN消息，现在需要进行确认”。Server发出的SYN+ACK帧的ACK标志位被设置为1，其确认序号AN（Acknowledgment Number）值被设置为Client的SN+1；SYN+ACK帧的SYN标志位被设置为1，SN值为Server生成的SN序号；SYN+ACK帧的MSS（最大报文段长度）表示的是Server的最大数据块长度。

(3) 第三次握手：Client在收到Server的第二次握手SYN+ACK确认帧之后，首先将自己的状态从SYN_SENT变成ESTABLISHED，表示自己方向的连接通道已经建立成功，Client可以发送数据给Server了。然后，Client发ACK帧给Server，该ACK帧的ACK标志位被设置为1，其确

认序号AN（Acknowledgment Number）值被设置为Server的SN+1。还有一种情况，Client可能会将ACK帧和第一帧要发送的数据合并到一起发送给Server。

Server在收到Client的ACK帧之后会从SYN_RCV状态进入ESTABLISHED状态。至此，Server方向的通道连接建立成功。Server可以发送数据给Client，TCP的全双工连接建立完成。

三次握手的交互过程如图10-10所示。

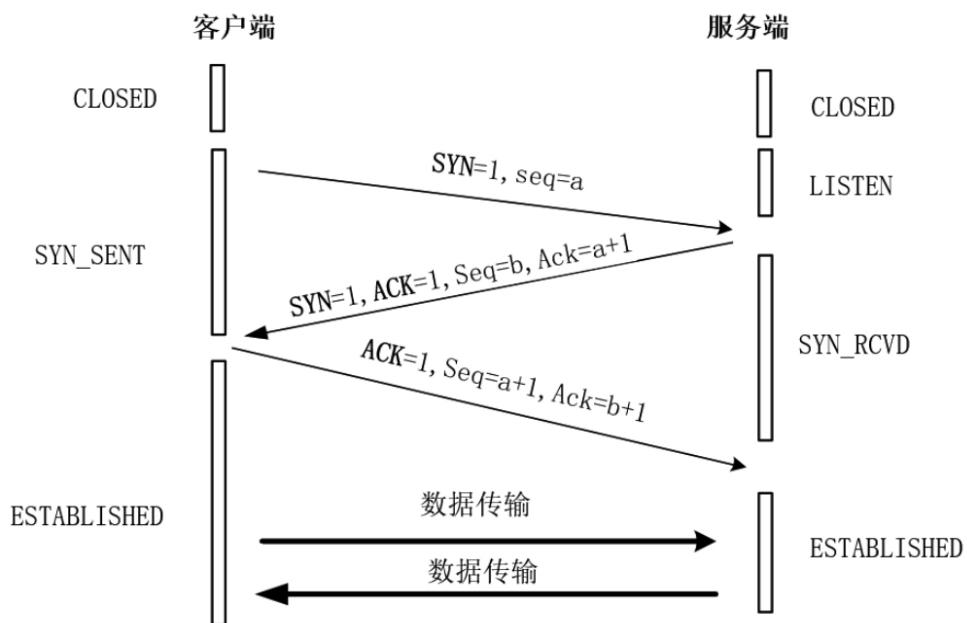


图10-10 TCP建立连接时的三次握手示意图

Client和Server完成了三次握手后，双方就进入数据传输阶段。数据传输完成后，连接将断开。连接断开的过程需要经历四次挥手。

10.2.5 TCP的四次挥手

在TCP连接开始断开（或者拆接）的过程中，连接的每个端都能独立、主动地发起。四次挥手的具体过程如下：

(1) 第一次挥手：主动断开方（可以是客户端，也可以是服务端），向对方发送一个FIN结束请求报文，此报文的FIN位被设置为1，并且正确设置Sequence Number（序列号）和Acknowledgment Number（确认号）。发送完成后，主动断开方进入FIN_WAIT_1状态，表示主动断开方没有业务数据要发送给对方，准备关闭Socket连接了。

(2) 第二次挥手：正常情况下，在收到了主动断开方发送的FIN断开请求报文后，被动断开方会发送一个ACK响应报文，报文的Acknowledgment Number（确认号）值为断开请求报文的Sequence Number（序列号）加1，该ACK确认报文的含义是：“我同意你的连接断开请求”。之后，被动断开方就进入了CLOSE-WAIT（关闭等待）状态，TCP服务会通知高层的应用进程，对方向本地方向的连接已经关闭，已经没有数据要发送了，若本地还要发送数据给对方，对方依然会接收。被动断开方的CLOSE-WAIT（关闭等待）还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

主动断开方在收到了ACK报文后，由FIN_WAIT_1转换成FIN_WAIT_2状态。

(3) 第三次挥手：在发送完成ACK报文后，被动断开方还可以继续完成业务数据的发送，待剩余数据发送完成或者CLOSE-WAIT（关闭等待）截止后，被动断开方会向主动断开方发送一个FIN+ACK结束响应报文，表示被动断开方的数据都发送完了，然后被动断开方进入LAST_ACK状态。

(4) 第四次挥手：主动断开方收到FIN+ACK断开响应报文后，还需要进行最后的确认，向被动断开方发送一个ACK确认报文，然后自己就进入TIME_WAIT状态，等待超时后最终关闭连接。处于TIME_WAIT状态的主动断开方在等待完成2MSL的时间后，如果期间没有收到其他报文，则证明对方已正常关闭，主动断开方的连接最终关闭。

被动断开方在收到主动断开方的最后ACK报文以后，最终关闭连接，什么也不用管了。

四次挥手的全部交互过程如图10-11所示。

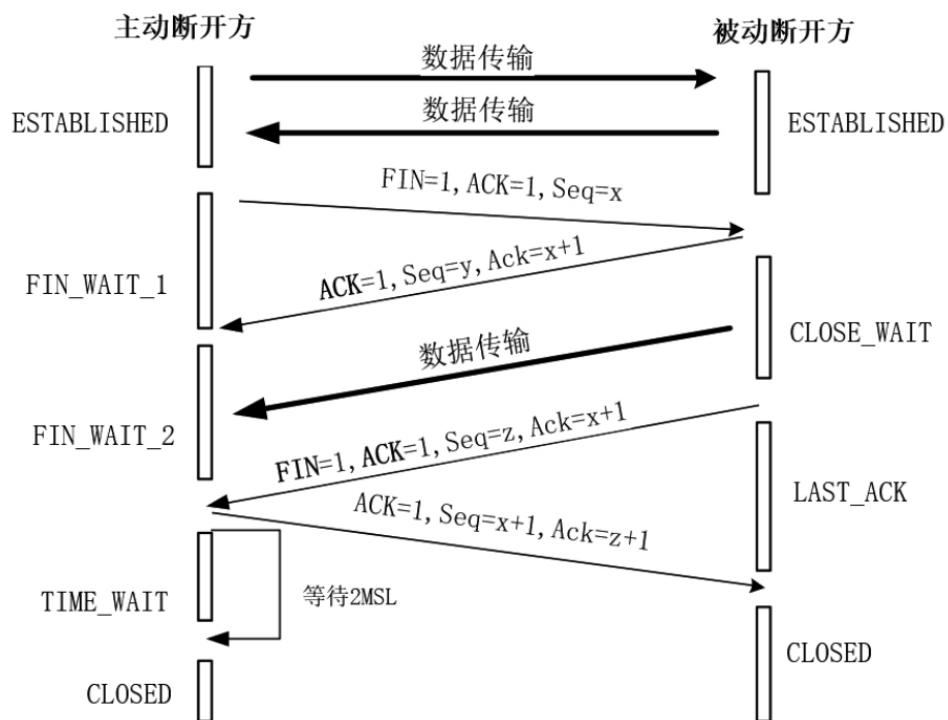


图10-11 TCP断开连接时的四次挥手示意图

处于TIME_WAIT状态的主动断开方在等待2MSL时间后才真正关闭连接通道。2MSL翻译过来就是两倍的MSL。MSL（Maximum Segment Lifetime）指的是一个TCP报文片段在网络中的最大存活时间，2MSL就

是一次消息来回（一个发送和一个回复）所需的最大时间。如果直到2MSL主动断开方都没有再一次收到对方的报文（如FIN报文），则可以推断ACK已经被对方成功接收。此时，主动断开方将最终结束自己的TCP连接。所以，TCP的TIME_WAIT状态也称为2MSL等待状态。

有关MSL的具体时间长度，在RFC1122协议中推荐为2分钟，在SICS（瑞典计算机科学院）开发的一个小型开源的TCP/IP协议栈——LwIP开源协议栈中默认为1分钟，在源自Berkeley的TCP协议栈实现中默认为30秒。总体来说，TIME_WAIT（2MSL）等待状态的时间长度一般维持在1~4分钟。

通过三次握手建立连接和四次挥手拆除连接，一次TCP的连接建立及拆除至少进行7次通信，可见其成本是很高的。

10.2.6 三次握手、四次挥手的常见面试题

TCP连接建立的三次握手及拆除过程的四次挥手的面试问题是技术面试过程中出现频率很高的重点和难点问题，常见问题大致如下：

问题（1）：为什么关闭连接时需要四次挥手，而建立连接却只要三次握手？

关闭连接时，被动断开方在收到对方的FIN结束请求报文时很可能没有发送完业务数据，并不能立即关闭连接，被动方只能先回复一个ACK响应报文，告诉主动断开方：“你发的FIN报文我收到了，只有等到我所有的业务报文都发送完了，我才能真正结束，在结束之前，我会发给你FIN+ACK报文的，你先等着”。所以，被动断开方的确认报文需要拆成两步，故总共需要四次挥手。

在建立连接场景中，Server的应答可以稍微简单一些。当Server收到Client的SYN连接请求报文后，其中ACK报文表示对请求报文的应答，SYN报文表示服务端的连接也已经同步开启了，而ACK报文和SYN报文之间不会有其他报文需要发送，故而可以合二为一，可以直接发送一个SYN+ACK报文。所以，在建立连接时，只需要三次握手即可。

问题（2）：为什么连接建立的时候是三次握手，可以改成两次握手吗？

三次握手完成两个重要的功能：一是双方都做好发送数据的准备工作，而且双方都知道对方已准备好；二是双方完成初始SN序列号的协商，双方的SN序列号在握手过程中被发送和确认。

如果把三次握手改成两次握手，可能发生死锁。两次握手的话，缺失了Client的二次确认ACK帧，假想的TCP建立连接时的二次挥手可以如图10-12所示。

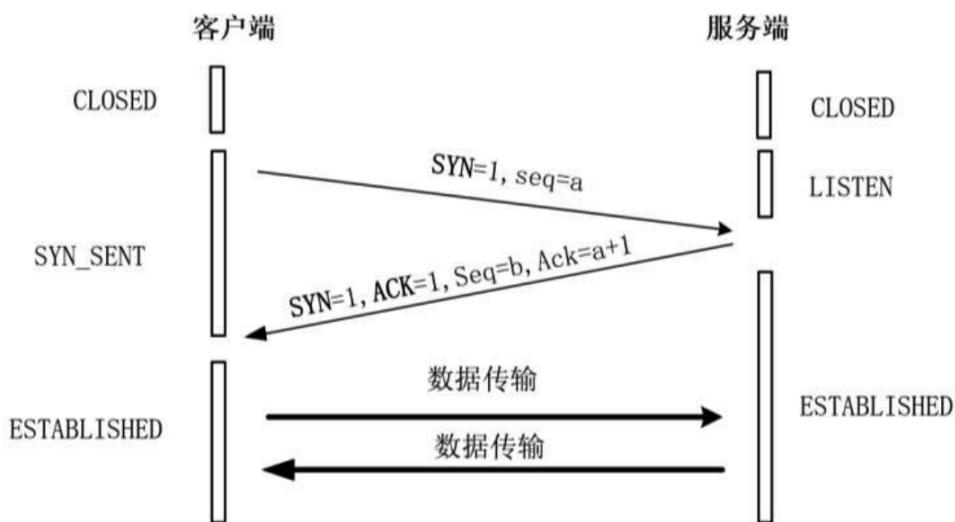


图10-12 假想的TCP建立连接时二次握手的示意图

在假想的TCP建立连接时的二次握手过程中，Client给Server发送一个SYN请求帧，Server收到后发送确认应答SYN+ACK帧。按照两次握手的协定，Server认为连接已经成功地建立，可以开始发送数据帧。在这个过程中，如果确认应答SYN+ACK帧在传输中被丢失，Client没有收到，Client将不知道Server是否已准备好，也不知道Server的SN序列号，Client认为连接还未建立成功，将忽略Server发来的任何数据分组，会一直等待Server的SYN+ACK确认应答帧。Server在发出的数据帧后，一直没有收到对应的ACK确认后就会产生超时，重复发送同样的数据帧。这样就形成了死锁。

问题（3）：为什么主动断开方在TIME-WAIT状态必须等待2MSL？

原因之一：主动断开方等待2MSL的时间是为了确保两端都能最终关闭。假设网络是不可靠的，被动断开方发送FIN+ACK报文后，其主动方的ACK响应报文有可能丢失，这时的被动断开方处于LAST-ACK状态，由于收不到ACK确认被动方一直不能正常地进入CLOSED状态。在这种场景下，被动断开方会超时重传FIN+ACK断开响应报文，如果主动断开方在2MSL时间内收到这个重传的FIN+ACK报文，就会重传一次ACK报文，然后再一次重新启动2MSL计时等待，这样就能确保被动断开方能收到ACK报文，从而能确保被动方顺利进入CLOSED状态。只有这样，双方才能够确保关闭。反过来说，如果主动断开方在发送完ACK响应报文后不是进入TIME_WAIT状态去等待2MSL时间，而是立即释放连接，则将无法收到被动方重传的FIN+ACK报文，所以不会再发送一次ACK确认报文，此时处于LAST-ACK状态的被动断开方无法正常进入CLOSED状态。

原因之二：防止“旧连接已失效的数据报文”出现在新连接中。主动断开方在发送完最后一个ACK报文后再经过2MSL才能最终关闭和释放端口。这就意味着，相同端口的新TCP新连接需要在2MSL的时间之后

才能够正常建立。2MSL这段时间内，旧连接所产生的所有数据报文都已经从网络中消失了，从而确保下一个新的连接中不会出现这种旧连接请求报文。

问题（4）：如果已经建立了连接，但是Client端突然出现故障了怎么办？

TCP还设有一个保活计时器，Client如果出现故障，Server不能一直等下去，这样会浪费系统资源。每收到一次Client的数据帧后，Server的保活计时器都会复位。计时器的超时时间通常设置为2小时，若2小时还没有收到Client的任何数据帧，Server就会发送一个探测报文段，以后每隔75秒发送一次。若一连发送10个探测报文仍然没有反应，Server就认为Client出了故障，接着关闭连接。如果觉得保活计时器的两个多小时的间隔太长，可以自行调整TCP连接的保活参数。

10.3 TCP连接状态的原理与实验

本节首先介绍TCP连接的11种状态，然后介绍查看连接状态的netstat指令。

10.3.1 TCP/IP连接的11种状态

TCP建立连接、传输数据和断开连接是一个复杂的过程，为了准确地描述这一过程，可以采用有限状态机来完成。有限状态机包含有限个状态，在某一时刻，连接必然处于某一特定状态，当在一个状态下发生特定事件时，连接会进入一个新的状态。

TCP连接的11种状态如下：

(1) LISTEN：表示服务端的某个ServerSocket监听连接处于监听状态，可以接受客户端的连接。

(2) SYN_SENT：这个状态与SYN_RCVD状态相呼应，当客户端Socket连接的底层开始执行connect()方法发起连接请求时，本地连接会进入SYN_SENT状态，并发送SYN报文，并等待服务端发送三次握手中的SYN+ACK报文。SYN_SENT状态表示客户端连接已发送SYN报文。

(3) SYN_RCVD：表示服务端ServerSocket接收到了来自客户端连接的SYN报文。在正常情况下，这个状态是ServerSocket连接在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用netstat指令很难看到这种状态，除非故意写一个监测程序，将三次

TCP握手过程中最后一个ACK报文不予发送。当TCP连接处于此状态时，再收到客户端的ACK报文，它就会进入ESTABLISHED状态。

(4) ESTABLISHED：表示TCP连接已经成功建立。

(5) FIN_WAIT_1：当连接处于ESTABLISHED状态时，想主动关闭连接时，主动断发会调用底层的close()方法，要求主动关闭连接，此时主动断开方进入FIN_WAIT_1状态。当对方回应ACK报文后，主动方进入FIN_WAIT_2状态。当然，在实际的正常情况下，无论对方处于任何种情况，都应该马上回应ACK报文，所以FIN_WAIT_1状态一般是比较难见到的，而FIN_WAIT_2状态有时仍可以用netstat指令看到。

(6) FIN_WAIT_2：主动断开方处于FIN_WAIT_1状态后，如果收到对方的ACK报文，主动方会进入FIN_WAIT_2状态，此状态下的双向通道处于半连接（半开）状态，即被动方还可以传递数据过来，但主动方不可再发送数据出去。需要注意的是，FIN_WAIT_2是没有超时的（不像TIME_WAIT状态），这种状态下如果对方不发送FIN+ACK关闭响应

（不配合完成4次挥手过程），那么FIN_WAIT_2状态将一直保持，该连接会一直被占用，资源不会被释放，越来越多处于FIN_WAIT_2状态的半连接堆积会导致操作系统内核崩溃。

(7) TIME_WAIT：该状态表示主动断开方已收到了对方的FIN+ACK关闭响应，并发送出ACK报文。TIME_WAIT状态下的主动方TCP连接会等待2MSL的时间，然后回到CLOSED状态。如果FIN_WAIT_1状态下收到了对方同时带FIN+ACK关闭响应报文，就可以直接进入TIME_WAIT状态，而无须经过FIN_WAIT_2状态。这种情况下，四次挥手变成三次挥手。

(8) CLOSING: 这种状态在实际情况中很少见，属于一种比较罕见的例外状态。正常情况下，当一方发送FIN报文后，理论上应该先收到对方的ACK报文再收到对方的FIN+ACK关闭响应报文，或同时收到。CLOSING状态表示一方发送FIN报文后并没有收到对方的ACK报文，却也收到了对方的FIN报文。什么情况下会出现此种情况呢？当双方几乎在同时发出close()双向连接时，就会出现双方同时发送FIN报文的情况，这时就会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

(9) CLOSE_WAIT: 表示正在等待关闭。在主动断开方调用close()方法关闭一个连接后，主动方会发送FIN报文给被动态，被动态在收到之后会回应一个ACK报文给主动方，回复完成之后，被动态的TCP连接进入CLOSE_WAIT状态。接下来，被动态需要检查是否还有数据要发送给主动方，如果没有，就意味着被动态也就可以关闭连接了，此时给主动方发送FIN+ACK报文，即关闭自己到对方这个方向的连接。简单地说，当连接处于CLOSE_WAIT状态下，可以继续传输数据，传输完成之后关闭连接。

(10) LAST_ACK: 当被动态断开方发送完FIN+ACK确认断开之后，就处于LAST_ACK状态，等待主动断开方的最后一个ACK报文。当收到对方的ACK报文后，被动态关闭方也就可以进入CLOSED可用状态了。

(11) CLOSED: 关闭状态或者初始状态，表示TCP连接是“关闭着的”或“未打开的”状态，或者说连接是可用的。

10.3.2 通过netstat指令查看连接状态

netstat是一款命令行工具，用于列出系统中所有TCP/IP的连接情况，包括TCP、UDP以及UNIX套接字，而且该工具也能列出处于监听状态的服务端监听套接字。

总体来说，netstat是一个非常有用的工具，可以用于查看路由表、实际的网络连接甚至每一个网络接口设备的状态信息。举个例子，使用netstat -ant指令查看当前Linux系统中所有的TCP/IP网络的连接信息，大致结果如下：

```
[root@localhost ~]# netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address
State
tcp      0      0      127.0.0.1:57144        127.0.0.1:4369
TIME_WAIT
tcp6     0      0      :::18899             ::::*
LISTEN
tcp6     0      0      :::22                ::::*
LISTEN
tcp6     0      0      127.0.0.1:40748        127.0.0.1:2889
ESTABLISHED
tcp6     0      0      192.168.233.128:37806
192.168.233.128:8848  ESTABLISHED
tcp6     0      0      127.0.0.1:2889        127.0.0.1:40748
ESTABLISHED
tcp6     0      0      192.168.233.128:8848
192.168.233.128:37806  ESTABLISHED
```

```
tcp6      0      0    192.168.233.128:34880  
192.168.233.128:7777  ESTABLISHED  
tcp6      0      0    192.168.233.128:7777  
192.168.233.128:34880  ESTABLISHED  
tcp6      0      0    127.0.0.1:3888          127.0.0.1:56316  
ESTABLISHED  
tcp6      0      0    192.168.233.128:18899  192.168.233.1:12405  
ESTABLISHED  
tcp6      0      0    127.0.0.1:56316          127.0.0.1:3888  
ESTABLISHED  
...
```

对以上netstat -ant命令展示结果中的列，具体的介绍如下：

(1) Proto列表示套接字所使用的协议，比如TCP、UDP、UDPL、RAW等。

(2) Recv-Q、Send-Q列分别表示网络接收队列、发送队列中的字节数，其中的字母Q是Queue的缩写。具体来说，Recv-Q表示套接字连接的本地接收缓冲区中没有被应用进程取走的字节数，其统计单位是字节，该值表示套接字总共还有多少字节的数据，没有从内核空间的套接字缓存区复制到用户空间缓冲区。Send-Q表示套接字连接的发送队列中，对方没有收到的数据或者说没有被对方确认（Ack）的数据，其统计单位也是字节。

(3) Local Address、Foreign Address两列用于展示套接字连接的本地地址、对端地址，地址中包含了套接字的IP和端口号。如果netstat命令中使用了-n（--numeric）选项，地址和端口会以数字的

形式展示；否则，地址将被解析为规范主机名（FQDN），并且一些默认的端口号将被解析为相应的协议名称，例如地址127.0.0.1会被解析为localhost、端口80会被解析为http。

（4）State列用于展示套接字连接的状态。如果是TCP连接，那么此列将展示TCP连接的11种状态中的某种状态。

上述命令结果中各列的具体含义还可以通过Linux命令man netstat查看。

netstat命令的选项比较多，大致如表10-3所示。

表10-3 netstat命令的基础选项

基础选项	说 明
-a 或--all	显示所有 socket 连接，包括服务端监听套接字
-c 或--continuous	持续列出 socket 连接的状态
-e 或--extend	显示 socket 连接其他相关信息
-g 或--groups	显示多重广播功能群组中的成员名单
-h 或--help	显示该命令的在线帮助
-i 或--interfaces	显示网络接口（网卡）信息
-l 或--listening	显示监听中的服务端监听 socket
-n 或--numeric	直接以数字的形式展示 IP 地址和端口
-o 或--timers	显示计时器
-p 或--programs	显示正在使用 socket 的 PID（进程 ID 和进程名称）
-r 或--route	显示路由表
-s 或--statistice	显示网络统计信息
-t 或--tcp	显示 TCP 传输协议的连接信息
-u 或--udp	显示 UDP 传输协议的连接信息
-w 或--raw	显示 RAW 传输协议的连线信息

一般情况下，可以使用netstat -antp指令去查看TCP的连接信息，包含其进程的PID和名称。在实际的连接状态查看过程中，有一个

持续查看的过程，会用到Shell脚本中的while循环。一段简单的通过while循环查看服务端特定监听端口（如18899）的所有TCP连接的Shell脚本大致如下：

```
root@localhost ~]# while [ 1 -eq 1 ] ; do netstat -antp|grep  
18899 ; sleep 2; echo --; done  
tcp6 0 0 ::::18899          ::::*                      LISTEN  
8422/java  
  
tcp6 0 0 192.168.233.128:18899 192.168.233.1:47624  
ESTABLISHED 8422/java  
--  
tcp6 0 0 ::::18899          ::::*                      LISTEN  
8422/java  
  
tcp6 0 0 192.168.233.128:18899 192.168.233.1:47624  
ESTABLISHED 8422/java  
--  
tcp6 0 0 ::::18899          ::::*                      LISTEN  
8422/java  
  
tcp6 0 0 192.168.233.128:18899 192.168.233.1:47624  
ESTABLISHED 8422/java  
  
...
```

10.4 HTTP长连接原理

HTTP属于TCP/IP模型中的应用层协议，HTTP长连接和HTTP短连接指的是传输层的TCP连接是否被多次使用。

一般来说，用户通过浏览器输入URL后按回车键，浏览器会通过DNS解析域名得到服务器的IP地址，然后通过解析出来的IP和URL中的端口（默认为80）发起建立TCP连接请求，通过三次握手之后建立TCP连接。

10.4.1 HTTP长连接和短连接

默认情况下，在HTTP的1.0版本协议中，HTTP在每次请求结束后都会主动释放TCP连接，因此HTTP连接是一种“短连接”。客户端与服务端通过HTTP短连接的交互过程如图10-13所示。

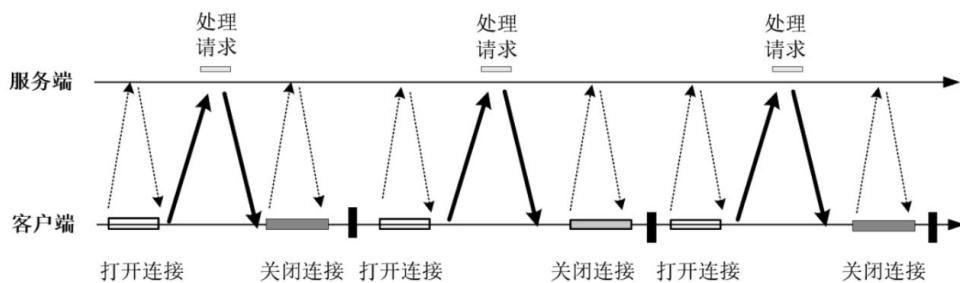


图10-13 客户端与服务端通过HTTP短连接的交互过程

在短连接通信场景下，要保持客户端程序的在线状态，客户端需要不断地向服务器发起连接请求。通常的做法是即使不需要获得任何数据，客户端也保持每隔一段时间向服务器发送一次“保持连

接”的请求，服务器在收到该请求后对客户端进行回复，表明知道客户端“在线”。若服务器长时间无法收到客户端的请求，则认为客户端“下线”，若客户端长时间无法收到服务器的回复，则认为网络已经断开。

在高并发场景使用HTTP“短连接”通信，会出现两个问题：

(1) 性能较差：传输层的TCP连接不会复用，每一次请求都需要建立和拆除一次TCP连接，也就是说，每次请求均需要TCP三次握手建立连接、TCP四次挥手关闭连接，性能较差。

(2) 很容易出现端口被占满：在主动断开方，系统会出现大量的TIME_WAIT状态的TCP连接，只有等2MSL后TCP连接才会关闭。在高并发场景中，如果服务器主动断开连接，则很容易发生端口耗尽。当然，如果连接被设置了SO_REUSEADDR特性，则其端口可能被其他连接复用。尽管如此，还是会存在不少的约束条件影响到端口复用。

出于以上两个原因，在高并发场景下使用HTTP“短连接”进行通信肯定是不行的。

HTTP长连接也叫HTTP持久连接，指的是TCP连接建立后该传输层连接不再进行释放，供应用层反复使用。客户端与服务端通过HTTP长连接的交互过程如图10-14所示。

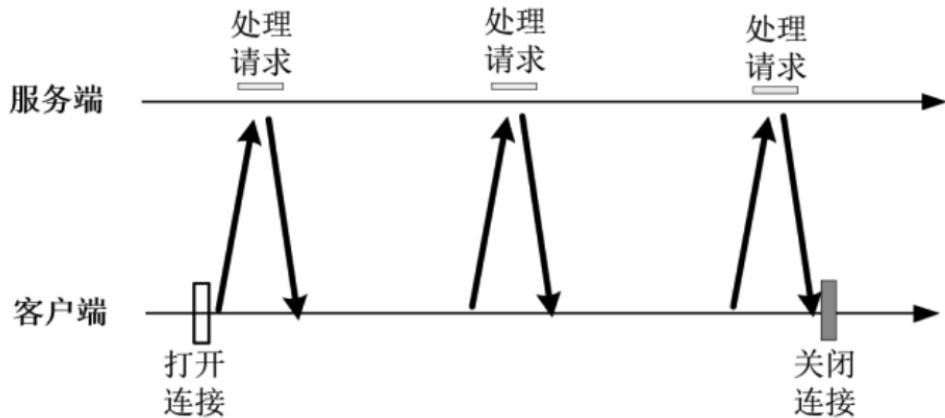


图10-14 客户端与服务端通过HTTP长连接通信的交互过程

HTTP长连接的特点是：

- (1) 性能较高，不需要重复建立TCP连接或者关闭TCP连接。
- (2) TCP数据传输连接基本上不会出现CLOSE_WAIT和TIME_WAIT的问题，系统资源的使用效率会大大提升。

HTTP长连接也有缺点：一般需要一个连接池来对可供复用的TCP长连接进行管理和监测。常见的数据库连接池、HTTP连接池本质上都属于TCP连接池。

10.4.2 不同HTTP版本中的长连接选项

首先回顾一下HTTP/1.0版本中的长连接扩展协议。

从1996年开始，很多HTTP/1.0浏览器与服务器都对HTTP协议进行了扩展，那就是Keep-Alive扩展协议，该扩展作为HTTP/1.0版本的补充“实验型持久连接”协议出现，在HTTP/1.0协议基础上增加一些选项，从而实现HTTP长连接的建立和使用。

在Keep-Alive扩展中，如果客户端在首部加上Connection:Keep-Alive请求头，表示请求服务端将传输层TCP连接保持在打开状态；如果服务端同意将这条TCP连接保持在打开状态，就会在HTTP响应中包含同样的首部；如果HTTP响应中没有包含该首部，则客户端会认为服务端不支持Keep-Alive扩展协议，会在发送完响应报文之后关闭当前的TCP连接。如果客户端与服务端都支持Keep-Alive扩展协议，则双方可以使用HTTP长连接，实现TCP连接的复用。

包含Keep-Alive扩展头的HTTP报文首部如图10-15所示。

```
Cache-Control: max-age=120
Connection: keep-alive
Keep-Alive: timeout=20
Content-Encoding: gzip
Content-Type: text/html; charset=GB2312
Date: Fri, 27 Apr 2018 09:43:31 GMT
Expires: Fri, 27 Apr 2018 09:45:31 GMT
Server: squid/3.5.24
Transfer-Encoding: chunked
```

图10-15 包含Keep-Alive扩展协议首部的HTTP报文首部

HTTP/1.0的Keep-Alive扩展协议存在一些问题：

(1) 该扩展不是标准协议，客户端必须发送Connection:Keep-Alive请求头，请求服务端将传输层TCP连接保持在打开状态；如果没有发送该请求头，则服务端回复后会将TCP连接关闭。

(2) 处于客户端与服务器数据链路中间的反向代理服务器可能无法支持Keep-Alive扩展协议，导致无法使用HTTP长连接。

很多人会把HTTP/1.0协议的Keep-Alive和TCP协议的Keepalive两个概念搞混淆。Keepalive是Socket连接的一个可选项，主要用于Socket连接的保活，在新建Socket的时候可以设置SO_KEEPALIVE套接字可选项，打开保活机制。SO_KEEPALIVE套接字保活可选项主要有三个参数：

(1) `tcp_keepalive_time`: 最后一次数据交换到TCP发送第一个保活探测报文的时间，即允许连接空闲的时间，单位为秒，默认为7200秒，也就是2小时。

(2) `tcp_keepalive_probes`: 发送TCP保活探测数据包的最大数量，默认是9，如果发送9个保活探测包后对端仍然没有响应，便发送RST关闭连接。

(3) `tcp_keepalive_intvl`: 发送两个TCP保活探测数据包的间隔时间，默认是75秒。

SO_KEEPALIVE只是TCP连接的一个可选项，其参数配置可能会引起一些问题，所以该可选项默认是关闭的。TCP连接的保活也可以通过应用程序自己完成，类似的如Netty中保活报文和空闲监测机制。

HTTP/1.0协议的Keep-Alive是一个HTTP连接复用的扩展协议，是属于应用层的协议内容。

介绍完HTTP/1.0版本中的长连接方案之后，接下来介绍一下HTTP/1.1版本中的长连接选项。

虽然很多客户端与服务器程序延续支持HTTP/1.0的Keep-Alive扩展协议，但是HTTP/1.1标准协议并没有使用HTTP/1.0的Keep-Alive扩展协议，而是定义了自己的连接复用方案。

HTTP/1.1默认使用长连接而不是短连接，除非显式关闭TCP连接。如果要显式关闭连接，需要在HTTP报文首部加上Connection:Close请求头，也就是说在HTTP/1.1协议中，默认情况下所有的TCP连接都可以进行复用。

不发送Connection:Close请求头并不意味着服务器承诺TCP连接永远保持打开。空闲的TCP连接也可以被客户端与服务端关闭。

10.5 服务端HTTP长连接技术

本节介绍主流的反向代理服务器Nginx和应用服务器Tomcat的服务端长连接配置。

10.5.1 应用服务器Tomcat的长连接配置

生产环境所用的Java应用服务器不一定是Tomcat，可能是JBoss、Jetty或者其他的应用服务器。无论使用哪一种服务器，其HTTP长连接配置的原理都是类似的，所以这里以Tomcat为例进行应用服务器的长连接配置介绍。

服务端Tomcat的长连接配置主要分为两种场景：

(1) 独立部署的Tomcat

在传统的Nginx+Tomcat架构的Web应用中，一般使用独立部署的Tomcat作为Web服务器。

(2) 内嵌部署的Tomcat

在目前主流的Spring Cloud微服务架构中的微服务Provider实例一般使用内嵌的Tomcat作为Web服务器。

以上两种细分场景的Tomcat使用如图10-16所示。

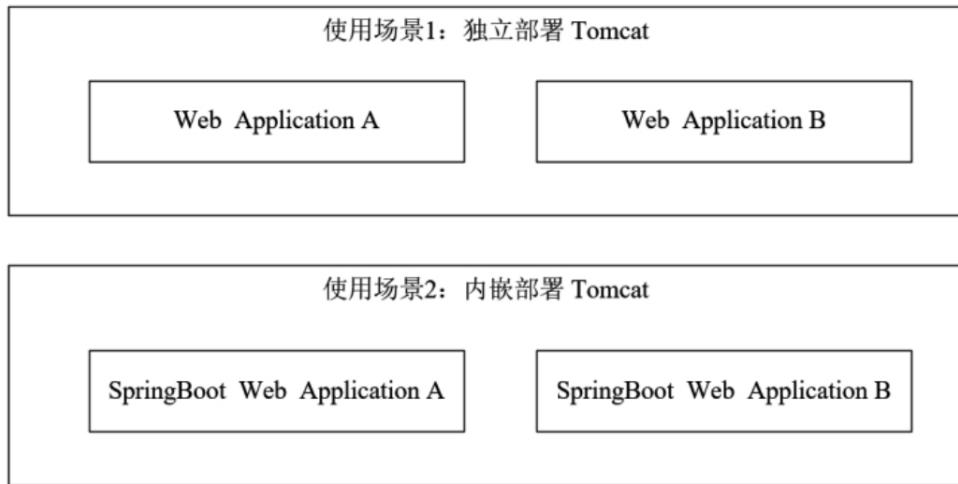


图10-16 服务端Tomcat使用的两种细分场景

1. 独立部署Tomcat的长连接配置

针对细分场景1中的独立部署Tomcat，其长连接配置是通过修改Tomcat配置文件中Connector（连接器）的配置完成的。一个使用HTTP长连接的Connector连接器的配置示例大致如下（Tomcat版本假定8.0或以上）：

```

<Connector port="8080" redirectPort="8443"
protocol="org.apache.coyote.http11.Http11NioProtocol"
connectionTimeout="20000"
URIEncoding="UTF-8"

keepAliveTimeout="15000"
maxKeepAliveRequests="-1"
maxConnections="3000"

```

```
    maxThreads="1000"  
    maxIdleTime="300000"  
    minSpareThreads="200"  
    acceptCount="100"  
  
    enableLookups="false" />
```

对以上配置示例中用到的三个长连接配置选项，介绍如下：

(1) keepAliveTimeout

此选项为TCP连接保持时长，单位为毫秒，表示在下次请求过来之前该连接将被Tomcat保持多久。在keepAliveTimeout时间范围内，假如客户端不断有新的请求过来，则该连接将一直被保持。

KeepAliveTimeout选项决定一个不活跃的连接能保持多少时间。

(2) maxKeepAliveRequests

此选项表示长连接最大支持的请求数。超过该请求数的连接将被关闭，关闭的时候Tomcat会返回一个带Connection: close响应头的消息给客户端。

当maxKeepAliveRequests的值为-1时，表示没有最大请求数限制；如果其值被设置为1，将会禁用掉HTTP长连接。

默认情况下Tomcat是使用长连接的，要关闭长连接，只要将maxKeepAliveRequests设置为1即可。

(3) maxConnections

Tomcat在任意时刻能接收和处理的最大连接数。如果其值被设置为-1，则连接数不受限制。由于Linux的内核默认限制了单进程最大打开文件句柄数1024，因此如果此配置项的值超过1024，则相应的需要对Linux系统的单进程最大打开文件句柄数限制进行修改。

以上是对Tomcat的HTTP长连接配置选项的介绍。总的来说，使用长连接能提高服务性能，不过使用不当也会带来一些不利的结果。

使用长连接意味着一个TCP连接在当前请求结束后如果没有新的请求到来，Socket连接不会立马释放，而是等keepAliveTimeout到期之后才释放，如果一个高负载的Tomcat服务器建立很多长连接，就将无法继续建立新的连接，无法为新的客户端提供服务。所以，对于Tomcat长连接的配置需要慎重，错误的参数可能导致严重的性能问题，需要根据具体的负载配置合适的KeepAliveTimeout和MaxKeepAliveRequests选项值。

2. 内嵌式部署Tomcat的长连接配置

针对细分场景2中的内嵌式Tomcat，其长连接配置可以通过一个自动配置类完成。在自动配置类中，可以配置一个TomcatServletWebServerFactory容器工厂Bean实例，Spring Boot将通过该工厂实例在运行时获取内嵌式Tomcat容器实例。在容器工厂配置代码中，可以对Tomcat的Connector的三个长连接相关属性进行具体的配置。

一段简单的定制化TomcatServletWebServerFactory容器工厂的配置代码大致如下：

```
package com.crazymaker.springcloud.standard.config;  
//省略import  
  
@Configuration  
@ConditionalOnClass({Connector.class})  
public class TomcatConfig  
{  
  
    @Autowired  
    private HttpConnectionProperties httpConnectionProperties;  
  
    @Bean  
    public TomcatServletWebServerFactory  
        createEmbeddedServletContainerFactory()  
    {  
        TomcatServletWebServerFactory tomcatFactory =  
            new  
        TomcatServletWebServerFactory();  
  
        //增加连接器的定制配置  
        tomcatFactory.addConnectorCustomizers(connector ->  
        {  
            Http11NioProtocol protocol =  
                (Http11NioProtocol)  
            connector.getProtocolHandler();  
  
            //定制keepAliveTimeout，确定下次请求过来之前Socket连接保  
持多久  
        });  
    }  
}
```

```
//设置600秒内没有请求则服务端自动断开Socket连接  
protocol.setKeepAliveTimeout(600000);  
  
//当客户端发送的请求超过10000个时强制关闭Socket连接  
protocol.setMaxKeepAliveRequests(1000);  
  
//设置最大连接数  
protocol.setMaxConnections(3000);  
  
//省略其他配置  
});  
return tomcatFactory;  
}  
}
```

以上示例是Spring Boot 2.0.8中的内嵌式Tomcat长连接配置，具体的三个配置选项的语义和独立Tomcat的配置是相同的，仅仅是形式上的不同。

说明

以上内嵌式Tomcat的配置代码来自《Spring Cloud、Nginx高并发核心编程》一书的配套源码。

10.5.2 Nginx承担服务端角色时的长连接配置

无论在传统的Nginx + Tomcat架构中，还是在目前主流的Nginx + Spring Cloud架构中，反向代理Nginx都承担了两种角色：对于下游客户端来说Nginx承担了服务端角色，对于上游的Web服务器来说Nginx承担了客户端角色。Nginx承担的两种角色如图10-17所示。

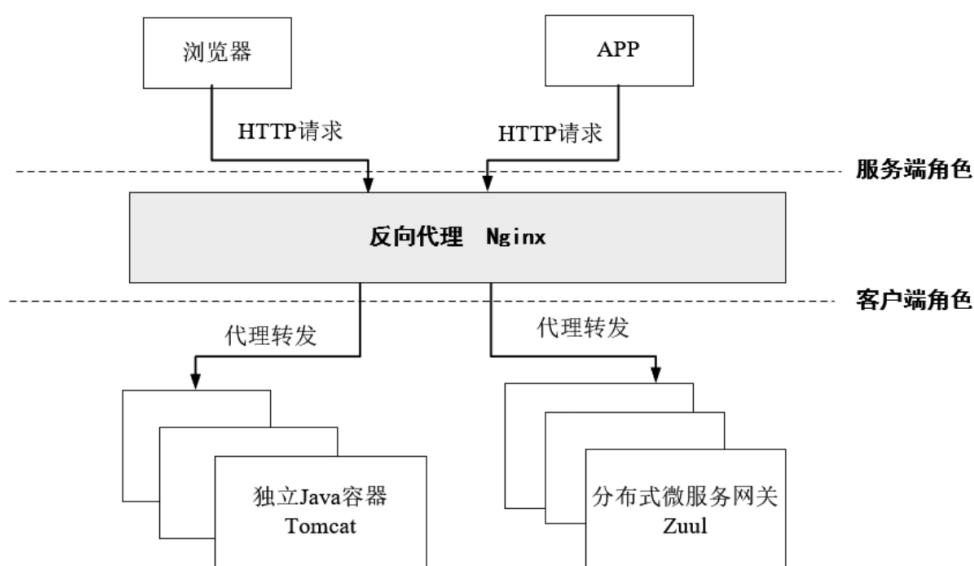


图10-17 Nginx承担的两种角色

Nginx是一个高性能的HTTP和反向代理Web服务器，是由伊戈尔·赛索耶夫为俄罗斯访问量第二的Rambler.ru站点开发的Web服务器。Nginx源代码以类BSD许可证的形式发布，其第一个公开版本0.1.0发布于2004年10月4日，其1.0.4版本发布于2011年6月1日。Nginx因高稳定性、丰富的功能集、内存消耗少、并发能力强而闻名全球，目前得到非常广泛的使用，比如百度、京东、新浪、网易、腾讯、淘宝等都是其用户。

说明

Nginx是Java工程师必备的技能之一，有关Nginx的原理和详细知识，请参考笔者的另外一本书《Spring Cloud、Nginx高并发核心编程》。

Nginx承担服务端角色时的长连接，主要通过keepalive_timeout 和keepalive_requests两个指令完成相关设置。一段简单的Nginx承担服务端角色时的长连接配置代码如下：

```
#...  
http {  
    include          mime.types;  
    default_type    application/octet-stream;  
  
    #长连接保持时长  
    keepalive_timeout  65s;  
    #长连接最大处理请求数  
    keepalive_requests 1000;  
    #...  
    server {  
        listen          80;  
        server_name    openresty localhost;  
        #长连接保持时长  
        keepalive_timeout  10s;  
        #长连接最大处理请求数
```

```
keepalive_requests 10;

location / {
    root    html;
    index  index.html index.htm;
}

#...

}

}
```

对以上配置代码中涉及的两个长连接相关指令的具体介绍如下：

(1) keepalive_requests

此指令设置同一个长连接可以处理的最大请求数，请求数超过此值，长连接将关闭。其格式如下：

语法：keepalive_requests number

默认值：keepalive_requests 100

上下文：http、server、location

keepalive_requests指令用于设置一个长连接上可以服务的最大请求数量，当最大请求数量达到时长连接将被关闭，Nginx中的默认值是100。一个长连接建立之后，Nginx就会为这个连接设置一个计数器，记录这个长连接上已经接收并处理的客户端请求的数量。如果达到这个参数设置的最大值，则Nginx会强行关闭这个长连接，逼迫客户端不得不重新建立新的长连接。

(2) keepalive_timeout

此指令用于设置长连接的空闲保持时长，表示在下次请求过来之前该连接将被Nginx保持多久。在keepalive_timeout时间范围内，假如客户端不断有新的请求过来，则该连接将一直被保持。

语法：keepalive_timeout timeout [header_timeout];

默认值：keepalive_timeout 60s;

上下文：http、server、location

keepalive_timeout指令的第一个参数用于设置客户端的长连接在服务端保持的最长时间（默认为60秒），如果值设置为0，就会禁用HTTP长连接。对于一些并发量较高的内部服务器通信的场景，其值可以适当加大，比如增加到120秒甚至300秒。

keepalive_timeout指令的第二个参数是一个可选参数，其作用是为HTTP响应报文增加一个Keep-Alive: timeout=time头部选项，用于告知客户端长连接的保持时间，通常可以不用设置。该响应头可以被Mozilla浏览器识别和处理，Mozilla浏览器会在timeout空闲时间之后关闭TCP长连接；MSIE浏览器则在大约60秒后关闭长连接。

Nginx承担客户端角色时的长连接设置将在下节专门介绍。

10.5.3 服务端长连接设置的注意事项

在进行服务端长连接设置时，keepalive_timeout和keepalive_requests的值并不是越大越好，而是要根据具体场景而定。

1. 单个客户端的HTTP请求数较少时

假设客户端是普通用户，客户端是网页浏览器，当用户通过浏览器在访问服务端时，其单个用户的请求数是比较有限的，1分钟之内所发出的请求数最多在百位数左右。在这种场景下，如果Nginx的服务端长连接设置如下：

```
#长连接保持时长  
keepalive_timeout 65s;  
  
#长连接最大处理请求数  
keepalive_requests 1000;
```

则会导致大量的长连接由于请求数达不到1000而一直在空闲等待，需要等到65秒结束之后才被关闭，造成服务器资源的浪费。所以，需要减少长连接最大处理请求数和长连接保持时长，初步优化后的配置大致如下：

```
#长连接保持时长  
keepalive_timeout 10s;  
  
#长连接最大处理请求数  
keepalive_requests 100;
```

如果配置得比较极端，将长连接最大处理请求数减小得太多，就可能会导致其他问题。比如，将长连接最大处理请求数减到10，其配置如下：

```
#长连接保持时长  
keepalive_timeout 10s;  
  
#长连接最大处理请求数  
keepalive_requests 10;
```

当QPS=10000时，假定一共有100个用户，单个客户端每秒发出100个请求。由于以上配置中每个连接只能最多处理10次请求，因此单个客户端每秒发出100个请求相当于每个用户需要10个连接，在总体100个用户的情况下，意味着平均每秒钟就会有1000个长连接将被Nginx主动关闭。在这种情况下，了解前面介绍的TCP连接四次挥手知识的读者就会知道，服务端Nginx会有大量的TIME_WAIT的Socket连接。

所以，keepalive_requests的值不能比单个客户端在keepalive_timeout时间范围的实际请求数少太多，如果少太多，在QPS较高的场景，就会大量连接被服务端主动关闭，从而导致大量TIME_WAIT连接。

当然，keepalive_requests的值也不能比单个客户端在keepalive_timeout时间范围的实际请求数多太多，这样会导致大量的TCP长连接出现空闲等待。

总体而言，keepalive_requests的值与单客户端在keepalive_timeout时间范围的实际请求数量要做到基本匹配。

2. 单个客户端的请求数较多时

假设客户端不是普通用户，而是下游的代理服务器，在这种场景下，客户端数量是很少的，而单个客户端与服务器之间的请求数是非常多的。

这种场景的设置比较简单，可以尽可能地对长连接进行复用，keepalive_requests值可以设置偏大，示例的配置如下：

```
#长连接保持时长  
keepalive_timeout 65s;  
  
#长连接最大处理请求数  
keepalive_requests 100000;
```

在此场景中，选项`keepalive_timeout`可以配置一个较大的值。但是，对于Nginx来说，不能不对单个连接的处理请求数做限制，必须定期关闭连接，才能释放每个连接所分配的内存。由于使用过大请求数可能会导致内存占用过度，因此不建议为`keepalive_requests`设置太大的值，当然更不能不做`keepalive_requests`设置。

无论是Nginx、Tomcat还是其他的服务器，有关服务端长连接的设置，其原理是类似的，仅仅是具体参数的命名规则不同或者是配置形式稍微有点不同。

10.6 客户端HTTP长连接技术原理与实验

使用HTTP长连接通信，光靠服务端是不够的，还需要客户端配合。一般来说，除了浏览器这些不涉及Java编程的客户端外，还涉及与Java编程有关的HTTP客户端编程与配置技术，主要有：

- (1) Java内置的HttpURLConnection的HTTP短连接通信编程技术。
- (2) 第三方开源HTTP长连接通信编程技术，如Apache HttpClient客户端。
- (3) 反向代理（如Nginx）在承担客户端角色访问上游RS（真实服务器）时的HTTP长连接配置技术。

10.6.1 HttpURLConnection短连接技术

客户端通过Java内置的HttpURLConnection短连接访问远程服务的流程如图10-18所示。

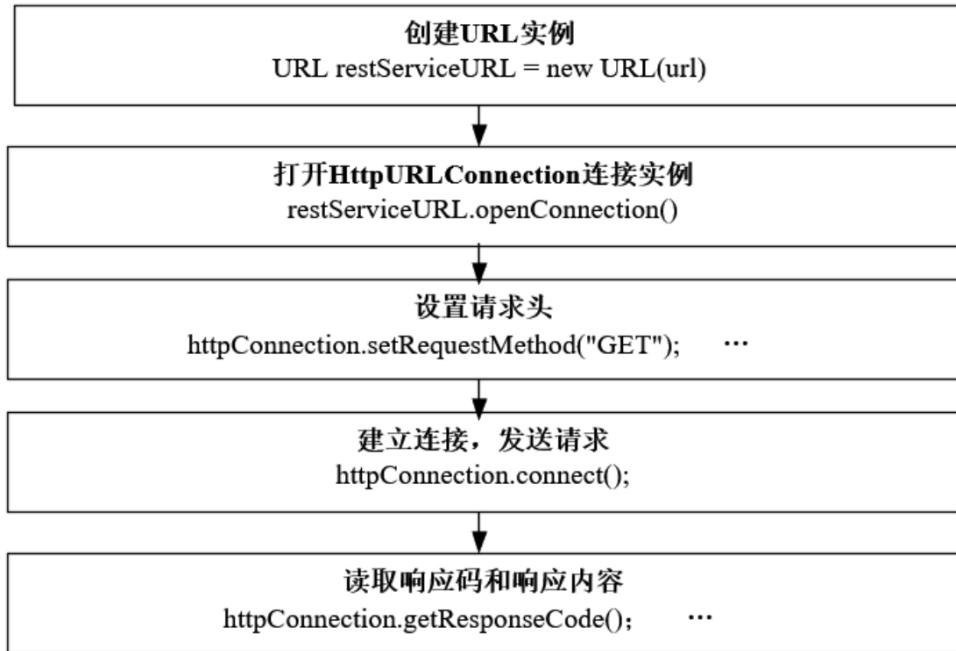


图10-18 Java客户端通过HttpURLConnection短连接访问远程服务的流程

对客户端通过HttpURLConnection短连接访问远程服务的流程中的主要步骤介绍如下：

(1) 创建URL实例

```
URL restServiceURL = new URL(url)
```

(2) 打开HttpURLConnection连接实例

```
HttpURLConnection  
httpConnection=restServiceURL.openConnection()
```

HttpURLConnection只是一个抽象类，并不是底层的连接，其具体的请求实例可以通过URL.openConnection()方法创建。

(3) 设置请求头

HTTP请求头允许一个键（Key）带多个用逗号分开的值（Value），但是HttpURLConnection只提供了单个键-值对的操作：

```
setRequestProperty(key,value) //重置请求头的Key, Value  
addRequestProperty(key,value) //新增请求头的Key, Value
```

setRequestProperty和addRequestProperty的区别是：

setRequestProperty会覆盖已经存在的键的所有值，有清零之后重新赋值的作用； addRequestProperty则是在原来键的基础上继续添加其他值。

（4）建立连接，发送请求

```
httpConnection.connect(); //发送URL请求
```

建立实际TCP连接之后的工作就是发送请求。如果需要发送请求体（Request Body）到服务器，则需要获取其输出流outputStream，并通过该流写入要发送的数据。

```
OutputStream outputStream= httpUrlConnection.getOutputStream();
```

如果调用了getOutputStream()方法，就会隐含地调用上面的connect()连接方法。所以，在开发中如果获取了输出流，则可以不显式调用上面的connect()方法。

（5）读取响应码和响应内容

请求发送成功之后即可获取响应的状态码，如果返回成功就可以读取响应中的返回数据。获取这些返回数据的方法如下：

```
getContent(); //获取响应内容  
getHeaderField(); //获取响应头  
getInputStream(); //获取输入流
```

在响应处理过程中，`getInputStream()`和`getContent()`这两个方法是用得最多的。

(6) 关闭连接

每一个`HttpURLConnection`请求结束之后，应该调用`HttpURLConnection`实例的`InputStream`（输入流）或`OutputStream`（输出流）的`close()`方法，以释放请求的网络资源。

以上是客户端通过`HttpURLConnection`短连接访问远程服务的大致步骤。在本书的随书源码中，编写了一个HTTP客户端处理帮助类`HttpClientHelper`，其中的`jdkGet(String url)`方法实现了以上请求逻辑，主要代码如下：

```
package com.crazymakercircle.util;  
//省略import  
  
//HTTP 客户端处理帮助类  
@Slf4j  
public class HttpClientHelper  
{  
    /**  
     * 使用JDK的 java.net.HttpURLConnection 发起HTTP请求  
     */
```

```
public static String jdkGet(String url)
{
    InputStream inputStream = null; //输入流
    HttpURLConnection httpConnection = null; //HTTP
```

连接实例

```
StringBuilder builder = new StringBuilder();
try
{
    URL restServiceURL = new URL(url);

    //打开HttpURLConnection连接实例
    httpConnection =
        (HttpURLConnection)
    restServiceURL.openConnection();

    //设置请求头
    httpConnection.setRequestMethod("GET");
    httpConnection.setRequestProperty(
        "Accept", "application/json");

    //建立连接，发送请求
    httpConnection.connect();

    //读取响应码
    if (httpConnection.getResponseCode() != 200)
    {
        throw new RuntimeException("Failed with Error
```

```
code : "  
        + httpConnection.getResponseCode()) ;  
    }  
  
    //读取响应内容 (字节流)  
    inputStream = httpConnection.getInputStream();  
    byte[] b = new byte[1024];  
    int length = -1;  
    while ((length = inputStream.read(b)) != -1)  
    {  
        builder.append(new String(b, 0, length));  
    }  
} catch (MalformedURLException e)  
{  
    e.printStackTrace();  
} catch (IOException e)  
{  
    e.printStackTrace();  
} finally  
{  
    //关闭流和连接  
    quietlyClose(inputStream);  
    httpConnection.disconnect();  
}  
return builder.toString();  
}
```

```
//...
}
```

10.6.2 HTTP短连接的通信实验

接下来通过上面自定义的 jdkGet (String url) 方法进行HTTP短连接的通信实验。这里访问的服务端应用是上一章所编写的HTTP Echo回显服务。为了在服务端查看连接的状态和信息，将该HTTP Echo服务部署在虚拟机上，其IP在这里为192.168.233.128。

为了方便HTTP Echo服务的独立部署，笔者已经为该应用增加了 Spring Boot 的引导类和Linux启动脚本start. sh，读者只需要通过 Maven 打包该应用，将 zip 包上传到虚拟机解压缩，然后使用 start. sh 脚本进行启动即可。

调用 jdkGet (String url) 方法访问HTTP Echo服务的实验用例代码具体如下：

```
package com.crazymakercircle;

//省略import

public class HTTPKeepAliveTester
{
    //HTTP echo 回显服务的地址，该服务部署在虚拟机192.168.233.128上
    private String url = "http://192.168.233.128:18899/";

    private ExecutorService pool =
        Executors.newFixedThreadPool(10);
```

```
/*
 * 测试用例：使用JDK的 java.net.HttpURLConnection发起HTTP请求
 */
@Test
public void simpleGet() throws IOException,
InterruptedException
{
    int index = 1000000; //提交的请求总次数
    while (--index > 0)
    {
        String target = url + index;
        //使用固定10个线程的线程池发起请求，并发为10
        pool.submit(() ->
        {
            //使用JDK的 java.net.HttpURLConnection发起HTTP请
求
            String out = HttpClientHelper.jdkGet(target);
            System.out.println("out = " + out);
        }) ;
    }
    Thread.sleep(Integer.MAX_VALUE);
}
...
}
```

在Linux虚拟机上，可以通过netstat指令看到具体的TCP连接信息。通过仔细观察可以看到，上面实验中所建立的HTTP通信连接都是HTTP短连接，都没有进行复用。为什么呢？因为HTTP协议下层的TCP连接端口，每一轮循环的输出都是不相同的。通过观察实验结果可以看到，netstat指令程序每隔1秒输出一批ESTABLISHED状态的连接（10个），它们的端口都是不同的。

使用netstat指令所看到的连接信息，部分节选如下：

```
[root@localhost ~]# while [ 1 -eq 1 ] ; do netstat -antp|grep  
18899    ; sleep 2; echo ; done  
  
tcp6      0      0 ::::18899          ::::*  
LISTEN     8422/java  
  
...  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11184  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11187  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11195  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11190  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11194  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11193  
ESTABLISHED 8422/java  
  
tcp6      0      0 192.168.233.128:18899  192.168.233.1:11188
```

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11186

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11189

ESTABLISHED 8422/java

tcp6 0 0 ::18899 ::*:*

LISTEN 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11381

ESTABLISHED 8422/java
tcp6 0 308 192.168.233.128:18899 192.168.233.1:11374

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11367

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11378

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11376

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11349

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11379

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11382

ESTABLISHED 8422/java
tcp6 0 308 192.168.233.128:18899 192.168.233.1:11369

ESTABLISHED 8422/java
tcp6 0 0 192.168.233.128:18899 192.168.233.1:11373

ESTABLISHED 8422/java

tcp6 0 0 ::::18899 ::::*
LISTEN 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11592
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11608
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11609
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11602
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11604
ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:11382
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11600
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11598
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11574
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11605
ESTABLISHED 8422/java

tcp6 0 308 192.168.233.128:18899 192.168.233.1:11607
ESTABLISHED 8422/java

本小节案例所呈现的是使用JDK自带的HttpURLConnection进行的短连接实验。尽管服务端HTTP Echo服务是支持长连接的，但是由于客户端完成请求之后关闭了连接，因此通信过程中仍然是一次性的HTTP短连接。

在客户端使用长连接，还需要有一个活跃连接的管理和复用组件，这些组件一般为开源或者自制的TCP连接池，其原理和数据库连接池类似。

10.6.3 Apache HttpClient客户端的HTTP长连接技术

在开发Java应用的过程中所涉及的HTTP连接复用的高并发场景大致有以下几种：

- (1) 反向代理Nginx到Java Web应用服务之间的HTTP高并发通信。
- (2) 微服务网关之间、网关与微服务Provider实例之间的HTTP高并发通信。
- (3) 分布式微服务Provider实例与Provider实例之间RPC远程调用的HTTP高并发通信。
- (4) Java通过HTTP客户端调用其他Java Web应用的HTTP接口时的高并发通信。

以上4场景中，除了第1种场景之外，后面的3种场景都需要Java客户端高性能访问远程HTTP接口，都需要Java客户端具备HTTP长连接管理和复用的能力。

比如，在第3种场景中，Spring Cloud微服务Provider实例的客户端RPC组件为Feign，通过合理配置，Feign可以使用Apache HttpClient组件或Google的OkHttp组件进行HTTP连接的高效复用，其原理可以参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

再比如，随着服务粒度越来越细化，Java应用之间的REST API调用也就越来越频繁，这些REST远程调用属于上面的第4种场景。第4种场景还可以细分为两小类：Java应用之间的REST API之间调用、通过REST API网关（或ESB）进行REST API间接调用。

一个Java应用之间的REST API直接调用的示例如图10-19所示。

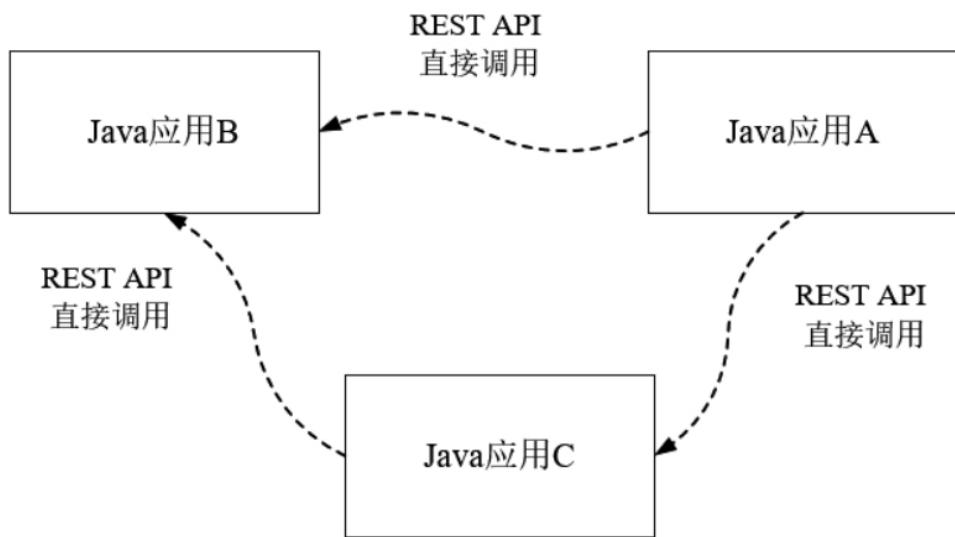


图10-19 Java应用之间的REST API直接调用示例

通常情况下，企业内部的Java应用对外REST API都会统一注册在API网关或者ESB企业服务总线，其他的Java应用如果有需要，可以通过访问网关或ESB总线进行REST API的间接调用。

一个Java应用通过API网关（或者ESB企业服务总线）进行REST API间接调用的示例，具体如图10-20所示。

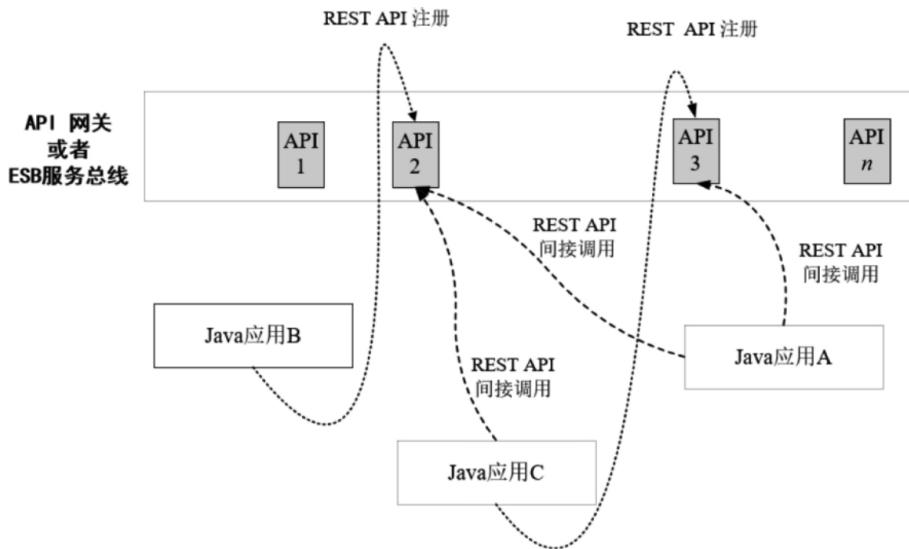


图10-20 Java应用通过API网关进行REST API的间接调用示例

对于Java应用来说，无论是直接REST API调用还是间接REST API调用，在客户端都需要借着高性能的HTTP客户端组件进行REST API远程访问。在高并发场景下，需要HTTP客户端组件具备长连接管理和复用的能力。

带连接池的、具备长连接管理和复用能力的HTTP客户端开源组件很多，著名的有Apache HttpClient组件和Google的OkHttp组件。这里以Apache HttpClient组件为例，为大家介绍Java客户端的HTTP长连接使用技术。

前面讲到，在客户端使用长连接，还需要有一个活跃连接的管理和复用组件，该组件一般为开源或者自制的TCP连接池，其原理和数据库连接池类似。JDK自带的HttpURLConnection连接类缺少一个有效的

连接管理组件（如连接池），尽管其底层通过Map类型的内存映射组件实现了非常简单的TCP连接的缓存和复用，但是实际的复用效率很低。

HttpClient中使用连接池来管理持有连接，在原理上，无论是数据库连接池还是HTTP连接池，连接的“池化管理”技术都是一种通用的设计，其原理并不复杂：

- (1) 在请求连接时，如果池中没有连接，则建立一条新的连接。
- (2) 在归还连接时，连接不直接关闭，而归还到池中。
- (3) 在请求连接时，如果池中有可用连接，则可从池中获取一个可用连接。
- (4) 定期清理过期连接。

Apache HttpClient客户端组件实现了自己的连接池组件，该连接池组件负责长连接的创建、监控和释放，其具体的类为PoolingHttpClientConnectionManager。

Apache HttpClient的连接池组件原理如图10-21所示。

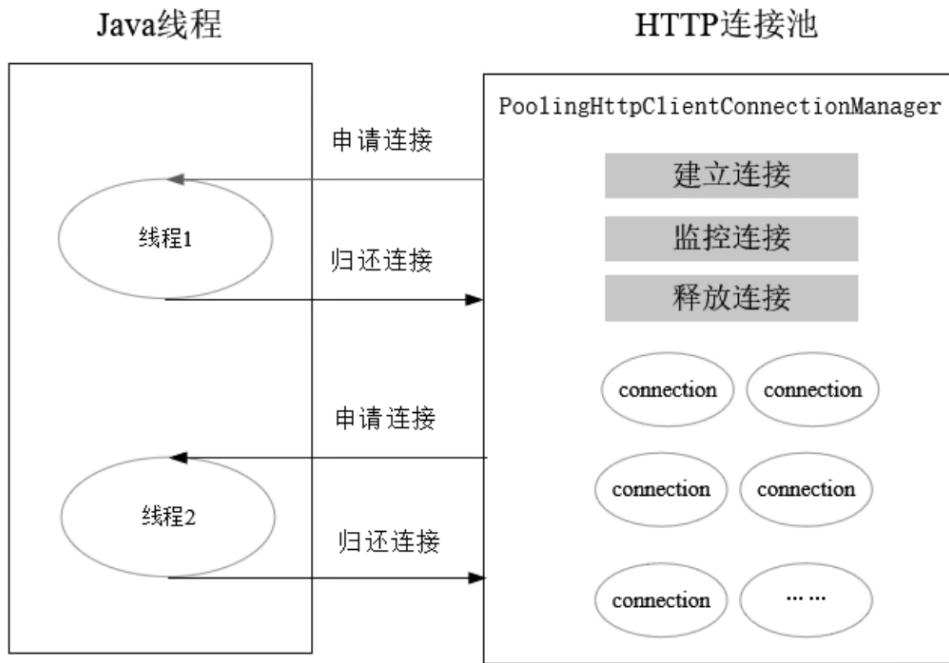


图10-21 Apache HttpClient的连接池组件的原理

在随书源码的HTTP处理帮助类HttpClientHelper中实现了一个Apache HttpClient的全局实例，其连接池的创建方法createHttpClientConnectionManager()的代码如下：

```

package com.crazymakercircle.util;

//省略import

//HTTP 协议处理帮助类

@Slf4j

public class HttpClientHelper
{
    //长连接的保持时长，单位为ms（毫秒）

    private static final long KEEP_ALIVE_DURATION = 600000;
}

```

```
//客户端和服务器建立连接的超时时长，单位ms
private static final int CONNECT_TIMEOUT = 2000;

//建立连接后，客户端从服务器读取数据的超时时长，单位ms
private static final int SOCKET_TIMEOUT = 2000;

//从连接池获取连接的超时时长，单位ms
private static final int REQUEST_TIMEOUT = 2000;

//无效长连接的清理间隔，单位ms
private static final int EXPIRED_CHECK_GAP = 6000;

//连接池内对不活跃连接的检查间隔，单位ms
private static final int VALIDATE_AFTER_INACTIVITY = 2000;

//最大的连接数
private static final int POOL_MAXTOTAL = 500;

//每一个路由(可以理解为IP+端口)的最大连接数
private static final int MAX_PER_ROUTE = 500;

//单例：HTTP长连接管理器，也就是连接池
private static PoolingHttpClientConnectionManager
    httpClientConnectionManager;

//单例：全局的池化HTTP客户端实例
private static CloseableHttpClient pooledHttpClient;
```

```
//线程池：负责HTTP连接池的无效连接清理
private static ScheduledExecutorService monitorExecutor =
null;

//创建全局连接池：HTTP连接管理器
public static void createHttpClientConnectionManager()
{
    //DNS解析器
    DnsResolver dnsResolver =
SystemDefaultDnsResolver.INSTANCE;
    //负责HTTP传输的套接字工厂
    ConnectionSocketFactory plainSocketFactory =
PlainConnectionSocketFactory.getSocketFactory();
    //负责HTTPS传输的安全套接字工厂
    LayeredConnectionSocketFactory sslSocketFactory =
SSLConnectionSocketFactory.getSocketFactory();
    //根据应用层协议，为其注册传输层的套接字工厂
    Registry<ConnectionSocketFactory> registry =
    RegistryBuilder.<ConnectionSocketFactory>create()
        .register("http", plainSocketFactory)
        .register("https", sslSocketFactory)
        .build();
}
```

```
//创建连接管理器

httpClientConnectionManager =
    new PoolingHttpClientConnectionManager(
        registry, //传输层套接字注册器
        null,
        null,
        dnsResolver, //DNS解析器
        KEEP_ALIVE_DURATION, //长连接的连接保持时长
        TimeUnit.MILLISECONDS); //保持时长的时间单位

//连接池内，连接不活跃多长时间后，需要进行一次验证
//默认为2秒 TimeUnit.MILLISECONDS
httpClientConnectionManager.setValidateAfterInactivity(
    VALIDATE_AFTER_INACTIVITY);

//最大连接数，高于这个值时的新连接请求，需要阻塞和排队等待
httpClientConnectionManager.setMaxTotal(POOL_MAXTOTAL);
//设置每个route默认的最大连接数，路由是对MaxTotal的细分
//每个路由实际最大连接数默认值是由DefaultMaxPerRoute控制的
//MaxPerRoute设置过小，无法支持大并发

httpClientConnectionManager.setDefaultMaxPerRoute(MAX_PER_ROUTE);
}
```

```
//省略其他方法  
}
```

通常，服务端不会允许无限期的长连接存在，会通过设置keepalive_timeout选项或者其他类似选项关闭超过保持时长的空闲连接。但是，长连接在被服务端关闭之后，客户端不一定能收到通知，很可能没有及时从客户端的连接池中清理出去。

在客户端，如果将服务端已经关闭的HTTP连接提供给Java线程，就会导致Java线程在发送请求和获取响应时发生异常。为此，客户端需要开启监控线程，每隔一段时间就检测一下连接池中长连接的情况，及时关闭异常连接。客户端关闭异常连接的定时执行代码，大致如下：

```
package com.crazymakercircle.util;  
  
//省略import  
//HTTP 协议处理帮助类  
@Slf4j  
public class HttpClientHelper  
{  
    //省略其他方法  
    /**  
     * 定时处理线程：对异常连接进行关闭  
     */  
    private static void startExpiredConnectionsMonitor()  
    {  
        //空闲监测，配置文件默认为6秒，生产环境建议稍微放大一点
```

```
int idleCheckGap = IDLE_CHECK_GAP;
//设置保持连接的时长，根据实际情况调整配置

long keepAliveTimeout = KEEP_ALIVE_DURATION;
//开启监控线程，关闭异常和空闲线程

monitorExecutor = Executors.newScheduledThreadPool(1);
monitorExecutor.scheduleAtFixedRate(new TimerTask()

{

    @Override

    public void run()

    {

        //关闭异常连接，包括被服务端关闭的长连接

        httpClientConnectionManager.closeExpiredConnections();

        //关闭keepAliveTimeout（保持连接时长）超时的不活跃连接

        httpClientConnectionManager.closeIdleConnections(

            keepAliveTimeout,

            TimeUnit.MILLISECONDS);

        //获取连接池的状态

        PoolStats status =


httpClientConnectionManager.getTotalStats();

        //输出连接池的状态，仅供测试使用

        /*

```

```
        log.info(" manager.getRoutes().size():" +  
  
manager.getRoutes().size());  
        log.info(" status.getAvailable():" +  
status.getAvailable());  
        log.info(" status.getPending():" +  
status.getPending());  
        log.info(" status.getLeased():" +  
status.getLeased());  
        log.info(" status.getMax():" +  
status.getMax());  
        */  
    }  
}, idleCheckGap, idleCheckGap, TimeUnit.MILLISECONDS);  
}  
}
```

一般来说，在Java程序中可以维护一个全局静态的带连接池的HttpClient客户端实例。如果需要使用HTTP长连接，只需要通过全局静态的实例获取即可，不必每一次请求都去创建新的带连接池的HttpClient客户端实例。下面是一段创建带连接池的全局客户端实例pooledHttpClient的代码，大致如下：

```
package com.crazymakercircle.util;  
  
//省略import  
//HTTP 协议处理帮助类
```

```
@Slf4j
public class HttpClientHelper
{
    //省略其他方法

    /**
     * 创建带连接池的 pooledHttpClient 全局客户端实例
     */
    public static CloseableHttpClient pooledHttpClient()
    {
        if (null != pooledHttpClient)
        {
            return pooledHttpClient;
        }

        createHttpClientConnectionManager();
        log.info(" Apache httpclient 初始化HTTP连接池
starting===");
        //请求配置实例
        RequestConfig.Builder requestConfigBuilder =
            RequestConfig.custom();
        //读取数据的超时设置
        requestConfigBuilder.setSocketTimeout(SOCKET_TIMEOUT);
        //建立连接的超时设置
        requestConfigBuilder.setConnectTimeout(CONNECT_TIMEOUT);
        //从连接池获取连接的等待超时时间设置
    }
}
```

```
requestConfigBuilder.setConnectionRequestTimeout (REQUEST_TIMEOUT);

RequestConfig config = requestConfigBuilder.build();

//httpClient建造者实例
HttpClientBuilder httpClientBuilder =
HttpClientBuilder.create();

//设置连接池管理器
httpClientBuilder.setConnectionManager(
    httpClientConnectionManager);

//设置HTTP请求配置信息
httpClientBuilder.setDefaultRequestConfig(config);

//httpClient默认提供了一个Keep-Alive策略
//这里进行定制：确保客户端与服务端在长连接的保持时长一致
httpClientBuilder.setKeepAliveStrategy(
    new
ConnectionKeepAliveStrategy()

{
    @Override
    public long getKeepAliveDuration (
        HttpResponse response, HttpContext
context)
    {
        //获取响应头中HTTP.CONN_KEEP_ALIVE中的Keep-Alive部
分值
    }
}
```

```
//如果服务端响应"Keep-Alive: timeout=60", 表示保持  
时长为60秒  
//则客户端也设置连接的保持时长为60秒  
//目的：确保客户端与服务端在长连接的保持时长一致  
HeaderElementIterator it = new  
BasicHeaderElementIterator  
  
(response.headerIterator(HTTP.CONN_KEEP_ALIVE));  
while (it.hasNext())  
{  
    HeaderElement he = it.nextElement();  
    String param = he.getName();  
    String value = he.getValue();  
    if (value != null && param.equalsIgnoreCase  
        ("timeout"))  
    {  
        try  
        {  
            return Long.parseLong(value) *  
1000;  
        } catch (final NumberFormatException  
ignore)  
        {  
        }  
    }  
}  
//如果服务端响应头中没有设置保持时长，则使用客户端统一定
```

义时长为600秒

```
        return KEEP_ALIVE_DURATION;
    }
}

//实例化：全局的池化HTTP客户端实例
pooledHttpClient = httpClientBuilder.build();
log.info(" Apache httpclient 初始化HTTP连接池
finished===");
//启动定时处理线程：对异常和空闲连接进行关闭
startExpiredConnectionsMonitor();
return pooledHttpClient;
}
}
```

对于同一条HTTP长连接，服务端会设置一个保持时长，客户端也会有一个保持时长，因此需要尽量保证双方的保持时长一致。在创建长连接时，有的服务端（如Nginx）可以通过设置将自己的保持时长值返回给客户端。所以，客户端在设置保持时长时，可以优先获取服务端返回的保持时长，如果没有，可以退而求其次，使用自己配置的保持时长。在上面的代码中，Apache HttpClient通过定制Keep-Alive策略实现类，在长连接建立时优先获取服务端响应头中的保持时长来作为客户端的连接保持时长。

如果服务端和客户端都可以自己配置，则尽量将双方的HTTP长连接的保持时长配置成同一个值。

在创建HttpClient客户端实例时，需要进行
requestConfigBuilder（请求配置建造者）实例配置，其中大致可以

设置三个超时时长，分别为：

(1) CONNECT_TIMEOUT：该选项表示TCP连接的建立时间，也就是三次握手完成的最长时间，超时后一般会抛出 ConnectionTimeOutException。

(2) SOCKET_TIMEOUT：客户端从服务器读取数据的时间长度，相当于数据传输过程中数据包之间间隔的最大时间，超时后一般会抛出 SocketTimeOutException。

(3) REQUEST_TIMEOUT：设置从连接池获取一个连接的请求超时时间，主要指连接池中连接不够用时阻塞等待的超时时间。

10.6.4 Apache HttpClient客户端长连接实验

从连接池中获取连接，然后发送HTTP请求，与前面介绍的单独创建HTTP连接发送请求在代码的编写逻辑上并没有太多的不同，大致有以下三步：

第一步：获取带连接池的HttpClient客户端实例。此步骤的前提是存在一个提前创建、初始化了的静态HttpClient客户端实例或者Spring IOC容器化的HttpClient客户端实例，该实例一般使用单例模式。

第二步：创建一个HTTP请求实例。这一步所创建的HTTP请求实例一般可以为HttpGet、HttpPost、HttpHead、HttpPut、HttpDelete等类型，具体类型需要根据请求头的METHOD方法类型而定。

第三步：发送请求，然后获取响应结果。使用带连接池的HTTP客户端发送请求，在完成发送请求之后，可以通过response响应实例读取到最终的内容，一般会以字符串的形式返回给调用者。

通过带连接池的HttpClient客户端实例发送请求和获取响应的代码大致如下：

```
package com.crazymakercircle.util;  
//省略import  
//HTTP 协议处理帮助类  
@Slf4j  
public class HttpClientHelper  
{  
    /**  
     * 使用带连接池的HTTP客户端，发送GET请求  
     * @param url 连接地址  
     * @return 请求字符串  
     */  
    public static String get(String url)  
    {  
        //1 取得带连接池的客户端  
        CloseableHttpClient client =  
            pooledHttpClient();  
        //2 创建一个HTTP请求实例  
        HttpGet httpGet = new HttpGet(url);  
        //3 使用带连接池的HTTP客户端，发送请求，并且获取结  
        果
```

```
        return poolRequestData(url, client,  
        httpGet);  
  
    }  
  
    /**  
     * 使用带连接池的HTTP客户端，发送请求  
     * @param url      连接地址  
     * @param client   客户端  
     * @param request  post、get或者其他请求  
     * @return 响应字符串  
     */  
  
    private static String poolRequestData(  
            String url, CloseableHttpClient  
client, HttpRequest request)  
  
    {  
        CloseableHttpResponse response = null;  
        InputStream in = null;  
        String result = null;  
        try  
        {  
            //从url中获取HttpHost实例，含主机和端口  
            HttpHost httpHost = getHost(url);  
            //执行HTTP请求  
            response = client.execute(  
                    httpHost, request,  
                    HttpClientContext.create());  
            //获取HTTP响应  
        }  
    }
```

```
        HttpEntity entity =
response.getEntity();
        if (entity != null)
{
    in = entity.getContent();
    result =
IOUtils.toString(in, "utf-8");
}
} catch (IOException e)
{
    e.printStackTrace();
} finally
{
    quietlyClose(in);
    quietlyClose(response);
//无论执行成功还是出现异常，HttpClient 都
会自动处理并保证释放连接
}
return result;
}
//省略其他代码
}
```

接下来，通过调用以上帮助类中的get(String url)方法进行HTTP长连接实验。

这里访问的服务端应用仍然是上一章所编写的HTTP Echo服务，为了在服务端查看连接的状态和信息，将HTTP Echo服务部署在虚拟机上。只有这样，在实验的过程中通过netstat指令才能很方便地在虚拟机上查看HTTP长连接的信息和状态。

调用get(String url)方法访问HTTP Echo服务的实验用例代码如下：

```
package com.crazymakercircle;  
//省略import  
  
public class HTTPKeepAliveTester  
{  
    /**  
     * 测试用例：使用带连接池的Apache HttpClient提交的HTTP请求  
     */  
  
    @Test  
    public void pooledGet() throws IOException,  
    InterruptedException  
    {  
        int index = 1000000;  
        while (--index > 0)  
        {  
            String target = url + index;  
            //使用固定10个线程的线程池发起请求  
            pool.submit(() ->  
            {  
                //使用Apache HttpClient提交的HTTP请求  
            });  
        }  
    }  
}
```

```

        String out = HttpClientHelper.get(target);
        System.out.println("out = " + out);
    });
}

Thread.sleep(Integer.MAX_VALUE);
}

//...
}

```

在Linux虚拟机上，可以通过netstat指令看到具体的TCP连接信息。通过仔细观察可以看到，上面实验中HTTP通信所建立的传输层TCP连接都进行了复用，所以这些HTTP连接都是HTTP长连接，而不是短连接。为什么呢？因为每一轮TCP连接端口的循环输出都是相同的。通过观察实验结果可以看到，netstat指令程序每隔1秒输出一批ESTABLISHED状态的连接（10个），其端口都是相同的。

对于使用netstat指令所看到的连接信息，部分节选如下：

```

[root@localhost ~]# while [ 1 -eq 1 ] ; do netstat -antp|grep
18899    ; sleep 2; echo ; done
tcp6      0      0 ::::18899      ::::*
LISTEN      8422/java
...
tcp6      0      339 192.168.233.128:18899  192.168.233.1:45363
ESTABLISHED 8422/java
tcp6      0      339 192.168.233.128:18899  192.168.233.1:45368
ESTABLISHED 8422/java
tcp6      0      339 192.168.233.128:18899  192.168.233.1:45364

```

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45362

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45366

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45359

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45361

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45360

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45365

ESTABLISHED 8422/java

tcp6 0 339 192.168.233.128:18899 192.168.233.1:45367

ESTABLISHED 8422/java

tcp6 0 0 :::18899 ::::*

LISTEN 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45363

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45368

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45364

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45362

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45366

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45359

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45361

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45360

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45365

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45367

ESTABLISHED 8422/java

tcp6 0 0 :::18899 ::::*

LISTEN 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45363

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45368

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45364

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45362

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45366

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45359

ESTABLISHED 8422/java

tcp6 0 0 192.168.233.128:18899 192.168.233.1:45361

```
ESTABLISHED 8422/java
tcp6      0      0 192.168.233.128:18899  192.168.233.1:45360
ESTABLISHED 8422/java
tcp6      0      0 192.168.233.128:18899  192.168.233.1:45365
ESTABLISHED 8422/java
tcp6      0      0 192.168.233.128:18899  192.168.233.1:45367
ESTABLISHED 8422/java
```

10.6.5 Nginx承担客户端角色时的长连接技术

无论是在传统的Nginx + Tomcat架构中还是在当前主流的Nginx + Spring Cloud架构中，反向代理Nginx都承担了两种角色：对于终端用户（或者下游代理）来说，Nginx承担了服务端角色；对于上游的真实服务器（如Web服务器）来说，Nginx承担了客户端角色。

在反向代理和上游服务器之间，Nginx承担了客户端角色，此时在Nginx一端使用短连接肯定是不合适的。为什么呢？如果Nginx服务器使用短连接去请求上游服务器，当请求完成后Nginx进行连接的主动断开，就会造成Nginx所在的服务器产生大量的TIME_WAIT连接，压力增大时很可能导致Nginx服务器无法提供新的连接。所以，在反向代理和上游服务器之间一定需要使用HTTP长连接进行通信。

面对上面的问题，需要调整Nginx的参数，在Nginx与上游服务器上都保持一定数据量的长连接，这样就能有效地避免连接的频繁创建与释放。与Apache HttpClient类似，Nginx也有自己的类似客户端TCP连接池的连接管理组件。对于池中单个TCP连接的保持配置，可以通过在upstream区块中使用keepalive指令完成，该指令的具体格式如下：

语法: `keepalive connections;`

默认值: —

上下文: `upstream`

`keepalive`指令的`connections`参数用于设置到上游服务器之间保持的长连接的最大数量，这些连接保留在每个工作进程的连接池中。池化的TCP连接超过此数目时，最近使用的最少的TCP长连接将关闭。

说明

在Nginx承担客户端角色时，`keepalive`指令用于设置长连接的参数，所以只能用于`upstream`区块中，不能用于`http`、`server`、`location`区块中。有关`upstream`区块的原理和具体介绍，请参考笔者的另一本专著《Spring Cloud、Nginx高并发核心编程》。

`keepalive`指令的使用示例大致如下：

```
upstream memcached_backend {  
    server 127.0.0.1:11211;  
    server 10.0.0.2:11211;  
    //可以理解为连接池可以缓存32个连接  
    keepalive 32;  
}
```

需要特别注意的是：当反向代理Nginx承担客户端角色时，`keepalive`指令并没有限制可以打开的到上游服务器之间的连接总数。

该指令的connections参数不能设置为一个太大的值，如果其值太大，Nginx与上游服务器将保持太多的长连接，可能导致上游服务器连接耗尽，将无法处理新传入的连接。

要想keepalive指令生效，还需要两个必要的条件：

(1) 需要强制Nginx与后端上游服务器之间使用1.1版本的HTTP，因为该版本的HTTP连接默认是长连接。

(2) 反向代理对于下游来说是透明的，下游可能发送Connection:close头部关闭TCP连接。如果下游客户端传过来Connection:close头部，直接被Nginx转发到上游的后端服务器，那么后端服务器会以为Nginx要求关闭连接，此时后端服务器将主动关闭TCP连接，Nginx与后端服务器之间的TCP连接也就无法保持了。所以，为了保证反向代理到其上游之间的TCP连接能复用，需要将下游客户端发送过来的HTTP请求头Connection:close重置掉，将其值重置成空白字符串。

综合以上两点，在Nginx上负责下游HTTP请求路由和转发的location配置区块中，需要使用proxy_http_version指令和proxy_set_header指令完成HTTP请求头的配置优化，具体代码如下：

```
server {  
    listen 8080 default_server;  
    server_name "";  
    //...  
    //处理下游客户端请求转发的location配置区块  
    location / {
```

```
proxy_pass http://memcached_backend;  
  
                                //转发之前，进行请求头重置，重置HTTP协议的  
版本为1.1  
  
proxy_http_version 1.1;  
  
                                //转发之前，进行请求头重置，重置请求  
Connection:close头部值  
  
proxy_set_header Connection "";  
}  
}  
}
```

经过以上调整就能在Nginx作为客户端角色时实现与上游服务器之间使用长连接进行HTTP通信，从而最大限度地实现其下层的TCP连接复用。

第11章 WebSocket原理与实战

WebSocket是一种全双工通信的协议，其通信在TCP连接上进行，所以属于应用层协议。WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在WebSocket编程中，浏览器和服务器只需要完成一次升级握手就直接可以创建持久性的连接，并进行双向数据传输。

对于WebSocket的Java开发，Java官方发布了JSR-356规范，该规范的全称为Java API for WebSocket。不少Web容器（如Tomcat、Jetty等）都支持JSR-356规范，提供了WebSocket应用开发API。Tomcat从7.0.27开始支持WebSocket，从7.0.47开始支持JSR-356规范。

无论是Tomcat还是Jetty，其性能在高并发场景下的表现并不是非常理想。Netty是一款高性能的NIO网络编程框架，在通信连接数与信息量激增时表现依然出色。所以，编写WebSocket服务端程序时，一般基于Netty框架进行编写。

说明

本章介绍了一个小的WebSocket服务端演示程序——WebSocket Echo。如果能够顺利掌握此程序，可以开启下一个进阶实验：参考疯狂创客圈的Netty + WebSocket开源项目，完成一个具备在线聊

天、在线推送功能的综合性WebSocket实战练习。该开源项目的地址为https://gitee.com/crazymaker/Websocket_chat_room。

11.1 WebSocket协议简介

WebSocket协议的目标是在一个独立的持久连接上提供全双工双向通信。客户端和服务器可以向对方主动发送和接收数据。WebSocket通信协议于2011年被IETF发布为RFC6455标准，后又发布了RFC7936标准补充规范。WebSocket API也被W3C (World Wide Web Consortium, 万维网联盟) 定为标准。

11.1.1 Ajax短轮询和Long Poll长轮询的原理

在WebSocket双向通信技术之前，浏览器与服务器之间的双向通信大致有两种方式：Ajax短轮询和Long Poll长轮询。

1. Ajax短轮询

Ajax短轮询即浏览器周期性地向服务器发起HTTP请求，不管服务器是否真正获取到数据，都会向浏览器返回响应，如图11-1所示。浏览器通过HTTP 1.1的持久连接（建立一次TCP连接，发送多个请求），可以在建立一次TCP连接之后发起多个异步请求。

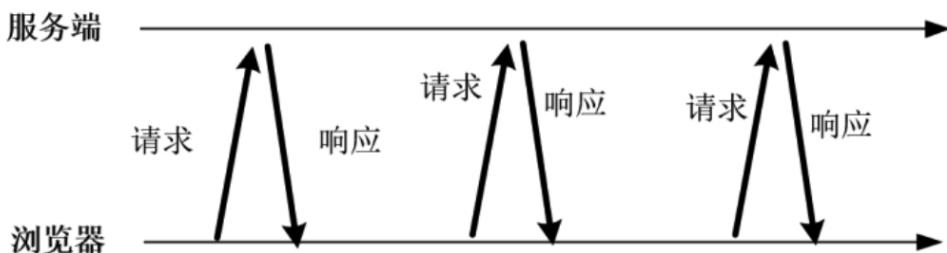


图11-1 Ajax短轮询示意图

Ajax短轮询的原理非常简单，让浏览器隔几秒就发送一次请求，询问服务器是否有新信息。在Ajax短轮询中，每个请求对应一个响应。这种模式有很明显的缺点，即浏览器需要不断地向服务器发出请求，然而HTTP请求在每次发送时都会带上很长的请求头部字段，其中真正有效的数据可能只是很小的一部分（如Cookie字段），显然会浪费服务器带宽和CPU资源。

是否可以通过加大Ajax的轮询间隔时间来降低资源的浪费比例呢？比如，将轮询间隔改为10秒或者更长。问题是轮询时间长了，对于实时性要求比较高的项目来说，客户端页面更新数据的速度也就太慢了，违背了双向通信的初衷。

2. Long Poll长轮询

Long Poll长轮询在原理上跟Ajax短轮询差不多，都是采用轮询的方式，不过采取的是服务端阻塞模型。在轮询过程中，服务端在收到浏览器的请求后，如果暂时没有消息需要推送给浏览器，服务端就会一直阻塞，不会立即返回响应。直到服务端有消息才返回响应给客户端。客户端收到响应之后，开始发送下一轮的轮询请求，如此周而复始。

无论是Ajax短轮询还是Long Poll长轮询，都不是最好的双向通信方式，都需要很多资源。WebSocket则不同，该协议只需要经过一次HTTP请求就可以做到源源不断的信息传送了，当传输协议完成HTTP到WebSocket协议升级后，服务端就可以主动推送信息给客户端，高效率地实现双向通信。

11.1.2 WebSocket与HTTP之间的关系

WebSocket的最大特点就是全双工通信，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息。WebSocket与HTTP之间的关系是：WebSocket是一个新协议，通信过程与HTTP基本没有关系，只是为了兼容现有浏览器，所以在握手阶段使用了HTTP。

WebSocket协议的握手和通信过程如图11-2所示。

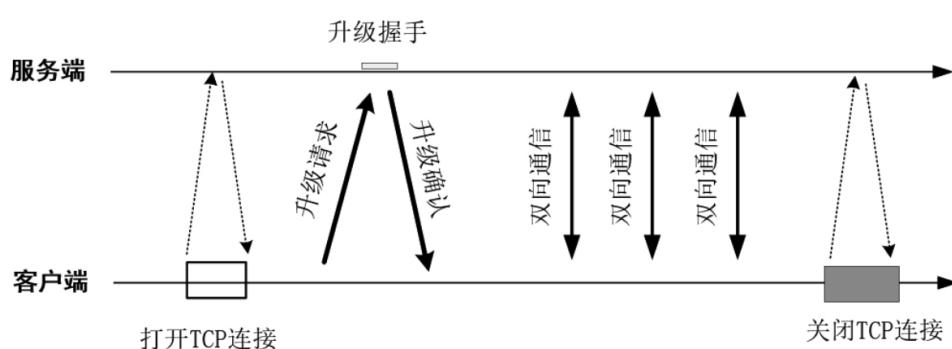


图11-2 WebSocket协议的握手和通信过程

WebSocket协议与HTTP一样，处于TCP/IP协议栈的应用层，都是TCP/IP协议的子集。WebSocket协议和HTTP有一个显著的不同：HTTP是单向通信协议，只有客户端发起HTTP请求，服务端才会返回数据；WebSocket协议是双向通信协议，在建立连接之后，客户端和服务器都可以主动向对方发送或接收数据。

WebSocket协议和HTTP还是有关系的：WebSocket的通信连接建立的前提需要借助HTTP，完成通信连接建立之后，通信连接上的双向通信就与HTTP无关了。

11.2 WebSocket回显演示程序开发

本节通过一个WebSocket回显程序开发实战介绍如何使用JavaScript开发WebSocket客户端程序、如何使用Netty开发WebSocket服务端程序。

本WebSocket回显演示程序的功能：客户端通过WebSocket向服务器发送任意一段字符串消息，服务器将该消息通过WebSocket回写到客户端，最后客户端将回显消息展现到网页。

11.2.1 WebSocket回显程序的客户端代码

WebSocket回显程序客户端通过JavaScript完成以下操作：

- (1) 建立WebSocket连接。
- (2) 监听WebSocket接收到的消息，并且展示在网页上。
- (3) 通过WebSocket连接发送消息给服务端。

WebSocket回显演示客户端的效果，大致如图11-3所示。



图11-3 WebSocket回显演示客户端的效果

使用JavaScript实现WebSocket协议通信相对简单，这里分为三个步骤进行介绍。

(1) 建立WebSocket连接

使用JavaScript建立WebSocket连接的代码大致如下：

```
socket = new WebSocket("ws://192.168.0.5:18899/ws", "echo");
```

以上调用的`WebSocket()`方法的第一个参数为服务端的WebSocket监听URL地址，第二个参数为服务端配置的WebSocket子协议（业务协议），子协议为应用程序自己使用的某个标识或者命名，客户端与服务端保持一致即可。

WebSocket有自己的协议规范，其URL规则与HTTP的URL规则不同。

WebSocket中未加密的URL Schema为ws://，而不是http://。

WebSocket中加密的URL Schema为wss://，而不是https://。

建立WebSocket连接时，传递的URL参数没有同源策略的限制。那么什么是同源策略呢？如果两个通信协议的URL的主机名（域名或者IP）和端口都相同，则两个URL是同源的。同源策略是浏览器的一个安全功能，不同源的客户端脚本在没有明确授权的情况下不能读写对方资源。WebSocket并不受同源策略的限制，可以向不同源的URL发起WebSocket连接请求。

（2）监听WebSocket连接的open事件

在成功建立WebSocket连接后，客户端可以通过onopen()方法监听连接的open事件，在连接成功之后，可以进行后续的业务处理，大致代码如下：

```
socket.onopen = function (event) {  
    var target =  
    document.getElementById('responseText');  
    target.value = "Web Socket 连接已经开启！";  
};
```

（3）监听WebSocket连接的message消息事件

当服务端的消息推送过来时，客户端会触发message消息事件，客户端代码可以通过onmessage()方法监听该message消息事件，然后在监听方法中获取所接收到的服务端数据。大致的代码如下：

```
socket.onmessage = function (event) {  
    var ta = document.getElementById('responseText');  
    ta.value = ta.value + '\n' + event.data  
};
```

完整的WebSocket回显演示程序的客户端JavaScript脚本大致如下：

```
<script type="text/javascript">  
var socket;  
  
if (!window.WebSocket) {  
    window.WebSocket = window.MozWebSocket;  
}  
  
//获取浏览器上URL中的主机名称  
  
var domain = window.location.host;  
  
if (window.WebSocket) {  
    //建立WebSocket连接  
  
    socket = new WebSocket("ws://" + domain + "/ws", "echo");  
  
    socket.onmessage = function (event) {  
        var ta = document.getElementById('responseText');  
        ta.value = ta.value + '\n' + event.data  
    };  
  
    //连接打开事件  
  
    socket.onopen = function (event) {  
        var target =  
            document.getElementById('responseText');
```

```
target.value = "Web Socket 连接已经开启!";
};

//连接关闭事件

socket.onclose = function (event) {

    var target =
document.getElementById('responseText');

    target.value = ta.value + "Web Socket 连接已经断开";

};

} else {

    alert("Your browser does not support Web Socket.");
}

//发送WebSocket消息，在JavaScript发送WebSocket消息时调用

function send(message) {

    if (!window.WebSocket) {

        return;
    }

    if (socket.readyState == WebSocket.OPEN) {

        //通过套接字发送消息

        socket.send(message);
    } else {

        alert("The socket is not open.");
    }
}

</script>
```

以上代码处于演示工程NettyWebSocketServerDemo子模块的resources资源目录的index.html文件中，大家可以通过随书源码查看。

11.2.2 WebSocket相关的Netty内置处理类

接下来介绍基于Netty进行WebSocket服务端的开发。WebSocket协议中大致包含了5种类型的数据帧，与这5种数据帧相对应，Netty包含了5种WebSocket数据帧的封装类型。这些类型都是WebSocketFrame类的子类，具体如表11-1所示。

表11-1 WebSocketFrame数据帧子类

WebSocket 数据帧名称	功 能
BinaryWebSocketFrame	封装二进制数据的 WebSocketFrame 数据帧
TextWebSocketFrame	封装文本数据的 WebSocketFrame 数据帧
CloseWebSocketFrame	表示一个 CLOSE 结束请求，数据帧中包含结束的状态和结束的原因，此帧属于控制帧
ContinuationWebSocketFrame	当发送的内容多于一个数据帧时，消息将被拆分为多个 WebSocketFrame 数据帧发送，而此类型的数据帧专用于发送剩余的内容。ContinuationWebSocketFrame 可以发送后续的文本或者二进制数据帧
PingWebSocketFrame	Ping 和 Pong 是 WebSocket 通信中的心跳帧，用来保证客户端是在线的，一般来说只有服务端给客户端发送 Ping，然后客户端发送 Pong 来回应，表明自己仍然在线。PingWebSocketFrame 属于控制帧，其对应的协议报文中的操作码 opcode 值为 0x9
PongWebSocketFrame	此帧是对 PingWebSocketFrame 请求的响应帧，也属于控制帧，其对应的协议报文中的操作码 opcode 值为 0xA

与服务端WebSocket通信相关的Netty内置Handler处理器，主要如表11-2所示。

表11-2 与WebSocket相关的Netty服务端的Handler处理器

处理器名称	功 能
WebSocketServerProtocolHandler	负责协议开始升级时的请求处理，也就是开启握手处理。另外，在协议升级握手完成后的 WebSocket 通信过程中，此处理器还负责对 WebSocket 协议的三个控制帧 Close、Ping、Pong 进行处理
WebSocketServerProtocolHandshakeHandler	此处理器负责进行协议升级握手处理；在握手完成后，此处处理器会触发 HANDSHAKE_COMPLETE 用户事件，表示握手完成
WebSocketFrameEncoder	WebSocketFrame 数据帧编码器，负责 WebSocket 数据帧编码。在握手时，针对不同的 WebSocket 协议版本，握手处理器会在流水线上装配对应的编码器子类
WebSocketFrameDecoder	WebSocketFrame 数据帧解码器，负责 WebSocket 数据帧解码。在握手时，针对不同的 WebSocket 协议版本，握手处理器会在流水线上装配对应的解码器子类

以上四个内置处理器中，WebSocketServerProtocolHandler是非常关键的处理器，负责开始升级握手和控制帧的处理，可以理解为握手处理器。握手完成后，双方的通信协议会从HTTP升级到WebSocket协议，老的HTTP协议处理器会被该握手处理器替换掉，新的与WebSocket协议相关的解码器会被成功地添加到流水线上。

以WebSocket回显演示程序为例，在协议升级之前，通道处理流水线的状态如图11-4所示。

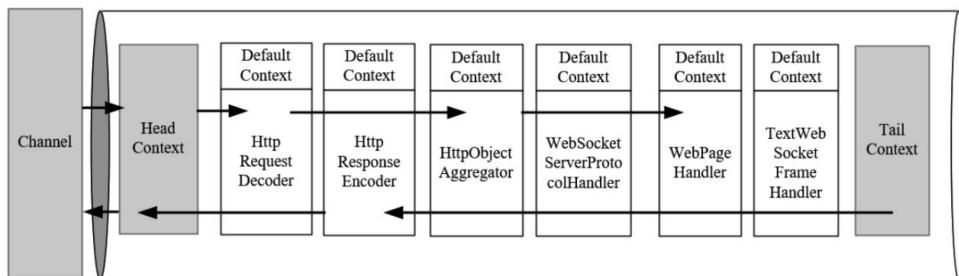


图11-4 WebSocket协议升级之前的通道处理流水线

在握手升级过程中，握手处理器

WebSocketServerProtocolHandshakeHandler会被加入到通道处理流水线上，负责进行协议的升级握手。握手完成之后，握手处理器会将解码器HttpRequestDecoder替换为WebSocketFrameDecoder对应WebSocket版本的子类实例，也会将编码器HttpResponseEncoder替换为WebSocketFrameEncoder对应版本的子类实例。

为了尽可能提高性能，在业务处理器中，用户程序可以通过监听WebSocket的握手完成事件将后续WebSocket通信过程中不需要用到的处理器移除。例如，本演示实例中的网页处理器WebPageHandler在WebSocket握手之前需要，在完成了WebSocket握手之后就不再需要了。

以WebSocket回显演示程序为例，在握手完成协议升级之后，通道处理流水线的状态如图11-5所示。

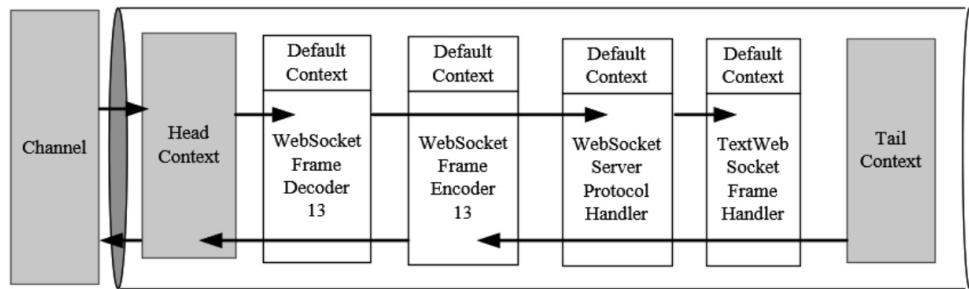


图11-5 WebSocket协议升级之后的通道处理流水线

WebSocket的解码器WebSocketFrameDecoder和编码器

WebSocketFrameEncoder有多个版本，具体使用哪个版本是由握手处理过程中服务端根据客户端在握手请求中所发送的支持WebSocket协议版

本确定的。例如，在演示程序的执行过程中，客户端发送的协议版本是13，则服务端使用WebSocketFrameEncoder13和WebSocketFrameDecoder13这组编解码器。

目前Netty 4.1版本可以处理的WebSocket协议版本包括00、07、08、13四种，每种版本都有配套的编码器、解码器、握手处理类。在协议升级的时候，Netty会根据握手请求的Sec-WebSocket-Version头部的协议版本值来决定使用哪一个版本的WebSocket协议。比如Sec-WebSocket-Version头部的版本值为13，则装配到流水线的WebSocket编解码器分别为WebSocketFrameEncoder13和WebSocketFrameDecoder13。

说明

Sec-WebSocket-Version头部是WebSocket协议中一个重要的请求头。有关WebSocket协议的内容，请查看后面的章节。

11.2.3 WebSocket的回显服务器

WebSocket回显服务器的代码大致如下：

```
package com.crazymakercircle.netty.Websocket;  
//省略import  
@Slf4j  
public final class WebSocketEchoServer
```

```
{  
    //流水线装配器  
    static class EchoInitializer extends  
        ChannelInitializer<SocketChannel>  
{  
        @Override  
        public void initChannel(SocketChannel ch)  
        {  
            ChannelPipeline pipeline = ch.pipeline();  
            //HTTP请求解码器  
            pipeline.addLast(new HttpRequestDecoder());  
            //HTTP响应编码器  
            pipeline.addLast(new HttpResponseEncoder());  
            //HttpObjectAggregator 将HTTP消息的多个部分合成一条完整  
            //的HTTP消息  
            pipeline.addLast(new HttpObjectAggregator(65535));  
            //WebSocket协议处理器，配置WebSocket的监听URI、协议包长度  
            //限制  
            pipeline.addLast(  
                new WebSocketServerProtocolHandler("/ws",  
                    "echo",  
                    true,  
                    10 * 1024));  
            //增加网页的处理逻辑  
            pipeline.addLast(new WebPageHandler());  
            //TextWebSocketFrameHandler 是自定义的WebSocket业务处  
            //理器
```

```
        pipeline.addLast(new TextWebSocketFrameHandler());  
    }  
}  
  
/**  
 * 启动  
 */  
public static void start(String ip) throws Exception  
{  
    //创建连接监听reactor 轮询组  
    EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
    //创建连接处理 reactor 轮询组  
    EventLoopGroup workerGroup = new NioEventLoopGroup();  
    try  
    {  
        //服务端启动引导实例  
        ServerBootstrap b = new ServerBootstrap();  
        b.group(bossGroup, workerGroup)  
            .channel(NioServerSocketChannel.class)  
            .handler(new  
LoggingHandler(LogLevel.DEBUG))  
            .childHandler(new EchoInitializer());  
  
        //监听端口，返回同步通道  
        Channel ch = b.bind(18899).sync().channel();  
        log.info("WebSocket服务已经启动 http://{}:  
{}/",ip,18899);  
    }  
}
```

```
        ch.closeFuture().sync();

    } finally

    {

        bossGroup.shutdownGracefully();

        workerGroup.shutdownGracefully();

    }

}

}
```

以上演示程序构造了一个Netty内置的握手处理器WebSocketServerProtocolHandler实例，并且为握手处理器实例设置了WebSocket的URL和子协议，以及最大的WebSocket传输帧的大小。当客户端通过HTTP对握手处理器配置的URL和子协议发起请求时，服务端开始WebSocket的握手处理和协议升级。

在上面的代码中，演示程序所设置的WebSocket服务监听的URL为“/ws”、子协议为“echo”。这就要求客户端在发起WebSocket连接时使用同样的URL和子协议，否则会连接失败。所以，当服务端收到客户端的URL为“/ws”、子协议为“echo”的HTTP请求时，握手处理器WebSocketServerProtocolHandler将启动协议升级机制，着手将HTTP升级为WebSocket协议，握手完成之后，双方正式进入WebSocket双向通信阶段。

11.2.4 WebSocket的业务处理器

在握手处理之前，握手处理器WebSocketServerProtocolHandshakeHandler会被加入通道处理流水线

上，负责升级握手。握手完成之后，会触发HANDSHAKE_COMPLETE事件，该事件可以被业务处理器监听和处理。

WebSocket回显服务器的业务处理器为TextWebSocketFrameHandler，在监听到握手完成事件之后，将WebSocket通信中不需要的WebPageHandler网页处理器移除。

演示程序的业务处理器TextWebSocketFrameHandler的代码如下：

```
package com.crazymakercircle.netty.Websocket;
//省略import
@Slf4j
public class TextWebSocketFrameHandler extends
SimpleChannelInboundHandler<WebSocketFrame>
{
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
                               WebSocketFrame frame)
        throws Exception
    {
        //Ping 和Pong 帧已经被前面WebSocketServerProtocolHandler处
理器处理过了
        if (frame instanceof TextWebSocketFrame)
        {
            //取得WebSocket的通信内容
            String request = ((TextWebSocketFrame)
frame).text();
        }
    }
}
```

```
    log.debug("服务端收到: " + request);

    //回显字符串

    String echo = Dateutil.getTime() + ":" + request;

    //构造TextWebSocketFrame文本帧，用于回复

    TextWebSocketFrame echoFrame = new

TextWebSocketFrame(echo);

    //发送回显字符串

    ctx.channel().writeAndFlush(echoFrame);

} else

{

    //如果不是文本消息，抛出异常

    //本演示不支持二进制消息

    String message = "unsupported frame type: " +


frame.getClass().getName();

    throw new UnsupportedOperationException(message);

}

}

//处理用户事件

@Override

public void userEventTriggered(ChannelHandlerContext ctx,

                                Object evt) throws

Exception

{

    //判断是否为握手成功事件，该事件表明通信协议已经升级为 WebSocket

    协议
```

```
if (evt instanceof  
  
WebSocketServerProtocolHandler.HandshakeComplete)  
{  
    //握手成功，移除 WebPageHandler，因此将不会接收到任何HTTP  
请求  
    ctx.pipeline().remove(WebPageHandler.class);  
    log.debug("WebSocket HandshakeComplete 握手成功");  
    log.debug("新的WebSocket 客户端加入，通道为：" +  
ctx.channel());  
}  
else  
{  
    super.userEventTriggered(ctx, evt);  
}  
}  
}
```

当和客户端的WebSocket握手成功完成之后，流水线上的握手处理器实例会触发一个

WebSocketServerProtocolHandler.HandshakeComplete事件，在业务处理器中可以进行监听。监听到该事件之后，业务处理器可以通过 instanceof运算符进行判断，如果接收到的事件为握手完成事件，就说明通信的协议已经完成从HTTP到WebSocket协议的升级，接下来双方开始WebSocket的通信，HTTP协议的请求将不再被服务端处理。

客户端发送过来的WebSocket数据帧在解码后会被TextWebSocketFrameHandler（自定义的业务处理器）的

channelRead0()方法读取到。在该方法中，程序将对WebSocket数据帧进行判断，如果接收到的是TextWebSocketFrame数据帧，则通过其text()方法取得数据帧中的文本内容，然后构建一个新的TextWebSocketFrame文本帧，并回写到客户端。

由于Netty的WebSocketServerProtocolHandler协议处理器已经帮助处理诸如升级握手、Close、Ping、Pong控制帧等基础性工作，只有Text和Binary两种消息数据帧会被发送到其后面的业务处理器，因此业务处理可以不用理会Close、Ping、Pong等控制帧，这样也简化了业务处理器的处理逻辑。

Netty在进行WebSocket解码时，WebSocketFrameDecoder13解码处理器会通过请求数据帧中的Opcode标志来判断读取的数据帧是文本类型还是二进制类型，然后相对应地解析成文本TextWebSocketFrame帧实例或二进制BinaryWebSocketFrame帧实例，再发送给流水线后面的业务处理器。

数据帧的类型包含在WebSocket数据帧的操作码中。一个WebSocket请求数据帧抓包示意图大致如图11-6所示。

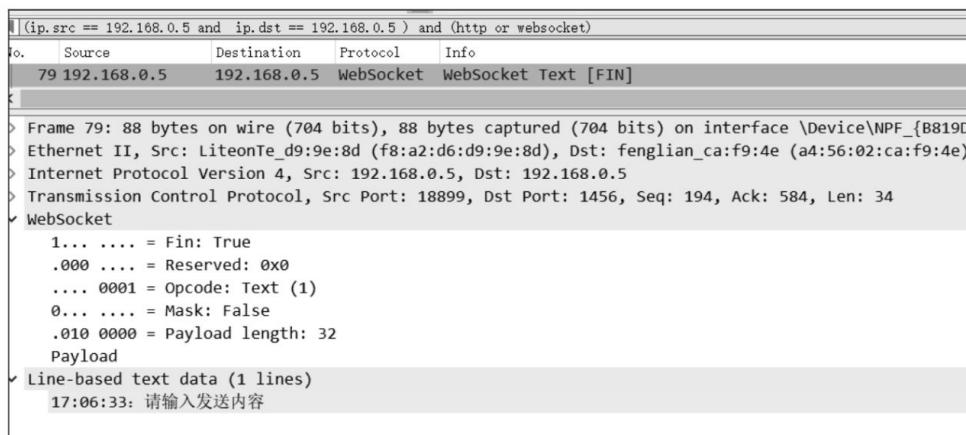


图11-6 一个WebSocket请求数据帧抓包示意图

11.3 WebSocket协议通信的原理

接下来结合WebSocket协议的通信数据包为大家介绍一下WebSocket协议通信的原理。

11.3.1 抓取WebSocket协议的本机数据包

WebSocket回显演示程序的用例在本地开发机器（localhost）执行，而浏览器发出的WebSocket请求也是发向本地localhost或者127.0.0.1的。默认情况下，WireShark是抓取不到通信报文的，需要进行特殊的设置才可以。具体的抓包准备请参考测试用例中的注释说明。WebSocket回显演示程序的测试用例代码，具体如下：

```
package com.crazymakercircle.NettyTest;  
//省略import  
/**  
 * WebSocket回显服务器的测试用例  
 **/  
@Slf4j  
public class WebSocketEchoTester  
{  
    @Test  
    public void startServer() throws Exception  
    {  
        //抓包说明：由于WireShark只能抓取经过所监控的  
        网卡的数据包
```

```
//因此请求到localhost的本地包，默认是不能抓取  
到的。  
  
//如果要抓取本地的调试包，需要通过route指令增  
加服务器IP的路由表项配置  
  
//只有这样，让发往本地localhost的报文才会经过  
路由网关所绑定的网卡 //从而，发往  
localhost的本地包就能被抓包工具从监控网卡抓取到  
  
//具体的办法是通过增加路由表项来完成，其命令为  
route add，下面是一个例子  
  
//route add 192.168.0.5 mask  
255.255.255.255 192.168.0.1  
  
//以上命令表示：目标为192.168.0.5的报文，经  
过192.168.0.1网关绑定的网卡  
  
//该路由项在使用完毕后，建议删除，其删除指令如  
下：  
  
//route delete 192.168.0.5 mask  
255.255.255.255 192.168.0.1删除  
  
//如果没有删除，则所有本机报文都经过网卡到达路  
由器  
  
//然后，绕一圈再回来，会很消耗性能  
//如果该路由表项并没有保存，在电脑重启后将会失  
效  
  
//注意：以上用到的本地IP和网关IP需要结合自己的  
电脑网卡和网关去更改  
  
//启动WebSocket回显服务器
```

```

    WebSocketEchoServer.start("192.168.0.5");
}

}

```

在抓包准备工作完成之后，启动WireShark抓包工具，监控通信网卡。准备工作完成之后，可以通过以上测试用例去启动WebSocket回显服务端程序，然后在浏览器打开回显服务的客户端网页，通过该网页对服务器发起WebSocket连接。

11.3.2 WebSocket握手过程

前面讲到WebSocket是基于HTTP来完成握手和协议升级的，通过WireShark抓包工具抓取数据包可以清晰地看到这一点。通过WireShark抓包工具抓取到的WebSocket回显演示程序的数据帧大致如图11-7所示。

The screenshot shows a Wireshark capture window titled '*WLAN'. The packet list pane displays several frames. A search filter at the top is set to '(ip.src == 192.168.0.5 and ip.dst == 192.168.0.5) and (http or websocket)'. The columns in the packet list are: No., Source, Destination, Protocol, and Info. The data is as follows:

No.	Source	Destination	Protocol	Info
35	192.168.0.5	192.168.0.5	HTTP	GET / HTTP/1.1
41	192.168.0.5	192.168.0.5	HTTP	HTTP/1.1 200 OK (text/html)
73	192.168.0.5	192.168.0.5	HTTP	GET /favicon.ico HTTP/1.1
79	192.168.0.5	192.168.0.5	HTTP	HTTP/1.1 200 OK (text/html)
83	192.168.0.5	192.168.0.5	HTTP	GET /ws HTTP/1.1
92	192.168.0.5	192.168.0.5	HTTP	HTTP/1.1 101 Switching Protocols
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN] [MASKED]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN] [MASKED]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN] [MASKED]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN] [MASKED]
1...	192.168.0.5	192.168.0.5	WebSocket	WebSocket Text [FIN]

图11-7 WebSocket回显演示程序通信数据帧列表

WebSocket是应用层协议，是TCP/IP协议的子集，通过HTTP/1.1协议的101状态码完成握手。也就是说，WebSocket协议的建立需要先借助HTTP，在服务器返回101状态码之后才可以进行WebSocket全双工的双向通信，协议切换（或者升级）之后的通信就与HTTP协议没有任何关系了。

客户端创建WebSocket连接的握手请求是通过HTTP完成的。具体来说，客户端发起握手时需要向服务端WebSocket的监控URL（本节的Echo演示程序中URL为/ws）发送GET请求，其握手请求报文如图11-8所示。

The screenshot shows a network capture in Wireshark. A single frame is selected, which is a GET request to the URL /ws. The request details pane shows the following headers:

- Request Method: GET
- Request URI: /ws
- Request Version: HTTP/1.1
- Host: 192.168.0.5:18899
- Connection: Upgrade
- Pragma: no-cache
- Cache-Control: no-cache
- User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
- Upgrade: websocket
- Origin: http://192.168.0.5:18899
- Sec-WebSocket-Version: 13
- Accept-Encoding: gzip, deflate
- Accept-Language: zh-CN,zh;q=0.9
- Sec-WebSocket-Key: HYrapW5mTV9Vxb6YFnXFLA==
- Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
- Sec-WebSocket-Protocol: echo

图11-8 客户端创建WebSocket连接的HTTP握手请求报文

可以发现，以上报文和一个一般的HTTP报文没啥区别。但是，根据WebSocket协议规范，创建WebSocket连接的握手请求必须是一个HTTP请求，请求的方法必须是GET，并且HTTP协议版本不可以低于1.1。

握手请求HTTP报文需要携带一些WebSocket协议规范约定的请求头，主要有以下几个：

(1) Sec-WebSocket-Key请求头

该请求头的值是一个Base64编码的值，是客户端浏览器随机生成的，服务端从请求（HTTP的请求头）信息中提取Sec-WebSocket-Key，服务端会对此值进行加密，之后将加密结果响应给客户端。WebSocket协议规范约定，握手报文必须包含Sec-WebSocket-Key请求头。

(2) Upgrade请求头

WebSocket协议规范约定，握手请求报文必须包含Upgrade请求头，并且此请求头的值必须包含“WebSocket”。

(3) Connection请求头

WebSocket协议规范约定，握手报文必须包含Connection请求头，并且此请求头的值必须包含“Upgrade”。

(4) Sec-WebSocket-Version请求头

WebSocket协议规范约定，握手报文必须包含Sec-WebSocket-Version请求头，其值若为13，则表示客户端支持WebSocket的协议版本号为13（该版本为目前最为常用的协议版本，其余版本号包括00、07、08等）。

(5) Sec-WebSocket-Protocol请求头

Sec-WebSocket-Protocol是表示通信使用的子协议，属于用户自定义的协议名称，只要与服务端的子协议名称保持一致即可，否则会握手失败。

其他的握手请求HTTP报文头部字段，大部分与HTTP协议头部字段含义相同，具体可以参见WebSocket标准规范RFC6455，这里不再赘述。

服务端在收到客户端的URL为”/ws”、子协议为”echo”的握手请求后，握手处理器WebSocketServerProtocolHandler将启动服务端升级握手的机制，进行握手检查。如果客户端能够满足WebSocket通信的要求，握手处理器就会向客户端发送Switching Protocols（转换协议）HTTP响应报文，具体如图11-9所示。

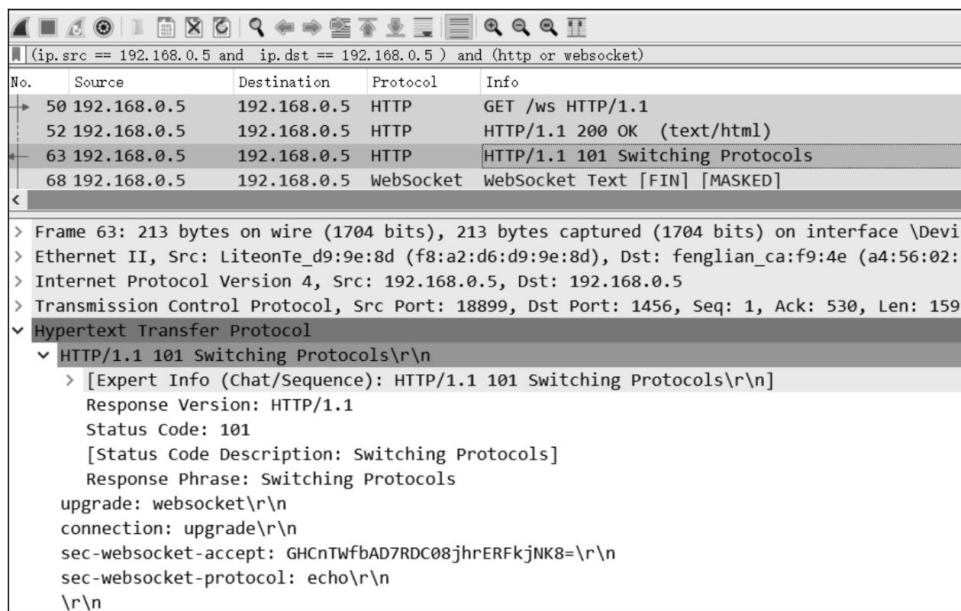


图11-9 服务端发送的Switching Protocols HTTP响应报文

首先，该响应报文的响应状态码为101，表示服务端同意客户端协议升级请求，并将协议从HTTP转换为WebSocket协议。

响应报文中所涉及的比较重要的响应头包括如下几项：

(1) Upgrade响应头

响应报文中Upgrade头值为“WebSocket”，服务端通过该头告诉客户端即将升级的通信协议是WebSocket协议，而不是其他协议。

(2) Sec-WebSocket-Accept响应头

服务端通过Sec-WebSocket-Accept响应头去确认客户端的Sec-WebSocket-Key，该响应头的值为加密过后、客户端握手请求中的Sec-WebSocket-Key值。

客户端收到报文后，会对该值进行校验，只有当握手请求的Sec-WebSocket-Key值经过固定算法加密后的结果和响应头里Sec-WebSocket-Accept的值保持一致，该连接才会被认可建立。

(3) Sec-WebSocket-Protocol请求头

Sec-WebSocket-Protocol表示最终使用的子协议，属于应用程序的自定义协议。

总体来说，WebSocket服务端的响应报文与普通Web服务的HTTP响应报文有以下几点不同：

(1) 该报文的响应码为101。

(2) 响应头Upgrade和Connection头与值都是WebSocket协议规定好的。

(3) 响应头Sec-WebSocket-Accept与Sec-WebSocket-Key请求头是成对使用的，用于进行安全性校验。

客户端收到服务端响应之后，如果校验通过，则握手成功，WebSocket连接建立，双向通信便可以开始了。

说明

以上握手报文的头部字段在IETF所发布的WebSocket标准规范RFC6455中有更加详细的定义，具体可以参考标准规范。

11.3.3 WebSocket通信报文格式

在WebSocket握手过程中，客户端首先发起一个HTTP请求，服务端会有一个HTTP响应。握手过程是HTTP，但是握手完成之后客户端与服务端之间的通信采用的是WebSocket协议。

具体来说，WebSocket协议是基于TCP传输层之上的协议，属于应用层协议，所以也是可以通过抓包工具WireShark抓取到。一个WebSocket通信报文截图如图11-10所示。

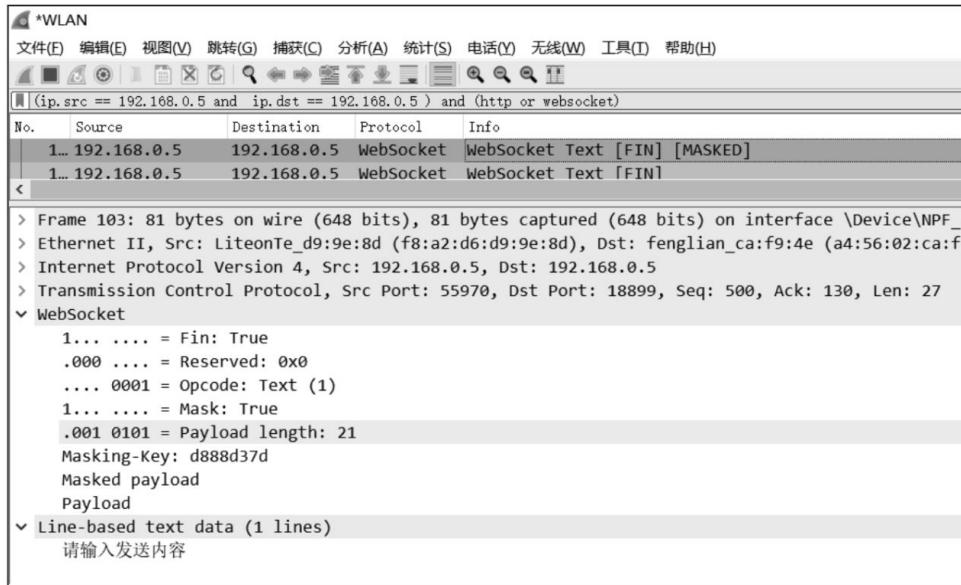


图11-10 WebSocket通信报文截图

WebSocket协议的通信报文是二进制格式，大致包含以下字段：

(1) FIN：占用一位。如果其值是1（抓包工具显示为true），表示该帧是消息的最后一个数据帧；如果其值是0（抓包工具显示为false），表示该帧不是消息的最后一个数据帧。

(2) Opcode：WebSocket帧的操作码，占用4位。操作码的值决定了应该如何解析后续的数据载荷（Data Payload）。如果操作码是不认识的，那么接收端应该断开链接。WebSocket协议的操作码取值说明具体如表11-3所示。

表11-3 WebSocket协议操作码的取值说明

操作码值（Opcode）	码值含义
0x0	表示一个延续帧。当 Opcode 为 0 时，表示本次数据传输采用了多个数据分片，当前收到的数据帧为其中一个数据分片
0x01	表示一个文本帧
0x02	表示一个二进制帧
0x03~07	保留的操作代码，用于后续定义的非控制帧
0x08	表示连接断开的控制帧
0x09	表示一个 Ping 操作，是心跳控制帧之一
0x0A	表示一个 Pong 操作，是心跳控制帧之一，是 Ping 的响应帧
0xB~F	保留的操作代码，用于后续定义的控制帧

WebSocket控制帧有3种：Close、Ping以及Pong。控制帧的Opcode操作码定义为0x08（关闭帧）、0x09（Ping帧）、0x0A（Pong帧）。Close帧很容易理解：客户端如果接收到关闭帧，就关闭连接；当然，客户端也可以发送关闭帧给服务端，服务端收到该帧之后也会关闭连接。

Ping和Pong是WebSocket的心跳帧，用来保证客户端维持正常在线状态。WebSocket为了保持客户端、服务端的实时双向通信，需要确保客户端、服务端之间的TCP通道保持连接没有断开。然而，如果长时间没有数据往来的连接，依旧保持着双向连接，就可能会浪费服务端的连接资源，所以需要关闭这些长时间空闲的连接。

有一些场景需要保持那些长时间空闲的连接，这时可以采用Ping和Pong两个心跳帧来完成。一般来说，服务端给客户端发送Ping，然后客户端发送Pong来回应，表明自己仍然在线。

(3) Mask：一位（值为1时抓包会显示为true），表示是否要对数据载荷进行掩码操作。客户端向服务端发送数据时，需要对数据执行掩码操作；从服务端向客户端发送数据时，不需要对数据进行掩码

操作，如果服务端接收到的数据没有进行掩码操作，那么服务器需要断开连接。所有的客户端发送到服务端的数据帧，Mask值都是1。

(4) Masking-Key：掩码键，如果Mask值为1，就需要用这个掩码键来对数据进行反掩码，以获取到真实的通信数据。为了避免被网络代理服务器误认为是HTTP请求，从而招致代理服务器被恶意脚本攻击，WebSocket客户端必须对所有送给服务器的数据帧执行掩码操作。

客户端必须为发送的每一个数据帧选择新的不同掩码值，并要求这个掩码值是无序、无法预测的。在掩码算法的选择上，为了保证随机性，可以借助密码学中的随机数生成器生成每一个新掩码值。

(5) Payload Length：通信报文中数据载荷的长度。

(6) Payload：通信报文数据帧的有效数据载荷，也就是真正的通信消息内容。

说明

以上WebSocket通信报文字段在IETF所发布的WebSocket标准规范RFC6455中有更加详细的定义和说明，具体可以参考标准规范。

在掌握了WebSocket Echo服务器的开发之后，可以开启下一个进阶实验：参考疯狂创客圈的Netty + WebSocket开源项目，完成一个具备在线聊天、在线推送功能的综合性的WebSocket实战练习。有关该开源项目的实战交流，具体可以参见疯狂创客圈社群博客。

第12章 SSL/TLS核心原理与实战

在HTTP协议中，信息是明文传输的，因此为了通信安全就有了HTTPS（Hyper Text Transfer Protocol over Secure Socket Layer）协议。HTTPS也是一种超文本传送协议，在HTTP的基础上加入了SSL/TLS协议，SSL/TLS依靠证书来验证服务端的身份，并为浏览器和服务端之间的通信加密。

HTTPS是一种通过计算机网络进行安全通信的传输协议，使用HTTP进行通信，借助SSL/TLS建立安全通道和加密数据包。使用HTTPS的主要目的是提供对网站服务端的身份认证，同时保护交换数据的隐私与完整性。

TLS是传输层加密协议，前身是SSL协议，由网景（Netscape）公司1995年发布，有时候TLS和SSL两者不做太多区分。

12.1 什么是SSL/TLS

SSL (Secure Sockets Layer, 安全套接层) 是1994年由网景公司为Netscape Navigator浏览器设计和研发的安全传输技术。Netscape Navigator浏览器是著名的浏览器Firefox (Firefox是继Chrome和Safari之后最受欢迎的浏览器) 的前身。

12.1.1 SSL/TLS协议的版本演进

TCP是传输层的协议，但是它是明文传输的，是不安全的。SSL的诞生给TCP加了一层保险，为TCP通信提供安全及数据完整性保护。TLS只是SSL的升级版，它们的作用是一样的。TLS (Transport Layer Security, 传输层安全协议) 由两层组成：TLS记录 (TLS Record) 和TLS握手 (TLS Handshake)。TLS协议是更新、更安全的SSL协议版本。

SSL/TLS可以理解为安全传输层协议不同发展阶段的版本。1999年，SSL应用广泛，已经成为互联网上的事实标准。IETF (Internet Engineering Task Force, 国际互联网工程任务组) 在1999年把SSL标准化。完成标准化之后，SSL协议名称被改为TLS。

SSL/TLS位于应用层和传输层之间，除了HTTP外，它可以为任何基于TCP传输层以上的应用层协议（如WebSocket协议）提供安全性保证。

理论上，SSL/TLS协议属于传输层。从理论模型的维度来说，该协议在TCP/IP协议栈的分层结构中所处的层次位置大致如图12-1所示。但是，在具体的编码实现上，SSL/TLS协议属于应用层。从实现的维度来说，该协议在TCP/IP协议栈分层结构中所处的层次位置大致如图12-2所示。



图12-1 理论上SSL/TLS协议所处的层次位置



图12-2 实现维度上SSL/TLS协议所处的层次位置

综合起来可以表述为：SSL/TLS协议理论上属于传输层，却实现于应用层。

在客户端浏览器中，目前应用最广泛的是SSL 3.0、TLS 1.0（有时被标为SSL 3.1）、TLS 1.1（有时被标为SSL 3.2）、TLS 1.2（有时被标为SSL 3.3）四个版本的协议。比如，在IE浏览器上，用户可以设置是否使用SSL/TLS协议，还可以设置支持哪一些版本的协议，具体如图12-3所示。

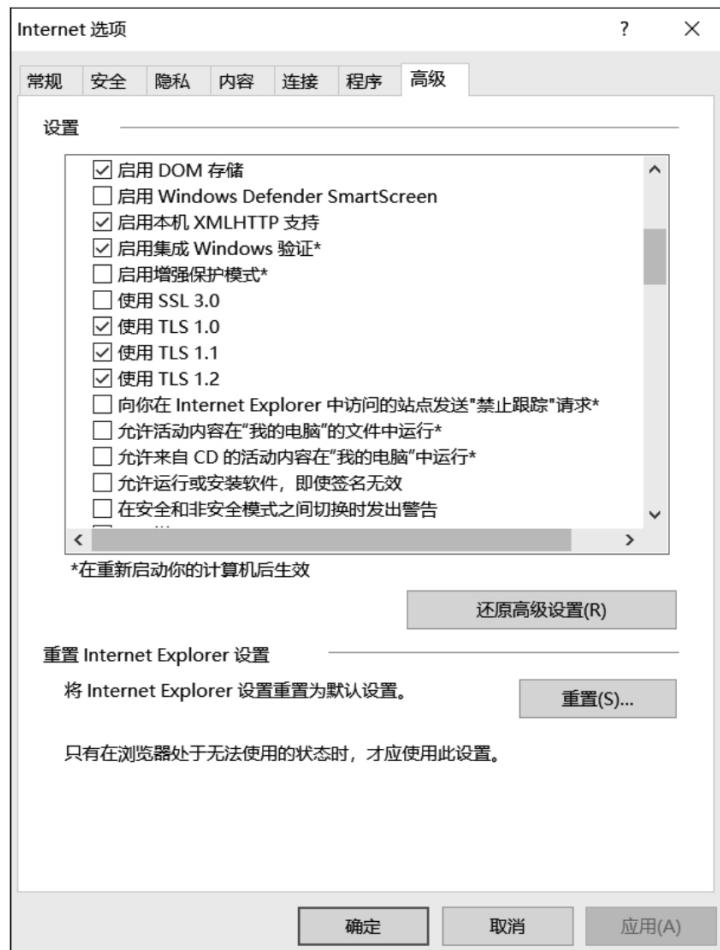


图12-3 IE浏览器的SSL/TLS协议设置

SSL/TLS协议的版本演进过程如表12-1所示。

表12-1 SSL/TLS协议的版本演进

版 本	发布时间	说 明
SSL 1.0		1.0 版本存在严重的安全漏洞，未发布
SSL 2.0	1995 年	网景公司发布了 SSL 2.0 版本
SSL 3.0	1996 年	网景公司完全重新设计了 SSL 3.0 版本。作为历史文献，IETF 通过互联网标准 RFC6101 文件发表了 SSL 3.0 版本
TLS 1.0	1999 年	IETF 通过互联网标准 RFC2246 文件发表了 TLS 1.0，并将其命名为 TLS，TLS 1.0 与 SSL 3.0 的差异非常微小
TLS 1.1	2006 年	IETF 通过互联网标准 RFC4346 文件发表了 TLS 1.1 版本
TLS 1.2	2008 年	IETF 通过互联网标准 RFC5246 文件发表了 TLS 1.2 版本
TLS 1.3	2018 年	IETF 通过互联网标准 RFC8446 文件发表了 TLS 1.3 版本

需要说明的是，每一次版本的演进升级，SSL/TLS协议的安全性都增强了。

12.1.2 SSL/TLS协议的分层结构

SSL/TLS协议包括握手协议（Handshake Protocol）、密码变化协议（SSL Change Cipher Spec Protocol）、警告协议（Alert Protocol）、记录协议（Record Protocol）。

(1) 握手协议：SSL/TLS协议非常重要的组成部分，用来协商通信过程中使用的加密套件（加密算法、密钥交换算法和MAC算法等）、在服务端和客户端之间安全地交换密钥、实现服务端和客户端的身份验证。

(2) 密码变化协议：客户端和服务端通过密码变化协议通知对端，随后的报文都将使用新协商的加密套件和密钥进行保护和传输。

(3) 警告协议：用来向对端发送告警信息，消息中包含告警的严重级别和描述。

(4) 应用数据协议：负责将SSL/TLS承载的应用数据传达给通信对端。

(5) 记录协议：主要负责对上层的数据（SSL/TLS握手协议、SSL/TLS密码变化协议、SSL/TLS警告协议和应用数据协议）进行分块计算、添加MAC值、加密等处理，并把处理后的记录块传输给对端。

SSL/TLS协议的分层结构如图12-4所示。

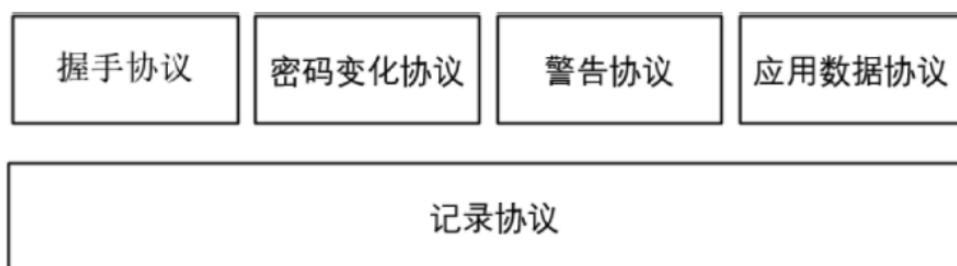


图12-4 SSL/TLS协议的分层结构

SSL/TLS协议主要分为两层（上层的是握手协议、密码变化协议、警告协议和应用数据协议，下层的是记录协议），主要负责使用对称密码对消息进行加密。其中，握手协议（Handshake Protocol）是SSL/TSL通信中最复杂的子协议，也是安全通信所涉及的第一个子协议。

12.2 加密算法原理与实战

为了理解SSL/TLS原理，大家需要掌握一些加密算法的基础知识。当然，这不是为了让大家成为密码学专家，所以只需对基础的加密算法有一些了解即可。基础的加密算法主要有哈希（Hash，或称为散列）、对称加密（Symmetric Cryptography）、非对称加密（Asymmetric Cryptography）、数字签名（Digital Signature）。

12.2.1 哈希单向加密算法原理与实战

哈希算法（或称为散列算法）比较简单，就是为待加密的任意大小的信息（如字符串）生成一个固定大小（比如通过MD5加密之后是32个字符）的字符串摘要。常用的哈希算法有MD5、SHA1、SHA-512等。哈希是不可逆的加密技术，一些数据一旦通过哈希转换为其他形式，源数据将永远无法恢复。

在哪些场景下使用哈希加密呢？一般来说，在用户注册的时候，服务端保存用户密码的时候会将明文密码的哈希密码存储在数据库中，而不是直接存储用户的明文密码。当用户下次登录时，会对用户的登入密码（明文）使用相同的哈希算法进行处理，并将哈希结果与来自数据库的哈希密码进行匹配，如果是相同的，那么用户将登录成功，否则用户将登录失败。

哈希加密也称为单向哈希加密，是通过对不同输入长度的信息进行哈希计算得到固定长度的输出，是单向、不可逆的。所以，即使保存用户密码的数据库被攻击，也不会造成用户的密码泄漏。

最常见的哈希算法为MD5（Message-Digest Algorithm 5，信息摘要算法5），也是计算机广泛使用的哈希算法之一。主流编程语言普遍都提供MD5实现，MD5的前身有MD2、MD3和MD4。

曾经，MD5一度被广泛应用于安全领域。随着MD5的弱点不断被发现，以及计算机能力的不断提升，该算法不再适合当前的安全环境。目前，MD5计算广泛应用于错误检查。例如，在一些文件下载中，软件通过计算MD5和检验下载所得文件的完整性。

MD5将输入的不定长度信息经过程序流程生成四个32位（Bit）数据，最后联合起来输出一个固定长度128位的摘要，基本处理流程包括求余、取余、调整长度、与链接变量进行循环运算等，最终得出结果。

除了MD5，Java还提供了SHA1、SHA256、SHA512等哈希摘要函数的实现。除了在算法上有些差异之外，这些哈希函数的主要不同在于摘要长度，MD5生成的摘要是128位，SHA1生成的摘要是160位，SHA256生成的摘要是256位，SHA512生成的摘要是512位。

SHA-1与MD5的最大区别在于其摘要比MD5摘要长32位（相当于长4字节，转换十六进制后比MD5多8个字符）。对SHA-1强行攻击的强度比对MD5攻击的强度要大。但是SHA-1哈希过程的循环步骤比MD5多，且需要的缓存大，因此SHA-1的运行速度比MD5慢。

以下代码使用Java提供的MD5、SHA1、SHA256、SHA512等哈希摘要函数生成哈希摘要（哈希加密结果）并进行验证的案例：

```
package com.crazymakercircle.secure.crypto;  
//省略import
```

```
public class HashCrypto
{
    /**
     * 哈希单向加密测试用例
     */
    public static String encrypt(String plain)
    {
        StringBuffer md5Str = new StringBuffer(32);
        try
        {
            /**
             * MD5
             */
            //MessageDigest md =
            MessageDigest.getInstance("MD5");
            /**
             * SHA-1
             */
            //MessageDigest md =
            MessageDigest.getInstance("SHA-1");
            /**
             * SHA-256
             */
            //MessageDigest md =
            MessageDigest.getInstance("SHA-256");
            /**
             * SHA-512
             */
        }
    }
}
```

```
*/  
  
MessageDigest md = MessageDigest.getInstance("SHA-  
512");  
  
String charset = "UTF-8";  
byte[] array = md.digest(plain.getBytes(charset));  
for (int i = 0; i < array.length; i++)  
{  
    //转成十六进制字符串  
    String hexString = Integer.toHexString(  
        (0x000000FF & array[i]) |  
        0xFFFFFFF0);  
    log.debug("hexString: {}, 第6位之后: {}",  
        hexString,  
        hexString.substring(6));  
    md5Str.append(hexString.substring(6));  
}  
}  
catch (Exception ex)  
{  
    ex.printStackTrace();  
}  
return md5Str.toString();  
}  
  
public static void main(String[] args)  
{  
    //原始的明文字符串，也是需要加密的对象
```

```
String plain = "123456";

//使用哈希函数加密

String cryptoMessage = HashCrypto.encrypt(plain);
log.info("cryptoMessage:{}", cryptoMessage);

//验证

String cryptoMessage2 = HashCrypto.encrypt(plain);
log.info("验证 {},\n是否一致: {}", cryptoMessage2,
cryptoMessage.equals(cryptoMessage2));

//验证2

String plainOther = "654321";
String cryptoMessage3 = HashCrypto.encrypt(plainOther);
log.info("验证 {},\n是否一致: {}", cryptoMessage3,
cryptoMessage.equals(cryptoMessage3));

}
```

运行以上程序，部分结果大致如下：

```
10:38:12.740 [main] INFO HashCrypto - cryptoMessage:ba3253876...
10:38:12.743 [main] INFO HashCrypto - 验证 ba3253876...,
是否一致: true
```

```
10:38:12.747 [main] INFO HashCrypto - 验证 690437692d9...,  
是否一致: false
```

12.2.2 对称加密算法原理与实战

对称加密（Symmetric Cryptography）指的是客户端自己封装一种加密算法，将给服务端发送的数据进行加密，并且将数据加密的方式（密钥）发送给密文，服务端收到密钥和数据，用密钥进行解密。

对称加密的典型处理流程大致如图12-5所示。

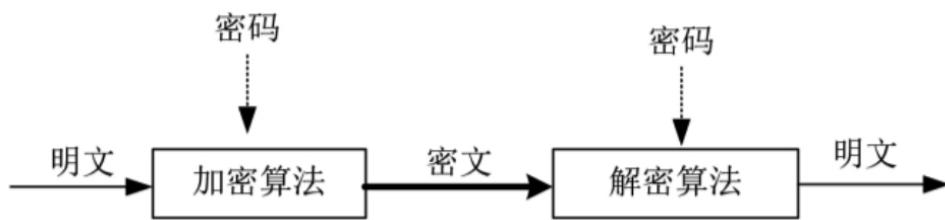


图12-5 对称加密的典型处理流程

对称加密：使用同一个密钥加密和解密，优点是速度快；但是它要求共享密钥，缺点是密钥管理不方便、容易泄露。

常见的对称加密算法有DES、AES等。DES加密算法出自IBM的数学研究，被美国政府正式采用之后开始广泛流传，但是近些年来使用越来越少，因为DES使用56位密钥，以现代计算能力24小时内即可被破解。虽然如此，但是在对安全要求不高的应用中，还是可以使用DES加密算法。

下面是一段使用Java语言编写的进行DES加密的演示代码：

```
package com.crazymakercircle.secure.crypto;  
//省略import  
public class DESCrypto  
{  
    /**  
     * 对称加密  
     */  
    public static byte[] encrypt(byte[] data, String password)  
    {  
        try{  
            SecureRandom random = new SecureRandom();  
            //使用密码，创建一个密钥描述符  
            DESKeySpec desKey = new  
            DESKeySpec(password.getBytes());  
            //创建一个密钥工厂，然后用它把 DESKeySpec 密钥描述符实例转  
            //换成密钥  
            SecretKeyFactory keyFactory =  
  
            SecretKeyFactory.getInstance("DES");  
            //通过密钥工程生成密钥  
            SecretKey secretKey =  
            keyFactory.generateSecret(desKey);  
            //Cipher对象实际完成加密操作  
            Cipher cipher = Cipher.getInstance("DES");  
            //用密钥初始化Cipher对象  
            cipher.init(Cipher.ENCRYPT_MODE, secretKey,  
            random);  
        }  
    }  
}
```

```
//为数据执行加密操作
    return cipher.doFinal(data);
} catch (Throwable e) {
    e.printStackTrace();
}
return null;
}

/**
 * 对称解密
 */
public static byte[] decrypt(byte[] cryptData,
String password) ...{

//DES算法要求有一个可信任的随机数源
SecureRandom random = new SecureRandom();
//创建一个 DESKeySpec 密钥描述符对象
DESKeySpec desKey = new
DESKeySpec(password.getBytes());
//创建一个密钥工厂
SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance("DES");
//将 DESKeySpec 对象转换成 SecretKey 对象
SecretKey secretKey =
keyFactory.generateSecret(desKey);
//Cipher对象实际完成解密操作
Cipher cipher = Cipher.getInstance("DES");
```

```
//用密钥初始化Cipher对象  
cipher.init(Cipher.DECRYPT_MODE, secretKey, random);  
//真正开始解密操作  
return cipher.doFinal(cryptData);  
}  
  
public static void main(String args[]) {  
    //待加密内容  
    String str = "123456";  
    //密码长度要是8的倍数  
    String password = "12345678";  
  
    byte[] result =  
DESCrypto.encrypt(str.getBytes(),password);  
    log.info("str:{} 加密后: {}",str,new String(result));  
    //直接将如上内容解密  
    try {  
        byte[] decryResult = DESCrypto.decrypt(result,  
password);  
        log.info("解密后: {}",new String(decryResult));  
    } catch (Exception e1) {  
        e1.printStackTrace();  
    }  
}
```

以上程序的运行结果非常简单，在这里不再赘述。需要注意的是，在DES加密和解密过程中，密钥长度都必须是8的倍数。

12.2.3 非对称加密算法原理与实战

非对称加密算法（Asymmetric Cryptography）又称为公开密钥加密算法，需要两个密钥：一个称为公开密钥（公钥）；另一个称为私有密钥（私钥）。公钥与私钥需要配对使用，如果用公钥对数据进行加密，只有用对应的私钥才能解密；如果使用私钥对数据加密，那么需要用对应的公钥才能解密。由于加解密使用不同的密钥，因此这种算法为非对称加密算法。

非对称加密的典型处理流程如图12-6所示。

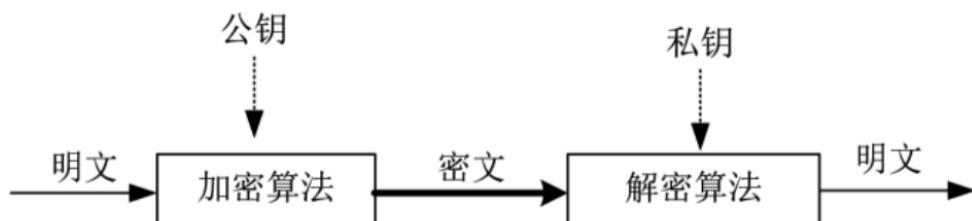


图12-6 非对称加密的典型处理流程

非对称加密算法的优点是密钥管理很方便，缺点是速度慢。典型的非对称加密算法有RSA、DSA等。

下面是一段使用Java代码进行RSA加密的演示代码：

```
package com.crazymakercircle.secure.crypto;  
//省略import  
/**
```

```
* RSA 非对称加密算法
*/
@Slf4j
public class RSAEncrypt
{
    /**
     * 指定加密算法为RSA
     */
    private static final String ALGORITHM = "RSA";
    /**
     * 常量，用来初始化密钥长度
     */
    private static final int KEY_SIZE = 1024;
    /**
     * 指定公钥存放文件
     */
    private static final String PUBLIC_KEY_FILE =
        SystemConfig.getKeystoreDir() + "/PublicKey";
    /**
     * 指定私钥存放文件
     */
    private static final String PRIVATE_KEY_FILE =
        SystemConfig.getKeystoreDir() + "/PrivateKey";

    /**
     * 生成密钥对
     */
}
```

```
protected static void generateKeyPair() throws Exception
{
    /**
     * 为RSA算法创建一个KeyPairGenerator对象
     */
    KeyPairGenerator keyPairGenerator =
        KeyPairGenerator.getInstance(ALGORITHM);

    /**
     * 利用上面的密钥长度初始化这个KeyPairGenerator对象
     */
    keyPairGenerator.initialize(KEY_SIZE);

    /**
     * 生成密钥对 */
    KeyPair keyPair = keyPairGenerator.generateKeyPair();

    /**
     * 得到公钥 */
    PublicKey publicKey = keyPair.getPublic();

    /**
     * 得到私钥 */
    PrivateKey privateKey = keyPair.getPrivate();

    ObjectOutputStream oos1 = null;
    ObjectOutputStream oos2 = null;
    try
```

```
{  
    log.info("生成公钥和私钥，并且写入对应的文件");  
  
    File file = new File(PUBLIC_KEY_FILE);  
    if (file.exists())  
    {  
        log.info("公钥和私钥已经生成，不需要重复生成，  
path:{}",  
PUBLIC_KEY_FILE);  
        return;  
    }  
    /** 用对象流将生成的密钥写入文件 */  
    log.info("PUBLIC_KEY_FILE 写入: {}",  
PUBLIC_KEY_FILE);  
    oos1 = new ObjectOutputStream(  
        new  
FileOutputStream(PUBLIC_KEY_FILE));  
    log.info("PRIVATE_KEY_FILE 写入: {}",  
PRIVATE_KEY_FILE);  
    oos2 = new ObjectOutputStream(  
        new  
FileOutputStream(PRIVATE_KEY_FILE));  
    oos1.writeObject(publicKey);  
    oos2.writeObject(privateKey);  
} catch (Exception e)  
{  
    throw e;  
}
```

```
        } finally
    {
        /** 清空缓存，关闭文件输出流 */
        IOUtil.closeQuietly(oos1);
        IOUtil.closeQuietly(oos2);
    }
}

/**
 * 加密方法，使用公钥加密
 * @param plain 明文数据
 */
public static String encrypt(String plain) throws Exception
{
    //从文件加载公钥
    Key publicKey = loadPublicKey();

    /** 得到Cipher对象，来实现对源数据的RSA加密 */
    Cipher cipher = Cipher.getInstance(ALGORITHM);
    cipher.init(Cipher.ENCRYPT_MODE, publicKey);
    byte[] b = plain.getBytes();
    /** 执行加密操作 */
    byte[] b1 = cipher.doFinal(b);
    BASE64Encoder encoder = new BASE64Encoder();
    return encoder.encode(b1);
}

/**

```

```
* 从文件加载公钥
*/
public static PublicKey loadPublicKey() throws Exception
{
    PublicKey publicKey=null;
    ObjectInputStream ois = null;
    try
    {
        log.info("PUBLIC_KEY_FILE 读取: {}",
PUBLIC_KEY_FILE);
        /** 读出文件中的公钥 */
        ois = new ObjectInputStream(
            new
FileInputStream(PUBLIC_KEY_FILE));
        publicKey = (PublicKey) ois.readObject();
    } catch (Exception e)
    {
        throw e;
    } finally
    {
        IOUtil.closeQuietly(ois);
    }
    return publicKey;
}
```

```
//方法：对密文解密，使用私钥解密
public static String decrypt(String crypto) throws
```

```
Exception

{

    PrivateKey privateKey = loadPrivateKey();

    /**
     * 得到Cipher对象，对已用公钥加密的数据进行RSA解密 */
    Cipher cipher = Cipher.getInstance(ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, privateKey);
    BASE64Decoder decoder = new BASE64Decoder();
    byte[] b1 = decoder.decodeBuffer(crypto);

    /**
     * 执行解密操作 */
    byte[] b = cipher.doFinal(b1);
    return new String(b);
}

/**
 * 从文件加载私钥
 * @throws Exception
 */
public static PrivateKey loadPrivateKey() throws Exception
{
    PrivateKey privateKey;
    ObjectInputStream ois = null;
    try
    {
        log.info("PRIVATE_KEY_FILE 读取: {}",
PRIVATE_KEY_FILE);
    }
}
```

```
    /** 读出文件中的私钥 */
    ois = new ObjectInputStream(
        new
FileInputStream(PRIVATE_KEY_FILE));
    privateKey = (PrivateKey) ois.readObject();
} catch (Exception e)
{
    e.printStackTrace();
    throw e;
} finally
{
    IOUtil.closeQuietly(ois);
}
return privateKey;
}

public static void main(String[] args) throws Exception
{
    //生成密钥对
    generateKeyPair();
    //待加密内容
    String plain = "疯狂创客圈 Java 高并发研习社群";

    //公钥加密
    String dest = encrypt(plain);
    log.info("{} 使用公钥加密后: \n{}", plain, dest);
}
```

```
//私钥解密  
String decrypted = decrypt(dest);  
log.info(" 使用私钥解密后: \n{} ", decrypted);  
}  
}
```

执行以上RSA演示程序，运行的结果大致如下：

```
[main] INFO RSAEncrypt - 生成公钥和私钥，并且写入对应的文件  
[main] INFO RSAEncrypt - PUBLIC_KEY_FILE 写入: F:/.../PublicKey  
[main] INFO RSAEncrypt - PRIVATE_KEY_FILE 写入: F:/.../PrivateKey  
[main] INFO RSAEncrypt - PUBLIC_KEY_FILE 读取: F:/.../PublicKey  
[main] INFO RSAEncrypt - 疯狂创客圈 Java 高并发研习社群 使用公钥加密  
后:  
V1INyGwg97EvF1/xUT4x0rsrrkslzcm8ckvrxA1d8wTCR9rpE1A69eRJTo+VCnO  
14emJkK/urQb3WcwFiNLk+PS5XnoVufV4IebH0FF5UjkOOokHEjTgvbqhTdNnY0p  
mLfhSmcoBSzif9Jgxez7hBIF7cJd7rsipbhSd1Dzr6iJI=  
[main] INFO RSAEncrypt - PRIVATE_KEY_FILE 读取: F:/.../PrivateKey  
[main] INFO RSAEncrypt - 使用私钥解密后:  
疯狂创客圈Java高并发研习社群
```

非对称加密算法包含两种密钥，其中的公钥本来是公开的，不需要像对称加密算法那样将私钥给对方，对方解密时使用公开的公钥即可，大大地提高了加密算法的安全性。退一步讲，即使不法之徒获知了非对称加密算法的公钥，甚至获知了加密算法的源码，只要没有获取公钥对应的私钥，也是无法进行解密的。

12.2.4 数字签名原理与实战

数字签名（Digital Signature）是确定消息发送方身份的一种方案。在非对称加密算法中，发送方A通过接收方B的公钥将数据加密后的密文发送给接收方B，B利用私钥解密就得到了需要的数据。这里还存在一个问题，接收方B的公钥是公开的，接收方B收到的密文都是使用自己的公钥加密的，那么如何检验发送方A的身份呢？

一种非常简单的检验发送方A身份的方法为：发送方A可以利用A自己的私钥进行消息加密，然后B利用A的公钥来解密，由于私钥只有A知道，接收方只要解密成功，就可以确定消息来自A而不是其他地方。

数字签名的原理就基于此，通常为了证明发送数据的真实性，利用发送方的私钥对待发送的数据生成数字签名。

数字签名的流程比较简单，首先通过哈希函数为待发数据生成较短的消息摘要，然后利用私钥加密该摘要，所得到的摘要密文基本上就是数字签名。发送方A将待发送数据以及数字签名一起发送给接收方B，接收方B收到之后使用A的公钥校验数字签名，如果校验成功，就说明内容来自发送方A，否则为非法内容。

数字签名的大致流程如图12-7所示。

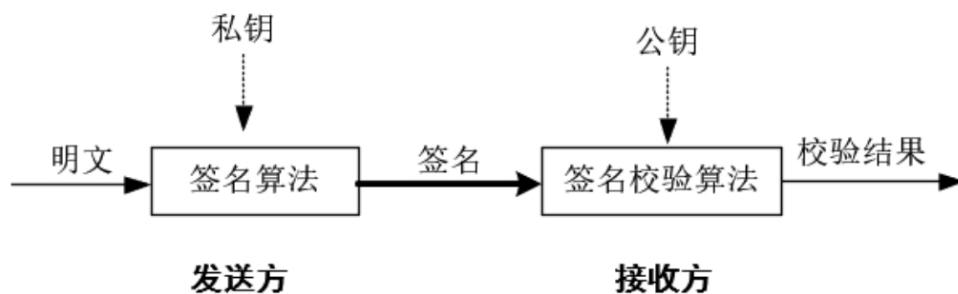


图12-7 数字签名的大致处理流程

Java为数字签名提供了良好的支持，`java.security.Signature`接口提供了数字签名的基本操作API，Java规范要求各JDK版本需要提供表12-2中所列出的标准签名实现。

表12-2 Java规范要求各JDK版本提供的数字签名实现

Java 规范要求提供的数字签名实现	说 明
SHA1withDSA	使用 SHA1 算法生成摘要，使用 DSA 算法进行摘要加密
SHA1withRSA	使用 SHA1 算法生成摘要，使用 RSA 算法进行摘要加密
SHA256withRSA	使用 SHA256 算法生成摘要，使用 RSA 算法进行摘要加密

下面是一段使用JSHA512withRSA算法实现数字签名的Java演示代码：

```
package com.crazymakercircle.secure.crypto;  
//省略import  
/**  
 * RSA签名演示  
 */  
@Slf4j  
public class RSASignDemo  
{  
    /**  
     * RSA签名  
     *  
     * @param data 待签名的字符串  
     * @param priKey RSA私钥字符串
```

```
* @return 签名结果
* @throws Exception 签名失败则抛出异常
*/
public byte[] rsaSign(byte[] data, PrivateKey priKey)
        throws SignatureException
{
    try
    {
        Signature signature =
Signature.getInstance("SHA512withRSA");
        signature.initSign(priKey);
        signature.update(data);

        byte[] signed = signature.sign();
        return signed;
    } catch (Exception e)
    {
        throw new SignatureException("RSAcontent = " + data
                + "; charset = ", e);
    }
}
/***
 * RSA验签
 * @param data 被签名的内容
 * @param sign 签名后的结果
 * @param pubKey RSA公钥
 * @return 验签结果

```

```
 */
public boolean verify(byte[] data, byte[] sign, PublicKey
pubKey)
throws SignatureException
{
try
{
Signature signature =
Signature.getInstance("SHA512withRSA");
signature.initVerify(pubKey);
signature.update(data);
return signature.verify(sign);

} catch (Exception e)
{
e.printStackTrace();
throw new SignatureException("RSA验证签名[content =
" + data +
"; charset = " + "; signature =
" + sign + "]发生异常!", e);
}
}

/**
 * 私钥
 */
private PrivateKey privateKey;
```

```
/*
 * 公钥
 */
private PublicKey publicKey;

/*
 * 加密过程
 * @param publicKey      公钥
 * @param plainTextData 明文数据
 * @throws Exception 加密过程中的异常信息
 */
public byte[] encrypt(PublicKey publicKey, byte[]
plainTextData)
        throws Exception
{
    if (publicKey == null)
    {
        throw new Exception("加密公钥为空， 请设置");
    }
    Cipher cipher = null;
    try
    {
        cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] output = cipher.doFinal(plainTextData);
        return output;
    }
```

```
        } catch (NoSuchAlgorithmException e)
        {
            throw new Exception("无此加密算法");
        }

        ...
    }

    /**
     * 解密过程
     * @param privateKey 私钥
     * @param cipherData 密文数据
     * @return 明文
     * @throws Exception 解密过程中的异常信息
     */
    public byte[] decrypt(PrivateKey privateKey, byte[]
cipherData)...
{
    if (privateKey == null)
    {
        throw new Exception("解密私钥为空, 请设置");
    }

    Cipher cipher = null;
    try
    {
        cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] output = cipher.doFinal(cipherData);
        return output;
    }
}
```

```
        } catch (NoSuchAlgorithmException e)
        {
            throw new Exception("无此解密算法");
        }

        ...
    }

/**
 * Main 测试方法
 * @param args
 */
public static void main(String[] args) throws Exception
{
    RSASignDemo RSASignDemo = new RSASignDemo();
    //加载公钥
    RSASignDemo.publicKey = RSAEncrypt.loadPublicKey();
    //加载私钥
    RSASignDemo.privateKey = RSAEncrypt.loadPrivateKey();

    //测试字符串
    String sourceText = "疯狂创客圈 Java 高并发社群";
    try
    {
        log.info("加密前的字符串为: {}", sourceText);

        //公钥加密
        byte[] cipher = RSASignDemo.encrypt(
            RSASignDemo.publicKey,
```

```
sourceText.getBytes() ;  
  
        //私钥解密  
byte[] decryptText = RSASignDemo.decrypt(  
                                         RSASignDemo.privateKey,  
cipher);  
  
log.info("私钥解密的结果是: {}", new  
String(decryptText));  
  
//字符串生成签名  
byte[] rsaSign = RSASignDemo.rsaSign(  
                                         sourceText.getBytes(),  
RSASignDemo.privateKey);  
  
//签名验证  
Boolean succeed =  
RSASignDemo.verify(sourceText.getBytes(),  
                                         rsaSign,  
RSASignDemo.publicKey);  
  
log.info("字符串签名为: \n{}", byteToHex(rsaSign));  
log.info("签名验证结果是: {}, succeed");  
  
String fileName =  
  
IOUtil.getResourcePath("/system.properties");  
  
byte[] fileBytes = readFileByBytes(fileName);  
//文件签名验证  
byte[] fileSign =
```

```
        RSASignDemo.rsaSign(fileBytes,
RSASignDemo.privateKey);
        log.info("文件签名为: \n{}" , byteToHex(fileSign));

        //文件签名保存
        String signPath =
                SystemConfig.getKeystoreDir() +
"/fileSign.sign";
        ByteUtil.saveFile(fileSign,signPath );
        Boolean verifyOK = RSASignDemo.verify(
                fileBytes, fileSign,
RSASignDemo.publicKey);
        log.info("文件签名验证结果是: {}" , verifyOK);

        //读取验证文件
        byte[] read = readFileByBytes(signPath);
        log.info("读取文件签名: \n{}" , byteToHex(read));
        verifyOK= RSASignDemo.verify(
                fileBytes, read,
RSASignDemo.publicKey);
        log.info("读取文件签名验证结果是: {}" , verifyOK);
    } catch (Exception e)
{
    System.err.println(e.getMessage());
}
}
```

执行以上数字签名的Java演示程序，运行的结果大致如下：

```
[main] INFO RSAEncrypt - PUBLIC_KEY_FILE 读取: F:\ .../PublicKey  
[main] INFO RSAEncrypt - PRIVATE_KEY_FILE 读取: F:\ .../PrivateKey  
[main] INFO RSASignDemo - 加密前的字符串为: 疯狂创客圈 Java 高并发社群  
[main] INFO RSASignDemo - 私钥解密的结果是: 疯狂创客圈 Java 高并发社群  
[main] INFO RSASignDemo - 字符串签名为:  
2f04c6d64a9184a1319301c0a9700a4e85be3b7b81c4d0d98fb9dc276328072  
8860d68cf9bb9ec076122f930a64d979240ade21bfe01be57562ccf1fc236  
a853aaef7945dbb1db4eed53107167e0cbb47b0fca5ef0a52ff3f08200254429  
ab24c76b73eff494588306e8a461366f4fab486dcb1784c230b61c74b0df5b4  
3534  
[main] INFO RSASignDemo - 签名验证结果是: true  
[main] INFO RSASignDemo - 文件签名为:  
7bdc8faecc8bdd48e5500f7dbfbce1cc3626dc322e5a6f540f003e496d0914  
638b706bcea2079c4243d7ff070dedf6bcf30c19cd16b40d7640382954a8d5c  
17c420d7292873720209c97f333fe0c2aaefb4735a150cdafcl1d02d770459918  
3b47bc5324ddfc1b69266e4b07b9f3c7715d3833af695fb6ec0fc35ddd6d963  
e2f9  
[main] INFO RSASignDemo - 文件签名验证结果是: true  
[main] INFO RSASignDemo - 读取文件签名为:  
7bdc8faecc8bdd48e5500f7dbfbce1cc3626dc322e5a6f540f003e496d0914  
638b706bcea2079c4243d7ff070dedf6bcf30c19cd16b40d7640382954a8d5c  
17c420d7292873720209c97f333fe0c2aaefb4735a150cdafcl1d02d770459918  
3b47bc5324ddfc1b69266e4b07b9f3c7715d3833af695fb6ec0fc35ddd6d963
```

e2f9

[main] INFO RSASignDemo - 读取文件签名验证结果是: true

12.3 SSL/TLS运行过程

SSL/TLS协议实现通信安全的基本思路是：消息发送之前，发送方A先向接收方B申请公钥，发送方A采用公钥加密法对发出去的通信内容进行加密，接收方B收到密文后，用自己的私钥对通信密文进行解密。

SSL/TLS协议运行的基本流程如下：

- (1) 客户端向服务端索要并验证公钥。
- (2) 双方协商生成“对话密钥”。
- (3) 双方采用“对话密钥”进行加密通信。

前两步又称为“握手阶段”，每一个TLS连接都会以握手开始。“握手阶段”涉及四次通信，并且所有通信都是明文的。在握手过程中，客户端和服务端将进行以下四个主要阶段：

- (1) 交换各自支持的加密套件和参数，经过协商后，双方就加密套件和参数达成一致。
- (2) 验证对方（主要指服务端）的证书，或使用其他方式进行服务端身份验证。
- (3) 对将用于保护会话的共享主密钥达成一致。
- (4) 验证握手消息是否被第三方修改。

12.3.1 SSL/TLS第一阶段握手

客户端与服务端通过TCP三次握手建立传输层连接后，通信双方需要交换各自支持的加密套件和参数，经过协商后，使通信双方的加密套件和参数达成一致。

SSL/TLS“握手”第一个阶段的工作为：由客户端发一个Client Hello报文给服务端，并且第一个阶段只有这一个数据帧（报文）。Client Hello数据帧的内容大致包括以下信息：

- (1) 客户端支持的SSL/TLS协议版本，比如TLS 1.2版。
- (2) 一个客户端生成的随机数，这是握手过程中的第一个随机数，称之为Random_C。
- (3) 客户端支持的签名算法、加密方法、摘要算法（比如RSA公钥签名算法）。
- (4) 客户端支持的压缩方法。

使用WireShark抓取的客户端所发送的Client Hello请求报文大致如图12-8所示。

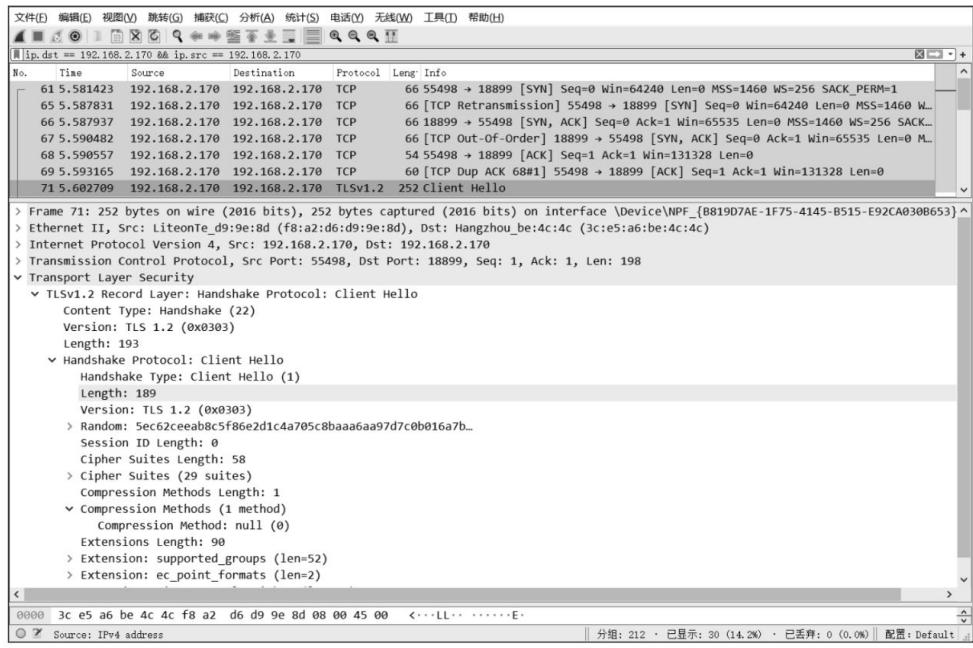


图12-8 使用WireShark抓取的客户端所发送的Client Hello请求报文

从Client Hello请求报文的截图可以看出，Client Hello请求报文是处于TCP层之上的应用层报文。Client Hello请求报文所包含的字段大致如下：

(1) Handshake Type

此字段为握手协议的类型，这里为Client Hello类型，其值为1，表示此报文为客户端发起的TSL/SSL握手请求的第一个报文。

(2) Version

此字段为TSL/SSL的协议版本，指示客户端支持的最佳协议版本，以上截图的示例报文中的版本为TLS 1.2。

(3) Random

一个客户端生成的随机数，为握手过程中的第一个随机数，这里记为“Random_C”，稍后用于生成“对话密钥”。Random字段是在1994年Netscape Navigator浏览器中发现了一个严重故障之后为防御弱随机数生成器而引入的。在握手时，客户端和服务端都会提供随机数。这种随机性对每次握手都是独一无二的，在身份验证中起着举足轻重的作用，它可以防止重放攻击，并确认初始数据交换的完整性。Random随机数字段包含32字节的数据。其中，只有28字节是随机生成的，剩余的4字节包含额外的信息，受客户端时钟的影响。最初，剩余的4字节为部分精确时间，目前由于担心客户端时间可能被用于大规模浏览器指纹采集，因此一些浏览器会给它们的时间添加时钟扭曲，或者简单粗暴地发送随机的4字节。

(4) Session ID

在第一次连接时，Session ID字段是空的，表示客户端并不希望恢复某个已存在的会话，希望开始新的会话。在后续的连接中，这个字段可以存放Session ID（唯一标识），服务端可以借助收到的Session ID在自己的缓存中找到对应的交互会话。

(5) Cipher Suite

此字段用于发送客户端支持的密钥套件（Cipher Suite）列表，是由客户端支持的所有密钥套件组成的列表（按优先级顺序排列）。一个密钥套件一般由“密钥交换算法+签名算法+对称加密算法+摘要算法”组成，示例如下：

Cipher Suite: TLS_ECDHE_RSA_AES_256_GCM_SHA384

以上密钥套件表示握手时使用的密钥交换算法为ECDHE算法，并且签名算法用RSA签名和身份认证算法，握手后使用AES对称加密算法进行通信加密和解密，并且AES的密钥长度为256位，分组模式是GCM，套件使用SHA384摘要算法产生随机数和消息验证。

(6) Compression

Compression字段表示客户端支持的压缩方法。客户端可以提交一个或多个支持压缩的方法，默认的压缩方法是null，代表没有压缩。

(7) Extensions

Extensions（扩展块）由任意数量的Extension（扩展）组成。这些扩展会携带额外数据，比如服务端名称等。

12.3.2 SSL/TLS第二阶段握手

SSL/TLS握手第二个阶段的工作为：服务端对客户端的Client Hello请求进行响应。在收到客户端请求（Client Hello）后，服务端向客户端发出回应，这个阶段的服务端回应帧（报文）一般包含4个回复帧：Server Hello帧、Certificate帧、Server Key Exchange帧、Server Hello Done帧。

1. Server Hello帧

服务端回复的Server Hello帧主要包含以下内容：

- (1) 回复服务端使用的加密通信协议版本，比如TLS 1.2版本。

(2) 一个服务端生成的随机数，是整个握手过程中的第二个随机数，记为“Random_S”，稍后用于生成“对话密钥”。

(3) 确认使用的加密方法，比如RSA公钥加密。

(4) 服务端的证书。

如果浏览器与服务端支持的SSL/TSL通信协议版本不一致，那么服务端将关闭加密通信。使用WireShark抓取到的服务端的Server Hello响应报文实例如图12-9所示。

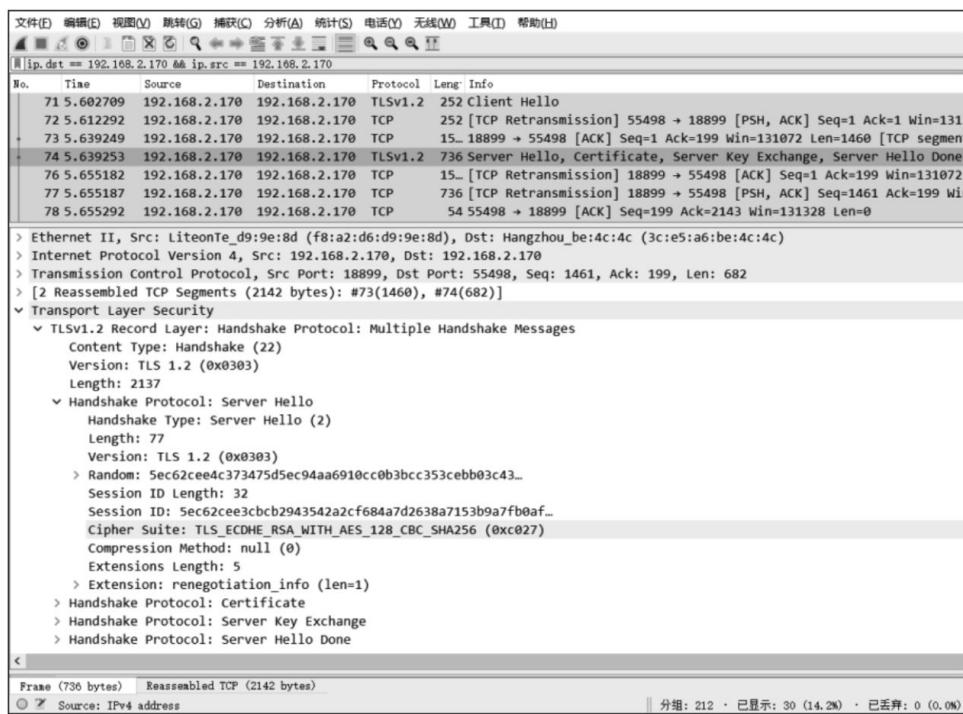


图12-9 使用Wireshark抓取的服务端所发送的Server Hello帧实例

服务端回应的Server Hello报文包含如下字段：

(1) Handshake Type

此字段标识当前握手报文的类型，这里为Server Hello类型，其值为2。

(2) Version

确认服务端使用的SSL/TLS通信协议版本，以上实例报文中Version的值为TLS 1.2。服务端无须支持客户端支持的最佳版本。如果服务端不支持客户端发送的版本，可以提供某个其他版本以期待客户端能够接受。

(3) Random

一个服务端生成的随机数，稍后用于生成“对话密钥”。

(4) Session ID

服务端会创建新的会话，返回新会话的Session ID。如果客户端之前发送的Session ID为已存在会话的ID，那么服务端会查找已经存在的会话，并返回其Session ID。

(5) Cipher Suite

服务端选择的密码套件，以上示例中选择的密码套件为：

Cipher Suite: TLS_ECDHE_RSA_AES_256_GCM_SHA384

以上密钥套件表示握手时使用的密钥交换算法为ECDHE；签名算法为RSA；握手后使用AES对称加密算法进行通信加密和解密，并且AES的密钥长度为256位，分组模式是GCM；套件使用SHA384摘要算法产生随机数和消息验证。

总之，服务端回复的Server Hello消息表示如何将服务端所选择的通信参数传送给客户端。这个消息的结构与Client Hello消息类似，只是每个字段只包含一个选项。服务端无须支持客户端支持的最佳版本，如果服务端不支持与客户端相同的版本，可以提供某个其他版本以期待客户端能够接受。

2. Certificate帧

Certificate帧用于返回服务端证书，该证书中含有服务端的证书清单（包括服务端公钥），用于身份验证和密钥协商。在多数电子商务应用中，客户端都需要进行服务端身份验证，服务端通过Certificate帧发送自己的证书给客户端。

使用WireShark抓取到的服务端的Certificate帧实例如图12-10所示。

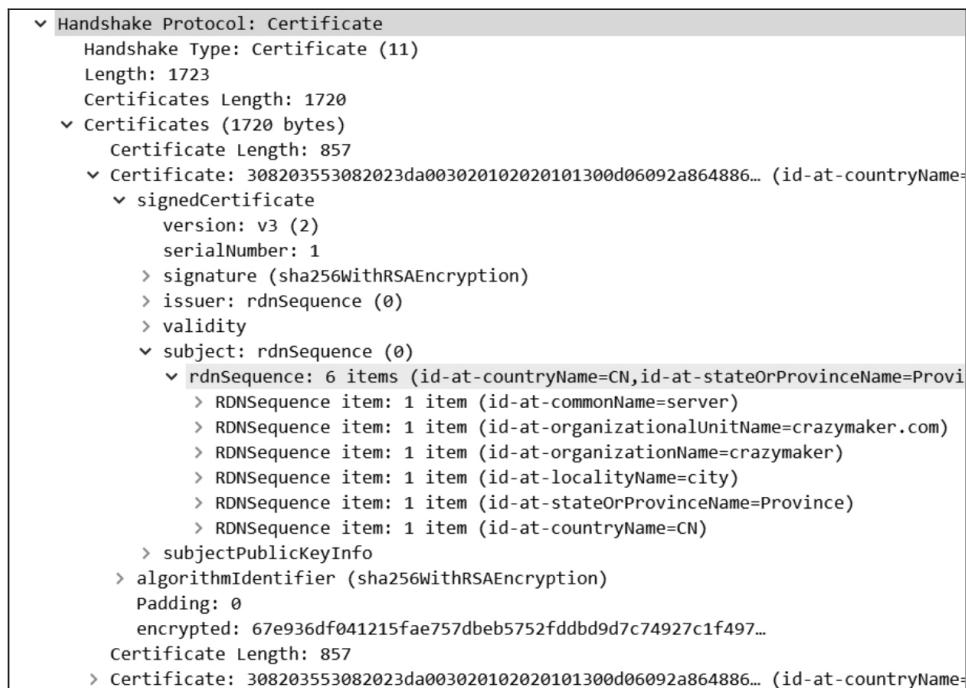


图12-10 使用WireShark抓取的服务端所发送的Certificate帧实例

服务端通过Certificate帧给客户端提供身份信息，那么客户端是否需要提供自己的身份证件给服务端呢？虽然大部分场景中服务端不需要验证客户端的身份，但是只要服务端需要验证客户端的身份，服务端就会发一个Certificate Request证书请求给客户端。比如，在一些安全性要求较高的机构（如金融机构）往往需要验证客户端身份证件，这些机构只允许通过认证客户连入自己的网络，并且会给正式客户提供USB密钥，里面就包含了一张客户端身份证件，在通信握手时要求客户端提供证书。

3. Server Key Exchange帧

Server Key Exchange帧的目的是携带密钥交换的额外数据，其消息内容对于不同的协商算法套件都会存在差异。在某些场景中，服务端不需要发送Server Key Exchange握手消息。如果在Server Hello消息中使用DHE/ECDHE非对称密钥协商算法来进行SSL握手，就将发送该类型握手消息。对于使用RSA算法的SSL握手，不会发送该类型握手消息。另外，使用DH、ECDH算法进行握手时也不会发送该类型握手消息。

使用WireShark抓取到的服务端的Server Key Exchange报文实例如图12-11所示。

```
▼ TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
  Content Type: Handshake (22)
  Version: TLS 1.2 (0x0303)
  Length: 2137
  > Handshake Protocol: Server Hello
  > Handshake Protocol: Certificate
  ▼ Handshake Protocol: Server Key Exchange
    Handshake Type: Server Key Exchange (12)
    Length: 329
    ▼ EC Diffie-Hellman Server Params
      Curve Type: named_curve (0x03)
      Named Curve: secp256r1 (0x0017)
      Pubkey Length: 65
      Pubkey: 0491366bf3fe3427ecc0b348cdf2ec836c5c328c601f6237...
      > Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
      Signature Length: 256
      Signature: 3aa40959778c74bd01caab5ee380faa3921f063d4f946bb8...
    > Handshake Protocol: Server Hello Done
```

图12-11 使用Wireshark抓取的服务端所发送的Server Key Exchange帧实例

Server Key Exchange帧的内容大致如下：

(1) Handshake Type

此字段标识当前握手报文的类型，这里为Server Key Exchange类型，其值为12。

(2) EC Diffie-Hellman Server Params

前面的Server Hello帧选择了

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256密钥套件，该密钥套件指定了密钥协商算法是ECDHE（椭圆曲线协商算法）非对称密钥协商算法。此附件消息用于告知客户端，服务端是通过Diffie-Hellman算法生成最终密钥的（也就是Sessionkey会话密钥）。

(3) Pubkey

Pubkey是Diffie-Hellman算法中的一个参数，这个参数需要通过网络传给客户端，即使它被截取也没有影响安全性。

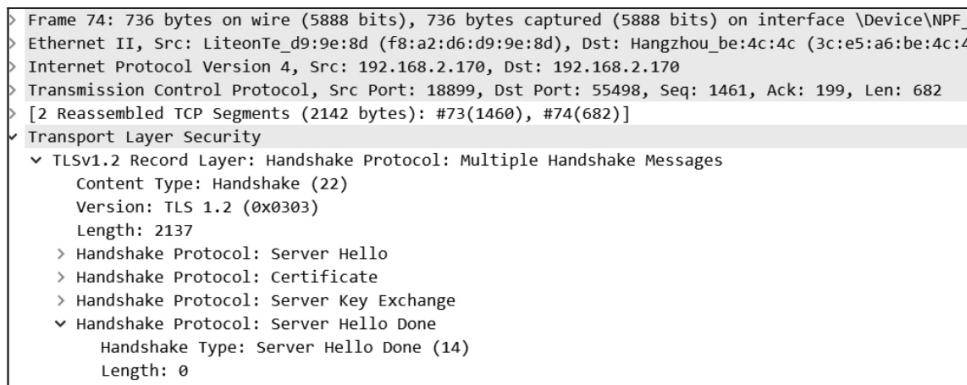
(4) Signature

其签名算法使用Client Hello握手消息Extension拓展中提供的签名算法对服务端发送的部分数据进行签名。客户端使用这段内容验证报文的有效性。

4. Server Hello Done帧

Server Hello Done帧是第二阶段的最后一帧，标记服务端对客户端的Client Hello请求帧的所有响应报文发送完毕，Server Hello Done帧的长度为0。

使用WireShark抓取到的服务端的Server Hello Done报文实例如图12-12所示。



```
Frame 74: 736 bytes on wire (5888 bits), 736 bytes captured (5888 bits) on interface \Device\NPF_{...}
  Ethernet II, Src: LiteonTe_d9:9e:8d (f8:a2:d6:d9:9e:8d), Dst: Hangzhou_be:4c:4c (3c:e5:a6:be:4c:4c)
  Internet Protocol Version 4, Src: 192.168.2.170, Dst: 192.168.2.170
  Transmission Control Protocol, Src Port: 18899, Dst Port: 55498, Seq: 1461, Ack: 199, Len: 682
  [2 Reassembled TCP Segments (2142 bytes): #73(1460), #74(682)]
  Transport Layer Security
    TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 2137
      Handshake Protocol: Server Hello
      Handshake Protocol: Certificate
      Handshake Protocol: Server Key Exchange
      Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

图12-12 使用WireShark抓取的服务端所发送的Server Hello Done帧实例

客户端收到服务端证书后，进行验证，如果证书不是可信机构颁发的，或者域名不一致，或者证书已经过期，那么客户端会进行警

告；如果证书没有问题，就继续进行通信。

12.3.3 SSL/TLS第三阶段握手

SSL/TLS握手（Handshake）第三个阶段的工作为：客户端进行回应。在这个阶段，客户端会发送Client Key Exchange、Change Cipher Spec、Encrypted Handshake三个数据帧。

客户端收到第二个阶段的服务端回应报文以后，首先验证服务端证书。如果证书不是可信机构颁布、或者证书中的域名与实际域名不一致、或者证书已经过期，就会向访问者显示一个警告，由其选择是否还要继续通信。

如果证书没有问题，客户端就会从证书中取出服务端的公钥，然后向服务端发送三项信息：

(1) 一个随机数。该随机数用服务端公钥加密，防止被第三方窃听。

此随机数是整个握手阶段出现的第三个随机数，又称Pre-master key。有了它以后，客户端和服务端就同时有了三个随机数，接着双方用事先商定的加密方法各自生成本次会话所用的同一把“会话密钥”。

(2) 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥加密后发送。

(3) 客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时也是前面发送的所有内容的哈希值，用来供服务端校验。

服务端的证书信息会包含Public Key（公钥），稍后客户端进行证书验证（身份验证）的流程大致为：Client随机生成一串数，然后用Server发送的Public Key加密（RSA算法）后发给Server；而Server会用其对应的Private key（私钥）解密后再返回给Client；Client将其与原文比较，如果一致，则说明Server拥有Private key，与自己通信的对端Server正是证书的拥有者，因为Public key加密的数据只有Private key才能解密。在实际通信过程中，这个认证过程会复杂很多，包含多次哈希、伪随机等复杂运算。

1. Client Key Exchange帧

服务端的身份证书验证通过后，客户端会生成整个握手过程中的第三个随机数，并且从证书中取出公钥，利用公钥以及双方实现商定的加密算法进行加密，生成Pre-master key，然后发送给服务端，如图12-13所示。

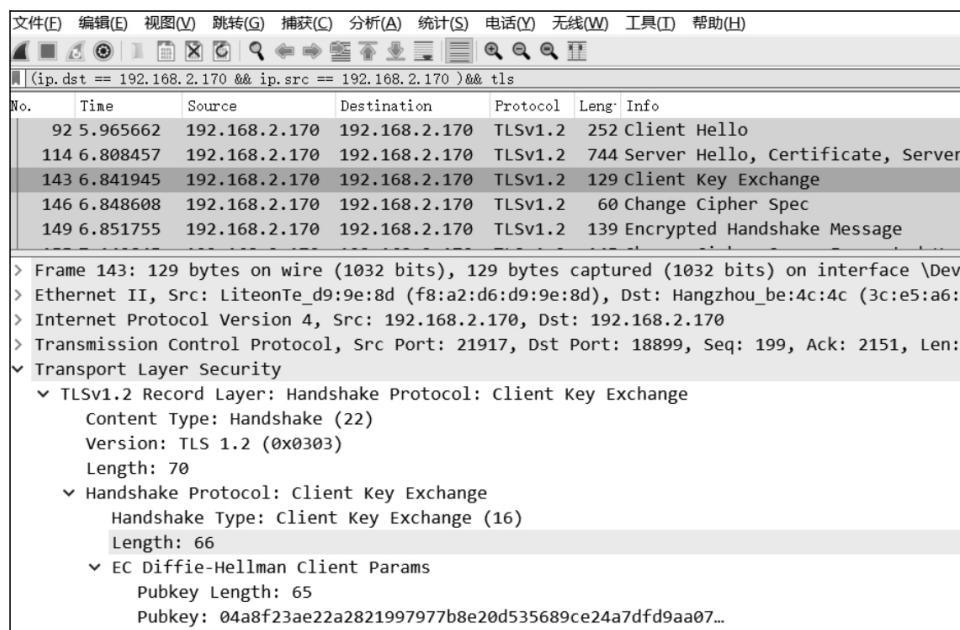


图12-13 使用Wireshark抓取的客户端所发送的Client Key Exchange帧实例

Client Key Exchange帧的内容大致如下：

(1) Handshake Type

此字段标识当前握手报文的类型，这里为Client Key Exchange类型，其值为16。

(2) Pubkey

客户端将生产的随机数Pre-master key通过利用服务端公钥以及双方前期商定的加密算法（这里为DH算法）计算得到的Pubkey生成服务端解密私钥。服务端收到Pubkey后，利用私钥解密出第三个随机数Pre-master key，此时客户端和服务端同时拥有了三个随机数：Random_C、Random_S、Pre-master key，两端同时利用这三个随机数以及事先商定好的加密算法进行对称加密，生成最终的“会话密钥”，后续的通信都用该密钥进行加密。

在“会话密钥”的生成过程中，由于第三个随机数Pre-master key是通过非对称加密进行加密的，因此不容易泄漏，即“会话密钥”是安全的，后续的通信也就是安全的。那么为什么一定要用三个随机数来生成“会话密钥”呢？一方面，能保证这样生成的密钥每次都不一样；另一方面，由于SSL协议中的证书是静态的，因此十分有必要引入一种随机因素来保证协商出来的密钥的随机性。

客户端随机数、服务端随机数再加上Pre-master key一共三个随机数，一同生成的密钥的最大优势为：不容易被预测出来，做到真正的没有规律。为什么呢？虽然一个伪随机数可能不完全随机，但是三个伪随机数就十分接近真正的随机和没有规律了。

2. Change Cipher Spec帧

编码改变通知，客户端通知服务端，随后的信息都是用商定好的加密算法和“会话密钥”加密发送，如图12-14所示。

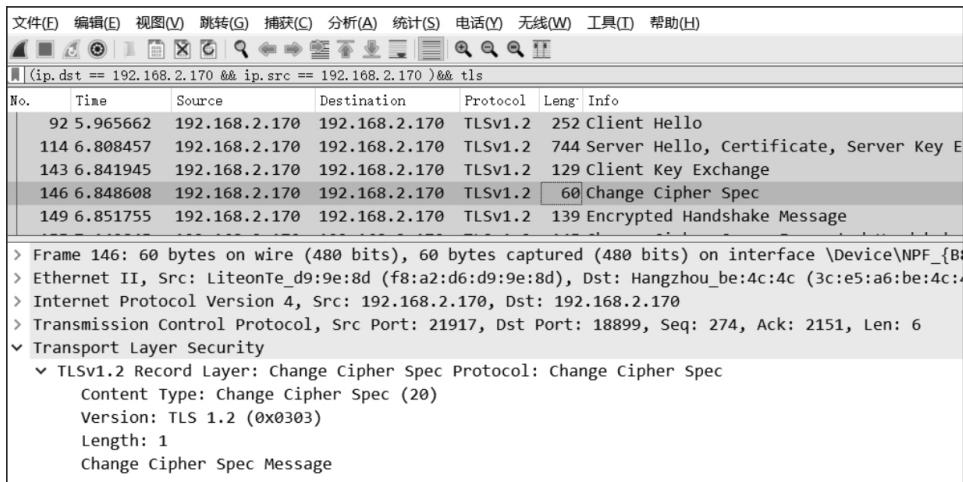


图12-14 使用Wireshark抓取的客户端所发送的Change Cipher Spec帧实例

3. Encrypted Handshake Message帧

客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时也是前面发送的所有内容的哈希值，用来供服务端进行安全校验，如图12-15所示。

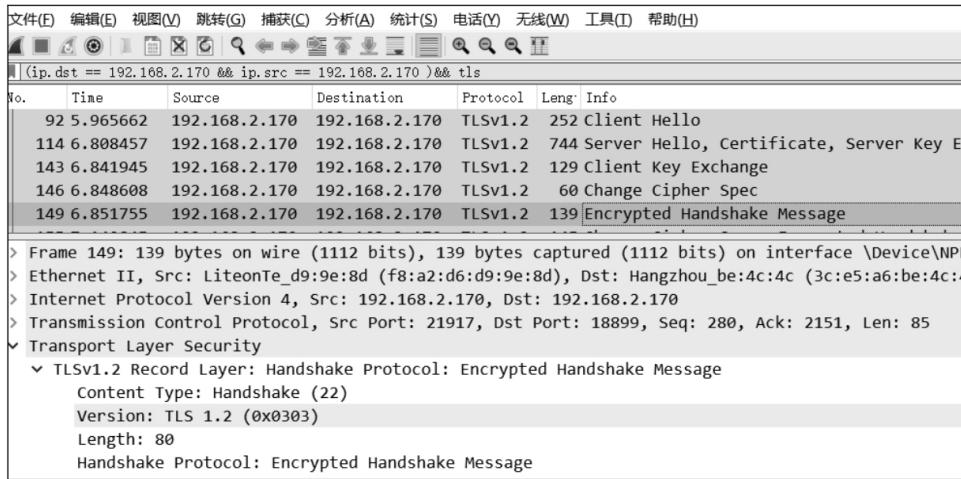


图12-15 使用Wireshark抓取的客户端所发送的Encrypted Handshake Message帧实例

12.3.4 SSL/TLS第四阶段握手

SSL/TLS握手（Handshake）第四个阶段的工作为：服务端进行最后的回应。在收到客户端的第三个随机数Pre-master key之后，服务端计算并生成本次会话所用的“会话密钥”，然后向客户端最后发送下面的数据帧：

(1) Change Cipher Spec帧：此帧为服务端的编码改变通知报文。

(2) Encrypted Handshake Message帧：此帧为服务端的握手结束通知报文。

1. Change Cipher Spec帧

此帧为编码改变通知报文，服务端通知客户端，随后的通信内容都是用商定好的加密算法和会话密钥加密后再发送，如图12-16所示。

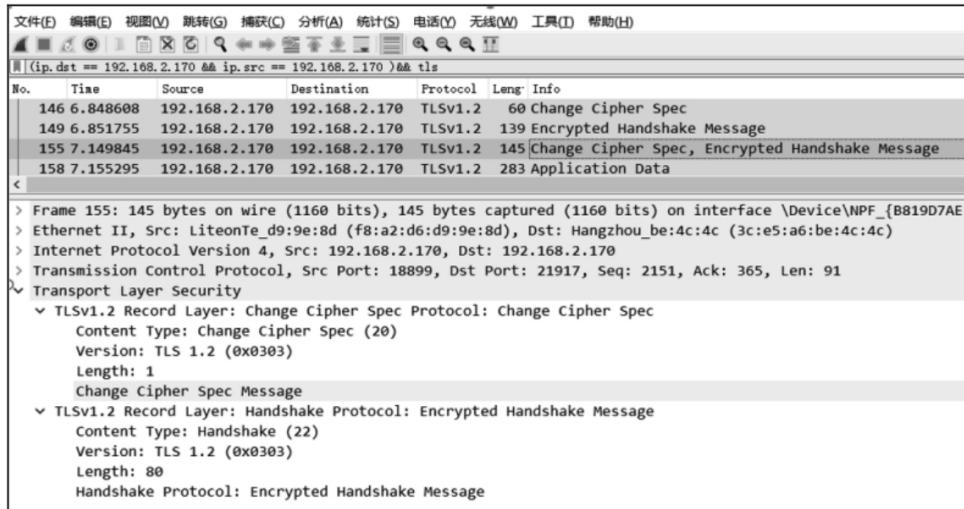


图12-16 使用Wireshark抓取的服务端所发送的Change Cipher Spec帧实例

2. Encrypted Handshake Message帧

服务端握手结束通知帧，表示服务端的握手阶段已经结束。这一报文同时包括前面发送的所有内容的哈希值，用来供客户端进行一次简单的安全校验，如图12-17所示。

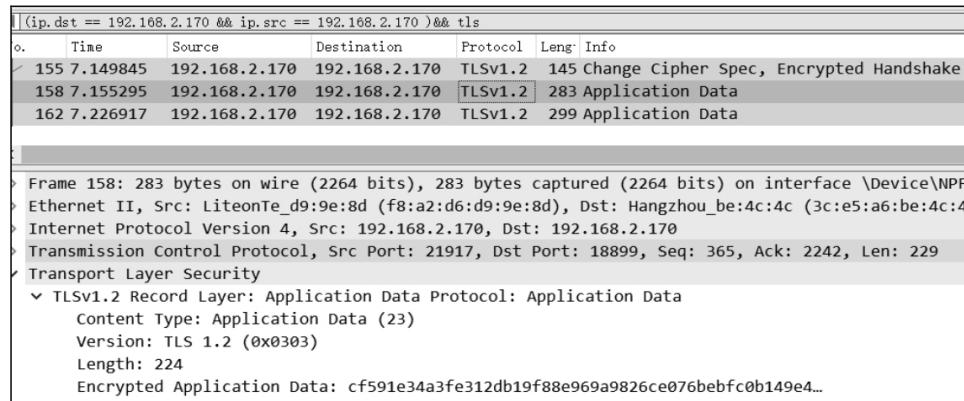


图12-17 使用Wireshark抓取的服务端所发送的Encrypted Handshake Message帧实例

至此，整个握手阶段全部结束。接下来，客户端与服务端开始进入安全通信的过程，此通信过程仍然是使用普通的应用层协议（如HTTP协议或者WebSocket协议）完成，只不过应用层报文内容用“会话密钥”加密。

12.4 详解Keytool工具

SSL/TSL在握手过程中，客户端需要服务端提供身份证书（也叫数字证书），有的场景下甚至要求客户端也提供身份证书。安全数字证书主要包含自己的身份信息（如所有人的名称），以及对外的公钥。

12.4.1 数字证书与身份识别

在SSL/TSL加密传输开始的时候，客户端会通过Client Hello帧获得服务端的公钥，这个过程可能会被第三方劫持，具体如图12-18所示。

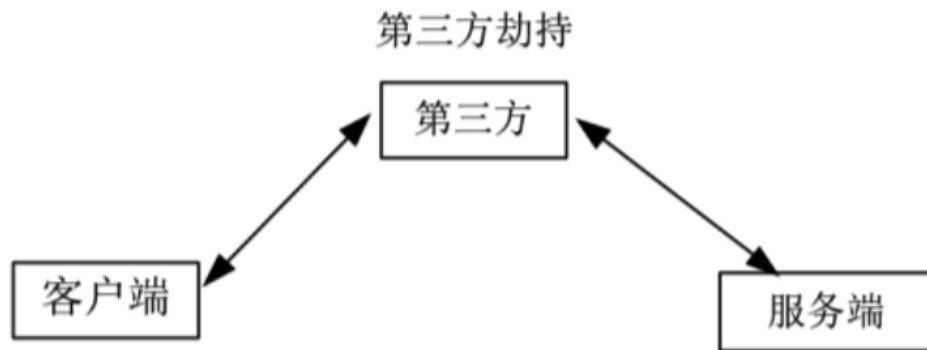


图12-18 客户端的Client Hello请求被第三方劫持

当客户端的Client Hello帧被劫持时，服务端发送到客户端的公钥会被第三方截获，然后第三方自己会伪造一对密钥（包含公钥和私钥），并将伪造的公钥发送给客户端。当服务端发送数据给客户端的时候，第三方也会将信息劫持，用一开始截获的公钥进行解密，然后使用自己的私钥将数据再一次加密后发送给客户端，客户端收到后使

用第三方（劫持方）的公钥去解密。反过来也是如此，当客户端发送数据给服务端时，报文亦会被劫持方截取和转发，并且整个截取和转发的过程对于客户端和服务端都是透明和不可见的，但信息却被悄然泄露了。

数据帧被劫持的过程如图12-19所示。

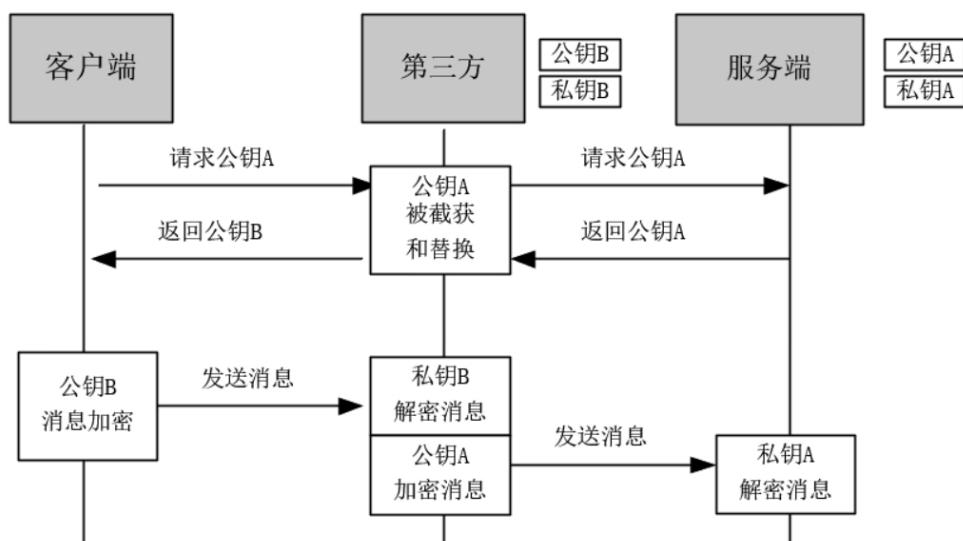


图12-19 数据帧被劫持的过程

为了防止这种情况，数字证书出现了。数字证书就是互联网通信中标志通信各方身份信息的一串数字，是由权威机构——CA (Certificate Authority, 认证中心) 发行的，人们可以在网上用它来识别对方的身份。

一般数字证书的颁发过程为：用户首先产生自己的密钥对，并将公钥及身份信息提供给CA机构（认证中心）。认证中心在核实身份后，将执行一些必要的步骤，以确信请求确实是用户提交的，然后认证中心将发给用户一个数字证书。一个证书中含有三个部分：证书内容、哈希算法、加密密文。该证书内包含服务端的个人信息和公钥

信息；加密密文为证书内容通过哈希算法计算出摘要之后使用CA机构的私钥进行非对称加密后的密文，也可以理解成CA机构自己的数字签名。

当客户端发起请求时，服务端将该数字证书发送给客户端，客户端需要对证书进行验证，具体方法为：通过CA机构提供的公钥对服务端的证书的数字签名（加密密文）进行解密，以获得服务端证书的内容摘要（哈希值），同时将证书内容使用相同的哈希算法获取摘要，对比两个摘要。如果两者相等，就说明证书中的公钥仍然是服务端原始公钥，没有被第三方篡改，即服务端证书没有问题、服务端并没有被劫持。

数字证书的格式普遍采用的是X.509国际标准。X.509是一种进行身份认证的行业安全标准，在该标准中，用户可生成一段信息及其摘要（亦称作信息“指纹”），并用专用密钥对摘要加密以形成签名，接收者用发送者的公共密钥对签名解密，并将之与收到的信息“指纹”进行比较，以确定其真实性。

X.509标准有不同的版本，其中X.509/V2和X.509/V3都是目前比较新的版本，但是都在原有版本（X.509/V1）的基础上进行功能的扩充。每一版的数字证书大致包含下列信息：

- (1) 证书的版本信息。
- (2) 证书的序列号，每个证书都有一个唯一的证书序列号。
- (3) 证书所使用的签名算法。
- (4) 证书的发行机构名称，命名规则一般采用X.500协议格式。

(5) 证书的有效期，通用的证书一般采用UTC时间格式，计时范围为1950—2049。

(6) 证书所有人的名称，命名规则一般采用X. 500协议格式。

(7) 证书所有人的公开密钥。

(8) 证书发行者对证书的数字签名。

说明

命名规则一般采用X. 500协议格式。X. 500协议可以理解为用来查询有关人员信息（如邮政地址、电话号码、电子邮件地址等）的一种协议。X. 500协议是构成全球分布式的名录服务系统的协议，该协议组织起来的数据就像一个很全的电话号码簿。X. 500系统是一个分门别类的图书馆，某一机构建立和维护的X. 500子数据库只是全球X. 500协议名录数据库的一部分。

通过浏览器的“管理证书”入口可以查看到浏览器所缓存的服务端证书。图12-20是一个浏览器缓存的一个服务端证书的例子。



图12-20 查看到浏览器所缓存的服务端证书

在校验证书时，浏览器会用到CA机构的公钥。实际上，浏览器和操作系统都会维护一个权威、可行的第三方CA机构列表（包括它们的公钥）。客户端接收到的证书中也会写有颁发机构，客户端根据这个颁发机构找到其公钥，然后完成证书的校验。

12.4.2 存储密钥与证书文件格式

SSL/TLS协议中存储密钥与证书的文件格式比较多，很容易被大家搞混，这里做个简单的梳理，大致会用到的文件格式如下：

(1) .jks

“.jks”格式文件表示Java密钥存储仓库（Java KeyStore）。这种格式是Java的专利，表示一个密钥库，可以同时容纳多个公钥和私钥。Java的Keytool工具能直接生成“.jks”格式文件，可以将“.pfx”格式文件转为“.jks”格式文件。

(2) .keystore

“.keystore”格式文件跟“.jks”基本是一样的，是默认生成的密钥存储库格式。

(3) .cer

“.cer”格式文件俗称数字证书文件，其中只包含了公钥以及证书拥有者和颁发者的消息。数字证书文件肯定不会有私钥。“.cer”格式文件既可以是BASE64编码的文本文件，也可以是DER编码的二进制文件。

可以通过Java的Keytool工具将“.cer”证书文件导入密钥存储仓库（如“.jks”格式文件），或者从密钥存储仓库导出证书文件，如图12-21所示。

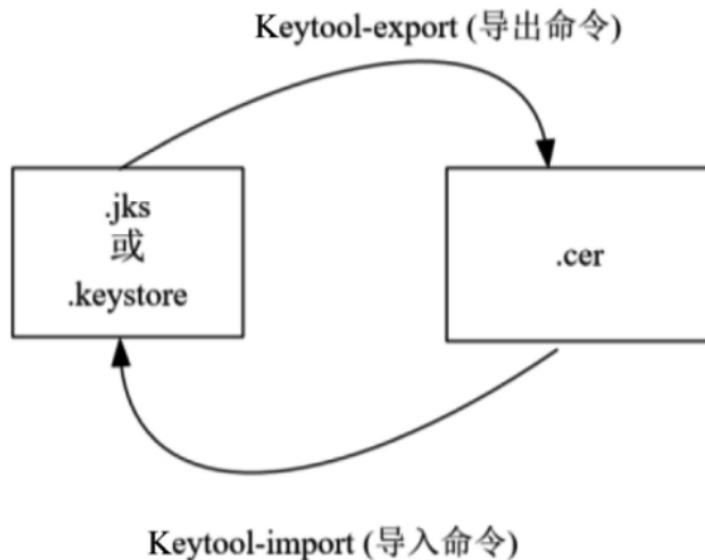


图12-21 证书文件和密钥仓库之间的互导

(4) .truststore

“.truststore”格式文件表示信任证书存储库，仅仅包含了被信任的通信对方的公钥。

(5) .pfx

“.pfx”格式文件也称为证书文件，是包含了公钥和私钥的二进制格式的证书文件，一般供客户端浏览器使用。与“.cer”格式文件不同，“.pfx”格式的数字证书是包含有私钥的，而“.cer”格式的数字证书里面只有公钥。当然，“.pfx”格式文件一般有密码保护，不输入密码是解不了密的。

有些时候我们需要把“.pfx”转换为“.jks”密钥仓库，以便于用Java进行安全通信，也可以通过浏览器从“.pfx”文件中导出包含公钥的“.cer”证书文件，具体如图12-22所示。

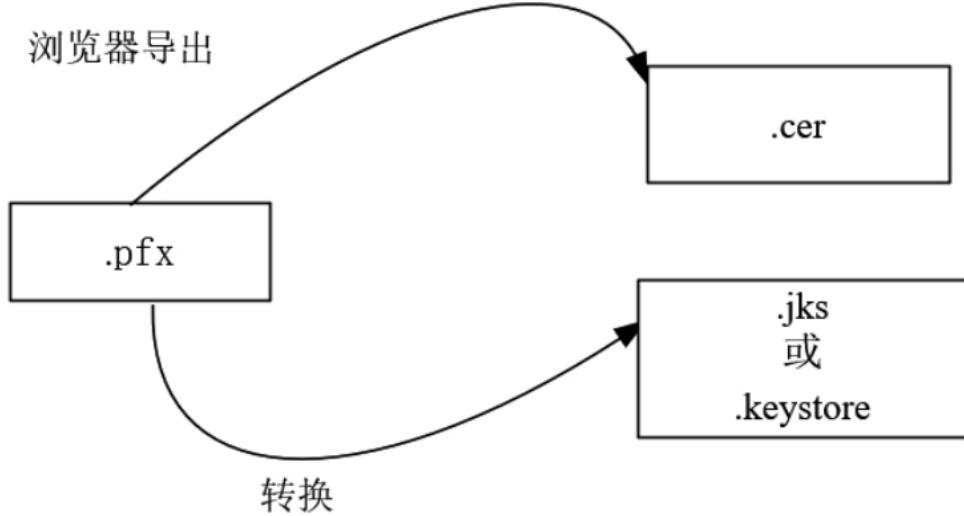


图12-22 “.pfx” 转换为 “.jks” 密钥仓库或导出证书

12.4.3 使用Keytool工具管理密钥和证书

除了从CA机构获取证书，还可以通过工具生成自签名证书。CA机构证书是需要费用的，除非是很正式的项目或者生产需要（比如微信小程序不能使用自签名证书而需要CA证书），否则使用自己签发的证书即可。

Java中管理和生成自签名证书的工具为Keytool。Keytool是Java中自带的工具，将密钥（Key）和证书（Certificates）存在一个格式为“.keystore”（或.jks）的文件中，然后可以导出自签发的数字证书。在JDK安装过程中，Keytool工具已经解压到对应的JDK的/bin目录中，其可执行文件的文件名为keytool.exe。

作为铺垫，首先介绍一下使用密钥的场景：假设客户端需要和服务端进行安全通信，客户端要用到服务端公钥进行通信加密。在这种

场景下，首先需要生成服务端和客户端的密钥仓库，然后导出服务端证书，并导入到客户端密钥仓库中，具体流程如图12-23所示。

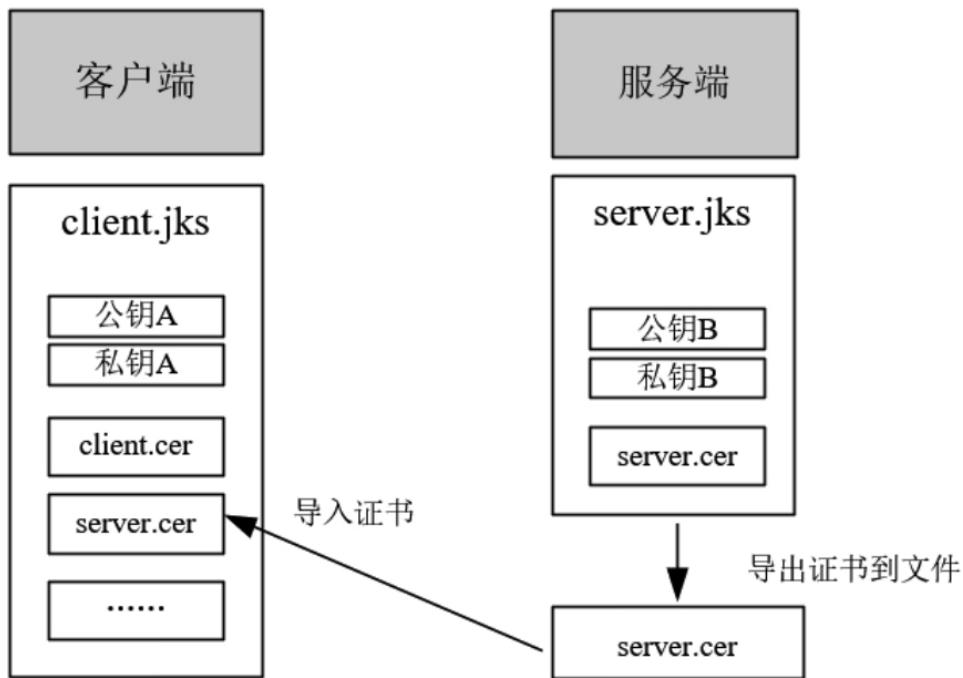


图12-23 将服务端证书导入客户端仓库

将服务端证书导入客户端的工作使用Keytool工具大致有如下四步。

(1) 第一步：创建服务端（如Netty服务器）密钥并且保存到服务端密钥仓库文件。使用Keytool工具的genkey选项完成，具体命令大致如下：

```
keytool -genkey -alias server -keypass 123456 -keyalg RSA -  
keysize 2048 -validity 365 -keystore f:\server.jks -storepass  
123456 -dname "CN=server"
```

对于以上Keytool命令用到的常用选项，大致说明如下：

① -genkey：该选项主要用于创建密钥，并且保存到密钥仓库。

② -alias：该选项用于设置密钥别名，每个密钥都关联一个独一无二的别名，别名通常不区分字母大小写。

③ -keypass：该选项用于设置指定密钥的访问密码，也就是私钥的原始密码。

④ -keyalg：该选项用于指定密钥的加密算法，如RSA、DSA等，如果不指定，则默认采用DSA非对称加密算法，这里指定为RSA非对称加密算法。

⑤ -keysize：该选项用于指定密钥长度，示例中设置的密钥长度为2048位，这个长度的密钥目前可认为无法被暴力破解。

⑥ -validity：该选项用于指定创建的密钥有效期为多少天。365表示证书的有效期为365天。

⑦ -keystore：该选项用于指定生成的密钥仓库文件，示例中的仓库文件为f:\server.jks，如果只指定文件名而不指定路径，那么会生成至当前的系统用户目录下。如果不指定，则会在当前的系统用户目录下创建一个“.keystore”默认仓库文件。对于2010版本的Windows系统，该文件处于C:\Users\<UserName>\目录下。

如果密钥仓库文件已经存在，将不会创建新的仓库文件，而是直接将密钥加入现有的仓库文件中。

⑧ -storepass：该选项用于指定密钥仓库的访问密码。其实这个密码和密钥密码keypass可以设置成一样的，主要是为了方便记忆。

⑨ -dname：该选项用于指定X.500协议格式的证书拥有者信息，例如“CN=名字与姓氏, OU=组织单位名称, O=组织名称, L=城市或区域名称, ST=州或省份名称, C=用两字母代表的国家或地区代码”。这里设置“CN=server”表示密钥拥有者的名称为server。

生成密钥后，可以使用Keytool工具的list选项查看服务端（如Netty服务器）的密钥仓库，具体的输出大致如下：

```
C:\Users\UserName\.ssh> keytool -list -v -keystore  
f:\server.jks  
输入密钥库口令： 123456
```

密钥库类型：JKS

密钥库提供方：SUN

您的密钥库包含 1 个条目

别名：server

创建日期：2020-5-23

条目类型：PrivateKeyEntry

证书链长度：1

证书[1]：

所有者：CN=server

发布者：CN=server

序列号：7cef8ac8

有效期开始日期：Sat May 23 16:07:56 CST 2020, 截止日期：Sun May 23
16:07:56 CST 2021

证书指纹：

MD5： 4A:02:4F:DB:AD:69:68:39:A9:DC:78:E1:D8:9E:0F:F7

SHA1：

04:14:63:D6:68:1C:14:FC:FE:AA:25:05:B2:65:36:47:4C:4D:9B:29

SHA256：

8C:ED:B5:15:B5:5B:A5:1E:11:40:67:67:0E:A9:A0:A5:0E:C9:F8:3C:E4:

B6:64:FE:01:1C:78:F7:4B:1E:41:2C

签名算法名称： SHA256withRSA

版本： 3

扩展：

```
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
    KeyIdentifier [
        0000: C8 C5 19 3E F3 13 89 5C      3A 2A 84 44 BF 32 E3 FB ...>...
        \:*.D.2..
        0010: 5B 30 9F 75                      [0.u
    ]
]
```

(2) 第二步：生成客户端的密钥到客户端的密钥仓库。还是使用 Keytool 工具的 genkey 选项完成，具体的命令大致如下：

```
keytool -genkey -alias client -keysize 2048 -validity 365 -
keyalg RSA -dname "CN=client" -keypass 123456 -storepass 123456
-keystore f:/client.jks
```

(3) 第三步：需要将服务端的证书导出，然后导入到客户端的授信证书仓库（这里使用客户端密钥仓库）中。首先通过Keytool工具的export选项完成服务端的数字证书“server.cer”文件导出，具体的命令大致如下：

```
keytool -export -alias server -keystore f:/server.jks -  
storepass 123456 -file server.cer
```

证书被导出后，可以使用Keytool工具的printcert选项查看所导出的证书中的内容，具体的命令为：

```
C:\Users\UserName> keytool -printcert -file server.cer  
所有者: CN=server  
发布者: CN=server  
序列号: 7cef8ac8  
有效期开始日期: Sat May 23 16:07:56 CST 2020, 截止日期: Sun May 23  
16:07:56 CST 2021
```

证书指纹：

MD5: 4A:02:4F:DB:AD:69:68:39:A9:DC:78:E1:D8:9E:0F:F7

SHA1:

04:14:63:D6:68:1C:14:FC:FE:AA:25:05:B2:65:36:47:4C:4D:9B:29

SHA256:

8C:ED:B5:15:B5:5B:A5:1E:11:40:67:67:0E:A9:A0:A5:0E:C9:F8:3C:E4:
B6:64:FE:01:1C:78:F7:4B:1E:41:2C

签名算法名称: SHA256withRSA

版本: 3

扩展：

```
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
    KeyIdentifier [
        0000: C8 C5 19 3E F3 13 89 5C      3A 2A 84 44 BF 32 E3 FB ...>...
        \:*.D.2..
        0010: 5B 30 9F 75                      [0.u
    ]
]
```

(4) 第四步：将服务的证书导入到客户端仓库（严格来说是信任仓库，只不过可以和密钥仓库合用）。使用Keytool工具的import选项完成，具体的命令大致如下：

```
keytool -import -trustcacerts -alias server -file server.cer -
keystore f:/client.jks -storepass 123456
```

导入过程中会提示是否信任该证书，在确认之后，证书就会被成功添加到密钥库中。客户端就可以和服务器进行安全通信了。

最后介绍一下Keytool工具的常用选项：

① -list选项用于查看一个密钥存储仓库文件（如Java KeyStore）中的密钥和证书。其具体使用示例如下：

```
keytool -list -v -keystore f:\client.jks
```

② -export选项用于从密钥存储仓库文件中导出一个证书文件。其具体使用示例如下：

```
keytool -export -alias server -keystore f:/server.jks -  
storepass 123456 -file server.cer
```

③ -import选项用于添加一个信任证书到密钥存储仓库文件。使用示例如下：

```
keytool -import -trustcacerts -alias server -file server.cer -  
keystore f:/client.jks
```

④ -delete选项用于根据别名从密钥存储仓库文件中删除一个证书或者密钥。其具体使用示例如下：

```
keytool -delete -keystore server.jks -alias server
```

上面通过Keytool工具管理密钥和证书，也是大家日常用得比较多的方式。除此之外，还可以通过Java程序完成密钥和证书的管理。

12.5 使用Java程序管理密钥与证书

作为铺垫，回顾一下上一节使用密钥的场景：假设客户端需要和服务端进行安全通信，客户端要用到服务端公钥进行通信加密。在这种场景下，首先需要生成服务端和客户端的密钥仓库，然后导出服务端证书，并导入到客户端密钥仓库中。上一节通过Keytool工具管理密钥和证书，本节将通过Java程序完成同样的操作。

12.5.1 使用Java操作数据证书所涉及的核心类

使用Java代码去创建、管理密钥、密钥仓库、数字证书会用到的核心类如表12-3所示。

表12-3 使用Java操作密钥和仓库涉及的核心类

Java类	说 明
java.security.KeyStore	此类表示密钥和证书的存储设施，也就是密钥仓库
java.security.PrivateKey	私钥的超级接口，包括 DHPrivateKey、DSAPrivateKey、ECPrivateKey、RSAPrivateCrtKey、RSAPrivateKey 等子接口
java.security.PublicKey	公钥的超级接口
java.security.Signature	数字签名算法类，签名算法可以为 MD5withRSA 或 SHA1withRSA 等。获取数字签名时，没有默认的算法名称，所以必须为其指定名称
java.security.cert.Certificate	数字证书的抽象类。不同的证书类型（X.509、PGP 等）共享通用的证书功能（如编码和验证）和部分信息类型（如公钥等）
java.security.cert.X509Certificate	X.509 证书的抽象类。此类提供了一种访问 X.509 证书所有属性的标准方式

12.5.2 使用Java程序创建密钥与仓库

在前面的章节中，使用Keytool工具的genkey选项完成了服务端密钥创建并且保存到密钥仓库文件server.jks，具体的命令大致如下：

```
keytool -genkey -alias server -keypass 123456 -keyalg RSA -  
keysize 2048 -validity 365 -keystore f:\server.jks -storepass  
123456 -dname "CN=server"
```

和使用Keytool工具类似，通过Java程序创建密钥和仓库时，也需要用到以下信息：密钥别名、私钥密码、密钥的加密算法、密钥有效期、密钥仓库文件、证书拥有者信息等。

这里实现了一个KeyStoreHelper帮助类，用于帮助创建密钥和证书，并且保存到密钥仓库文件，其代码节选如下：

```
package com.crazymakercircle.keystore;  
//省略import  
public class KeyStoreHelper  
{  
    private static final byte[] CRLF = new byte[]{'\r', '\n'};  
    /**  
     * 存储密钥仓库的文件  
     */  
    private String keyStoreFile;  
  
    /**  
     * 获取KeyStore信息所需的密码  
     */  
    private String storePass;
```

```
/***
 * 设置指定别名条目的密码，也就是私钥原始密码
 */
private String keyPass;

/***
 * 每个KeyStore都关联一个独一无二的别名，这个别名通常不区分大小写
 */
private String alias;

/***
 * 指定证书拥有者信息
 * 例如："CN=名字与姓氏,OU=组织单位名称,O=组织名称,L=城市或区域名称,ST=州或省份名称,C=用两字母代表的国家或地区代码"
 */
private String dname ;
KeyStore keyStore;

private static String keyType = "JKS";

public KeyStoreHelper(String keyStoreFile, String
storePass,
String keyPass, String alias, String
dname)
{
    this.keyStoreFile = keyStoreFile;
    this.storePass = storePass;
```

```
        this.keyPass = keyPass;
        this.alias = alias;
        this.dname = dname;
    }

/**
 * 创建密钥和证书并且保存到密钥仓库文件
 */
public void createKeyEntry() throws Exception
{
    KeyStore keyStore = loadStore();
    CertHelper certHelper = new CertHelper(dname);
    /**
     * 生成证书
     */
    Certificate cert = certHelper.genCert();
    cert.verify(certHelper.getKeyPair().getPublic());
    PrivateKey privateKey =
    certHelper.getKeyPair().getPrivate();

    //访问仓库时需要用到仓库密码
    char[] caPasswordArray = storePass.toCharArray();
    /**
     * 设置密钥和证书到密钥仓库
     */
    keyStore.setKeyEntry(alias, privateKey,
```

```
        caPasswordArray, new Certificate[] {cert});
```



```
FileOutputStream fos = null;
```

```
    try
```

```
    {
```

```
        fos = new java.io.FileOutputStream(keyStoreFile);
```

```
        /**
         * 密钥仓库保存到文件
         */
        keyStore.store(fos, caPasswordArray);
```

```
    } finally
```

```
    {
```

```
        closeQuietly(fos);
    }
```

```
}
```



```
/**
 * 从文件加载KeyStore密钥仓库
 */
```

```
public KeyStore loadStore() throws Exception
```

```
{
```

```
    log.debug("keyStoreFile: {}", keyStoreFile);
```

```
    if (!new File(keyStoreFile).exists())
```

```
    {
        createEmptyStore();
    }
```

```
    KeyStore ks = KeyStore.getInstance(keyType);
```

```
java.io.FileInputStream fis = null;
try
{
    fis = new java.io.FileInputStream(keyStoreFile);
    ks.load(fis, storePass.toCharArray());
} finally
{
    closeQuietly(fis);
}
return ks;
}

/**
 * 建立一个空的KeyStore仓库
 */
private void createEmptyStore() throws Exception
{
    KeyStore keyStore = KeyStore.getInstance(keyType);
    File parentFile = new
File(keyStoreFile).getParentFile();
    if (!parentFile.exists())
    {
        parentFile.mkdirs();
    }
    java.io.FileOutputStream fos = null;
    keyStore.load(null, storePass.toCharArray());
    try
```

```
        {

            fos = new java.io.FileOutputStream(keyStoreFile);

            keyStore.store(fos, storePass.toCharArray());

        } finally

        {

            closeQuietly(fos);

        }

    }

//...
```

使用此KeyStoreHelper类完成创建服务端（如Netty服务器）密钥并且保存到服务端密钥仓库文件，其代码如下：

```
package com.crazymakercircle.secure.Test.keyStore;

//省略import

public class ServerKeyStoreTester

{



    /**
     * 存储密钥的文件
     */

    private String keyStoreFile=

        SystemConfig.getKeystoreDir() +



    "/server.jks";





    /**

```

```
* 访问KeyStore时所需的密码
*/
private String storePass = "123456";
/***
 * 设置指定别名条目的密码，也就是私钥密码
*/
private String keyPass = "123456";

/***
 * 每个KeyStore都关联一个独一无二的别名，这个别名通常不区分字母大小写
*/
private String alias= "server_cert";

/***
 * 指定证书拥有者信息
 * 例如： "CN=名字与姓氏,OU=组织单位名称,O=组织名称,L=城市或区域名
称,ST=州或省份名称,C=用两字母代表的国家或地区代码"
*/
private String dname =
"C=CN, ST=Province, L=city, O=crazymaker, OU=crazymaker.com, CN=serv
er";
/***
 * 创建密钥和证书并且保存到密钥仓库文件
*/
@Test
```

```
public void testCreateKey() throws Exception
{
    KeyStoreHelper keyStoreHelper = new
    KeyStoreHelper(keyStoreFile,
                  storePass, keyPass, alias, dname);
    //创建密钥和证书
    keyStoreHelper.createKeyEntry();
}

/**
 * 在服务端仓库，打印仓库的所有证书
 */
@Test
public void testPrintEntries() throws Exception
{
    String dir = SystemConfig.getKeystoreDir();
    log.debug(" client dir = " + dir);
    KeyStoreHelper keyStoreHelper = new KeyStoreHelper(
        keyStoreFile, storePass, keyPass, alias,
        dname);
    //打印仓库的所有证书
    keyStoreHelper.doPrintEntries();
}

//...
```

}

运行上面的第一个测试用例，会在工程目录下创建一个 server.jks 密钥仓库文件；运行第二个测试用例，会打印仓库的所有证书，大致输出如下：

```
[main] DEBUG ServerKeyStoreTester - client dir = F:\ ...
\SecureTransferDemo

[main] DEBUG KeyStoreHelper - keyStoreFile: F:\ ... \server.jks

[main] INFO KeyStoreHelper - server_cert 别名的证书信息如下:

[main] INFO KeyStoreHelper - Owner: C=CN, ST=Province, L=city,
O=crazymaker, OU=crazymaker.com, CN=server

[main] INFO KeyStoreHelper - Issuer: C=CN, ST=Province, L=city,
O=crazymaker, OU=crazymaker.com, CN=server

[main] INFO KeyStoreHelper - Serial number: 1

[main] INFO KeyStoreHelper - Valid from: Sat May 23 21:21:18
CST 2020

[main] INFO KeyStoreHelper - Valid until: Sun May 23 21:21:18
CST 2021

[main] INFO KeyStoreHelper - Certificate fingerprints SHA1:
[main] INFO KeyStoreHelper -
2C:5B:D7:64:4C:70:E6:36:1F:4C:A0:7E:24:05:60:4E:EB:6D:8C:D8

[main] INFO KeyStoreHelper - Certificate fingerprints SHA256:
[main] INFO KeyStoreHelper -
84:BB:D7:52:43:19:42:AD:29:D3:0C:B3:A3:A1:53:E9:68:73:80:54:F3:
82:18:1F:9D:E5:40:1E:9A:2C:9F:7A
```

```
[main] INFO KeyStoreHelper - Signature algorithm name:  
SHA256withRSA  
[main] INFO KeyStoreHelper - Version: 3
```

12.5.3 使用Java程序导出证书文件

在前面的章节中，导出服务端的证书是使用Keytool工具的export选项完成的，命令如下：

```
keytool -export -alias server -keystore f:/server.jks -  
storepass 123456 -file server.cer
```

在帮助类KeyStoreHelper中使用Java代码实现数字证书文件（“.cer”文件）导出的代码，其方法名称为exportCert，代码如下：

```
package com.crazymakercircle.keystore;  
//省略import  
public class KeyStoreHelper  
{  
    //省略成员属性  
    /**  
     * 导出证书  
     * @param outDir 导出的目标目录  
     */  
    public boolean exportCert(String outDir) throws Exception  
    {
```

```
    assert (StringUtils.isNotEmpty(alias));
    assert (StringUtils.isNotEmpty(keyPass));
    KeyStore ks = loadStore();
    PasswordProtection protection =
        new
    PasswordProtection(keyPass.toCharArray());
    if (ks.isKeyEntry(alias))
    {
        //根据别名获取密钥条目
        PrivateKeyEntry entry=
            (PrivateKeyEntry) ks.getEntry(alias,
protection);
        //从密钥条目中获取证书
        X509Certificate cert =
            (X509Certificate)
entry.getCertificate();
        //进行过期校验
        if (new Date().after(cert.getNotAfter()))
        {
            return false;
        }
    } else
    {
        //导出到文件
        String certPath = outDir + "/" + alias +
".cer";
        FileWriter wr =

```

```

        new java.io.FileWriter(new
File(certPath));

        String encode =
new
BASE64Encoder().encode(cert.getEncoded()));

        String strCertificate = "-----BEGIN
CERTIFICATE-----\r\n"
+ encode + "\r\n-----END CERTIFICATE---"
--\r\n";
//写入证书的编码

        wr.write(strCertificate);

        wr.flush();
        closeQuietly(wr);
        return true;
    }

}

return false;
}

//...
}

```

调用此KeyStoreHelper类的exportCert()方法，导出创建服务端（如Netty服务器）密钥的数字证书，其测试用例代码如下：

```

package com.crazymakercircle.secure.Test.keyStore;
//省略import
@Slf4j

```

```
public class ServerKeyStoreTester
{
    /**
     * 服务端密钥仓库测试用例
     */
    @Test
    public void testExportCert() throws Exception
    {
        String dir = SystemConfig.getKeystoreDir();
        log.debug("dir = " + dir);
        KeyStoreHelper keyStoreHelper = new
        KeyStoreHelper(keyStoreFile,
                      storePass, keyPass, alias, dname);
        boolean ok = keyStoreHelper.exportCert(dir);
        log.debug("Export Cert ok = " + ok);
    }
    ...
}
```

运行以上测试用例，会在工程目录下创建一个server_cert.cer的数字证书。使用文本工具可以打开该证书文件，其内容如图12-24所示。

```

-----BEGIN CERTIFICATE-----
MIIDUTCCAjmgIBAgIBATANBgkqhkiG9w0BAQsFADBsMQ0wCwYDVQQDEwR1c2VyMRcwFQYDVQQL
Ew5jcmF6eW1ha2VyLmNvbTETMBEGA1UEChMKY3JhenlyWt1cjENMASGA1UEBxMEY210eTERMA8G
A1UECBMlUHJvdmluY2UxCzAJBgNVBAYTAkNOMB4XDThMDUwNTA3NDczOfoXDTwMDUwNTA3NDcz
OFowbDENMasGA1UEAxMEdXNlcjEXMBUGA1UECxMOY3JhenlyWt1ci5jb20xEzARBgNVBAoTCmNy
YXp5bWRzXIXdTALBgNVBAcTBGNpdHkxETAPBgNVBAgTCFBByb3ZpbmNlMjswCQYDVQQGEwJDTjCC
ASIwDQYJKoZIhvCNQEBBQADggEPADCCAQoCggEBAK3dp9/+TBW3+W6sSYoKvmAk9kEcVMd5P/k
X5ONneR5Wez7ywkvXv4pKzcsdBiaBsti5GeA/VLvmBEuHSnKvxoxDZArAIgnpFJTY0HcZThZ1Tn5M
CoiqxegkfyzASXCDQRyJkBwlwMi053Cc7Y3gm74ZzVw37UwhbS3f+3sxXGvnn0KjVHaGF9YeZuSA
QnYSR2kiHOf225GizdnErJtf+fshKAMj/qEqaUE1o5c370JEyeWAit/UVyjhzzwscwnHNYq/jKcn8
73Jqo4Fh4cZ47TsTRf/+M46RT13nVC0k4a8u+ryPcBsLu4M+/AaOemQCoNlpSsftTy4Esfxr+kN
Zc0CawEAATANBgkqhkiG9w0BAQsFAAOCAQEARG+emj9HmzKdMSDP1n7Zuz19LNd02Ta49CWUekA4
eU7sHva41vNugRx1v6QAPH4L2hs1ISO/jYMDpiRvwJnEMvd7EAYUmrijqaNMZ5kL1EvplIsZ3v6An
qkKjyhJnMQ2syBnNX/tNmHqAgtcM6xRlUK4bCism95Nx61JatbikwYn7HEg1SoXxPMGNpg1SBkVa
puj4dqR2hY9HCQlJ9fouWWs3sSwte7O9UnrBnvBFUg2jFxq/uJ9W+3S6c0zr9ff2msfeJilcrTNn
rhjUbdiw05Cme1+0g1kPoqPQ7UtEeaXvaM516d7frmzA5o28WCTt04EDB8gdNaKYGmC0jmflA==

-----END CERTIFICATE-----

```

图12-24 BASE64编码后的数字证书内容

12.5.4 使用Java程序将数字证书导入信任仓库

如果需要将服务的证书导入到信任仓库（如客户端仓库），可以使用Keytool工具的import选项来完成，具体命令如下：

```
keytool -import -trustcacerts -alias server -file server.cer -
keystore f:/client.jks -storepass 123456
```

还是在KeyStoreHelper类中使用Java实现导入数字证书到信任仓库的代码，其方法的名称为importCert，代码如下：

```

package com.crazymakercircle.keystore;
//省略import

public class KeyStoreHelper
{
    //省略成员属性
    /**
     * 导入数字证书到信任仓库
    
```

```
*/  
  
public void importCert(String importAlias, String certPath)  
...{  
    if (null == keyStore)  
    {  
        keyStore = loadStore();  
    }  
  
    InputStream inStream = null;  
    if (certPath != null)  
    {  
        inStream = new FileInputStream(certPath);  
    }  
  
    //将证书按照别名增加到仓库中  
    boolean succeed = addTrustedCert(importAlias,  
inStream);  
    if (succeed)  
    {  
        log.debug("导入成功");  
    } else  
    {  
        log.error("导入失败");  
    }  
}  
  
/**  
 * 将证书按照别名增加到仓库中  
 */
```

```
private boolean addTrustedCert(String alias, InputStream
in)
throws Exception
{
if (alias == null)
{
throw new Exception("Must.specify.alias");
}
//如果别名已经存在，则抛出异常
if (keyStore.containsAlias(alias))
{
throw new Exception("别名已经存在");
}

//从输入流中读取到证书
X509Certificate cert = null;
try
{
cert = (X509Certificate) generateCertificate(in);
} catch (ClassCastException | CertificateException ce)
{
throw new Exception("证书读取失败");
}
//根据别名进行设置
keyStore.setCertificateEntry(alias, cert);
//写回到仓库文件
char[] caPasswordArray = storePass.toCharArray();
```

```

        java.io.FileOutputStream fos = null;
        try
        {
            fos = new java.io.FileOutputStream(keyStoreFile);
            keyStore.store(fos, caPasswordArray);
        } finally
        {
            closeQuietly(fos);
        }
        return true;
    }

    ...
}

```

调用KeyStoreHelper类的importCert()方法把创建服务端的数字证书导入到客户端的密钥仓库。接下来进行一下自测，其测试用例代码如下：

```

package com.crazymakercircle.keystore;
//省略import
/**
 * 客户端密钥仓库测试类
 */
@Slf4j
@Data
public class ClientKeyStoreTester
{

```

```
//省略成员属性

/**
 * 在客户端仓库导入服务器的证书
 */

@Test
public void testImportServerCert() throws Exception
{
    String dir = SystemConfig.getKeystoreDir();
    log.debug(" client dir = " + dir);
    KeyStoreHelper keyStoreHelper = new KeyStoreHelper(
        keyStoreFile, storePass, keyPass, alias,
        dname);
    /**
     * 服务器证书的文件
     */
    String importAlias = "server_cert";
    String certPath = SystemConfig.getKeystoreDir() +
        "/" + importAlias +
        ".cer";
    //导入服务器证书
    keyStoreHelper.importCert(importAlias, certPath);
}

/**
 * 在客户端仓库打印仓库的所有证书
 */

@Test
public void testPrintEntries() throws Exception
```

```
{  
    String dir = SystemConfig.getKeystoreDir();  
    log.debug(" client dir = " + dir);  
    KeyStoreHelper keyStoreHelper = new KeyStoreHelper(  
        keyStoreFile, storePass, keyPass, alias,  
        dname);  
    //打印仓库的所有证书  
    keyStoreHelper.doPrintEntries();  
}  
  
...  
}
```

运行以上测试用例之前，要先创建客户端的密钥仓库才能完成服务器证书的导入。运行第一个测试用例，会给客户端仓库导入服务器证书。运行第二个测试用例，可以打印客户端仓库的所有证书，输出结果如下：

```
KeyStoreHelper - keyStoreFile: F:/ .../client.jks  
KeyStoreHelper - client_cert 别名的证书信息如下:  
KeyStoreHelper - Owner: C=CN, ST=Province, L=city,  
O=crazymaker, OU=crazymaker.com, CN=client  
KeyStoreHelper - Issuer: C=CN, ST=Province, L=city,  
O=crazymaker, OU=crazymaker.com, CN=client  
KeyStoreHelper - Serial number: 1  
KeyStoreHelper - Valid from: Sat May 23 21:58:00 CST 2020  
KeyStoreHelper - Valid until: Sun May 23 21:58:00 CST 2021  
KeyStoreHelper - Certificate fingerprints SHA1:
```

```
KeyStoreHelper -  
B9:83:6A:75:F6:B5:4B:28:BA:0B:DE:15:CF:6D:33:A5:A9:9E:2A:DC  
KeyStoreHelper - Certificate fingerprints SHA256:  
KeyStoreHelper -  
BF:24:49:2E:52:71:79:30:EA:A8:C6:68:79:13:FC:90:63:88:7E:0D:9A:  
C0:9E:81:C7:F0:D3:66:2C:4C:82:28  
KeyStoreHelper - Signature algorithm name: SHA256withRSA  
KeyStoreHelper - Version: 3  
KeyStoreHelper - server_cert 别名的证书信息如下:  
KeyStoreHelper - Owner: C=CN, ST=Province, L=city,  
O=crazymaker, OU=crazymaker.com, CN=server  
KeyStoreHelper - Issuer: C=CN, ST=Province, L=city,  
O=crazymaker, OU=crazymaker.com, CN=server  
KeyStoreHelper - Serial number: 1  
KeyStoreHelper - Valid from: Sat May 23 21:21:18 CST 2020  
KeyStoreHelper - Valid until: Sun May 23 21:21:18 CST 2021  
KeyStoreHelper - Certificate fingerprints SHA1:  
KeyStoreHelper -  
2C:5B:D7:64:4C:70:E6:36:1F:4C:A0:7E:24:05:60:4E:EB:6D:8C:D8  
KeyStoreHelper - Certificate fingerprints SHA256:  
KeyStoreHelper -  
84:BB:D7:52:43:19:42:AD:29:D3:0C:B3:A3:A1:53:E9:68:73:80:54:F3:  
82:18:1F:9D:E5:40:1E:9A:2C:9F:7A  
KeyStoreHelper - Signature algorithm name: SHA256withRSA  
KeyStoreHelper - Version: 3
```

从客户端密钥仓的证书清单中可以看到两个数字证书，其中包括通过importCert()方法导入的服务器证书（别名为server_cert）和自己的数字证书（别名为client_cert）。

12.6 OIO通信中的SSL/TLS使用实战

SSL/TLS协议是安全的通信模式。对于这些底层协议，如果要每个开发者都自己去实现显然会带来不必要的麻烦。为了解决这个问题，Java为广大开发者提供了Java安全套接字扩展（Java Secure Socket Extension, JSSE），包含了实现Internet安全通信的一系列包的集合，是SSL和TLS的纯Java实现，同时它是一个开放的标准，每个公司都可以自己实现JSSE，通过它可以透明地提供数据加密、服务器认证、信息完整性等功能，就像使用普通的套接字一样使用安全套接字，大大减轻了开发者的负担，使开发者可以很轻松地将SSL协议整合到程序中，并且JSSE能将安全隐患降到最低点。

JSSE扩展包括数据加密、服务器数字证书管理、消息完整性以及可选的客户数字证书管理等功能。借助于JSSE，开发者能够快速完成应用层协议（如Http、Telnet、FTP）的安全数据通道。

在JSSE的使用过程中，SSL/TSL通信握手过程中的日志是可以打印出来的。在实际编写程序的时候可能会在这些环节遇到问题，导致无法通信，排查起来往往令人无从下手。这时我们可以将SSL/TSL通信的握手日志开关打开进行观察，打开该开关的Java命令选项为：

```
-Djavax.net.debug=ssl,handshake
```

当然也可以调用System.setProperty()方法在代码中打开该开关。打开日志开关后，可以搜索“ClientHello”“ServerHello”等关键字，通过阅读日志来定位SSL/TSL通信问题。更详细的开关信息可以使用下面的Java命令选项设置开启：

```
java -Djavax.net.debug=SSL,handshake,data,trustmanager
```

如果使用集成开发工具（如IDEA），就可以将该参数加入VM options选项中，具体如图12-25所示。

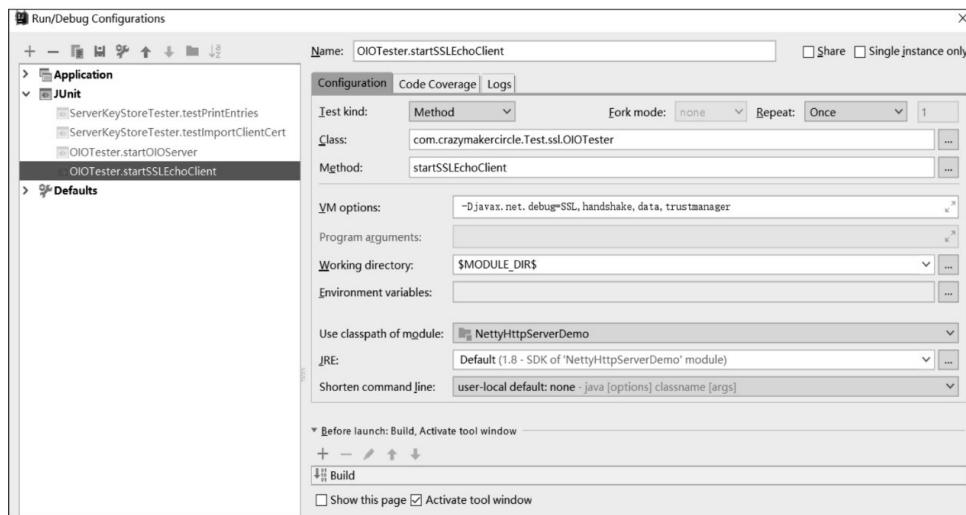


图12-25 在IDEA中加入VM options选项 javax.net.debug

12.6.1 JSSE安全套接字扩展核心类

在用JSSE实现SSL/TLS的通信过程中会涉及加解密、密钥生成等安全运算的框架和实现，所以也会间接用到JCE（Java Cryptography Extension）包的一些类（如加密、解密等）。

JSSE安全通信的核心类主要如表12-4所示。

表12-4 JSSE安全通信库的包含核心类

核心类	说 明
SSLSocket	安全通信核心类, 对应于 TCP 通信的传输套接字 ServerSocket 类。该类是 Socket 类的子类, 表示一种实现了 SSL 协议的子类。SSLSocket 负责设置加密套件、管理 SSL 会话、处理握手结束时间、设置客户端模式或服务端模式
SSLServerSocket	安全通信核心类, 对应于 TCP 通信的服务端监听套接字 ServerSocket 类, SSLServerSocket 是实现了 SSL 协议的监听套接字, 是 ServerSocket 类的子类。SSLServerSocket 的主要职责是通过接收连接来创建 SSLSocket。SSLServerSocket 包含了一些状态参数, 包括启用的密钥套件和协议、客户端验证是否必需, 以及所创建的 SSLSocket 是以客户端模式还是以服务端模式开始握手等, 这些状态参数在创建传输套接字 SSLSocket 实例时由新的套接字所继承
SSLSocketFactory 和 SSLServerSocketFactory	这是一组客户端与服务端套接字工厂类。这两个工厂类分别用于创建安全传输套接字 SSLSocket 实例、安全监听套接字 SSLServerSocket 实例
SSLSession	SSL 安全套字的通信会话。为了提高通信的效率, SSL 协议允许多个 SSLSocket 共享同一个 SSL 会话, 在同一个会话中, 由第一个 SSLSocket 负责 SSL 握手、生成密钥及交换密钥, 其余 SSLSocket 都共享密钥信息

(续表)

核心类	说 明
SSLEngine	SSL 安全传输引擎。SSLEngine 从底层的 I/O 传输机制中分离出了 SSL/TLS 抽象安全操作, 并且将 SSL/TLS 安全机制应用在入站和出站的字节流上, 使之与底层的传输机制无关, 所以 SSLEngine 传输引擎可以被用于各种 I/O 类型, 包括 NIO、OIO、Input/Output Streams、本地 ByteBuffers 缓冲区或字节数组、未来的异步 I/O 模型等
SSLContext	SSL 安全套接字的上下文类。此类作为安全套接字协议的重要实现类, 该类的实例负责创建 SSLSocketFactory、SSLServerSocketFactory 和 SSLEngine 三大重要工厂的实例。SSLContext 还负责设置和管理安全通信过程中的各种信息, 例如跟密钥仓库、证书相关的信息
KeyManager	密钥管理器。此接口的实例负责管理用于证实自己身份的安全证书和公钥, 并在握手时用于发给通信对端。如果没有密钥仓库可以使用, 则套接字将不能提供安全证书供对方验证。在 JSSE 通信的握手程序中, 一般服务端需要发送数字证书给客户端进行身份认证, 此时可通过 KeyManager 实例从其管理的 keyStore 密钥仓库中获取自己的数字证书。反过来, 如果握手过程中需要客户端发送安全证书给服务端, 也需要通过 KeyManager 实例从客户端的 keyStore 密钥仓库获取含有公钥的数字证书。KeyManager 实例通过 KeyManagerFactory 工厂类实例创建
TrustManager	信任管理器 TrustManager 负责管理受到自己信任的数字证书。在对端证书发送过来时, JSSE 将通过 TrustManager 获取到自己管理的外部信任证书, 然后完成对端证书的信任校验。TrustManager 实例由 TrustManagerFactory 工厂类生成

使用上面这些核心类, 基本就可以完成 Java 的 SSL/TLS 的安全通信了。在进行安全通信时, 客户端跟服务端都必须支持 SSL/TLS 协议, 不然将无法进行通信。客户端和服务端都可能要设置用于证实自己身份的安全证书, 并且还要设置信任对方的哪些安全证书。

发起握手请求时，有个名词叫客户端模式。一般情况下，由处于客户端模式的一方发起SSL/TSL的握手报文Client Hello。使用传输套接字SSLSocket的setUseClientMode(Boolean mode)方法可以设置本端处于客户端模式还是处于服务端模式。

说明

这里的客户端模式是SSL/TSL的专用概念，表示由这一方发起Client Hello握手请求，通信双方只能有一方为客户端模式。如果一方设置为客户端模式，则另一方不能设置为客户端模式。

在JSSE中，可以通过设置SSLSocket.setNeedClientAuth(true)来启用是否需要对对方认证，以便要求对方提供数字证书。如果设置了true，并且对方选择不提供其自身的验证信息，则协商将会停止，SSLEngine引擎将开始SSLSocket的关闭过程。

12.6.2 JSSE安全套接字的创建过程

JSSE扩展中提供了javax.net.ssl.SSLSocket和javax.net.ssl.SSLServerSocket安全套接字类用于支持在阻塞式OIO模式套接字的安全传输。下面先介绍一下安全套接字的创建过程。

(1) 第一步：加载本地的密钥仓库，创建KeyManager密钥管理器。安全套接字在握手过程中可能需要发送代表自己的数字证书，该

证书来自本地密钥仓，并且在验证对方的数字证书时也需要用本地的信任证书仓（一般也用本地密钥仓）查找对应的信任证书。

加载本地的密钥仓库需要用到仓库密码和仓库文件，大致代码如下：

```
//仓库密钥  
String pass = "123456";  
String keyStoreFile = SystemConfig.getKeystoreDir() +  
"/server.jks";  
char[] passArray = pass.toCharArray();  
KeyStore keyStore = KeyStore.getInstance("JKS");  
//加载 keyStoreFile, 生成的密钥仓库  
FileInputStream inputStream = new  
FileInputStream(keyStoreFile);  
keyStore.load(inputStream, passArray);  
  
//创建密钥管理器，并且初始化  
String algorithm = KeyManagerFactory.getDefaultAlgorithm();  
KeyManagerFactory kmf =  
KeyManagerFactory.getInstance(algorithm);  
kmf.init(keyStore, passArray);
```

一般而言，服务端必须要有证书以证明身份，并且证书应该描述此服务器所有者的一些基本信息，例如公司名称、联系人名等。JSSE 中密钥实体在创建的时候可能会有两类数字证书：自我签名的数字证书和CA机构签名的数字证书。自我签名的数字证书在创建密钥时已经保存在密钥仓中。

如果通过CA机构购买数字证书，数字签名由CA机构完成，所购买的证书也需要在服务端进行配置，加入服务端的密钥仓Java KeyStore中。

数字证书中的数字签名是证书摘要信息用签名机构的私钥加密后的数据，这条数据可以用签名机构的公共钥匙解密，这里用到的核心算法是非对称加密算法。

对于客户端，如果服务端使用的是自我签名数字证书而不是CA机构的数字证书，则需要将该证书导入客户端的信任仓（TrustStore）中。理论上TrustStore是由TrustManager进行管理的，在验证对端的数字证书时，需要获取本地信任仓TrustStore中的信任证书，进行证书比对和校验。JSSE程序为了方便，可以将本地TrustStore和本地KeyStore合二为一，所以可以直接从本地的KeyStore获取对方的数字证书。当然，在认证之前，也需提前将信任的数字证书导入KeyStore。

理论上，KeyManager密钥管理器负责管理本地KeyStore密钥仓，TrustManager信任管理器负责管理本地TrustStore信任证书仓。但是，对于信任管理器，在具体开发中应用程序会编写一个自己的实现类（如X509TrustManagerFacade），直接从本地KeyStore密钥仓中获取导入的信任数字证书。

(2) 第二步：创建SSL上下文实例。创建SSLContext上下文实例时会用到KeyManager密钥管理器和TrustManager信任管理器实例，其代码大致如下：

```
//初始化KeyManagerFactory之后，创建SSLContext并初始化  
SSLContext sslContext =  
SSLContext.getInstance("SSL");  
  
//信任库  
  
//如果是单向认证，服务端不需要验证客户端的合法性，此时  
TrustManager 可以为空  
  
TrustManager[] trustManagers =  
createTrustManagers(keyStore);  
  
//安全随机数不需要设置  
  
sslContext.init(kmf.getKeyManagers(), trustManagers,  
null);
```

(3) 第三步：创建安全套接字。通过SSL上下文实例完成客户端SSLSocketFactory传输套接字工厂、服务端SSLServerSocketFactory监听套接字工厂实例创建，再由两大工厂实例进一步创建安全套接字。服务端的监听套接字的创建代码如下：

```
//创建服务端SSL上下文实例  
SSLContext serverSSLContext =  
createServerSSLContext();  
  
SSLServerSocketFactory sslServerSocketFactory =  
  
serverSSLContext.getServerSocketFactory();  
  
//通过服务端SSL上下文实例创建服务端SSL监听套接字  
serverSocket = (SSLServerSocket)
```

```
sslServerSocketFactory.createServerSocket(SOCKET_SERVER_PORT);
```

创建完安全套接字之后，就可以基于SSL/TLS进行安全传输了。

12.6.3 OIO安全通信的Echo服务端实战

一个简单的基于OIO（Java阻塞式I/O）进行安全通信的Echo演示实战案例的服务端的主要代码如下：

```
package com.crazymakercircle.secure.nio;  
  
//省略import  
  
//服务端  
  
public class SSLEchoServer  
{  
  
    //服务端SSL监听套接字  
  
    static SSLServerSocket serverSocket;  
  
    public static void start()  
    {  
  
        try  
        {  
  
            //创建服务端SSL上下文实例  
  
            SSLContext serverSSLContext =  
                createServerSSLContext();  
  
            SSLServerSocketFactory sslServerSocketFactory =  
  
                serverSSLContext.getServerSocketFactory();  
        }
```

```
//通过服务端SSL上下文实例创建服务端SSL监听套接字
serverSocket = (SSLServerSocket)
sslServerSocketFactory.createServerSocket(SOCKET_SERVER_PORT);

//单向认证：在服务端设置不需要验证对端身份，不需要客户端证实自己的数字证书
serverSocket.setNeedClientAuth(true);
//在握手的时候，使用服务端模式，由客户端发起Client Hello帧
serverSocket.setUseClientMode(false);

String[] supported =
serverSocket.getEnabledCipherSuites();
serverSocket.setEnabledCipherSuites(supported);
log.info("SSL OIO ECHO 服务已经启动 {}:{}",
SystemConfig.SOCKET_SERVER_NAME,
SystemConfig.SOCKET_SERVER_PORT);

//监听和接收客户端连接
while (!Thread.interrupted())
{
    Socket client = serverSocket.accept();

System.out.println(client.getRemoteSocketAddress());
//向客户端发送接收到的字节序列
OutputStream output = client.getOutputStream();
//当一个普通socket连接上来，这里会抛出异常
InputStream input = client.getInputStream();
```

```
byte[] buf = new byte[1024];
int len = 0;
StringBuffer buffer = new StringBuffer();
while ((len = input.read(buf)) != -1)
{
    String sf = new String(buf, 0, len, "UTF-
8");
    log.info("服务端收到: {}", sf);
    buffer.append(sf);
    if (sf.contains("\r\n\r\n"))
    {
        break;
    }
}
//发送消息到客户端
output.write(buffer.toString().getBytes("UTF-
8"));
output.flush();
//关闭socket连接
closeQuietly(input);
closeQuietly(output);
closeQuietly(client);
}
} catch (Exception e)
{
    e.printStackTrace();
} finally
```

```
    { //关闭serverSocket监听套接字  
        closeQuietly(serverSocket);  
    }  
}  
}
```

服务端设置为单向认证模型：只是客户端对服务端进行认证，服务端不需要认证客户端。也就是说，在服务端设置不需要验证对端身份，不需要客户端提供自己的数字证书，关键代码如下：

```
serverSocket.setNeedClientAuth(true);
```

12.6.4 OIO安全通信的Echo客户端实战

一个简单的基于OIO（Java阻塞式I/O）进行安全通信的Echo演示实战案例的客户端的主要代码如下：

```
package com.crazymakercircle.secure.nio;  
  
//省略import  
  
//客户端  
  
@Slf4j  
  
public class SSLEchoClient  
{  
    //安全套接字  
    static SSLSocket sslSocket;  
    static OutputStream output;  
    static InputStream input;
```

```
public static void connect()
{
    try
    {
        //创建客户端SSL 上下文
        SSLContext clientSSLContext =
createClientSSLContext();

        SSLSocketFactory factory =
clientSSLContext.getSocketFactory();

        sslSocket =
factory.createSocket("localhost", 18899);

        //在握手的时候，使用客户端模式，由客户端发起
        Client Hello帧

        sslSocket.setUseClientMode(true);

        //单向认证：设置需要验证对端身份，这里需验证服
务端身份

        sslSocket.setNeedClientAuth(true);

        log.info("连接服务器成功");

        output = sslSocket.getOutputStream();
        input = sslSocket.getInputStream();

output.write("hello\r\n\r\n".getBytes());
        output.flush();

        log.info("sent hello finished!");

        byte[] buf = new byte[1024];
        int len = 0;
```

```
        while ((len = input.read(buf)) != -1)

        {
            log.info("客户端收到: {}", new
String(buf, 0, len, "UTF-8"));

        }

    } catch (Exception e)
{
    e.printStackTrace();
} finally
{
    closeQuietly(output);
    closeQuietly(input);
    closeQuietly(sslSocket);
}

}
```

客户端也设置为单向认证模型：客户端对服务端进行认证，服务端不需要认证客户端。也就是说，在客户端设置需要验证对端身份，需要服务端提供自己的数字证书，关键代码如下：

```
//单向认证：设置需要验证对端身份，这里需验证服务端身份  
sslSocket.setNeedClientAuth(true);
```

12.7 单向认证与双向认证

单向认证和双向认证的具体含义如下：

(1) SSL/TLS单向认证：客户端会认证服务端身份，服务端不对客户端进行认证。

(2) SSL/TLS双向认证：客户端和服务端都会互相认证，即双方之间都要发送数字证书给对端，并且对证书进行安全认证。

12.7.1 SSL/TLS单向认证

SSL/TLS单向认证就是用户到服务器之间只存在单方面的认证，即客户端会认证服务端身份，而服务端对客户端身份进行验证。

前面所介绍的SSL/TSL协议的握手过程的四个阶段是以单向认证的握手流程为蓝本进行介绍的。第一个阶段，客户端发起握手请求；第二个阶段，服务器收到握手请求后会选择适合双方的协议版本和密钥套件，然后将协商的结果和服务端的证书（含公钥）一起发送给客户端；第三个阶段，客户端利用服务端的公钥对要发送的数据（主要是第三个随机数）进行加密，并发送给服务端；第四个阶段，服务端收到第三个随机数后，会用本地私钥对收到的客户端加密数据进行验证，验证通过后计算会话密钥，然后给客户端进行最后的回复确认。完成握手之后，通信双方都会使用生成的会话密钥，就可以开始安全通信过程了。

单向认证场景下服务端serverSocket的设置需要调用setNeedClientAuth()成员方法，参数为false，表示不要求客户端发送其数字证书，在服务端不进行客户端的数字证书校验。核心代码如下：

```
public class SSLEchoServer
{
    ...
    //通过服务端SSL上下文实例创建服务端SSL监听套接字
    serverSocket
    =sfactory.createServerSocket(SOCKET_SERVER_PORT);

    ...
    //单向认证：不需要客户端证实自己的身份，无须验证客户端身份
    serverSocket.setNeedClientAuth(false);
    //在握手的时候使用服务端模式
    serverSocket.setUseClientMode(false);
    ...
}

}
```

单向认证场景下，客户端socket需要调用setNeedClientAuth()成员方法，不过这里的参数为true，表示需要服务端发送数字证书，客户端也会对服务端进行证书的校验。其核心代码如下：

```
public class SSLEchoClient
{
    ...
    //创建客户端SSL 上下文
```

```
SSLContext clientSSLContext =  
createClientSSLContext();  
  
SSLocketFactory cfactory =  
clientSSLContext.getSocketFactory();  
  
sslSocket = cfactory.createSocket("localhost",  
SOCKET_SERVER_PORT);  
  
//在握手的时候使用客户端模式  
sslSocket.setUseClientMode(true);  
  
//设置需要验证对端身份，需验证服务端身份  
sslSocket.setNeedClientAuth(true);  
  
...  
}
```

在上面的代码中，`setNeedClientAuth()`方法用于设置是否需要为对端进行身份证证书的认证。但是，还存在一个与其名字比较类似的`setUseClientMode()`方法，这两个方法比较容易混淆，下面对后者做一个简单的介绍。

`setUseClientMode()`方法用于设置安全通信双方是处于客户端模式还是服务端模式。每个SSL/TLS连接的两端都必须有一个角色，或者是客户端角色，或者是服务端角色，因此每一端必须决定担任哪种角色。在SSL/TLS握手过程中，由客户端角色负责开始握手的流程，发送Client Hello帧，后续每个角色都有各自明确的握手报文。客户端模式或服务端模式的设置就决定了由谁开始握手过程，以及应该发送哪种类型的报文。

对于同一个SSL/TLS连接的通信双方来说，只能有一方处于“服务端模式”，另一方必须处于“客户端模式”。实质上，无论是客户方

还是服务器方，都可处于“客户端模式”或者“服务端模式”。

通常情况下，实际的客户端程序会调用setUseClientMode(true)将自己设置为“客户端模式”，而实际的服务器程序会调用setUseClientMode(false)将自己设置为“服务端模式”。所以，一般由客户端程序发送Client Hello帧。

单向认证场景下，只有服务端在第二阶段握手时发送其数字证书给客户端，客户端通过其信任管理器进行服务端证书的验证。其握手流程具体如图12-26所示。

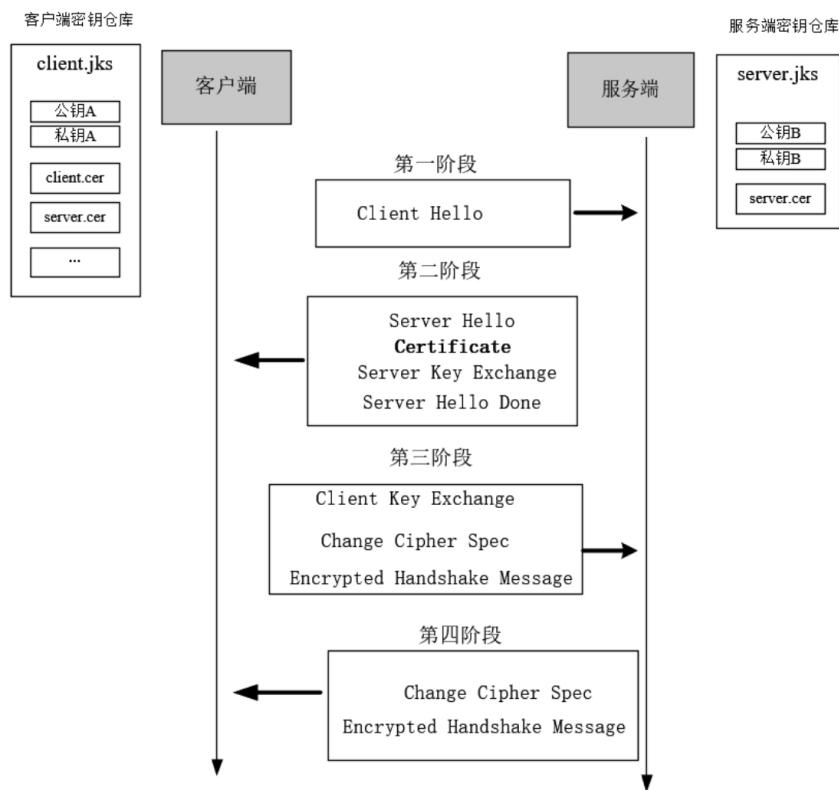


图12-26 服务端在握手的第二阶段发送其数字证书给客户端

以上流程可以在OIO安全通信实例的双方通信过程中通过WireShark工具抓取报文予以验证。抓包工具WireShark是通过监控网

卡（网络接口）抓包的，只能抓取经过网卡的包，开发场景下本机客户端发往本机地址（localhost、127.0.0.1）的调试包并没有经过网卡，所以WireShark工具监控不到，也就是说默认情况下WireShark抓不到本地的调试报文。如果要抓包，需要通过route指令增加本地路由表中的路由配置，让发往本机的IP报文路由到监控网卡所指向的网关，这样发送到本机的报文才能被WireShark拦截到。假设本机调试的网卡地址为192.168.0.5，所以为该IP增加路由项目的指令如下：

```
//route add 添加路由项目到本地路由表  
route add 192.168.0.5 mask 255.255.255.255 192.168.0.1
```

启动OIO安全传输的服务端与客户端演示实例，然后在抓包工具WireShark上可以看到双方的SSL/TLS单向认证的握手过程的交互报文，大致如图12-27所示。

The screenshot shows a Wireshark capture window titled '*WLAN'. The packet list pane displays several TLSv1.2 handshake messages. A search filter '(ip.src == 192.168.0.5) && (ip.dst == 192.168.0.5)' is applied. The columns in the table are No., Source, Destination, Protocol, and Info. The captured packets are:

No.	Source	Destination	Protocol	Info
37	192.168.0.5	192.168.0.5	TLSv1.2	Client Hello
39	192.168.0.5	192.168.0.5	TLSv1.2	Server Hello, Certificate, Server Key Exchange, Server Hello Done
41	192.168.0.5	192.168.0.5	TLSv1.2	Client Key Exchange
45	192.168.0.5	192.168.0.5	TLSv1.2	Change Cipher Spec, Encrypted Handshake Message
47	192.168.0.5	192.168.0.5	TLSv1.2	Change Cipher Spec
52	192.168.0.5	192.168.0.5	TLSv1.2	Encrypted Handshake Message
54	192.168.0.5	192.168.0.5	TLSv1.2	Application Data
56	192.168.0.5	192.168.0.5	TLSv1.2	Application Data

图12-27 SSL/TLS单向认证的握手过程的交互报文

这里有个问题，为什么使用WireShark而不能使用Fiddler作为抓包工具呢？原因是，Fiddler是用于Web开发过程中的应用层抓包工具，只能抓取应用层（如HTTP）的报文，抓取不到传输层或者IP层的报文，而WireShark则可以抓取到传输层报文。虽然Fiddler使用简单方便，但是在这里只能使用WireShark。

在自签名证书认证（非CA机构颁发的证书认证）场景下的单向认证必须在客户端密钥仓库导入服务端的数字证书，然后客户端还要能通过TrustManager信任管理器读取到本地仓库信任的数字证书。接下来为大家介绍证书信任管理器的使用。

12.7.2 使用证书信任管理器

在JSSE的核心类中，TrustManager接口用于对信任证书进行管理，负责管理受到自己信任的数字证书。握手过程中，在对端证书发送过来时，JSSE将通过TrustManager获取到自己管理的证书，然后完成对端证书的校验。在进行对端证书校验时，如果对方的证书不在信任库中，则校验会失败。

JSSE的核心信任证书管理器接口叫作X509TrustManager接口。我们可以自己实现该接口，主要包括以下三个抽象方法：

```
(1) void checkClientTrusted(X509Certificate[] chain,  
String authType)
```

该方法检查客户端的证书，若不信任该证书则抛出异常。在单向认证场景中，由于不需要对客户端进行认证，因此我们只需要执行默认的信任管理器TrustManager的方法实现接口即可，其默认实现是什么也不做，不对证书进行检查。

```
(2) void checkServerTrusted(X509Certificate[] chain,  
String authType)
```

该方法用于检查服务端的证书，若不信任该证书则抛出异常。通过自己实现该方法，可以使之信任我们指定的任何证书。如果不需要验证服务端的数字证书，也可以不做任何处理，即只有一个空的函数体，也就不会抛出异常，它会信任任何证书。

(3) X509Certificate[] getAcceptedIssuers()

该方法返回受信任的X509证书数组，一般用于返回本地仓库受信任的数字证书。

在JSSE编程过程中，如果希望自定义信任库管理器的一些行为，如需要从本地密钥仓库加载信任证书、自定义检验对方证书等，可以实现X509TrustManager接口，通过定制自己的方法来实现。下面是一个简单的定制示例，通过本地密钥仓库初始化信任管理器工厂，然后获取其X509的数字证书库：

```
package com.crazymakercircle.ssl;  
//省略import  
/**  
 * 定制的信任管理器  
 */  
@Slf4j  
public final class X509TrustManagerFacade implements  
X509TrustManager  
{  
    /**  
     * 内部的x509TrustManager委托成员  
     */
```

```
private X509TrustManager x509TrustManager;

/**
 * 使用密钥仓库初始化信任管理器
 * @param keyStore 密钥仓库
 */
public void init(KeyStore keyStore) throws Exception
{
    TrustManagerFactory factory = TrustManagerFactory.getInstance(
        TrustManagerFactory.getDefaultAlgorithm());
    //使用密钥仓库初始化信任管理器工厂
    factory.init(keyStore);
    //从信任管理器工厂的信任库中筛选出X509格式的证书库
    TrustManager[] trustManagers =
factory.getTrustManagers();
    for (int i = 0; i < trustManagers.length; i++)
    {
        TrustManager trustManager = trustManagers[i];
        if (trustManager instanceof X509TrustManager)
        {
            this.x509TrustManager = (X509TrustManager)
trustManager;
        }
    }
    if (this.x509TrustManager == null)
    {

```

```
        throw new Exception("Couldn't find
X509TrustManager");
    }
}

//客户端证书检验
public final void checkClientTrusted(
    X509Certificate[] chain, String
authType) {
    log.info("checkClient {}, type is {}", chain,
authType);
    X509TrustManager x509TrustManager =
this.x509TrustManager;
    if (x509TrustManager != null)
    {
        x509TrustManager.checkClientTrusted(chain,
authType);
    }
}

//对服务端证书的校验
public final void checkServerTrusted(
    X509Certificate[] chain, String
authType){
    log.info("checkServer {}, type is {}", chain,
authType);
    if (this.x509TrustManager != null)
```

```
        {

            this.x509TrustManager.checkServerTrusted(chain,
authType);

        }

    }

//返回受信任的X509证书数组

public final X509Certificate[] getAcceptedIssuers()

{

    X509Certificate[] issuers = null;

    if (this.x509TrustManager != null)

    {

        issuers = x509TrustManager.getAcceptedIssuers();

    }

    if (null == issuers)

    {

        log.error("信任的X509证书数组 is null");

    }

    return issuers;

}

}
```

假定客户端程序要对服务端程序的证书进行校验，则需要在checkServerTrusted()方法中进行；假定服务端程序要对客户端程序的证书进行校验，就需要在checkClientTrusted()方法中进行。

定义好TrustManager实现类之后，如何使用呢？在创建SSLContext上下文实例的时候，第二个参数需要一个TrustManager数组，该数组的作用是为SSLContext上下文提供信任证书管理器。使用TrustManager实现类的代码大致如下：

```
package com.crazymakercircle.ssl;  
...  
@Slf4j  
public class SSLContextHelper  
{...  
    public static SSLContext createSslContext(  
        char[] passArray, KeyStore  
        keyStore) {...  
        String algorithm =  
            KeyManagerFactory.getDefaultAlgorithm();  
        KeyManagerFactory kmf =  
            KeyManagerFactory.getInstance(algorithm);  
        kmf.init(keyStore, passArray);  
  
        //初始化KeyManagerFactory之后，创建sslContext上下文实例并初  
始化  
        SSLContext sslContext = SSLContext.getInstance("SSL");  
        //信任库  
        X509TrustManagerFacade facade = new  
        X509TrustManagerFacade();  
        facade.init(keyStore);  
        TrustManager[] trustManagers = new TrustManager[]
```

```
{facade};  
  
    //安全随机数不需要设置  
    //如果是单向认证，不需要验证对端的合法性，trustManagers参数可以  
    //为空  
    sslContext.init(kmf.getKeyManagers(), trustManagers,  
    null);  
    return sslContext;  
}  
  
...  
}
```

12.7.3 SSL/TLS双向认证

SSL/TLS双向认证就是双方都会互相认证，也就是两者之间将会交换证书。双向认证的基本握手过程和单向认证完全一样，只是在协商阶段多了几个步骤。

在握手的第二个阶段，服务端在将协商的结果和自己的数字证书一起发送给客户端后，服务端会请求客户端的证书。在握手的第三阶段，客户端会将自己的数字证书发送给服务端，服务端则会验证客户端数字证书的合法性。

双向认证场景的第一阶段握手、第四阶段握手以及建立握手之后的加密通信过程与单向认证完全保持一致。SSL/TLS双向认证的握手流程如图12-28所示。

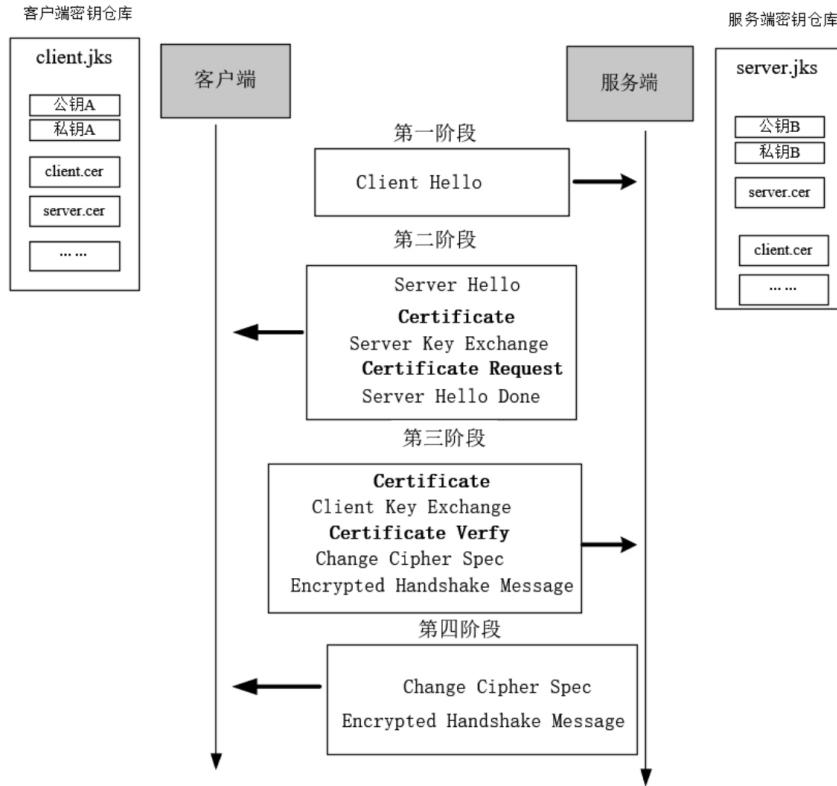


图12-28 SSL/TLS双向认证的握手流程

双向认证场景下服务端serverSocket的设置需要调用setNeedClientAuth()成员方法，参数为true，表示需要客户端发送其数字证书，并在服务端进行客户端的数字证书校验。核心代码如下：

```
public class SSLEchoServer
{
    ...
    //通过服务端SSL上下文实例创建服务端SSL监听套接字
    serverSocket = (SSLServerSocket)
    sslServerSocketFactory.createServerSocket(18899);
}
```

```
//双向认证：在服务端设置要验证对端身份，需要客户端证实自己的  
身份  
  
serverSocket.setNeedClientAuth(true);  
  
//在握手的时候，使用服务端模式  
  
serverSocket.setUseClientMode(false);  
  
...  
}
```

双向认证的客户端设置与单向认证场景下的客户端设置是相同的。客户端socket需要调用自己的setNeedClientAuth()成员方法，参数为true，表示需要服务端发送数字证书，也会对服务端进行证书的校验。客户端的核心代码如下：

```
public class SSLEchoClient  
{  
  
    ...  
  
    //创建客户端SSL 上下文  
  
    SSLContext clientSSLContext =  
    createClientSSLContext();  
  
    SSLSocketFactory factory =  
    clientSSLContext.getSocketFactory();  
  
    sslSocket  
    =factory.createSocket("192.168.0.5", 18899);  
  
    //在握手的时候，使用客户端模式  
  
    sslSocket.setUseClientMode(true);  
  
    //设置需要验证对端身份，需验证服务端身份  
  
    sslSocket.setNeedClientAuth(true);
```

```
...  
}
```

在单向认证的场景下，仅仅需要将服务端的数字证书导入客户端的密钥库（或者信任库）。在双向认证之前，还需要在服务端密钥库（或者信任库）导入客户端的数字证书，具体如图12-29所示。

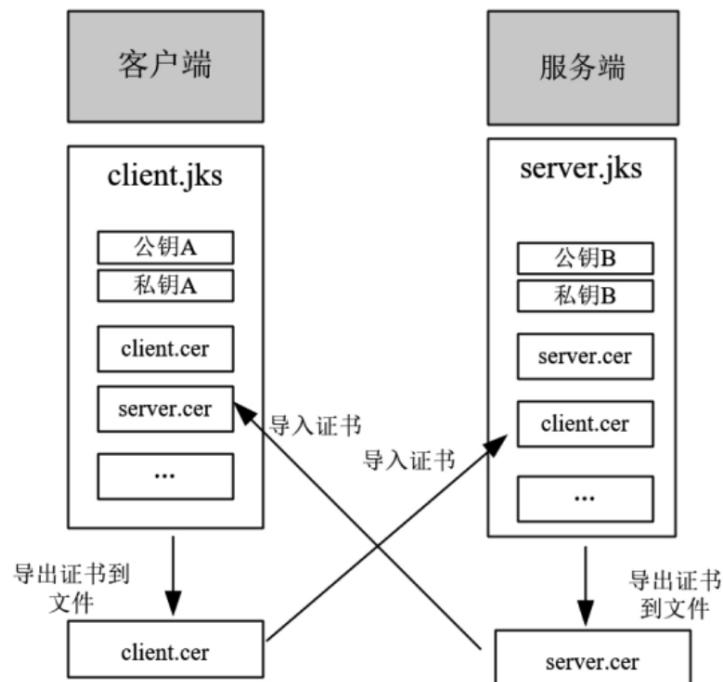


图12-29 双向认证场景下的数字证书导入关系

可以通过Java代码或Keytool工具先从客户端的密钥仓库中导出客户端的数字证书，具体代码如下：

```
/**  
 * 客户端密钥仓库操作的测试用例  
 * create by 尼恩 @ 疯狂创客圈  
 */
```

```
@Slf4j
public class ClientKeyStoreTester
{
    /**
     * 导出客户端数字证书
     */
    @Test
    public void testExportCert() throws Exception
    {
        String dir = SystemConfig.getKeystoreDir();
        log.debug(" client dir = " + dir);
        KeyStoreHelper keyStoreHelper = new
        KeyStoreHelper(keyStoreFile,
                      storePass, keyPass, alias, dname);
        boolean ok = keyStoreHelper.exportCert(dir);
        log.debug(" client ExportCert ok = " + ok);
    }
    ...
}
```

然后可以通过Java代码或Keytool工具将客户端数字证书导入服务端的密钥仓库或者信任仓库。该Java导入用例具体如下：

```
/**
 * 服务端密钥仓库操作的测试用例
 * create by 尼恩 @ 疯狂创客圈
**/
```

```
@Slf4j
public class ServerKeyStoreTester
{
    /**
     * 在服务器密钥仓库导入客户端证书
     */
    @Test
    public void testImportClientCert() throws Exception
    {
        String dir = SystemConfig.getKeystoreDir();
        log.debug(" server dir = " + dir);
        KeyStoreHelper keyStoreHelper = new KeyStoreHelper(
                keyStoreFile, storePass, keyPass, alias,
                dname);
        /**
         * 服务器证书的文件
         */
        String importAlias = "client_cert";
        String certPath = SystemConfig.getKeystoreDir() +
                "/" + importAlias +
                ".cer";
        //导入服务器证书
        keyStoreHelper.importCert(importAlias, certPath);
    }
}
```

在以上准备工作都完成后，启动OIO安全传输的服务端与客户端演示实例，然后在抓包工具上可以看到双方SSL/TLS单向认证的握手过程的交互报文，如图12-30所示。

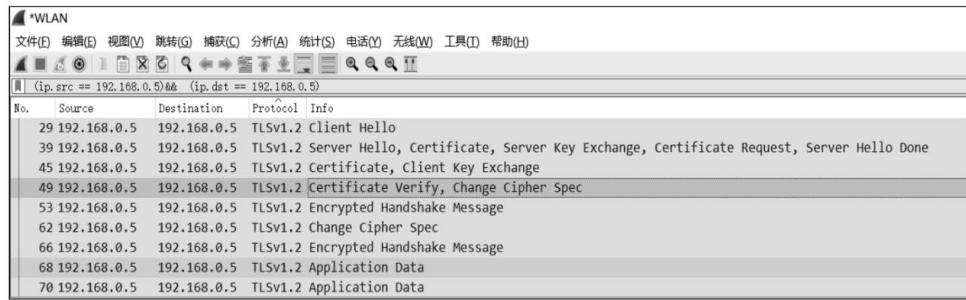


图12-30 SSL/TLS双向认证的握手过程的交互报文

JSSE提供了OIO的开发基础类，但是对于非阻塞NIO通信，JSSE并没有提供现成可用的类库去简化程序的开发。Netty基于JDK的SSLEngine基础类提供了内置处理器SslHandler，用于对NIO通信SSL/TLS安全传输予以支持。该处理器极大地简化了NIO非阻塞安全通信开发的工作量，降低了开发难度。接下来介绍一下基于Netty的SSL/TLS使用。

12.8 Netty通信中的SSL/TLS使用实战

这里通过一个Netty安全通信聊天演示实例介绍Netty安全通信处理器流水线的构成。

12.8.1 Netty安全通信演示实例

本小节的演示实例是一个简单的Netty安全聊天器，使用SSL/TLS协议进行通信加密。聊天演示实例的服务端处理器流水线的构成如图12-31所示，客户端处理器流水线的构成如图12-32所示。

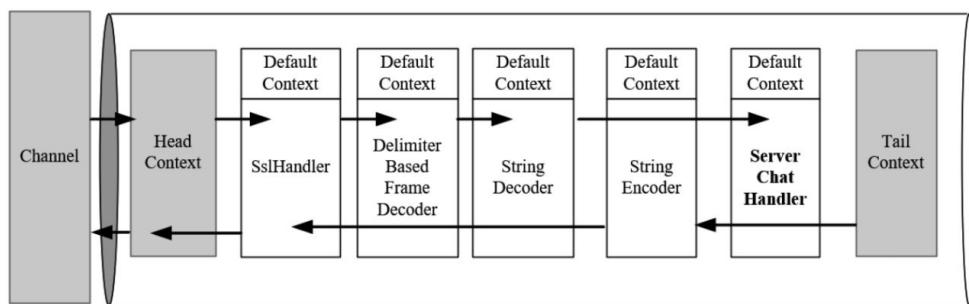


图12-31 Netty安全聊天器服务端处理器流水线的构成

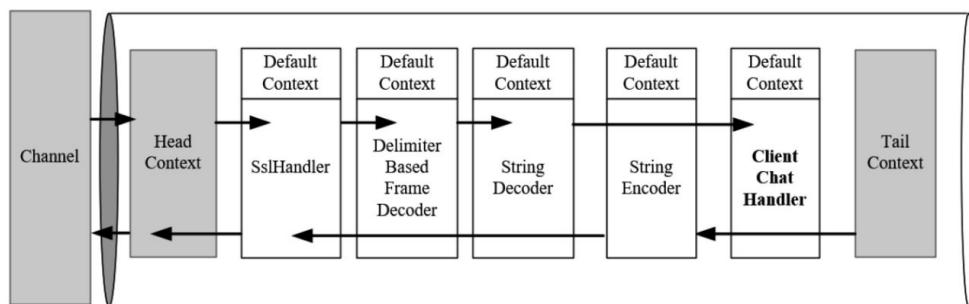


图12-32 Netty安全聊天器客户端处理器流水线的构成

为了简单，本演示实例的通信内容使用字符串直接通信，没有使用Json、Protobuf等应用层协议，消息使用“\r\n”回车换行符作为结束标准，然后使用DelimiterBasedFrameDecoder分包处理器按照“\r\n”进行分包处理。

由于需要安全通信，因此流水线加入了Netty内置的SSLEngine处理器，由其负责SSL/TLS协议的安全通信握手、传输的加密和解密处理。接下来详细介绍一下该安全处理器。

12.8.2 Netty内置SSLEngine处理器详解

为了支持SSL/TLS，Java中的JSSE提供了SSLContext和SSLEngine基础类，从而使得SSL/TLS握手、解密加密变得相当简单直接。Netty的内置SslHandle处理器是基于JSSE的SSLEngine来完成安全传输的；SslHandle使用SSLEngine完成入站和出站字节流的安全处理。

SSLEngine从底层的I/O传输机制中分离出SSL/TLS抽象安全操作，并且将SSL/TLS安全机制应用在入站和出站的字节流上，使之与底层的传输机制无关，所以SSLEngine传输引擎可以被用于各种I/O类型，包括NIO、OIO、I/O流、ByteBuffers缓冲区或字节数组、未来的异步I/O模型等。

在介绍SslHandle之前，先介绍一下JSSE类库中SSLEngine的五个阶段。

(1) 创建阶段。该阶段的SSLEngine实例已经被创建和初始化，但尚未被使用和开启握手。在此阶段，应用程序可以修改SSLEngine实例的设置（如密钥套件、握手时处于客户端还是服务端模式等）。一

一旦握手开始，新的设置都将在下一次握手时才被启用，但是对客户端/服务端模式的设置除外。

(2) 初始握手阶段。初始握手阶段是两端交换通信参数，直到SSLSession安全会话完全建立为止。在此阶段不能发送应用程序数据。

(3) 应用通信阶段。一旦通信参数建立起来且握手完成，就可以通过SSLEngine传输应用程序数据。出站的应用程序报文被SSLEngine加密并进行完整性保护，入站的报文进行相反的过程。

(4) 重新握手阶段。在应用通信阶段的任何时刻，每一方都可以请求重新协商SSLSession安全会话。当然，新的握手消息可以混入应用程序数据中。在开始重新握手阶段之前，应用程序可以重置SSL/TLS通信参数。例如，可以重新设置已启用的密钥套件列表，也可以重新设置是否对客户端进行身份验证等。但是，重新握手时，不能更改客户端/服务端模式。如前所述，一旦重新握手开始，任何新的SSLEngine设置都将在下一次握手时才被使用。

(5) 关闭阶段。当不再需要SSLEngine实例时，应用程序应该关闭SSLEngine，并且在关闭底层传输机制之前应该发送所有剩余的报文到对方。一旦SSLEngine引擎关闭，该实例将不可重用。

如何获取JSSE类库SSLEngine的实例呢？可以通过SSLContext实例创建一个SSLEngine实例，调用SSLContext.createSSLEngine()方法。大致的创建代码如下：

```
//创建客户端SSL 上下文  
SSLContext clientSSLContext = createClientSSLContext();
```

```
//创建客户端 SSL 引擎  
SSLEngine sslEngine = clientSSLContext.createSSLEngine();
```

Netty的SslHandle处理器使用了JSSE类库SSLEngine的实例。在创建SslHandle实例之前，需提前创建SSLEngine引擎实例，并且以SSLEngine实例作为输入，去实例化新建的SslHandle处理器实例，然后将新建的SslHandle实例作为第一个处理器加入到通道流水线。

安全聊天演示中客户端流水线初始化的代码如下：

```
ChannelPipeline pipeline = ch.pipeline();  
//创建客户端ssl上下文  
SSLContext clientSSLContext = createClientSSLContext();  
//创建客户端SSL引擎  
SSLEngine sslEngine = clientSSLContext.createSSLEngine();  
//在握手的时候使用客户端模式  
sslEngine.setUseClientMode(true);  
//设置需要验证对端(服务端)身份，需服务端证实自己的身份  
sslEngine.setNeedClientAuth(true);  
//创建ssl处理器，并加入到流水线  
pipeline.addLast("ssl", new SslHandler(sslEngine));
```

在客户端流水线初始化代码中有以下两个要点：

(1) 设置了SSLEngine实例为客户端模式，是通过调用setUseClientMode(true)来完成的，表明第一个SSL握手报文Client Hello由本端发起。

(2) 设置了需要对对端进行身份认证（如果对端提供数字证书）。如果不要为对端（服务端）进行身份认证，`setNeedClientAuth()`的参数可以设置为`false`，或者不做专门设置而使用默认值`false`。这里调用了`setNeedClientAuth(true)`，表示引擎`SSLEngine`需要为对端（服务端）进行身份认证。

安全聊天演示中服务端流水线的初始化代码如下：

```
ChannelPipeline pipeline = sc.pipeline();
//创建服务端SSL上下文实例
SSLContext serverSSLContext = createServerSSLContext();
//通过上下文实例创建服务端的 SSL 引擎
SSLEngine sslEngine =serverSSLContext.createSSLEngine();
//单向认证：在服务端设置不需要验证对端身份，无须客户端证实自己的身份
sslEngine.setNeedClientAuth(false);
//在握手的时候，使用服务端模式
sslEngine.setUseClientMode(false);
//创建SslHandler处理器
ChannelHandler sslHandler=new SslHandler(sslEngine);
//将处理器加入到流水线
pipeline.addLast(sslHandler);
```

在服务端流水线初始化代码中有以下两个要点：

(1) 设置了`SSLEngine`实例为服务端模式，是通过调用`setUseClientMode(false)`来完成的。SSL/TSL连接的双方只能有一方为客户端，另一方为服务端。

(2) 这里调用了setNeedClientAuth (false)，表示服务端引擎SSLEngine不需要为对端（客户端）进行身份认证，这种模式属于SSL/TSL单向认证模式。如果需要进行双向认证，在服务端需要调用setNeedClientAuth (true)，用于设置在握手阶段对客户端进行身份认证。

入站的加密的安全传输数据包被SslHandler拦截后，交由SSLEngine解密后入站；普通的出站数据也会被SslHandler拦截，交由SSLEngine加密后成为安全传输数据包，之后再出站。SslHandler既是一个入站处理器，也是一个出站处理器，如图12-33所示。

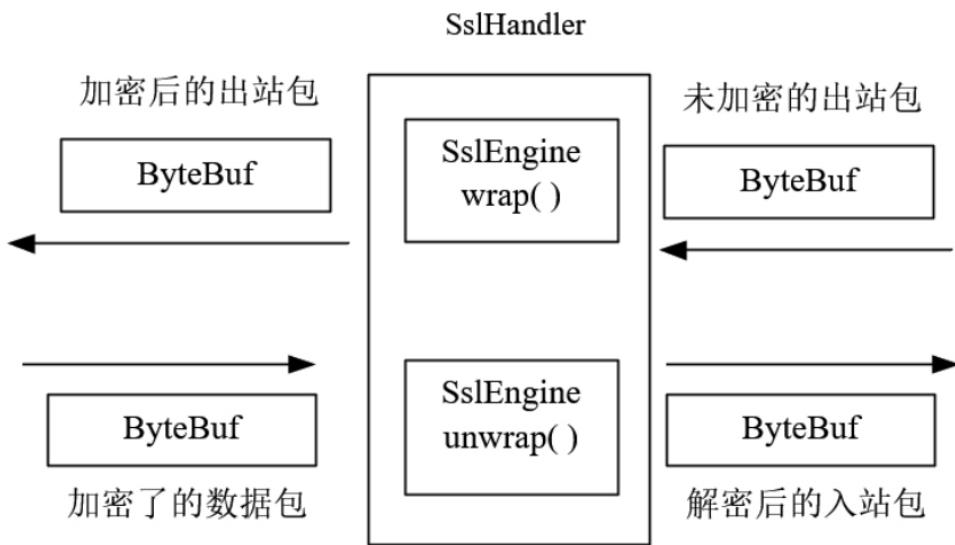


图12-33 SslHandler处理器示意图

Netty的SSLEngine类有三个重要的方法，具体如下：

(1) beginHandshake(): 在当前SSLEngine实例上发起握手（初始握手或重新握手）。

(2) wrap(): 尝试把应用出站数据包编码成SSL/TLS安全传输数据包。

(3) unwrap(): 尝试把入站的SSL/TLS安全传输数据包解码成应用出站数据包。

12.8.3 Netty的简单安全聊天器服务端程序

基于Netty的简单安全聊天器服务端程序除了在通道初始化时在流水线上增加安全处理器SSLEngine的实例之外，其他代码与普通的聊天器服务端程序没有任何区别。

服务端的主要代码大致如下：

```
package com.crazymakercircle.secure.netty.securechat;  
//省略import  
/**基于Netty的简单安全聊天器服务端程序*/  
public class SecureChatServer  
{  
    /**  
     * 通道初始化处理器  
     */  
    static class SecureChatServerInitializer  
        extends  
    ChannelInitializer<SocketChannel>  
    {  
        @Override
```

```
protected void initChannel(SocketChannel sc) throws
Exception
{
    ChannelPipeline pipeline = sc.pipeline();
    //创建服务端SSL上下文实例
    SSLContext serverSSLContext =
createServerSSLContext();
    //通过上下文实例， 创建服务端的 SSL 引擎
    SSLEngine sslEngine
=serverSSLContext.createSSLEngine();
    //单向认证：在服务端设置不需要验证对端身份，无须客户端证实自
己的身份
    sslEngine.setNeedClientAuth(false);
    //在握手时， 使用服务端模式
    sslEngine.setUseClientMode(false);
    //创建SslHandler处理器
    ChannelHandler sslHandler=new
SslHandler(sslEngine);
    //将处理器加入到流水线
    pipeline.addLast(sslHandler);

    //添加分包器
    pipeline.addLast("framer",
        new DelimiterBasedFrameDecoder(8192,
Delimiters.lineDelimiter()));

    //添加字符串解码器
```

```
        pipeline.addLast("decoder", new StringDecoder());
        //添加字符串编码器

        pipeline.addLast("encoder", new StringEncoder());
        //添加聊天处理器

        pipeline.addLast("handler", new
ServerChatHandler());
    }

}
```

客户端的处理器流水线的装配流程与服务端流水线的装配流程是基本相同的，具体请参见疯狂创客圈社群源码，这里不再赘述。

在聊天的过程中，客户端通过控制台收集输入内容，然后发送给服务端。客户端的相关代码大致如下：

```
package com.crazymakercircle.secure.netty.securechat;
//省略import

/**
 * 基于Netty的简单安全聊天器客户端程序
 */

public class SecureChatClient
{
    /**
     * 开始客户端
     */

    public void start(String host, int port) throws Exception
    {
        EventLoopGroup group = new NioEventLoopGroup();
```

```
try
{
    Bootstrap b = new Bootstrap();
    b.group(group).channel(NioSocketChannel.class)
        .handler(new SecureChatClientInitializer());
    //开始连接服务器
    Channel ch = b.connect(host,
port).sync().channel();

    //从控制台获取输入的内容
    ChannelFuture writeFuture = null;
    BufferedReader reader =
        new BufferedReader(new
InputStreamReader(System.in));
    for (; ; )
    {
        String line = reader.readLine();
        if (line != null)
        {
            //发送控制台输入内容
            writeFuture = ch.writeAndFlush(line +
"\r\n");
        }
        //如果输入bye， 表示终止连接
        if ("bye".equals(line.toLowerCase()))
        {
            ch.closeFuture().sync();
        }
    }
}
```

```

        break;
    }

}

//发送完成之后，再接收下一轮的输入

if (writeFuture != null)

{
    writeFuture.sync();

}

} finally

{

    //优雅关闭

group.shutdownGracefully();

}

}

}

```

安全聊天器服务端与客户端的测试用例，具体如下：

```

package com.crazymakercircle.secure.test.SecureChat;

//省略import

/**
 * 基于Netty的简单安全聊天器，测试用例
 * create by 尼恩 @ 疯狂创客圈
 */

public class SecureChatTester
{
    /**

```

```

    * 启动安全聊天器服务端
    */
    @Test
    public void startSecureChatServer() throws Exception
    {
        new SecureChatServer().start(18899);
    }

    /**
     * 启动安全聊天器客户端
     */
    @Test
    public void startClient() throws Exception
    {
        new SecureChatClient().start("localhost", 18899);
    }
}

```

开发工具IDEA在执行JUnit测试用例时，默认是不能控制输入的（Eclipse好像不存在这个问题）。在IDEA中如何为JUnit测试用例开启控制台输入的呢？需要为IDEA进行一下简单的设置，在idea64.exe.vmoptions配置文件中添加选项，具体如下：

```
-Deditable.java.test.console=true
```

具体的操作方法为：在IDEA中单击菜单栏最右侧的Help菜单，然后单击Edit Custom VM Options菜单项，打开其虚拟机选项配置文件，然后在该文件中添加上面的选项。以上操作过程如图12-34所示。

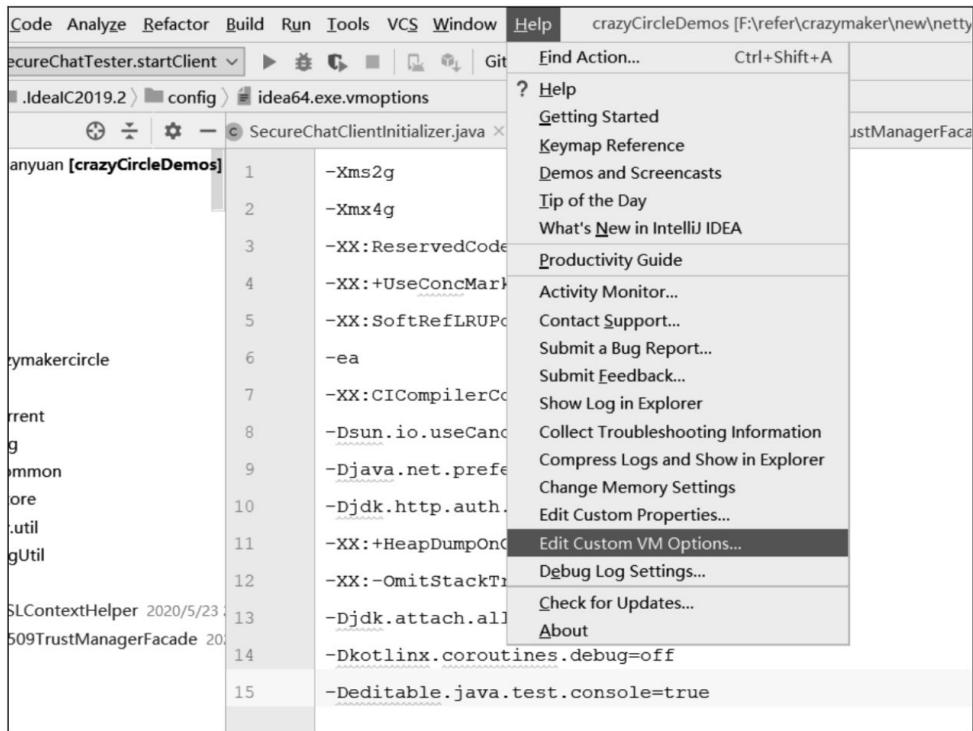


图12-34 在IDEA中为Junit测试用例开启控制台输入

设置好之后重启IDEA开发工具，客户端程序就可以在控制台中输入测试内容了。

说明

为了节省篇幅，以上安全聊天程序实例代码仅仅节选了一部分，由于源码工程一直在迭代，因此完整的最新代码要从疯狂创客圈社群的Git仓库下载（具体地址请参见社群公告）。由于代码在持续更新、优化，因此即使本书已出版，工程代码后续也可能会局部更新。强烈建议大家认真阅读、执行本实例的Git仓库源码。

12.9 HTTPS协议安全通信实战

HTTPS (Hyper Text Transfer Protocol over Secure Socket Layer) 是以安全为目标的HTTP通信协议，简单地说就是HTTP的安全版。Netty默认提供HTTP，所以Netty内置的SslHandler处理器同样支持HTTPS协议。

12.9.1 使用Netty实现HTTPS回显服务端程序

本节的演示实例是一个简单的基于Netty的HTTPS回显服务器，使用SSL/TLS协议进行HTTP通信加密。回显服务器的服务端处理器流水线的构成如图12-35所示。

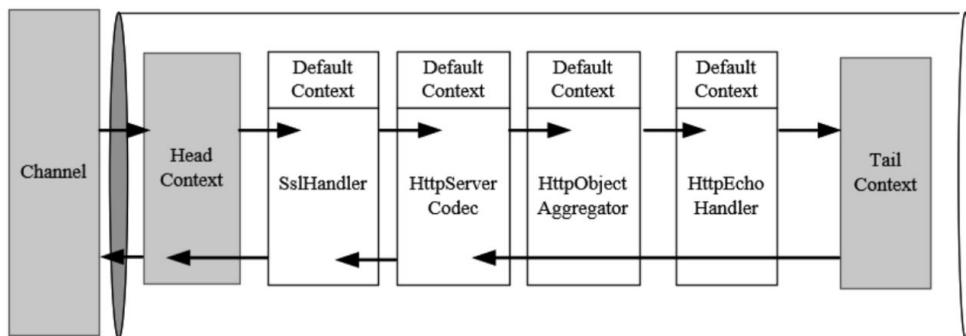


图12-35 HTTPS回显服务器的服务端处理器流水线的构成

Netty的HTTPS回显服务器的服务端处理器流水线的装配代码大致如下：

```
package com.crazymakercircle.secure.netty.https.server;  
//省略import  
public class HttpsServerInitializer extends
```

```
ChannelInitializer<SocketChannel>      {  
  
    @Override  
    protected void initChannel(SocketChannel ch) throws  
Exception  
    {  
        ChannelPipeline pipeline = ch.pipeline();  
        //创建服务端SSL上下文实例  
        SSLContext serverSSLContext = createServerSSLContext();  
        //通过上下文实例创建服务端的 SSL 引擎  
        SSLEngine sslEngine  
        =serverSSLContext.createSSLEngine();  
        //在握手的时候，使用服务端模式  
        sslEngine.setUseClientMode(false);  
        //单向认证：在服务端设置不需要验证对端身份，无须客户端证实自己的身  
份  
        sslEngine.setNeedClientAuth(false);  
        //创建SslHandler处理器，并加入到流水线  
        pipeline.addLast(new SslHandler(sslEngine));  
  
        //请求解码器和响应编码器  
        pipeline.addLast(new HttpServerCodec());  
        //HttpObjectAggregator 将HTTP消息的多个部分合成一条完整的  
HTTP消息  
        pipeline.addLast(new HttpObjectAggregator(65535));  
        //自定义的业务handler，回显HTTP请求URI、请求方法、请求参数
```

```
        pipeline.addLast(new HttpEchoHandler());  
    }  
}
```

说明

以上代码中自定义的业务处理器HttpEchoHandler前面已经介绍过，主要用于向客户端回显发送HTTP的请求URI、请求方法、请求参数等内容。具体的HttpEchoHandler代码，请参见[疯狂创客圈社群Git仓库](#)中的工程源码。

12.9.2 通过HttpsURLConnection发送HTTPS请求

在没有性能要求的场景下，大家可以使用JDK内置的HttpsURLConnection访问HTTP服务器，使用JDK内置的HttpsURLConnection（注意多了个字母s）访问HTTPS服务器。在访问HTTPS服务器时，客户端同样会涉及服务器身份证书导入、客户端SSLContext上下文的创建等与安全相关的工作。

安全连接类HttpsURLConnection是对HttpURLConnection的扩展，支持各种特定于HTTPS协议的通信功能。该类提供了setSSLSocketFactory(SSLocketFactory)静态方法，用于设置其创建连接时用到的SSLSocketFactory安全套接字工厂实例。

这里使用HttpsURLConnection实现HTTPS回显服务器的客户端程序，核心代码节选如下：

```
package com.crazymakercircle.secure.netty.https.client;  
//省略import  
  
/**  
 * HTTPS回显服务器的客户端程序  
 * 通过JDK自带的HttpURLConnection发送HTTPS请求  
 */  
  
@Slf4j  
public class SecureHttpClient  
{  
    /**  
     * 通过JDK自带的HttpURLConnection发送HTTPS请求  
     * @param path 请求地址  
     */  
  
    public static void sentRequest(String path) throws  
Exception  
    {  
        //创建客户端SSLContext上下文  
        SSLContext clientSSLContext = createClientSSLContext();  
        //创建安全套接字工厂  
        SSLSocketFactory factory =  
clientSSLContext.getSocketFactory();  
  
        //主机名称校验  
        HostnameVerifier hostnameVerifier = new
```

```
HostnameVerifier()
{
    public boolean verify(String hostname, SSLSession
sslsession)
    {
        //验证请求的主机名称，这里假设只能请求服务端配置的主机名
        if
(SystemConfig.SOCKET_SERVER_IP.equals(hostname))
        {
            return true;
        } else
        {
            log.error("主机名称校验失败");
            return false;
        }
    }
};

//设置连接的主机名称校验

HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier)
;

//设置连接的安全套接字工厂
HttpsURLConnection.setDefaultSSLSocketFactory(factory);

URL url = new URL(path);
//打开连接
```

```
HttpURLConnection conn = url.openConnection();

//获取响应码
int code = conn.getResponseCode();
//log.info("收到消息", conn.getResponseMessage());
if (code < 400)

{
    //输入流
    BufferedInputStream bis =
        new
        BufferedInputStream(conn.getInputStream());
    StringBuffer buffer = new StringBuffer();
    //累积完成的长度
    long finished = 0;
    int len = 0;
    byte[] buff = new byte[1024 * 8];
    while ((len = bis.read(buff)) != -1)
    {
        buffer.append(new String(buff, "UTF-8"));
        finished += len;
        log.info("共完成传输字节数 {}", finished);
    }
    System.out.println("echo = " + buffer.toString());
}
}
```

安全连接类HttpsURLConnection除了需要设置SSL安全套接字工厂实例外，还需要设置主机名称校验器，该校验器用于校验来自请求URL的主机名称是否为安全的主机名称。如果URL的主机名和服务器的标识主机名不匹配，则请求不能发送出去。

12.9.3 HTTPS服务端与客户端的测试用例

HTTPS回显程序的服务端与客户端测试用例代码如下：

```
package com.crazymakercircle.secure.test.https;  
//省略import  
/**  
 * HTTPS回显服务器的测试用例  
 */  
@Slf4j  
public class HttpsTester  
{  
    /**  
     * HTTPS回显服务器的服务端程序的测试用例  
     */  
    @Test  
    public void startHttpsNettyServer() throws Exception  
    {  
        NettyHttpsServer.start();  
    }  
    /**
```

```
* HTTPS回显服务器的客户端程序的测试用例
*/
@Test
public void startClient() throws Exception
{
    //抓包说明：由于WireShark只能抓取经过网卡的包，
    //如果要抓取本地的调试包，需要通过route指令增加服务器IP的路由配
置
    //让发往服务器的报文首先发送到被抓包工具监控的网卡所指向的网关
    //route add 增加路由
    //表示发往192.168.0.5（网卡IP）的请求下一跳网关为192.168.0.1
    //route add 192.168.0.5 mask 255.255.255.255
    192.168.0.1
    SecureHttpClient.sendRequest(
        "https://192.168.0.5:18899/?param1=value1");
}
}
```

在测试的过程中，如果需要使用WireShark抓包工具查看SSL/TSL握手报文，就需要为网卡地址添加路由配置，具体配置命令请参见代码中的注释说明。

如果抓包成功，就会看到服务端和客户端之间是单向认证，握手报文大致如图12-36所示。

The screenshot shows a Wireshark capture window titled "正在捕获 WLAN". The menu bar includes: 文件(E) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(I) 帮助(H). The toolbar includes: 新建(N), 打开(O), 保存(S), 退出(X), 捕获(C), 分析(A), 统计(S), 电话(Y), 无线(W), 工具(I), 帮助(H). A search bar at the top right contains: 捕获(F), 分析(A), 统计(S), 电话(Y), 无线(W), 工具(I), 帮助(H). The main pane displays a list of network frames. A search filter is applied: (ip.src == 192.168.0.5)&& (ip.dst == 192.168.0.5). The columns are: No., Source, Destination, Protocol, Info. The frames show the following sequence:

- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Client Hello
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Server Hello, Certificate, Server Key Exchange, Server Hello Done
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Client Key Exchange
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Change Cipher Spec
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Encrypted Handshake Message
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Change Cipher Spec, Encrypted Handshake Message
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Encrypted Alert
- 1... 192.168.0.5 192.168.0.5 TLSv1.2 Encrypted Alert

No.	Source	Destination	Protocol	Info
1...	192.168.0.5	192.168.0.5	TLSv1.2	Client Hello
1...	192.168.0.5	192.168.0.5	TLSv1.2	Server Hello, Certificate, Server Key Exchange, Server Hello Done
1...	192.168.0.5	192.168.0.5	TLSv1.2	Client Key Exchange
1...	192.168.0.5	192.168.0.5	TLSv1.2	Change Cipher Spec
1...	192.168.0.5	192.168.0.5	TLSv1.2	Encrypted Handshake Message
1...	192.168.0.5	192.168.0.5	TLSv1.2	Change Cipher Spec, Encrypted Handshake Message
1...	192.168.0.5	192.168.0.5	TLSv1.2	Encrypted Alert
1...	192.168.0.5	192.168.0.5	TLSv1.2	Encrypted Alert

图12-36 HTTPS回显程序服务端与客户端的握手报文

第13章 ZooKeeper分布式协调

高并发系统为了应对流量增长需要进行节点的横向扩展，所以高并发系统往往都是分布式系统。高并发系统基本都需要进行节点与节点之间的配合协调，这就需要用到分布式协调中间件（如ZooKeeper）。

ZooKeeper（本书简称ZK）是Hadoop的正式子项目，是一个针对大型分布式的可靠的协调系统，提供的功能包括配置维护、名字服务、分布式同步、组服务等。

ZooKeeper的目标就是封装好复杂、易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

ZooKeeper在实际生产环境中应用非常广泛，比如SOA的服务监控系统，大数据基础平台Hadoop、Spark的分布式调度系统。

13.1 ZooKeeper伪集群安装和配置

ZooKeeper的运行一般是集群模式，而不是单节点模式，现在我们开始使用一台机器来搭建一个ZooKeeper学习集群。由于没有多余的服务器，因此这里将三个ZooKeeper节点都安装到一台机器上，故称之为“伪集群模式”。

说明

伪集群模式只便于开发、单元测试，不能用于生产环境。实际上，伪集群模式下的安装和配置与生产环境下的步骤差不多。

首先是下载ZooKeeper。在Apache的官方网站提供了很多镜像下载地址，找到对应的版本（比如3.4.13）即可。其下载地址为
<http://mirrors.cnnic.cn/apache/ZooKeeper/ZooKeeper-3.4.13/ZooKeeper-3.4.13.tar.gz>。

在Windows下安装，需要把下载的ZooKeeper文件解压到指定目录，比如C:\devtools\ZooKeeper-3.4.13\>。

接下来将用以上目录作为默认安装目录。

13.1.1 创建数据目录和日志目录

安装ZooKeeper之前需要规划一下节点。ZooKeeper节点数有以下要求：

(1) ZooKeeper集群节点数必须是基数。

ZooKeeper集群中需要一个主节点，称为Leader节点，并且Leader节点是集群通过选举规则从所有节点中选出来的，简称为选主。选主规则中很重要的一条是：要求“可用节点数量 > 总节点数量/2”。如果是偶数个节点，则会出现不满足这个规则的情况，比如出现“可用节点数量=总节点数量/2”的情况时就不满足选主的规则。

说明

为什么要“可用节点数量 > 总节点数量/2”呢？为了防止集群脑裂（Split-Brain）。脑裂是分布式系统的共性问题，ElasticSearch集群也面临此问题。脑裂是一个形象的比喻，好比“大脑分裂”，也就是说本来一个“大脑”，却被拆分为两个或多个“大脑”。集群脑裂是由于网络断了，一个集群被分成了两个集群。ZooKeeper集群、ElasticSearch集群都使用一种简单的节点数过半机制来确保集群被分裂后还能正常工作。过半机制是指“可用节点数量 > 总节点数量/2”时，集群才是可用的，才可以对外服务；否则集群是非可用的，不可以提供服务。笔者经常在Java工程师、高级工程师甚至架构师的面试中使用脑裂问题去考察候选人，有很多候选人答不上来。

(2) ZooKeeper集群至少有三个节点。

一个节点的ZooKeeper服务可以正常启动和提供服务，但是一个节点的ZooKeeper服务不能叫作集群，其可靠性会大打折扣，仅仅作为学习使用。正常情况下，搭建ZooKeeper集群至少需要三个节点。

作为学习案例，这里在本地机器（Windows系统）上规划搭建一个三个节点的伪集群。安装集群的第一步是在安装目录下提前为每一个伪节点创建好两个目录：日志目录、数据目录。

首先创建日志目录。为三个节点中的每一个伪节点创建一个日志目录，分别为log/zoo-1、log/zoo-2、log/zoo-3，如图13-1所示。

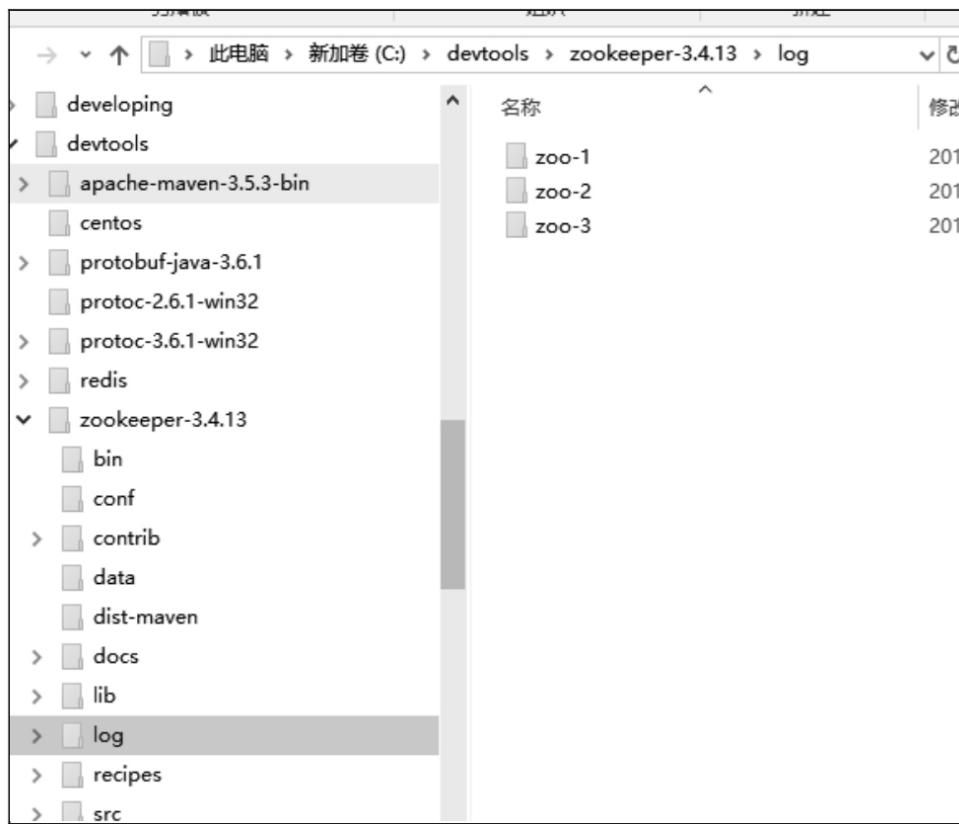


图13-1 集群中三个伪节点的日志目录

接下来开始创建数据目录。在安装目录下为伪集群三个节点中的每一个伪节点创建一个数据目录，分别为data/zoo-1、data/zoo-2、data/zoo-3。

13.1.2 创建myid文本文件

安装集群的第二步是为每一个节点创建一个id文件。什么是id文件呢？每一个节点都需要有一个存放节点id的文本文件，文件名为myid。myid文件的特点如下：

- (1) myid文件的唯一作用就是存放（伪）节点的编号。
- (2) myid文件是一个文本文件，文件名为myid。
- (3) myid文件内容为一个数字，表示节点的编号。
- (4) myid文件中只能有一个数字，不能有其他的内容。
- (5) myid文件默认存放在data目录下。

下面分别为三个节点创建二个myid文件：

- (1) 在第一个伪节点的数据目录C:\devtools\ZooKeeper-3.4.13\data\zoo-1\文件夹下创建一个myid文件，文件的内容为“1”，表示第一个节点的编号为1。
- (2) 在第二个伪节点的数据目录C:\devtools\ZooKeeper-3.4.13\data\zoo-2\文件夹下创建一个myid文件，文件的内容为“2”，表示第二个节点的编号为2。

(3) 在第三个伪节点的数据目录C:\devtools\ZooKeeper-3.4.13\data\zoo-3\文件夹下创建一个myid文件，文件的内容为“3”，表示第三个节点的编号为3。

ZooKeeper对id值有两点要求：

- (1) myid文件中的id值只能是一个数字，即一个节点的编号ID。
- (2) id的范围是1~255，表示集群最多的节点个数为255个。

13.1.3 创建和修改配置文件

安装集群的第三步是为每一个节点创建一个配置文件。创建配置文件不需要从零开始，在ZooKeeper的配置目录conf下，官方有一个配置文件的样例——zoo_sample.cfg。复制这个样例，修改其中的某些配置项即可。

接下来分别为三个节点创建三个“.cfg”配置文件，具体的步骤为：

(1) 将配置文件的样例zoo_sample.cfg文件复制三份，为每一个节点复制一份，分别命名为zoo-1.cfg、zoo-2.cfg、zoo-3.cfg，这些名称对应到三个节点。

(2) 修改每一个节点的“.cfg”配置文件。将前面准备的日志目录、数据目录配置到“.cfg”的正确选项中。

```
dataDir = C:/devtools/ZooKeeper-3.4.13/data/zoo-1/  
dataLogDir= C:/devtools/ZooKeeper-3.4.13/log/zoo-1/
```

两个选项的介绍如下：

- ① dataDir：数据目录选项，配置为前面准备的数据目录。非常关键的myid文件处于此目录下。
 - ② dataLogDir：日志目录选项，配置为前面准备的日志目录。如果没有设置该参数，默认将使用和dataDir相同的设置。
- (3) 配置集群中的端口信息、节点信息、时间选项等。

① 端口选项的配置示例如下：

```
clientPort = 2181
```

选项clientPort表示客户端程序连接ZooKeeper集群中节点的端口号。在生产环境的集群中，不同的节点处于不同的机器上，clientPort端口号一般都相同，以便于记忆和使用。由于这里是伪集群模式，三个节点集中在一台机器上，所以三个端口号需要配置为不一样的，以避免端口冲突。

选项clientPort的值一般设置为2181。伪集群下，不同节点的clientPort不能相同，可以按照编号进行累加，比如第一个节点为2181、第二个节点为2182、第三个节点为2183。

② 配置文件“.cfg”的集群节点信息，示例如下：

```
server.1=127.0.0.1:2888:3888  
server.2=127.0.0.1:2889:3889  
server.3=127.0.0.1:2890:3890
```

集群节点信息需要配置集群中所有节点的ID编号、IP、端口，每个节点的格式为：

```
server.id=host:port:port
```

在ZooKeeper集群中，每个节点都需要感知到整个集群是由哪些节点组成的，所以每个配置文件都需要配置全部节点。在“.cfg”配置文件中可以使用“server. id”格式进行节点的配置，每一行都代表一个节点。配置节点时注意几点：

- 不能有相同id的节点。
- 每一行“`server. id=host:port:port`”中的id值需要与所对应节点的数据目录下的`myid`中的id值保持一致。
- 每个配置文件都需要配置全部的节点信息，不仅仅是配置自己的，而是需要所有节点的id、ip、端口配置。
- 每一行“`server. id=host:port:port`”中需要配置两个端口。前一个端口（如示例中的2888）用于节点之间的通信，为通信端口；后一个端口（如示例中的3888）用于选举Leader主节点，为选主端口。
- 在伪集群的模式下，每一行记录相同的端口必须修改为不一样，主要是为了避免端口冲突。

在分布式集群模式下，由于不同节点的IP不同，因此在每一行的节点配置记录中，通信端口和选主端口都可以相同，例如：

```
server.1=10.10.10.1:2888:3888  
server.2=10.10.10.2:2888:3888  
server.3=10.10.10.3:2888:3888
```

③ 最后是时间相关选项的配置，示例如下：

```
tickTime=4000  
initLimit = 10  
syncLimit = 5
```

对以上时间选项的说明如下：

- **tickTime**: 配置单元时间。单元时间是ZooKeeper的时间计算单元，其他的时间间隔都是使用**tickTime**的倍数来表示的。如果不进行配置，则单元时间默认值为3000，单位是毫秒（ms）。
- **initLimit**: 节点的初始化时间。该参数用于Follower（从节点）启动，并完成从Leader（主节点）同步数据的时间。
Follower节点在启动过程中会与Leader建立连接并完成对数据的同步，从而确定自己的起始状态。Leader节点允许Follower在**initLimit**时间内完成这个工作。该参数的默认值为10，表示是参数**tickTime**值的10倍。此参数必须配置，且为正整数。
- **syncLimit**: 心跳最大延迟周期。该参数用于配置Leader服务器和Follower之间进行心跳检测的最大延时时间。在ZooKeeper集群运行的过程中，Leader服务器会通过心跳检测来确定Follower服务器是否存活。如果Leader服务器在**syncLimit**时间内无法获取到Follower的心跳检测响应，那么Leader就会认为该Follower已经脱离了与自己的同步。该参数默认值为5，表示是参数**tickTime**值的5倍。此参数必须配置，且为正整数。

13.1.4 配置文件示例

完成了伪集群的日志目录、数据目录、myid文件、“.cfg”配置文件的准备和配置之后，伪集群的安装工作基本完成了。

在伪集群配置过程中，“.cfg”文件的配置是最为关键的环节。下面给出三份配置文件实际的代码。

第一个节点的配置文件zoo-1.conf:

```
tickTime=4000
initLimit = 10
syncLimit = 5
dataDir = C:/devtools/ZooKeeper-3.4.13/data/zoo-1/
dataLogDir= C:/devtools/ZooKeeper-3.4.13/log/zoo-1/

clientPort = 2181
server.1 = 127.0.0.1:2888:3888
server.2 = 127.0.0.1:2889:3889
server.3 = 127.0.0.1:2890:3890
```

第二个节点的配置文件zoo-2.conf:

```
tickTime=4000
initLimit = 10
syncLimit = 5
dataDir = C:/devtools/ZooKeeper-3.4.13/data/zoo-2/
dataLogDir= C:/devtools/ZooKeeper-3.4.13/log/zoo-2/

clientPort = 2182
```

```
server.1 = 127.0.0.1:2888:3888  
server.2 = 127.0.0.1:2889:3889  
server.3 = 127.0.0.1:2890:3890
```

第三个节点的配置文件zoo-3.conf:

```
tickTime=4000  
initLimit = 10  
syncLimit = 5  
dataDir = C:/devtools/ZooKeeper-3.4.13/data/zoo-3/  
dataLogDir= C:/devtools/ZooKeeper-3.4.13/log/zoo-3/  
clientPort = 2183  
server.1 = 127.0.0.1:2888:3888  
server.2 = 127.0.0.1:2889:3889  
server.3 = 127.0.0.1:2890:3890
```

通过三个配置文件可以看出，对于不同的节点，“.cfg”的配置项的内容大部分相同。

说明

每个节点的“.cfg”配置文件中的集群节点信息都是全量的；不同的是每个节点的数据目录dataDir、日志目录dataLogDir和对外服务端口clientPort，仅仅配置自己的那份。

13.1.5 启动ZooKeeper伪集群

为了很方便地启动每个节点，需要为每个节点制作一份启动命令，在Windows平台上启动命令为一份“.cmd”文件。

在ZooKeeper的bin目录下，通过复制zkServer.cmd样本文档，为每个伪节点创建一个启动的命令文件，分别为zkServer-1.cmd、zkServer-2.cmd、zkServer-3.cmd。

修改复制后的“.cmd”文件主要为每个节点增加“.cfg”配置文件的选项，选项名称为ZOOCFG。修改之后，第一个节点的启动命令zkServer-1.cmd的代码如下：

```
setlocal
call "%~dp0zkEnv.cmd"

set ZOOCFG=C:\devtools\ZooKeeper-3.4.13\conf\zoo-1.cfg

set ZOOMAIN=org.apache.ZooKeeper.server.quorum.QuorumPeerMain
echo on
call %JAVA% "-DZooKeeper.log.dir=%ZOO_LOG_DIR%" "-
DZooKeeper.root.logger=%ZOO_LOG4J_PROP%" -cp "%CLASSPATH%" "%ZOOMAIN%" "%ZOOCFG%" %

endlocal
```

另外两个“.cmd”文件与zkServer-1.cmd做同样的修改即可，这里不再赘述。

接下来打开一个Windows的命令控制台，进入bin目录，并且启动zkServer-1.cmd，这个脚本中会启动第一个节点的Java服务进程：

```
C:\devtools\ZooKeeper-3.4.13>cd bin  
C:\devtools\ZooKeeper-3.4.13\bin>  
C:\devtools\ZooKeeper-3.4.13\bin> zkServer-1.cmd
```

ZooKeeper集群需要有1/2以上的节点启动才能完成集群的启动，对外提供服务。所以，至少需要再启动两个节点。

打开另外一个Windows的命令控制台进入bin目录，并且启动zkServer-2.cmd，这个脚本中会启动第一个节点的Java服务进程：

```
C:\devtools\ZooKeeper-3.4.13>cd bin  
C:\devtools\ZooKeeper-3.4.13\bin>  
C:\devtools\ZooKeeper-3.4.13\bin>zkServer-2.cmd
```

由于这里没有使用后台服务启动的模式，因此这两个节点服务的命令窗口在服务期间不能关闭。启动之后，如何验证集群的启动成功呢？有以下两种方法。

方法一：通过jps命令，可以看到QuorumPeerMain进程的数量。

```
C:\devtools\ZooKeeper-3.4.13\bin >jps  
1344 QuorumPeerMain  
13380 QuorumPeerMain  
9740 Jps
```

方法二：通过ZooKeeper客户端命令zkCli.cmd去尝试连接ZooKeeper的服务，判断是否能连接集群。如果最后显示出CONNECTED

连接状态，表示已经成功连接，大致的输出如下：

```
PS C:\devtools\ZooKeeper-3.4.13\bin> .\zkCli.cmd -server  
127.0.0.1:2181  
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8  
Connecting to 127.0.0.1:2181  
//省略一些连接日志  
WatchedEvent state:SyncConnected type:None path:null  
[zk: 127.0.0.1:2181(CONNECTED) 0]
```

连接成功后，可以通过输入ZooKeeper的客户端命令操作“ZNode”树的节点。

说明

在Windows下，ZooKeeper是通过cmd的批处理命令运行的，官方没有提供Windows后台服务方案。为了避免每次关闭后还需要使用cmd启动带来的不便，可以通过第三方工具prunsrv将ZooKeeper做成Windows后台服务。

一般情况下，ZooKeeper都运行在Linux操作系统上，有关在Linux下的伪集群安装，可以查看疯狂创客圈的博客《Linux ZooKeeper安装，带视频》，此文章的具体地址请参考疯狂创客圈社群博客首页。

13.2 使用ZooKeeper进行分布式存储

本节首先给大家介绍一下ZooKeeper存储模型，然后介绍如何使用客户端命令操作ZooKeeper的存储模型。

13.2.1 详解ZooKeeper存储模型

ZooKeeper的存储模型非常简单，和Linux的文件系统非常类似。简单地说，ZooKeeper的存储模型是一棵以“/”为根节点的树，存储模型中的每一个节点叫作ZNode（ZooKeeper Node）节点。所有的ZNode节点通过树的目录结构按照层次关系组织在一起，构成一棵ZNode树。

每个ZNode节点都用一个完整路径来唯一标识，完整路径以“/”（斜杠）符号分隔，而且每个ZNode节点都有父节点（根节点除外）。例如，“/foo/bar”表示一个ZNode节点，它的父节点为“/foo”节点，祖父节点的路径为“/”。“/”节点是ZNode树的根节点，没有父节点。

通过ZNode树，ZooKeeper提供一个多层级的树状命名空间。该树状命名空间与文件目录系统中的目录树有所不同，这些ZNode节点可以保存二进制负载数据（Payload）。文件系统目录树中的目录只能存放路径信息，而不能存放负载数据。

一个节点的负载数据（Payload）能放多少二进制数据呢？ZooKeeper为了保证高吞吐和低延迟，整个树状的目录结构全部都放在内存中。与硬盘和其他的外存设备相比，机器的内存比较有限，使得

ZooKeeper的目录结构不能用于存放大量的数据。ZooKeeper官方的要求是，每个节点存放的Payload负载数据的上限仅仅为1MB。

13.2.2 zkCli客户端指令清单

用客户端命令zkCli.cmd (zkCli.sh) 连接上ZooKeeper服务后，用help能列出所有指令，大致如表13-1所示。

表13-1 zk客户端常用的指令

zk 客户端常用的指令	功能简介
Create	创建 ZNode 路径节点
Ls	查看路径下的所有节点
Get	获得节点上的值
Set	修改节点上的值
Delete	删除节点

比如，使用get指令可以查看ZNode树的根节点“/”，大致的输出如下：

```
[zk: 127.0.0.1:2181(CONNECTED) 1] get /  
cZxid = 0x0  
ctime = Thu Jan 01 08:00:00 CST 1970  
mZxid = 0x0  
mtime = Thu Jan 01 08:00:00 CST 1970  
pZxid = 0x400000193  
cversion = 1  
dataVersion = 0  
aclVersion = 0
```

```
ephemeralOwner = 0x0  
dataLength = 0  
numChildren = 3
```

get指令所返回的节点信息主要有事务ID、时间戳、版本号、数据长度、子节点数量等，比较复杂的是事务ID和版本号。事务ID记录着节点的状态，ZooKeeper状态的每一次改变都对应着一个递增的事务ID（Transaction id），该ID称为Zxid，它是全局有序的，每次ZooKeeper的更新操作都会产生一个新的Zxid。Zxid不仅仅是一个唯一的事务ID，它还具有递增性。比如，有两个Zxid，并且Zxid1<Zxid2，就说明Zxid1变化事件发生在Zxid2变化之前。

一个ZNode的建立或者更新都会产生一个新的Zxid值，所以在节点信息中保存了3个Zxid事务ID值，分别如下：

- (1) cZxid：ZNode节点创建时的事务ID（Transaction id）。
- (2) mZxid：ZNode节点修改时的事务ID，与子节点无关。
- (3) pZxid：ZNode节点的子节点的最后一次创建或者修改时间，与孙子节点无关。

get指令所返回的节点信息包含的时间戳有两个：

- (1) ctime：ZNode节点创建时的时间戳。
- (2) mtime：ZNode节点最新一次更新发生时的时间戳。

get指令所返回的节点信息，包含的版本号有三个：

- (1) dataversion: 数据版本号。
- (2) cversion: 子节点版本号。
- (3) aclversion: 节点的ACL权限修改版本号。

对节点的每次操作都会使节点相应的版本号增加。

ZNode节点信息的主要属性如表13-2所示。

表13-2 ZNode节点信息的主要属性

属性名称	说 明
cZxid	创建节点时的 Zxid 事务 ID
ctime	创建节点时的时间
mZxid	最后修改节点时的事务 ID
mtime	最后修改节点时的时间
pZxid	表示该节点的子节点列表最后一次修改的事务 ID, 添加子节点或删除子节点就会影响 pZxid 的值, 但是修改子节点的数据内容则不影响该 ID
cversion	子节点版本号, 子节点每次修改版本号加 1
dataversion	数据版本号, 数据每次修改该版本号加 1
aclversion	权限版本号, 权限每次修改该版本号加 1
dataLength	该节点的数据长度
numChildren	该节点拥有子节点的数量

在实际开发过程中, 使用zkCli客户端指令去查看ZNode节点的效率不是太高, 可以借助一下开源客户端工具。笔者在疯狂创客圈的网盘上传了一个简单的图形化工具, 该工具的名称为ZooViewer, 其连接ZooKeeper成功之后的界面如图13-2所示。

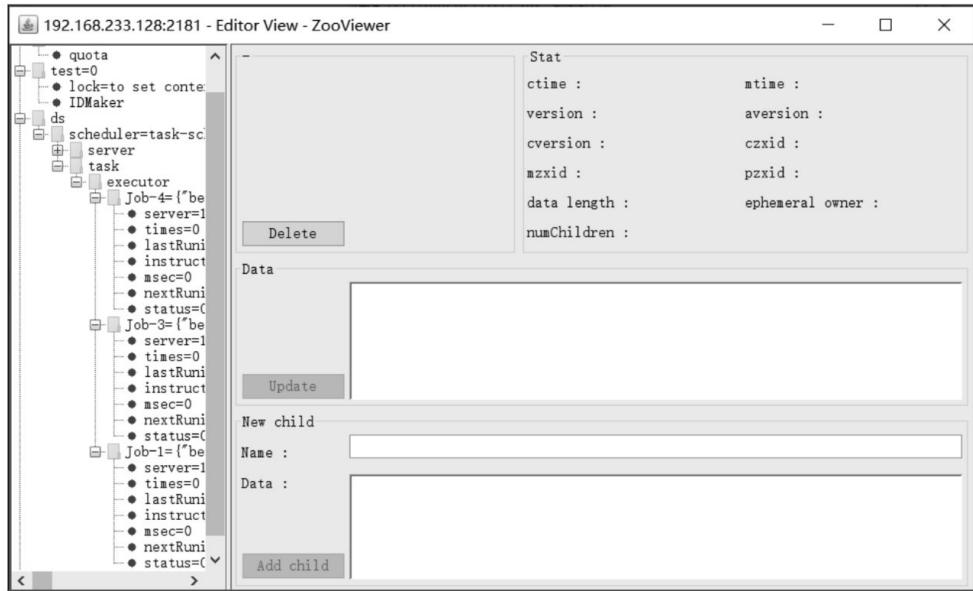


图13-2 使用ZooViewer小工具查看ZooKeeper中的ZNode信息

13.3 ZooKeeper应用开发实战

ZooKeeper应用开发主要通过Java客户端API去连接和操作ZooKeeper集群。可供选择的Java客户端API有：

(1) ZooKeeper官方的Java客户端API。

(2) 第三方的Java客户端API。

ZooKeeper官方的客户端API提供了基本的操作，比如创建会话、创建节点、读取节点、更新数据、删除节点和检查节点是否存在等。对于实际开发来说，ZooKeeper官方API有一些不足之处，具体如下：

(1) ZooKeeper的Watcher监测是一次性的，每次触发之后都需要重新进行注册。

(2) Session超时之后没有实现重连机制。

(3) 异常处理烦琐，ZooKeeper提供了很多异常，对于开发人员来说可能根本不知道该如何处理这些异常信息。

(4) 只提供了简单的byte[]数组类型的接口，没有提供Java POJO级别的序列化数据处理接口。

(5) 创建节点时如果节点存在抛出异常，就需要自行检查节点是否存在。

(6) 无法实现级联删除。

总之，ZooKeeper官方API的功能比较简单，在实际开发过程中比较笨重，一般不推荐使用。可以使用的第三方开源客户端API主要有ZkClient和Curator。

13.3.1 ZkClient开源客户端

ZkClient是一个开源客户端，在ZooKeeper原生API接口的基础上进行了包装，更便于开发人员使用。ZkClient客户端在一些著名的互联网开源项目中得到了应用，比如阿里的分布式Dubbo框架，集成了ZkClient客户端。

开源客户端ZkClient解决了ZooKeeper原生客户端API接口的很多问题。比如，ZkClient提供了更加简洁的API，实现了Session会话超时重连、Watcher反复注册等问题。尽管如此，ZkClient也有它自身的不少不足之处，具体如下：

- (1) ZkClient社区不活跃，文档不够完善，几乎没有参考文档。
- (2) 异常处理简化（抛出RuntimeException）。
- (3) 重试机制比较难用。
- (4) 没有提供各种使用场景的参考实现。

介于ZkClient的以上不足，本书不对ZkClient的使用做详细介绍。

13.3.2 Curator开源客户端

Curator是Netflix公司开源的一套ZooKeeper客户端框架。和ZkClient一样，Curator提供了非常底层的细节开发工作，包括Session会话超时重连、掉线重连、反复注册Watcher和NodeExistsException异常等。

Curator是Apache基金会的顶级项目之一，具有更加完善的文档，另外还提供了一套易用性和可读性更强的Fluent风格的客户端API框架。

Curator还提供了ZooKeeper一些比较普遍的分布式开发的开箱即用的解决方案，比如Recipes、共享锁服务、Master选举机制和分布式计算器等。Java应用开发时在这些小的组件上可以不用重复造轮子。

另外，Curator提供了一套非常优雅的链式调用API。总之，与ZkClient客户端API相比，Curator的API优雅太多，下面以创建ZNode节点为例为大家对比一下。

使用ZkClient客户端创建ZNode节点的代码为：

```
ZkClient client =  
    new ZkClient("192.168.1.105:2181",  
                10000, 10000, new SerializableSerializer());  
    //根节点路径  
    String PATH = "/test";  
    //判断是否存在  
    boolean rootExists = zkClient.exists(PATH);  
    //如果存在，获取地址列表  
    if (!rootExists) {
```

```
        zkClient.createPersistent(PATH);  
    }  
  
    String zkPath = "/test/node-1";  
    boolean serviceExists = zkClient.exists(zkPath);  
  
    if(!serviceExists){  
        zkClient.createPersistent(zkPath);  
    }  
}
```

使用Curator客户端创建节点的代码如下：

```
CuratorFramework client =  
    CuratorFrameworkFactory.newClient(connectionString,  
    retryPolicy);  
  
String zkPath = "/test/node-1";  
client.create().withMode(mode).forPath(zkPath);
```

总之，尽管Curator不是官方的客户端，但是由于Curator客户端的确非常优秀，就连ZooKeeper的作者Patrick Hunt都对Curator给予了高度评价，他的评语是：“Guava is to Java what Curator is to ZooKeeper”。

在实际的开发场景中，使用Curator客户端，就足以应付日常的ZooKeeper集群操作需求。对于ZooKeeper的客户端，我们这里只学习和研究Curator的使用，最终的疯狂创客圈社群的高并发“CrazyIM实战项目”也通过Curator客户端来操作ZooKeeper集群。

13.3.3 准备Curator开发环境

打开Curator的官网，我们可以看到Curator包含了以下几个包：

- (1) curator-framework：对ZooKeeper底层API的一些封装。
- (2) curator-client：提供一些客户端的操作，例如重试策略等。
- (3) curator-recipes：封装了一些高级特性，如Cache事件监听、选举、分布式锁、分布式计数器、分布式Barrier等。

以上三个包在使用之前首先在Maven的pom文件中依赖坐标。这里使用的Curator版本为4.0.0，与之对应ZooKeeper的版本为3.4.x。pom文件的依赖代码如下：

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-client</artifactId>
    <version>4.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.ZooKeeper</groupId>
            <artifactId>ZooKeeper</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.apache.curator</groupId>
```

```
<artifactId>curator-framework</artifactId>
<version>4.0.0</version>
<exclusions>
    <exclusion>
        <groupId>org.apache.ZooKeeper</groupId>
        <artifactId>ZooKeeper</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>4.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.ZooKeeper</groupId>
            <artifactId>ZooKeeper</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

说明

如果Curator与ZooKeeper的版本不是相互匹配的，就会有兼容性问题，很有可能导致节点操作失败。如何确保Curator与

ZooKeeper具体的版本是匹配的呢？可以去Curator的官网查看具体的配套关系。

13.3.4 创建Curator客户端实例

在使用Curator-framework组件操作ZooKeeper前，首先要创建一个客户端实例——这是一个CuratorFramework类型的对象。有两种方法可以创建该实例：

(1) 使用工厂类CuratorFrameworkFactory的静态newClient()方法。

(2) 使用工厂类CuratorFrameworkFactory的静态builder()构造者方法。

下面使用上面的两种方法来创建一下Curator客户端实例，代码如下：

```
/**
 * create by 尼恩 @ 疯狂创客圈
 */
public class ClientFactory
{
    /**
     * 方式一
     * @param connectionString ZK的连接地址
     * @return CuratorFramework 实例
     */
}
```

```
public static CuratorFramework createSimple(
    String connectionString) {
    //重试策略:第一次重试等待1秒, 第二次重试等待2秒, 第三次重试等待4
    秒
    //第一个参数: 等待时间的基础单位, 单位为毫秒
    //第二个参数: 最大重试次数
    ExponentialBackoffRetry retryPolicy =
        new ExponentialBackoffRetry(1000, 3);

    //调用工厂类CuratorFrameworkFactory的静态newClient()方法
    //第一个参数: ZK的连接地址
    //第二个参数: 重试策略
    return CuratorFrameworkFactory.newClient(
        connectionString,
        retryPolicy);
}
```

```
/**方式二
 * @param connectionString      ZK的连接地址
 * @param retryPolicy          重试策略
 * @param connectionTimeoutMs 连接超时时间
 * @param sessionTimeoutMs    会话超时时间
 * @return CuratorFramework 实例
 */
public static CuratorFramework createWithOptions(
    String connectionString, retryPolicy,
    int connectionTimeoutMs, int sessionTimeoutMs)
```

```
{  
    //调用工厂类CuratorFrameworkFactory的静态builder()构造者方法  
  
    //调用builder 模式创建 CuratorFramework 实例  
    return CuratorFrameworkFactory.builder()  
        .connectString(connectionString)  
        .retryPolicy(retryPolicy)  
  
.connectionTimeoutMs(connectionTimeoutMs)  
    .sessionTimeoutMs(sessionTimeoutMs)  
    //其他的创建选项  
    .build();  
}  
}
```

这里用到两种创建CuratorFramework客户端实例的方式：前一个通过newClient()函数去创建，相当于一个简化版本，只需要设置ZK集群的连接地址和重试策略；后一个通过CuratorFrameworkFactory.builder()函数去创建，相当于一个复杂的版本，可以设置连接超时connectionTimeoutMs、会话超时sessionTimeoutMs等其他的会话创建选项。

这里将两种创建客户端的方式封装成一个通用的ClientFactory连接工具类，大家可以直接使用。

13.3.5 通过Curator创建节点

通过Curator框架创建ZNode节点，调用create()方法即可。create()方法不需要传入ZNode的节点路径，所以并不会立即创建节点，仅仅返回一个CreateBuilder构造者实例。

通过该CreateBuilder构造者实例可以设置创建节点时的一些行为参数，最终通过构造者实例的forPath(String znodePath, byte[] payload)方法去完成真正的节点创建工作。一般来说，可以使用链式调用完成节点的创建。在链式调用的最后，需要调用forPath()方法带上需要创建的节点路径，具体的代码如下：

```
/**  
 * 创建节点  
 */  
  
@Test  
public void createNode() {  
    //客户端实例  
    CuratorFramework client =  
        ClientFactory.createSimple(ZK_ADDRESS);  
  
    try {  
        //启动客户端实例，连接服务器  
        client.start();  
  
        //创建一个 zNode 节点  
        //节点的数据为 payload  
  
        String data = "hello";
```

```
byte[] payload = data.getBytes("UTF-8");

String zkPath = "/test/CRUD/node-1";

client.create()

    .creatingParentsIfNeeded()

    .withMode(CreateMode.PERSISTENT)

    .forPath(zkPath, payload);

} catch (Exception e) {

    e.printStackTrace();

} finally {

    CloseableUtils.closeQuietly(client);

}

}
```

在上面的代码中，在链式调用的forPath()创建节点之前，通过该CreateBuilder构造者实例的withMode()方法设置了节点的类型为CreateMode.PERSISTENT类型，表示节点的类型为持久化节点。

ZooKeeper节点有四种类型，具体定义和联系如下：

(1) 持久化节点 (PERSISTENT)。所谓持久节点，是指在节点创建后就一直存在，直到有删除操作来主动清除这个节点。持久节点的生命周期是永久有效的，不会因为创建该节点的客户端会话失效而消失。

(2) 持久化顺序节点 (PERSISTENT_SEQUENTIAL)。这类节点的生命周期和持久节点是一致的。额外的特性是，持久化顺序节点的每个父节点会为它的第一级子节点维护一份次序，会记录每个子节点创

建的先后顺序。如果在创建子节点的时候可以设置这个属性，那么在创建节点的过程中ZK会自动为给定节点名加上一个表示次序的数字后缀作为新的节点名。这个次序后缀的最大值可以是整数类型的最大值。比如，在创建持久化顺序节点的时候只需要传入节点“/test_”，之后ZooKeeper就会自动给“test_”后面补充数字次序。

(3) 临时节点 (EPHEMERAL)。与持久节点不同的是，临时节点的生命周期和客户端会话绑定。也就是说，如果客户端会话失效，那么这个节点就会自动被清除掉。注意，这里提到的是会话失效，而非连接断开。这里还要注意一件事，就是当客户端会话失效后，所产生的临时节点不是一下子就消失，要过一段时间，大概是10秒以内。可以试着在本机操作生成节点，在服务端用命令来查看当前的节点数目，当客户端停下来后，我们就会发现临时节点短时间之内还在。

另外，在临时节点下面不能创建子节点。

(4) 临时顺序节点 (EPHEMERAL_SEQUENTIAL)。此节点属于临时节点，不过带有顺序编号，客户端会话结束节点就会消失。

13.3.6 通过Curator读取节点

在Curator框架中，与节点读取的有关方法主要有三个：

- (1) 调用checkExists()方法判断节点是否存在。
- (2) 调用getData()方法获取节点的数据。
- (3) 调用getChildren()方法获取子节点列表。

演示代码如下：

```
/**  
 * 读取节点  
 */  
  
@Test  
  
public void readNode() {  
    //创建客户端  
    CuratorFramework client =  
        ClientFactory.createSimple(ZK_ADDRESS);  
  
    try {  
        //启动客户端实例，连接服务器  
        client.start();  
  
        String zkPath = "/test/CRUD/node-1";  
  
        Stat stat = client.checkExists().forPath(zkPath);  
  
        if (null != stat) {  
            //读取节点的数据  
            byte[] payload = client.getData().forPath(zkPath);  
            String data = new String(payload, "UTF-8");  
            log.info("read data:", data);  
  
            String parentPath = "/test";  
            List<String> children =  
  
                client.getChildren().forPath(parentPath);  
  
            for (String child : children) {  
                System.out.println("child: " + child);  
            }  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

```
        log.info("child:", child);

    }

}

} catch (Exception e) {
    e.printStackTrace();
} finally {
    CloseableUtils.closeQuietly(client);
}

}
```

无论是checkExists()、getData()还是getChildren()方法都有一个共同的特点：

- (1) 这些方法返回的是构造者实例，不会立即执行。
- (2) 通过构造者实例的链式调用为自己增加具体的操作，在调用末端使用forPath(String znodePath)方法，在节点上去执行实际的操作。

13.3.7 通过Curator更新节点

节点的更新操作可以分为同步更新与异步更新。同步更新就是更新线程是阻塞的，一直阻塞到更新操作执行完成。异步更新就是更新线程是非阻塞的，调用后立即返回，真正的更新操作异步去执行完成。

调用setData()方法进行同步更新，代码如下：

```
/**
 * 同步更新节点
 */
@Test
public void updateNode() {
    //创建客户端
    CuratorFramework client =
        ClientFactory.createSimple(ZK_ADDRESS);
    try {
        //启动客户端实例，连接服务器
        client.start();
        String data = "hello world";
        byte[] payload = data.getBytes("UTF-8");
        String zkPath = "/test/CRUD/node-1";
        client.setData()
            .forPath(zkPath, payload);

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        CloseableUtils.closeQuietly(client);
    }
}
```

在上面的代码中，通过setData()方法返回一个SetDataBuilder构造者实例，执行该实例的forPath(zkPath, payload)方法，完成同步

更新操作。

如果需要进行异步更新，如何处理呢？其实很简单：通过 SetDataBuilder构造者实例的inBackground(AsyncCallback callback)方法设置一个 AsyncCallback回调实例。简简单单的一个函数就将更新数据的行为从同步执行变成了异步执行。异步执行完成后，SetDataBuilder构造者实例会再执行 AsyncCallback实例的 processResult()方法中的回调逻辑，完成更新后的其他操作。异步更新的代码如下：

```
package com.crazymakercircle.zk.basicOperate;  
//省略import  
  
public class CRUD {  
    private static final String ZK_ADDRESS = "127.0.0.1:2181";  
  
    //省略其他  
    /**  
     * 更新节点 - 异步模式  
     */  
    @Test  
    public void updateNodeAsync() {  
        //创建客户端  
        CuratorFramework client =  
ClientFactory.createSimple(ZK_ADDRESS);  
        try {  
  
            //异步更新完成，回调此实例
```

```
        AsyncCallback.StringCallback callback =
                new
        AsyncCallback.StringCallback() {
            //回调方法
            @Override
            public void processResult(int i, String s,
                                      Object o,
String s1) {
                System.out.println(
                        "i = " + i + " | " +
                        "s = " + s + " | " +
                        "o = " + o + " | " +
                        "s1 = " + s1
                );
            }
        };
        //启动客户端实例，连接服务器
        client.start();

        String data = "hello ,every body! ";
        byte[] payload = data.getBytes("UTF-8");
        String zkPath = "/test/CRUD/remoteNode-1";
        client.setData()
                .inBackground(callback)    //设置回调实例
                .forPath(zkPath, payload);

        Thread.sleep(10000);
    }
}
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            CloseableUtils.closeQuietly(client);
        }
    }
}
```

13.3.8 通过Curator删除节点

删除节点非常简单，只需调用`delete()`方法即可，实例代码如下：

```
package com.crazymakercircle.zk.basicOperate;
//省略import

public class CRUD {
    private static final String ZK_ADDRESS = "127.0.0.1:2181";

    //省略其他

    @Test
    public void deleteNode() {
        //创建客户端
        CuratorFramework client =
ClientFactory.createSimple(ZK_ADDRESS);

        try {
            //启动客户端实例，连接服务器

```

```
client.start();

//删除节点

String zkPath = "/test/CRUD/remoteNode-1";

client.delete().forPath(zkPath);

//删除后查看结果

String parentPath = "/test";
List<String> children =

client.getChildren().forPath(parentPath);

for (String child : children) {
    log.info("child:", child);
}

} catch (Exception e) {
    e.printStackTrace();
} finally {
    CloseableUtils.closeQuietly(client);
}
}
```

在上面的代码中，通过delete()方法返回一个执行删除操作的DeleteBuilder构造者实例，执行该实例的forPath(zkPath, payload)方法，完成同步删除操作。

删除和更新操作一样，也可以异步进行。那么如何异步删除呢？可以使用DeleteBuilder构造者实例的inBackground(AsyncCallback

asyncCallback)方法去设置删除之后的回调实例，实际操作很简单，这里不再赘述。

至此，Curator的CRUD操作已经介绍完成，下面介绍基于Curator的基本操作来完成一些基础的分布式应用。

13.4 分布式命名服务实战

命名服务是为系统中的资源提供标识能力。ZooKeeper的命名服务主要是利用ZooKeeper节点的树形分层结构和子节点的次序维护能力为分布式系统中的资源命名。

典型的分布式命名服务场景如下。

1. 分布式API目录

为分布式系统中各种API接口服务的名称、链接地址提供类似JNDI（Java命名和目录接口）中文件系统的能力。借助于ZooKeeper的树形分层结构就能提供分布式的API调用的能力。

典型的应用著名的Dubbo分布式框架就是应用了ZooKeeper分布式的JNDI能力。在Dubbo中，使用ZooKeeper维护全局的服务接口API地址列表。大致的思路为：

(1) 服务提供者（Provider）在启动的时候向ZK上的指定节点写入自己的API地址，这个操作就相当于服务的公开。类似的API地址节点如下：

```
/dubbo/${serviceName}/providers
```

(2) 服务消费者（Consumer）启动的时候，订阅节点/dubbo/{serviceName}/providers下的Provider服务提供者URL地址，获得所有访问提供者的API。

2. 分布式ID生成器

在分布式系统中，为每一个数据资源提供唯一的ID标识能力。在单体服务环境下，通常可以利用数据库的主键自增功能唯一标识一个数据资源。但是，在大量服务器集群的场景下，依赖单体服务的数据库主键自增生成唯一ID的方式没有办法满足高并发和高负载的需求。这时就需要分布式的ID生成器，保障分布式场景下的ID唯一性。

3. 分布式节点的命名

一个分布式系统通常会由很多节点组成，而且节点的数量不是固定的，是不断动态变化的。比如说，当业务不断膨胀和流量洪峰到来时，可能会动态加入大量的节点到集群中。一旦流量洪峰过去，就需要下线大量的节点。再比如说，由于机器或者网络的原因，一些节点会主动离开集群。

如何为大量的动态节点命名呢？一种简单的办法是，通过配置文件手动进行每一个节点的命名。如果节点数据量太大，或者说变动频繁，手动命名是不现实的，这就需要用到分布式节点的命名服务。

疯狂创客圈的高并发“CrazyIM实战项目”也会使用分布式命名服务为每一个IM节点动态命名。

上面列举了三个分布式的命名服务场景，实际上需要用到分布式资源标识能力的场景远不止这些，这里只是抛砖引玉。

13.4.1 ID生成器

在分布式系统中，分布式ID生成器的使用场景非常多：

- (1) 大量的数据记录，需要分布式ID。

- (2) 大量的系统消息，需要分布式ID。
- (3) 大量的请求日志，如RESTful的操作记录，需要唯一标识，以便进行后续的用户行为分析和调用链路分析。
- (4) 分布式节点的命名服务，往往也需要分布式ID。

传统的数据库自增主键，或者单体Java应用的自增主键，已经不能满足分布式ID生成器的需求。在分布式系统环境中，迫切需要一种全新的唯一ID系统，这种系统需要满足以下需求：

- (1) 全局唯一：不能出现重复ID。
- (2) 高可用：ID生成系统是非常基础的系统，被许多关键系统调用，一旦宕机，就会造成严重影响。

分布式的ID生成器方案大致如下：

- (1) Java的UUID。
- (2) 分布式缓存Redis生成ID，利用Redis的原子操作INCR和INCRBY生成全局唯一的ID。
- (3) Twitter的Snowflake算法。
- (4) ZooKeeper生成ID，利用ZooKeeper的顺序节点生成全局唯一的ID。
- (5) MongoDB的ObjectId。MongoDB是一个分布式的非结构化NoSQL数据库，每插入一条记录，就会自动生成全局唯一的“_id”字

段值，该值是一个12字节的字符串，可以作为分布式系统中全局唯一的ID。

以上几种方案有哪些利弊呢？首先，分析一下Java语言中的UUID方案。UUID（Universally Unique Identifier）是在一定的范围内（从特定的名字空间到全球）唯一的机器生成的标识符，所以UUID在其他语言中也叫GUID。

在Java中，生成UUID的代码很简单，代码如下：

```
String uuid = UUID.randomUUID().toString()
```

UUID经由一定的算法由机器生成。为了保证UUID的唯一性，规范定义了网卡MAC地址、时间戳、名字空间（Namespace）、随机或伪随机数、时序等元素，以及从这些元素生成UUID的算法。

一个UUID是16字节长的数字，一共128位。转成字符串之后，会变成一个36字节的字符串，比如3F2504E0-4F89-11D3-9A0C-0305E82C3301。使用的时候，可以把中间的4个“-”去掉，剩下32字节的字符串。

UUID的优点：本地生成ID，不需要进行远程调用，时延低，性能高。

UUID的缺点：16字节128位过长，通常以36位的字符串表示，很多场景不适用。比如，UUID没有排序，无法保证趋势递增，用作数据库索引字段的效率就很低，新增记录存储入库的时的性能差。

对于高并发和数据量的系统，不建议使用UUID。

13.4.2 ZooKeeper分布式ID生成器的实战案例

ZK的四种节点中有两种节点具备自动编号的能力：持久化顺序节点和临时顺序节点。

ZooKeeper的每一个节点都会为它的第一级子节点维护一份顺序编号，会记录每个子节点创建的先后顺序。这个是分布式同步的，也是全局唯一的。

在创建子节点的时候，如果设置为上面的类型，ZK会自动为创建后的节点路径末尾加上一个数字，用来表示次序。这个次序范围是整数类型的最大值。

比如，在创建节点的时候只需要传入节点“/test_”，ZooKeeper就会自动给“test_”后面补充数字次序，比如“/test_0000000010”。

通过创建ZK临时顺序节点的方法生成全局唯一ID的演示代码，大致如下：

```
package com.crazymakercircle.zk.NameService;  
//省略import  
  
public class IDMaker {  
    //省略其他方法  
  
    /**  
     * 创建临时顺序节点  
     * @param pathPefix 节点路径  
     * @return 创建后的完整路径名称
```

```
*/  
  
private String createSeqNode(String pathPefix) {  
  
    try {  
  
        //创建一个zNode 顺序节点  
  
        //为了避免ZooKeeper的顺序节点暴增，建议创建后直接删除创建的  
        //节点  
  
        String destPath = client.create()  
            .creatingParentsIfNeeded()  
            .withMode(CreateMode.EPHEMERAL_SEQUENTIAL)  
            .forPath(pathPefix);  
  
        return destPath;  
  
    } catch (Exception e) {  
  
        e.printStackTrace();  
  
    }  
  
    return null;  
}  
  
  
//获取ID值  
  
public String makeId(String nodeName) {  
  
    String str = createSeqNode(nodeName);  
  
    if (null == str) {  
  
        return null;  
    }  
  
    //取得ZK节点的末尾序号  
  
    int index = str.lastIndexOf(nodeName);  
  
    if (index >= 0) {  
  
        index += nodeName.length();  
    }  
}
```

```
        return index <= str.length() ? str.substring(index)
: "";
    }
    return str;
}
}
```

节点创建完成后，会返回节点完整的层次路径，所生成的序号处于路径的末尾，一般为10位数字字符，例如：

```
/test/IDMaker/ID-0000000001
```

可以截取路径末尾的数字作为新生成的ID。自制IDMaker的单元测试用例，代码如下：

```
@Slf4j
public class IDMakerTester {

    @Test
    public void testMakeId() {
        IDMaker idMaker = new IDMaker();
        idMaker.init();
        String nodeName = "/test/IDMaker/ID-";
        for (int i = 0; i < 10; i++) {
            String id = idMaker.makeId(nodeName);
            log.info("第" + i + "个创建的id为：" + id);
        }
    }
}
```

```
        }

        idMaker.destroy();

    }

//省略其他用例

}
```

下面是部分的运行输出：

第0个创建的id为:0000000010

第1个创建的id为:0000000011

//省略其他输出

13.4.3 集群节点的命名服务的实战案例

前面讲到，在分布式集群中可能需要部署的大量机器节点。在节点少时，节点的命名可以手工完成。在节点数量大的场景下，手工命名维护成本高，还需要考虑到自动部署、运维等问题，手工去命名不现实。总之，节点的命名最好由系统自动维护。

节点的命名主要是为节点进行唯一编号，其主要诉求是不同节点的编号绝对不能重复。一旦编号重复，就会有不同的节点碰撞，导致集群异常。

有以下两个方案可生成集群节点编号：

(1) 使用数据库的自增ID特性，用数据表存储机器的MAC地址或者IP来维护。

(2) 使用ZooKeeper持久顺序节点的次序特性来维护节点的NodeID编号。

这里为大家介绍的是第二种。在第二种方案中，集群节点命名服务的基本流程是：

(1) 启动节点服务，连接ZooKeeper，检查命名服务根节点是否存在，如果不存在就创建系统根节点。

(2) 在根节点下创建一个临时顺序ZNode节点，取回ZNode的编号，作为分布式系统中节点的NodeID。

(3) 如果临时节点太多，可以根据需要删除临时顺序ZNode节点。

基于ZooKeeper集群节点的命名服务的代码实现，主要代码如下：

```
package com.crazymakercircle.zk.NameService;  
//省略import  
public class SnowflakeIdWorker  
{  
  
    //ZK客户端  
    transient private CuratorFramework zkClient = null;  
  
    //工作节点的路径  
    private String pathPrefix = "/test/IDMaker/worker-";  
    private String pathRegistered = null;  
    //保持节点id，不需要每次计算
```

```
private Long nodeId = null;

public static SnowflakeIdWorker instance = new
SnowflakeIdWorker();

private SnowflakeIdWorker()
{
    this.zkClient = ZKclient.instance.getClient();
    this.init();
}

//应用启动的时候，在zooKeeper中创建顺序临时节点
public void init()
{
    //创建一个 zNode 节点，节点的 payload 为当前worker 实例
    try
    {
        byte[] payload = pathPrefix.getBytes();
        //创建一个非持久化的临时节点，其前缀需要提前定义
        pathRegistered = zkClient.create()
            .creatingParentsIfNeeded()
            .withMode(CreateMode.EPHEMERAL_SEQUENTIAL)
            .forPath(pathPrefix, payload);
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

```
}

/**
 * 获取节点ID
 */
public long getNodeID()
{
    if (null != nodeId) return nodeId;

    String sid = null;
    if (null == pathRegistered)
    {
        throw new RuntimeException("节点注册失败");
    }

    int index = pathRegistered.lastIndexOf(pathPrefix);
    if (index >= 0)
    {
        index += pathPrefix.length();
        sid = index <= pathRegistered.length() ?
            pathRegistered.substring(index) :
            null;
    }

    if (null == sid)
    {
        throw new RuntimeException("节点ID生成失败");
    }

    nodeId = Long.parseLong(sid);
    return nodeId;
}
```

```
    }  
}  
}
```

13.4.4 结合ZooKeeper实现SnowFlake ID算法

Twitter的SnowFlake算法是一种著名的分布式服务器用户ID生成算法。首先来介绍一下SnowFlake ID的组成。

1. SnowFlake ID的组成

SnowFlake算法所生成的ID是一个64位的长整数类型数字，被划分成四部分，其中后面三部分分别表示时间戳、机器编码、序列号，如图13-3所示。

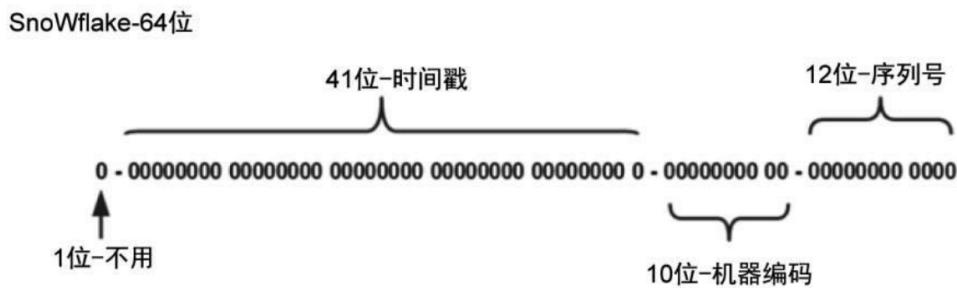


图13-3 SnowFlake ID的四部分

- (1) 第一位：占用1位，始终是0，没有实际作用。
- (2) 时间戳：占用41位，精确到毫秒，总共可以容纳约69年的时间。
- (3) 机器编码：占用10位，最多可以容纳1024个节点。

(4) 序列号：占用12位，最多可以累加到4095。这个值在同一毫秒同一节点上从0开始不断累加。

总体来说，在工作节点达到1024顶配的场景下，SnowFlake算法在同一毫秒内最多可以生成的ID数量为 $1024 \times 4096 = 4194304$ ，总计400多万个。也就是说，在绝大多数并发场景下都是够用的。

SnowFlake ID的第三个部分是机器编码，可以结合前面的命名方法，通过ZooKeeper管理NodeId，免去手动频繁修改集群节点去配置机器ID的麻烦。

上面的SnowFlake ID的位数分配只是一个官方的推荐，实际使用时是可以微调的。例如，1024的节点数不够，可以增加3位，扩大到8192个；每毫秒生成4096个ID比较多，可以从12位减小到10位，则单个节点每毫秒生成1024个ID，1秒可以生成 1024×1000 个ID数，其数量也是巨大的；剩下的位数为剩余时间，还剩下40位时间戳，如果调整为比原来少1位，则可以持续32年。

2. SnowFlake ID的实现

按照上面的SnowFlake ID组成规则实现一下SnowFlake算法，代码如下：

```
package com.crazymakercircle.zk.NameService;

/**
 * snowflake ID 算法实现
 * create by 尼恩 @ 疯狂创客圈
 */
```

```
public class SnowflakeIdGenerator {

    /**
     * 单例
     */
    public static SnowflakeIdGenerator instance =
        new SnowflakeIdGenerator();

    /**
     * 初始化单例
     */
    public synchronized void init(long workerId) {
        if (workerId > MAX_WORKER_ID) {
            //ZK分配的workerId过大
            throw new IllegalArgumentException(
                "woker Id wrong: " +
                workerId);
        }
        instance.workerId = workerId;
    }

    private SnowflakeIdGenerator() {
    }

    /**
     * 开始使用该算法的时间为2017-01-01 00:00:00
     */
}
```

```
private static final long START_TIME = 1483200000000L;

/***
 * worker id 的位数，最多支持8192个节点
 */

private static final int WORKER_ID_BITS = 13;

/***
 * 序列号，支持单节点最高每毫秒的最大ID数1024
 */

private final static int SEQUENCE_BITS = 10;

/***
 * 最大的 worker id, 8091
 * -1 的补码（二进制全1）右移13位，然后取反
 */

private final static long MAX_WORKER_ID = ~(-1L <<
WORKER_ID_BITS);

/***
 * 最大的序列号，1023
 * -1 的补码（二进制全1）右移10位，然后取反
 */

private final static long MAX_SEQUENCE = ~(-1L <<
SEQUENCE_BITS);

/***
```

```
* worker 节点编号的移位
*/
private final static long APP_HOST_ID_SHIFT =
SEQUENCE_BITS;

/***
 * 时间戳的移位
*/
private final static long TIMESTAMP_LEFT_SHIFT =
    WORKER_ID_BITS +
APP_HOST_ID_SHIFT;

/***
 * 该项目的worker 节点 id
*/
private long workerId;

/***
 * 上次生成ID的时间戳
*/
private long lastTimestamp = -1L;

/***
 * 当前毫秒生成的序列
*/
private long sequence = 0L;
```

```
/***
 * Next id long.
 *
 * @return the nextId
 */
public Long nextId() {
    return generateId();
}

/***
 * 生成唯一ID的具体实现
 */
private synchronized long generateId() {
    long current = System.currentTimeMillis();

    if (current < lastTimestamp) {
        //如果当前时间小于上一次ID生成的时间戳
        //说明系统时钟回退过，出现问题返回-1，生成ID失败
        return -1;
    }

    if (current == lastTimestamp) {
        //如果当前生成ID的时间还是上次的时间，那么对sequence序列号
        //进行+1
        sequence = (sequence + 1) & MAX_SEQUENCE;
        if (sequence == MAX_SEQUENCE) {
            //当前毫秒生成的序列数已经大于最大值
    }
}
```

```
//那么阻塞到下一毫秒再获取新的时间戳
current = this.nextMs(lastTimestamp);
}

} else {
    //当前的时间戳已经是下一毫秒
    sequence = 0L;
}

//更新上次生成ID的时间戳
lastTimestamp = current;

//进行移位操作生成int64的唯一ID

//时间戳右移23位
long time = (current - START_TIME) <<
TIMESTAMP_LEFT_SHIFT;

//workerId 右移10位
long workerId = this.workerId << APP_HOST_ID_SHIFT;

return time | workerId | sequence; //返回ID
}

/**
 * 阻塞到下一毫秒
 */
private long nextMs(long timeStamp) {
```

```
long current = System.currentTimeMillis();

while (current <= timeStamp) {

    current = System.currentTimeMillis();

}

return current;

}

}
```

在上面的代码中，大量使用到了位运算。如果对位运算不清楚，估计很难看懂上面的代码。这里需要特别说明一下：-1的8位二进制编码为1111 1111，也就是全1。因为在8位二进制的场景下，-1的原码是1000 0001，反码是1111 1110（符号位为1、数值部分按位取反），补码是反码加1，最终-1的编码计算后的结果是全1。16位、32位、64位的-1与8位二进制相同，其二进制编码也是全为1。这就是大家从计算机基础课中所学到的知识：负数的编码为其补码。另外，这里的二进制位移算法以及二进制按位或的算法都比较简单，如果不懂，可以去查看Java的相关基础书。

上面的代码是一个相对比较简单的SnowFlake实现版本，对其中的关键算法解释如下：

(1) 在单节点上获得下一个ID，使用Synchronized控制并发，没有使用CAS的方式，是因为CAS不适合并发量非常高的场景。

(2) 如果一台机器上当前毫秒在的序列号已经增长到最大值1023，则使用while循环等待直到下一毫秒。

(3) 如果当前时间小于记录的上一个毫秒值，就说明这台机器的时间回拨了，于是阻塞，直等到下一毫秒。

说明

在生产环境下，不建议大家使用自己实现的SnowFlake算法，可以使用开源的SnowFlake算法实现，如百度uid-generator开源项目、美团ecp-uid开源项目，这些开源实现是经过了严酷的运行检验的。自己实现，仅仅是为了学习基础知识，掌握一些基础的原理。

编写一个测试用例，测试一下SnowflakeIdGenerator，代码如下：

```
package com.crazymakercircle.zk.NameService;  
//省略import  
@Slf4j  
public class SnowflakeIdTest {  
    @Test  
    public void snowflakeIdTest() {  
        //获取节点的id  
        long workId = SnowflakeIdWorker.instance.getId();  
        //初始化id生成器  
        SnowflakeIdGenerator.instance.init(workId);  
        //创建一个线程池，并生成ID  
        ExecutorService es = Executors.newFixedThreadPool(10);  
    }  
}
```

```
final HashSet idSet = new HashSet();
Collections.synchronizedCollection(idSet);
long start = System.currentTimeMillis();
log.info(" 开始生成 *");
for (int i = 0; i < 10; i++) {
    es.execute(() -> {
        for (long j = 0; j < 5000000; j++) {
            long id =
SnowflakeIdGenerator.instance.nextId();
            synchronized (idSet) {
                idSet.add(id);
            }
        }
    });
}

//关闭线程池
es.shutdown();
try {
    es.awaitTermination(10, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
long end = System.currentTimeMillis();
log.info(" 生成ID 结束");
log.info("* 耗费: " + (end - start) + " ms!");
}
```

测试用例中用到了上一小节实现的SnowflakeIdWorker节点的命令服务，并且通过它取得了节点workerId。

SnowFlake算法的优点是：

- 生成ID时不依赖于数据库，完全在内存生成，高性能，高可用。
- 容量大，每秒可生成几百万ID。
- ID呈趋势递增，后续插入数据库的索引树时性能较高。

SnowFlake算法的缺点是：

- 依赖于系统时钟的一致性，如果某台机器的系统时钟回拨，就有可能造成ID冲突或者ID乱序。
- 在启动之前，如果这台机器的系统时间回拨过，就有可能出现ID重复的危险。

13.5 分布式事件监听的重点

实现对ZooKeeper服务端节点操作事件的监听是客户端操作服务器的一项重点工作。在Curator的API中，事件监听有两种模式：第一种是标准的观察者模式，通过Watcher监听器去实现；第二种是缓存监听模式，通过引入一种本地缓存视图Cache机制去实现。第二种Cache事件监听机制可以理解为一个本地缓存视图与远程ZooKeeper视图的对比过程。简单来说，Cache在客户端缓存了ZNode的各种状态，当感知到ZooKeeper集群的ZNode状态变化会触发事件时，注册在这些事件上的监听器会处理这些事件。

虽然Cache是一种缓存机制，但是可以借助Cache实现事件的监听。另外，Cache提供了事件监听器反复注册的能力，而观察模式的Watcher监听器只能监听一次。

在类型上，Watcher监听器比较简单，只有一种。Cache事件监听的种类有三种，包括PathCache、NodeCache、TreeCache。

13.5.1 Watcher标准的事件处理器

在ZooKeeper中，接口类型Watcher用于表示一个标准的事件处理器，用来定义收到事件通知后相关的回调处理逻辑。接口类型Watcher包含KeeperState和EventType两个内部枚举类，分别代表了通知状态和事件类型。

定义回调处理逻辑需要使用Watcher接口的事件回调方法：

```
Process(WatchedEvent event)
```

定义一个Watcher的回调实例很简单，代码如下：

```
//演示：定义一个监听器
Watcher w = new Watcher() {
    @Override
    public void process(WatchedEvent watchedEvent) {
        log.info("监听器watchedEvent: " + watchedEvent);
    }
};
```

可以利用GetDataBuilder、GetChildrenBuilder、ExistsBuilder等实现Watchable<T>接口的构造者实例，通过usingWatcher(Watcher)方法为构造者实例设置Watcher监听器实例。

在Curator中，Watchable<T>接口的源码如下：

```
package org.apache.curator.framework.api;
import org.apache.zookeeper.Watcher;
public interface Watchable<T> {
    T watched();
    T usingWatcher(Watcher w);
    T usingWatcher(CuratorWatcher cw);
}
```

GetDataBuilder、GetChildrenBuilder、ExistsBuilder构造者分别通过getData()、getChildren()、checkExists()等方法返回，也就

是说，至少在以上三个方法的调用链上可以通过加上usingWatcher()方法去设置监听器，典型的代码如下：

```
//为GetDataBuilder实例设置监听器  
byte[] content = client.getData()  
.usingWatcher(w).forPath(workerPath);
```

一个Watcher监听器在向服务端完成注册后，当服务端的一些事件触发了这个Watcher时，就会向注册过的客户端会话发送一个事件通知来实现分布式的通知功能。在Curator客户端收到服务器的通知后会封装一个WatchedEvent事件实例，传递给监听器的process(WatchedEvent)回调方法。

WatchedEvent包含了三个基本属性：通知状态（KeeperState）、事件类型（EventType）和节点路径（path）。需要说明的是，WatchedEvent并不是从ZooKeeper集群直接传递过来的事件实例，而是被Curator封装过的事件实例。WatchedEvent类型没有实现序列化接口java.io.Serializable，因此不能用于网络传输。那么，被封装的从ZooKeeper服务端直接通过网络传输过来的事件实例是什么呢？一个WatcherEvent类型的实例，其名称与Curator的WatchedEvent封装实例只有一个字母之差，而且功能也是一样的，表示的是同一个服务端事件。

1. Watcher接口定义的通知状态和事件类型

这里聚焦于Curator封装过的WatchedEvent实例。WatchedEvent中所用到的通知状态和事件类型定义在Watcher接口中，具体如表13-3所示。

表13-3 Watcher接口中定义的通知状态和事件类型

KeeperState	EventType	触发条件	说 明
SyncConnected (0)	None (-1)	客户端与服务端成功建立连接	
	NodeCreated (1)	监听的对应数据节点被创建	
	NodeDeleted (2)	监听的对应数据节点被删除	此时客户端和服务器处于连接状态
	NodeDataChanged (3)	监听的对应数据节点的数据内容发生变更	
	NodeChildChanged (4)	监听的对应数据节点的子节点列表发生变更	
Disconnected (0)	None (-1)	客户端与 ZooKeeper 服务器断开连接	此时客户端和服务器处于断开连接状态
Expired (-112)	Node (-1)	会话超时	此时客户端会话失效，通常同时也会收到SessionExpiredException 异常
AuthFailed (4)	None (-1)	通常有两种情况：一是使用错误的 schema 进行权限检查，二是 SASL 权限检查失败	通常同时也会收到AuthFailedException 异常

2. Watcher使用实战

利用Watcher对节点事件进行监听，实例程序如下：

```
package com.crazymakercircle.zk.publishSubscribe;

//省略import

import java.io.UnsupportedEncodingException;
/***
 * 客户端监听实战
 */
public class ZkWatcherDemo {

    private String workerPath = "/test/listener/remoteNode";
    private String subWorkerPath =
}

```

```
"/test/listener/remoteNode/id-";

//利用watcher来对节点进行监听操作

@Test
public void testWatcher() {
    CuratorFramework client =
        ZKclient.instance.getClient();

    //检查节点是否存在，没有则创建
    boolean isExist =
        ZKclient.instance.isNodeExist(workerPath);

    if (!isExist) {
        ZKclient.instance.createNode(workerPath, null);
    }

    try {
        Watcher w = new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent)
            {
                System.out.println("监听到的变化 " + watchedEvent);
            }
        };

        byte[] content = client.getData()
            .usingWatcher(w).forPath(workerPath);
    }
}
```

```
        log.info("监听节点内容: " + new String(content));

        //第一次变更节点数据

        client.setData().forPath(workerPath,
                                  "第一次更改内

容".getBytes());

        //第二次变更节点数据

        client.setData().forPath(workerPath,
                                  "第二次更改内

容".getBytes());

        Thread.sleep(Integer.MAX_VALUE);

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

}
```

运行代码，输出的结果如下：

```
//...

监听到的变化 watchedEvent = WatchedEvent state:SyncConnected
type:NodeDataChanged path:/test/listener/node
```

以上程序中在节点路径“/test/listener/node”注册一个
Watcher监听器实例，随后调用setData方法改变该节点内容，虽然改
变了两次，但是监听器仅仅监听到了一个事件。换句话说，监听器是

注册的，是一次性的，当第二次改变节点内容时，注册已经失效，无法再次捕获节点变动事件。

既然Watcher监听器是一次性的，如果要反复使用，怎么办呢？需要反复通过构造者的usingWatcher方法去提前进行注册。所以，Watcher监听器不适用于节点的数据频繁变动或者节点频繁变动这样的业务场景，而是适用于一些特殊的、变动不频繁的场景，比如会话超时、授权失败等这样的特殊场景。

Watcher需要反复注册，比较烦琐，所以Curator引入了Cache来监听ZooKeeper服务端的事件。Cache对ZooKeeper事件监听进行了封装，能够自动处理反复注册监听。

13.5.2 NodeCache节点缓存的监听

Curator引入的Cache缓存拥有一系列的类型，包括NodeCache、PathCache、TreeCache。NodeCache节点缓存可以用于ZNode节点的监听，包括新增、删除和更新等。

1. NodeCache的使用步骤

使用NodeCache的第一步就是构造一个NodeCache缓存实例。有两个构造方法具体如下：

```
NodeCache(CuratorFramework client, String path)  
NodeCache(CuratorFramework client, String path, boolean  
dataIsCompressed)
```

第一个参数传入创建的Curator的框架客户端实例，第二个参数传入监听节点的路径，第三个重载参数dataIsCompressed表示是否对数据进行压缩。

使用NodeCache的第二步就是构造一个NodeCacheListener监听器回调实例。该接口的定义如下：

```
package org.apache.curator.framework.recipes.cache;  
public interface NodeCacheListener {  
    void nodeChanged() throws Exception;  
}
```

NodeCacheListener监听器回调接口只定义了一个简单的方法nodeChanged()，当节点变化时这个方法就会被回调到。大致的使用实例代码如下：

```
NodeCacheListener listener = new NodeCacheListener() {  
    @Override  
    public void nodeChanged(){  
        ChildData data = nodeCache.getCurrentData();  
        log.info("ZNode 节点状态改变, path={},  
data.getPath());  
        log.info("ZNode 节点状态改变, data={},  
new String(data.getData(), "Utf-8"));  
        log.info("ZNode 节点状态改变, stat={},  
data.getStat());  
    }  
};
```

第三步，在创建完NodeCacheListener的实例之后，需要将这个实例注册到nodeCache缓存实例，调用缓存实例的addListener()方法。

第四步，调用缓存实例NodeCache的start()方法，启动节点的事件监听。

```
//第三步，注册回调监听器  
nodeCache.getListenable().addListener(listener);  
  
//第四步，启动节点的事件监听  
nodeCache.start();
```

NodeCache的start()方法能进行缓存和事件监听，有两个版本：

```
void      start() //Start the cache.  
void      start(boolean buildInitial) //true代表缓存当前节点
```

start()方法唯一的一个参数buildInitial代表是否将该节点的数据立即进行缓存。如果设置为true，就在start()启动时立即调用nodeCache的getCurrentData()方法，能够得到对应节点的信息ChildData实例，如果设置为false就得不到对应的信息。

2. NodeCache事件监听的实战案例

使用NodeCache监听节点事件的完整实例代码如下：

```
package com.crazymakercircle.zk.publishSubscribe;  
  
//省略import  
  
/**  
 * 客户端监听实战
```

```
* create by 尼恩 @ 疯狂创客圈
*/
@Slf4j
@Data
public class ZkWatcherDemo {

    private String workerPath = "/test/listener/remoteNode";
    private String subWorkerPath =
"/test/listener/remoteNode/id-";

    /**
     * NodeCache 节点缓存的监听
     */
    @Test
    public void testNodeCache() {
        //检查节点是否存在，没有则创建
        boolean isExist =
ZKclient.instance.isNodeExist(workerPath);
        if (!isExist) {
            ZKclient.instance.createNode(workerPath, null);
        }
        CuratorFramework client =
ZKclient.instance.getClient();
        try {
            NodeCache nodeCache =

```

```
        new NodeCache(client, workerPath, false);

    NodeCacheListener listener = new
NodeCacheListener() {
    @Override
    public void nodeChanged()...{
        ChildData childData =
nodeCache.getCurrentData();
        log.info("ZNode节点状态改变, path={ }",
childData.getPath());
        log.info("ZNode节点状态改变, data={ }",
new String(childData.getData(),
"Utf-8"));
        log.info("ZNode节点状态改变, stat={ }",
childData.getStat());
    }
};

//启动节点的事件监听
nodeCache.getListenable().addListener(listener);
nodeCache.start();

//第一次变更节点数据
client.setData().forPath(workerPath,
"第一次更改内
容".getBytes());
Thread.sleep(1000);
```

```
//第二次变更节点数据
client.setData().forPath(workerPath,
                           "第二次更改内
容".getBytes());
Thread.sleep(1000);

//第三次变更节点数据
client.setData().forPath(workerPath,
                           "第三次更改内
容".getBytes());
Thread.sleep(1000);

} catch (Exception e) {
    log.error("创建NodeCache监听失败, path={}",
              workerPath);
}
}
```

通过运行的结果可以看到：NodeCache节点缓存能够重复进行事件节点的监听。代码中的第三次监听的输出节选如下：

```
//省略前两次的输出
- ZNode 节点状态改变, path=/test/listener/node
- ZNode 节点状态改变, data=第三次更改内容
```

- ZNode 节点状态改变, stat=17179869191,

...

在监听的时候, NodeCache 监听的节点为空 (也就是说ZNode路径不存在) 也是可以的。之后, 如果创建了对应的节点, 也是会触发事件, 从而回调nodeChanged()方法。

13.5.3 PathCache子节点监听

PathCache子节点缓存用于子节点的监听, 监控当前节点的子节点被创建、更新或者删除, 需要强调两点:

- (1) 只能监听子节点, 监听不到当前节点。
- (2) 不能递归监听, 子节点下的子节点不能递归监控。

1. PathCache的使用步骤

第一步使用PathChildrenCache子节点缓存构造一个缓存实例。PathChildrenCache有多个重载版本的构造方法, 选择4个进行说明, 具体如下:

```
//重载版本一
public PathChildrenCache(CuratorFramework client,
                        String path, boolean
cacheData)

//重载版本二
public PathChildrenCache(CuratorFramework client,
                        String path, boolean
```

```
cacheData,  
        boolean dataIsCompressed, final ExecutorService  
executorService)  
  
//重载版本三  
public PathChildrenCache(CuratorFramework client,  
        String path, boolean cacheData,  
        boolean dataIsCompressed, ThreadFactory threadFactory)  
  
//重载版本四  
public PathChildrenCache(CuratorFramework client,  
        String path, boolean cacheData,  
        ThreadFactory threadFactory)
```

所有的PathChildrenCache构造方法的前三个参数都是一样的，具体操作如下：

- (1) 第一个参数就是传入创建的Curator的框架客户端。
- (2) 第二个参数就是监听节点的路径。
- (3) 第三个重载参数cacheData表示是否把节点内容缓存起来。如果值为true，那么接收到节点列表变更事件的同时会获得节点内容。

除了上面的三个参数，其他参数的说明如下：

- (1) dataIsCompressed参数表示是否对节点数据进行压缩。

(2) threadFactory参数表示线程池工厂，当PathChildrenCache内部需要开启新的线程异步执行时，使用该线程池工厂来创建线程。

(3) executorService参数和threadFactory参数差不多，表示通过传入的线程池或者线程工厂来异步处理监听事件。

构造完缓冲实例外，PathChildrenCache缓存使用的第二步是构造一个子节点缓存监听器PathChildrenCacheListener实例。

PathChildrenCacheListener监听器接口的定义如下：

```
package org.apache.curator.framework.recipes.cache;
import org.apache.curator.framework.CuratorFramework;
public interface PathChildrenCacheListener {
    void childEvent(CuratorFramework client,
PathChildrenCacheEvent e) throws Exception;
}
```

PathChildrenCacheListener监听器接口中只定义了一个简单的方法childEvent()，当子节点有变化时，这个方法就会被回调。

PathChildrenCacheListener回调监听器的参考代码大致如下：

```
PathChildrenCacheListener listener = new
PathChildrenCacheListener()
{
    @Override
    public void childEvent(CuratorFramework client,
        PathChildrenCacheEvent
event)  {
```

```
try {

    ChildData data = event.getData();
    switch (event.getType()) {
        case CHILD_ADDED:
            log.info("子节点增加, path={}, data={}",
                     data.getPath(),
                     new String(data.getData(),
                     "UTF-8"));

        break;

        case CHILD_UPDATED:
            log.info("子节点更新, path={}, data={}",
                     data.getPath(),
                     new String(data.getData(),
                     "UTF-8"));

        break;

        case CHILD_REMOVED:
            log.info("子节点删除, path={}, data={}",
                     data.getPath(),
                     new String(data.getData(),
                     "UTF-8"));

        break;

        default:
            break;
    }
}
```

```
        }

    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
}

};


```

创建完PathChildrenCacheListener的实例之后，需要将这个回调监听器实例注册到PathChildrenCache缓存实例，具体是调用缓存实例的addListener()方法。然后调用缓存实例nodeCache的start()方法启动节点的事件监听。

启动节点的事件监听start()方法，可以传入启动模式作为参数，启动模式定义在StartMode枚举中，具体如下：

(1) BUILD_INITIAL_CACHE模式：启动时同步初始化Cache，表示创建Cache后就从服务器拉取对应的数据。

(2) POST_INITIALIZED_EVENT模式：启动时异步初始化Cache，表示创建Cache后从服务器拉取对应的数据，完成后触发PathChildrenCacheEvent.Type#INITIALIZED事件，Cache中Listener会收到该事件的通知。

(3) NORMAL模式：启动时，异步初始化Cache，完成后不会发出通知。

2. PathCache事件监听的实战案例

使用PathChildrenCache来监听节点的事件，完整的实例代码如下：

```
package com.crazymakercircle.zk.publishSubscribe;  
//省略import  
  
public class ZkWatcherDemo {  
  
    private String workerPath = "/test/listener/remoteNode";  
    private String subWorkerPath =  
        "/test/listener/remoteNode/id-";  
  
    /**  
     * 子节点监听  
     */  
  
    @Test  
    public void testPathChildrenCache() {  
  
        //检查节点是否存在，没有则创建  
        boolean isExist =  
            ZKclient.instance.isNodeExist(workerPath);  
        if (!isExist) {  
            ZKclient.instance.createNode(workerPath, null);  
        }  
        CuratorFramework client =  
            ZKclient.instance.getClient();  
        try {  
            PathChildrenCache cache =  
                new PathChildrenCache(client, workerPath,
```

```
true);

PathChildrenCacheListener listener=//... 省略监听器实现

代码

//增加监听器

cache.getListenable().addListener(listener);

//设置启动模式


cache.start(PathChildrenCache.StartMode.BUILD_INITIAL_CACHE);

Thread.sleep(1000);


//创建三个子节点

for (int i = 0; i < 3; i++) {

    ZKclient.instance.createNode(subWorkerPath + i,
null);

}

Thread.sleep(1000);


//删除三个子节点

for (int i = 0; i < 3; i++) {

    ZKclient.instance.deleteNode(subWorkerPath +
i);

}

}

} catch (Exception e) {

    log.error("PathCache监听失败, path=", workerPath);

}
```

```
    }  
}
```

运行的结果如下：

```
- 子节点增加, path=/test/listener/node/id-0, data=to set  
content  
- 子节点增加, path=/test/listener/node/id-2, data=to set  
content  
- 子节点增加, path=/test/listener/node/id-1, data=to set  
content  
...  
- 子节点删除, path=/test/listener/node/id-2, data=to set  
content  
- 子节点删除, path=/test/listener/node/id-0, data=to set  
content  
- 子节点删除, path=/test/listener/node/id-1, data=to set  
content
```

从执行结果可以看到，PathChildrenCache能够反复监听到节点的新增和删除。

至此，已经讲完两个系列的缓存监听。简单回顾一下：

(1) NodeCache用来观察ZNode自身，如果ZooKeeper上的ZNode节点被创建，更新或者删除，那么NodeCache会更新缓存，并触发事件给注册的监听器。NodeCache是通过NodeCache类来实现的，监听器对应的事件回调接口为NodeCacheListener。

(2) PathCache子节点缓存用来观察ZNode的子节点、缓存子节点的状态，如果ZNode的某个子节点被创建、更新或者删除，那么PathCache会更新缓存，并且触发事件给注册的监听器。PathCache是通过PathChildrenCache类来实现的，监听器对应的事件回调接口为PathChildrenCacheListener。

13.5.4 TreeCache节点树缓存

TreeCache可以看作是NodeCache、PathCache的合体；TreeCache不仅仅能监听子节点，也能监听节点自身。

1. TreeCache的使用步骤

TreeCache使用的第一步就是构造一个TreeCache缓存实例。TreeCache类有两个构造方法，具体如下：

```
//TreeCache构造器之一
TreeCache(CuratorFramework client, String path)

//TreeCache构造器之二
TreeCache(CuratorFramework client, String path,
          boolean cacheData, boolean dataIsCompressed, int
          maxDepth,
          ExecutorService executorService, boolean
          createParentNodes,
          TreeCacheSelector selector)
```

第一个参数传入创建的Curator的框架客户端，第二个参数传入监听节点的路径，其他参数简单说明如下：

- **dataIsCompressed**: 表示是否对数据进行压缩。
- **maxDepth**: 表示缓存的层次深度，默认为整数最大值。
- **executorService**: 表示监听的执行线程池，默认会创建一个单一线程的线程池。
- **createParentNodes**: 表示是否创建父节点，默认为false。

如果要监听一个ZNode节点，一般使用TreeCache的第一个构造函数即可。

TreeCache使用的第二步是构造一个TreeCacheListener监听器实例。该接口的定义如下：

```
//TreeCache事件监听器接口定义  
package org.apache.curator.framework.recipes.cache;  
import org.apache.curator.framework.CuratorFramework;  
public interface TreeCacheListener {  
    void childEvent(CuratorFramework var1, TreeCacheEvent var2)  
throws Exception;  
}
```

在TreeCacheListener监听器接口中，只定义了一个简单的方法childEvent()，当子节点有变化时，这个方法就会被回调。TreeCacheListener事件回调监听器的参考实现大致如下：

```
TreeCacheListener listener =  
new TreeCacheListener() {
```

```
    @Override
    public void childEvent(CuratorFramework client,
TreeCacheEvent event) {
    try {
        ChildData data = event.getData();
        if (data == null) {
            log.info("数据为空");
            return;
        }
        switch (event.getType()) {
            case NODE_ADDED:
                log.info("[TreeCache] 节点增加, path={}, data={}",
data,
data.getPath(), new
String(data.getData(), "UTF-8"));
                break;

            case NODE_UPDATED:
                log.info("[TreeCache] 节点更新,
path={}, data={}",
path, data,
data.getPath(), new
String(data.getData(), "UTF-8"));
                break;

            case NODE_REMOVED:
                log.info("[TreeCache] 节点删除,
path={}, data={}",
path, data,
```

```
        data.getPath(), new
String(data.getData(), "UTF-8"));
break;
default:
break;
}
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}
}
};
```

在创建完TreeCacheListener的实例之后，调用其addListener()方法将TreeCacheListener监听器实例注册起来，然后调用缓存实例TreeCacheListener的start()方法启动节点的事件监听流程。

2. TreeCache事件监听的实战案例

TreeCache事件监听的实战案例的完整代码如下：

```
package com.crazymakercircle.zk.publishSubscribe;
//省略import
/**
 * 客户端监听实战
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
@Data
```

```
public class ZkWatcherDemo {  
  
    private String workerPath = "/test/listener/remoteNode";  
    private String subWorkerPath =  
        "/test/listener/remoteNode/id-";  
  
    /**  
     * TreeCache既能监听子节点，也能监听节点自身  
     */  
  
    @Test  
    public void testTreeCache() {  
  
        //检查节点是否存在，没有则创建  
        boolean isExist =  
            ZKclient.instance.isNodeExist(workerPath);  
        if (!isExist) {  
            ZKclient.instance.createNode(workerPath, null);  
        }  
        CuratorFramework client =  
            ZKclient.instance.getClient();  
        try {  
            TreeCache treeCache = new TreeCache(client,  
                workerPath);  
            TreeCacheListener listener =//省略回调监听器实现代码  
  
            //设置监听器  
            treeCache.getListenable().addListener(listener);  
        }  
    }  
}
```

```
//启动缓存视图  
treeCache.start();  
  
Thread.sleep(1000);  
  
//创建三个子节点  
for (int i = 0; i < 3; i++) {  
    ZKclient.instance.createNode(subWorkerPath + i,  
        null);  
}  
  
Thread.sleep(1000);  
  
//删除三个子节点  
for (int i = 0; i < 3; i++) {  
    ZKclient.instance.deleteNode(subWorkerPath +  
        i);  
}  
Thread.sleep(1000);  
  
//删除当前节点  
ZKclient.instance.deleteNode(workerPath);  
  
Thread.sleep(Integer.MAX_VALUE);  
  
} catch (Exception e) {  
    log.error("PathCache监听失败, path=", workerPath);  
}
```

```
    }  
}
```

运行的结果如下：

- [TreeCache] 节点增加, path=/test/listener/node, data=to set content
- [TreeCache] 节点增加, path=/test/listener/node/id-0, data=to set content
- [TreeCache] 节点增加, path=/test/listener/node/id-1, data=to set content
- [TreeCache] 节点增加, path=/test/listener/node/id-2, data=to set content
- [TreeCache] 节点删除, path=/test/listener/node/id-2, data=to set content
- [TreeCache] 节点删除, path=/test/listener/node/id-1, data=to set content
- [TreeCache] 节点删除, path=/test/listener/node/id-0, data=to set content
- [TreeCache] 节点删除, path=/test/listener/node, data=to set content

最后，补充说明TreeCacheEvent的事件类型：

- **NODE_ADDED**: 对应于节点的增加。
- **NODE_UPDATED**: 对应于节点的修改。
- **NODE_REMOVED**: 对应于节点的删除。

TreeCache的事件类型与PathCache的事件类型是不同的，
PathCache的事件类型如下：

- CHILD_ADDED：对应于子节点的增加。
- CHILD_UPDATED：对应于子节点的修改。
- CHILD_REMOVED：对应于子节点的删除。

3. Curator事件监听的原理

无论是PathChildrenCache还是TreeCache，所谓的监听都是进行Curator本地缓存视图和ZooKeeper服务器远程的数据节点的对比，并且进行数据同步时会触发相应的事件。以NODE_ADDED（节点新增事件）的触发为例进行简单说明。在本地缓存视图开始创建的时候，本地视图为空，从服务器进行数据同步时，本地的监听器就能监听到NODE_ADDED事件。因为刚开始本地缓存并没有内容，然后本地缓存和服务器缓存进行对比，发现ZooKeeper服务器是有节点数据的，这才将服务器的节点缓存到本地，触发本地缓存的NODE_ADDED事件。

13.6 分布式锁原理与实战

在单体的应用开发场景中涉及并发同步的时候，大家往往采用 synchronized或者Lock的方式来解决多线程间的同步问题。在分布式集群工作的开发场景中需要一种更加高级的锁机制来处理跨JVM进程之间的数据同步问题，这就是分布式锁。

13.6.1 公平锁和可重入锁的原理

最经典的分布式锁是可重入的公平锁。什么是可重入的公平锁呢？直接讲解概念和原理会比较抽象难懂，这里用一个简单的故事来类比一下。

故事发生在没有自来水的古代，在一个村子里有一口井，水质非常好，村民都抢着取井里的水。井就一口，村里的人很多，村民为争抢取水而打架，甚至头破血流。

问题总是要解决的，村委会主任绞尽脑汁想出了一个凭号取水的方案。井边安排一个看井人，维护取水的秩序。取水秩序很简单：

- (1) 取水之前先取号。
- (2) 号排在前面的可以先取水。
- (3) 先到的排在前面，后到的一个一个挨着在井边排成一队。

取水示意图如图13-4所示。

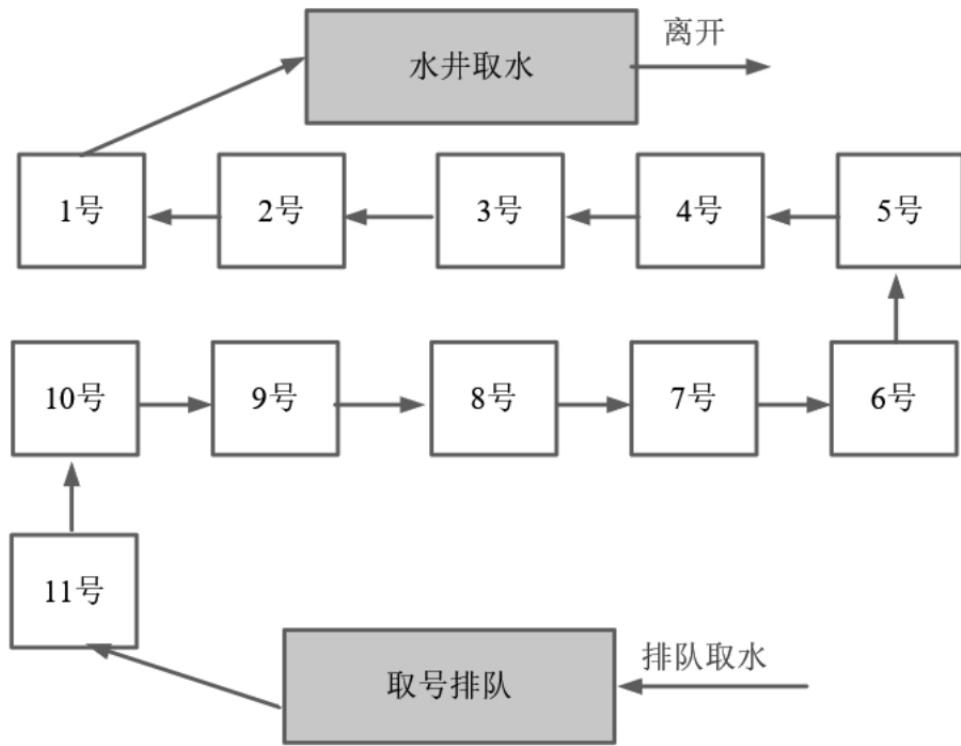


图13-4 排队取水示意图

这种排队取水模型就是一种锁的模型。排在最前面的号拥有取水权，就是一种典型的独占锁。另外，先到先得，号排在前面的人先取到水，取水之后就轮到下一个号取水，挺公平的，说明它是一种公平锁。

什么是可重入锁？假定取水时以家庭为单位，家庭的某人拿到号，其他的家庭成员过来打水，这时不用再取号，如图13-5所示。

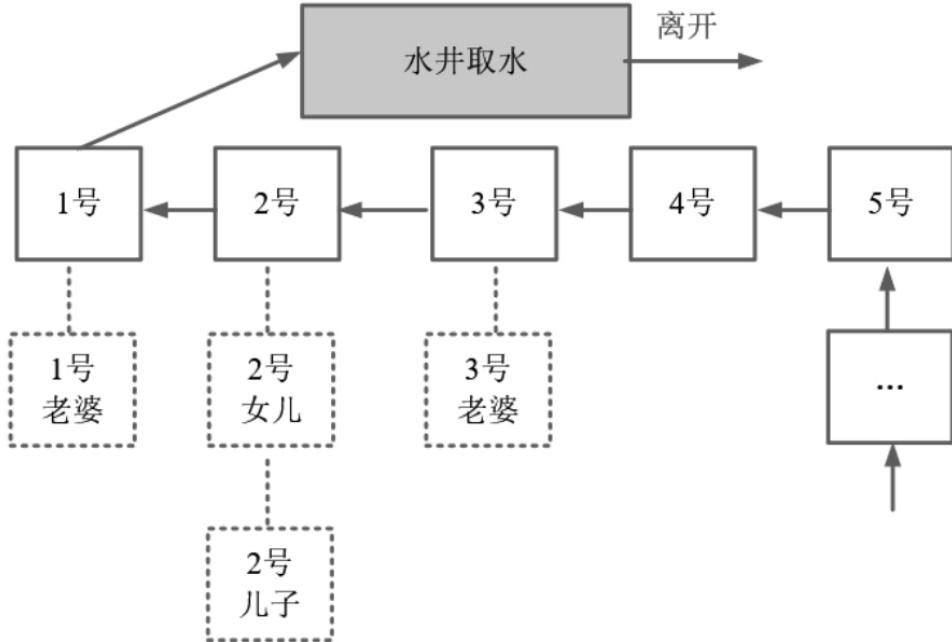


图13-5 同一家庭的人不需要重复排队

在图13-5中，排在1号的家庭，老公取号，假设其老婆来了，直接排第一个。再看2号，父亲正在打水，假设其儿子和女儿也到井边了，直接排第二个。总之，如果取水时以家庭为单位，则同一个家庭可以直接复用排号，不用从后面排起重新取号。

在上面这个故事模型中，取号一次，可以多次取水，其原理为可重入锁的模型。在重入锁模型中，一把独占锁可以被多次锁定，这就叫作可重入锁。

13.6.2 ZooKeeper分布式锁的原理

理解了经典的公平可重入锁的原理后，再来看在分布式场景下的公平可重入锁的原理。通过前面的分析基本可以判定：ZooKeeper的临时顺序节点天生就有一副实现分布式锁的胚子。为什么呢？

(1) ZooKeeper的每一个节点都是一个天然的顺序发号器。

在每一个节点下面创建临时顺序节点(EPHEMERAL_SEQUENTIAL)类型，新的子节点后面会加上一个次序编号，而这个生成的次序编号是上一个生成的次序编号加一。

例如，有一个用于发号的节点“/test/lock”为父节点，可以在这个父节点下面创建相同前缀的临时顺序子节点，假定相同的前缀为“/test/lock/seq-”。第一个创建的子节点基本上应该为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，以此类推，如图13-6所示。

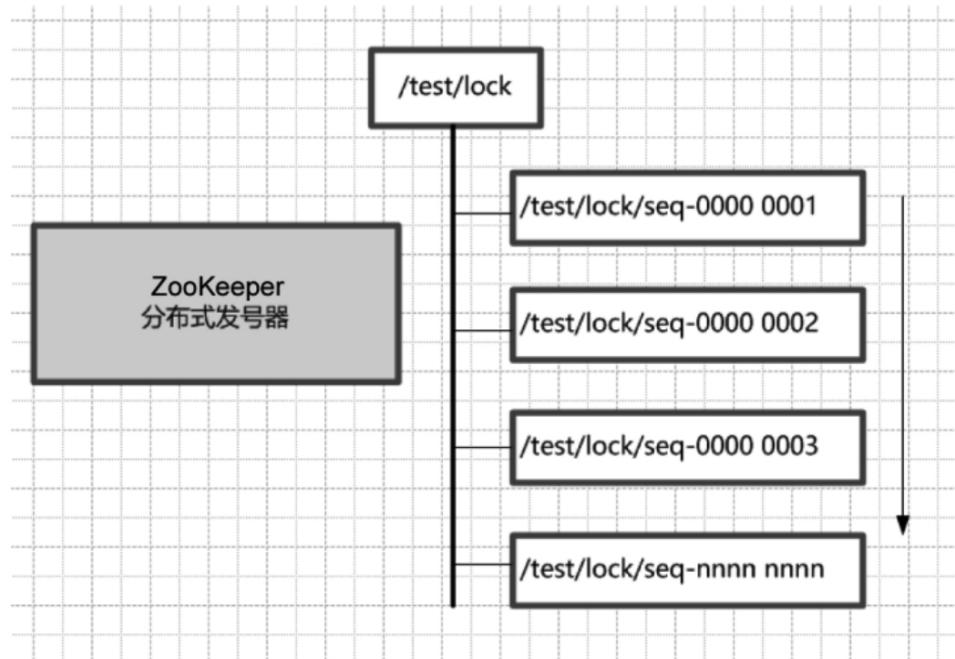


图13-6 ZooKeeper临时顺序节点的天然发号器作用

(2) ZooKeeper节点的递增有序性可以确保锁的公平。

一个ZooKeeper分布式锁需要创建一个父节点，尽量是持久节点（PERSISTENT类型），然后每个要获得锁的线程都在这个节点下创建一个临时顺序节点，因为ZooKeeper节点是按照创建的次序依次递增的。

为了确保公平，可以简单地规定：编号最小的那个节点表示获得了锁。所以，每个线程在尝试占用锁之前首先判断自己的排号是不是当前最小，如果是就获取锁。

(3) ZooKeeper的节点监听机制可以保障占有锁的传递有序而且高效。

每个线程抢占锁之前，先尝试创建自己的ZNode。同样，释放锁的时候需要删除创建的ZNode。创建成功后，如果不是排号最小的节点，就处于等待通知的状态。前一个ZNode删除的时候，会触发ZNode事件，当前节点监听到删除事件就是轮到了自己占有锁的时候。第一个通知第二个，第二个通知第三个，击鼓传花似的依次向后。

ZooKeeper的节点监听机制能够非常完美地实现这种击鼓传花似的信息传递。具体的方法是，每一个等通知的ZNode节点只需要监听（listen）或者监视（watch）排号在自己前面的那个，而且紧挨在自己前面的那个节点就能收到其删除事件了。只要上一个节点被删除了，就进行再一次判断，看看自己是不是序号最小的那个节点，如果是，自己就获得锁。

另外，ZooKeeper的内部优越机制能保证由于网络异常或者其他原因，集群中占用锁的客户端失联时锁能够被有效释放。一旦占用ZNode锁的客户端与ZooKeeper集群服务器失去联系，这个临时ZNode也将自

动删除。排在它后面的那个节点也能收到删除事件，从而获得锁。正是由于这个原因，在创建取号节点的时候尽量创建临时ZNode节点。

(4) ZooKeeper的节点监听机制能避免羊群效应。

ZooKeeper这种首尾相接、后面监听前面的方式可以避免羊群效应。所谓羊群效应，就是一个节点挂掉，所有节点都去监听，然后做出反应，这样会给服务器带来巨大压力，所以有了临时顺序节点，当一个节点挂掉，只有它后面的那一个节点才做出反应。

13.6.3 分布式锁的基本流程

接下来基于ZooKeeper实现一下分布式锁。首先，定义一个锁的接口Lock，很简单，仅仅实现两个抽象方法：一个加锁方法，一个解锁方法。Lock接口的代码如下：

```
package com.crazymakercircle.zk.distributedLock;
```

```
/**  
 * 锁的接口  
 * create by 尼恩 @ 疯狂创客圈  
 */  
public interface Lock {
```

```
    /**  
     * 加锁方法  
     *
```

```
* @return 是否成功加锁
*/
boolean lock();

/**
 * 解锁方法
 *
 * @return 是否成功解锁
*/
boolean unlock();
}
```

使用ZooKeeper实现分布式锁的算法有以下几个要点：

(1) 一把分布式锁通常使用一个ZNode节点表示，如果锁对应的ZNode节点不存在，就先创建ZNode节点。这里假设为“/test/lock”，代表一把需要创建的分布式锁。

(2) 抢占锁的所有客户端，使用锁的ZNode节点的子节点列表来表示。如果某个客户端需要占用锁，则在“/test/lock”下创建一个临时有序的子节点。

这里所有临时有序子节点尽量共用一个有意义的子节点前缀。

比如，子节点的前缀为“/test/lock/seq-”，则第一次抢锁对应的子节点为“/test/lock/seq-000000000”，第二次抢锁对应的子节点为“/test/lock/seq-000000001”，以此类推。

再比如，子节点前缀为“/test/lock/”，则第一次抢锁对应的子节点为“/test/lock/000000000”，第二次抢锁对应的子节点为“/test/lock/000000001”，以此类推，也非常直观。

(3) 如果判定客户端是否占有锁呢？很简单，客户端创建子节点后，需要判断自己创建的子节点是否为当前子节点列表中序号最小的子节点。如果是，就认为加锁成功；如果不是，则监听前一个ZNode子节点的变更消息，等待前一个节点释放锁。

(4) 一旦队列中后面的节点获得前一个子节点变更通知，则开始进行判断，判断自己是否为当前子节点列表中序号最小的子节点，如果是，就认为加锁成功；如果不是，则持续监听，一直到获得锁。

(5) 获取锁后，开始处理业务流程。完成业务流程后，删除自己对应的子节点，完成释放锁的工作，以方便后继节点能捕获到节点变更通知，获得分布式锁。

13.6.4 加锁的实现

Lock接口中加锁的方法是lock()。lock()方法的大致流程是：首先尝试着加锁，如果加锁失败就去等待，然后重复。

1. lock()方法的实现代码

lock()方法加锁的实现代码，大致如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
public class ZkLock implements Lock {
```

```

//ZKLock的节点链接

private static final String ZK_PATH = "/test/lock";
private static final String LOCK_PREFIX = ZK_PATH + "/";
private static final long WAIT_TIME = 1000;

//ZK客户端

CuratorFramework client = null;

private String locked_short_path = null;
private String locked_path = null;
private String prior_path = null;
final AtomicInteger lockCount = new AtomicInteger(0);
private Thread thread;

public ZkLock() {
    ZKclient.instance.init();
    if (!ZKclient.instance.isNodeExist(ZK_PATH)) {
        ZKclient.instance.createNode(ZK_PATH, null);
    }
    client = ZKclient.instance.getClient();
}

/**
 * 加锁的实现
 *
 * @return 是否加锁成功
 */
@Override

```

```
public boolean lock() {  
  
    //可重入，确保同一线程，可以重复加锁  
  
    synchronized (this) {  
  
        if (lockCount.get() == 0) {  
  
            thread = Thread.currentThread();  
  
            lockCount.incrementAndGet();  
  
        } else {  
  
            if (!thread.equals(Thread.currentThread())) {  
  
                return false;  
  
            }  
  
            lockCount.incrementAndGet();  
  
        }  
  
        return true;  
    }  
  
}  
  
try {  
  
    boolean locked = false;  
  
    //首先尝试着去加锁  
  
    locked = tryLock();  
  
    if (locked) {  
  
        return true;  
    }  
  
    //如果加锁失败就去等待  
}
```

```
        while (!locked) {  
  
            //等待  
            await();  
  
            //获取等待的子节点列表  
            List<String> waiters = getWaiters();  
  
            //判断是否加锁成功  
            if (checkLocked(waiters)) {  
                locked = true;  
            }  
            return true;  
        } catch (Exception e) {  
            e.printStackTrace();  
            unlock();  
        }  
        return false;  
    }  
    //省略其他方法  
}
```

2. tryLock() 尝试加锁

尝试加锁的tryLock()方法是关键，做了两件重要的事情：

- (1) 创建临时顺序节点，并且保存自己的节点路径。

(2) 判断是否是第一个：如果是第一个，则加锁成功；如果不是，则找到前一个ZNode节点，并且保存其路径到prior_path。

尝试加锁的tryLock()方法，其实现代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
  
public class ZkLock implements Lock {  
  
    ...  
  
    /**  
     * 尝试加锁  
     *  
     * @return 是否加锁成功  
     * @throws Exception 异常  
     */  
  
    private boolean tryLock(){  
        //创建临时ZNode  
        locked_path =  
            ZKclient.instance  
                .createEphemeralSeqNode(LOCK_PREFIX);  
        if (null == locked_path) {  
            throw new Exception("zk error");  
        }  
  
        //取得加锁的排队编号  
        locked_short_path = getShortPath(locked_path);
```

```
//获取加锁的队列
List<String> waiters = getWaiters();

//获取等待的子节点列表，判断自己是否为第一个
if (checkLocked(waiters)) {
    return true;
}

//判断自己排第几个
int index = Collections.binarySearch(waiters,
locked_short_path);
if (index < 0) {
    //网络抖动，获取到的子节点列表里可能已经没有自己了
    throw new Exception("节点没有找到：" +
locked_short_path);
}

//如果自己没有获得锁
//保存前一个节点，稍候会监听前一个节点
prior_path = ZK_PATH + "/" + waiters.get(index - 1);
return false;
}

//省略其他方法
}
```

创建临时顺序节点后，其完整路径存放在locked_path成员中；另外，还截取了一个后缀路径放在locked_short_path成员中，后缀路径是一个短路径，只有完整路径的最后一层。为什么要单独保存短路径呢？因为在获取的远程子节点列表中的其他路径返回结果时，返回的都是短路径，都只有最后一层路径。所以，为了方便后续进行比较，也把自己的短路径保存下来。

创建了自己的临时节点后，调用checkLocked()方法判断是否是锁定成功。如果锁定成功，则返回true；如果没有获得锁，则要监听前一个节点，此时需要找出前一个节点的路径，并保存在prior_path成员中，供后面的await()等待方法去监听使用。这里先介绍一下checkLocked锁定判断方法。

3. checkLocked() 检查是否持有锁

在checkLocked()方法中，判断是否可以持有锁。判断规则很简单：当前创建的节点是否在上一步获取到的子节点列表的第一个位置。

- 如果是，说明可以持有锁，就返回true，表示加锁成功。
- 如果不是，说明有其他线程早已持有了锁，就返回false。

checkLocked()方法的代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
public class ZkLock implements Lock {  
    ...  
    /**
```

```
* 判断是否加锁成功
* @param waiters 排队列表
* @return 成功状态
*/
private boolean checkLocked(List<String> waiters) {

    //节点按照编号升序排列
    Collections.sort(waiters);

    //如果是第一个，代表自己已经获得了锁
    if (locked_short_path.equals(waiters.get(0))) {
        log.info("成功地获取分布式锁，节点为{}",
        locked_short_path);
        return true;
    }
    return false;
}
//省略其他方法
}
```

checkLocked()方法比较简单，将参与排队的所有子节点列表从小到大根据节点名称进行排序。排序主要依靠节点的编号，也就是后ZNode路径的10位数字，因为前缀都是一样的。排序之后进行判断：如果自己的locked_short_path编号位置排在第一个，则代表自己已经获得了锁；如果不是，则返回false。

如果checkLocked()为false，那么外层一般会执行await()等待方法，执行夺锁失败以后的等待逻辑。

4. await() 监听前一个节点释放锁

await()也很简单，就是监听前一个ZNode节点（prior_path成员）的删除事件，代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
  
public class ZkLock implements Lock {  
  
    ...  
  
    /**  
     * 等待，监听前一个节点的删除事件  
     */  
  
    private void await()...{  
        if (null == prior_path) {  
            throw new Exception("prior_path error");  
        }  
        final CountDownLatch latch = new CountDownLatch(1);  
        //监听方式一： Watcher 一次性订阅  
        //订阅比自己次小顺序节点的删除事件  
        Watcher w = new Watcher() {  
            @Override  
            public void process(WatchedEvent watchedEvent) {  
                System.out.println("监听到的变化 watchedEvent = "  
                +
```

```
watchedEvent);

    log.info("[WatchedEvent] 节点删除");
    latch.countDown();
}

};

//开始监听

client.getData().usingWatcher(w).forPath(prior_path);

//限时等待，最长加锁时间为3秒

latch.await(WAIT_TIME, TimeUnit.SECONDS);

}

//省略其他方法

}
```

首先添加一个Watcher监听，而监听的节点正是前面所保存在prior_path成员的前一个节点的路径。这里仅仅去监听自己前一个节点的变动，而不是其他节点的变动，提升效率。完成监听之后，调用latch.await()，线程进入等待状态，一直到线程被监听回调代码中的latch.countDown()所唤醒，或者等待超时。

说明

以上代码用到的CountDownLatch的核心原理和实战知识请参阅本书的下一卷《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》。

在上面的代码中，监听前一个节点的删除可以使用以下两种监听方式：

- **Watcher**订阅。
- **TreeCache**订阅。

两种方式的效果差不多，但是这里的删除事件只需要监听一次，不需要反复监听，所以使用的是**Watcher**一次性订阅。**TreeCache**订阅的代码在源码工程中已经被注释，仅供大家参考。

一旦前一个节点prior_path被删除，就将线程从等待状态唤醒，重新一轮的锁争夺，直到获取锁，并且完成业务处理。

至此，分布式Lock加锁的算法还差一点就介绍完成，即实现锁的可重入。

5. 可重入的实现代码

什么是可重入？只需要保障同一个线程进入加锁的代码，可以重复加锁成功即可。修改前面的lock()方法，在前面加上可重入的判断逻辑。代码如下：

```
@Override  
public boolean lock() {  
    //可重入的判断  
    synchronized (this) {  
        if (lockCount.get() == 0) {  
            thread = Thread.currentThread();  
            lockCount.incrementAndGet();  
        }  
    }  
}
```

```
        } else {
            if (!thread.equals(Thread.currentThread())) {
                return false;
            }
            lockCount.incrementAndGet();
            return true;
        }
    }
//...
}
```

为了变成可重入，在代码中增加了一个加锁的计数器lockCount，计算重复加锁的次数。如果是同一个线程加锁，只需要增加次数，直接返回，就表示加锁成功。至此，lock()方法介绍完成。接下来去释放锁。

13.6.5 释放锁的实现

Lock接口中的unLock()方法表示释放锁。释放锁主要有两项工作：

- 减少重入锁的计数，如果最终的值不是0，就直接返回，表示成功地释放了一次。
- 如果计数器为0，就移除Watchers监听器，并且删除创建的ZNode临时节点。

unLock()方法的代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
  
public class ZkLock implements Lock {  
    //...  
    /**  
     * 释放锁  
     *  
     * @return 是否成功释放锁  
     */  
    @Override  
    public boolean unlock() {  
  
        //只有加锁的线程能够解锁  
        if (!thread.equals(Thread.currentThread())) {  
            return false;  
        }  
  
        //减少可重入的计数  
        int newLockCount = lockCount.decrementAndGet();  
  
        //计数不能小于0  
        if (newLockCount < 0) {  
            throw new IllegalMonitorStateException("计数不对：" +  
                locked_path);  
        }  
    }  
}
```

```
//如果计数不为0，就直接返回
if (newLockCount != 0) {
    return true;
}

try {
    //删除临时节点
    if (ZKclient.instance.isNodeExist(locked_path)) {
        client.delete().forPath(locked_path);
    }
} catch (Exception e) {
    e.printStackTrace();
    return false;
}
return true;
}

//省略其他方法
}
```

这里为了尽量保证线程安全，可重入计数器的类型，使用的不是int类型，而是Java并发包中的原子类型——AtomicInteger。

13.6.6 分布式锁的使用

编写一个用例来测试一下Zlock（分布式锁）的使用，代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
@Slf4j  
public class ZkLockTester {  
  
    //需要锁来保护的公共资源  
    //变量  
    int count = 0;  
    /**  
     * 测试自制分布式锁  
     *  
     * @throws InterruptedException 异常  
     */  
    @Test  
    public void testLock() throws InterruptedException {  
        //10个并发任务  
        for (int i = 0; i < 10; i++) {  
            FutureTaskScheduler.add(() -> {  
                //创建锁  
                ZkLock lock = new ZkLock();  
                lock.lock();  
                //每条线程执行10次累加  
                for (int j = 0; j < 10; j++) {  
                    //公共的资源变量累加  
                    count++;  
                }  
            });  
        }  
    }  
}
```

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
log.info("count = " + count);
//释放锁
lock.unlock();
});
}
Thread.sleep(Integer.MAX_VALUE);
}
}
```

以上代码是10个并发任务，每个任务累加10次。执行以上用例，就会发现结果是预期的和100。如果不使用锁，结果可能就不是100，因为上面的count是一个普通变量，不是线程安全的。

说明

有关线程安全的核心原理和实战知识，请参阅本书的下一卷《Java高并发核心编程 卷2：多线程、锁、JMM、JUC、高并发设计模式》。

原理上一个Zlock实例代表一把锁，并需要占用一个ZNode永久节点，如果需要很多分布式锁，则需要很多不同的ZNode节点。以上代码如果要扩展为多个分布式锁的版本，还需要进行简单改造（自行实现）。

13.6.7 Curator的InterProcessMutex可重入锁

分布式锁Zlock自主实现主要的价值：学习分布式锁的原理和基础开发，仅此而已。在实际的开发中，如果需要使用分布式锁，建议直接使用Curator客户端中各种官方实现的分布式锁，比如其中的InterProcessMutex可重入锁。

这里提供一个简单的InterProcessMutex可重入锁的使用实例，代码如下：

```
package com.crazymakercircle.zk.distributedLock;  
//省略import  
@Slf4j  
public class ZkLockTester {  
    //需要锁来保护的公共资源  
    //变量  
    int count = 0;  
    /**  
     * 测试Curator客户端自带的互斥锁  
     *  
     * @throws InterruptedException 异常  
    */
```

```
*/  
@Test  
public void testzkMutex() throws InterruptedException {  
    CuratorFramework client =  
        ZKclient.instance.getClient();  
    //创建互斥锁  
    final InterProcessMutex zkMutex =  
        new InterProcessMutex(client, "/mutex");  
    //每条线程执行10次累加  
    for (int i = 0; i < 10; i++) {  
        FutureTaskScheduler.add(() -> {  
            try {  
                //获取互斥锁  
                zkMutex.acquire();  
                for (int j = 0; j < 10; j++) {  
                    //公共的资源变量累加  
                    count++;  
                }  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                log.info("count = " + count);  
                //释放互斥锁  
                zkMutex.release();  
            } catch (Exception e) {  
            }  
        });  
    }  
}
```

```
        e.printStackTrace();
    }
}

Thread.sleep(Integer.MAX_VALUE);
}

}
```

13.6.8 ZooKeeper分布式锁的优缺点

总结一下ZooKeeper分布式锁：

- 优点：ZooKeeper分布式锁（如InterProcessMutex）能有效地解决分布式问题、不可重入问题，使用起来较为简单。
- 缺点：ZooKeeper实现的分布式锁性能不太高。因为每次在创建锁和释放锁的过程中都要动态创建、销毁瞬时节点。在ZooKeeper中，创建和删除节点只能通过Leader服务器来执行，然后Leader服务器还需要将数据同步到所有的Follower机器上，这样频繁的网络通信，性能的短板是非常突出的。

总之，在高性能、高并发的场景下，不建议使用ZooKeeper的分布式锁。由于ZooKeeper具有高可用特性，因此在并发量不是太高的场景推荐使用ZooKeeper的分布式锁。

在目前分布式锁的实现方案中，比较成熟、主流的方案有两种：

(1) 基于ZooKeeper的分布式锁，适用于高可靠（高可用）而并发量不是太大的场景。

(2) 基于Redis的分布式锁，适用于并发量很大、性能要求很高、可靠性问题可以通过其他方案去弥补的场景。

总之，没有谁好谁坏的问题，而是谁更适合的问题。

第14章 分布式缓存Redis实战

数据库的查询比较耗时，使用缓存能大大节省数据访问的时间。例如，表中有两千万个用户信息，在加载用户信息时，一次数据库查询大致的时间在数百毫秒级别。这仅仅是一次查询，如果是频繁多次的数据库查询，效率就会更低。

提升效率的通用做法是把数据加入缓存，每次加载数据之前先去缓存中加载，如果为空，就再去查询数据库并将数据加入缓存。这样可以大大提高数据访问的效率。

从大的层面来说，在开发高并发系统时，有三把利器用来保护系统：缓存、降级和限流。其中，缓存是最为重要的一个应对高并发的方式。Redis缓存中间件目前已经成为缓存的事实标准。

14.1 Redis入门

本节主要介绍Redis的安装和配置，以及Redis的客户端操作。

14.1.1 Redis的安装和配置

Redis在Windows下的安装很简单，根据系统的实际情况选择32位或者64位的Redis安装版本，而后下载安装即可，下载地址为<https://github.com/MSOpenTech/redis/releases>。

Redis在Linux下的版本需要先编译再安装，下载地址为<http://redis.io/download>。下载较为稳定的版本即可，本教程使用的版本为3.2.0。

无论是在Linux还是Windows下安装Redis，具体安装过程都涉及很多烦琐细节，文字描述的效果不甚理想，而使用视频的方式呈现的效果更佳。这部分安装过程已经通过博客和视频的形式在疯狂创客圈的社群博客发布，具体如下：

Linux下 Redis 安装（带视频）的博客地址：

<https://www.cnblogs.com/crazymakercircle/p/11985983.html>

Windows下 Redis 安装（带视频）的博客地址：

<https://www.cnblogs.com/crazymakercircle/p/11973314.html>

按照视频的说明在自己的机器上安装即可。在使用之前，可能需要查看和修改Redis的配置项，大致有两种方式：

- 通过配置文件查看和修改。
- 通过配置命令查看和修改。

第一种方式是通过配置文件修改Redis的配置项。Redis在Windows中安装完成后，配置文件位于Redis安装目录下，文件名为redis.windows.conf。可以复制它，保存一份自己的配置版本redis.conf，以自己的这份文件作为运行时的配置文件。Redis在Linux中安装完成后，redis.conf是一个默认的配置文件。通过redis.conf文件，可以查看和修改配置项的值。

第二种方式是通过命令修改Redis的配置项。启动Redis的命令客户端工具，连接上Redis服务，可以使用以下命令来查看和修改Redis配置项：

```
CONFIG GET CONFIG_SETTING_NAME  
CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VALUE
```

前一个命令CONFIG GET是配置项的查看命令，使用时在后面加上配置项的名称；后一个命令CONFIG SET。是配置项的修改命令，使用时在后面加配置项的名称和要设置的新值。其中，CONFIG GET查看命令可以使用通配符，支持一次查看多个配置项。

说明

Redis的客户端命令是不区分字母大小写的。

例如，查看Redis的服务端口可以使用CONFIG GET port：

```
127.0.0.1:6379>config get port
1) "port"
2) "6379"
```

通过控制台输出的结果，我们可以看到当前的Redis服务端口为6379。

Redis的配置项比较多，大致清单如下：

(1) port：端口，用于查看和设置Redis监听端口，默认端口为6379。

(2) bind：主机地址，用于查看和绑定的主机地址，默认地址值为127.0.0.1。在单网卡的机器上，这个选项一般不需要修改。

(3) timeout：连接空闲多长要关闭连接，表示客户端闲置一段时间后要关闭连接。如果指定为0，就表示连接的时长不限制。这个选项的默认值为0，表示默认不限制连接的空闲时长。

(4) dbfilename：指定保存缓存数据的本地文件名，默认值为dump.rdb。

(5) dir：指定保存缓存数据的本地文件所存放的目录，默认值为安装目录。

(6) rdbcompression：指定存储缓存数据至本地文件时是否压缩数据，默认为yes。Redis采用LZF压缩。为了节省CPU时间，可以关闭该选项，但会导致本地文件变得巨大。

(7) save: 指定在多长时间内有多少次键-值对 (Key-Value Pair) 更新操作，就将缓存数据同步到本地文件。save配置项的格式为：

```
save <seconds> <changes>
```

其中，seconds表示时间段的长度，changes表示变化的次数。如果在seconds时间段内变化了changes次，则将Redis缓存数据同步到文件。设置为900秒（15分钟）内有1个更改，则同步到文件的命令为：

```
127.0.0.1:6379> config set save "900 1"
```

```
OK
```

```
127.0.0.1:6379> config get save
```

```
1) "save"
```

```
2) "jd 900"
```

设置为只要满足900秒（15分钟）内有1个更改、300秒（5分钟）内有10个更改或者60秒内有10000个更改，则同步到文件：

```
127.0.0.1:6379> config set save "900 1 300 10 60 10000"
```

```
OK
```

```
127.0.0.1:6379> config get save
```

```
1) "save"
```

```
2) "jd 900 jd 300 jd 60"
```

(8) requirepass: 设置Redis连接密码。如果配置了连接密码，那么客户端在连接Redis时需要通过AUTH <password>命令提供密码。默认这个选项是关闭的。

(9) slaveof: 在主从复制模式下, 如果当前节点为Slave(从)节点, 就设置为Master(主)节点的IP地址及端口, 在Redis启动时自动从Master(主)节点进行数据同步。如果已经是Slave(从)服务器, 则会丢掉旧数据集, 从新的Master主服务器同步缓存数据。

设置为Slave节点命令的格式为:

```
slaveof <masterip> <masterport>
```

(10) masterauth: 在主从复制模式下, 当Master(主)服务器节点设置了密码保护时, Slave(从)服务器用此命令设置连接Master(主)服务器的密码。设置Master服务器节点密码的命令格式为:

```
masterauth <master-password>
```

(11) databases: 设置缓存数据库的数量, 默认数据库数量为16个。这16个数据库的ID为0~15, 默认使用的数据库是第0个。可以使用SELECT <dbid>命令在连接时通过数据库ID来指定要使用的数据库。

databases配置选项可以设置多个缓存数据库, 不同的数据库存放不同应用的缓存数据。类似于MySQL数据库, 不同的应用程序数据存储在不同的数据库下。在Redis中, 数据库的名称由一个整数索引标识, 而不是由一个字符串名称来标识。默认情况下, 一个客户端连接到数据库0。可以通过SELECT <dbid>命令来切换到不同的数据库。例如, 命令select 2将Redis操作库切换到第3个数据库, 随后所有的Redis客户端命令将使用缓存数据库3。

Redis存储的形式是键-值对，其中键（Key）不能发生冲突。每个数据库都有属于自己的空间，不必担心之间的键相冲突。在不同的数据库中，相同的键可以分别取到各自的值（Value）。

清除缓存数据时使用flushdb命令，但是只会清除当前数据库中的数据，而不会影响到其他数据库。flushall命令会清除Redis实例所有数据库（0~15）的缓存数据，因此在执行flushall命令前要格外小心。

在Java编程中，配置连接Redis的Uri连接字符串时可以指定到具体的数据库，格式为：

```
redis://用户名:密码@host:port/Redis库名
```

例如：

```
redis://testRedis:foobared@119.254.166.136:6379/1
```

表示连接到第二个Redis缓存库，其中的用户名是可以随意填写的。

14.1.2 Redis客户端命令

通过安装目录下的redis-cli客户端可以连接到Redis服务器。如果需要在远程Redis服务上执行命令，我们使用的也是redis-cli命令。Windows/Linux命令的格式为：

```
redis-cli -h host -p port -a password
```

实例如下：

```
redis-cli -h 127.0.0.1 -p 6379 -a "123456"
```

此命令实例表示使用Redis命令客户端连接到的远程主机为127.0.0.1、端口为6379、密码为"123456"的Redis服务。

一旦连接上Redis本地服务或者远程服务，就可以通过命令客户端完成Redis的命令执行，这些命令包括了基础键-值（Key-Value）缓存操作。基础的键-值缓存操作大致有：

(1) set命令：根据键设置值。

(2) get命令：根据键获取值，当键不存在时返回空结果。

set、get两个命令的使用很简单，与Java中Map数据类型的键-值设置与获取非常相似。比如键为"foo"、值为"bar"，其设置和获取的示例如下：

```
127.0.0.1:6379>set foo bar
```

```
OK
```

```
127.0.0.1:6379>get foo
```

```
"bar"
```

(3) keys命令：查找所有符合给定模式（Pattern）的键。模式支持多种通配符，如表14-1所示。

表14-1 键的匹配模式支持的多种通配符

符 号	说 明
?	匹配一个字符
*	匹配任意个（包括0个）字符
[-]	匹配区间内的任一字符，如 a[b-d]可以匹配"ab" "ac" "ad"
\	转义字符。使用\?，可以匹配"?"字符

(4) exists命令：判断一个键是否存在。如果键存在，就返回1，否则返回0。例如：

```
127.0.0.1:6379> exists foo
```

```
(integer) 1
```

```
127.0.0.1:6379> exists bar
```

```
(integer) 0
```

(5) expire命令：指定的键生存过期时间，以秒为单位。

(6) ttl命令：获取指定键的剩余生存时间（Time To Live，TTL），以秒为单位。

```
127.0.0.1:6379>set foo2 bar2
```

```
OK
```

```
127.0.0.1:6379>expire foo2 10000
```

```
(integer) 1
```

```
127.0.0.1:6379>ttl foo2
```

```
(integer) 9995
```

```
127.0.0.1:6379>ttl foo2
```

```
(integer) 9987
```

```
127.0.0.1:6379>ttl foo
```

```
(integer) -1
```

没有指定剩余时间时， 默认的剩余生存时间为-1， 表示永久存在。

(7) type命令：返回键所存储的值的类型。Redis中有5种数据类型：String（字符串类型）、Hash（哈希类型）、List（列表类型）、Set（集合类型）、ZSet（有序集合类型）。最简单的值类型为String类型，后面会详细介绍。

(8) del命令：删除键，可以删除一个或多个键，返回值是删除的键的个数。实例如下：

```
127.0.0.1:6379> del foo
(integer) 1
127.0.0.1:6379> del foo2
(integer) 1
```

(9) ping命令：检查客户端是否连接成功。如果连接成功，就返回pong。

14.1.3 Redis键的命名规范

在实际开发中为了更好地进行命令空间的区分，键会有很多层次间隔，就像一棵目录树一样。例如，在“疯狂创客圈”的CrazyIM系统中既有用于缓存用户的键，也有用于缓存IM消息的键。为了以示区分，方便统计、更新、清除，可以将键的名称组织成一种目录树一样的层次关系。很多人会习惯用英文句号（点号）作为键层次关系的分隔符，例如：

superkey.subkey.subsubkey.subsubsubkey...

使用Redis，建议使用冒号作为上级和下级之间的分隔符，具体如下：

superkey:subkey:subsubkey:subsubsubkey: ...

例如，在“疯狂创客圈”的CrazyIM系统中有缓存用户的键，也有缓存IM消息的键，使用下面的规范进行命名：

- 缓存用户的键，命名为CrazyIMKey:User:0001。
- 缓存消息的键，命名为CrazyIMKey:IMMessage:0001。

说明

在上面的例子中，缓存用户的键和缓存消息的键的最后部分（如0001）表示的是业务ID。

键的命名规范使用冒号分隔，大致的优势如下：

(1) 方便分层展示。Redis的很多客户端可视化管理工具（如Redis Desktop Manager）是以冒号作为分类展示的，这就能方便用户快速查到Redis键对应的值。

(2) 方便删除与维护。对于某一层级的键，可以使用通配符进行批量查询和批量删除。

14.2 Redis数据类型

Redis中有5种数据类型：String（字符串类型）、Hash（哈希类型）、List（列表类型）、Set（集合类型）、ZSet（有序集合类型）。

14.2.1 String

String类型是Redis中最简单的数据结构，既可以存储文字（例如"hello world"），又可以存储数字（例如整数10086和浮点数3.14），还可以存储二进制数据（例如10010100）。下面对String类型的主要操作进行简要介绍。

（1）设值：SET Key Value [EX seconds]。

SET命令为键（Key）设置了指定的值（Value）。如果键已经存在，并且已经绑定了一个旧值，旧值会被覆盖，不论旧值的类型是否为String，都会被忽略掉。如果键不存在，就会在数据库中添加一个键，保存的值就是刚刚设置的新值。

[EX seconds]选项表示键过期的时间，单位为秒。如果不加设置，表示键永不过期。另外，SET命令还有一些选项，使用较少，这里就不展开说明了。

（2）批量设值：MSET Key Value[Key Value ...]。

一次性设置多个键-值对（Key-Value Pair），相当于同时调用多次SET命令。需要注意的是，这个操作是原子的。也就是说，所有的键都一次性设置的。如果同时运行两个MSET来设置相同的键，那么操作的结果也只会是两次MSET中后一次的结果，而不会是混杂的结果。

(3) 批量添加： MSETNX Key Value [Key Value…]。

一次性添加多个键-值对。如果任何一个键已经存在，那么这个操作中全部的添加都不会执行。所以，当使用MSETNX时，要么全部键被添加，要么全部不被添加。这个命令是在MSET命令后面增加了一个后缀NX（if Not eXist），表示只有键不存在的时候才会设置键的值。

(4) 获取： GET Key。

使用GET命令可以取得单个键所绑定的值，可以是字符串、数字、二进制数据。

(5) 批量获取： MGET Key[Key…]。

在GET命令的前面增加一个前缀M，表示一次获取多个（Multi）键的值。使用MGET命令一次性获取多个值，这和多次使用GET命令取得单个值有什么区别呢？MGET主要在于减少网络传输的次数，提升了性能。

(6) 获取长度： STRLEN Key。

返回键对应的String的长度，如果键对应的不是String，则报错。如果键不存在，则返回0。

(7) 为键对应的整数值增加1： INCR Key。

(8) 为键对应的整数值减少1: DECR Key。

(9) 为键对应的整数值增加increment: INCRBY Key increment。

(10) 为键对应的整数值减少decrement: DECRBY Key decrement。

(11) 为键对应的浮点数值增加increment: INCRBYFLOAT Key increment。

说明一下: Redis并没有为浮点数值减少decrement的操作DECRBYFLOAT。如果要为浮点数值减少decrement, 那么把INCRBYFLOAT命令的increment设成负值即可。

```
127.0.0.1:6379>set foo 1.0
OK
127.0.0.1:6379>incrbyfloatfoo10.01
"11.01"
127.0.0.1:6379>incrbyfloatfoo -5.0
"6.01"
```

在上面的例子中, 首先为foo设置了一个浮点数, 然后使用INCRBYFLOAT命令为foo的值加上10.01; 最后将INCRBYFLOAT命令的参数设置成负数, 为foo的值减少5.0。

14.2.2 List

Redis的List（列表）类型是基于双向链表实现的，可以支持正向、反向查找和遍历。从用户角度来说，List是简单的字符串列表，字符串按照添加的顺序排序。可以添加一个元素到List列表的头部（左边）或者尾部（右边）。一个List最多可以包含 $2^{32}-1$ 个元素（最多可超过40亿个元素）。

List的典型应用场景：网络社区中最新的发帖列表、简单的消息队列、最新新闻分页列表、博客的评论列表、排队系统等。举个具体的例子，在“双11”秒杀、抢购这样的大型活动中，短时间内有大量的用户请求发向服务器，而后台的程序不可能立刻响应每一个用户的请求，有什么好的办法来解决这个问题呢？我们需要一个排队系统。根据用户的请求时间将用户的请求放入List中，后台程序依次从列表中获取任务，处理并将结果返回到结果队列。换句话说，通过List，可以将并行的请求转换成串行的任务队列，之后依次处理。总体来说，List的使用场景是非常多的。

下面对List类型的主要操作进行简要介绍。

(1) 右推入：RPUSH Key Value [Value ...]。

右推入也叫后推入。将一个或多个值依次推入到列表的尾部（右端）。如果键不存在，那么右推入之前会先自动创建一个空的列表。如果键的值不是一个List类型，则会返回一个错误。如果同时右推入多个值，则多个值会依次从尾部进入列表。RPUSH命令的返回值为操作完成后列表包含的元素量。RPUSH时间复杂度为 $O(N)$ ，如果只推入一个值，那么命令的复杂度为 $O(1)$ 。

(2) 左推入：LPUSH Key Value [Value ...]。

左推入也叫前推入。这个命令和RPUSH几乎一样，只是推入元素的地点不同，是从列表的头部（左侧）推入的。

(3) 左弹出：LPOP Key。

PUSH用于增加元素，而POP操作则是获取元素并删除。LPOP命令是从列表的左边（前端）获取并移除一个元素，复杂度 $O(1)$ 。如果列表为空，则返回nil。

(4) 右弹出：RPOP key。

与LPOP功能基本相同，是从列表的右边（后端）获取并移除一个元素，复杂度 $O(1)$ 。

(5) 获取列表的长度：LLEN Key。

(6) 获取列表指定位置上的元素：LINDEX Key index。

(7) 获取指定索引范围之内的所有元素：LRANGE Key start stop。

(8) 设置指定索引上的元素：LSET Key index Value。

下边是一个使用以上命令操作List的例子：

```
127.0.0.1:6379> del foo
(integer) 1
127.0.0.1:6379> rpush foo a b c d e f g
(integer) 7
127.0.0.1:6379> llen foo
```

```
(integer) 7  
127.0.0.1:6379>lrange foo 0 4  
1) "a"  
2) "b"  
3) "c"  
4) "d"  
5) "e"  
127.0.0.1:6379>lindex foo 3  
"d"  
127.0.0.1:6379>lindex foo -1  
"g"  
127.0.0.1:6379>lindex foo 6  
"g"
```

List的下标或索引是从0开始的，下标为负的时候是从后向前数。-1表示最后一个元素。当下标超出边界时，会返回nil。

14.2.3 Hash

Redis中的Hash（哈希表）是一个String类型的Field（字段）和Value（值）之间的映射表，类似于Java中的HashMap。一个哈希表由多个字段-值对（Field-Value Pair）组成，值可以是文字、整数、浮点数或者二进制数据。在同一个哈希表中，每个字段的名称必须是唯一的。下面对哈希表的主要操作进行简要介绍。

（1）设置字段-值：HSET Key Field Value。

在缓存键为Key的哈希表中，给Field（字段）设置Value。如果Field之前没有设置值，那么命令返回1；如果Field已经有关联值，那么命令用新值覆盖旧值，并返回0。

(2) 获取字段-值：HGET Key Field。

在缓存键为Key的哈希表中，返回Field所关联的值（Value）。如果Field没有关联值，那么返回nil。

(3) 检查字段是否存在：HEXISTS Key Field。

在缓存键为Key的哈希表中，查看指定Field（字段）是否存在：存在则返回1，不存在则返回0。

(4) 删除给指定的字段：HDEL Key Field [Field …]。

在缓存键为Key的哈希表中，删除一个或多个指定Field，以及那些Field所关联的值，不存在的Field将被忽略。该命令返回哈希表被成功删除的字段-值（Field-Value）的数量。

(5) 查看字段是否存在：HEXISTS Key Field。

在缓存键为Key的哈希表中，查看指定的Field是否存在。

(6) 获取字段：HKEYS Key。

在缓存键为Key的哈希表中，获取所有的Field。

(7) 获取所有值：HVALS Key。

在缓存键为Key的哈希表中，获取所有的Value。

下面使用一个哈希表来缓存系统的IP、端口等配置信息，实例如下：

```
127.0.0.1:6379> del foo
(integer) 1
127.0.0.1:6379>hset config ip 127.0.0.1
(integer) 1
127.0.0.1:6379>hset config port 8080
(integer) 1
127.0.0.1:6379>hset config maxalive 5000
(integer) 1
127.0.0.1:6379>hkeys config
1) "ip"
2) "port"
3) "maxalive"
127.0.0.1:6379>hvals config
1) "127.0.0.1"
2) "8080"
3) "5000"
127.0.0.1:6379>hexists config timeout
(integer) 0
```

使用哈希列表的好处：

(1) 将数据集中存放。通过哈希表可以将一些相关的信息存储在同一个缓存键中，不仅方便数据管理，还可以尽量避免误操作的发生。

(2) 避免键名冲突。在介绍缓存键的命名规范时，可以使用冒号分隔符来避免命名冲突，但更好的避免冲突的办法是直接使用哈希键来存储字段-值对数据。

(3) 减少键的内存占用。在一般情况下，保存相同数量的字段-值对信息，使用哈希键比使用字符串键更节约内存。因为Redis创建一个键都带有很多的附加管理信息（例如这个键的类型、最后一次被访问的时间等），所以缓存的键越多，耗费的内存就越多，花在管理数据库键上的CPU处理时间也会越多。

应该尽量使用哈希表而不是字符串键来缓存字段-值对数据，其总体的优势为方便管理、能够避免键名冲突、能够节约内存。

14.2.4 Set

Set（集合）也是一个列表，特殊之处在于它是可以自动去掉重复元素的。Set类型的使用场景是：当需要存储一个列表，而又不希望有重复的元素（例如ID的集合）时，使用Set是一个很好的选择。Set类型拥有一个命令，可用于判断某个元素是否存在，而List类型并没有这种功能的命令。

通过Set类型的命令可以快速地向集合添加元素，或者从集合里面删除元素，也可以对多个Set进行集合运算，例如并集、交集、差集。

(1) 添加元素：SADD Key member1 [member2 …]。

可以向Key集合中添加一个或多个成员。

(2) 移除元素：SREM Key member1 [member2…]。

从Key集合中移除一个或多个成员。

(3) 判断某个元素: SISMEMBER Key member。

判断member元素是否为Key集合的成员。

在下面的例子中, 向foo集合中增加5个用户ID, 然后删除一个, 具体操作如下:

```
127.0.0.1:6379> del foo
(integer) 0
127.0.0.1:6379>sadd foo user0001
(integer) 1
127.0.0.1:6379>sadd foo user0002 user0003 user0004 user0005
(integer) 4
127.0.0.1:6379>srem foo user0005
(integer) 1
127.0.0.1:6379>sismember foo user0005
(integer) 0
127.0.0.1:6379>sismember foo user0004
(integer) 1
```

(4) 获取集合的成员数: SCARD Key。

(5) 获取集合中的所有成员: SMEMBERS Key。

```
127.0.0.1:6379>scard foo
(integer) 4
127.0.0.1:6379>smembers foo
```

- 1) "user0004"
- 2) "user0001"
- 3) "user0003"
- 4) "user0002"

14.2.5 ZSet

ZSet（有序集合）和Set（集合）的使用场景类似，区别是Zset会根据提供的score参数自动排序。当需要一个不重复且有序的集合列表时，可以选择ZSet类型，常用案例是游戏中的排行榜。

ZSet和Set不同的是，ZSet的每个元素都关联着一个分值(Score)，这是一个浮点数格式的关联值。ZSet会根据分值按从小到大的顺序来排列各个元素。

(1) 添加成员：ZADD Key Score1 member1 [ScoreN memberN...]。

向有序集合Key中添加一个或多个成员。如果memberN已经存在，则更新已存在成员的分数。

(2) 移除元素：ZREM Key member1 [memberN...].

从有序集合Key中移除一个或多个成员。

(3) 取得分数：ZSCORE Key member。

从有序集合Key中取得member成员的分数值。

(4) 取得成员排序: ZRANK Key member。

从有序集合Key中取得member成员的分数值的排名。

(5) 成员加分: ZINCRBY Key Score member。

在有序集合Key中对指定成员的分数加上增量Score。

(6) 区间获取: ZRANGEBYSCORE Key min max [WITHSCORES] [LIMIT]。

从有序集合Key中获取指定分数区间范围内的成员。WITHSCORES表示带上分数值返回; LIMIT选项可以用于翻页, 功能类似于MySQL查询的limit选项, 有offset、count两个参数值, 表示返回的偏移量和成员数量。

在默认情况下, min和max表示的范围是[min, max], 这组范围是闭区间范围, 而不是开区间范围, 即 $\min \leq score \leq \max$ 内的成员将被返回, 可以使用 -inf (负无穷) 和+inf (正无穷) 分别表示分数范围的最小值和最大值。

(7) 获取成员数: ZCARD Key。

(8) 区间计数: ZCOUNT Key min max。

在有序集合Key中计算指定分数区间的成员数。

下面以一个薪酬排序的有序集合为例演示一下上述命令的使用:

```
127.0.0.1:6379> del foo
```

```
(integer) 1
```

```
127.0.0.1:6379>zadd salary 1000 user0001
(integer) 1
127.0.0.1:6379>zadd salary 2000 user0002
(integer) 1
127.0.0.1:6379>zadd salary 3000 user0003
(integer) 1
127.0.0.1:6379>zadd salary 4000 user0004
(integer) 1
127.0.0.1:6379> type salary

127.0.0.1:6379>zrank salary user0004
(integer) 3
127.0.0.1:6379>zrank salary user0001
(integer) 0
127.0.0.1:6379>zrangebyscore salary 3000 +inf
1) "user0003"
2) "user0004"
```

14.3 Jedis基础编程的实战案例

Jedis是一个高性能的开源Java客户端，是Redis官方推荐的Java开发工具。如果要在Java开发中访问Redis缓存服务器，就必须对Jedis熟悉才能编写出“漂亮”的代码。Jedis的项目地址为<https://github.com/alphazero/jredis>。

使用Jedis，可以在Maven的pom文件中增加以下依赖：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>${redis.version}</version>
</dependency>
```

本书演示案例所使用的依赖版本为2.9.0。

Jedis基本的使用十分简单，在使用时构建Jedis对象即可。一个Jedis对象代表一条和Redis服务进行连接的Socket通道。使用完Jedis对象之后，可以调用Jedis.close()方法把连接关闭。创建Jedis对象时，可以指定Redis服务的host、port和password。大致的伪代码如下：

```
Jedis jedis = new Jedis("localhost", 6379); //指定Redis服务的主机和端口
jedis.auth("xxxx"); //如果Redis服务连接需要密码，就设置密码
```

```
//访问Redis服务  
jedis.close();           //使用完，就关闭连接
```

14.3.1 Jedis操作String

Jedis的String操作函数和Redis客户端操作String的命令基本上是一一对应的。正因为如此，本小节不对Jedis的String操作函数进行清单式的说明，只设计了一个比较全面的String操作的示例程序，其目的是演示一下这些函数的使用。

Jedis操作String的具体示例程序代码如下：

```
package com.crazymakercircle.redis.jedis;  
  
//省略import  
  
public class StringDemo {  
  
    /**  
     * Jedis字符串数据类型的相关命令  
     */  
  
    @Test  
    public void operateString() {  
        Jedis jedis = new Jedis("localhost", 6379);  
        //如果返回 pong就代表链接成功  
        Logger.info("jedis.ping()：" + jedis.ping());  
        //设置key0的值为 123456  
        jedis.set("key0", "123456");  
        //返回数据类型String  
        Logger.info("jedis.type(key0)：" + jedis.type("key0"));  
    }  
}
```

```
//取得值  
Logger.info("jedis.get(key0): " + jedis.get("key0"));  
  
//Key是否存在  
Logger.info("jedis.exists(key0):" +  
jedis.exists("key0"));  
  
//返回Key的长度  
Logger.info("jedis.strlen(key0): " +  
jedis.strlen("key0"));  
  
//返回截取字符串，范围"0,-1" 表示截取全部  
Logger.info("jedis.getrange(key0): " +  
jedis.getrange("key0", 0, -1));  
  
//返回截取字符串，范围"1,4" 表示区间[1,4]  
Logger.info("jedis.getrange(key0): " +  
jedis.getrange("key0", 1,  
4));  
  
//追加字符串  
Logger.info("jedis.append(key0): " +  
jedis.append("key0", "appendStr"));  
  
Logger.info("jedis.get(key0): " + jedis.get("key0"));  
  
//重命名  
jedis.rename("key0", "key0_new");  
  
//判断Key是否存在  
Logger.info("jedis.exists(key0): " +  
jedis.exists("key0"));  
  
//批量插入
```

```
jedis.mset("key1", "val1", "key2", "val2", "key3",
"100");
    //批量取出
Logger.info("jedis.mget(key1,key2,key3): " +
            jedis.mget("key1", "key2",
"key3"));
    //删除
Logger.info("jedis.del(key1): " + jedis.del("key1"));
Logger.info("jedis.exists(key1): " +
jedis.exists("key1"));
    //取出旧值并设置新值
Logger.info("jedis.getSet(key2): " +
            jedis.getSet("key2",
"value3"));
    //自增1
Logger.info("jedis.incr(key3): " + jedis.incr("key3"));
    //自增15
Logger.info("jedis.incrBy(key3): " +
jedis.incrBy("key3", 15));
    //自减1
Logger.info("jedis.decr(key3): " + jedis.decr("key3"));
    //自减15
Logger.info("jedis.decrBy(key3): " +
            jedis.decrBy("key3",
15));
    //浮点数加
Logger.info("jedis.incrByFloat(key3): " +
```

```
jedis.incrByFloat("key3",1.1));  
  
        //返回0， 只有在Key不存在的时候才设置  
        Logger.info("jedis.setnx(key3): " +  
                    jedis.setnx("key3",  
"existVal"));  
        Logger.info("jedis.get(key3): " +  
jedis.get("key3")); //3.1  
  
        //只有Key都不存在的时候才设置， 这里返回 null  
        Logger.info("jedis.msetnx(key2,key3): "  
                    + jedis.msetnx("key2", "exists1", "key3",  
"exists2"));  
        Logger.info("jedis.mget(key2,key3): " +  
                    jedis.mget("key2", "key3"));  
  
        //设置key， 2 秒后失效  
        jedis.setex("key4", 2, "2 seconds is invalid");  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        //2 seconds is invalid  
        Logger.info("jedis.get(key4): " + jedis.get("key4"));
```

```

jedis.set("key6", "123456789");
//下标从0开始，从第三位开始，用新值覆盖旧值
jedis.setrange("key6", 3, "abcdefg");
//返回：123abcdefg
Logger.info("jedis.get(key6): " + jedis.get("key6"));

//返回所有匹配的键
Logger.info("jedis.get(key*): " +
            jedis.keys("key*"));

jedis.close();
}

}

```

这个示例程序的运行结果篇幅较长，本节就不贴出了。建议大家运行源代码工程，查看并分析示例程序的运行结果。

14.3.2 Jedis操作List

Jedis的List操作函数和Redis客户端操作List的命令基本上也是一一对应的。也正因为如此，本小节也不对Jedis的List操作函数做清单式的说明，只设计一个比较全面的List操作的示例程序，演示一下这些函数的使用，具体如下：

```

package com.crazymakercircle.redis.jedis;
//...
public class ListDemo {
    /**

```

```
* Redis列表是简单的字符串列表，按照插入顺序排序
*/
@Test
public void operateList() {
    Jedis jedis = new Jedis("localhost");
    Logger.info("jedis.ping(): " + jedis.ping());
    jedis.del("list1");

    //从List尾部添加3个元素
    jedis.rpush("list1", "zhangsan", "lisi", "wangwu");

    //取得类型，list
    Logger.info("jedis.type(): " + jedis.type("list1"));

    //遍历区间[0,-1]，取得全部的元素
    Logger.info("jedis.lrange(0,-1): " +
                jedis.lrange("list1", 0, -1));
    //遍历区间[1,2]，取得区间的元素
    Logger.info("jedis.lrange(1,2): " +
                jedis.lrange("list1", 1, 2));

    //获取List的长度
    Logger.info("jedis.llen(list1): " +
+jedis.llen("list1"));
    //获取下标为 1 的元素
    Logger.info("jedis.lindex(list1,1): " +
+jedis.lindex("list1", 1));
}
```

```
//左侧弹出元素  
Logger.info("jedis.lpop(): " +jedis.lpop("list1"));  
  
//右侧弹出元素  
Logger.info("jedis.rpop(): " +jedis.rpop("list1"));  
  
//设置下标为0的元素val  
jedis.lset("list1", 0, "lisi2");  
  
//最后，遍历区间[0,-1]，取得全部的元素  
Logger.info("jedis.lrange(0,-1): " +  
            jedis.lrange("list1",  
0, -1));  
  
jedis.close();  
}  
}
```

运行示例程序，结果如下：

```
[main|ListDemo.operateList] |>jedis.ping(): PONG  
[main|ListDemo.operateList] |>jedis.type(): list  
[main|ListDemo.operateList] |>jedis.lrange(0,-1): [zhangsan,  
lisi, wangwu]  
[main|ListDemo.operateList] |>jedis.lrange(1,2): [lisi, wangwu]  
[main|ListDemo.operateList] |>jedis.llen(list1): 3  
[main|ListDemo.operateList] |>jedis.lindex(list1,1): lisi  
[main|ListDemo.operateList] |>jedis.lpop(): zhangsan  
[main|ListDemo.operateList] |>jedis.rpop(): wangwu  
[main|ListDemo.operateList] |>jedis.lrange(0,-1): [lisi2]
```

建议大家运行源代码工程，查看并分析示例程序的运行结果，最后做到熟练地掌握这组函数。

14.3.3 Jedis操作Hash

Jedis的Hash（哈希表）操作函数和Redis客户端操作Hash哈希表的命令也是一一对应的。所以，本小节不再罗列Hash操作函数，仅设计一个比较全面的Hash操作的示例程序，演示一下这些函数的使用，具体如下：

```
package com.crazymakercircle.redis.jedis;  
//...  
public class HashDemo {  
    /**  
     * Redis Hash 哈希表是一个Field (String类型) 和Value值的映射  
     * 表  
     * Hash特别适合用于存储对象  
     * Redis 中每个Hash可以存储  $2^{32}-1$  (40多亿) 个字段-值对  
     */  
    @Test  
    public void operateHash() {  
        Jedis jedis = new Jedis("localhost");  
        jedis.del("config");  
        //设置Hash的 Field-Value对: ip=127.0.0.1  
        jedis.hset("config", "ip", "127.0.0.1");  
        //取得Hash的 Field关联的value  
        Logger.info("jedis.hget(): " + jedis.hget("config",
```

```
"ip"));

//取得类型: Hash
Logger.info("jedis.type(): " + jedis.type("config"));

//批量添加 field-value 对, 参数为java map
Map<String, String> configFields = new HashMap<String,
String>();
configFields.put("port", "8080");
configFields.put("maxalive", "3600");
configFields.put("weight", "1.0");
//执行批量添加
jedis.hmset("config", configFields);

//批量获取: 取得全部 field-value 对, 返回 Java map映射表
Logger.info("jedis.hgetAll(): " +
jedis.hgetAll("config"));

//批量获取: 取得部分 Field对应的value, 返回Java map
Logger.info("jedis.hmget(): " +
jedis.hmget("config", "ip", "port"));

//浮点数加: 类似于String的incrByFloat
jedis.hincrByFloat("config", "weight", 1.2);
Logger.info("jedis.hget(weight): " +
jedis.hget("config",
"weight"));

//获取所有的Key
```

```

        Logger.info("jedis.hkeys(config): " +
jedis.hkeys("config"));

        //获取所有的val
        Logger.info("jedis.hvals(config): " +
jedis.hvals("config"));

        //获取长度
        Logger.info("jedis.hlen(): " + jedis.hlen("config"));

        //判断Field是否存在
        Logger.info("jedis.hexists(weight): " +
                jedis.hexists("config",
"weight"));

        //删除一个Field
        jedis.hdel("config", "weight");
        Logger.info("jedis.hexists(weight): " +
                jedis.hexists("config",
"weight"));

        jedis.close();
    }
}

```

运行示例程序，结果如下：

```

[main|HashDemo.operateHash] |>jedis.hget(): 127.0.0.1
[main|HashDemo.operateHash] |>jedis.type(): hash
[main|HashDemo.operateHash] |>jedis.hgetAll(): {port=8080,
weight=1.0, maxalive=3600, ip=127.0.0.1}

```

```
[main|HashDemo.operateHash] |>jedis.hmget(): [127.0.0.1, 8080]
[main|HashDemo.operateHash] |>jedis.hget(): 2.2
[main|HashDemo.operateHash] |>jedis.hkeys(): [weight, maxalive,
port, ip]
[main|HashDemo.operateHash] |>jedis.hvals(): [127.0.0.1, 8080,
2.2, 3600]
[main|HashDemo.operateHash] |>jedis.hlen(): 4
[main|HashDemo.operateHash] |>jedis.hexists(weight): true
[main|HashDemo.operateHash] |>jedis.hexists(weight): false
```

建议大家运行源代码工程，查看并分析示例程序的运行结果，最后做到熟练地掌握这组Hash操作函数。

14.3.4 Jedis操作Set

Jedis的Set操作函数和Redis客户端操作Set的命令基本上可以一一对应。本小节不再罗列Jedis操作Set集合的函数，仅设计一个比较简单的Set操作的示例程序，演示一下这些函数的使用，具体如下：

```
package com.crazymakercircle.redis.jedis;
//省略import
public class SetDemo {
    /**
     * Redis 的 Set是 String 类型的无序集合
     * 集合成员是唯一的，集合中不能出现重复的元素
     * Set集合是通过哈希表实现的，添加、删除、查找的复杂度都是 O(1)
     */
}
```

```
@Test  
public void operateSet() {  
    Jedis jedis = new Jedis("localhost");  
    jedis.del("set1");  
    Logger.info("jedis.ping(): " + jedis.ping());  
    Logger.info("jedis.type(): " + jedis.type("set1"));  
  
    //sadd函数：向集合添加元素  
    jedis.sadd("set1", "user01", "user02", "user03");  
    //smembers函数：遍历所有元素  
    Logger.info("jedis.smembers(): " +  
    jedis.smembers("set1"));  
    //scard函数：获取集合元素个数  
    Logger.info("jedis.scard(): " + jedis.scard("set1"));  
    //sismember判断是否是集合元素  
    Logger.info("jedis.sismember(user04): " +  
  
    jedis.sismember("set1", "user04"));  
    //srem函数：移除元素  
    Logger.info("jedis.srem(): " +  
    jedis.srem("set1", "user02",  
    "user01"));  
    //smembers函数：遍历所有元素  
    Logger.info("jedis.smembers(): " +  
    jedis.smembers("set1"));  
    jedis.close();
```

```
    }  
}  
}
```

运行示例程序，结果如下：

```
[main|SetDemo.operateSet] |>jedis.ping(): PONG  
[main|SetDemo.operateSet] |>jedis.type(): none  
[main|SetDemo.operateSet] |>jedis.smembers(): [user02, user03,  
user01]  
[main|SetDemo.operateSet] |>jedis.scard(): 3  
[main|SetDemo.operateSet] |>jedis.sismember(user04): false  
[main|SetDemo.operateSet] |>jedis.srem(): 2  
[main|SetDemo.operateSet] |>jedis.smembers(): [user03]
```

建议大家运行源代码工程，查看并分析示例的运行结果，最后做到熟练地掌握Set操作函数。

14.3.5 Jedis操作ZSet

Jedis的ZSet操作函数和Redis客户端操作ZSet的命令基本上可以一一对应。本小节不再罗列Jedis操作ZSet的函数，仅设计一个比较简单的有序集合操作的示例程序，演示一下这些函数的使用，具体如下：

```
package com.crazymakercircle.redis.jedis;  
//...  
public class ZSetDemo {  
    /**
```

- * ZSet有序集合和Set集合都是String类型元素的集合，且不允许重复的元素
- * 不同的是ZSet的每个元素都会关联一个double类型的分数，用于从小到大排序

- * 集合中最大的成员数为 $2^{32}-1$ (4294967295，每个集合可存储40多亿个元素)

*/

@Test

```
public void operateZset() {  
    Jedis jedis = new Jedis("localhost");  
    Logger.info("jedis.ping (): " + jedis.ping());  
  
    jedis.del("salary");  
  
    Map<String, Double> members = new HashMap<String,  
Double>();  
  
    members.put("u01", 1000.0);  
    members.put("u02", 2000.0);  
    members.put("u03", 3000.0);  
    members.put("u04", 13000.0);  
    members.put("u05", 23000.0);  
  
    //批量添加元素，类型为Java map映射表  
    jedis.zadd("salary", members);  
  
    //获取类型zset  
    Logger.info("jedis.type (): " + jedis.type("salary"));  
  
    //获取集合元素的个数  
    Logger.info("jedis.zcard (): " + jedis.zcard("salary"));  
  
    //按照下标[起,止]遍历元素
```

```
    Logger.info("jedis.zrange(): " +
                jedis.zrange("salary", 0,
-1));
    //按照下标 [起,止] 倒序遍历元素

    Logger.info("jedis.zrevrange(): " +
                jedis.zrevrange("salary",
0, -1));

    //按照分数 (薪资) [起,止] 遍历元素
    Logger.info("jedis.zrangeByScore(): " +
                jedis.zrangeByScore("salary", 1000,
10000));
    //按照薪资 [起,止] 遍历元素, 带分数返回
    Set<Tuple> res0 = jedis.zrangeByScoreWithScores(
                    "salary", 1000,
10000);
    for (Tuple temp : res0) {
        Logger.info("Tuple.get(): " + temp.getElement() + "
-> " +
                    temp.getScore());
    }
    //按照分数 [起,止] 倒序遍历元素
    Logger.info("jedis.zrevrangeByScore(): "
                + jedis.zrevrangeByScore("salary",
1000, 4000));
    //获取元素 [起,止] 分数区间的元素数量
    Logger.info("jedis.zcount(): " +
```

```
jedis.zcount("salary", 1000,
4000));

//获取元素score值: 薪资
Logger.info("jedis.zscore(): " + jedis.zscore("salary",
"u01"));

//获取元素的下标
Logger.info("jedis.zrank(u01): " +
jedis.zrank("salary", "u01"));

//倒序获取元素的下标
Logger.info("jedis.zrevrank(u01): " +
jedis.zrevrank("salary",
"u01"));

//删除元素
Logger.info("jedis.zrem(): " +
jedis.zrem("salary", "u01",
"u02"));

//删除元素, 通过下标范围
Logger.info("jedis.zremrangeByRank(): " +
jedis.zremrangeByRank("salary", 0,
1));

//删除元素, 通过分数范围
Logger.info("jedis.zremrangeByScore(): "
+ jedis.zremrangeByScore("salary", 20000,
30000));

//按照下标[起,止]遍历元素
Logger.info("jedis.zrange(): " + jedis.zrange("salary",
```

```

        0, -1));
    }

    Map<String, Double> members2 = new HashMap<String,
Double>();
    members2.put("u11", 1136.0);
    members2.put("u12", 2212.0);
    members2.put("u13", 3324.0);
    //批量添加元素
    jedis.zadd("salary", members2);
    //增加指定分数
    Logger.info("jedis.zincrby(10000): " +
                jedis.zincrby("salary", 10000,
"u13"));
    //按照下标[起,止]遍历元素
    Logger.info("jedis.zrange(): " + jedis.zrange("salary",
0, -1));
}

jedis.close();
}
}

```

在示例程序中，有一个salary（薪资）的ZSet，ZSet的键为用户ID，ZSet的score分数值保存的是用户的薪资。运行这个示例程序，结果如下：

```

[main|ZSetDemo.operateZset] |>jedis.ping(): PONG
[main|ZSetDemo.operateZset] |>jedis.type(): zset

```

```
[main|ZSetDemo.operateZset] |>jedis.zcard(): 5
[main|ZSetDemo.operateZset] |>jedis.zrange(): [u01, u02, u03,
u04, u05]
[main|ZSetDemo.operateZset] |>jedis.zrangeByScore(): [u01, u02,
u03]
[main|ZSetDemo.operateZset] |>Tuple.get(): u01 -> 1000.0
[main|ZSetDemo.operateZset] |>Tuple.get(): u02 -> 2000.0
[main|ZSetDemo.operateZset] |>Tuple.get(): u03 -> 3000.0
[main|ZSetDemo.operateZset] |>jedis.zrevrange(): [u05, u04,
u03, u02, u01]
[main|ZSetDemo.operateZset] |>jedis.zrevrangeByScore(): []
[main|ZSetDemo.operateZset] |>jedis.zscore(): 1000.0
[main|ZSetDemo.operateZset] |>jedis.zcount(): 3
[main|ZSetDemo.operateZset] |>jedis.zrank(u01): 0
[main|ZSetDemo.operateZset] |>jedis.zrevrank(u01): 4
[main|ZSetDemo.operateZset] |>jedis.zrem(): 2
[main|ZSetDemo.operateZset] |>jedis.zremrangeByRank(): 2
[main|ZSetDemo.operateZset] |>jedis.zremrangeByScore(): 1
[main|ZSetDemo.operateZset] |>jedis.zrange(): []
[main|ZSetDemo.operateZset] |>jedis.get(): 13324.0
[main|ZSetDemo.operateZset] |>jedis.zrange(): [u11, u12, u13]
```

建议大家运行源代码工程，查看并分析示例的运行结果，最后做到熟练地掌握这组Zset的操作函数。

14.4 JedisPool连接池的实战案例

使用Jedis API可以方便地在Java程序中操作Redis，就像通过 JDBC API操作数据库一样。但是，仅仅实现这一点还是不够的。因为数据库连接的底层是一条Socket通道，其创建和销毁很耗时间，需要有三次握手和四次挥手。

在数据库连接过程中，为了防止数据库连接的频繁创建、销毁带来的性能损耗，常常会用到连接池（Connection Pool），例如淘宝的Druid连接池、Tomcat的DBCP连接池。Jedis连接和数据库连接一样，也需要使用连接池来管理。

Jedis开源库提供了一个负责管理Jedis连接对象的池，名为 JedisPool类，位于redis.clients.jedis包中。

14.4.1 JedisPool的配置

在使用JedisPool类创建Jedis连接池之前，首先要了解其配置类——JedisPoolConfig配置类，它也位于redis.clients.jedis包中。这个配置类负责配置JedisPool的参数。JedisPoolConfig配置类涉及很多与连接管理和使用有关的参数。下面对它的一些重要参数进行说明。

(1) maxTotal：资源池中最大的连接数，默认值为8。

(2) maxIdle：资源池允许最大空闲的连接数，默认值为8。

(3) `minIdle`: 资源池确保最少空闲的连接数，默认值为0。如果JedisPool开启了空闲连接的有效性检测，并且空闲连接无效，就销毁。销毁连接后，连接数量就少了，如果小于`minIdle`，就新建连接，维护数量不少于`minIdle`的数量。`minIdle`确保线程池中有最小的空闲Jedis实例的数量。

(4) `blockWhenExhausted`: 当资源池用尽后，调用者是否要等待，默认值为`true`。当为`true`时，`maxWaitMillis`才会生效。

(5) `maxWaitMillis`: 当资源池连接用尽后，调用者的最大等待时间（单位为毫秒）。默认值为-1，表示永不超时，不建议使用默认值。

(6) `testOnBorrow`: 向资源池借用连接时，是否做有效性检测（ping命令），如果是无效连接，会被移除，默认值为`false`，表示不做检测。如果为`true`，则得到的Jedis实例均是可用的。在业务量小的应用场景下，建议设置为`true`，确保连接可用；在业务量很大的应用场景下，建议设置为`false`（默认值），少一次ping命令的开销，有助于提升性能。

(7) `testOnReturn`: 向资源池归还连接时，是否做有效性检测（ping命令），如果是无效连接，会被移除，默认值为`false`，表示不做检测。同样，在业务量很大的应用场景下，建议设置为`false`（默认值），少一次ping命令的开销。

(8) `testWhileIdle`: 如果为`true`，就表示用一个专门的线程对空闲的连接进行有效性的检测扫描，若连接的有效性检测失败，则表示监测到无效连接，会从资源池中移除。默认值为`true`，表示进行空

闲连接的检测。这个选项存在一个附加条件，需要空闲扫描间隔时间配置项timeBetweenEvictionRunsMillis的值大于0；否则，testWhileIdle不会生效。

(9) timeBetweenEvictionRunsMillis：表示两次空闲连接扫描的间隔时间，默认为30000毫秒，也就是30秒钟。

(10) minEvictableIdleTimeMillis：表示一个Jedis连接至少停留在空闲状态的最短时间，然后才能被空闲连接扫描线程进行有效性检测，默认值为60000毫秒，即60秒。也就是说，在默认情况下，一条Jedis连接只有在空闲60秒后才会参与空闲线程的有效性检测。这个选项存在一个附加条件，需要在timeBetweenEvictionRunsMillis大于0时才会生效。也就是说，如果不启动空闲检测线程，这个参数也没有什么意义。

(11) numTestsPerEvictionRun：表示空闲检测线程每次最多扫描的Jedis连接数，默认值为-1，表示扫描全部的空闲连接。

空闲扫描的选项在JedisPoolConfig的构造器中都有默认值，具体如下：

```
package redis.clients.jedis;  
import org.apache.commons.pool2.impl.GenericObjectPoolConfig;  
public class JedisPoolConfig extends GenericObjectPoolConfig {  
    public JedisPoolConfig() {  
        this.setTestWhileIdle(true);  
        this.setMinEvictableIdleTimeMillis(60000L);  
        this.setTimeBetweenEvictionRunsMillis(30000L);  
    }  
}
```

```
        this.setNumTestsPerEvictionRun(-1);  
    }  
}
```

(12) `jmxEnabled`: 是否开启JMX监控， 默认值为true， 建议开启。

有一个实际的问题：如何推算一个连接池的最大连接数 `maxTotal`? 实际上，这是一个很难精准回答的问题，主要是依赖的因素比较多。大致的推算方法是：业务QPS/单连接的QPS =最大连接数。

如何推算单个Jedis连接的QPS呢？假设一个Jedis命令操作的时间约为5ms（包含borrow + return + Jedis执行命令 + 网络延迟），那么单个Jedis连接的QPS大约是 $1000/5 = 200$ 。如果业务期望的QPS是100000，则需要的最大连接数为 $100000/200 = 500$ 。

事实上，上面的估算仅仅是理论值。在实际的生产场景中，还要预留一些资源，通常来讲所配置的`maxTotal`要比理论值大一些。

如果连接数确实太多，可以考虑Redis集群，那么单个Redis节点的最大连接数的公式为：`maxTotal = 预估的连接数 / nodes 节点数`。

说明

在并发量不大时，`maxTotal`设置过高会导致不必要的连接资源的浪费。可以根据实际总QPS和nodes节点数合理评估每个节点所使用的最大连接数。

再看一个问题：如何推算连接池的最大空闲连接数maxIdle值？

实际上，maxTotal只是给出了一个连接数量的上限，maxIdle实际上才是业务可用的最大连接数。从这个层面来说，maxIdle不能设置过小，否则会有创建、销毁连接的开销。使得连接池达到最佳性能的设置是maxTotal = maxIdle，应尽可能地避免由于频繁地创建和销毁Jedis连接所带来的连接池性能的下降。

14.4.2 JedisPool的创建和预热

创建JedisPool连接池的一般步骤为：首先，创建一个JedisPoolConfig配置实例；然后，以JedisPoolConfig实例、Redis IP、Redis端口和其他可选选项（如超时时间、Auth密码）为参数，构造一个JedisPool连接池实例。

```
package com.crazymakercircle.redis.jedisPool;  
//...  
public class JredisPoolBuilder {  
    public static final int MAX_IDLE = 50;  
    public static final int MAX_TOTAL = 50;  
    private static JedisPool pool = null;  
    //创建连接池  
    private static JedisPool buildPool() {  
        if (pool == null) {  
            long start = System.currentTimeMillis();  
            JedisPoolConfig config = new JedisPoolConfig();  
            config.setMaxTotal(MAX_TOTAL);  
        }  
    }  
}
```

```

        config.setMaxIdle(MAX_IDLE);

        config.setMaxWaitMillis(1000 * 10);

        //设置在borrow一个jedis实例时是否提前进行有效检测操作

        //如果为true，则得到的jedis实例均是可用的

        config.setTestOnBorrow(true);

        pool = new JedisPool(config, "127.0.0.1", 6379,
10000);

        long end = System.currentTimeMillis();

        Logger.info("buildPool毫秒数:", end - start);

    }

    return pool;

}

//...

}

```

虽然JedisPool定义了最大空闲资源数、最小空闲资源数，但是在创建的时候不会真的创建好Jedis连接并放到JedisPool池子里。这样会导致一个问题——刚创建好的连接池没有Jedis连接资源，在初次访问请求到来的时候才开始创建新的连接，会导致一定的时间开销。为了提升初次访问的性能，可以考虑在JedisPool创建后为JedisPool提前进行预热，一般以最小空闲数量作为预热数量。

```

package com.crazymakercircle.redis.jedisPool;

//...

public class JredisPoolBuilder {

    //连接池的预热

    public static void hotPool() {

```

```
long start = System.currentTimeMillis();

List<Jedis> minIdleJedisList = new ArrayList<Jedis>
(MAX_IDLE);

Jedis jedis = null;

for (int i = 0; i < MAX_IDLE; i++) {

    try {

        jedis = pool.getResource();
        minIdleJedisList.add(jedis);
        jedis.ping();
    } catch (Exception e) {

        Logger.error(e.getMessage());
    } finally {

    }
}

for (int i = 0; i < MAX_IDLE; i++) {

    try {

        jedis = minIdleJedisList.get(i);
        jedis.close();
    } catch (Exception e) {

        Logger.error(e.getMessage());
    } finally {

    }
}

long end = System.currentTimeMillis();

Logger.info(" hotPool毫秒数:", end - start);
```

```
    }  
}  
}
```

在自己定义的JedisPoolBuilder连接池Builder类中，创建好连接池实例，并且进行预热。然后，定义一个从连接池中获取Jedis连接的新方法——getJedis()，以供其他模块调用。

```
package com.crazymakercircle.redis.jedisPool;  
//...  
public class JredisPoolBuilder {  
//...  
    private static JedisPool pool = null;  
  
    static {  
        //创建连接池  
        buildPool();  
        //预热连接池  
        hotPool();  
    }  
    //省略buildPool、hotPool  
  
    //获取连接  
    public static Jedis getJedis() {  
        return pool.getResource();  
    }  
}
```

14.4.3 JedisPool的使用

获取JedisPool中的连接，可以调用自定义的getJedis()方法，间接通过pool.getResource()从连接池获取连接；也可以直接调用pool.getResource()方法。另外，JedisPool的池化连接在使用完后一定要调用close()方法关闭连接。这个关闭操作不是真正地关闭连接，而是归还给连接池。这一点和使用数据库连接池是一样的。一般来说，关闭操作放在finally代码段中，确保Jedis连接的关闭最终都会被执行到，使得连接归还到连接池。

```
package com.crazymakercircle.redis.jedisPool;  
//...  
  
public class JredisPoolTester {  
  
    public static final int NUM = 200;  
  
    public static final String ZSET_KEY = "zset1";  
  
    //测试删除  
  
    @Test  
  
    public void testDel() {  
  
        Jedis redis = null;  
  
        try {  
  
            redis = JredisPoolBuilder.getJedis();  
  
            long start = System.currentTimeMillis();  
  
            redis.del(ZSET_KEY);  
  
            long end = System.currentTimeMillis();  
  
            Logger.info("删除 zset1 毫秒数:", end - start);  
  
        } finally {
```

```
//使用后一定关闭，还给连接池
    if (redis != null) {
        redis.close();
    }
}
}

//...
}
```

由于Jedis类实现了java.io.Closeable接口，故而在JDK 1.7或者以上版本中可以使用try-with-resources语句，在其隐藏的finally部分自动调用close()方法。

```
package com.crazymakercircle.redis.jedisPool;
//...

public class JredisPoolTester {
    public static final int NUM = 200;
    public static final String ZSET_KEY = "zset1";
    //测试创建ZSet
    @Test
    public void testSet() {
        testDel(); //首先删除之前创建的ZSet
        try (Jedis redis = JredisPoolBuilder.getJedis()) {
            int loop = 0;
            long start = System.currentTimeMillis();
            while (loop < NUM) {
                redis.zadd(ZSET_KEY, loop, "field-" + loop);
            }
        }
    }
}
```

```
        loop++;

    }

    long end = System.currentTimeMillis();

    Logger.info("设置zSet :", loop, "次, 毫秒数:", end - start);

}

}

//...

}
```

使用try-with-resources和使用try-finally的写法是一样的，只是它会默认调用jedis.close()方法。这里优先推荐try-with-resources写法，因为比较简洁、干净。大家平时常用到的数据库连接、输入输出流的关闭都可以使用这种方法。

14.5 使用spring-data-redis完成CRUD的实战案例

无论是Jedis还是JedisPool，都只是完成对Redis操作极为基础的API，在不依赖任何中间件的开发环境中可以使用。但是，一般的Java开发都会使用Spring框架，可以使用spring-data-redis开源库来简化Redis操作的代码逻辑，做到最大限度的业务聚焦。

下面从缓存的应用场景入手，介绍spring-data-redis开源库的使用。

14.5.1 CRUD中应用缓存的场景

在普通CRUD应用场景中，很多情况下都需要同步操作缓存，推荐使用Spring的spring-data-redis开源库。注：CRUD是指Create（创建）、Retrieve（查询）、Update（更新）和Delete（删除）。

1. 创建缓存

在创建（Create）一个POJO实例时，对POJO实例进行分布式缓存，一般以“缓存前缀+ID”为缓存的键，POJO对象为缓存的值，直接缓存POJO的二进制字节。前提是：POJO必须可序列化，实现java.io.Serializable空接口。如果POJO不可序列化，也是可以缓存的，但是必须自己实现序列化的方式，例如使用JSON方式序列化。

2. 查询缓存

在查询 (Retrieve) 一个POJO实例时，首先应该根据POJO缓存的键从Redis缓存中返回结果。不存在时再去查询数据库，并将数据库的结果缓存起来。

3. 更新缓存

在更新 (Update) 一个POJO实例时，既需要更新数据库的POJO数据记录，也需要更新POJO的缓存记录。

4. 删除缓存

在删除 (Delete) 一个POJO实例时，既需要删除数据库的POJO数据记录，也需要删除POJO的缓存记录。

使用spring-data-redis开源库可以快速地完成上述缓存CRUD操作。

为了演示CRUD场景下Redis的缓存操作，首先定义一个简单的POJO实体类：聊天系统的用户类。此类拥有一些简单的属性，如uid和nickName，且这些属性都具备基本的getter和setter方法。

```
package com.crazymakercircle.im.common.bean;  
//...  
import java.io.Serializable;  
@Slf4j  
public class User implements Serializable {  
    String uid;  
    String devId;  
    String token;
```

```
    String nickName;  
    //省略 getter()、setter()、toString()等方法  
}
```

然后定义一个完成CRUD操作的Service接口，定义三个方法：

- (1) saveUser() 完成创建 (C)、更新 (U) 操作。
- (2) getUser() 完成查询 (R) 操作。
- (3) deleteUser() 完成删除 (D) 操作。

Service接口的代码如下：

```
package com.crazymakercircle.redis.springJedis;  
  
import com.crazymakercircle.im.common.bean.User;  
  
public interface UserService {  
  
    /**  
     * CRUD 的创建/更新  
     * @param user 用户  
     */  
  
    User saveUser(final User user);  
  
    /**  
     * CRUD 的查询  
     * @param id id  
     * @return 用户  
     */  
  
    User getUser(long id);
```

```
/*
 * CRUD 的删除
 * @param id id
 */
void deleteUser(long id);
}
```

定义完了Service接口之后，接下来定义Service服务的具体实现，不过这里聚焦的是如何通过spring-data-redis库使Service实现带缓存的功能。

14.5.2 配置spring-redis.xml

使用spring-data-redis库的第一步是要在Maven的pom文件中加上spring-data-redis库的依赖，具体如下：

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>${springboot}</version>
</dependency>
```

使用spring-data-redis库的第二步是配置spring-data-redis库的连接池实例和RedisTemplate模板实例。这是两个Spring Bean，既可以配置在项目统一的spring xml配置文件中，也可以编写一个独立的spring-redis.xml配置文件。这里使用的是第二种方式。

有关连接池实例和RedisTemplate模板实例的配置，节选如下：

```
<!--加载配置文件 -->
<context:property-placeholder
location="classpath:redis.properties"/>

<!--redis数据源 -->
<bean id="poolConfig"
class="redis.clients.jedis.JedisPoolConfig">
<!--最大空闲数 -->
<property name="maxIdle" value="${redis.maxIdle}"/>
<!--最大空连接数 -->
<property name="maxTotal" value="${redis.maxTotal}"/>
<!--最大等待时间 -->
<property name="maxWaitMillis" value="${redis.maxWaitMillis}"/>
<!--连接超时的时候是否阻塞，true表示阻塞，直到超过maxWaitMillis，默认为
true -->
<property name="blockWhenExhausted"
          value="${redis.blockWhenExhausted}"/>
<!--获取连接时，检测连接是否成功 -->
<property name="testOnBorrow" value="${redis.testOnBorrow}"/>
</bean>

<!-- Spring-redis连接池管理工厂 -->
<bean id="jedisConnectionFactory"
class="org.springframework.data.redis
      .connection.jedis.JedisConnectionFactory">
<!--IP地址-->
```

```
<property name="hostName" value="${redis.host}"/>
<!--端口号 -->
<property name="port" value="${redis.port}"/>
<!--连接池配置引用 -->
<property name="poolConfig" ref="poolConfig"/>
<!--usePool: 是否使用连接池 -->
<property name="usePool" value="true"/>
</bean>

<!--redis template definition -->
<bean id="redisTemplate" class="org.springframework.data.redis
.core.RedisTemplate">
<property name="connectionFactory"
ref="jedisConnectionFactory"/>
<property name="keySerializer">
<bean class="org.springframework.data.redis.serializer
.StringRedisSerializer"/>
</property>
<property name="valueSerializer">
<bean class="org.springframework.data.redis.serializer
.JdkSerializationRedisSerializer"/>
</property>
<property name="hashKeySerializer">
<bean class="org.springframework.data.redis.serializer
.StringRedisSerializer"/>
</property>
<property name="hashValueSerializer">
```

```
<bean class="org.springframework.data.redis.serializer  
       .JdkSerializationRedisSerializer"/>  
  
</property>  
<!--开启事务-->  
  
<property name="enableTransactionSupport" value="true">  
</property>  
</bean>  
  
<!--将redisTemplate封装成通用服务-->  
  
<bean id="springRedisService"  
      class="com.crazymakercircle.redis.springJedis.CacheOperationSer  
vice">  
  
<property name="redisTemplate" ref="redisTemplate"/>  
</bean>  
  
//省略其他的spring-redis.xml配置，具体参见源代码
```

说明

无论是使用XML配置Spring IOC Bean，还是通过Spring Boot的properties、yaml配置Spring IOC Bean，其原理都是类似的，仅仅是表达方式的不同。从学习的角度来说，最好是都有所了解。

spring-data-redis库在JedisPool连接池的基础上配置自己的连接池——RedisConnection Factory连接工厂，并且封装了一个短期、非线程安全的连接类，名为RedisConnection。RedisConnection类和

Jedis库中的Jedis类原理一样，提供了与Redis客户端命令一对一的API函数，用于操作远程Redis缓存数据。

RedisConnection的API命令操作的对象都是字节级别的键和值。为了更进一步地减少开发的工作，spring-data-redis库在RedisConnection连接类的基础上针对不同的缓存类型设计了五大数据类型的命令API集合，用于完成不同类型的数据缓存操作，并封装在RedisTemplate模板类中。

14.5.3 RedisTemplate模板API

RedisTemplate模板类位于核心包org.springframework.data.redis.core中，封装了五大数据类型的API集合：

- (1) ValueOperations：字符串类型操作API集合。
- (2) ListOperations：列表类型操作API集合。
- (3) SetOperations：集合类型操作API集合。
- (4) ZSetOperations：有序集合类型API集合。
- (5) HashOperations：哈希类型操作API集合。

每一种类型的操作API基本上都和每一种类型的Redis客户端命令一一对应。但是在名称上，API与Redis命令并不完全一致，RedisTemplate的API名称更加人性化。例如，Redis客户端命令setNX

表示在键-值对不存在时才会设置，不够直观；RedisTemplate的API名称为setIfAbsent，非常直观。显然，setIfAbsent比setNX易懂多了。

除了名称存在略微的调整，总体上而言，RedisTemplate模板类中的API函数和Redis客户端命令是一一对应的关系。所以，本小节不再一一赘述RedisTemplate模板类中的API函数，大家可以自行阅读API的源代码。

在实际开发中，为了尽可能减少对特定的第三方库的依赖，减少第三方库的“入侵”性，或者为了在不同的第三方库之间进行方便的切换，一般要对第三方库进行封装。

下面将RedisTemplate模板类的大部分缓存操作封装成一个自己的缓存操作Service服务——CacheOperationService，部分源代码节选如下：

```
package com.crazymakercircle.redis.springJedis;  
//...  
public class CacheOperationService {  
  
    private RedisTemplateredisTemplate;  
    public void setRedisTemplate(RedisTemplateredisTemplate) {  
        this.redisTemplate = redisTemplate;  
    }  
  
    //-----RedisTemplate基本操作 -----  
    /**  
     * 取得指定格式的所有键
```

```
* @param patens 匹配的表达式
* @return key 的集合
*/
public Set<Object> getKeys(Object patens) {
    try {
        return redisTemplate.keys(patens);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

/**
 * 指定缓存失效的时间
 *
 * @param key 键
 * @param time 时间(秒)
 * @return
 */
public Boolean expire(String key, long time) {
    try {
        if (time > 0) {
            redisTemplate.expire(key, time,
TimeUnit.SECONDS);
        }
        return true;
    } catch (Exception e) {
```

```
        e.printStackTrace();

        return false;
    }

}

/***
 * 判断key是否存在
 * @param key 键
 * @return true则存在, false则不存在
 */

public Boolean hasKey(String key) {

    try {

        return redisTemplate.hasKey(key);

    } catch (Exception e) {

        e.printStackTrace();

        return false;
    }
}

/***
 * 删除缓存
 * @param key 可以传一个值或多个
 * @return 删除的个数
 */

public void del(String... key) {

    if (key != null && key.length> 0) {

        if (key.length == 1) {

            redisTemplate.delete(key[0]);
        }
    }
}
```

```
        } else {

redisTemplate.delete(CollectionUtils.arrayToList(key));

    }

}

//-----RedisTemplate操作 String字符串 -----
-----

/***
 * 获取String
 * @param key 键
 * @return 值
 */

public Object get(String key) {
    return key == null ? null :
redisTemplate.opsForValue().get(key);
}

//省略String、Hash、Set、ZSet类型的封装操作，请参见随书源代码
}
```

完整的源代码比较长，可以在源代码工程中查阅。在代码中，除了基本数据类型的Redis操作（如keys、hasKey）直接使用redisTemplate实例完成。其他的API命令都是在不同类型的命令集合类上完成。

redisTemplate提供了5种方法，取得不同类型的命令集合，具体为：

(1) redisTemplate.opsForValue()取得String类型命令API集合。

(2) redisTemplate.opsForList()取得List类型命令API集合。

(3) redisTemplate.opsForSet()取得Set类型命令API集合。

(4) redisTemplate.opsForHash()取得Hash类型命令API集合。

(5) redisTemplate.opsForZSet()取得ZSet类型命令API集合。

然后，在不同类型的命令API集合上使用各种数据类型特有的API函数，完成具体的Redis API操作。

14.5.4 使用RedisTemplate模板API完成CRUD的实战案例

封装完了自己的CacheOperationService缓存管理服务之后，注入Spring的业务Service中，就可以完成缓存的CRUD操作了。

这里的业务类是UserServiceImplWithTemplate类，用于完成User实例缓存的CRUD。使用CacheOperationService后，就能非常方便地进行缓存的管理，同时在进行POJO的查询时能优先使用缓存数据，省去了数据库访问的时间。

```
package com.crazymakercircle.redis.springJedis;
import com.crazymakercircle.im.common.bean.User;
import com.crazymakercircle.util.Logger;
```

```
public class UserServiceImplWithTemplate implements UserService
{
    public static final String USER_UID_PREFIX = "user:uid:";

    protected CacheOperationService cacheOperationService; //需
提前赋值

    private static final long CASHE_LONG = 60 * 4;//4分钟

    ...
    /**
     * CRUD 的创建/更新
     * @param user 用户
     */
    @Override
    public User saveUser(final User user) {
        //保存到缓存
        String key = USER_UID_PREFIX + user.getId();
        Logger.info("user : ", user);
        cacheOperationService.set(key, user, CASHE_LONG);
        //保存到数据库
        //如MySQL
        return user;
    }

    /**
     * CRUD 的查询
     * @param id id
     * @return 用户
     */
}
```

```
@Override  
public User getUser(final long id) {  
    //首先从缓存中获取  
    String key = USER_UID_PREFIX + id;  
    User value = (User) cacheOperationService.get(key);  
    if (null == value) {  
        //如果缓存中没有，就从数据库中查询  
        //如MySQL  
        //然后保存到缓存，供下一次查询使用  
    }  
    return value;  
}  
  
/**  
 * CRUD 的删除  
 * @param id id  
 */  
@Override  
public void deleteUser(long id) {  
    //从缓存删除  
    String key = USER_UID_PREFIX + id;  
    cacheOperationService.del(key);  
    //从数据库删除  
    //如MySQL  
    Logger.info("delete User:", id);  
}  
}
```

在业务Service类中使用CacheOperationService缓存管理之前，还需要在配置文件（这里为spring-redis.xml）中配置好依赖：

```
<!--将redisTemplate封装成缓存service-->
<bean id="cacheOperationService"
      class="com.crazymakercircle.redis.springJedis.CacheOperationService">
    <property name="redisTemplate" ref="redisTemplate"/>
</bean>

<!--业务service, 依赖缓存service-->
<bean id="serviceImplWithTemplate"
      class="com.crazymakercircle.redis.springJedis
              .ServiceImplWithTemplate">
    <property name="cacheOperationService"
              ref="cacheOperationService"/>
</bean>
```

编写一个用例，测试一下ServiceImplWithTemplate，运行之后，可以从Redis客户端输入命令来查看缓存的数据。至此，缓存机制成功生效，数据访问的时间可以从查询数据库时的百毫秒级别缩小到毫秒级别，性能提升了100倍。

```
package com.crazymakercircle.redis.springJedis;
//省略import
public class SpringRedisTester {
    @Test
    public void testServiceImplWithTemplate() {
        ApplicationContext ac = new
```

```

ClassPathXmlApplicationContext
        ("classpath:spring-
redis.xml");

        UserServiceuserService =
                (UserService)
ac.getBean("serviceImplWithTemplate");

        long userId = 1L;

        userService.deleteUser(userId);

        User userInredis = userService.getUser(userId);

        Logger.info("delete user", userInredis);

        User user = new User();

        user.setUid("1");

        user.setNickName("foo");

        userService.saveUser(user);

        Logger.info("save user:", user);

        userInredis = userService.getUser(userId);

        Logger.info("get user", userInredis);

    }

//省略其他测试用例
}

```

14.5.5 使用RedisCallback回调完成CRUD的实战案例

前面讲到，RedisConnection连接类和RedisTemplate模板类都提供了整套Redis操作的API，只不过它们的层次不同。RedisConnection连接类更加底层，负责二进制层面的Redis操作，Key、Value都是二进

制字节数组。RedisTemplate模板类在RedisConnection的基础上使用在spring-redis.xml中配置的序列化、反序列化的工具类完成上层类型（如String、Object、POJO类等）的Redis操作。

如果不需要RedisTemplate配置的序列化、反序列化的工具类，或者由于其他的原因需要直接使用RedisConnection去操作Redis，可以调用RedisCallback的doInRedis()回调方法，在doInRedis()回调方法中直接使用实参RedisConnection连接类实例来完成操作。

当然，完成RedisCallback回调业务逻辑后，还需要使用RedisTemplate模板实例去执行，调用的是RedisTemplate.execute(RedisCallback)方法。

通过RedisCallback回调方法实现CRUD的实例代码如下：

```
package com.crazymakercircle.redis.springJedis;  
//省略import  
  
public class UserServiceImplInTemplate implements UserService {  
    public static final String USER_UID_PREFIX = "user:uid:";  
    private RedisTemplateredisTemplate;  
    public void setRedisTemplate(RedisTemplateredisTemplate) {  
        this.redisTemplate = redisTemplate;  
    }  
    private static final long CASHE_LONG = 60 * 4;//4分钟  
    /**  
     * CRUD 的创建/更新  
     * @param user 用户  
     */
```

```
@Override
public User saveUser(final User user) {
    //保存到缓存
    redisTemplate.execute(new RedisCallback<User>() {
        @Override
        public User doInRedis(RedisConnection connection)
                throws DataAccessException {
            byte[] key =
                serializeKey(USER_UID_PREFIX +
user.getId());
            connection.set(key, serializeValue(user));
            connection.expire(key, CACHE_LONG);
            return user;
        }
    });
    //保存到数据库
    //如MySQL
    return user;
}

private byte[] serializeValue(User s) {
    return redisTemplate.getValueSerializer().serialize(s);
}
private byte[] serializeKey(String s) {
    return redisTemplate.getKeySerializer().serialize(s);
}
private User deSerializeValue(byte[] b) {
```

```
        return (User)
redisTemplate.getValueSerializer().deserialize(b);
    }

/**
 * CRUD 的查询
 * @param id id
 * @return 用户
 */
@Override
public User getUser(final long id) {
    //首先从缓存中获取
    User value = (User) redisTemplate.execute(
        new RedisCallback<User>()
    {
        @Override
        public User doInRedis(RedisConnection connection)
            throws DataAccessException {
            byte[] key = serializeKey(USER_UID_PREFIX +
id);
            if (connection.exists(key)) {
                byte[] value = connection.get(key);
                return deSerializeValue(value);
            }
            return null;
        }
    });
}
```

```
    if (null == value) {
        //如果缓存中没有，就从数据库中获取
        //如MySQL
        //并且保存到缓存
    }
    return value;
}

/**
 * CRUD 的删除
 * @param id id
 */
@Override
public void deleteUser(long id) {
    //从缓存删除
    redisTemplate.execute(new RedisCallback<Boolean>() {
        @Override
        public Boolean doInRedis(RedisConnection
connection)
            throws DataAccessException {
            byte[] key = serializeKey(USER_UID_PREFIX +
id);
            if (connection.exists(key)) {
                connection.del(key);
            }
            return true;
        }
    });
}
```

```
//从数据库删除  
//如MySQL  
}  
}
```

同样，在使用UserServiceImplInTemplate之前，也需要在配置文件（这里为spring-redis.xml）中配置好依赖关系：

```
<bean id="serviceImplInTemplate"  
      class="com.crazymakercircle.redis.springJedis  
            .UserServiceImplInTemplate">  
    <property name="redisTemplate" ref="redisTemplate"/>  
  </bean>
```

14.6 Spring的Redis缓存注解

前面讲的Redis缓存实现都是基于Java代码实现的。在Spring中，合理地添加缓存注解也能实现和前面示例程序中一样的缓存功能。

为了方便地提供缓存能力，Spring提供了一组缓存注解。这组注解不仅仅是针对Redis，本质上不是一种具体的缓存实现方案（例如Redis、EHCache等），而是对缓存使用的统一抽象。利用这组缓存注解，以及与具体缓存相匹配的Spring配置，不用编码就可以快速达到缓存的效果。

下面先展示一下Spring缓存注解的应用实例，然后对Spring Cache的几个注解进行详细的介绍。

14.6.1 使用Spring缓存注解完成CRUD的实战案例

这里简单介绍一下Spring的三个缓存注解：@CachePut、@CacheEvict、@Cacheable。这三个注解通常都加在方法的前面，作用如下：

(1) @CachePut作用是设置缓存。先执行方法，再将执行结果缓存起来。

(2) @CacheEvict的作用是删除缓存。在执行方法前，删除缓存。

(3) @Cacheable的作用更多是查询缓存。首先检查注解中的Key是否在缓存中，如果是，则返回Key的缓存值，不再执行方法；否则，执行方法并将方法结果缓存起来。从后半部分来看，@Cacheable也具备@CachePut的能力。

在展开介绍三个注解之前，先演示一下它们的使用：用它们实现一个带缓存功能的用户操作UserService实现类，名为 UserServiceImplWithAnno类。其功能和前面介绍的 UserServiceImplWithTemplate类是一样的，只是这里使用注解去实现缓存，代码如下：

```
package com.crazymakercircle.redis.springJedis;  
//省略import  
  
@Service  
@CacheConfig(cacheNames = "userCache")  
public class UserServiceImplWithAnno implements UserService {  
  
    public static final String USER_UID_PREFIX =  
        "'userCache:'+";  
  
    /**  
     * CRUD 的创建/更新  
     * @param user 用户  
     */  
    @CachePut(key = USER_UID_PREFIX +  
        "T(String).valueOf(#user.uid)")  
    @Override  
    public User saveUser(final User user) {
```

```
//保存到数据库
//返回值将保存到缓存
Logger.info("user : save to redis");
return user;
}

/**
 * CRUD 的查询
 * @param id id
 * @return 用户
 */
@Cacheable(key = USER_UID_PREFIX +
"T(String).valueOf(#id)")
@Override
public User getUser(final long id) {
    //如果缓存没有，则从数据库中加载
    Logger.info("user : is null");
    return null;
}

/**
 * CRUD 的删除
 * @param id id
 */
@CacheEvict(key = USER_UID_PREFIX +
"T(String).valueOf(#id)")
@Override
```

```
public void deleteUser(long id) {  
    //从数据库中删除  
    Logger.info("delete User:", id);  
}  
}
```

在UserServiceImplWithAnno类中没有出现任何缓存操作API，但是它的缓存功能和前面的ServiceImplWithTemplate类使用RedisTemplate模板实现的缓存功能是一模一样的。可见，使用缓存注解@CachePut、@CacheEvict和@Cacheable能较大地减少代码量。

总之，通过这个实例可以发现，使用注解实现缓存和使用API实现缓存的功能相比，前者的代码简化得多。另外，使用注解实现缓存功能还能方便地在不同的缓存服务之间实现切换。

14.6.2 spring-redis.xml中配置的调整

在使用Spring缓存注解前，首先需要配置文件中启用Spring对基于注解的Cache的支持：在spring-redis.xml中，加上<cache:annotation-driven />配置项。

<cache:annotation-driven/>有一个cache-manager属性，用来指定所需要用到的缓存管理器（CacheManager）的Spring Bean的名称。如果不进行特别设置，默认的名称是CacheManager。也就是说，如果使用了<cache:annotation-driven />，还需要配置一个名为CacheManager的缓存管理器Spring Bean，这个Bean要求实现CacheManager接口。CacheManager接口是Spring定义的一个用来管理

Cache缓存的通用接口。对应于不同的缓存，需要使用不同的CacheManager实现。Spring自身已经提供了一种CacheManager的实现，是基于Java API的ConcurrentMap简单的内存Key-Value缓存实现。这里需要使用的缓存是Redis，所以使用spring-data-redis包中的RedisCacheManager实现。

spring-redis.xml增加的配置项具体如下：

```
<!--启用缓存注解功能，这个是必需的，否则注解不会生效 -->
<cache:annotation-driven/>

<!--自定义redis工具类，在需要缓存的地方注入此类 -->
<bean id="cacheManager" mode="proxy"

class="org.springframework.data.redis.cache.RedisCacheManager">
<constructor-arg ref="redisTemplate"/>
<constructor-arg name="cacheNames">
<set>
    <!--声明userCache-->
    <value>userCache</value>
</set>
</constructor-arg>
</bean>
```

<cache:annotation-driven/>还可以指定一个mode属性，可选值有proxy和aspectj，默认使用proxy。proxy和aspectj两个值的作用非常类似于使用<tx:annotation-driven/>来配置事务时mode属性的proxy和aspectj两个值的作用。

当mode为proxy时，只有当有被注解的方法被对象外部的方法调用时，Spring Cache注解才会发生作用；反过来说，一个缓存方法被其所在对象的内部方法调用时，Spring Cache是不会发生作用的。另外，当mode为proxy模式时，只有加在public类型方法上的Spring Cache注解会发生作用。

mode为aspectj模式时，缓存注解的作用与代理模式下的不同。Spring Cache注解可以在方法被自身内部调用时生效，也可以在非public方法上生效。

<cache:annotation-driven/>还可以指定一个proxy-target-class属性，设置代理类的创建机制，有两个值：

- (1) 值为true，表示使用CGLib创建代理类。
- (2) 值为false，表示使用JDK的动态代理机制创建代理类，默认为false。

JDK的动态代理的原理是生成一个实现代理接口的匿名类（Class-Based Proxies），在调用具体方法前通过调用InvokeHandler来调用实际的代理方法。

说明

有关该动态代理的原理性知识，具体请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

使用CGLib创建代理类则不同。CGLib底层采用ASM开源“.class”字节码生成框架，生成字节码级别的代理类（Interface-based Proxies）。对比来说，在实际的运行时，CGLib代理类比使用Java反射代理类的效率要高。

当proxy-target-class为true时，表示使用CGLib创建代理类，此时@Cacheable和@CacheInvalidate等注解必须标记在具体类（Concrete Class）上，不能标记在接口上，否则不会发生作用。当proxy-target-class为false时，@Cacheable和@CacheInvalidate等可以标记在接口上，也能发挥作用。

在配置RedisCacheManager缓存管理器Bean时，需要配置两个构造参数：

- **redisTemplate**: 模板Bean。
- **cacheNames**: 缓存名称。

不同的spring-data-redis版本，构造函数不同。这里使用的spring-data-redis的版本是1.4.3。对于2.0版本，在配置上发生了一些变化，但是原理大致相同。

14.6.3 @CachePut和@Cacheable注解

简单来说，@CachePut和@Cacheable注解都可以增加缓存，但是有细微的区别：@CachePut负责增加缓存；@Cacheable负责查询缓存，没有查到时，才去执行被注解的方法，并将方法的结果增加到缓存。

1. @CachePut注解

在支持Spring Cache的环境下，如果@CachePut加在方法上，则每次执行方法后会将结果存入指定缓存的键上，如下所示：

```
/**
 * CRUD 的创建/更新
 * @param user 用户
 */
@CachePut(key = USER_UID_PREFIX +
"T(String).valueOf(#user.uid)")

@Override
public User saveUser(final User user) {
    //保存到数据库
    //返回值将保存到缓存
    Logger.info("user : save to redis");
    return user;
}
```

Redis的缓存都是“键-值对”形式。Redis缓存中的键即为@CachePut注解配置的key属性值，一般是一个字符串，或者是结果为字符串的一个SpEL（SpringEL）表达式。Redis缓存的值就是方法的返回结果经过序列化后所产生的序列化数据。

一般来说，可以给@CachePut设置三个属性：value、key和condition。

(1) value属性指定Cache的名称。

value属性的值表示当前键被缓存在哪个Cache上，对于Spring配置文件中CacheManager缓存管理器cacheNames属性中配置的某个Cache名称，如userCache。可以配置一个Cache，也可以配置多个Cache，当配置多个Cache时，value是一个数组，如value={userCache, otherCache1, otherCache2, …}。

说明

value属性值中的Cache名称相当于缓存键所属的命名空间。当使用@CacheEvict注解清除缓存时，可以通过合理配置清除指定Cache名称下的所有键。

(2) key属性指定Redis的键属性值。

key属性用来指定Spring缓存方法的键，该属性支持SpEL表达式。当没有指定该属性时，Spring将使用默认策略生成键。有关SpEL表达式的内容，后面会有详细介绍。

(3) condition属性指定缓存的条件。

并不是所有的函数结果都希望加入Redis缓存，可以通过condition属性来实现这一功能。condition属性值默认为空，表示将缓存所有结果。可以通过SpEL表达式来设置：当表达式的值为true时，表示进行缓存处理；否则不进行缓存处理。如下示例程序表示只有当user的id大于1000时才会进行缓存：

```
@CachePut(key = "T(String).valueOf(#user.uid)",  
          condition =  
          "#user.uid>1000")  
  
public User cacheUserWithCondition(final User user) {  
    //保存到数据库  
    //返回值将保存到缓存  
    Logger.info("user : save to redis");  
    return user;  
}
```

2. @Cacheable注解

Cacheable注解主要是查询缓存。对于加上了@Cacheable注解的方法，Spring在每次执行前都会检查Redis缓存中是否存在相同键，如果存在，就不再执行该方法，而是直接从缓存中获取结果并返回。如果不存在，就会执行方法，并将返回结果存入Redis缓存中。与@CachePut注解一样，@Cacheable也具备增加缓存的能力。

@Cacheable与@CachePut的不同之处是：@Cacheable只有当键在Redis缓存不存在的时候才执行方法，将方法的结果缓存起来；如果键在Redis缓存中存在，则直接返回缓存结果。加了@CachePut注解的方法则缺少了检查的环节：@CachePut在方法执行前不进行缓存检查，无论之前是否有缓存都会将新的执行结果加入缓存中。

使用@Cacheable注解，一般也能指定三个属性：value、key和condition。三个属性的配置方法和@CachePut三个属性的配置方法是一样的，这里不再赘述。

@CachePut和@Cacheable注解也可以标注在类上，表示所有的方法都具有缓存处理的功能。但是这种情况用得比较少。

14.6.4 @CacheEvict注解

@CacheEvict注解主要用来清除缓存，可以指定的属性有value、key、condition、allEntries和beforeInvocation。其中，value、key和condition的语义与@Cacheable对应的属性类似。value表示清除哪些Cache（对应Cache的空间名称）；key表示清除哪个键；condition表示清除的条件。下面主要看一下allEntries和beforeInvocation两个属性。

(1) allEntries属性：表示是否全部清空。

allEntries表示是否需要清除缓存中的所有键，是boolean类型；默认为false，表示不需要清除全部。当指定了allEntries为true时，表示清空value属性所指向的Cache中所有的缓存，这时所配置的key属性值已经没有意义，将被忽略。allEntries为true用于需要全部清空某个Cache的场景，比一个一个清除键的效率更高。

在下面的例子中，一次清除Cache名称为userCache中的所有Redis缓存，代码如下：

```
package com.crazymakercircle.redis.springJedis;  
//省略import  
@Service  
@CacheConfig(cacheNames = "userCache")  
public class UserServiceImplWithAnno implements UserService {
```

```
//省略其他的  
/**  
 * 删除userCache名字空间的全部缓存  
 */  
  
@CacheEvict(value = "userCache", allEntries = true)  
public void deleteAll() {  
}  
}
```

(2) beforeInvocation属性：表示是否在方法执行前操作缓存。

一般情况下，在对应方法成功执行之后再触发清除操作。如果方法执行过程中有异常抛出，或者由于其他的原因导致线程终止，就不会触发清除操作。所以，通过设置beforeInvocation属性来确保清理。

beforeInvocation属性是boolean类型，当设置为true时，可以改变触发清除操作的次序，Spring会在执行注解的方法之前完成缓存的清除工作。

另外，注解@CacheEvict除了加在方法上，还可以加在类上。当加在一个类上时，表示该类所有的方法都会触发缓存清除。一般情况下，很少这样使用。

14.6.5 @Caching组合注解

@Caching注解是一个缓存处理的组合注解。@Caching可以一次指定多个Spring Cache注解的组合。@Caching的组合能力主要通过三个属性完成，具体如下：

(1) cacheable属性：用于指定一个或者多个@Cacheable注解的组合，可以指定一个，也可以指定多个。如果指定多个@Cacheable注解，则直接使用数组的形式，即使用花括号将多个@Cacheable注解包围起来，用于查询一个或多个键的缓存，如果没有，则按照条件将结果加入缓存。

(2) put属性：用于指定一个或者多个@CachePut注解的组合，可以指定一个，也可以指定多个，用于设置一个或多个键的缓存。如果指定多个@CachePut注解，则直接使用数组的形式。

(3) evict属性：用于指定一个或者多个@CacheEvict注解的组合，可以指定一个，也可以指定多个，用于删除一个或多个键的缓存。如果指定多个@CacheEvict注解，则直接使用数组的形式。

在数据库中，往往需要进行外键的级联删除：在删除一个主键时，需要将一个主键的所有级联的外键通通都删掉。如果外键都需要进行缓存，在级联删除时则可以使用@Caching注解组合多个@CacheEvict注解，在删除主键缓存时删除所有的外键缓存。下面有一个简单的实例，模拟在更新一个用户时需要删除与用户关联的多个缓存，包括用户信息、地址信息、用户的商品等，具体如下：

```
/**  
 * 在一种方法上一次性加上三大类cache处理  
 */
```

```
@Caching(cacheable = @Cacheable(key = "'userCache:'+"  
#uid"),  
put = @CachePut(key = "'userCache:'+"  
#uid),  
evict = {  
    @CacheEvict(key = "'userCache:'+" #uid),  
    @CacheEvict(key = "'addressCache:'+" #uid),  
    @CacheEvict(key = "'messageCache:'+" #uid)  
}  
)  
  
public User updateRef(String uid) {  
    //...业务逻辑  
    return null;  
}
```

以上示例程序仅仅是一个组合注解的演示。`@Caching`有`cacheable`、`put`、`evict`三大类型属性，在实际使用时，可以进行类型的灵活裁剪。例如，实际的开发场景并不需要添加缓存，完全可以不给`@Caching`注解配置`cacheable`属性。

至此，缓存注解已经介绍完毕。在缓存注解中用到的SpEL表达式将在下一节专门介绍。

14.7 详解SpEL

SpEL（Spring Expression Language，Spring表达式语言）。提供一种强大、简洁的Spring Bean的动态操作表达式。SpEL表达式可以在运行期间执行，其值可以动态装配到Spring Bean属性或构造函数中。SpEL表达式可以调用Java静态方法、访问Properties文件中的配置值等。SpEL能够与Spring功能完美整合，给静态Java语言增加了动态功能。

JSP页面的表达式使用\${}声明，而SpEL表达式使用#{}声明。SpEL支持如下表达式：

(1) 基本表达式：字面量表达式，关系、逻辑与算术运算表达式，字符串连接及截取表达式，三目运算，正则表达式，括号优先级表达式。

(2) 类型表达式：类型访问、静态方法/属性访问、实例访问、实例属性值存取、实例属性导航、instanceof、变量定义及引用、赋值表达式、自定义函数等。

(3) 集合相关表达式：列表、内联数组、集合、字典、数组、集合投影；不支持多维内联数组初始化；不支持内联字典定义。

(4) 其他表达式：模板表达式。

14.7.1 SpEL运算符

SpEL基本表达式是由各种基础运算符、常量、变量引用一起进行组合所构成的表达式。基础运算符主要包括算术运算符、关系运算符、逻辑运算符、字符串运算符、三目运算符、正则表达式匹配符、类型访问运算符、变量引用符等。

(1) 算术运算符：加（+）、减（-）、乘（*）、除（/）、求余（%）、幂（^）、求余（MOD）和除（DIV）等算术运算符。MOD与“%”等价，DIV与“/”等价，并且不区分字母大小写。例如：`#{1+2*3/4-2}`、`#{2^3}`、`#{100 mod 9}`都是算术运算SpEL表达式。

(2) 关系运算符：等于（==）、不等于（!=）、大于（>）、大于等于（>=）、小于（<）、小于等于（<=）、区间（between）运算等。例如：`#{2>3}`的值为false。

(3) 逻辑运算符：与（and）、或（or）、非（!或NOT）。例如：`#{2>3 or 4>3}`的值为true。与Java逻辑运算不同，SpEL不支持“&&”和“||”。

(4) 字符串运算符：SpEL提供了连接（+）和截取（[]）字符串运算符。例如：`#{'Hello' + 'World!'}`的结果为“Hello World!”；`#{'Hello World!' [0]}` 截取第一个字符“H”。不过，目前只支持获取一个字符。

(5) 三目运算符：SpEL提供了和Java一样的三目运算符，用法是“逻辑表达式？表达式1：表达式2”。例如：`#{3>4? 'Hello' : 'World'}` 将返回'World'。

(6) 正则表达式匹配符：SpEL提供了字符串的正则表达式匹配符matches。例如：`#{'123' matches '\\d{3}'}` 返回true。

(7) 类型访问运算符：SpEL提供了一个类型访问运算符 T(Type)。其中，“Type”表示某个Java类型，实际上对应于Java类的 java.lang.Class 实例。Type必须是类的全限定名（包括包名），但是核心包“java.lang”中的类除外。也就是说，“java.lang”包下的类可以不用指定完整的包名。例如：T(String)表示访问的是 java.lang.String类，#{T(String).valueOf(1)}表示将整数1转换成字符串。

(8) 变量引用符：SpEL提供了一个上下文变量的引用符“#”，可在表达式中使用“#variableName”引用上下文变量。

SpEL提供了一个变量定义的上下文接口——EvaluationContext，并且提供了标准的上下文实现——StandardEvaluationContext。通过上下文的setVariable(variableName, value)方法可以定义“上下文变量”，这些变量在表达式中以“#variableName”的方式引用。在创建变量上下文Context实例时，还可以在构造器参数中设置一个rootObject作为根，既可以使用“#root”引用根对象，也可以使用“#this”引用根对象。

下面使用前面介绍的运算符定义几个SpEL表达式，示例程序如下：

```
package com.crazymakercircle.redis.springJedis;  
//省略import  
  
public class SpElBean {  
  
    /**  
     * 算术运算符  
     */
```

```
@Value("#{10+2*3/4-2}")
private int algDemoValue;

/**
 * 字符串运算符
 */
@Value("#{Hello ' + 'World!'}")
private String stringConcatValue;

/**
 * 类型运算符
 */
@Value("#{ T(java.lang.Math).random() * 100.0 }")
private int randomInt;

/**
 * 展示SpEL上下文变量
 */
public void showContextVar() {
    ExpressionParser parser = new SpelExpressionParser();
    EvaluationContext context = new
StandardEvaluationContext();
    context.setVariable("foo", "bar");
    String foo = parser.parseExpression("#{foo}")

    .getValue(context, String.class);
    Logger.info(" foo:=", foo);
}
```

```

        context = new StandardEvaluationContext("I am root");

        String root = parser.parseExpression("#root")
                .getValue(context,
String.class);

        Logger.info(" root:=", root);

        String result3 =parser.parseExpression("#this") .
                getValue(context,
String.class);

        Logger.info(" this:=", result3 );

    }

}

```

以上示例程序代码的测试用例如下：

```

package com.crazymakercircle.redis.springJedis;
//...
/**
 * create by 尼恩 @ 疯狂创客圈
 */
public class SpringRedisTester {
    /**
     * 测试SpEL表达式
     */
    @Test
    public void testSpElBean() {

```

```
ApplicationContext ac =
    new ClassPathXmlApplicationContext("classpath:spring-
redis.xml");
    SpElBeanspElBean = (SpElBean) ac.getBean("spElBean");
    /**
     * 演示算术运算符
     */
    Logger.info(" spElBean.getAlgDemoValue():=",
        spElBean.getAlgDemoValue());
    /**
     * 演示字符串运算符
     */
    Logger.info("spElBean.getStringConcatValue():=",
        spElBean.getStringConcatValue());
    /**
     * 演示类型运算符
     */
    Logger.info(" spElBean.getRandomInt():=" ,
        spElBean.getRandomInt());
    /**
     * 展示SpEL上下文变量
     */
    spElBean.showContextVar();
}
}
```

说明

一般来说，SpEL表达式使用`#{}声明`。但是，不是所有注解中的SpEL表达式都需要使用`#{}进行声明`。例如，`@Value`注解中的SpEL表达式需要`#{}进行声明`；而`ExpressionParser.parseExpression`实例方法中的SpEL表达式不需要使用`#{}进行声明`。另外，`@CachePut`和`@Cacheable`等缓存注解中`key`属性值的SpEL表达式也不需要使用`#{}进行声明`。

运行上面的测试用例，输出的结果大致如下：

```
[...testSpElBean] |> spElBean.getAlgDemoValue():= 9
[...testSpElBean] |> spElBean.getStringConcatValue():= Hello
World!
[...testSpElBean] |> spElBean.getRandomInt():= 27
[..SpElBean.showContextVar] |> foo:= bar
[..SpElBean.showContextVar] |> root:= I am root
[..SpElBean.showContextVar] |> this:= I am root
```

14.7.2 缓存注解中的SpEL表达式

对应于加在方法上的缓存注解（如`@CachePut`和`@Cacheable`），Spring提供了专门的上下文类`CacheEvaluationContext`，继承于`MethodBasedEvaluationContext`（基础的方法注解上下文）类，而

MethodBasedEvaluationContext类又继承于
StandardEvaluationContext（标准注解上下文类）。

CacheEvaluationContext的构造器如下：

```
class CacheEvaluationContext extends  
MethodBasedEvaluationContext {  
    //构造器  
    CacheEvaluationContext(Object rootObject,           //根对象  
                          Methodmethod,             //当前方法  
                          Object[] arguments,      //当前方法的  
                          参数  
                          ParameterNameDiscovererparameterNameDiscoverer)  
    {  
        super(rootObject, method, arguments,  
              parameterNameDiscoverer);  
    }  
    //省略其他Spring源代码  
}
```

在配置缓存注解（如@CachePut）的Key时，可以用到
CacheEvaluationContext的rootObject（根对象）。通过该根对象，
可以获取如表14-2所示的属性。

表14-2 能够通过CacheEvaluationContext的rootObject获取的属性

属性名称	说 明	示 例
methodName	当前被调用的方法名	#root.methodName
Method	当前被调用的方法	#root.method.name
Target	当前被调用的目标对象	#root.target
targetClass	当前被调用的目标对象类	#root.targetClass
Args	当前被调用的方法的参数列表	当前被调用的方法的第 0 个参数: #root.args[0]
Caches	当前被调用方法使用的缓存列表, 比如 @Cacheable(value={"cache1","cache2"})) 就有两个 Cache	当前被调用方法的第 0 个 Cache 名称: #root.caches[0].name

在配置键属性时, 用到SpEL表达式root对象的属性时, 可以将“#root”省略, 因为Spring默认使用的就是root对象的属性。例如:

```
@Cacheable(value={"cache1", "cache2"}, key="caches[1].name")
public User find(User user) {
    //省略查询数据库的代码
}
```

在SpEL表达式中, 除了访问SpEL表达式root对象, 还可以访问当前方法的参数以及它们的属性。访问方法的参数有以下两种形式:

(1) 使用 “#p参数index” 形式访问方法的参数, 其中p为parameter参数的英文首字母。下面展示如何使用”#p参数index”形式访问第0个参数:

```
//访问第0个参数, 参数id
@Cacheable(value="users", key="#p0")
public User find(String id) {
    //省略查询数据库的代码
}
```

在下面的示例程序中访问参数的属性（比如user的id属性），具体如下：

```
//访问参数user的id属性  
@Cacheable(value="users", key="#p0.id")  
public User find(User user) {  
    //省略查询数据库的代码  
}
```

(2) 使用"#参数名"形式访问方法的参数。例如，使用"#user.id"的形式访问参数user的id属性，代码如下：

```
//使用"#参数名"的形式访问user参数的属性值，这里是id  
@Cacheable(value="users", key="#user.id")  
public User find(User user) {  
    //省略查询数据库的代码  
}
```

通过对比可以看出，在访问方法的参数以及参数的属性时，使用"#参数名"形式比"#p参数index"的形式更加直观。

第15章 亿级高并发IM架构与实战

本章结合分布式缓存Redis、分布式协调ZooKeeper、高性能通信Netty，从架构的维度设计一套亿级IM的高并发架构方案，并从学习和实战的角度出发，联合“疯狂创客圈”社群的高性能发烧友一起持续进行一个支持亿级流量的IM项目开发与迭代，该项目暂时被命名为CrazyIM。

15.1 支撑亿级流量的高并发IM架构的理论基础

支撑亿级流量的高并发IM通信需要用到Netty集群、ZooKeeper集群、Redis集群、MySQL集群、Spring Cloud Web服务集群、RocketMQ消息队列集群等，具体如图15-1所示。

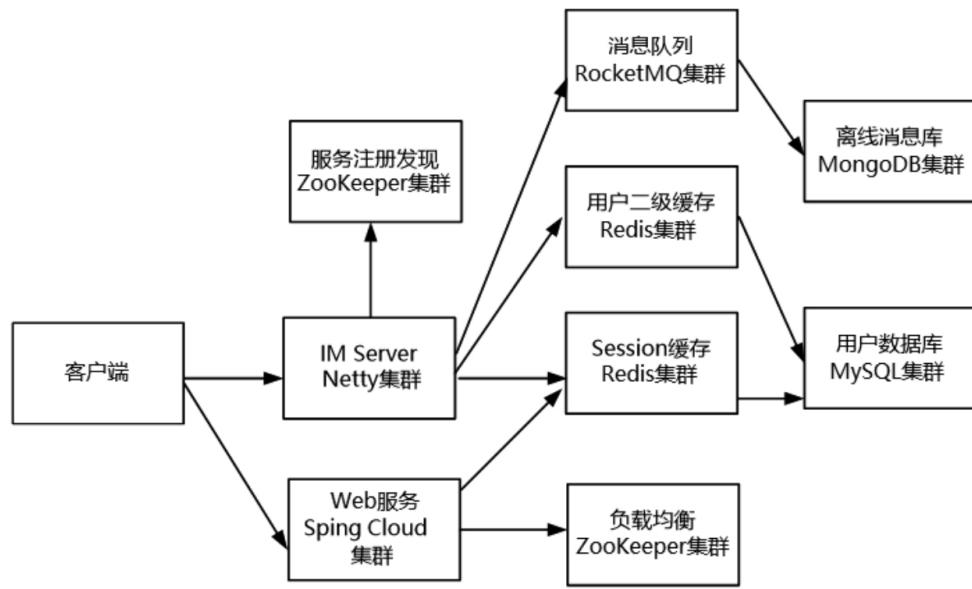


图15-1 支撑亿级流量的高并发IM架构

15.1.1 亿级流量的系统架构的开发实战

支撑亿级流量的高并发IM的几大集群中，最为核心的是Netty集群、ZooKeeper集群、Redis集群，它们是主要实现亿级流量通信功能不可缺少的集群。其次是Spring Cloud Web服务集群、MySQL集群，完成海量用户的登录和存储，用户关系、群组关系的维护。最后是RocketMQ消息队列集群、MongoDB半结构化存储集群，用于离线消息的保存。

(1) Netty服务集群：主要用来负责维持和客户端的TCP连接，完成消息的发送和转发。

(2) ZooKeeper集群：负责Netty Server集群的管理，包括注册、路由、负载均衡，集群IP注册和节点ID分配，主要在于为ZooKeeper集群提供底层服务。

(3) Redis集群：负责用户、用户绑定关系、用户群组关系、用户远程会话等数据的缓存，缓存其他的配置数据或者临时数据，加快读取速度。

(4) MySQL集群：保存用户、群组、离线消息等。

(5) RocketMQ消息队列集群：主要是将优先级不高的操作从高并发模式转成低并发模式。例如，将离线消息发向消息队列，然后通过低并发的异步任务保存到数据库。

说明

上面的架构是“疯狂创客圈”高性能社群的CrazyIM学习项目的架构，并且只涉及核心功能，并不是实践开发亿级流量系统架构的全部。从迭代的角度来看，还有很多完善的空间，“疯狂创客圈”高性能社群将持续对CrazyIM高性能项目的架构和实现进行更新和迭代，所以最终的架构图和实现以最后的版本为准。

从理论上来说，以上集群具备完全的扩展能力，进行合理的横向扩展和局部的优化后支撑亿级流量是没有任何问题的。单体的Netty服务器远不止支持10万个并发，在CPU、内存还不错的情况下，配置得当的话甚至能撑到100万级别的并发。所以，通过合理的高并发架构，能够让系统动态扩展到成百上千的Netty节点，支撑亿级流量是没有任何问题的。

至于如何通过配置让单体的Netty服务器支撑100万级别的高并发，请查询“疯狂创客圈”的社群文章《Netty 100万级高并发服务器配置》。

15.1.2 高并发架构的技术选型

明确了架构之后，接下来进行平台的技术选型。

(1) 核心组件：Netty4.x + Spring4.x + ZooKeeper 3.x + Redis 3.x + RocketMQ 3.x+ MySQL 5.x + Mongogo 3.x + Spring Cloud Finchley+ Nginx 15。

(2) 短连接服务：Spring Cloud。基于RESTful短连接的分布式微服务架构，完成用户在线管理、单点登录系统。

说明

Spring Cloud微服务集群往往与Nginx结合使用，具体请参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

(3) 长连接服务：Netty，主要用来负责维持和客户端的TCP连接，完成消息的发送和转发。

(4) 消息队列：RocketMQ高速消息队列。

(5) 数据库：MySQL + MongoDB。MySQL用来存储结构化数据，如用户数据；MongoDB用来存储半结构化的离线消息。

(6) 序列化协议：Protobuf + JSON。Protobuf是最高效的二进制序列化协议，用于长连接；JSON是最紧凑的文本协议，用于短连接。

15.1.3 详解IM消息的序列化协议选型

IM系统的客户端和服务器节点之间需要按照同一种数据序列化协议进行数据的交换，简而言之，就是规定网络中的字节流数据如何与应用程序需要的结构化数据相互转换。

序列化协议主要的工作有两部分，结构化数据到传输数据的序列化和反序列化；所涉及的协议类型为文本协议和二进制协议。

- 常见的文本协议包括XML和JSON。文本协议序列化之后，可读性好，便于调试，方便扩展。文本协议的缺点在于解析效率一般，有很多的冗余数据，这一点主要体现在XML格式上。
- 常见的二进制协议包括Protobuf、Thrift。这些协议都自带了数据压缩，编解码效率高，同时兼具扩展性。二进制协议的优势很明显，劣势也非常突出。二进制协议和文本协议相反，序

列化之后的二进制协议报文数据基本上没有什么可读性，不利于开发和调试。

因此，在协议的选择上给大家的建议是：

- 对于并发度不高的IM系统，使用文本协议，例如JSON。
- 对于并发度非常高，QPS在千万级、亿级的通信系统，尽量选择二进制协议。

在“疯狂创客圈”社群持续迭代的CrazyIM项目中，序列化协议选择的是Protobuf二进制协议，以便于容易达成对亿级流量的支撑。

15.1.4 详解长连接和短连接

什么是长连接呢？客户端向服务器发起连接，服务器接受客户端的连接，双方建立连接。客户端与服务器完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

TCP的连接过程是比较烦琐的，建立连接需要三次握手，而释放则需要四次挥手，所以说每个连接的建立都需要消耗资源和时间。

说明

TCP的原理以及其三次握手四次挥手的过程，具体请参考第10.2节。

在高并发的IM系统中，客户端和服务器之间需要大量发送通信的消息，如果每次发送消息都去建立连接，那么客户端和服务器的连接建立和断开的开销是非常巨大的。所以，IM消息的发送肯定需要长连接。

什么是短连接呢？客户端向服务器发起连接，服务器接受客户端连接，在三次握手之后双方建立连接。客户端与服务器完成一次读写，发送数据包并得到返回的结果之后，通过客户端和服务器的四次挥手断开连接。

短连接适用于数据请求频度较低的应用场景，例如网站的浏览和普通的Web请求。短连接的优点是管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

在高并发的IM系统中，客户端和服务器之间除了消息的通信外，还需要用户的登录与认证、好友的更新与获取等一些低频的请求，这些都使用短连接来实现。

在这个高并发IM系统中，存在两类后台服务：一类短连接后台服务；一类长连接后台服务。

短连接服务器也叫Web服务，主要功能是实现用户的登录鉴权和拉取好友、群组、数据档案等相对低频的请求操作。

长连接服务器也叫IM服务，主要作用是用来和客户端建立并维持长连接，实现消息的传递和即时转发。分布式网络非常复杂，长连接管理是重中之重，需要考虑到连接保活、连接检测、自动重连等方方面面的工作。

短连接Web服务和长连接IM服务之间是相互配合的。在分布式集群的环境下，用户首先通过短连接登录Web服务器。Web服务器在完成用户的账号/密码验证、返回UID和令牌（Token）时，还需要通过一定策略获取目标IM服务器的IP地址和端口号列表，并返回给客户端。客户端开始连接IM服务器，连接成功后，发送鉴权请求，鉴权成功则授权的长连接正式建立。

用户规模庞大时，无论是短连接Web服务器还是长连接IM服务器，都需要进行横向扩展可扩展到上十台、百台甚至千台服务器。只有这样，才能有良好性能的用户体验。因此，需要引入一个新的角色，短连接Web网关（WebGate）。

WebGate短连接网关的职责是代理大量的Web服务器，从而无感知地实现短连接的高并发。在客户端登录和进行其他短连接时，不直接连接Web服务器，而是连接Web网关。围绕Web网关和Web高并发的相关技术，可以使用Spring Cloud + Nginx架构，目前非常成熟，也很容易扩展。

除此之外，大量的IM服务器又如何协同和管理呢？基于ZooKeeper或者其他分布式协调中间件，可以非常方便、轻松地实现一个IM服务器集群的管理，包括而且不限于命名服务、服务注册、服务发现、负载均衡等管理。

当用户登录成功的时候，WebGate短连接网关可以通过负载均衡技术从ZooKeeper集群中找出一个可用的IM服务器的地址，返回给用户，让用户来建立长连接。

15.2 分布式IM的命名服务的实战案例

前面提到，一个高并发系统是由很多节点组成的，而且节点的数量是不断动态变化的。在一个即时消息（IM）通信系统中，从0到1再到N，用户量可能会越来越多，或者说由于某些活动影响会不断地出现流量洪峰。这时需要动态加入大量的节点。另外，由于服务器或者网络的原因，一些节点主动离开了集群。如何为大量的动态节点命名呢？最好的办法是使用分布式命名服务，按照一定的规则为动态上线和下线的工作节点命名。

“疯狂创客圈”的高并发CrazyIM实战学习项目基于ZooKeeper构建分布式命名服务，为每一个IM工作服务器节点动态命名。

15.2.1 IM节点的POJO类

首先定义一个POJO类，保存IM Worker节点的基础信息，如Netty服务IP、服务端口，以及Netty的服务连接数。具体如下：

```
package com.crazymakercircle.imServer.distributed;  
//省略import  
public class ImNode implements Comparable<ImNode> {  
  
    //worker 的Id，ZooKeeper负责生成  
    private long id;  
  
    //Netty服务的连接数
```

```
private AtomicInteger balance;

//Netty服务 IP
private String host;

//Netty服务端口
private String port;

public ImNode(String host, String port) {
    this.host = host;
    this.port = port;
}

public static ImNode getLocalInstance() {
    return null;
}

@Override
public Boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    ImNode node = (ImNode) o;
    return id == node.id && Objects.equals(host, node.host)
&&
        Objects.equals(port, node.port);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(id, host, port);
}

/**
 * 用来按照负载升序排列
 */
public int compareTo(ImNode o) {
    int weight1 = this.getBalance().get();
    int weight2 = o.getBalance().get();
    if (weight1 > weight2) {
        return 1;
    } else if (weight1 < weight2) {
        return -1;
    }
    return 0;
}
}
```

这个POJO类的IP、端口、balance负载和每一个节点的Netty服务器相关。id属性则利用ZooKeeper中的ZNode子节点可以顺序编号的性质，由ZooKeeper生成。

15.2.2 IM节点的ImWorker类

节点的命名服务的主要实现逻辑是：所有的工作节点都在ZooKeeper的同一个父节点下创建顺序节点，然后从返回的临时路径上取得属于自己的后缀编号。主要的代码如下：

```
package com.crazymakercircle.imServer.distributed;  
//省略import  
public class ImWorker {  
  
    //ZK Curator 客户端  
    private CuratorFramework client = null;  
  
    //保存当前zNode节点的路径，创建后返回  
    private String pathRegistered = null;  
  
    private ImNode node = ImNode.getLocalInstance();  
  
    private static ImWorker singleInstance = null;  
  
    //取得唯一的实例  
    public static ImWorker getInst() {  
        if (null == singleInstance) {  
            singleInstance = new ImWorker();  
            singleInstance.client=  
ZKclient.instance.getClient();  
            singleInstance.init();  
        }  
        return singleInstance;  
    }
```

```
}

private ImWorker() {
}

//在ZooKeeper中创建临时节点
public void init() {
    createParentIfNeeded(ServerConstants.MANAGE_PATH);

    //创建一个ZNode节点
    //节点的 payload 为当前Worker 实例
    try {
        byte[] payload = JsonUtil.Object2JsonBytes(node);

        pathRegistered = client.create()
            .creatingParentsIfNeeded()

        .withMode(CreateMode.EPHEMERAL_SEQUENTIAL)

        .forPath(ServerConstants.PATH_PREFIX, payload);
        //为node 设置Id
        node.setId(getId());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
/***
 * 取得IM 节点编号
 * @return 编号
 */
public long getId() {
    String sid = null;
    if (null == pathRegistered) {
        throw new RuntimeException("节点注册失败");
    }
    int index =
        pathRegistered.lastIndexOf(ServerConstants.PATH_PREFIX);
    if (index >= 0) {
        index += ServerConstants.PATH_PREFIX.length();
        sid = index <= pathRegistered.length() ?
            pathRegistered.substring(index) : null;
    }
    if (null == sid) {
        throw new RuntimeException("节点ID生成失败");
    }
    return Long.parseLong(sid);
}

/***
 * 增加负载，表示有用户登录成功

```

```

    * @return 成功状态
    */
public Boolean incBalance() {
    if (null == node) {
        throw new RuntimeException("还没有设置Node 节点");
    }
    //增加负载，并写回ZooKeeper
    while (true) {
        try {
            node.getBalance().getAndIncrement();
            byte[] payload =
                JsonUtil.Object2JsonBytes(this);
            client.setData().forPath(pathRegistered,
                payload);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}

/**
 * 减少负载，表示有用户下线
 * @return 成功状态
*/
public Boolean decrBalance() {
    if (null == node) {

```

```
        throw new RuntimeException("还没有设置Node 节点");
    }

    //增加负载，并写回ZooKeeper

    while (true) {

        try {

            int i = node.getBalance().decrementAndGet();

            if (i < 0) {

                node.getBalance().set(0);

            }

            byte[] payload =
JsonUtil.Object2JsonBytes(this);

            client.setData().forPath(pathRegistered,
payload);

            return true;

        } catch (Exception e) {

            return false;

        }

    }

}

/**
 * 创建父节点
 * @param managePath父节点路径
 */
private void createParentIfNeeded(String managePath) {
```

```
try {
    Stat stat =
client.checkExists().forPath(managePath);
    if (null == stat) {
        client.create()
            .creatingParentsIfNeeded()
            .withProtection()
            .withMode(CreateMode.PERSISTENT)
            .forPath(managePath);
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

注意，这里有三个ZNode相关的路径：MANAGE_PATH、pathPrefix和pathRegistered。

第一个MANAGE_PATH是一个常量，值为"/im/nodes"，为所有Worker临时工作节点的父亲节点的路径，在创建Worker节点之前，首先要检查一下父亲ZNode节点是否存在，否则先创建父亲节点。"/im/nodes"父亲节点的创建方式是，持久化节点，而不是临时节点。

第二路径pathPrefix是所有临时节点的前缀，值为"/im/nodes/"，是在工作路径后加上一个“/”分隔符；也可在工作

路径的后面加上其他的前缀字符，如”/im/nodes/id-”、”/im/nodes/seq-”等。

第三路径pathRegistered是临时节点创建成功之后返回的完整路径，例如/im/nodes/0000000000、/im/nodes/0000000001等。后边的编号是顺序的。

创建节点成功后，截取后边的编号数字，放在POJO对象id属性中供后边使用：

```
//为node 设置ID  
node.setId(getId());
```

15.3 Worker集群的负载均衡的实战案例

从理论上来说，负载均衡是一种手段，用来把对某种资源的访问分摊给不同的服务器，从而减轻单点的压力。在高并发的IM系统中，负载均衡需要将IM长连接分摊到不同的Netty服务器，防止单个Netty服务器负载过大，从而导致其不可用。

前面讲到，当用户登录成功的时候，短连接网关WebGate需要返回给用户一个可用的Netty服务器的地址，让用户来建立Netty长连接。每台Netty工作服务器在启动时都会去ZooKeeper的“/im/nodes”节点下注册临时节点。因此，短连接网关WebGate可以在用户登录成功之后，在“/im/nodes”节点下取得所有可用的Netty服务器列表，并通过一定的负载均衡算法计算得出一台Netty工作服务器，并返回给客户端。

15.3.1 ImLoadBalance负载均衡器

短连接网关WebGate需要通过查询ZooKeeper集群来获得最佳的Netty服务器。定义一个负载均衡器ImLoadBalance类，将计算最佳Netty服务器的算法放在负载均衡器中，ImLoadBalance的代码如下：

```
package com.crazymakercircle.Balance;  
//省略import  
public class ImLoadBalance {  
  
    //ZK客户端
```

```
private CuratorFramework client = null;

//工作节点的路径
private static String mangerPath = "/im/nodes";

public ImLoadBalance() {
}

public ImLoadBalance(String mangerPath) {
    this.client = ZKclient.INSTANCE.getClient();
    this.mangerPath = mangerPath;
}

public static final ImLoadBalance INSTANCE =
    new
ImLoadBalance(mangerPath);

/**
 * 获取负载最小的IM节点
 *
 * @return
 */
public ImNode getBestWorker() {
    List<ImNode> workers = getWorkers();
    ImNode best = balance(workers);
}
```

```
        return best;
    }

/**
 * 按照负载排序
 *
 * @param items 所有的节点
 * @return 负载最小的IM节点
 */
protected ImNodebalance(List<ImNode> items) {
    if (items.size() > 0) {
        //根据balance值从小到大排序
        Collections.sort(items);

        //返回balance值最小的那个
        return items.get(0);
    } else {
        return null;
    }
}

/**
 * 从ZooKeeper中拿到所有IM节点
 */
protected List<ImNode> getWorkers() {

```

```
List<ImNode> workers = new ArrayList<ImNode>();

List<String> children = null;

try {

    children =

client.getChildren().forPath(mangerPath);

} catch (Exception e) {

    e.printStackTrace();

    return null;

}

for (String child : children) {

    log.info("child:", child);

    byte[] payload = null;

    try {

        payload = client.getData().forPath(child);

    } catch (Exception e) {

        e.printStackTrace();

    }

    if (null == payload) {

        continue;

    }

    ImNode worker = JsonUtil.JsonBytes2Object(payload,

                                                ImNode.class);

    workers.add(worker);

}

return workers;
```

```
    }  
}
```

短连接网关WebGate会调用getBestWorker()方法取得最佳的IM服务器。在这个方法中，有两个很重要的方法：一个是取得所有的IM服务器列表，注意是带负载的；另一个是通过负载信息计算最小负载的服务器。

代码中的getWorkers()方法调用Curator的getChildren()方法获取子节点，取得"/im/nodes"目录下所有的临时节点。然后，调用getData方法取得每一个子节点的二进制负载。最后，将负载信息转成POJO ImNode对象。

获取到工作节点的POJO列表之后，在balance()方法中通过一个简单的排序算法计算出balance值最小的ImNode对象。

15.3.2 与WebGate的整合

短连接网关WebGate登录成功之后，需要通过负载均衡器ImLoadBalance类查询到最佳的Netty服务器，并且返回给客户端，代码如下：

```
package com.crazymakercircle.controller;  
//省略import  
@RequestMapping(value = "/user", produces =  
        MediaType.APPLICATION_JSON_UTF8_VALUE)  
@Api("User 相关的api")  
public class UserAction extends BaseController {
```

```
@Resource  
private UserService userService;  
  
/**  
 * Web短连接登录  
 * @param username 用户名  
 * @param password 命名  
 * @return 登录结果  
 */  
  
@ApiOperation(value = "登录", notes = "根据用户信息登录")  
@RequestMapping(value = "/login/{username}/{password}")  
public String loginAction(  
        @PathVariable("username") String username,  
        @PathVariable("password") String password) {  
  
    User user = new User();  
    user.setUserName(username);  
    user.setPassword(password);  
    User loginUser = userService.login(user);  
    LoginBack back = new LoginBack();  
  
    /**  
     * 取得最佳的Netty服务器  
     */  
  
    ImNodebestWorker =  
        ImLoadBalance.INSTANCE.getBestWorker();  
    back.setImNode(bestWorker);  
    back.setUser(loginUser);  
    back.setToken(loginUser.getUserId().toString());  
    String r = super.getJsonResult(back);
```

```
    return r;  
}  
  
//省略其他方法  
}
```

说明

出于学习的目的，CrazyIM在WebGate这块进行了极大的简化，使用一个非常简单的Web应用进行替代。用户登录也是一个模拟的操作，没有真正地去操作数据库。在生产场景项目中，WebGate必须对应到Spring Cloud + Nginx架构中的Nginx接入网关，整个WebGate是一个非常复杂的分布式应用。具体可以参考笔者的另一本书《Spring Cloud、Nginx高并发核心编程》。

很多小伙伴在开始上手学习分布式IM时，往往不能启动CrazyIM，这样的情况还不少。由于分布式系统依赖的组件非常多，因此启动起来确实很麻烦，这也算是分布式开发工程师、架构师比普通Java工程师、架构师“身价高”的原因。实际上，在顺利启动CrazyIM之前，在启动ZooKeeper集群、Redis之后，必须启动WebGate服务。具体的启动过程请参考“疯狂创客圈”的社群博客。

15.4 即时通信消息的路由和转发的实战案例

如果连接在不同的Netty Worker工作站点的客户端之间需要相互进行消息的发送，就需要在不同的Worker节点之间进行路由和转发。Worker节点的路由是指根据消息需要转发的目标用户找到用户连接所在的Worker节点。由于节点和节点之间都有可能需要相互转发，因此节点之间的连接是一种网状结构。每一个节点都需要具备路由的能力。

15.4.1 IM路由器WorkerRouter

为每一个Worker节点增加一个IM路由器类，名为WorkerRouter。为了能够转发到所有的节点，一是要订阅到集群中所有的在线Netty服务器，并且保存起来；二是要其他的Netty服务器建立一个长连接，用于转发消息。

WorkerRouter初始化代码如下：

```
package com.crazymakercircle.imServer.distributed;  
//省略import  
public class WorkerRouter {  
    //ZK客户端  
    private CuratorFramework client = null;  
  
    //唯一实例模式  
    private static WorkerRouter singleInstance = null;
```

```
//监听路径
private static final String path =
ServerConstants.MANAGE_PATH;

//节点的容器
private ConcurrentHashMap<Long, PeerSender> workerMap =
    new ConcurrentHashMap<>();

public static WorkerRouter getInst() {
    if (null == singleInstance) {
        singleInstance = new WorkerRouter();
        singleInstance.client =
ZKclient.instance.getClient();
        singleInstance.init();
    }
    return singleInstance;
}

//WorkerRouter初始化代码
private void init() {
    try {

        //订阅节点的增加和删除事件
        TreeCache treeCache = new TreeCache(client, path);
        TreeCacheListener l = new TreeCacheListener() {
            @Override
            public void childEvent(CuratorFramework client,
```

```
TreeCacheEvent event) throws  
Exception {  
  
    ChildData data = event.getData();  
  
    if (data != null) {  
  
        switch (event.getType()) {  
  
            case NODE_REMOVED:  
  
                processNodeRemoved(data);  
  
                break;  
  
            case NODE_ADDED:  
  
                processNodeAdded(data);  
  
                break;  
  
            default:  
  
                break;  
        }  
    } else {  
  
        log.info("[TreeCache] 节点数据为空, path=  
{ }",  
                data.getPath());  
    }  
}  
};  
  
treeCache.getListenable().addListener(l);  
  
treeCache.start();  
}  
} catch (Exception e) {  
  
    e.printStackTrace();  
}  
}
```

```
    }  
    //省略其他方法  
}
```

在上一节中，我们已经知道，一个节点上线时，首先要通过命名服务加入Netty集群中。在上面的代码中，WorkerRouter路由器使用Curator的TreeCache缓存订阅了节点的NODE_ADDED节点添加消息。当一个新的Netty节点加入时，调用processNodeAdded(data)方法在本地保存一份节点的POJO信息，并且建立一个消息中转的Netty客户连接。

处理节点添加的方法processNodeAdded(data)比较重要，代码如下：

```
/**  
 * 节点增加的处理  
 * @param data 新节点  
 */  
  
private void processNodeAdded(ChildData data) {  
    byte[] payload = data.getData();  
    ImNode n = ObjectUtil.JsonBytes2Object(payload,  
    ImNode.class);  
  
    long id =  
    ImWorker.getInst().getIdByPath(data.getPath());  
    n.setId(id);  
  
    log.info("[TreeCache] 节点更新端口, path={}, data={}",  
    data.getPath(), JsonUtil.pojoToJson(n));
```

```
        if (n.equals(getLocalNode())) {
            {
                log.info("[TreeCache]本地节点, path={}, data={}",
                         data.getPath(), JsonUtil.pojoToJson(n));
            }
            return;
        }

        PeerSender relaySender = workerMap.get(n.getId());
        //重复收到注册的事件
        if (null != relaySender &&
relaySender.getNode().equals(n)) {
            log.info("[TreeCache]节点重复增加, path={}, data={}",
                     data.getPath(), JsonUtil.pojoToJson(n));
            return;
        }

        if (null != relaySender) {
            //关闭老的连接
            relaySender.stopConnecting();
        }

        //创建一个消息转发器
        relaySender = new PeerSender(n);
        //建立转发的连接
        relaySender.doConnect();

        workerMap.put(n.getId(), relaySender);
    }
}
```

WorkerRouter路由器有一个容器成员workerMap，用于封装和保存所有的在线节点。当一个节点添加时，WorkerRouter取到添加的ZNode路径和负载。ZNode路径的后缀中有新节点的ID，而ZNode的payload负载中有新节点的Netty服务的IP地址和端口号，这三个信息共同构成新节点的POJO信息——ImNode节点信息。WorkerRouter在确定本地不存在该节点的转发器后，添加一个转发器PeerSender，将新节点的转发器保存在自己的容器中。

这里有一个问题，为什么在WorkerRouter路由器中不简单地保存新节点的POJO信息呢？因为WorkerRouter路由器的主要作用除了路由节点，还需要进行消息的转发，所以WorkerRouter路由器保存的是转发器PeerSender，而添加的远程Netty节点的POJO信息被封装在转发器中。

15.4.2 IM转发器PeerSender

IM转发器PeerSender封装了远程节点的IP地址、端口号以及ID。另外，PeerSender还维持了一个到远程节点的长连接。也就是说，它是一个Netty的NIO客户端，维护了一个到远程节点的Netty通道，通过这个通道将消息转发给远程的节点。

IM转发器PeerSender的核心代码如下：

```
package com.crazymakercircle.imServer.distributed;  
//省略import  
public class PeerSender {  
    //连接远程节点的Netty通道
```

```
private Channel channel;

//连接远程节点的POJO信息

private ImNode remoteNode;

/***
 * 连接标记
 */

private Boolean connectFlag = false;

private Bootstrap b;

private EventLoopGroup g;

public PeerSender(ImNode n) {

    this.remoteNode = n;

    b = new Bootstrap();
    g = new NioEventLoopGroup();

}

/***
 * 连接和重连
 */

public void doConnect() {

    //服务器IP地址
    String host = remoteNode.getHost();
    //服务端口
}
```

```
int port = Integer.parseInt(remoteNode.getPort());  
  
try {  
    if (b != null && b.group() == null) {  
        b.group(g);  
  
    b.channel(NioSocketChannel.class);  
  
    b.option(ChannelOption.SO_KEEPALIVE, true);  
    b.option(ChannelOption.ALLOCATOR,  
PooledByteBufAllocator.DEFAULT);  
    b.remoteAddress(host, port);  
  
    //设置通道初始化  
    b.handler(  
        new  
ChannelInitializer<SocketChannel>() {  
    public void  
initChannel(SocketChannel ch) {  
  
ch.pipeline().addLast(new ProtobufEncoder());  
    }  
    }  
);  
log.info(new Date()  
+ "开始连接分布式节点",
```

```
remoteNode.toString());  
  
        ChannelFuture f = b.connect();  
        f.addListener(connectedListener);  
  
        //阻塞  
        //f.channel().closeFuture().sync();  
    } else if (b.group() != null) {  
        log.info(new Date()  
                + "再一次开始连接分布式节点",  
        remoteNode.toString());  
        ChannelFuture f = b.connect();  
        f.addListener(connectedListener);  
    }  
} catch (Exception e) {  
    log.info("客户端连接失败!" +  
    e.getMessage());  
}  
}  
//省略其他方法  
}
```

在IM转发器中，主体是与Netty通信相关的代码，所以比较简单。严格来说，IM转发器仅仅是一个Netty的客户端，它比Netty服务器的代码简单一些。

转发器有一个消息转发的方法，直接通过Netty通道将消息发送到远程节点，代码如下：

```
/**  
 * 消息转发的方法  
 * @param pkg 聊天消息  
 */  
  
public void writeAndFlush(Object pkg) {  
    if (connectFlag == false) {  
        log.error("分布式节点未连接:", remoteNode.toString());  
        return;  
    }  
    channel.writeAndFlush(pkg);  
}
```

15.5 在线用户统计的实战案例

计数器是用来计数的。在分布式环境中，常规的计数器是不能使用的，在此介绍ZooKeeper实现的分布式计数器。利用ZooKeeper可以实现一个集群共享的计数器，只要使用相同的path就可以得到最新的计数器值，这是由ZooKeeper的一致性保证的。

15.5.1 Curator的分布式计数器

Curator有两个计数器：一个用int类型来计数（SharedCount），一个用long类型来计数（DistributedAtomicLong）。下面使用DistributedAtomicLong来实现高并发IM系统中的在线用户统计，代码如下：

```
package com.crazymakercircle.imServer.distributed;  
//省略import  
  
public class OnlineCounter {  
  
    private static final int QTY = 5;  
    private static final String PATH =  
        ServerConstants.COUNTER_PATH;  
    //ZK客户端  
    private CuratorFramework client = null;  
    //唯一实例模式  
    private static OnlineCounter singleInstance = null;  
    //分布式计数器
```

```
DistributedAtomicLong onlines = null;

public static OnlineCounter getInst() {
    if (null == singleInstance) {
        singleInstance = new OnlineCounter();
        singleInstance.client =
ZKclient.instance.getClient();
        singleInstance.init();
    }
    return singleInstance;
}

private void init() {
    //分布式计数器，失败时重试10次，每次间隔30毫秒
    onlines = new DistributedAtomicLong(client, PATH,
        new RetryNTimes(10, 30));
}

private OnlineCounter() {
}

/**
 * 增加计数
 */
public boolean increment() {
    boolean result = false;
```

```
AtomicValue<Long>val = null;

try {
    val = onlines.increment();
    result = val.succeeded();
    System.out.println("old cnt: " + val.preValue()
        + " new cnt : " +
    val.postValue()
        + " result:" +
    val.succeeded());
}

} catch (Exception e) {
    e.printStackTrace();
}

return result;
}

/**
 * 减少计数
 */
public boolean decrement() {
    boolean result = false;
    AtomicValue<Long> val = null;
    try {
        val = onlines.decrement();
        result = val.succeeded();
        System.out.println("old cnt: " + val.preValue()
```

```
+ "      new cnt : " +  
val.postValue()  
      + "      result:" +  
val.succeeded());  
} catch (Exception e) {  
    e.printStackTrace();  
}  
return result;  
}  
}
```

说明

此分布式计数器仅仅作为学习使用，让大家了解一下分布式计数器的概念和实现思路。使用ZooKeeper分布式计数器的优势是高可用，劣势是低性能。在高并发场景下，分布式计数器需要基于Redis去实现，这个可以作为练手项目，自己去实战一下。

15.5.2 用户上线和下线的统计

当用户上线的时候，调用increase()方法分布式地增加一次计数：

```
package com.crazymakercircle.imServer.server.session;  
//省略import
```

```
public class SessionManger {  
  
    /**  
     * 登录成功之后，增加session对象  
     */  
  
    public void addLocalSession(LocalSession session) {  
        String sessionId = session.getSessionId();  
        localSessionMap.put(sessionId, session);  
  
        String uid = session.getUser().getUserId();  
  
        //增加用户数  
        OnlineCounter.getInst().increment();  
        log.info("本地session增加: {}, 在线总数:{} ",  
            JsonUtil.pojoToJson(session.getUser()),  
            OnlineCounter.getInst().getCurValue());  
        ImWorker.getInst().incBalance();  
  
        //增加用户的session 信息到缓存  
        userSessionsDAO.cacheUser(uid, sessionId);  
        /**  
         * 通知其他节点  
         */  
        notifyOtherImNode(session, Notification.SESSION_ON);  
    }  
}
```

```
    }  
}
```

本章的实例代码来自“疯狂创客圈”社群的高并发学习项目CrazyIM。由于项目在不断迭代，因此在大家读到本章时书中代码可能已经过时，请参考社群最新版本的CrazyIM代码。不过，无论细节如何迭代，设计思路基本都是一致的。CrazyIM项目有两点需要特别说明：

- (1) 此项目的架构和互联网大厂的主流分布式Java项目的架构基本类似，可以作为进阶Java核心架构或者入职互联网大厂的理想练习项目。
- (2) 此项目的架构和很多大数据开源项目在架构上基本类似，可以作为大数据工程师的基础练习项目。

本章的目的仅仅是抛砖引玉，寥寥数千字，无法彻底地将一个支持亿级流量的IM项目的架构及其实现剖析得非常清楚。后续“疯狂创客圈”会结合本书将内容更加全面地呈现给大家。

说明

要在生产场景下扛住高并发，是需要针对实际问题进行性能的专项优化，并且需要堆积大量的硬件资源。CrazyIM项目的初衷仅仅是进行亿级流量的学习，只能从两个方面具备扛住亿级流量的潜力：一是提升单节点的高并发潜力，二是系统具备横向扩展的能力。生产场景下的亿级流量项目与CrazyIM学习项目在架构思路总体上是相同的，所以CrazyIM一定是高并发实战的优秀学习项目。

