



# Netty、Redis、 ZooKeeper高并发实战

聚焦实战技能，剖析底层原理

尼恩 编著

解读高并发开发、架构、面试中的核心难题



机械工业出版社  
China Machine Press

# Netty、Redis、Zookeeper高并发实战

尼恩 编著

ISBN: 978-7-111-63290-0

本书纸版由机械工业出版社于2019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

## 前言

### 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 高并发IM的综合实践

    1.4.1 高并发IM的学习价值

    1.4.2 庞大的应用场景

#### 1.5 Netty、Redis、ZooKeeper实践计划

    1.5.1 第1天：Java NIO实践

    1.5.2 第2天：Reactor反应器模式实践

    1.5.3 第3天：异步回调模式实践

    1.5.4 第4天：Netty基础实践

    1.5.5 第5天：解码器（Decoder）与编码器（Encoder）实践

    1.5.6 第6天：JSON和ProtoBuf序列化实践

    1.5.7 第7~10天：基于Netty的单聊实战

    1.5.8 第11天：ZooKeeper实践计划

    1.5.9 第12天：Redis实践计划

#### 1.6 本章小结

### 第2章 高并发IO的底层原理

#### 2.1 IO读写的基础原理

    2.1.1 内核缓冲区与进程缓冲区

    2.1.2 详解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（Blocking IO）

    2.2.2 同步非阻塞NIO（None Blocking IO）

    2.2.3 IO多路复用模型（IO Multiplexing）

    2.2.4 异步IO模型（Asynchronous IO）

#### 2.3 通过合理配置来支持百万级并发连接

#### 2.4 本章小结

### 第3章 Java NIO通信基础详解

#### 3.1 Java NIO简介

    3.1.1 NIO和OIO的对比

    3.1.2 通道（Channel）

3.1.3 Selector选择器

3.1.4 缓冲区（Buffer）

### 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

3.2.3 4个属性的小结

### 3.3 详解NIO Buffer类的重要方法

3.3.1 allocate()创建缓冲区

3.3.2 put()写入到缓冲区

3.3.3 flip()翻转

3.3.4 get()从缓冲区读取

3.3.5 rewind()倒带

3.3.6 mark( )和reset( )

3.3.7 clear( )清空缓冲区

3.3.8 使用Buffer类的基本步骤

### 3.4 详解NIO Channel（通道）类

3.4.1 Channel（通道）的主要类型

3.4.2 FileChannel文件通道

3.4.3 使用FileChannel完成文件复制的实践案例

3.4.4 SocketChannel套接字通道

3.4.5 使用SocketChannel发送文件的实践案例

3.4.6 DatagramChannel数据报通道

3.4.7 使用DatagramChannel数据包通道发送数据的实践案例

### 3.5 详解NIO Selector选择器

3.5.1 选择器以及注册

3.5.2 SelectableChannel可选择通道

3.5.3 SelectionKey选择键

3.5.4 选择器使用流程

3.5.5 使用NIO实现Discard服务器的实践案例

3.5.6 使用SocketChannel在服务器端接收文件的实践案例

### 3.6 本章小结

## 第4章 鼎鼎大名的Reactor反应器模式

### 4.1 Reactor反应器模式为何如此重要

4.1.1 为什么首先学习Reactor反应器模式

4.1.2 Reactor反应器模式简介

4.1.3 多线程OIO的致命缺陷

### 4.2 单线程Reactor反应器模式

4.2.1 什么是单线程Reactor反应器

4.2.2 单线程Reactor反应器的参考代码

4.2.3 一个Reactor反应器版本的EchoServer实践案例

4.2.4 单线程Reactor反应器模式的缺点

### 4.3 多线程的Reactor反应器模式

- 4.3.1 多线程池Reactor反应器演进
  - 4.3.2 多线程Reactor反应器的实践案例
  - 4.3.3 多线程Handler处理器的实践案例
- 4.4 Reactor反应器模式小结
- 4.5 本章小结

## 第5章 并发基础中的Future异步回调模式

- 5.1 从泡茶的案例说起
- 5.2 join异步阻塞
  - 5.2.1 线程的join合并流程
  - 5.2.2 使用join实现异步泡茶喝的实践案例
  - 5.2.3 详解join合并方法
- 5.3 FutureTask异步回调之重武器
  - 5.3.1 Callable接口
  - 5.3.2 初探FutureTask类
  - 5.3.3 Future接口
  - 5.3.4 再探FutureTask类
  - 5.3.5 使用FutureTask类实现异步泡茶喝的实践案例
- 5.4 Guava的异步回调
  - 5.4.1 详解FutureCallback
  - 5.4.2 详解ListenableFuture
  - 5.4.3 ListenableFuture异步任务
  - 5.4.4 使用Guava实现泡茶喝的实践案例
- 5.5 Netty的异步回调模式
  - 5.5.1 详解GenericFutureListener接口
  - 5.5.2 详解Netty的Future接口
  - 5.5.3 ChannelFuture的使用
  - 5.5.4 Netty的出站和入站异步回调
- 5.6 本章小结

## 第6章 Netty原理与基础

- 6.1 第一个Netty的实践案例DiscardServer
  - 6.1.1 创建第一个Netty项目
  - 6.1.2 第一个Netty服务器端程序
  - 6.1.3 业务处理器NettyDiscardHandler
  - 6.1.4 运行NettyDiscardServer
- 6.2 解密Netty中的Reactor反应器模式
  - 6.2.1 回顾Reactor反应器模式中IO事件的处理流程
  - 6.2.2 Netty中的Channel通道组件
  - 6.2.3 Netty中的Reactor反应器
  - 6.2.4 Netty中的Handler处理器
  - 6.2.5 Netty的流水线（Pipeline）
- 6.3 详解Bootstrap启动器类
  - 6.3.1 父子通道

- 6.3.2 EventLoopGroup线程组
  - 6.3.3 Bootstrap的启动流程
  - 6.3.4 ChannelOption通道选项
  - 6.4 详解Channel通道
    - 6.4.1 Channel通道的主要成员和方法
    - 6.4.2 EmbeddedChannel嵌入式通道
  - 6.5 详解Handler业务处理器
    - 6.5.1 ChannelInboundHandler通道入站处理器
    - 6.5.2 ChannelOutboundHandler通道出站处理器
    - 6.5.3 ChannelInitializer通道初始化处理器
    - 6.5.4 ChannelInboundHandler的生命周期的实践案例
  - 6.6 详解Pipeline流水线
    - 6.6.1 Pipeline入站处理流程
    - 6.6.2 Pipeline出站处理流程
    - 6.6.3 ChannelHandlerContext上下文
    - 6.6.4 截断流水线的处理
    - 6.6.5 Handler业务处理器的热拔插
  - 6.7 详解ByteBuf缓冲区
    - 6.7.1 ByteBuf的优势
    - 6.7.2 ByteBuf的逻辑部分
    - 6.7.3 ByteBuf的重要属性
    - 6.7.4 ByteBuf的三组方法
    - 6.7.5 ByteBuf基本使用的实践案例
    - 6.7.6 ByteBuf的引用计数
    - 6.7.7 ByteBuf的Allocator分配器
    - 6.7.8 ByteBuf缓冲区的类型
    - 6.7.9 三类ByteBuf使用的实践案例
    - 6.7.10 ByteBuf的自动释放
  - 6.8 ByteBuf浅层复制的高级使用方式
    - 6.8.1 slice切片浅层复制
    - 6.8.2 duplicate整体浅层复制
    - 6.8.3 浅层复制的问题
  - 6.9 EchoServer回显服务器的实践案例
    - 6.9.1 NettyEchoServer回显服务器的服务器端
    - 6.9.2 共享NettyEchoServerHandler处理器
    - 6.9.3 NettyEchoClient客户端代码
    - 6.9.4 NettyEchoClientHandler处理器
  - 6.10 本章小结
- 第7章 Decoder与Encoder重要组件
- 7.1 Decoder原理与实践
    - 7.1.1 ByteToMessageDecoder解码器
    - 7.1.2 自定义Byte2IntegerDecoder整数解码器的实践案例

- 7.1.3 ReplayingDecoder解码器
  - 7.1.4 整数的分包解码器的实践案例
  - 7.1.5 字符串的分包解码器的实践案例
  - 7.1.6 MessageToMessageDecoder解码器
  - 7.2 开箱即用的Netty内置Decoder
    - 7.2.1 LineBasedFrameDecoder解码器
    - 7.2.2 DelimiterBasedFrameDecoder解码器
    - 7.2.3 LengthFieldBasedFrameDecoder解码器
    - 7.2.4 多字段Head-Content协议数据帧解析的实践案例
  - 7.3 Encoder原理与实践
    - 7.3.1 MessageToByteEncoder编码器
    - 7.3.2 MessageToMessageEncoder编码器
  - 7.4 解码器和编码器的结合
    - 7.4.1 ByteToMessageCodec编解码器
    - 7.4.2 CombinedChannelDuplexHandler组合器
  - 7.5 本章小结
- 第8章 JSON和ProtoBuf序列化
- 8.1 详解粘包和拆包
    - 8.1.1 半包问题的实践案例
    - 8.1.2 什么是半包问题
    - 8.1.3 半包现象的原理
  - 8.2 JSON协议通信
    - 8.2.1 JSON序列化的通用类
    - 8.2.2 JSON序列化与反序列化的实践案例
    - 8.2.3 JSON传输的编码器和解码器之原理
    - 8.2.4 JSON传输之服务器端的实践案例
    - 8.2.5 JSON传输之客户端的实践案例
  - 8.3 Protobuf协议通信
    - 8.3.1 一个简单的proto文件的实践案例
    - 8.3.2 控制台命令生成POJO和Builder
    - 8.3.3 Maven插件生成POJO和Builder
    - 8.3.4 消息POJO和Builder的使用之实践案例
  - 8.4 Protobuf编解码的实践案例
    - 8.4.1 Protobuf编码器和解码器的原理
    - 8.4.2 Protobuf传输之服务器端的实践案例
    - 8.4.3 Protobuf传输之客户端的实践案例
  - 8.5 详解Protobuf协议语法
    - 8.5.1 proto的头部声明
    - 8.5.2 消息结构体与消息字段
    - 8.5.3 字段的数据类型
    - 8.5.4 其他的语法规范
  - 8.6 本章小结

## 第9章 基于Netty的单体IM系统的开发实践

### 9.1 自定义ProtoBuf编解码器

9.1.1 自定义Protobuf编码器

9.1.2 自定义Protobuf解码器

9.1.3 IM系统中Protobuf消息格式的设计

### 9.2 概述IM的登录流程

9.2.1 图解登录/响应流程的9个环节

9.2.2 客户端涉及的主要模块

9.2.3 服务器端涉及的主要模块

### 9.3 客户端的登录处理的实践案例

9.3.1 LoginConsoleCommand和User POJO

9.3.2 LoginSender发送器

9.3.3 ClientSession客户端会话

9.3.4 LoginResponceHandler登录响应处理器

9.3.5 客户端流水线的装配

### 9.4 服务器端的登录响应的实践案例

9.4.1 服务器流水线的装配

9.4.2 LoginRequestHandler登录请求处理器

9.4.3 LoginProcesser用户验证逻辑

9.4.4 EventLoop线程和业务线程相互隔离

### 9.5 详解ServerSession服务器会话

9.5.1 通道的容器属性

9.5.2 ServerSession服务器端会话类

9.5.3 SessionMap会话管理器

### 9.6 点对点单聊的实践案例

9.6.1 简述单聊的端到端流程

9.6.2 客户端的ChatConsoleCommand收集聊天内容

9.6.3 客户端的CommandController发送POJO

9.6.4 服务器端的ChatRedirectHandler消息转发

9.6.5 服务器端的ChatRedirectProcesser异步处理

9.6.6 客户端的ChatMsgHandler接收POJO

### 9.7 详解心跳检测

9.7.1 网络连接的假死现象

9.7.2 服务器端的空闲检测

9.7.3 客户端的心跳报文

### 9.8 本章小结

## 第10章 ZooKeeper分布式协调

### 10.1 ZooKeeper伪集群安装和配置

10.1.1 创建数据目录和日志目录:

10.1.2 创建myid文件

10.1.3 创建和修改配置文件

10.1.4 配置文件示例

- 10.1.5 启动ZooKeeper伪集群
  - 10.2 使用ZooKeeper进行分布式存储
    - 10.2.1 详解ZooKeeper存储模型
    - 10.2.2 zkCli客户端命令清单
  - 10.3 ZooKeeper应用开发的实践
    - 10.3.1 ZkClient开源客户端介绍
    - 10.3.2 Curator开源客户端介绍
    - 10.3.3 Curator开发的环境准备
    - 10.3.4 Curator客户端实例的创建
    - 10.3.5 通过Curator创建ZNode节点
    - 10.3.6 在Curator中读取节点
    - 10.3.7 在Curator中更新节点
    - 10.3.8 在Curator中删除节点
  - 10.4 分布式命名服务的实践
    - 10.4.1 ID生成器
    - 10.4.2 ZooKeeper分布式ID生成器的实践案例
    - 10.4.3 集群节点的命名服务之实践案例
    - 10.4.4 使用ZK实现SnowFlakeID算法的实践案例
  - 10.5 分布式事件监听的重点
    - 10.5.1 Watcher标准的事件处理器
    - 10.5.2 NodeCache节点缓存的监听
    - 10.5.3 PathChildrenCache子节点监听
    - 10.5.4 Tree Cache节点树缓存
  - 10.6 分布式锁的原理与实践
    - 10.6.1 公平锁和可重入锁的原理
    - 10.6.2 ZooKeeper分布式锁的原理
    - 10.6.3 分布式锁的基本流程
    - 10.6.4 加锁的实现
    - 10.6.5 释放锁的实现
    - 10.6.6 分布式锁的使用
    - 10.6.7 Curator的InterProcessMutex可重入锁
  - 10.7 本章小结
- 第11章 分布式缓存Redis
- 11.1 Redis入门
    - 11.1.1 Redis安装和配置
    - 11.1.2 Redis客户端命令
    - 11.1.3 Redis Key的命名规范
  - 11.2 Redis数据类型
    - 11.2.1 String字符串
    - 11.2.2 List列表
    - 11.2.3 Hash哈希表
    - 11.2.4 Set集合

- 11.2.5 Zset有序集合
  - 11.3 Jedis基础编程的实践案例
    - 11.3.1 Jedis操作String字符串
    - 11.3.2 Jedis操作List列表
    - 11.3.3 Jedis操作Hash哈希表
    - 11.3.4 Jedis操作Set集合
    - 11.3.5 Jedis操作Zset有序集合
  - 11.4 JedisPool连接池的实践案例
    - 11.4.1 JedisPool的配置
    - 11.4.2 JedisPool创建和预热
    - 11.4.3 JedisPool的使用
  - 11.5 使用spring-data-redis完成
    - 11.5.1 CRUD中应用缓存的场景
    - 11.5.2 配置spring-redis.xml
    - 11.5.3 使用RedisTemplate模板API
    - 11.5.4 使用RedisTemplate模板API完成CRUD的实践案例
    - 11.5.5 使用RedisCallback回调完成CRUD的实践案例
  - 11.6 Spring的Redis缓存注解
    - 11.6.1 使用Spring缓存注解完成CRUD的实践案例
    - 11.6.2 spring-redis.xml中配置的调整
    - 11.6.3 详解@CachePut和@Cacheable注解
    - 11.6.4 详解@CacheEvict注解
    - 11.6.5 详解@Caching组合注解
  - 11.7 详解SpringEL（SpEL）
    - 11.7.1 SpEL运算符
    - 11.7.2 缓存注解中的SpringEL表达式
  - 11.8 本章小结
- 第12章 亿级高并发IM架构的开发实践
- 12.1 如何支撑亿级流量的高并发IM架构的理论基础
    - 12.1.1 亿级流量的系统架构的开发实践
    - 12.1.2 高并发架构的技术选型
    - 12.1.3 详解IM消息的序列化协议选型
    - 12.1.4 详解长连接和短连接
  - 12.2 分布式IM的命名服务的实践案例
    - 12.2.1 IM节点的POJO类
    - 12.2.2 IM节点的ImWorker类
  - 12.3 Worker集群的负载均衡之实践案例
    - 12.3.1 ImLoadBalance负载均衡器
    - 12.3.2 与WebGate的整合
  - 12.4 即时通信消息的路由和转发的实践案例
    - 12.4.1 IM路由器WorkerRouter
    - 12.4.2 IM转发器WorkerReSender

## 12.5 Feign短连接RESTful调用

12.5.1 短连接API的接口准备

12.5.2 声明远程接口的本地代理

12.5.3 远程API的本地调用

## 12.6 分布式的在线用户统计的实践案例

12.6.1 Curator的分布式计数器

12.6.2 用户上线和下线的统计

## 12.7 本章小结

# 前言

移动时代、5G时代、物联网时代的大幕已经开启，它们对于高性能、高并发的开发知识和技术的要求，抬升了Java工程师的学习台阶和面试门槛。

大公司的面试题从某个侧面映射出生产场景中对专项技术的要求。高并发的面试题以前基本是BAT等大公司的专利，现在几乎蔓延至与Java项目相关的整个行业。例如，与Java NIO、Reactor模式、高性能通信、分布式锁、分布式ID、分布式缓存、高并发架构等技术相关的面试题，从以前的加分题变成了现在的基础题，这也映射出开发Java项目所必需的技术栈：分布式Java框架、Redis缓存、分布式搜索ElasticSearch、分布式协调ZooKeeper、消息队列Kafka、高性能通信框架Netty。

## 本书内容

本书的内容源于“疯狂创客圈社群”的博客，以及社群持续迭代的CrazyIM项目，虽然书中重在讲解Netty、Redis、ZooKeeper的使用方法，但是还有一个更大的价值，就是为大家打下Java高并发开发技术的坚实基础。

首先，本书从操作系统的底层原理开始讲解：浅显易懂地剖析高并发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OIO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；从Java的线程Join和线程池开始，介绍Java Future和Guava ListenableFuture两种常用异步回调技术。这些原理方面的基础知识非常重要，是大家在日常开发Java后台应用时解决实际问题的金钥匙。

接着，重点讲解Netty。这是目前当之无愧的高性能通信框架皇冠上的明珠，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。这里有两大特色：一是从Reactor模式入手，以四两拨千斤的方式来学习Netty原理；二是通过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。

然后，对ZooKeeper进行详细的介绍。除了全面地介绍使用Curator API操作ZooKeeper之外，还从实战的角度出发，介绍如何使用ZooKeeper来设计分布式ID生成器，并对重要的SnowFlake算法进行详细的介绍。另外，还通过图文并茂和结合小故事的方式浅显易懂地介绍分布式锁的基本原理，并完成一个ZooKeeper分布式锁的小实践案例。

接下来，从实践开发层面对Redis进行说明，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来完成两种方式的数据分布式缓存，并详尽地介绍Spring的缓存注解以及涉及的SpEL表达式语言。

最后，通过CrazyIM项目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

- (1) 对Java NIO、高性能IO、高并发编程感兴趣的大专院校学生。
- (2) 需要学习Java高并发技术、高并发架构的初、中级Java工程师。
- (3) 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/sfasdfasdfsdf/netty\\_redis\\_zookeeper\\_source\\_code.git](https://gitee.com/sfasdfasdfsdf/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](http://www.hzbook.com)）下载，先搜索到本书，然后在页面上的“资料下载”模块下载即可。如果下载有问题，请发送电子邮件至 [booksaga@126.com](mailto:booksaga@126.com)，邮件主题为“求Netty、Redis、ZooKeeper高并发实战下载资源”

## 勘误和支持

由于作者水平和能力有限，不妥之处在所难免，希望读者批评指正。本书的读者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

写书，不仅仅是一项技术活，而且是一项工匠活，为了确保书中知识的全面性、系统性，我需要不断地思考与总结。为了保证书中的每一行程序都是正确的，我需要反复地编写LLT用例去进行验证。总之，一本优质的书，意味着需要牺牲陪伴家人的大量时间。感谢我的妻子、孩子们给我一贯的支持和帮助！

感谢卞诚君老师在我写书过程中给予的指导和帮助。没有他的提议，我不会想到将自己发布在“疯狂创客圈”社群中高并发方面的博客文章整理成一本书出版。另外，还让我感觉到写博客和写书不是一个层面的事情。博客里的很多内容是不全面、不严谨的，甚至是错误的。

感谢“疯狂创客圈”社群中的小伙伴们，虽然你们的很多技术难题，我不一定能给出最佳的解答方案，但正是因为一路同行，一直坦诚、纯粹的技术交流，大家相互启发了许多技术灵感，拓展了彼此的技术视野，最终提升了水平。欢迎大家来“砸”问题，也欢迎大家多多交流。

尼恩

中国武汉

# 第1章 高并发时代的必备技能

高并发时代已然到来，Netty、Redis、ZooKeeper是高并发时代的必备工具。

## 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，它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。

注：NOI是指非阻塞输入输出（Non-Blocking IO），也称非阻塞IO。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IO，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、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、主流的分布式通信框架Dubbo，它们都使用了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Netty提供异步的、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IO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IO操作结果。

与JDK原生NIO相比，Netty提供了相对十分简单易用的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框架对比，Netty的综合性能最优。
- 成熟、稳定，Netty修复了已经发现的所有JDK NIO中的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之一。几乎所有互联网中间件或者大数据领域均离不开Netty，掌握Netty是作为一名初中级工程师迈向高级工程师重要的技能之一。

目前来说，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，就经常会问一些高性能通信框架方面的问题，还会问一些“你有没有读过什么著名框架的源代码？”等类似的问题。

如果掌握了Netty相关的技术问题，更进一步说，如果你能全面地阅读和掌握Netty源代码，相信面试大公司时，一定底气十足，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、高效的一款开源缓存。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，如果某个页面经常会被访问到，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，从而减轻了网站的负担，降低了网站的延迟。

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。Redis的Value值类型包括：string字符串类型、map映射类型、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：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（Key-Value）内存数据库（如Memcached）而言，Redis具有如下特点：

（1）速度快 不需要等待磁盘的IO，在内存之间进行的数据存储和查询，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 除了string之外，还有list、hash、set、sortedset，一共有五种类型。

（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，如果不同的节点需要访问到一个资源，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，需要使用到分布式锁。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

突破了单体瓶颈之后的高并发，就必须靠集群了，而集群的分布式架构和协调，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。在当时，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。在项目初期给这个项目命名的时候，准备和很多项目一样，按照雅虎公司的惯例要用动物的名字来命名的（例如著名的Pig项目）。在进行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，大家纷纷表示就叫动物园管理员吧，于是，ZooKeeper的名字由此诞生了。当然，取名ZooKeeper，也绝非没有一点儿道理。ZooKeeper的功能，正好是用来协调分布式环境的，协调各个以动物命名的分布式组件，所以，ZooKeeper这个名字也就“名副其实”了。

### 1.3.2 ZooKeeper的优势

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。需要注意的是，ZooKeeper官方文档中特别强调，由于FreeBSD系统的JVM（Java Virtual Machine，即Java虚拟机）对Java的NIO Selector选择器支持得不是很好，因此不建议在FreeBSD系统上部署生产环境的ZooKeeper服务器。

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、重复读。  
注：脏读是指在数据库存取中无效数据的读出。

ZooKeeper提供的功能都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而且要达到高吞吐、低延迟同时的还要保持一致性和可用性，实际上是非常困难的。因此，借助ZooKeeper提供的这些功能，开发者就可以轻松在ZooKeeper之上构建自己的各种分布式系统。

## 1.4 高并发IM的综合实践

为了方便交流和学习，笔者组织了一帮高性能发烧友，成立了一个高性能社群，叫作“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，叫作“CrazyIM”。

## 1.4.1 高并发IM的学习价值

为什么要开始一个高并发IM（即时通信）的实践呢？

首先，通过实践完成一个分布式、高并发的IM系统，具有相当的技术挑战性。这一点，对于从事传统的企业级Web开发者来说，相当于进入了一片全新的天地。企业级Web，QPS（Query Per Second，每秒查询率）峰值可能在1000以内，甚至在100以内，没有多少技术挑战性和含金量，属于重复性的CRUD的体力活。注：CRUD是指Create（创建）、Retrieve（查询）、Update（更新）和Delete（删除）。而一个分布式、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。对于此纵深多层次化的、递进的高并发需求，将无极限地考验着系统的性能。需要不断地从通信协议、到系统的架构进行优化，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，就应该具备全栈式的架构能力，对不同用户规模的、差异化的应用场景，提供和架构出与对应的应用场景相匹配的高并发IM系统。也就是说，IM系统综合性相对较强，相关的技术需要覆盖到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。

接下来具体看看高并发IM的应用场景吧。

## 1.4.2 庞大的应用场景

一切高实时性通信、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典型的应用场景如下：私信、聊天、大规模推送、视频会议、弹幕、抽奖、互动游戏、基于位置的应用（Uber、滴滴司机位置）、在线教育、智能家居等，如图1-1所示。

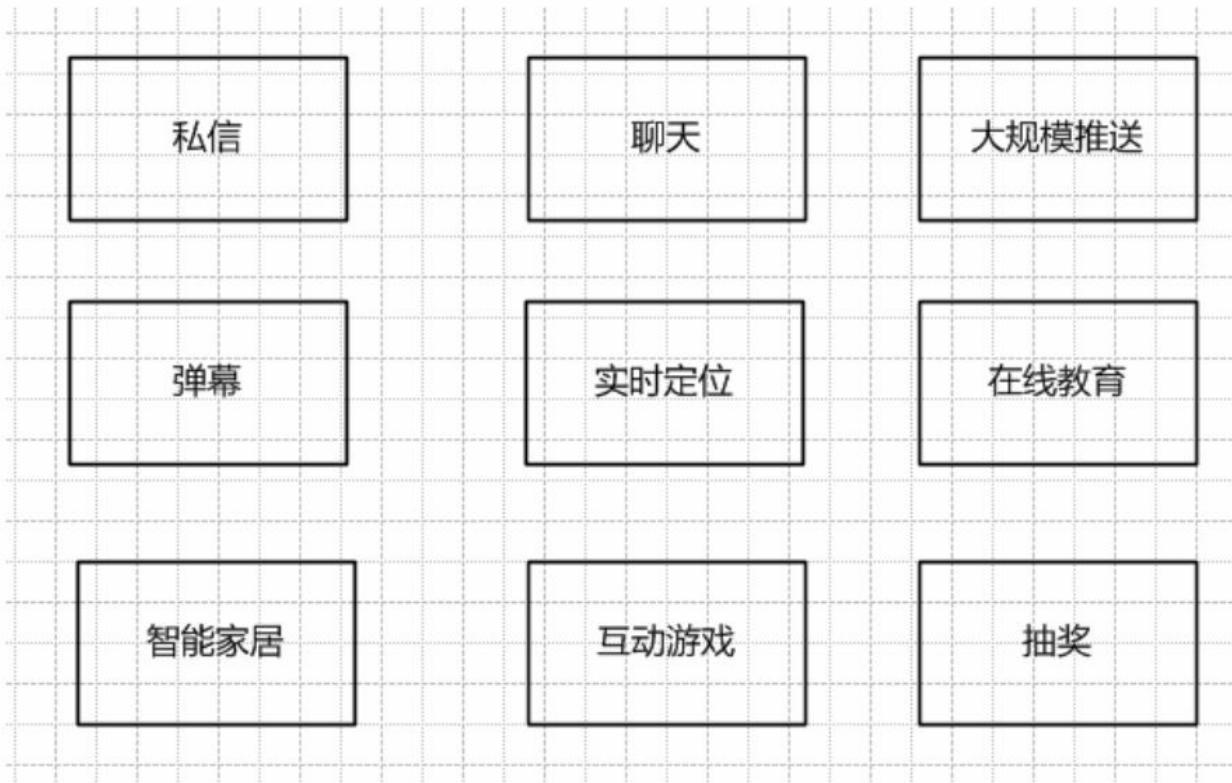


图1-1 高并发IM典型的应用场景

尤其是对于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应用不可或缺的重要组成部分，推送服务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，它们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，一定是需要强悍的后台系统。

有这么多的应用场景，对于想成长为Java高手的小伙伴们，高并发IM都是绕不开的一个话题。

对于想在后台有所成就的小伙伴们来说，高并发IM实践，更是在与终极BOSS PK之前的一场不可避免的打怪练手。

## 1.5 Netty、Redis、ZooKeeper实践计划

下面从最基础的NIO入手，列出一个大致12天的实践计划，帮助大家深入掌握Netty、Redis、ZooKeeper。

## 1.5.1 第1天：Java NIO实践

### 实践一：使用FileChannel复制文件

通过使用Channel通道，完成复制文件。

本环节的目标是掌握以下知识：Java NIO中ByteBuffer、Channel两个重要组件的使用。

接着是升级实战的案例，使用文件Channel通道的transferFrom方法，完成高效率的文件复制。

### 实践二：使用SocketChannel传输文件

本环节的目标是掌握以下知识：

- 非阻塞客户端在发起连接后，需要不断的自旋，检测连接是否完成的。
- SocketChannel传输通道的read读取方法、write写入方法。
- 在SocketChannel传输通道关闭前，尽量发送一个输出结束标志到对方端。

### 实践三：使用DatagramChannel传输数据

客户端使用DatagramChannel发送数据，服务器端使用DatagramChannel接收数据。

本环节的目标是掌握以下知识：

- 使用接收数据方法receive，使用发送数据方法send。
- DatagramChannel和SocketChannel两种通道，在发送、接收数据上的不同。

### 实战四：使用NIO实现Discard服务器

客户端功能：发送一个数据包到服务器端，然后关闭连接。服务器端也很简单，收到客户端的数据，直接丢弃。

本环节的目标是掌握以下知识：

- Selector选择器的注册，以及选择器的查询。
- SelectionKey选择键方法的使用。

·根据SelectionKey方法的四种IO事件类型，完成对应的IO处理。

## 1.5.2 第2天：Reactor反应器模式实践

### 实践一：单线程Reactor反应器模式的实现

使用单线程Reactor反应器模式，设计和实现一个EchoServer回显服务器。功能很简单：服务器端读取客户端的输入，然后回显到客户端。

本环节的目标是掌握以下知识：

- 单线程Reactor反应器模式两个重要角色——Reactor反应器、Handler处理器的编写。
- SelectionKey选择键两个重要的方法——attach和attachment方法的使用。

### 实践二：多线程Reactor反应器模式

使用多线程Reactor反应器模式，设计一个EchoServer回显服务器，主要的升级方式为：

- 引入ThreadPool线程池，将负责IOHandler输入输出处理器的执行，放入独立的线程池中，与负责服务监听和IO事件查询的反应器线程相隔离。
- 将反应器线程拆分为多个SubReactor子反应器线程，同时，引入多个Selector选择器，每一个子反应器线程负责一个选择器读取客户端的输入，回显到客户端。

本环节的目标是掌握以下知识：

- 线程池的使用。
- 多线程反应器模式的实现。

### 1.5.3 第3天：异步回调模式实践

#### 实践一：使用线程join方式，通过阻塞式异步调用的方式，实现泡茶喝的实例

为了在计算机中，实现华罗庚的课文《统筹方法》泡茶喝的流程，可以设计三条线程：主线程、清洗线程、烧水线程，主要介绍如下：

- 主线程（MainThread）的工作是：启动清洗线程、启动烧水线程，等清洗、烧水的工作完成后，泡茶喝。

- 清洗线程（WashThread）的工作是：洗茶壶、洗茶杯。

- 烧水线程（HotWarterThread）的工作是：洗好水壶，灌上凉水，放在火上，一直等水烧开。

本环节的目标是掌握以下知识：

不同的版本的join方法的使用。

#### 实践二：使用FutureTask类和Callable接口，启动阻塞式的异步调用，并且获取异步线程的结果。

还是实现华罗庚的课文《统筹方法》泡茶喝的流程，可以设计三条线程：主线程、清洗线程、烧水线程，主要改进如下：

- 主线程（MainThread）的工作是：启动清洗线程、启动烧水线程，然后阻塞，等待异步线程的返回值，根据异步线程的返回值，决定后续的动作。

- 清洗线程（WashThread）在异步执行完成之后，有返回值。

- 烧水线程（HotWarterThread）在异步执行完成之后，有返回值。

本环节的目标是掌握以下知识：

- Callable（可调用）接口的使用； Callable接口和Runnable（可执行）接口的不同。

- FutureTask异步任务类的使用。

#### 实践三：使用ListenableFuture类和FutureCallback接口，启动非阻塞异步调用，并且完成异步回调。

还是实现华罗庚的课文《统筹方法》泡茶喝的流程，可以设计三条线程：主线

程、清洗线程、烧水线程，主要改进如下：

·主线程（MainThread）的工作是：启动清洗线程、启动烧水线程，并且设置异步完成后的回调方法，这里主线程不阻塞等待，而是去干其他事情，例如读报纸。

·清洗线程（WashThread）在异步执行完成之后，执行回调方法。

·烧水线程（HotWarterThread）在异步执行完成之后，执行回调方法。

本环节的目标是掌握以下知识：

·FutureCallback接口的使用；FutureCallback接口和Callable接口的区别和联系。

·ListenableFuture异步任务类的使用，以及为异步任务设置回调方法。

## 1.5.4 第4天：Netty基础实践

实践一：Netty中Handler处理器的生命周期

操作步骤如下：

**步骤01** 定义一个非常简单的入站处理器——InHandlerDemo。这个类继承于ChannelInboundHandlerAdapter适配器，它实现了基类的所有入站处理方法，并在每一个方法的实现中，都加上了必要的输出信息。

**步骤02** 编写一个单元测试代码：将这个处理器加入到一个EmbeddedChannel嵌入式通道的流水线中。

**步骤03** 通过writeInbound方法，向EmbeddedChannel写一个模拟的入站ByteBuf数据包。InHandlerDemo作为一个入站处理器，就会处理到该ByteBuf数据包。

**步骤04** 通过输出，可以观测到处理器的生命周期。

本环节的目标是掌握以下知识：

- Netty中Handler处理器的生命周期。

- EmbeddedChannel嵌入式通道的使用。

### 实践二：ByteBuf的基本使用

操作步骤如下：

**步骤01** 使用Netty的默认分配器，分配了一个初始容量为9个字节，最大上限为100个字节的ByteBuf缓冲区。

**步骤02** 向ByteBuf写数据，观测ByteBuf的属性变化。

**步骤03** 从ByteBuf读数据，观测ByteBuf的属性变化。

本环节的目标是掌握以下知识：

- ByteBuf三个重要属性：readerIndex（读指针）、writerIndex（写指针）、maxCapacity（最大容量）。

- ByteBuf读写过程中，以上三个重要属性的变化规律。

### 实践三：使用Netty，实现EchoServer回显服务器

前面实现过Java NIO版本的EchoServer回显服务器，学习了Netty后，设计和实现一个Netty版本的EchoServer回显服务器。功能很简单：服务器端读取客户端的输入，然后将数据包直接回显到客户端。

本环节的目标是掌握以下知识：

- 服务器端ServerBootstrap的装配和使用。

- 服务器端NettyEchoServerHandler入站处理器的channelRead入站处理方法的编写。

- 服务器端实现Netty的ByteBuf缓冲区的读取、回显。

- 客户端Bootstrap的装配和使用。

- 客户端NettyEchoClientHandler入站处理器中接受回显的数据，并且释放内存。

- 客户端实现多种方式释放ByteBuf，包括：自动释放、手动释放。

## 1.5.5 第5天：解码器（Decoder）与编码器（Encoder）实践

### 实践一：整数解码实践

具体步骤如下：

**步骤01** 定义一个非常简单的整数解码器——Byte2IntegerDecoder。这个类继承于ByteToMessageDecoder字节码解码抽象类，并实现基类的decode抽象方法，将ByteBuf缓冲区中的数据，解码成以一个一个的Integer对象。

**步骤02** 定义一个非常简单的整数处理器——IntegerProcessHandler。读取上一站的入站数据，把它转换成整数，并且显示在Console控制台。

**步骤03** 编写一个整数解码实战的测试用例。在测试用例中，新建了一个EmbeddedChannel嵌入式的通道实例，将两个自己的入站处理器Byte2IntegerDecoder、IntegerProcessHandler加入到通道的流水线上。通过writeInbound方法，向EmbeddedChannel写入一个模拟的入站ByteBuf数据包。

**步骤04** 通过输出，可以观察整数解码器的解码结果。

本环节的目标是掌握以下知识：

- 如何基于Netty的ByteToMessageDecoder字节码解码抽象类，实现自己的ByteBuf二进制字节到POJO对象的解码。

- 使用ByteToMessageDecoder，如何管理ByteBuf的应用计数。

### 实践二：整数相加的解码器实践

具体步骤如下：

**步骤01** 继承ReplayingDecoder基础解码器，编写一个整数相加的解码器：一次解码两个整数，并把这两个数据相加之和，作为解码的结果。

**步骤02** 使用前面定义的整数处理器——IntegerProcessHandler。读取上一站的入站数据，把它转换成整数，并且显示在Console控制台。

**步骤03** 使用前面定义的测试类，测试整数相加的解码器，并且查看结果是否正确。

本环节的目标是掌握以下知识：

- 如何基于ReplayingDecoder解码器抽象类，实现自己的ByteBuf二进制字节到POJO对象的解码。
- ReplayingDecoder的成员属性——state阶段属性的使用。
- ReplayingDecoder的重要方法——checkpoint（IntegerAddDecoder.Status）方法的使用。

### 实践三：基于Head-Content协议的字符串分包解码器

具体步骤如下：

**步骤01** 继承ReplayingDecoder基础解码器，编写一个字符串分包解码器StringReplayDecoder。

在StringReplayDecoder的decode方法中，分两步：第1步，解码出字符串的长度；第2步，按照第一个阶段的字符串长度，解码出字符串的内容。

**步骤02** 编写一个简单的业务处理器StringProcessHandler。其功能是：读取上一站的入站数据，把它转换成字符串，并且显示在Console控制台。

**步骤03** 新建了一个EmbeddedChannel嵌入式的通道实例，将两个自己的入站处理器StringReplayDecoder、StringProcessHandler加入到通道的流水线上。为了测试入站处理器，使用writeInbound方法，向嵌入式通道EmbeddedChannel写入了100个ByteBuf入站缓冲；每一个ByteBuf缓冲，仅仅包含一个字符串。EmbeddedChannel通道接收到入站数据后，pipeline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，然后逐个地显示在Console控制台上。

本环节的目标是掌握以下知识：

- 如何基于ReplayingDecoder解码器抽象类，实现自己的ByteBuf二进制字节到字符串的解码。
- 巩固ReplayingDecoder的成员属性——state阶段属性的使用。
- 巩固ReplayingDecoder的重要方法——checkpoint（IntegerAddDecoder.Status）方法的使用。

### 实践四：多字段Head-Content协议数据包解析实践

具体步骤如下：

**步骤01** 使用LengthFieldBasedFrameDecoder解码器，解码复杂的Head-Content协议。例如协议中包含版本号、魔数等多个其他的数据字段。

**步骤02** 使用前面所编写那一个简单的业务处理器StringProcessHandler。其功能是：读取上一站的入站数据，把它转换成字符串，并且显示在Console控制台上。

**步骤03** 新建一个EmbeddedChannel嵌入式的通道实例，将第一步和第二步的两个入站处理器LengthFieldBasedFrameDecoder、StringProcessHandler加入到通道的流水线上。为了测试入站处理器，使用writeInbound方法，向嵌入式通道EmbeddedChannel写入100个ByteBuf入站缓冲；每一个ByteBuf缓冲，仅仅包含一个字符串。EmbeddedChannel通道接收到入站数据后，pipeline流水线上的两个入站处理器，就能不断地处理到这些入站数据：将接到的二进制字节，解码成一个一个的字符串，然后逐个地显示在控制台上。

本环节的目标是掌握以下知识：

- LengthFieldBasedFrameDecoder解码器的使用。
- LengthFieldBasedFrameDecoder解码器的长度的矫正公式，计算公式为：内容字段的偏移-长度字段的偏移-长度字段的长度。

## 1.5.6 第6天：JSON和ProtoBuf序列化实践

### 实践一：JSON通信实践

客户端将POJO转成JSON字符串，编码后发送到服务器端。服务器端接收客户端的数据包，并解码成JSON，转成POJO。

具体步骤如下：

#### 步骤01 客户端的编码过程：

先通过谷歌的Gson框架，将POJO序列化成JSON字符串；然后使用Netty内置的StringEncoder编码器，将JSON字符串编码成二进制字节数组；最后，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。

#### 步骤02 服务器端的解码过程：

先使用LengthFieldBasedFrameDecoder（Netty内置的自定义长度数据包解码器），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，使用自定义的JsonMsgDecoder解码器，将JSON字符串解码成POJO对象。

步骤03 编写一个JsonMsgDecoder自定义的JSON解码器。将JSON字符串，解码成特定的POJO对象。

步骤04 分别组装好服务器端、客户端的流水线，运行程序，查看两端的通信结果。

本环节的目标是掌握以下知识：

- LengthFieldPrepender编码器的使用：在发送端使用它加上Head-Content的头部长度。

- JsonMsgDecoder的编写。

- JSON传输时，客户端流水线编码器的组装，服务器端流水线解码器的组装。

## 实践二：ProtoBuf通信实践

设计一个简单的客户端/服务器端传输程序：客户端将ProtoBuf的POJO编码成二进制数据包，发送到服务器端；服务器端接收客户端的数据包，并解码成ProtoBuf的POJO。

具体步骤如下：

**步骤01** 设计好需要传输的ProtoBuf的“.proto”协议文件，并且生成ProtoBuf的POJO和Builder：

在“.proto”协议文件中，仅仅定义了一个消息结构体，并且该消息结构体也非常简单，只包含两个字段：消息ID、消息内容。

使用protobuf-maven-plugin插件，生成message的POJO类和Builder（构造者）类的Java代码。

**步骤02** 客户端的编码过程：

先使用Netty内置的ProtobufEncoder，将ProtobufPOJO对象编码成二进制的字节数组。

然后，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，发送到服务器端。

**步骤03** 服务器端的解码过程：

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，可以使用Netty内置的ProtobufDecoder解码器，将Protobuf字节码解码成Protobuf POJO对象。

最后，自定义一个ProtobufBusinessDecoder解码器，处理ProtobufPOJO对象。

本环节的目标是掌握以下知识：

- “.proto”基础协议。

- Netty内置的ProtobufEncoder、ProtobufDecoder两个专用的传输Protobuf序列化

数据的编码器/解码器的使用。

·Netty内置的两个ProtoBuf专用的可变长度Head-Content协议的半包编码、解码处理器：`ProtobufVarint32LengthFieldPrepender`编码器、`ProtobufVarint32FrameDecoder``ProtobufEncoder`解码器的使用。

## 1.5.7 第7~10天：基于Netty的单聊实战

### 实践一：自定义ProtoBuf编码器/解码器

具体步骤如下：

**步骤01** 为单聊系统，设计一套自定义的“.proto”协议文件；然后通过maven插件生成Protobuf Builder和POJO。

**步骤02** 继承Netty提供的MessageToByteEncoder编码器，编写一个自定义的Protobuf编码器，完成Head-Content协议的复杂数据包的编码。

通过自定义编码器，最终将ProtobufPOJO编码成Head-Content协议的二进制ByteBuf帧。

**步骤03** 继承Netty提供的ByteToMessageDecoder解码器，编写一个自定义的Protobuf解码器，完成Head-Content协议的复杂数据包的解码。

通过自定义的解码器，最终将Head-Content协议的复杂数据包，解码出Protobuf POJO。

**步骤04** 分别组装好服务器端、客户端的流水线，运行程序，查看两端的通信结果。

本环节的目标是掌握以下知识：

- 设计复杂的“.proto”协议文件。
- 自定义的Protobuf编码器的编写。
- 自定义的Protobuf解码器的编写。

### 实践二：登录实践

业务逻辑：

- 客户端发送登录数据包。
- 服务器端进行用户信息验证。
- 服务器端创建Session会话。

- 服务器端将登录结果信息返回给客户端，包括成功标志、Session ID等。

具体步骤如下：

从客户端到服务器端再到客户端，大致有以下几个步骤。

**步骤01** 编写LoginConsoleCommand控制台命令类，从客户端收集用户ID和密码。

**步骤02** 编写客户端LoginSender发送器类，组装Protobuf数据包，通过客户端通道发送到服务器端。

**步骤03** 编写服务器端UserLoginHandler入站处理器，处理收到的登录消息，完成数据的校验后，将数据包交给业务处理器LoginMsgProcesser，进行异步的处理。

**步骤04** 编写服务器端LoginMsgProcesser（业务处理器），将处理结果，写入用户绑定的子通道。

**步骤05** 编写客户端业务处理器LoginResponceHandler，处理登录响应，例如设置登录的状态，保存会话的SessionID等。

本环节的目标是掌握以下知识：

- Netty知识的综合运用。

- Channel通道容器功能的使用。

### 实践三：单聊实践

单聊的业务逻辑：

- 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。

- 这里的消息格式设定为——userId:content。其中的userId，就是消息接收方目标用户B的userId；其中的content，表示聊天的内容。

- 服务器端收到消息后，根据目标userId，进行消息帧的转发，发送到用户B所在的客户端。

- 客户端用户B收到用户A发来的消息，显示在自己的控制台上。

具体步骤如下：

从客户端到服务器端再到客户端，大致有5个步骤。

**步骤01** 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。

这里的消息格式设定为——userId:content，其中的userId，就是消息接收方目标用户B的userId，其中的content，表示聊天的内容。

**步骤02** CommandController在完成聊天内容和目标用户的收集后，调用chatSender发送器实例，将聊天消息组装成Protobuf消息帧，通过客户端channel通道发往服务器端。

**步骤03** 编写服务器端的消息转发处理器ChatRedirectHandler类，其功能是，对用户登录进行判断：如果没有登录，则不能发送消息；开启异步的消息转发，由负责转发的chatRedirectProcesser实例，完成消息转发。

**步骤04** 编写负责异步消息转发的ChatRedirectProcesser类，功能如下：根据目标用户ID，找出所有的服务器端的会话列表（Session List），然后对应每一个会话，发送一份消息。

**步骤05** 编写客户端ChatMsgHandler处理器，主要的工作如下：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，直接将消息交给流水线的下一站；如果是聊天消息，则将聊天消息显示在控制台上。

本环节目标是掌握以下知识：

- Netty知识的综合运用。
- 服务器端会话（Session）的管理。

## 1.5.8 第11天：ZooKeeper实践计划

### 实践一：分布式ID生成器

具体步骤如下：

**步骤01** 自定义一个分布式ID生成器——IDMaker类，通过创建ZK的临时顺序节点的方法，生成全局唯一的ID。

**步骤02** 基于自定义的IDMaker，编写单元测试的用例，生成ID。

本环节的目标是掌握以下知识：

- 分布式ID生成器原理。
- ZooKeeper客户端的使用。

### 实践二：使用ZK实现SnowFlake ID算法

具体步骤如下：

**步骤01** 编写一个实现SnowFlake ID算法的ID生成器——SnowflakeIdGenerator类，生成全局唯一的ID。

**步骤02** 基于自定义的SnowflakeIdGenerator，编写单元测试的用例，生成ID。

本环节的目标是掌握以下知识：

- SnowFlake ID算法原理。
- ZooKeeper客户端的使用。

## 1.5.9 第12天：Redis实践计划

### 实践一：使用RedisTemplate模板类完成Redis的缓存CRUD操作

具体步骤如下：

**步骤01** 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，称之为CacheOperationService类。

**步骤02** 编写业务类UserServiceImplWithTemplate类，使用CacheOperationService类，完成User对象的缓存CRUD。

**步骤03** 编写测试用例，访问UserServiceImplWithTemplate类。观察在进行User对象的查询时，能优先使用缓存数据，是否省去数据库访问的时间。

本环节目标是掌握以下知识：

- RedisTemplate模板类的使用。
- Jedis的客户端API。
- RedisTemplate模板API。

### 实践二：使用RedisTemplate模板类完成Redis的缓存CRUD操作

具体步骤如下：

**步骤01** 使用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，完成Redis的操作。

**步骤02** 编写业务类UserServiceImplInTemplate类，使用RedisTemplate模板实例去执行RedisCallback回调实例，完成User对象的缓存CRUD。

**步骤03** 编写测试用例，访问UserServiceImplInTemplate类。观察在进行User对象的查询时，优先使用缓存数据，看看是否省去了数据库访问的时间。

本环节的目标是掌握以下知识：

- RedisCallback回调接口的使用。

- Jedis的客户端API。

- RedisTemplate模板API。

### 实践三：使用Spring缓存注解，完成Redis的缓存CRUD操作

具体步骤如下：

**步骤01** 编写业务类UserServiceImplWithAnn类，使用缓存注解，完成User对象的缓存CRUD。

**步骤02** 编写测试用例，访问UserServiceImplWithAnn类。观察在进行User对象的查询时，优先使用缓存数据，看看是否省去了数据库访问的时间。

本环节目标是掌握以下知识：

- Spring缓存注解的使用。

- Jedis的客户端API。

- Spring缓存注解的配置。

## 1.6 本章小结

本章简单地给大家介绍了高并发时代，以及从业人员必须掌握的Netty、Redis、ZooKeeper等分布式高性能工具。同时，列出了一个大致12天的实践计划。

12天的实践计划不是综合性的大型实践，而是一些掌握单点知识和技能的小型实践案例。只有清楚地掌握了Netty、Redis、ZooKeeper这些工具，才能具备大型开发实践的能力。

在完成了单个功能点的实践案例之后，建议大家加入高性能社群“疯狂创客圈”，进一步参与高并发聊天项目“CrazyIM”的持续迭代，积累真正的实践经验。

## 第2章 高并发IO的底层原理

本书的原则是：从基础讲起。IO的原理和模型是隐藏在编程知识底下的，是开发人员必须掌握的基础原理，是基础的基础，更是通关大公司面试的必备知识。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读写的基础原理

大家知道，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&write两大系统调用。在不同的操作系统中，IO读写的系统调用的名称可能不完全一样，但是基本功能是一样的。

这里涉及一个基础的知识：read系统调用，并不是直接从物理设备把数据读取到内存中；write系统调用，也不是直接把数据写入到物理设备。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，调用操作系统的read，是把数据从内核缓冲区复制到进程缓冲区；而write系统调用，是把数据从进程缓冲区复制到内核缓冲区。

也就是说，上层程序的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&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）之间的交换，这项底层的读写交换，是由操作系统内核（Kernel）来完成的。  
注：本书后文如果没有特别指明，内核即指操作系统内核。

在用户程序中，无论是Socket的IO、还是文件IO操作，都属于上层应用的开发，它们的输入（Input）和输出（Output）的处理，在编程的流程上，都是一致的。

## 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，为什么要那么麻烦呢？缓冲区的目的，是为了减少频繁地与设备之间的物理交换。大家都知道，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，而结束中断之后，还需要恢复之前的进程数据和状态等信息。为了减少这种底层系统的时间损耗、性能损耗，于是出现了内存缓冲区。

有了内存缓冲区，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上层应用的缓冲区（进程缓冲区）；上层应用使用write系统调用时，仅仅把数据从进程缓冲区复制到内核缓冲区中。底层操作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IO设备的中断处理，集中执行物理设备的实际IO操作，这种机制提升了系统的性能。至于什么时候中断（读中断、写中断），由操作系统的内核来决定，用户程序则不需要关心。

从数量上来说，在Linux系统中，操作系统内核只有一个内核缓冲区。而每个用户程序（进程），有自己独立的缓冲区，叫作进程缓冲区。所以，用户的IO读写程序，在大多数情况下，并没有进行实际的IO操作，而是在进程缓冲区和内核缓冲区之间直接进行数据的交换。

## 2.1.2 详解典型的系统调用流程

前面讲到，用户程序所使用的系统调用read&write，它们不等价于数据在内核缓冲区和磁盘之间的交换。read把数据从内核缓冲区复制到进程缓冲区，write把数据从进程缓冲区复制到内核缓冲区，具体的流程，如图2-1所示。

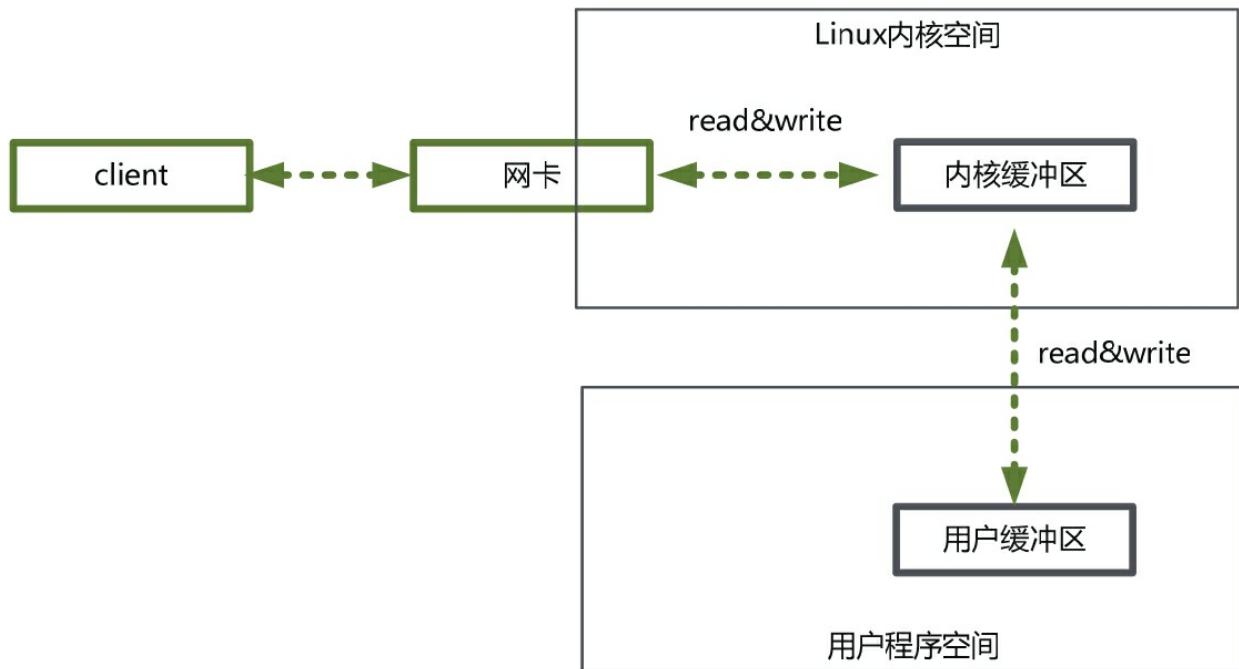


图2-1 系统调用read&write的流程

这里以read系统调用为例，先看下一个完整输入流程的两个阶段：

- 等待数据准备好。
- 从内核向进程复制数据。

如果是read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，等待数据从网络中到达网卡。当所等待的分组到达时，它被复制到内核中的某个缓冲区。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，就是把数据从内核缓冲区复制到应用进程缓冲区。

再具体一点，如果是在Java服务器端，完成一次socket请求和响应，完整的流程如下：

·客户端请求：Linux通过网卡读取客户端的请求数据，将数据读取到内核缓冲

区。

·获取请求数据：Java服务器通过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。

·服务器端业务处理：Java服务器在自己的用户空间中处理客户端的请求。

·服务器端返回数据：Java服务器完成处理后，构建好的响应数据，将这些数据从用户缓冲区写入内核缓冲区。这里用到的是write系统调用。

·发送给客户端：Linux内核通过网络IO，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，会将数据发送给目标客户端。

## 2.2 四种主要的IO模型

服务器端编程，经常需要构造高性能的网络应用，需要选用高性能的IO模型，这也是通关大公司面试必备的知识。

本章从最为基础的模型开始，为大家揭秘IO模型。常见的IO模型有四种：

### 1. 同步阻塞IO（Blocking IO）

首先，解释一下这里的阻塞与非阻塞：

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的操作。阻塞指的是用户空间程序的执行状态。传统的IO模型都是同步阻塞IO。在Java中，默认创建的socket都是阻塞的。

其次，解释一下同步与异步：

同步IO，是一种用户空间与内核空间的IO发起方式。同步IO是指用户空间的线程是主动发起IO请求的一方，内核空间是被动接受方。异步IO则反过来，是指系统内核是主动发起IO请求的一方，用户空间的线程是被动接受方。

### 2. 同步非阻塞IO（Non-blocking IO）

非阻塞IO，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执行用户的操作，即处于非阻塞的状态，与此同时内核会立即返回给用户一个状态值。

简单来说：阻塞是指用户空间（调用线程）一直在等待，而不能干别的事情；非阻塞是指用户空间（调用线程）拿到内核返回的状态值就返回自己的空间，IO操作可以干就干，不可以干，就去干别的事情。

非阻塞IO要求socket被设置为NONBLOCK。

强调一下，这里所说的NIO（同步非阻塞IO）模型，并非Java的NIO（New IO）库。

### 3. IO多路复用（IO Multiplexing）

即经典的Reactor反应器设计模式，有时也称为异步阻塞IO，Java中的Selector选择器和Linux中的epoll都是这种模型。

### 4. 异步IO（Asynchronous IO）

异步IO，指的是用户空间与内核空间的调用方式反过来。用户空间的线程变成被动接受者，而内核空间成了主动调用者。这有点类似于Java中比较典型的回调模式，用户空间的线程向内核空间注册了各种IO事件的回调函数，由内核去主动调用。

## 2.2.1 同步阻塞IO（Blocking IO）

在Java应用程序进程中，默认情况下，所有的socket连接的IO操作都是同步阻塞IO（Blocking IO）。

在阻塞式IO模型中，Java应用程序从IO系统调用开始，直到系统调用返回，在这段时间内，Java进程是阻塞的。返回成功后，应用进程开始处理用户空间的缓存区数据。

同步阻塞IO的具体流程，如图2-2所示。

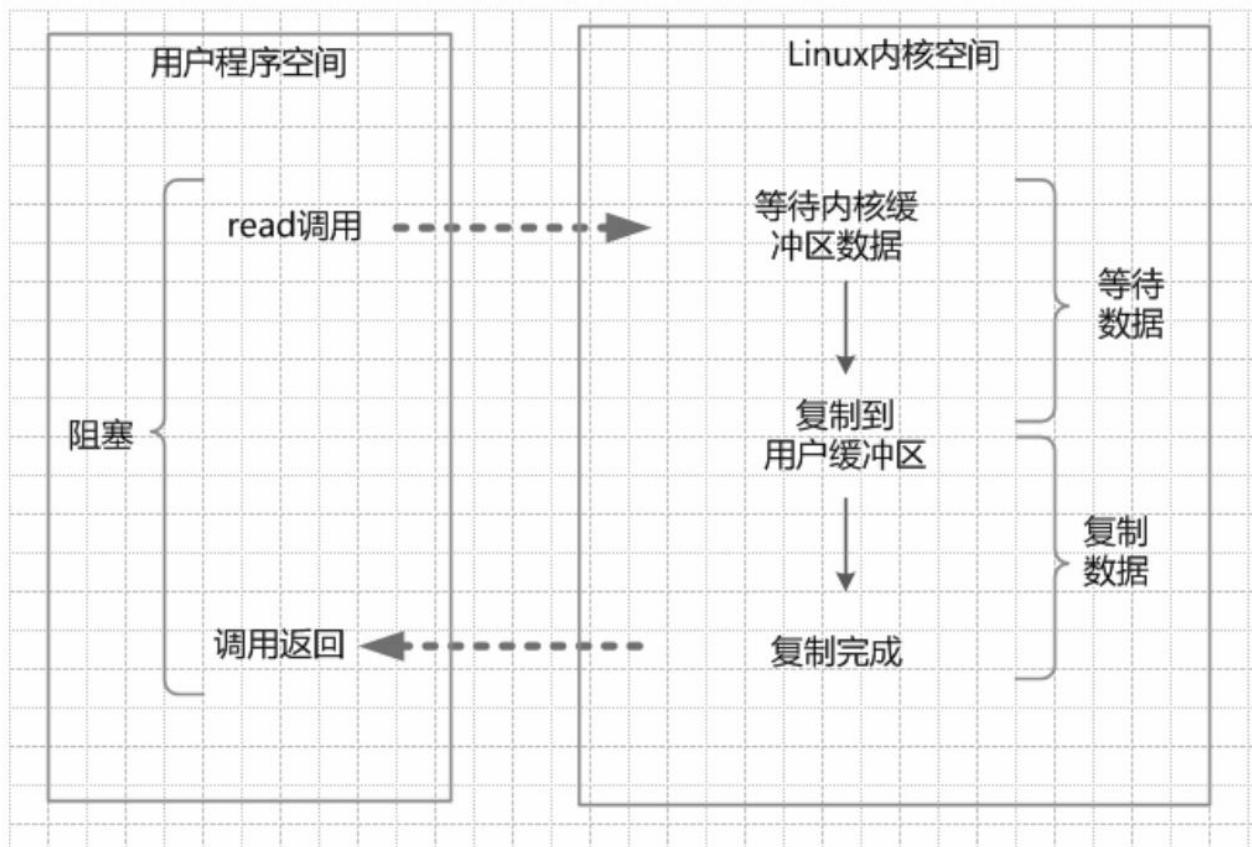


图2-2 同步阻塞IO的流程

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

- (1) 从Java启动IO读的read系统调用开始，用户线程就进入阻塞状态。
- (2) 当系统内核收到read系统调用，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），这个时候内核就要等待。

(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。

总之，阻塞IO的特点是：在内核进行IO执行的两个阶段，用户线程都被阻塞了。

阻塞IO的优点是：应用的程序开发非常简单；在阻塞等待数据期间，用户线程挂起。在阻塞期间，用户线程基本不会占用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；反过来说，就是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。但是，当在高并发的应用场景下，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。因此，基本上阻塞IO模型在高并发应用场景下是不可用的。

## 2.2.2 同步非阻塞NIO（None Blocking IO）

socket连接默认是阻塞模式，在Linux系统下，可以通过设置将socket变成为非阻塞的模式（Non-Blocking）。使用非阻塞模式的IO读写，叫作同步非阻塞IO（None Blocking IO），简称为NIO模式。在NIO模型中，应用程序一旦开始IO系统调用，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，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，是阻塞的，直到数据从内核缓冲复制到用户进程缓冲。复制完成后，系统调用返回成功，应用进程开始处理用户空间的缓存数据。

同步非阻塞IO的流程，如图2-3所示。

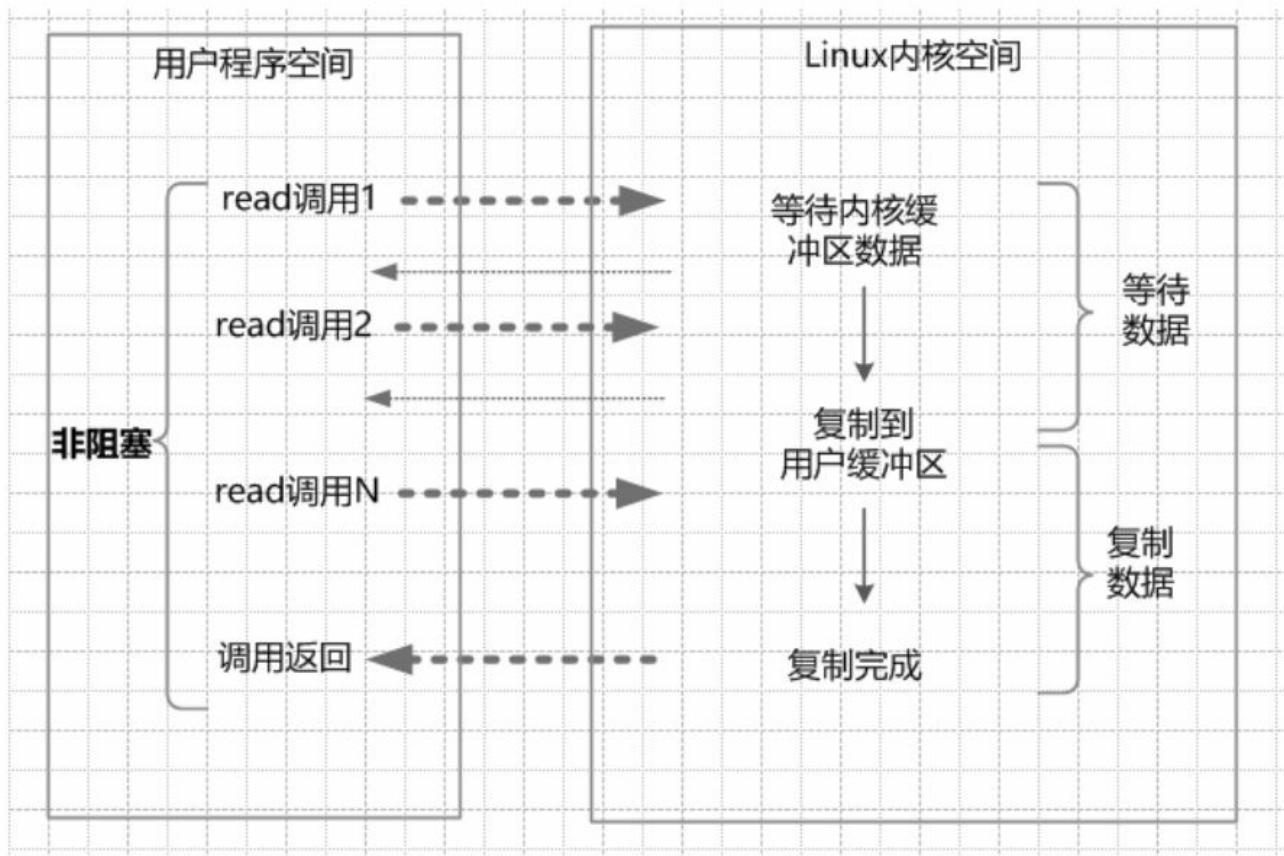


图2-3 同步非阻塞IO的流程

举个例子。发起一个非阻塞socket的read读操作的系统调用，流程如下：

(1) 在内核数据没有准备好的阶段，用户线程发起IO请求时，立即返回。所以，为了读取到最终的数据，用户线程需要不断地发起IO系统调用。

(2) 内核数据到达后，用户线程发起系统调用，用户线程阻塞。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（用户空间的内存），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，用户进程需要经过多次的尝试，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：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：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：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，在高并发应用场景下，同步非阻塞IO也是不可用的。一般Web服务器不使用这种IO模型。这种IO模型一般很少直接使用，而是在其他IO模型中使用非阻塞IO这一特性。在Java的实际开发中，也不会涉及这种IO模型。

这里说明一下，同步非阻塞IO，可以简称为NIO，但是，它不是Java中的NIO，虽然它们的英文缩写一样，希望大家不要混淆。Java的NIO（New IO），对应的不是四种基础IO模型中的NIO（None Blocking IO）模型，而是另外的一种模型，叫作IO多路复用模型（IO Multiplexing）。

## 2.2.3 IO多路复用模型（IO Multiplexing）

如何避免同步非阻塞IO模型中轮询等待的问题呢？这就是IO多路复用模型。

在IO多路复用模型中，引入了一种新的系统调用，查询IO的就绪状态。在Linux系统中，对应的系统调用为select/epoll系统调用。通过该系统调用，一个进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），内核能够将就绪的状态返回给应用程序。随后，应用程序根据就绪的状态，进行相应的IO系统调用。

目前支持IO多路复用的系统调用，有select、epoll等等。select系统调用，几乎在所有的操作系统上都有支持，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，当某个或者某些socket网络连接有IO就绪的状态，就返回对应的可以执行的读写操作。

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

（1）选择器注册。在这种模式中，首先，将需要read操作的目标socket网络连接，提前注册到select/epoll选择器中，Java中对应的选择器类是Selector类。然后，才可以开启整个IO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，查询注册过的所有socket连接的就绪状态。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中的数据准备好了，内核缓冲区有数据（就绪）了，内核就将该socket加入到就绪的列表中。

当用户进程调用了select查询方法，那么整个线程会被阻塞掉。

（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。

IO多路复用模型的流程，如图2-4所示。

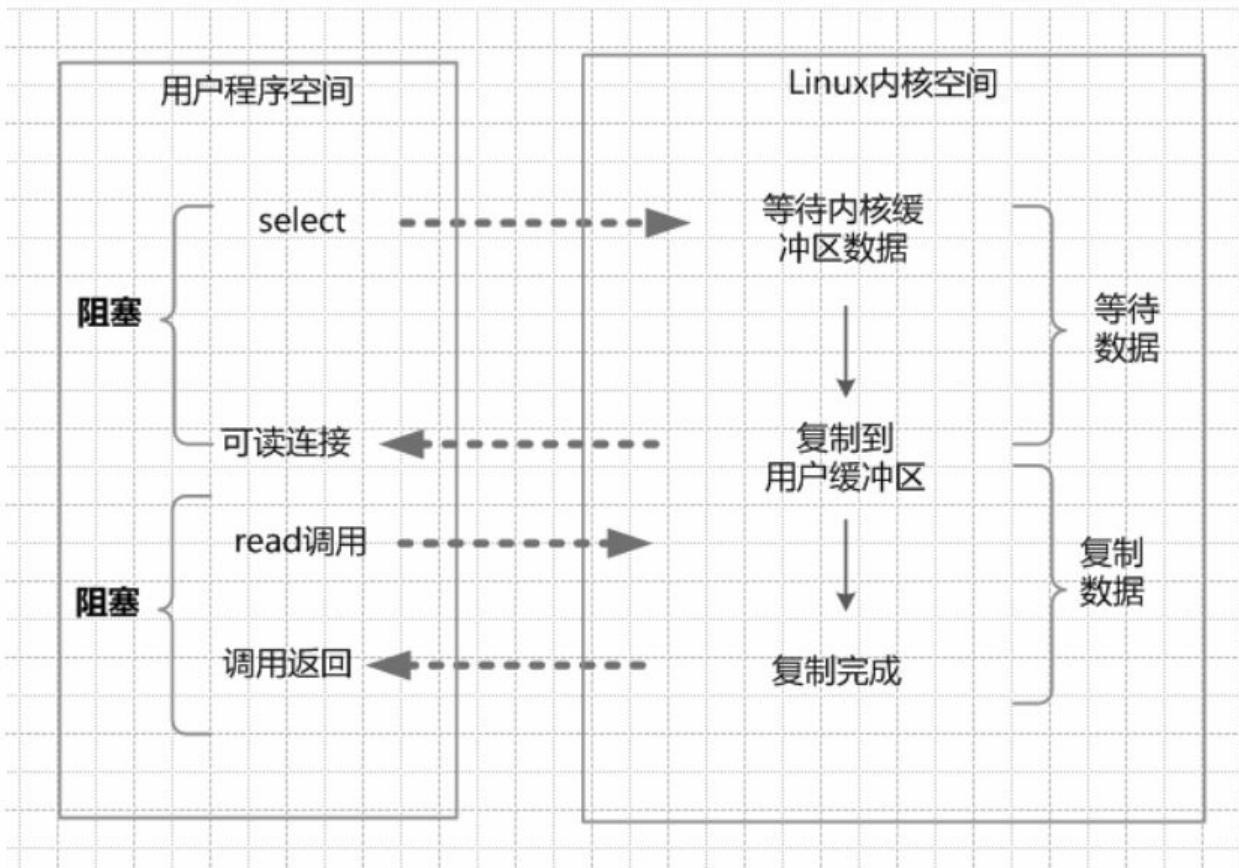


图2-4 IO多路复用模型的流程

IO多路复用模型的特点：IO多路复用模型的IO涉及两种系统调用（System Call），另一种是select epoll（就绪查询），一种是IO操作。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，查找出达到IO操作就绪的socket连接。

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

IO多路复用模型的优点：与一个线程维护一个连接的阻塞IO模式相比，使用select epoll的最大优势在于，一个选择器查询线程可以同时处理成千上万个连接（Connection）。系统不必创建大量的线程，也不必维护这些线程，从而大大减小了系统的开销。

Java语言的NIO（New IO）技术，使用的就是IO多路复用模型。在Linux系统上，使用的是epoll系统调用。

IO多路复用模型的缺点：本质上，select/epoll系统调用是阻塞式的，属于同步IO。都需要在读写事件就绪后，由系统调用本身负责进行读写，也就是说这个读写过程是阻塞的。

如何彻底地解除线程的阻塞，就必须使用异步IO模型。

## 2.2.4 异步IO模型（Asynchronous IO）

异步IO模型（Asynchronous IO，简称为AIO）。AIO的基本流程是：用户线程通过系统调用，向内核注册某个IO操作。内核在整个IO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IO模型中，在整个内核的数据处理过程中，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，用户程序都不需要阻塞。

异步IO模型的流程，如图2-5所示。

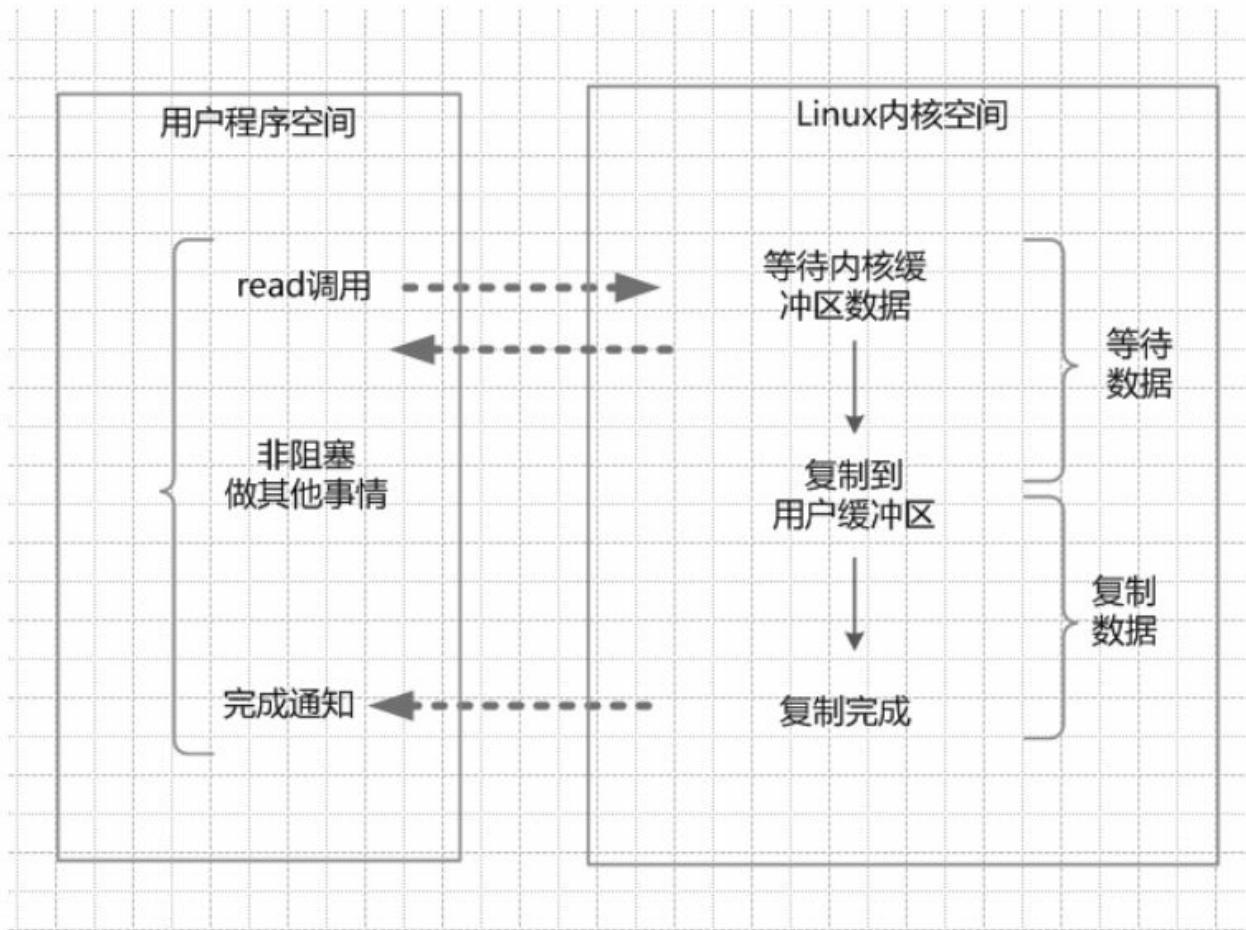


图2-5 异步IO模型的流程

举个例子。发起一个异步IO的read读操作的系统调用，流程如下：

(1) 当用户线程发起了read系统调用，立刻就可以开始去做其他的事，用户线程不阻塞。

(2) 内核就开始了IO的第一个阶段：准备数据。等到数据准备好了，内核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）。

(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接口，告诉用户线程read操作完成了。

(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IO模型的特点：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IO操作完成的事件，或者用户线程需要注册一个IO操作完成的回调函数。正因为如此，异步IO有的时候也被称为信号驱动IO。

异步IO异步模型的缺点：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IO是真正的异步输入输出，它的吞吐量高于IO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IO。而在Linux系统下，异步IO模型在2.6版本才引入，目前并不完善，其底层实现仍使用epoll，与IO多路复用相同，因此在性能上没有明显的优势。

大多数的高并发服务器端的程序，一般都是基于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IO多路复用模型。

大名鼎鼎的Netty框架，使用的就是IO多路复用模型，而不是异步IO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，如果不进行合理的配置，也没有办法支撑百万级的网络连接并发。

这里所涉及的配置，就是Linux操作系统中文件句柄数的限制。

顺便说下，在生产环境中，大家都使用Linux系统，所以，后续文字中假想的生产操作系统，都是Linux系统。另外，由于大多数同学使用Windows进行学习和工作，因此，后续文字中假想的开发所用的操作系统都是Windows系统。

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的系统默认值为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。

本书的原则是：从基础讲起。

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，它是一个非负整数（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，可以看到单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
```

什么是ulimit命令呢？它是用来显示和修改当前用户进程一些基础限制的命令，-n命令选项用于引用或设置当前的文件句柄数量的限制值。Linux的系统默认值为1024。

默认的数值为1024，对绝大多数应用（例如Apache、桌面应用程序）来说已经足够了。但是，是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，就会发出“Socket/File:Can't open so many files”的错误提示。

对于高并发、高负载的应用，就必须要调整这个系统参数，以适应处理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的设置值越大，可以打开的文件句柄数量就越大。建议以root用户来执行此命令。

然而，使用ulimit命令来修改当前用户进程的一些基础限制，仅在当前用户环境有效。直白地说，就是在当前的终端工具连接当前shell期间，修改是有效的；一旦断开连接，用户退出后，它的数值就又变回系统默认的1024了。也就是说，ulimit只能作为临时修改，系统重启后，句柄数量又会恢复为默认值。

如果想永久地把设置值保存下来，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
```

增加-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。硬性极限是实际的限制，就是最大可以是100万，不能再多了。软性极限是系统警告（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，可将软极限更改到硬极限的最大设置值。如果要更改硬极限，必须拥有root用户权限。

终极解除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来解决，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。

在使用和安装目前非常火的分布式搜索引擎——ElasticSearch，就必须去修改这个文件，增加最大的文件句柄数的极限值。

在服务器运行Netty时，也需要去解除文件句柄数量的限制，修改/etc/security/limits.conf文件即可。

## 2.4 本章小结

本书的原则是：从基础讲起。本章彻底体现了这个原则。

本章聚焦的主题：一是底层IO操作的两个阶段，二是最为基础的四种IO模型，三是操作系统对高并发的底层的支持。

四种IO模型，基本上概况了当前主要的IO处理模型，理论上来说，从阻塞IO到异步IO，越往后，阻塞越少，效率也越优。在这四种IO模型中，前三种属于同步IO，因为真正的IO操作都将阻塞应用线程。

只有最后一种异步IO模型，才是真正的异步IO模型，可惜目前Linux操作系统尚欠完善。不过，通过应用层优秀框架如Netty，同样能在IO多路复用模型的基础上，开发出具备支撑高并发（如百万级以上的连接）的服务器端应用。

最后强调一下，本章是理论课，比较抽象，但是一定要懂。理解了这些理论之后，再学习后面的章节就会事半功倍。

## 第3章 Java NIO通信基础详解

高性能的Java通信，绝对离不开Java NIO技术，现在主流的技术框架或中间件服务器，都使用了Java NIO技术，譬如Tomcat、Jetty、Netty。学习和掌握NIO技术，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的“攻城狮”（工程师的谐音），都必须掌握NIO的原理和开发实践技能。

## 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，基于这个原因，更多的人喜欢称Java NIO为非阻塞IO（Non-Block 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，它为标准Java代码提供了高速的、面向缓冲区的IO。

Java NIO由以下三个核心组件组成：

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

如果理解了第1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。当然，Java NIO组件，提供了统一的API，为大家屏蔽了底层的不同操作系统的差异。

后面的章节，我们会对以上的三个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) OIO是面向流（Stream Oriented）的，NIO是面向缓冲区（Buffer Oriented）的。

何谓面向流，何谓面向缓冲区呢？

OIO是面向字节流或字符流的，在一般的OIO操作中，我们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，我们不能随意地改变读取指针的位置。而在NIO操作中则不同，NIO中引入了Channel（通道）和Buffer（缓冲区）的概念。读取和写入，只需要从通道中读取数据到缓冲区中，或将数据从缓冲区中写入到通道中。NIO不像OIO那样是顺序操作，可以随意地读取Buffer中任意位置的数据。

(2) OIO的操作是阻塞的，而NIO的操作是非阻塞的。

NIO如何做到非阻塞的呢？大家都知道，OIO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，那么调用read的线程会被阻塞住，直到read操作完成。

而在NIO的非阻塞模式中，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直接返回，而不会阻塞当前线程。NIO的非阻塞，是如何做到的呢？其实在上一章，答案已经揭晓了，NIO使用了通道和通道的多路复用技术。

(3) OIO没有选择器（Selector）概念，而NIO有选择器的概念。

NIO的实现，是基于底层的选择器的系统调用。NIO的选择器，需要底层操作系统提供支持。而OIO不需要用到选择器。

### 3.1.2 通道（Channel）

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

在NIO中，同一个网络连接使用一个通道表示，所有的NIO的IO操作都是从通道开始的。一个通道类似于OIO中的两个流的结合体，既可以从通道读取，也可以向通道写入。

### 3.1.3 Selector选择器

首先，回顾一个基础的问题，什么是IO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（一个网络连接，操作系统底层使用一个文件描述符来表示），一旦其中的一个或者多个文件描述符可读或者可写，系统内核就通知该进程/线程。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——Selector选择器。

选择器的神奇功能是什么呢？它一个IO事件的查询器。通过选择器，一个线程可以查询多个通道的IO事件的就绪状态。

实现IO多路复用，从具体的开发层面来说，首先把通道注册到选择器中，然后通过选择器内部的机制，可以查询（select）这些注册的通道是否有已经就绪的IO事件（例如可读、可写、网络连接完成等）。

一个选择器只需要一个线程进行监控，换句话说，我们可以很简单地使用一个线程，通过选择器去管理多个通道。这是非常高效的，这种高效来自于Java的选择器组件Selector，以及其背后的操作系统底层的IO多路复用的支持。

与OIO相比，使用选择器的最大优势：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，从而大大减小了系统的开销。

### 3.1.4 缓冲区（Buffer）

应用程序与通道（Channel）主要的交互操作，就是进行数据的read读取和write写入。为了完成如此大任，NIO为大家准备了第三个重要的组件——NIO Buffer（NIO缓冲区）。通道的读取，就是将数据从通道读取到缓冲区中；通道的写入，就是将数据从缓冲区中写入到通道中。

缓冲区的使用，是面向流的OIO所没有的，也是NIO非阻塞的重要前提和基础之一。

下面从缓冲区开始，详细介绍NIO的Buffer（缓冲区）、Channel（通道）、Selector（选择器）三大核心组件。

## 3.2 详解NIO Buffer类及其属性

NIO的Buffer（缓冲区）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。NIO的Buffer类，是一个抽象类，位于java.nio包中，其内部是一个内存块（数组）。

NIO的Buffer与普通的内存块（Java数组）不同的是：NIO Buffer对象，提供了一组更加有效的方法，用来进行写入和读取的交替访问。

需要强调的是：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，分别如下： ByteBuffer、 CharBuffer、 DoubleBuffer、 FloatBuffer、 IntBuffer、 LongBuffer、 ShortBuffer、 MappedByteBuffer。

前7种Buffer类型，覆盖了能在IO中传输的所有的Java基本数据类型。第8种类型 MappedByteBuffer是专门用于内存映射的一种ByteBuffer类型。

实际上，使用最多的还是ByteBuffer二进制字节缓冲区类型，后面会看到。

### 3.2.2 Buffer类的重要属性

Buffer类在其内部，有一个byte[]数组内存块，作为内存缓冲区。为了记录读写的状态和位置，Buffer类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）、limit（读写的限制）。

除此之外，还有一个标记属性：mark（标记），可以将当前的position临时存入mark中；需要的时候，可以再从mark标记恢复到position位置。

#### 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容量，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分配内部的内存。在内存分配好之后，它的大小当然就不能改变了。

再强调一下，capacity容量不是指内存块byte[]数组的字节的数量。capacity容量指的是写入的数据对象的数量。

前面讲到，Buffer类是一个抽象类，Java不能直接用来新建对象。使用的时候，必须使用Buffer的某个子类，例如使用DoubleBuffer，则写入的数据是double类型，如果其capacity是100，那么我们最多可以写入100个double数据。

#### 2.position属性

Buffer类的position属性，表示当前的位置。position属性与缓冲区的读写模式有关。在不同的模式下，position属性的值是不同的。当缓冲区进行读写的模式改变时，position会进行调整。

在写入模式下，position的值变化规则如下：（1）在刚进入到写模式时，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position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的值变化规则如下：（1）当缓冲区刚刚开始进入到读模式时，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）position最大的值为最大可读上限limit，当position达到limit时，表明缓冲区已经无数据可读。

起点在哪里呢？当新建一个缓冲区时，缓冲区处于写入模式，这时是可以写数

据的。数据写入后，如果要从缓冲区读取数据，这就要进行模式的切换，可以使用（即调用）`flip`翻转方法，将缓冲区变成读取模式。

在这个`flip`翻转过程中，`position`会进行非常巨大的调整，具体的规则是：`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。`flip`翻转的另外一半工作，就是要调整`limit`属性。

### 3.limit属性

`Buffer`类的`limit`属性，表示读写的最大上限。`limit`属性，也与缓冲区的读写模式有关。在不同的模式下，`limit`的值的含义是不同的。

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

在读模式下，`limit`的值含义为最多能从缓冲区中读取到多少数据。

一般来说，是先写入再读取。当缓冲区写入完成后，就可以开始从`Buffer`读取数据，可以使用`flip`翻转方法，这时，`limit`的值也会进行非常大的调整。

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，作为可以读取的上限值。

在`flip`翻转时，属性的调整，将涉及`position`、`limit`两个属性，这种调整比较微妙，不是太好理解，举一个简单例子：

首先，创建缓冲区。刚开始，缓冲区处于写模式。`position`为0，`limit`为最大容量。

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。假定写入了5个数，当写入完成后，`position`的值为5。

这时，使用（即调用）`flip`方法，将缓冲区切换到读模式。`limit`的值，先会被设置成写模式时的`position`值。这里新的`limit`是5，表示可以读取的最大上限是5个数。同时，新的`position`会被重置为0，表示可以从0开始读。

### 3.2.3 4个属性的小结

除了前面的3个属性，第4个属性mark（标记）比较简单。就是相当一个暂存属性，暂时保存position的值，方便后面的重复使用position值。

下面用一个表格总结一下Buffer类的4个重要属性，参见表3-1。

表3-1 Buffer四个重要属性的取值说明

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	上限，缓冲区中当前的数据量
position	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()可以让 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

本小节将详细介绍Buffer类使用中常用的几个方法，包含Buffer实例的获取、对Buffer实例的写入、读取、重复读、标记和重置等。

### 3.3.1 allocate()创建缓冲区

在使用Buffer（缓冲区）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。

为了获取一个Buffer实例对象，这里并不是使用子类的构造器new来创建一个实例对象，而是调用子类的allocate()方法。

下面的程序片段就是用来获取一个整型Buffer类的缓冲区实例对象，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    public static void allocatTest()  
    {  
        //调用allocate方法，而不是使用new  
        intBuffer = IntBuffer.allocate(20);  
        //输出buffer的主要属性值  
        Logger.info("-----after allocate-----");  
        Logger.info("position=" + intBuffer.position());  
        Logger.info("limit=" + intBuffer.limit());  
        Logger.info("capacity=" + intBuffer.capacity());  
    }  
    //...  
}
```

例子中，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个字节的内存空间。

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，处于写入的模式，position写入位置为0，最大可写上限limit为的初始化值（这里是20），而缓冲区的容量capacity也是初始化值。

### 3.3.2 put()写入到缓冲区

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。要写入缓冲区，需要调用put方法。put方法很简单，只有一个参数，即为所需要写入的对象。不过，写入的数据类型要求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缓存实例对象中，写入的5个整数，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了创建缓冲区的代码，具体看源代码工程  
    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.info("-----after put-----");  
        Logger.info("position=" + intBuffer.position());  
        Logger.info("limit=" + intBuffer.limit());  
        Logger.info("capacity=" + intBuffer.capacity());  
    }  
    //...  
}
```

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，position变成了5，指向了第6个可以写入的元素位置。而limit最大写入元素的上限、capacity最大容量的值，并没有发生变化。

### 3.3.3 flip()翻转

向缓冲区写入数据之后，是否可以直接从缓冲区中读取数据呢？呵呵，不能。

这时缓冲区还处于写模式，如果需要读取数据，还需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，它的作用就是将写入模式翻转成读取模式。

接着前面的例子，演示一下flip()方法的使用：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的创建、写入的代码，具体看源代码工程  
    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());  
    }  
    //...  
}
```

在调用flip进行模式翻转之后，缓冲区的属性有了奇妙的变化，输出如下：

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

调用flip方法后，之前写入模式下的position值5，变成了可读上限limit值5；而新的读取模式下的position值，简单粗暴地变成了0，表示从头开始读取。

对flip()方法的从写入到读取转换的规则，详细的介绍如下：

首先，设置可读的长度上限limit。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。在读模式下，如果继续使用旧的mark标记，会造成位置混乱。

有关上面的三步，其实可以查看flip方法的源代码，Buffer.flip()方法的源代码如下：

```
public final Buffer flip() {  
    limit = position; //设置可读的长度上限limit,为写入的position  
    position = 0; //把读的起始位置position的值设为0, 表示从头开始读  
    mark = UNSET_MARK; // 清除之前的mark标记  
    return this;  
}
```

至此，大家都知道了，如何将缓冲区切换成读取模式。

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入模式呢？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。

Buffer的模式转换，大致如图3-1所示。

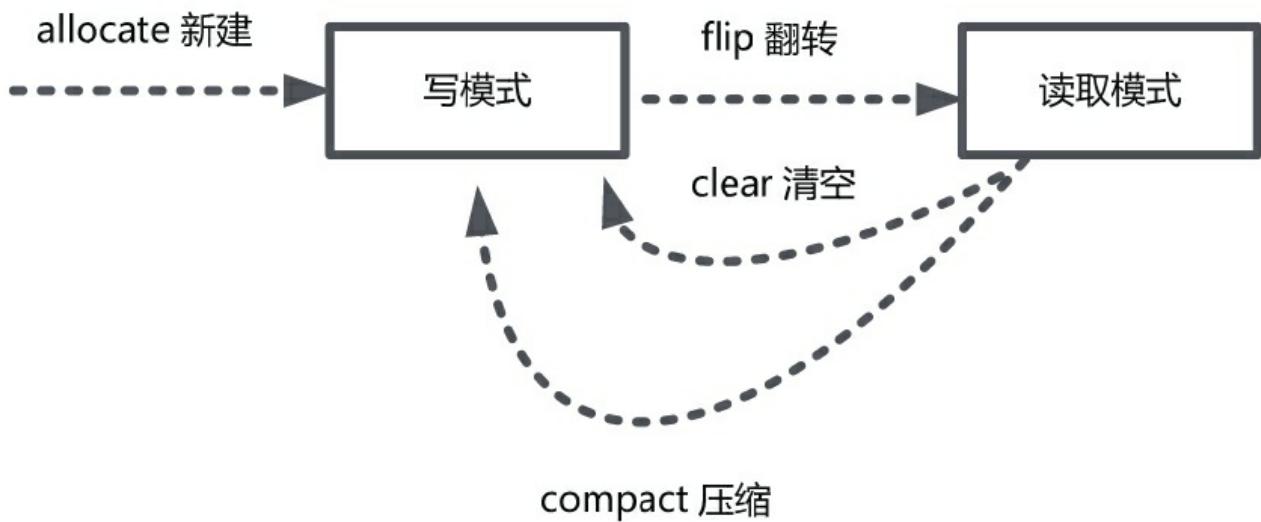


图3-1 缓冲区读写模式的转换

### 3.3.4 get()从缓冲区读取

调用flip方法，将缓冲区切换成读取模式。这时，可以开始从缓冲区中进行数据读取了。读数据很简单，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flip的使用实例，演示一下缓冲区的读取操作，代码如下：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的创建、写入、翻转的代码，具体看源代码工程  
    public static void getTest()  
    {  
        //先读2个  
        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  
        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());  
    }  
    //...  
}
```

先读2个，再读3个，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，我们可以看到，读取操作会改变可读位置position的值，而limit值不会改变。如果position值和limit的值相等，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，会抛出

BufferUnderflowException异常。

这里强调一下，在读完之后，是否可以立即进行写入模式呢？不能。现在还处于读取模式，我们必须调用Buffer.clear()或Buffer.compact()，即清空或者压缩缓冲区，才能变成写入模式，让其重新可写。

另外，还有一个问题：缓冲区是不是可以重复读呢？答案是可以的。

### 3.3.5 rewind()倒带

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。

接着前面的代码，继续rewind方法使用的演示，示例代码如下：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的创建、写入、读取的代码，具体看源代码工程  
    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());  
    }  
    //...  
}
```

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，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多少个元素。
- (3) mark标记被清理，表示之前的临时位置不能再用了。

Buffer.rewind()方法的源代码如下：

```
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标记被清理，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()很相似，区别在于： rewind()不会影响limit属性值；而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;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了缓冲区的读取、倒带的代码，具体看源代码工程  
    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());  
    }  
    //...  
}
```

这段代码，和前面的读取示例代码基本相同，只是增加了一个mark调用。

### 3.3.6 mark( )和reset( )

Buffer.mark()方法的作用是将当前位置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；之后，可以调用Buffer.reset()方法将mark的值恢复到position中。

也就是说，Buffer.mark()和Buffer.reset()方法是配套使用的。两种方法都需要内部mark属性的支持。

在前面重复读取缓冲区的示例代码中，读到第3个元素（ $i==2$ 时），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，就可以调用reset方法，将mark属性的值恢复到position中。然后可以从位置2（第三个元素）开始读。

继续接着前面的重复读取的代码，进行reset的示例演示，代码如下：

```
package com.crazymakercircle.bufferDemo;
//...
public class UseBuffer
{
    static IntBuffer intBuffer = null;
    //省略了缓冲区之前的mark等代码，具体看源代码工程
    public static void afterReset() {
        Logger.info("-----after reset-----");
        //把前面保存在mark中的值恢复到position
        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);
        }
    }
    //...
}
```

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3个元素开始读取数据。上面的程序代码的输出结果是：

```
afterReset |> -----after reset-----
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为2、3、4。

### 3.3.7 clear( )清空缓冲区

在读取模式下，调用clear()方法将缓冲区切换为写入模式。此方法会将position清零，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。代码如下：

```
package com.crazymakercircle.bufferDemo;  
//...  
public class UseBuffer  
{  
    static IntBuffer intBuffer = null;  
    //省略了之前的buffer操作代码，具体看源代码工程  
    public static void clearDemo() {  
        Logger.info("-----after clear-----");  
        //清空缓冲区，进入写入模式  
        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());  
    }  
    //...  
}
```

这个程序运行之后，结果如下：

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

在缓冲区处于读取模式时，调用clear()，缓冲区会被切换成写入模式。调用clear()之后，我们可以看到清空了position的值，即设置写入的起始位置为0，并且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，将数据写入到缓冲区中。
- (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，从缓冲区中读取数据。
- (5) 读取完成后，调用Buffer.clear()或Buffer.compact()方法，将缓冲区转换为写入模式。

## 3.4 详解NIO Channel（通道）类

前面讲到，NIO中一个连接就是用一个Channel（通道）来表示。大家知道，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，远远不止如此，除了可以对应到底层文件描述符，Java NIO的通道还可以更加细化。例如，对应不同的网络传输协议类型，在Java中都有不同的NIO Channel（通道）实现。

### 3.4.1 Channel（通道）的主要类型

这里不对纷繁复杂的Java NIO通道类型进行过多的描述，仅仅聚焦于介绍其中最为重要的四种Channel（通道）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Socket套接字TCP连接的数据读写。
- (3) ServerSocketChannel服务器嵌套字通道（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套接字通道。
- (4) DatagramChannel数据报通道，用于UDP协议的数据读写。

这个四种通道，涵盖了文件IO、TCP网络、UDP IO基础IO。下面从Channel（通道）的获取、读取、写入、关闭四个重要的操作，来对四种通道进行简单的介绍。

### 3.4.2 FileChannel文件通道

FileChannel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，也可以将数据写入到文件中。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、关闭四个操作。

#### 1. 获取FileChannel通道

可以通过文件的输入流、输出流获取FileChannel文件通道，示例如下：

```
//创建一条文件输入流  
FileInputStreamfis = new FileInputStream(srcFile);  
//获取文件流的通道  
FileChannelinChannel = fis.getChannel();  
//创建一条文件输出流  
FileOutputStreamfos = new FileOutputStream(destFile);  
//获取文件流的通道  
FileChanneloutchannel = fos.getChannel();
```

也可以通过RandomAccessFile文件随机访问类，获取FileChannel文件通道：

```
// 创建RandomAccessFile随机访问对象  
RandomAccessFileaFile = new RandomAccessFile("filename.txt", "rw");  
//获取文件流的通道  
FileChannelinChannel = aFile.getChannel();
```

#### 2. 读取FileChannel通道

在大部分应用场景，从通道读取数据都会调用通道的int read（ByteBufferbuf）方法，它从通道读取到数据写入到ByteBuffer缓冲区，并且返回读取到的数据量。

```
RandomAccessFileaFile = new RandomAccessFile(fileName, "rw");  
//获取通道  
FileChannelinChannel=aFile.getChannel();  
//获取一个字节缓冲区  
ByteBufferbuf = 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 = inChannel.read(buf)) != -1) {  
//省略.....处理读取到的buf中的数据  
}
```

#### 注意

虽然对于通道来说是读取数据，但是对于ByteBuffer缓冲区来说是写入数据，

这时， ByteBuffer缓冲区处于写入模式。

### 3. 写入FileChannel通道

写入数据到通道，在大部分应用场景，都会调用通道的int write (ByteBuffer buf) 方法。此方法的参数——ByteBuffer缓冲区，是数据的来源。write方法的作用，是从ByteBuffer缓冲区中读取数据，然后写入到通道自身，而返回值是写入成功的字节数。

```
//如果buf刚写完数据，需要flip翻转buf，使其变成读取模式  
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);  
}
```

#### 注意

此时的ByteBuffer缓冲区要求是可读的，处于读模式下。

### 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，出于性能原因，操作系统不可能每次都实时将数据写入磁盘。如果需要保证写入通道的缓冲数据，最终都真正地写入磁盘，可以调用FileChannel的force()方法。

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

### 3.4.3 使用FileChannel完成文件复制的实践案例

下面是一个简单的实战案例：使用文件通道复制文件。其功能是：使用 FileChannel 文件通道，将原文件复制一份，也就是把原文中的数据都复制到目标文件中。完整代码如下：

```
package com.crazymakercircle.iodeemo.fileDemos;
//...省略import的类，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
        nioCopyResourceFile();
    }
    /**
     * 复制两个资源目录下的文件
     */
    public static void nioCopyResourceFile() {
        String sourcePath = NioDemoConfig.FILE_RESOURCE_SRC_PATH;
        String srcPath = IOUtil.getResourcePath(sourcePath);
        Logger.info("srcPath=" + srcPath);
        String destPath = NioDemoConfig.FILE_RESOURCE_DEST_PATH;
        String destDecodePath = IOUtil.builderResourcePath(destPath);
        Logger.info("destDecodePath=" + destDecodePath);
        nioCopyFile(srcDecodePath, destDecodePath);
    }
    /**
     * nio方式复制文件
     * @param srcPath
     * @param destPath
     */
    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;
            FileChannel outChannel = null;
            try {
                fis = new FileInputStream(srcFile);
                fos = new FileOutputStream(destFile);
                inChannel = fis.getChannel();
                outChannel = fos.getChannel();
                int length = -1;
                ByteBuffer buf = ByteBuffer.allocate(1024);
                //从输入通道读取到buf
                while ((length = inChannel.read(buf)) != -1) {
                    //第一次切换：翻转buf，变成读取模式
                    buf.flip();
                    int outLength = 0;
                    //将buf写入到输出的通道
                    while ((outLength = outChannel.write(buf)) != 0) {
                        System.out.println("写入的字节数：" + outLength);
                    }
                    //第二次切换：清除buf，变成写入模式
                    buf.clear();
                }
                //强制刷新到磁盘
                outChannel.force(true);
            } finally {
                //关闭所有的可关闭对象
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (inChannel != null) {
                try {
                    inChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (outChannel != null) {
                try {
                    outChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
        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();
}
}
```

---

特别强调一下，除了FileChannel的通道操作外，还需要注意ByteBuffer的模式切换。新建的ByteBuffer，默认是写入模式，可以作为inChannel.read(ByteBuffer)的参数。inChannel.read方法将从通道inChannel读到的数据写入到ByteBuffer。

此后，需要调用缓冲区的flip方法，将ByteBuffer切换成读取模式，才能作为outchannel.write(ByteBuffer)方法的参数，从ByteBuffer读取数据，再写入到outchannel输出通道。

如此，便是完成一次复制。在进入下一次复制前，还要进行一次缓冲区的模式切换。ByteBuffer数据读完之后，需要将通过clear方法切换成写入模式，才能进入下一次的复制。

在示例代码中，外层的每一轮while循环，都需要两次模式ByteBuffer切换：第一次切换时，翻转buf，变成读取模式；第二次切换时，清除buf，变成写入模式。

上面的示例代码，主要的目的在于：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，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

### 3.4.4 SocketChannel套接字通道

在NIO中，涉及网络连接的通道有两个，一个是SocketChannel负责连接传输，另一个是ServerSocketChannel负责连接的监听。

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应用于服务器端，而SocketChannel同时处于服务器端和客户端。换句话说，对于一个连接，两端都有一个负责传输的SocketChannel传输通道。

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通道的connect连接、read读、write写操作，都是同步的和阻塞式的，在效率上与Java旧的OIO的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的通道的具体操作。在非阻塞模式下，通道的操作是异步、高效率的，这也是相对于传统的OIO的优势所在。下面详细介绍在非阻塞模式下通道的打开、读写和关闭操作等操作。

#### 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套接字设置为非阻塞模式；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));
```

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，检查当前是否是连接到了主机：

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

```
}
```

在服务器端，如何获取传输套接字呢？

当新连接事件到来时，在服务器端的ServerSocketChannel能成功地查询出一个新连接事件，并且通过调用服务器端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);
```

强调一下，NIO套接字通道，主要用于非阻塞应用场景。所以，需要调用configureBlocking (false)，从阻塞模式设置为非阻塞模式。

## 2. 读取SocketChannel传输通道

当SocketChannel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。

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

在读取时，因为是异步的，因此我们必须检查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。如果返回-1，那么表示读取到对方的输出结束标志，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呢？这就需要用到NIO的新组件——Selector通道选择器，稍后介绍。

## 3. 写入到SocketChannel传输通道

和前面的把数据写入到FileChannel文件通道一样，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是读取模式  
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。

```
//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);
```

### 3.4.5 使用SocketChannel发送文件的实践案例

下面的实践案例是使用FileChannel文件通道读取本地文件内容，然后在客户端使用SocketChannel套接字通道，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");
    /**
     * 向服务器端传输文件
     */
    public void sendFile() throws Exception {
        try {
            String sourcePath = NioDemoConfig.SOCKET_SEND_FILE;
            String srcPath = IOUtil.getResourcePath(sourcePath);
            Logger.info("srcPath=" + srcPath);
            String destFile = NioDemoConfig.SOCKET_RECEIVE_FILE;
            Logger.info("destFile=" + destFile);
            File file = new File(srcPath);
            if (!file.exists()) {
                Logger.info("文件不存在");
                return;
            }
            FileChannelfileChannel = new FileInputStream(file).getChannel();
            SocketChannelsocketChannel = SocketChannel.open();
            socketChannel.socket().connect(
                InetSocketAddress(NioDemoConfig.SOCKET_SERVER_IP,
                    NioDemoConfig.SOCKET_SERVER_PORT));
            socketChannel.configureBlocking(false);
            while(! socketChannel.finishConnect() ){
                //不断地自旋、等待，或者做一些其他的事情
            }
            Logger.info("Client成功连接服务器端");
            //发送文件名称
            ByteBufferfileNameByteBuffer = charset.encode(destFile);
            socketChannel.write(fileNameByteBuffer);
            //发送文件长度
            ByteBuffer buffer = ByteBuffer.allocate
                (NioDemoConfig.SEND_BUFFER_SIZE);
            buffer.putLong(file.length());
            buffer.flip();
            socketChannel.write(buffer);
            buffer.clear();
            //发送文件内容
            Logger.info("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.info("["+ +(100 * progress / file.length()) + "% |");
            }
            if (length == -1) {
                IOUtil.closeQuietly(fileChannel);
                //在SocketChannel传输通道关闭前，尽量发送一个输出结束标志到对端
                socketChannel.shutdownOutput();
                IOUtil.closeQuietly(socketChannel);
            }
            Logger.info("===== 文件传输成功 =====");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
    public static void main(String[] args) {
        NioSendClient client = new NioSendClient(); // 启动客户端连接
        client.sendFile(); // 传输文件
    }
}
```

---

以上代码中的文件发送过程：首先发送目标文件名称（不带路径），然后发送文件长度，最后是发送文件内容。代码中的配置项，如服务器的IP、服务器端口、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，需要先运行服务器端的程序。服务器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。

### 3.4.6 DatagramChannel数据报通道

和Socket套接字的TCP传输协议不同，UDP协议不是面向连接的协议。使用UDP协议时，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中使用UDP协议传输数据，比TCP协议更加简单。在Java NIO中，使用DatagramChannel数据报通道来处理UDP协议的数据传输。

#### 1. 获取DatagramChannel数据报通道

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数据报通道
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. 读取DatagramChannel数据报通道数据

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

```
//创建缓冲区
ByteBufferbuf = ByteBuffer.allocate(1024);
//从DatagramChannel读入，再写入到ByteBuffer缓冲区
SocketAddressclientAddr= datagramChannel.receive(buffer);
```

通道读取receive（ByteBufferbuf）方法的返回值，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，稍后介绍。

#### 3. 写入DatagramChannel数据报通道

向DatagramChannel发送数据，和向SocketChannel通道发送数据的方法也是不同的。这里不是调用write方法，而是调用send方法。示例代码如下：

```
//把缓冲区翻转到读取模式  
buffer.flip();  
//调用send方法，把数据发送到目标IP+端口  
dChannel.send(buffer, new InetSocketAddress(NioDemoConfig.SOCKET_SERVER_IP,  
                                              NioDemoConfig.SOCKET_SERVER_PORT));  
//清空缓冲区，切换到写入模式  
buffer.clear();
```

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

#### 4.关闭DatagramChannel数据报通道

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();
```

### 3.4.7 使用DatagramChannel数据包通道发送数据的实践案例

下面是一个使用DatagramChannel数据包通道到发送数据的客户端示例程序代码。其功能是：获取用户的输入数据，通过DatagramChannel数据报通道，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数据报通道
        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.println("UDP客户端启动成功！");
        Print.println("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数据报通道发送数据
            dChannel.send(buffer, new InetSocketAddress(NioDemoConfig.SOCKET_SERVER_IP, NioDemoConfig.SOCKET_SERVER_PORT));
            buffer.clear();
        }
        //操作四：关闭DatagramChannel数据报通道
        dChannel.close();
    }
    public static void main(String[] args) throws IOException {
        new UDPClient().send();
    }
}
```

通过示例程序代码可以看出，在客户端使DatagramChannel数据报通道发送数据，比起在客户端使用套接字SocketChannel发送数据，简单很多。

接下来看看在服务器端应该如何使用DatagramChannel数据包通道接收数据呢？

下面贴出服务器端通过DatagramChannel数据包通道接收数据的程序代码，可能大家目前不一定可以看懂，因为代码中用到了Selector选择器，但是不要紧，下一个节就介绍它。

服务器端的接收功能是：通过DatagramChannel数据报通道，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。服务器端的完整代码如下：

```
package com.crazymakercircle.iodemo.udpDemos;
//...
public class UDPServer {
    public void receive() throws IOException {
        //获取DatagramChannel数据报通道
        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(NioDemoConfig.SOCKET
    _SERVER_IP, NioDemoConfig.SOCKET_SERVER_PORT));
Print.tcfo("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数据报通道的数据
            SocketAddress client = datagramChannel.receive(buffer);
            buffer.flip();
            Print.tcfo(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();
}
}
```

---

在服务器端，首先调用了bind方法绑定datagramChannel的监听端口。当数据到来后，调用了receive方法，从datagramChannel数据包通道接收数据，再写入到ByteBuffer缓冲区中。

除此之外，在服务器端代码中，为了监控数据的到来，使用了Selector选择器。什么是选择器？如何使用选择器呢？欲知后事如何，请听下节分解。

## 3.5 详解NIO Selector选择器

Java NIO的三大核心组件：Channel（通道）、Buffer（缓冲区）、Selector（选择器）。其中通道和缓冲区，二者的联系也比较密切：数据总是从通道读到缓冲区内，或者从缓冲区写入到通道中。

至此，前面两个组件已经介绍完毕，下面迎来了最后一个非常重要的角色——选择器（Selector）。

### 3.5.1 选择器以及注册

选择器（Selector）是什么呢？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IO的多路复用。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IO（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）所监控的通道拥有哪些已经准备好的、就绪的IO操作事件。

一般来说，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。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系，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

事件类型的定义在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事件呢？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，而是通道的某个IO操作的一种就绪状态，表示通道具备完成某个IO操作的条件。

比方说，某个SocketChannel通道，完成了和对端的握手连接，则处于“连接就

绪”（OP\_CONNECT）状态。

再比方说，某个ServerSocketChannel服务器通道，监听到一个新连接的到来，则处于“接收就绪”（OP\_ACCEPT）状态。

还比方说，一个有数据可读的SocketChannel通道，处于“读就绪”（OP\_READ）状态；一个等待写入数据的，处于“写就绪”（OP\_WRITE）状态。

### 3.5.2 SelectableChannel可选择通道

并不是所有的通道，都是可以被选择器监控或选择的。比方说，`FileChannel`文件通道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）。如果继承了`SelectableChannel`，则可以被选择，否则不能。

简单地说，一条通道若能被选择，必须继承`SelectableChannel`类。

`SelectableChannel`类，是何方神圣呢？它提供了实现通道的可选择性所需要的公共方法。Java NIO中所有网络链接Socket套接字通道，都继承了`SelectableChannel`类，都是可选择的。而`FileChannel`文件通道，并没有继承`SelectableChannel`，因此不是可选择通道。

### 3.5.3 SelectionKey选择键

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，和调用选择器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，返回注册过的感兴趣的那些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。

这里出现一个新的概念——SelectionKey选择键。SelectionKey选择键是什么呢？简单地说，SelectionKey选择键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选择键集合中；如果之前没有注册过，即使发生了IO事件，也不会被选择器选中。SelectionKey选择键和IO的关系，可以简单地理解为：选择键，就是被选中了的IO事件。

在编程时，选择键的功能是很强大的。通过SelectionKey选择键，不仅仅可以获得通道的IO事件类型，比方说SelectionKey.OP\_READ；还可以获得发生IO事件所在的通道；另外，也可以获得选出选择键的选择器实例。

### 3.5.4 选择器使用流程

使用选择器，主要有以下三步：

(1) 获取选择器实例； (2) 将通道注册到选择器中； (3) 轮询感兴趣的IO就绪事件（选择键集合）。

#### 第一步：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选择器的类方法open()的内部，是向选择器SPI（SelectorProvider）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SPI全称为（Service Provider Interface，服务提供者接口），是JDK的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，其他的提供商可以通过SPI的方式，提供定制化版本的选择器的动态替换或者扩展。

#### 第二步：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：

```
// 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(SystemConfig.SOCKET_SERVER_PORT));  
// 5.将通道注册到选择器上，并制定监听事件为：“接收连接”事件  
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

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

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文件通道不能与选择器一起使用，因为FileChannel文件通道只有阻塞模式，不能切换到非阻塞模式；而Socket套接字相关的所有通道都可以。

其次，还需要注意：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，仅仅支持Accept（接收到新连接）IO事件；而SocketChannel传输通道，则不支持Accept（接收到新连接）IO事件。

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

### 第三步：选出感兴趣的IO就绪事件（选择键集合）

通过Selector选择器的select()方法，选出已经注册的、已经就绪的IO事件，保存到SelectionKey选择键集合中。SelectionKey集合保存在选择器实例内部，是一个元素为SelectionKey类型的集合（Set）。调用选择器的selectedKeys()方法，可以取得选择键集合。

接下来，需要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，需要将选择键从这个SelectionKey集合中移除，防止下一次循环的时候，被重复的处理。SelectionKey集合不能添加元素，如果试图向SelectionKey选择键集合中添加元素，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，一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()方法返回的整数值（int整数类型），表示发生了IO事件的通道数量。更

准确地说，是从上一次select到这一次select之间，有多少通道发生了IO事件。强调一下，`select()`方法返回的数量，指的是通道数，而不是IO事件数，准确地说，是指发生了选择器感兴趣的IO事件的通道数。

### 3.5.5 使用NIO实现Discard服务器的实践案例

Discard服务器的功能很简单：仅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；并且读取到的数据直接抛弃掉（Discard）。Discard服务器足够简单明了，作为第一个学习NIO的通信实例，较有参考价值。

下面的Discard服务器代码，将选择器使用流程中的步骤进行了细化：

```
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(NioDemoConfig
                .SOCKET_SERVER_PORT));
        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”事件，就获取客户端连接
                    SocketChannel socketChannel = serverSocketChannel.accept();
                    // 11. 切换为非阻塞模式
                    socketChannel.configureBlocking(false);
                    // 12. 将该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”事件，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一共分为16步，其中第7到第15步是循环执行的。不断选择感兴趣的IO事件到选择器的选择键集合中，然后通过selector.selectedKeys()获取该选择键集合，并且进行迭代处理。对于新建立的socketChannel客户端传输通道，也要注册到同一个选择器上，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选择。

在DiscardServer程序中，涉及到两次选择器注册：一次是注册serverChannel服务器通道；另一次，注册接收到的socketChannel客户端传输通道。serverChannel服务器通道注册的，是新连接的IO事件SelectionKey.OP\_ACCEPT；客户端socketChannel传输通道注册的，是可读IO事件SelectionKey.OP\_READ。

DiscardServer在对选择键进行处理时，通过对类型进行判断，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serverChannel服务器通道发生了新连接事件，则通过服务器通道的accept方法，获取新的socketChannel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socketChannel传输通道的数据，然后丢弃。

客户端的DiscardClient代码，则更为简单。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。代码如下：

---

```
package com.crazymakercircle.iodemo.NioDiscard;
//...
public class NioDiscardClient {
    public static void startClient() throws IOException {
        InetSocketAddress address = new InetSocketAddress(NioDemoConfig.
            SOCKET_SERVER_IP, NioDemoConfig.SOCKET_SERVER_PORT);
        // 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();
    }
}
```

---

如果需要执行整个程序，首先要执行前面的服务器端程序，然后执行后面的客户端程序。

通过Discard服务器的开发实践，大家对NIO Selector（选择）的使用流程，应该了解得非常清楚了。

下面来看一个稍微复杂一点的案例：在服务器端接收文件和内容。

### 3.5.6 使用SocketChannel在服务器端接收文件的实践案例

本示例演示文件的接收，是服务器端的程序。和前面介绍的文件发送的SocketChannel客户端程序是相互配合使用的。由于在服务器端，需要用到选择器，所以在介绍完选择器后，才开始介绍NIO文件传输的Socket服务器端程序。服务器端接收文件的示例代码如下所示：

```
package com.crazymakercircle.iodemo.socketDemos;
//...
public class NioReceiveServer {
    private Charset charset = Charset.forName("UTF-8");
    /**
     * 内部类，服务器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;
    }
    private ByteBuffer buffer
        = ByteBuffer.allocate(NioDemoConfig.SERVER_BUFFER_SIZE);
    //使用Map保存每个文件传输，当OP_READ可读时，根据通道找到对应的对象
    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(NioDemoConfig.SOCKET_SERVER_PORT);
        serverSocket.bind(address);
        // 5.将通道注册到选择器上，并注册的IO事件为：“接收新连接”
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        Print.tcfo("serverChannel is listening...");
        // 6.选择感兴趣的IO就绪事件（选择键集合）
        while (selector.select() > 0) {
            // 7.获取选择键集合
            Iterator<SelectionKey> 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.若接受的事件是“新连接”事件，就获取客户端新连接
                    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选择器上
                    SelectionKey selectionKey =
                        socketChannel.register(selector, SelectionKey.OP_READ);
                    // 为每一条传输通道，建立一个Client客户端对象，放入map，供后面使用
                }
            }
        }
    }
}
```

```

        Client client = new Client();
        client.remoteAddress
            = (InetSocketAddress) socketChannel.getRemoteAddress();
        clientMap.put(socketChannel, client);
        Logger.info(socketChannel.getRemoteAddress() + "连接成功...");;
    } else if (key.isReadable()) {
        // 13.若接收的事件是“数据可读”事件,就读取客户端新连接
        processData(key);
    }
    }
}
/***
 * 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();
            if (null == client.fileName) {
                //客户端发送过来的, 首先是文件名
                //根据文件名, 创建服务器端的文件, 将文件通道保存到客户端
                String fileName = charset.decode(buffer).toString();
                String destPath = IOUtil.getResourcePath(
                    NioDemoConfig.SOCKET_RECEIVE_PATH);
                File directory = new File(destPath);
                if (!directory.exists()) {
                    directory.mkdir();
                }
                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);
                FileChannel fileChannel
                    = new FileOutputStream(file).getChannel();
                client.outChannel = fileChannel;
            } else if (0 == client.fileLength) {
                //客户端发送过来的, 其次是文件长度
                long fileLength = buffer.getLong();
                client.fileLength = fileLength;
                client.startTime = System.currentTimeMillis();
                Logger.info("NIO 传输开始: ");
            } else {
                //客户端发送过来的, 最后是文件内容, 写入文件内容
                client.outChannel.write(buffer);
            }
            buffer.clear();
        }
        key.cancel();
    } catch (IOException e) {
        key.cancel();
        e.printStackTrace();
        return;
    }
    // 读取数量-1, 表示客户端传输结束标志到了
    if (num == -1) {
        IOUtil.closeQuietly(client.outChannel);
        System.out.println("上传完毕");
        key.cancel();
        Logger.info("文件接收成功, File Name: " + client.fileName);
        Logger.info(" Size: " +
            IOUtil.getFormatFileSize(client.fileLength));
        long endTime = System.currentTimeMillis();
        Logger.info("NIO IO传输毫秒数: " + (endTime - client.startTime));
    }
}
}

```

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

---

由于客户端每次传输文件，都会分为多次传输：

- (1) 首先传入文件名称。
- (2) 其次是文件大小。
- (3) 然后是文件内容。

对应于每一个客户端socketChannel，创建一个Client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：建立连接的时候，以socketChannel作为键（Key），Client对象作为值（Value），将Client保存在map中。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socketChannel通道。然后通过socketChannel通道，从map中取到对应的Client对象。

接收到数据时，如果文件名为空，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；接下来再读到数据，说明接收到了文件大小，把文件大小保存到Client对象；接下来再接到数据，说明是文件内容了，则写入Client对象的outChannel文件通道中，直到数据读取完毕。

运行方式：启动这个NioReceiveServer服务器程序后，再启动前面介绍的客户端程序NioSendClient，即可以完成文件的传输。

## 3.6 本章小结

在编程难度上，Java NIO编程的难度比同步阻塞Java OIO编程大很多。请注意，前面的实践案例，是比较简单的，并不是复杂的通信程序，没有看到“粘包”和“拆包”等问题。如果加上这些问题，代码将会更加复杂。

与Java OIO相比，Java NIO编程大致的特点如下：

- (1) 在NIO中，服务器接收新连接的工作，是异步进行的。不像Java的OIO那样，服务器监听连接，是同步的、阻塞的。NIO可以通过选择器（也可以说成：多路复用器），后续不断地轮询选择器的选择键集合，选择新到来的连接。
- (2) 在NIO中，SocketChannel传输通道的读写操作都是异步的。如果没有可读写的数据，负责IO通信的线程不会同步等待。这样，线程就可以处理其他连接的通道；不需要像OIO那样，线程一直阻塞，等待所负责的连接可用为止。
- (3) 在NIO中，一个选择器线程可以同时处理成千上万个客户端连接，性能不会随着客户端的增加而线性下降。

总之，有了Linux底层的epoll支持，有了Java NIO Selector选择器这样的应用层IO复用技术，Java程序从而可以实现IO通信的高TPS、高并发，使服务器具备并发数十万、数百万的连接能力。Java的NIO技术非常适合用于高性能、高负载的网络服务器。鼎鼎大名的通信服务器中间件Netty，就是基于Java的NIO技术实现的。

当然，Java NIO技术仅仅是基础，如果要实现通信的高性能和高并发，还离不开高效率的设计模式。下一章将开始为大家介绍高性能服务必备的设计模式：Reactor反应器模式。

## 第4章 鼎鼎大名的Reactor反应器模式

本书的原则：从基础讲起。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。为什么呢？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，才能轻松地学习和掌握Netty框架。同时，反应器模式也是BAT级别大公司必不可少的面试题。

## 4.1 Reactor反应器模式为何如此重要

在详细介绍什么是Reactor反应器模式之前，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反应器模式。很多著名的服务器软件或者中间件都是基于反应器模式实现的。

比如说，“全宇宙最有名的、最高性能”的Web服务器Nginx，就是基于反应器模式的；如雷贯耳的Redis，作为最高性能的缓存服务器之一，也是基于反应器模式的；目前火得“一塌糊涂”、在开源项目中应用极为广泛的高性能通信中间件Netty，更是基于反应器模式的。

从开发的角度来说，如果要完成和胜任高性能的服务器开发，反应器模式是必须学会和掌握的。从学习的角度来说，反应器模式相当于高性能、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，反应器模式相关的问题是经常出现的面试问题。

总之，反应器模式是高性能网络编程的必知、必会的模式。

### 4.1.1 为什么首先学习Reactor反应器模式

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反应器模式呢？

写多了代码的程序员都知道，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了代码的组织方式。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、多态的设计模式。

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，往往会晕头转向，不知身在何处，很难读懂别人的代码，对代码跟踪很成问题。反过来，如果先了解代码的设计模式，再去看代码，就会阅读得很轻松，不会那么难懂了。

当然，在写代码时，不了解设计模式，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Netty。Netty本身很抽象，大量应用了设计模式。学习像Netty这样的“精品中的精品”，肯定也是需要先从设计模式入手的。而Netty的整体架构，就是基于这个著名反应器模式。

总之，反应器模式非常重要。首先学习和掌握反应器模式，对于学习Netty的人来说，一定是磨刀不误砍柴工。

## 4.1.2 Reactor反应器模式简介

什么是Reactor反应器模式呢？本文站在巨人的肩膀上，引用一下Doug Lea（那是一位让人无限景仰的大师，Java中Concurrent并发包的重要作者之一）在文章《Scalable IO in Java》中对反应器模式的定义，具体如下：

反应器模式由Reactor反应器线程、Handlers处理器两大角色组成：

- (1) Reactor反应器线程的职责：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。

在这里，为了方便大家学习，将Doug Lea著名的文章《Scalable IO in Java》的链接地址贴出来：<http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf>，建议大家去阅读一下，提升自己的基础知识，开阔下眼界。

从上面的反应器模式定义，看不出这种模式有什么神奇的地方。当然，从简单到复杂，反应器模式也有很多版本。根据前面的定义，仅仅是最为简单的一个版本。

如果需要彻底了解反应器模式，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，最初和最原始的网络服务器程序，是用一个while循环，不断地监听端口是否有新的连接。如果有，那么就调用一个处理函数来处理，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，那么后面的连接请求没法被接收，于是后面的请求通通会被阻塞住，服务器的吞吐量就太低了。对于服务器来说，这是一个严重的问题。

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：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 */ }  
    }  
    //处理器对象  
    static class Handler implements Runnable {  
        final Socket socket;  
        Handler(Socket s) {  
            socket = s;  
        }  
        public void run() {  
            while (true) {  
                try {  
                    byte[] input = new byte[NioDemoConfig.SERVER_BUFFER_SIZE];  
                    /* 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 */ }  
            }  
        }  
    }  
}
```

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输入和输出。当然，服务器的监听线程也是独立的，任何的socket连接的输入和输出处理，不会阻塞到后面新socket连接的监听和建立。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，极大地提高了服务器的吞吐量。

这里有个问题：如果一个线程同时负责处理多个socket连接的输入和输出，行不行呢？

看上去，没有什么不可以。但是，实际上没有用。为什么？传统OIO编程中每一个socket的IO读写操作，都是阻塞的。在同一时刻，一个线程里只能处理一个socket，前一个socket被阻塞了，后面连接的IO操作是无法被并发执行的。所以，不论怎么处理，OIO中一个线程也只能是处理一个连接的IO操作。

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。如果线程数太多，系统无法承受。而且，线程的反复创建、销毁、线程的切换也需要代价。因此，在高并发的应用场景下，多线程OIO的缺陷是致命的。

如何解决Connection Per Thread模式的巨大缺陷呢？一个有效路径是：使用Reactor反应器模式。用反应器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。它是如何做到呢？首先来看简单的版本——单线程的Reactor反应器模式。

## 4.2 单线程Reactor反应器模式

总体来说，Reactor反应器模式有点儿类似事件驱动模式。

在事件驱动模式中，当有事件触发时，事件源会将事件dispatch分发到handler处理器进行事件处理。反应器模式中的反应器角色，类似于事件驱动模式中的dispatcher事件分发器角色。

前面已经提到，在反应器模式中，有Reactor反应器和Handler处理器两个重要的组件：

(1) Reactor反应器：负责查询IO事件，当检测到一个IO事件，将其发送给相应的Handler处理器去处理。这里的IO事件，就是NIO中选择器监控的通道IO事件。

(2) Handler处理器：与IO事件（或者选择键）绑定，负责IO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、负责将结果写出到通道等。

## 4.2.1 什么是单线程Reactor反应器

什么是单线程版本的Reactor反应器模式呢？简单地说，Reactor反应器和Handlers处理器处于一个线程中执行。它是最简单的反应器模型，如图4-1所示。

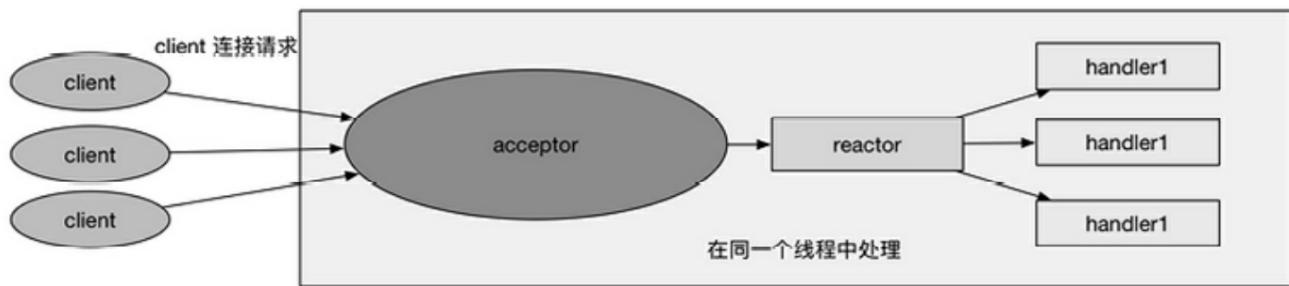


图4-1 单线程Reactor反应器模式

基于Java NIO，如何实现简单的单线程版本的反应器模式呢？需要用到SelectionKey选择键的几个重要的成员方法：

方法一：void attach(Object o)

此方法可以将任何的Java POJO对象，作为附件添加到SelectionKey实例，相当于附件属性的setter方法。这方法非常重要，因为在单线程版本的反应器模式中，需要将Handler处理器实例，作为附件添加到SelectionKey实例。

方法二：Object attachment()

此方法的作用是取出之前通过attach(Object o)添加到SelectionKey选择键实例的附件，相当于附件属性的getter方法，与attach(Object o)配套使用。

这个方法同样非常重要，当IO事件发生，选择键被select方法选到，可以直接将事件的附件取出，也就是之前绑定的Handler处理器实例，通过该Handler，完成相应的处理。

总之，在反应器模式中，需要进行attach和attachment结合使用：在选择键注册完成之后，调用attach方法，将Handler处理器绑定到选择键；当事件发生时，调用attachment方法，可以从选择键取出Handler处理器，将事件分发到Handler处理器中，完成业务处理。

## 4.2.2 单线程Reactor反应器的参考代码

Doug Lea在《Scalable IO in Java》中，实现了一个单线程Reactor反应器模式的参考代码。这里，我们站在巨人的肩膀上，借鉴Doug Lea的实现，对其进行介绍。为了方便说明，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;
//...
class Reactor implements Runnable {
    Selector selector;
    ServerSocketChannel serverSocket;
    EchoServerReactor() throws IOException {
        //....省略：打开选择器、serverSocket连接监听通道
        //注册serverSocket的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());
    }
    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(); }
    }
    //反应器的分发方法
    void dispatch(SelectionKey k) {
        Runnable handler = (Runnable) (k.attachment());
        //调用之前绑定到选择键的handler处理器对象
        if (handler != null) {
            handler.run();
        }
    }
    // 新连接处理器
    class AcceptorHandler implements Runnable {
        public void run() {
            //接受新连接
            //需要为新连接，创建一个输入输出的handler处理器
        }
    }
    //...
}
```

在上面的代码中，设计了一个Handler处理器，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，被设置（attach）到了SelectionKey中。

```
//注册serverSocket的接受(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, 取出了之前attach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();  
    }  
}
```

---

AcceptorHandler处理器的两大职责: 一是接受新连接, 二是在为新连接创建一个输入输出的Handler处理器, 称之为IOHandler。

---

```
// 新连接处理器  
class AcceptorHandler implements Runnable {  
    public void run() {  
        // 接受新连接  
        // 需要为新连接创建一个输入输出的handler处理器  
    }  
}
```

---

IOHandler, 顾名思义, 就是负责socket的数据输入、业务处理、结果输出。示例代码如下:

---

```
package com.crazymakercircle.ReactorModel;  
//...  
class IOHandler implements Runnable {  
    final SocketChannel channel;  
    final SelectionKey sk;  
    IOHandler (Selector selector, SocketChannel c) throws IOException {  
        channel = c;  
        c.configureBlocking(false);  
        //仅仅取得选择键, 稍候设置感兴趣的IO事件  
        sk = channel.register(selector, 0);  
        //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, 注册到了反应器Reactor类的同一个选择器中。这样保证了Reactor类和Handler类在同一个线程中执行。

(2) Channel传输通道注册完成后，将IOHandler自身作为附件，attach到了选择键中。这样，在Reactor类分发事件（选择键）时，能执行到IOHandler的run方法。

如果上面的示例代码比较绕口，不要紧。为了彻底地理解个中妙处，自己动手开发一个可以执行的实例。下面基于反应器模式，实现了一个EchoServer回显服务器实例。仔细阅读和运行这个实例，就可以明白上面这段绕口的程序代码的真正含义了。

### 4.2.3 一个Reactor反应器版本的EchoServer实践案例

EchoServer回显服务器的功能很简单：读取客户端的输入，回显到客户端，所以也叫回显服务器。基于Reactor反应器模式来实现，设计3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;
//.....
//反应器
class EchoServerReactor implements Runnable {
    Selector selector;
    ServerSocketChannel serverSocket;
    EchoServerReactor() throws IOException {
        //...获取选择器、开启serverSocket服务监听通道
        //...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();
        }
    }
    void dispatch(SelectionKey sk) {
        Runnable handler = (Runnable) sk.attachment();
        //调用之前attach绑定到选择键的handler处理器对象
        if (handler != null) {
            handler.run();
        }
    }
    // Handler: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();
    }
}
```

```
    }  
}
```

EchoHandler回显处理器，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);  
    static final int RECEIVING = 0, SENDING = 1;  
    int state = RECEIVING;  
    EchoHandler(Selector selector, SocketChannel c) throws IOException {  
        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  
        sk.attach(this);  
        //注册Read就绪事件  
        sk.interestOps(SelectionKey.OP_READ);  
        selector.wakeup();  
    }  
    public void run() {  
        try {  
            if (state == SENDING) {  
                //写入通道  
                channel.write(byteBuffer);  
                //写完后,准备开始从通道读,byteBuffer切换成写入模式  
                byteBuffer.clear();  
                //写完后,注册read就绪事件  
                sk.interestOps(SelectionKey.OP_READ);  
                //写完后,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,准备开始写入通道,byteBuffer切换成读取模式  
                byteBuffer.flip();  
                //读完后,注册write就绪事件  
                sk.interestOps(SelectionKey.OP_WRITE);  
                //读完后,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();  
        }  
    }  
}
```

以上两个类，是一个基于反应器模式的EchoServer回显服务器的完整实现。它是一个单线程版本的反应器模式，Reactor反应器和所有的Handler处理器，都执行在同一条线程中。

运行EchoServerReactor类中的main方法，可以启动回显服务器。如果要看到回显输出，还需要启动客户端。客户端的代码，在同一个包下，类名为EchoClient，负责数据的发送。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反应器模式的缺点

单线程Reactor反应器模式，是基于Java的NIO实现的。相对于传统的多线程OIO，反应器模式不再需要启动成千上万条线程，效率自然是大大提升了。

在单线程反应器模式中，Reactor反应器和Handler处理器，都执行在同一条线程上。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，如果被阻塞的Handler不仅仅负责输入和输出处理的业务，还包括负责连接监听的AcceptorHandler处理器。这个是非常严重的问题。

为什么？一旦AcceptorHandler处理器阻塞，会导致整个服务不能接收新的连接，使得服务器变得不可用。因为这个缺陷，因此单线程反应器模型用得比较少。

另外，目前的服务器都是多核的，单线程反应器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，单线程反应器模式实际使用的很少。

## 4.3 多线程的Reactor反应器模式

既然Reactor反应器和Handler处理器，挤在一个线程会造成非常严重的性能缺陷。那么，可以使用多线程，对基础的反应器模式进行改造和演进。

### 4.3.1 多线程池Reactor反应器演进

多线程池Reactor反应器的演进，分为两个方面：

(1) 首先是升级Handler处理器。既要使用多线程，又要尽可能的高效率，则可以考虑使用线程池。

(2) 其次是升级Reactor反应器。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，多线程池反应器的模式，大致如下：

(1) 将负责输入输出处理的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服务监听和IO事件查询的反应器线程相隔离，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，每一个SubReactor子线程负责一个选择器。这样，充分释放了系统资源的能力；也提高了反应器管理大量连接，提升选择大量通道的能力。

### 4.3.2 多线程Reactor反应器的实践案例

在前面的“回显服务器”（EchoServer）的基础上，完成多线程Reactor反应器的升级。多线程反应器的实践案例设计如下：

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

为了提升效率，建议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，引起的效率降低。这个实践案例的代码如下：

```
package com.crazymakercircle.ReactorModel;
//....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();
        selectors[1] = Selector.open();
        serverSocket = ServerSocketChannel.open();
        InetSocketAddress address =
            new InetSocketAddress(NioDemoConfig.SOCKET_SERVER_IP,
        NioDemoConfig.SOCKET_SERVER_PORT);
        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], SelectionKey.OP_ACCEPT);
        //绑定Handler: attach新连接监控handler处理器到SelectionKey(选择键)
        sk.attach(new AcceptorHandler());
        //第一个子反应器,一子反应器负责一个选择器
        SubReactor subReactor1 = new SubReactor(selectors[0]);
        //第二个子反应器,一子反应器负责一个选择器
        SubReactor subReactor2 = new SubReactor(selectors[1]);
        subReactors = new SubReactor[]{subReactor1, subReactor2};
    }
    private void startService() {
        // 一子反应器对应一个线程
        new Thread(subReactors[0]).start();
        new Thread(subReactors[1]).start();
    }
    //子反应器
    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;
        }
        @Override
        public void run() {
            while(true) {
                selector.select();
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectedKeys.iterator();
                while(iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    if(key.isAcceptable()) {
                        accept(key);
                    } else if(key.isReadable()) {
                        read(key);
                    }
                    iterator.remove();
                }
            }
        }
        void accept(SelectionKey key) {
            try {
                SocketChannel channel = ((ServerSocketChannel)key.channel()).accept();
                channel.configureBlocking(false);
                channel.register(selector, SelectionKey.OP_READ);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        void read(SelectionKey key) {
            try {
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                channel.read(buffer);
                String message = new String(buffer.array());
                System.out.println("收到消息：" + message);
                buffer.clear();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                selector.select();
                Set<SelectionKey> keySet = selector.selectedKeys();
                Iterator<SelectionKey> it = keySet.iterator();
                while (it.hasNext()) {
                    //反应器负责dispatch收到的事件
                    SelectionKey sk = it.next();
                    dispatch(sk);
                }
                keySet.clear();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    void dispatch(SelectionKey sk) {
        Runnable handler = (Runnable) sk.attachment();
        //调用之前attach绑定到选择键的handler处理器对象
        if (handler != null) {
            handler.run();
        }
    }
}
// Handler:新连接处理器
class AcceptorHandler implements Runnable {
    public void run() {
        try {
            SocketChannel channel = serverSocket.accept();
            if (channel != null)
                new MultiThreadEchoHandler(selectors[next.get()], channel);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (next.incrementAndGet() == selectors.length)
            next.set(0);
    }
}
public static void main(String[] args) throws IOException {
    MultiThreadEchoServerReactor server =
        new MultiThreadEchoServerReactor();
    server.startService();
}
}

```

---

上面是反应器的演进代码，再来看看Handler处理器的多线程演进实践。

### 4.3.3 多线程Handler处理器的实践案例

基于前面的单线程反应器的EchoHandler回显处理器的程序代码，予以改进，新的回显处理器为：MultiThreadEchoHandler。主要的升级是引入了一个线程池(ThreadPool)，业务处理的代码执行在自己的线程池中，彻底地做到业务处理线程和反应器IO事件线程的完全隔离。这个实践案例的代码如下：

```
package com.crazymakercircle.ReactorModel;
//...
class MultiThreadEchoHandler implements Runnable {
    final SocketChannel channel;
    final SelectionKey sk;
    final ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
    static final int RECEIVING = 0, SENDING = 1;
    int state = RECEIVING;
    //引入线程池
    static ExecutorService pool = Executors.newFixedThreadPool(4);
    MultiThreadEchoHandler(Selector selector, SocketChannel c) throws IOException {
        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);
        //将本Handler作为sk选择键的附件，方便事件分发(dispatch)
        sk.attach(this);
        //向sk选择键注册Read就绪事件
        sk.interestOps(SelectionKey.OP_READ);
        selector.wakeup();
    }
    public void run() {
        //异步任务，在独立的线程池中执行
        pool.execute(new AsyncTask());
    }
    //业务处理，不在反应器线程中执行
    public synchronized void asyncRun() {
        try {
            if (state == SENDING) {
                //写入通道
                channel.write(byteBuffer);
                //写完后，准备开始从通道读，byteBuffer切换成写入模式
                byteBuffer.clear();
                //写完后，注册read就绪事件
                sk.interestOps(SelectionKey.OP_READ);
                //写完后，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，准备开始写入通道，byteBuffer切换成读取模式
                byteBuffer.flip();
                //读完后，注册write就绪事件
                sk.interestOps(SelectionKey.OP_WRITE);
                //读完后，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();
        }
    }
    //异步任务的内部类
    class AsyncTask implements Runnable {
```

```
    public void run() {
        MultiThreadEchoHandler.this.asyncRun();
    }
}
```

---

代码中设计了一个内部类**AsyncTask**，是一个简单的异步任务的提交类。它使得异步业务**asyncRun**方法，可以独立地提交到线程池中。另外，既然业务处理异步执行，需要在**asyncRun**方法的前面加上**synchronized**同步修饰符。

至此，多线程版本的反应器模式，实践案例的代码就演示完了。执行新版本的多线程**MultiThreadEchoServerReactor**服务器，可以使用之前的**EchoClient**客户端与之配置，完成整个回显（echo）的通信演示。

演示的输出和之前单线程版本的**EchoServer**回显服务器示例，是一模一样的。

## 4.4 Reactor反应器模式小结

在总结反应器模式前，首先看看和其他模式的对比，加强一下对它的理解。

### 1.反应器模式和生产者消费者模式对比

相似之处：在一定程度上，反应器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到一个队列中，一个或多个消费者主动地从这个队列中提取（Pull）事件来处理。

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

### 2.反应器模式和观察者模式（Observer Pattern）对比

相似之处在于：在反应器模式中，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，同步地分发这些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。

不同之处在于：在反应器模式中，Handler处理器实例和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处理器；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处理器；而在观察者模式中，同一个时刻，同一个主题可以被订阅过的多个观察者处理。

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

- 响应快，虽然同一反应器线程本身是同步的，但不会被单个连接的同步IO所阻塞；

- 编程相对简单，最大程度避免了复杂的多线程同步，也避免了多线程的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。

反应器模式的缺点如下：

- 反应器模式增加了一定的复杂性，因而有一定的门槛，并且不易于调试。

·反应器模式需要操作系统底层的IO多路复用的支持，如Linux中的epoll。如果操作系统的底层不支持IO多路复用，反应器模式不会有那么高效。

·同一个Handler业务线程中，如果出现一个长时间的数据读写，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（Client）的响应时间。因而对于这种操作，还需要进一步对反应器模式进行改进。

## 4.5 本章小结

反应器（Reactor）模式是高性能网络编程在设计和架构层面的基础模式。同时，反应器模式，也是BAT级别大公司必不可少的面试题。

本章首先从最简单的Connection Per Thread（一个线程处理一个连接）模式入手，介绍了该模式的严重缺陷，从而引出来了单线程的反应器模式。

为了充分利用系统资源，最大限度地减少阻塞，在单线程的反应器模式的基础上，又演进出来了多线程的反应器模式实现。

本章的反应器模式的实现，仅仅是抛砖引玉，在充分利用系统资源、最大限度地减少阻塞两个维度，都有很大的提升空间，建议大家自行尝试。

## 第5章 并发基础中的Future异步回调模式

随着业务模块系统越来越多，各个系统的业务架构变得越来越错综复杂，特别是这几年微服务架构的兴起，跨设备跨服务的接口调用越来越频繁。打个简单的比方：现在的一个业务流程，可能需要调用N次第三方接口，获取N种上游数据。因此，面临一个大的问题是：如何高效率地异步去调取这些接口，然后同步去处理这些接口的返回结果呢？这里涉及线程的异步回调问题，这也是高并发的一个基础问题。

在Netty源代码中，大量地使用了异步回调技术，并且基于Java的异步回调，设计了自己的一整套异步回调接口和实现。

在本章中，我们从Java Future异步回调技术入手，然后介绍比较常用第三方异步回调技术——谷歌公司的Guava Future相关技术，最后介绍一下Netty的异步回调技术。

总之，学习高并发编程，掌握异步回调技术是编程人员必须具备的一项基础技术。

## 5.1 从泡茶的案例说起

在进入异步回调模式的正式解读之前，先来看一个比较好理解的异步生活示例。笔者尼恩就想到自己中学8年级的语文中，有一篇华罗庚的课文——《统筹方法》，其中举了一个合理安排工序、以便提升效率的泡茶案例。下面分别是用阻塞模式和异步回调模式来实现其中的异步泡茶流程。强调一下：这里直接略过顺序执行的冒泡工序，那个效率太低了。

为了异步执行整个泡茶流程，分别设计三条线程：主线程、清洗线程、烧水线程。

(1) 主线程（MainThread）的工作是：启动清洗线程、启动烧水线程，等清洗、烧水的工作完成后，泡茶喝。

(2) 清洗线程（WashThread）的工作是：洗茶壶、洗茶杯。

(3) 烧水线程（HotWarterThread）的工作是：洗好水壶，灌上凉水，放在火上，一直等水烧开。

下面分别使用阻塞模式、异步回调模式来实现泡茶喝的案例。

## 5.2 join异步阻塞

阻塞模式实现泡茶实例，首先从最为基础的多线程join合并开始。join操作的原理是：阻塞当前的线程，直到准备合并的目标线程的执行完成。

### 5.2.1 线程的join合并流程

在Java中，线程（Thread）的合并流程是：假设线程A调用了线程B的B.join方法，合并B线程。那么，线程A进入阻塞状态，直到B线程执行完成。

在泡茶喝的例子中，主线程通过分别调用烧水线程和清洗线程的join方法，等待烧水线程和清洗线程执行完成，然后执行自己的泡茶操作。具体如图5-1所示。

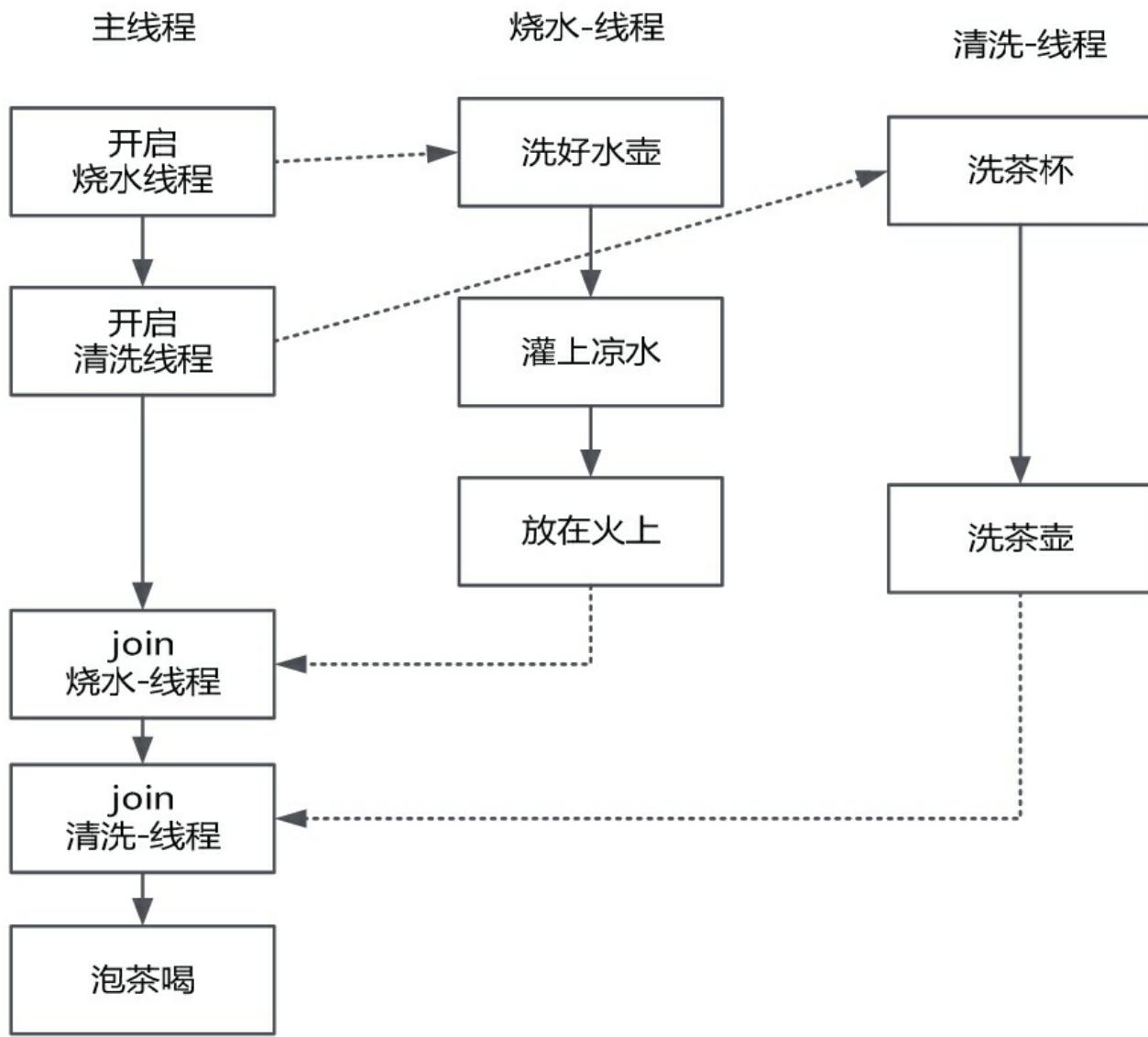


图5-1 使用join实现泡茶实例的流程

## 5.2.2 使用join实现异步泡茶喝的实践案例

使用join实现泡茶喝，这是一个异步阻塞版本，具体的代码实现如下：

```
package com.crazymakercircle.coccurent;
//...
public class JoinDemo {
    public static final int SLEEP_GAP = 500;
    public static String getCurThreadName() {
        return Thread.currentThread().getName();
    }
    static class HotWarterThread extends Thread {
        public HotWarterThread() {
            super("** 烧水-Thread");
        }
        public void run() {
            try {
                Logger.info("洗好水壶");
                Logger.info("灌上凉水");
                Logger.info("放在火上");
                //线程睡眠一段时间，代表烧水中
                Thread.sleep(SLEEP_GAP);
                Logger.info("水开了");
            } catch (InterruptedException e) {
                Logger.info("发生异常被中断.");
            }
            Logger.info("运行结束.");
        }
    }
    static class WashThread extends Thread {
        public WashThread() {
            super("$$ 清洗-Thread");
        }
        public void run() {
            try {
                Logger.info("洗茶壶");
                Logger.info("洗茶杯");
                Logger.info("拿茶叶");
                //线程睡眠一段时间，代表清洗中
                Thread.sleep(SLEEP_GAP);
                Logger.info("洗完了");
            } catch (InterruptedException e) {
                Logger.info("发生异常被中断.");
            }
            Logger.info("运行结束.");
        }
    }
    public static void main(String args[]) {
        Thread hThread = new HotWarterThread();
        Thread wThread = new WashThread();
        hThread.start();
        wThread.start();
        try {
            // 合并烧水-线程
            hThread.join();
            // 合并清洗-线程
            wThread.join();
            Thread.currentThread().setName("主线程");
            Logger.info("泡茶喝");
        } catch (InterruptedException e) {
            Logger.info(getCurThreadName() + "发生异常被中断.");
        }
        Logger.info(getCurThreadName() + "运行结束.");
    }
}
```

程序中有三个线程：（1）主线程main；（2）烧水线程hThread；（3）清洗线程wThread。main主线程调用了hThread.join()实例方法，合并烧水线程，也调用了wThread.join()实例方法，合并清洗线程。

说明一下：hThread、wThread是线程实例，在示例代码中，hThread对应的线程名称为“\*\* 烧水-Thread”，wThread对应的线程名称为“\$\$ 清洗-Thread”。

### 5.2.3 详解join合并方法

join方法的应用场景：A线程调用B线程的join方法，等待B线程执行完成；在B线程没有完成前，A线程阻塞。

join方法是有三个重载版本：

(1) void join(): A线程等待B线程执行结束后，A线程重新恢复执行。

(2) void join(long millis): A线程等待B线程执行一段时间，最长等待时间为millis毫秒。超过millis毫秒后，不论B线程是否结束，A线程重新恢复执行。

(3) void join(long millis, int nanos): 等待B线程执行一段时间，最长等待时间为millis毫秒，加nanos纳秒。超过时间后，不论B线程是否结束，A线程重新恢复执行。

强调一下容易混淆的几点：

(1) join是实例方法，不是静态方法，需要使用线程对象去调用，如thread.join()。

(2) join调用时，不是线程所指向的目标线程阻塞，而是当前线程阻塞。

(3) 只有等到当前线程所指向的线程执行完成，或者超时，当前线程才能重新恢复执行。

join有一个问题：被合并的线程没有返回值。例如，在烧水的实例中，如果烧水线程的执行结束，main线程是无法知道结果的。同样，清洗线程的执行结果，main线程也是无法知道的。形象地说，join线程合并就是一像一个闷葫芦。只能发起合并线程，不能取到执行结果。

如果需要获得异步线程的执行结果，怎么办呢？可以使用Java的FutureTask系列类。

## 5.3 FutureTask异步回调之重武器

为了获取异步线程的返回结果，Java在1.5版本之后提供了一种新的多线程的创建方式——FutureTask方式。FutureTask方式包含了一系列的Java相关的类，在java.util.concurrent包中。其中最为重要的是FutureTask类和Callable接口。

### 5.3.1 Callable接口

在介绍Callable接口之前，先得重提一下旧的Runnable接口。Runnable接口是在Java多线程中表示线程的业务代码的抽象接口。但是，Runnable有一个重要的问题，它的run方法是没有返回值的。正因为如此，Runnable不能用于需要有返回值的应用场景。

为了解决Runnable接口的问题，Java定义了一个新的和Runnable类似的接口——Callable接口。并且将其中的代表业务处理的方法命名为call，call方法有返回值。

Callable的代码如下：

```
package java.util.concurrent;
@FunctionalInterface
public interface Callable<V> {
    //call方法有返回值
    V call() throws Exception;
}
```

Callable接口是一个泛型接口，也声明为了“函数式接口”。其唯一的抽象方法call有返回值，返回值的类型为泛型形参的实际类型。call抽象方法还有一个Exception的异常声明，容许方法内部的异常不经过捕获。

总之，Callable接口可以对应到Runnable接口；Callable接口的call方法可以对应到Runnable接口的run方法。相比较而言，Callable接口的功能更强大一些。

Callable接口与Runnable接口相比，还有一个很大的不同：Callable接口的实例不能作为Thread线程实例的target来使用；而Runnable接口实例可以作为Thread线程实例的target构造参数，开启一个Thread线程。

问题来了，Java中的线程类型，只有一个Thread类，没有其他的类型。如果Callable实例需要异步执行，就要想办法赋值给Thread的target成员，一个Runnable类型的成员。为此，Java提供了在Callable实例和Thread的target成员之间一个搭桥的类——FutureTask类。

### 5.3.2 初探FutureTask类

顾名思义，FutureTask类代表一个未来执行的任务，表示新线程所执行的操作。FutureTask类也位于java.util.concurrent包中。FutureTask类的构造函数的参数为Callable类型，实际上是对Callable类型的二次封装，可以执行Callable的call方法。FutureTask类间接地继承了Runnable接口，从而可以作为Thread实例的target执行目标。

FutureTask类的构造函数的源代码，如下：

```
public FutureTask(Callable<V> callable) {  
    if (callable == null)  
        throw new NullPointerException();  
    this.callable = callable;  
    this.state = NEW;          // ensure visibility of callable  
}
```

FutureTask类就像一座搭在Callable实例与Thread线程实例之间的桥。FutureTask类的内部封装一个Callable实例，然后自身又作为Thread线程的target。

在外部，如何要获取Callable实例的异步执行结果，不是调用其call方法，而是需要通过FutureTask类的相应方法去获取。

总体来说，FutureTask类首先是一个搭桥类的角色，FutureTask类能当作Thread线程去执行目标target，被异步执行；其次，如果要获取异步执行的结果，需要通过FutureTask类的方法去获取，在FutureTask类的内部，会将Callable的call方法的真正结果保存起来，以供外部获取。

在Java语言中，将FutureTask类的一系列操作，抽象出来作为一个重要的接口——Future接口。当然，FutureTask类也实现了此接口。

### 5.3.3 Future接口

Future接口不复杂，主要是对并发任务的执行及获取其结果的一些操作。主要提供了3大功能：

- (1) 判断并发任务是否执行完成。
- (2) 获取并发的任务完成后的结果。
- (3) 取消并发执行中的任务。

Future接口的源代码如下：

```
package java.util.concurrent;
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit) throws InterruptedException,
        ExecutionException, TimeoutException;
}
```

关于Future接口的方法，详细说明如下：

**V get():** 获取并发任务执行的结果。注意，这个方法是阻塞性的。如果并发任务没有执行完成，调用此方法的线程会一直阻塞，直到并发任务执行完成。

**V get(Long timeout, TimeUnit unit):** 获取并发任务执行的结果。也是阻塞性的，但是会有阻塞的时间限制，如果阻塞时间超过设定的timeout时间，该方法将抛出异常。

**boolean isDone():** 获取并发任务的执行状态。如果任务执行结束，则返回true。

**boolean isCancelled():** 获取并发任务的取消状态。如果任务完成前被取消，则返回true。

**boolean cancel(boolean mayInterruptIfRunning):** 取消并发任务的执行。

### 5.3.4 再探FutureTask类

FutureTask类实现了Future接口，提供了外部操作异步任务的能力。现在，回到FutureTask类自身。为了完成异步执行Callable类型的任务、获取任务结果的使命，在FutureTask类的内部，又有哪些成员和方法呢？

首先，FutureTask内部有一个Callable类型的成员，代表异步执行的逻辑：

```
private Callable<V> callable;
```

callable实例属性必须要在FutureTask类的实例构造时进行初始化。

其次，FutureTask内部有一个run方法。这个run方法是Runnable接口的抽象方法，在FutureTask类的内部提供了自己的实现。在Thread线程实例执行时，会将这个run方法作为target目标去异步执行。在FutureTask内部的run实现代码中，会执行其callable成员的call方法。执行完成后，将结果保存起来。保存在哪里呢？

再次，FutureTask内部有另一个重要的成员——outcome属性，用于保存结果：

```
private Object outcome;
```

outcome属性所保存的结果，可供FutureTask类的结果获取方法（如get）来获取。

至此，这个很重要的FutureTask搭桥类就介绍完了。如果还不是很清楚，也不要緊，相信通过实践一个FutureTask版本的喝茶示例，就明白了。

### 5.3.5 使用FutureTask类实现异步泡茶喝的实践案例

在前面的join版本喝茶示例中，有一个很大的问题：就是主线程获取不到异步线程的返回值。打个比方，如果烧水线程出了问题，或者清洗线程出了问题，主线程是没办法知道的。哪怕不具备泡茶条件，主线程也只能继续泡茶喝。

使用FutureTask类和Callable接口，进行异步结果的获取，代码如下：

```
package com.crazymakercircle.coccurent;
//...
public class JavaFutureDemo {
    public static final int SLEEP_GAP = 500;
    public static String getCurThreadName() {
        return Thread.currentThread().getName();
    }
    static class HotWarterJob implements Callable<Boolean> //①
    {
        @Override
        public Boolean call() throws Exception //②
        {
            try {
                Logger.info("洗好水壶");
                Logger.info("灌上凉水");
                Logger.info("放在火上");
                //线程睡眠一段时间，代表烧水中
                Thread.sleep(SLEEP_GAP);
                Logger.info("水开了");
            } catch (InterruptedException e) {
                Logger.info("发生异常被中断.");
                return false;
            }
            Logger.info("运行结束.");
            return true;
        }
    }
    static class WashJob implements Callable<Boolean> {
        @Override
        public Boolean call() throws Exception {
            try {
                Logger.info("洗茶壶");
                Logger.info("洗茶杯");
                Logger.info("拿茶叶");
                //线程睡眠一段时间，代表清洗中
                Thread.sleep(SLEEP_GAP);
                Logger.info("洗完了");
            } catch (InterruptedException e) {
                Logger.info("清洗工作发生异常被中断.");
                return false;
            }
            Logger.info("清洗工作运行结束.");
            return true;
        }
    }
    public static void drinkTea(boolean warterOk, boolean cupOk) {
        if (warterOk&&cupOk) {
            Logger.info("泡茶喝");
        } else if (!warterOk) {
            Logger.info("烧水失败，没有茶喝了");
        } else if (!cupOk) {
            Logger.info("杯子洗不了，没有茶喝了");
        }
    }
    public static void main(String args[]) {
        Callable<Boolean>hJob = new HotWarterJob(); //③
        FutureTask<Boolean>hTask = new FutureTask<>(hJob); //④
```

```
    Thread hThread = new Thread(hTask, "*** 烧水-Thread");//⑤
    Callable<Boolean> wJob = new WashJob();//③
    FutureTask<Boolean> wTask = new FutureTask<>(wJob);//④
    Thread wThread = new Thread(wTask, "$$ 清洗-Thread");//⑤
    hThread.start();
    wThread.start();
    Thread.currentThread().setName("主线程");
    try {
        boolean warterOk = hTask.get();
        boolean cupOk = wTask.get();
        drinkTea(warterOk, cupOk);
    } catch (InterruptedException e) {
        Logger.info(getCurThreadName() + "发生异常被中断.");
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
    Logger.info(getCurThreadName() + " 运行结束.");
}
}
```

---

首先，在上面的喝茶实例代码中使用了Callable接口，替代了Runnable接口，并且在call方法中返回了异步线程的执行结果。

```
static class WashJob implements Callable<Boolean>
{
    @Override
    public Boolean call() throws Exception
    {
        //..业务代码，并且有清洗的结果
    }
}
```

---

其次，从Callable异步逻辑到异步线程，需要创建一个FutureTask类的实例，并通过FutureTask类的实例，创建新的线程：

```
Callable<Boolean> hJob = new HotWarterJob();//异步逻辑
FutureTask<Boolean> hTask = new FutureTask<Boolean>(hJob);//搭桥实例
Thread hThread = new Thread(hTask, "*** 烧水-Thread");//异步线程
```

---

FutureTask和Callable都是泛型类，泛型参数表示返回结果的类型。所以，在使用的时候，它们两个实例的泛型参数一定需要保持一致的。

最后，通过FutureTask类的实例，取得异步线程的执行结果。一般来说，通过FutureTask实例的get方法，可以获取线程的执行结果。

总之，FutureTask类的实现比join线程合并操作更加高明，能取得异步线程的结果。但是，也就未必高明到哪里去了。为啥呢？

因为通过FutureTask类的get方法，获取异步结果时，主线程也会被阻塞的。这一点，FutureTask和join也是一样的，它们俩都是异步阻塞模式。

异步阻塞的效率往往是比较低的，被阻塞的主线程不能干任何事情，唯一能干的，就是在傻傻地等待。原生Java API，除了阻塞模式的获取结果外，并没有实现

非阻塞的异步结果获取方法。如果需要用到获取异步的结果，则需要引入一些额外的框架，这里首先介绍谷歌公司的Guava框架。

## 5.4 Guava的异步回调

何为Guava？它是谷歌公司提供的Java扩展包，提供了一种异步回调的解决方案。相关的源代码在com.google.common.util.concurrent包中。包中的很多类，都是对java.util.concurrent能力的扩展和增强。例如，Guava的异步任务接口ListenableFuture，扩展了Java的Future接口，实现了非阻塞获取异步结果的功能。

总体来说，Guava的主要手段是增强而不是另起炉灶。为了实现非阻塞获取异步线程的结果，Guava对Java的异步回调机制，做了以下的增强：

- (1) 引入了一个新的接口ListenableFuture，继承了Java的Future接口，使得Java的Future异步任务，在Guava中能被监控和获得非阻塞异步执行的结果。
- (2) 引入了一个新的接口FutureCallback，这是一个独立的新接口。该接口的目的，是在异步任务执行完成后，根据异步结果，完成不同的回调处理，并且可以处理异步结果。

## 5.4.1 详解FutureCallback

FutureCallback是一个新增的接口，用来填写异步任务执行完后的监听逻辑。FutureCallback拥有两个回调方法：

(1) `onSuccess`方法，在异步任务执行成功后被回调；调用时，异步任务的执行结果，作为`onSuccess`方法的参数被传入。

(2) `onFailure`方法，在异步任务执行过程中，抛出异常时被回调；调用时，异步任务所抛出的异常，作为`onFailure`方法的参数被传入。

FutureCallback的源代码如下：

```
package com.google.common.util.concurrent;
public interface FutureCallback<V> {
    void onSuccess(@Nullable V var1);
    void onFailure(Throwable var1);
}
```

注意，Guava的FutureCallback与Java的Callable，名字相近，但实质不同，存在本质的区别：

(1) Java的Callable接口，代表的是异步执行的逻辑。

(2) Guava的FutureCallback接口，代表的是Callable异步逻辑执行完成之后，根据成功或者异常两种情况，所需要执行的善后工作。

Guava是对Java Future异步回调的增强，使用Guava异步回调，也需要用到Java的Callable接口。简单地说，只有在Java的Callable任务执行的结果出来之后，才可能执行Guava中的FutureCallback结果回调。

Guava如何实现异步任务Callable和FutureCallback结果回调之间的监控关系呢？Guava引入了一个新接口ListenableFuture，它继承了Java的Future接口，增强了监控的能力。

## 5.4.2 详解ListenableFuture

看ListenableFuture接口的名称，就知道它与Java中Future接口的亲戚关系。没错，Guava的ListenableFuture接口是对Java的Future接口的扩展，可以理解为异步任务的实例。源代码如下：

```
package com.google.common.util.concurrent;
import java.util.concurrent.Executor;
import java.util.concurrent.Future;
public interface ListenableFuture<V> extends Future<V> {
    //此方法由Guava内部调用
    void addListener(Runnable r, Executor e);
}
```

ListenableFuture仅仅增加了一个方法——addListener方法。它的作用就是将前一小节的FutureCallback善后回调工作，封装成一个内部的Runnable异步回调任务，在Callable异步任务完成后，回调FutureCallback进行善后处理。

注意，这个addListener方法只在Guava内部调用，如果对它感兴趣，可以查看Guava源代码。在实际编程中，我们不会调用addListener。

在实际编程中，如何将FutureCallback回调逻辑绑定到异步的ListenableFuture任务呢？可以使用Guava的Futures工具类，它有一个addCallback静态方法，可以将FutureCallback的回调实例绑定到ListenableFuture异步任务。下面是一个简单的绑定实例：

```
Futures.addCallback(listenableFuture, newFutureCallback<Boolean>()
{
    public void onSuccess(Boolean r)
    {
        // listenableFuture内部的Callable 成功时回调此方法
    }
    public void onFailure(Throwable t)
    {
        // listenableFuture内部的Callable异常时回调此方法
    }
});
```

现在的问题来了，既然Guava的ListenableFuture接口是对Java的Future接口的扩展，都表示异步任务。那么Guava的异步任务实例，从何而来呢？

### 5.4.3 ListenableFuture异步任务

如果要获取Guava的ListenableFuture异步任务实例，主要是通过向线程池(ThreadPool)提交Callable任务的方式来获取。不过，这里所说的线程池，不是Java的线程池，而是Guava自己定制的Guava线程池。

Guava线程池，是对Java线程池的一种装饰。创建Guava线程池的方法如下：

```
//java 线程池
ExecutorService jPool= Executors.newFixedThreadPool(10);
//Guava线程池
ListeningExecutorService gPool= MoreExecutors.listeningDecorator(jPool);
```

首先创建Java线程池，然后以它作为Guava线程池的参数，再构造一个Guava线程池。有了Guava的线程池之后，就可以通过submit方法来提交任务了；任务提交之后的返回结果，就是我们所要的ListenableFuture异步任务实例了。

简单地说，获取异步任务实例的方式，是通过向线程池提交Callable业务逻辑来实现。代码如下：

```
//调用submit方法来提交任务，返回异步任务实例
ListenableFuture<Boolean> hFuture = gPool.submit(hJob);
//绑定回调实例
Futures.addCallback(listenableFuture, new FutureCallback<Boolean>()
{
    //实现回调方法，有两个
});
```

获取了ListenableFuture实例之后，通过Futures.addCallback方法，将FutureCallback回调逻辑的实例绑定到ListenableFuture异步任务实例，实现异步执行完成后的回调。

总结一下，Guava异步回调的流程如下：

第1步：实现Java的Callable接口，创建异步执行逻辑。还有一种情况，如果不需要返回值，异步执行逻辑也可以实现Java的Runnable接口。

第2步：创建Guava线程池。

第3步：将第1步创建的Callable/Runnable异步执行逻辑的实例，通过submit提交到Guava线程池，从而获取ListenableFuture异步任务实例。

第4步：创建FutureCallback回调实例，通过Futures.addCallback将回调实例绑定到ListenableFuture异步任务上。

完成以上四步，当Callable/Runnable异步执行逻辑完成后，就会回调异步回调实例FutureCallback的回调方法onSuccess/onFailure。

## 5.4.4 使用Guava实现泡茶喝的实践案例

前面已经完成了join版本、FutureTask版本的泡茶喝实践案例。大家对此实例的业务功能，应该已经非常熟悉了，这里不再赘述。下面是Guava的异步回调的演进版本，代码如下：

```
package com.crazymakercircle.coccurent;
//...
public class GuavaFutureDemo {
    public static final int SLEEP_GAP = 500;
    public static String getCurThreadName() {
        return Thread.currentThread().getName();
    }
    //业务逻辑：烧水
    static class HotWarterJob implements Callable<Boolean> {
        @Override
        public Boolean call() throws Exception {
            //.....省略，与使用FutureTask实现异步泡茶喝相同
        }
    }
    //业务逻辑：清洗
    static class WashJob implements Callable<Boolean> {
        @Override
        public Boolean call() throws Exception {
            //.....省略，与使用FutureTask实现异步泡茶喝相同
        }
    }
    //新创建一个异步业务类型，作为泡茶喝主线程类
    static class MainJob implements Runnable {
        boolean warterOk = false;
        boolean cupOk = false;
        int gap = SLEEP_GAP / 10;
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(gap);
                    Logger.info("读书中.....");
                } catch (InterruptedException e) {
                    Logger.info(getCurThreadName() + "发生异常被中断.");
                }
                if (warterOk && cupOk) {
                    drinkTea(warterOk, cupOk);
                }
            }
        }
        public void drinkTea(Boolean wOk, Boolean cOK) {
            if (wOk && cOK) {
                Logger.info("泡茶喝，茶喝完");
                this.warterOk = false;
                this.gap = SLEEP_GAP * 100;
            } else if (!wOk) {
                Logger.info("烧水失败，没有茶喝了");
            } else if (!cOK) {
                Logger.info("杯子洗不了，没有茶喝了");
            }
        }
    }
    public static void main(String args[]) {
        //创建一个新的线程实例，作为泡茶主线程
        MainJob mainJob = new MainJob();
        Thread mainThread = new Thread(mainJob);
        mainThread.setName("主线程");
        mainThread.start();
    }
}
```

```
//烧水的业务逻辑实例
Callable<Boolean>hotJob = new HotWarterJob();
//清洗的业务逻辑实例
Callable<Boolean>washJob = new WashJob();
//创建Java 线程池
ExecutorServicejPool = Executors.newFixedThreadPool(10);
//包装Java线程池，构造Guava 线程池
ListeningExecutorServicegPool
    = MoreExecutors.listeningDecorator(jPool);
//提交烧水的业务逻辑实例，到Guava线程池获取异步任务
ListenableFuture<Boolean>hotFuture = gPool.submit(hotJob);
//绑定异步回调，烧水完成后，把喝水任务的warterOk标志设置为true
Futures.addCallback(hotFuture, new FutureCallback<Boolean>() {
    public void onSuccess(Boolean r) {
        if (r) {
            mainJob.warterOk = true;
        }
    }
    public void onFailure(Throwable t) {
        Logger.info("烧水失败，没有茶喝了");
    }
});
//提交清洗的业务逻辑实例，到Guava线程池获取异步任务
ListenableFuture<Boolean>washFuture = gPool.submit(washJob);
//绑定任务执行完成后的回调逻辑到异步任务
Futures.addCallback(washFuture, new FutureCallback<Boolean>() {
    public void onSuccess(Boolean r) {
        if (r) {
            mainJob.cupOk = true;
        }
    }
    public void onFailure(Throwable t) {
        Logger.info("杯子洗不了，没有茶喝了");
    }
});
});
```

---

Guava异步回调和Java的FutureTask异步回调，本质的不同在于：

- Guava是非阻塞的异步回调，调用线程是不阻塞的，可以继续执行自己的业务逻辑。
- FutureTask是阻塞的异步回调，调用线程是阻塞的，在获取异步结果的过程中，一直阻塞，等待异步线程返回结果。

## 5.5 Netty的异步回调模式

Netty官方文档中指出Netty的网络操作都是异步的。在Netty源代码中，大量使用了异步回调处理模式。在Netty的业务开发层面，Netty应用的Handler处理器中的业务处理代码，也都是异步执行的。所以，了解Netty的异步回调，无论是Netty应用级的开发还是源代码级的开发，都是十分重要的。

Netty和Guava一样，实现了自己的异步回调体系：Netty继承和扩展了JDK Future系列异步回调的API，定义了自身的Future系列接口和类，实现了异步任务的监控、异步执行结果的获取。

总体来说，Netty对JavaFuture异步任务的扩展如下：

(1) 继承Java的Future接口，得到了一个新的属于Netty自己的Future异步任务接口；该接口对原有的接口进行了增强，使得Netty异步任务能够以非阻塞的方式处理回调的结果；注意，Netty没有修改Future的名称，只是调整了所在的包名，Netty的Future类的包名和Java的Future接口的包名不同。

(2) 引入了一个新接口——GenericFutureListener，用于表示异步执行完成的监听器。这个接口和Guava的FutureCallback回调接口不同。Netty使用了监听器的模式，异步任务的执行完成后的回调逻辑抽象成了Listener监听器接口。可以将Netty的GenericFutureListener监听器接口加入Netty异步任务Future中，实现对异步任务执行状态的事件监听。

总体上说，在异步非阻塞回调的设计思路上，Netty和Guava的思路是一致的。对应关系为：

- Netty的Future接口，可以对应到Guava的ListenableFuture接口。
- Netty的GenericFutureListener接口，可以对应到Guava的FutureCallback接口。

## 5.5.1 详解GenericFutureListener接口

前面提到，和Guava的FutureCallback一样，Netty新增了一个接口来封装异步非阻塞回调的逻辑——它就是GenericFutureListener接口。

GenericFutureListener位于io.netty.util.concurrent包中，源代码如下：

```
package io.netty.util.concurrent;
import java.util.EventListener;
public interface GenericFutureListener<F extends Future<?>> extends EventListener {
    //监听器的回调方法
    void operationComplete(F var1) throws Exception;
}
```

GenericFutureListener拥有一个回调方法：operationComplete，表示异步任务操作完成。在Future异步任务执行完成后，将回调此方法。在大多数情况下，Netty的异步回调的代码编写在GenericFutureListener接口的实现类中的operationComplete方法中。

说明一下，GenericFutureListener的父接口EventListener是一个空接口，没有任何的抽象方法，是一个仅仅具有标识作用的接口。

## 5.5.2 详解Netty的Future接口

Netty也对Java的Future接口进行了扩展，并且名称没有变，还是叫作Future接口，代码实现位于io.netty.util.concurrent包中。

和Guava的ListenableFuture一样，Netty的Future接口，扩展了一系列的方法，对执行的过程的进行监控，对异步回调完成事件进行监听（Listen）。Netty的Future接口的源代码如下：

```
public interface Future<V> extends java.util.concurrent.Future<V> {  
    boolean isSuccess(); // 判断异步执行是否成功  
    boolean isCancelled(); // 判断异步执行是否取消  
    Throwable cause(); // 获取异步任务异常的原因  
    //增加异步任务执行完成与否的监听器Listener  
    Future<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);  
    //移除异步任务执行完成与否的监听器Listener  
    Future<V> removeListener(GenericFutureListener<? extends Future<? super V>> listener);  
    //....  
}
```

Netty的Future接口一般不会直接使用，而是会使用子接口。Netty有一系列的子接口，代表不同类型的异步任务，如ChannelFuture接口。

ChannelFuture子接口表示通道IO操作的异步任务；如果在通道的异步IO操作完成后，需要执行回调操作，就需要使用到ChannelFuture接口。

### 5.5.3 ChannelFuture的使用

在Netty的网络编程中，网络连接通道的输入和输出处理都是异步进行的，都会返回一个ChannelFuture接口的实例。通过返回的异步任务实例，可以为它增加异步回调的监听器。在异步任务真正完成后，回调才会执行。

Netty的网络连接的异步回调，实例代码如下：

```
//connect是异步的，仅提交异步任务
ChannelFuture future = bootstrap.connect(
    new InetSocketAddress("www.manning.com", 80));
//connect的异步任务真正执行完成后，future回调监听器才会执行
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
        if(channelFuture.isSuccess()){
            System.out.println("Connection established");
        } else {
            System.err.println("Connection attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

GenericFutureListener接口在Netty中是一个基础类型接口。在网络编程的异步回调中，一般使用Netty中提供的某个子接口，如ChannelFutureListener接口。在上面的代码中，使用的是这个子接口。

## 5.5.4 Netty的出站和入站异步回调

Netty的出站和入站操作都是异步的。异步回调的方法，和上面Netty建立的异步回调是一样的。

以最为经典的NIO出站操作——write为例，说明一下ChannelFuture的使用。

在调用write操作后，Netty并没有完成对Java NIO底层连接的写入操作，因为是异步执行的。代码如下：

```
//write输出方法，返回的是一个异步任务
ChannelFuture future = ctx.channel().write(msg);
//为异步任务，加上监听器
future.addListener(
    new ChannelFutureListener()
{
    @Override
    public void operationComplete(ChannelFuture future)
    {
        // write操作完成后的回调代码
    }
});
```

在调用write操作后，是立即返回，返回的是一个ChannelFuture接口的实例。通过这个实例，可以绑定异步回调监听器，这里的异步回调逻辑需要我们编写。

如果大家运行以上的EchoServer实践案例，就会发现一个很大的问题：客户端接收到的回显信息和发送到服务器的信息，不是一对一对应输出的。看到的比较多的情况是：客户端发出很多次信息后，客户端才收到一次服务器的回显。

这是什么原因呢？这就是网络通信中的粘包/半包问题。对于这个问题的解决方案，在后面会做非常详细的解答，这里暂时搁置。粘包/半包问题的出现，说明了一个问题：仅仅基于Java的NIO，开发一套高性能、没有Bug的通信服务器程序，远远没有大家想象的简单，有一系列的“坑”、一大堆的基础问题等着大家解决。

在进行大型的Java通信程序的开发时，尽量基于一些实现了成熟、稳定的基础通信的Java开源中间件（如Netty）。这些中间件已经帮助大家解决了很多的基础问题，如前面出现的粘包/半包问题。

至此，大家已经学习了Java NIO、Reactor反应器模式、Future模式，这些都是学习Netty应用开发的基础。基础知识已经铺垫得差不多了，接下来到了正式进入学习Netty的阶段。

## 5.6 本章小结

随着高并发系统越来越多，异步回调模式也越来越重要。在Netty源代码中，大量地使用了异步回调技术，所以，在开始介绍Netty之前，开辟整整一章，非常详细地、由浅入深地为大家介绍了异步回调模式。

本章首先为大家介绍了Java的join合并线程时“闷葫芦式”的异步阻塞，然后介绍了Java的FutureTask阻塞式的获取异步任务结果，最后介绍了Guava和Netty的异步回调方式。

Guava和Netty的异步回调是非阻塞的，而Java的join、FutureTask都是阻塞的。

## 第6章 Netty原理与基础

首先引用Netty官网的内容对Netty进行一个正式的介绍。

Netty是为了快速开发可维护的高性能、高可扩展、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。换句话说，Netty是一个Java NIO客户端/服务器框架。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，它可以大大简化了网络编程流程。例如，Netty极大地简化TCP、UDP套接字、HTTP Web服务程序的开发。

Netty的目标之一，是要使开发可以做到“快速和轻松”。除了做到“快速和轻松”的开发TCP/UDP等自定义协议的通信程序之外，Netty经过精心设计，还可以做到“快速和轻松”地开发应用层协议的程序，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Reactor反应器模式。后面会详细介绍Netty中反应器模式的实现。在基于Netty的反应器模式实现中的Channel（通道）、Handler（处理器）等基类，能快速扩展以覆盖不同协议、完成不同业务处理的大量应用类。

## 6.1 第一个Netty的实践案例DiscardServer

在开始实践之前，第一步就是要准备Netty的版本，配置好开发环境。

## 6.1.1 创建第一个Netty项目

首先我们需要创建项目，项目名称是NettyDemos。这是一个丢弃服务器（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，首先需要考虑一下JDK的版本，官网建议使用JDK1.6以上，本书使用的是JDK1.8。然后是Netty自己的版本，建议使用Netty 4.0以上的版本。虽然Netty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。本书使用的Netty版本是4.1.6。

使用maven导入Netty以依赖到工程（或项目），Netty的maven依赖如下：

```
<!-- https://mvnrepository.com/artifact/io.netty/netty-all -->
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.6.Final</version>
</dependency>
```

那么现在可以正式开始编写第一个项目程序了。

## 6.1.2 第一个Netty服务器端程序

这里创建一个服务端类： NettyDiscardServer， 用以实现消息的Discard“丢弃”功能， 它的源代码如下：

```
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() {
        //创建反应器线程组
        EventLoopGroupbossLoopGroup = new NioEventLoopGroup(1);
        EventLoopGroupworkerLoopGroup = new NioEventLoopGroup();
        try {
            //1 设置反应器线程组
            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);
            b.option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
            //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(SocketChannelch) throws Exception {
                    // 流水线管理子通道中的Handler处理器
                    // 向子通道流水线添加一个handler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
            ChannelFuturechannelFuture = 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
            ChannelFuturecloseFuture = channelFuture.channel().closeFuture();
            closeFuture.sync();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 8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) throws InterruptedException {
        int port = NettyDemoConfig.SOCKET_SERVER_PORT;
        new NettyDiscardServer(port).runServer();
    }
}
```

如果是第一次看Netty开发的程序， 上面的代码是看不懂的， 因为代码里边涉及很多的Netty组件。

Netty是基于反应器模式实现的。还好，大家已经非常深入地了解了反应器模式，现在大家顺藤摸瓜学习Netty的结构就相对简单了。

首先要说的是Reactor反应器。前面讲到，反应器的作用是进行一个IO事件的select查询和dispatch分发。Netty中对应的反应器组件有多种，应用场景不同，用到的反应器也各不相同。一般来说，对应于多线程的Java NIO通信的应用场景，Netty的反应器类型为：NioEventLoopGroup。

在上面的例子中，使用了两个NioEventLoopGroup实例。第一个通常被称为“包工头”，负责服务器通道新连接的IO事件的监听。第二个通常被称为“工人”，主要负责传输通道的IO事件的处理。

其次要说的是Handler处理器（也称为处理程序）。Handler处理器的作用是对应到IO事件，实现IO事件的业务处理。Handler处理器需要专门开发，稍后，将专门对它进行介绍。

再次，在上面的例子中，还用到了Netty的服务启动类ServerBootstrap，它的职责是一个组装和集成器，将不同的Netty组件组装在一起。另外，ServerBootstrap能够按照应用场景的需要，为组件设置好对应的参数，最后实现Netty服务器的监听和启动。服务启动类ServerBootstrap也是本章重点之一，稍候另起一小节进行详细的介绍。

### 6.1.3 业务处理器NettyDiscardHandler

在反应器（Reactor）模式中，所有的业务处理都在Handler处理器中完成。这里编写一个新类：NettyDiscardHandler。NettyDiscardHandler的业务处理很简单：把收到的任何内容直接丢弃（discard），也不会回复任何消息。代码如下：

```
package com.crazymakercircle.netty.basic;
//...
public class NettyDiscardHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        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();
        } finally {
            ReferenceCountUtil.release(msg);
        }
    }
}
```

首先说明一下，这里将引入一个新的概念：入站和出站。简单来说，入站指的是输入，出站指的是输出。后面也会有详细介绍。

Netty的Handler处理器需要处理多种IO事件（如可读、可写），对应于不同的IO事件，Netty提供了一些基础的方法。这些方法都已经提前封装好，后面直接继承或者实现即可。比如说，对于处理入站的IO事件的方法，对应的接口为ChannelInboundHandler入站处理接口，而ChannelInboundHandlerAdapter则是Netty提供的入站处理的默认实现。

也就是说，如果要实现自己的入站处理器Handler，只要继承ChannelInboundHandlerAdapter入站处理器，再写入自己的入站处理的业务逻辑。如果要读取入站的数据，只要写在了入站处理方法channelRead中即可。

在上面例子中的channelRead方法，它读取了Netty的输入数据缓冲区ByteBuf。Netty的ByteBuf，可以对应到前面介绍的NIO的数据缓冲区。它们在功能上是类似的，不过相对而言，Netty的版本性能更好，使用也更加方便。后面会另开一节进行详细的介绍。

## 6.1.4 运行NettyDiscardServer

在上面的例子中，出现了Netty中的各种组件：服务器启动器、缓冲区、反应器、Handler业务处理器、Future异步任务监听、数据传输通道等。这些Netty组件都是需要掌握的，也都是我们在后面要专项学习的。

如果看不懂这个NettyDiscardServer程序，一点儿也没关系。这个程序在本章的目的，仅仅是为大家展示一下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，体验一下Netty程序的运行。

打开源代码工程com.crazymakercircle.netty.basic包，从中找到服务器类：NettyDiscardServer。启动它的main方法，就启动了这个服务器。

但是，如果要看到最终的丢弃效果，不能仅仅启动服务器，还需要启动客户端，由客户端向服务器发送消息。客户端在哪儿呢？

这里的客户端，只要能发消息到服务器即可，不需要其他特殊的功能。因此，可以直接使用前面示例中的EchoClient程序来作为客户端运行即可，因为端口是一致的。

在源代码工程的com.crazymakercircle.ReactorModel包中，我们可以找到发送消息到服务器的客户端类：EchoClient。启动它的main方法，就启动了这个客户端。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。

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。因为NettyDiscardServer的底层也是使用Java NIO。

## 6.2 解密Netty中的Reactor反应器模式

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Java程序往往找不到头绪，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中的反应器模式是如何实现的。

现在，现回顾一下Java NIO中IO事件的处理流程和反应器模式的基础内容。

## 6.2.1 回顾Reactor反应器模式中IO事件的处理流程

一个IO事件从操作系统底层产生后，在Reactor反应器模式中的处理流程如图6-1所示。

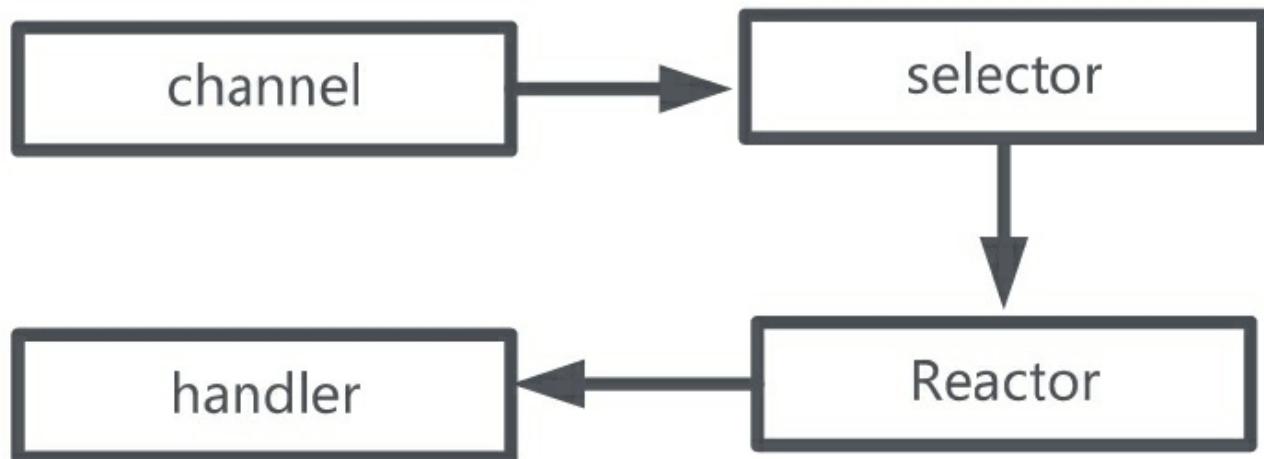


图6-1 Java Reactor反应器模式中IO事件的处理流程

整个流程大致分为4步，具体如下：

第1步：通道注册。IO源于通道（Channel）。IO是和通道（对应于底层连接而言）强相关的。一个IO事件，一定属于某个通道。但是，如果要查询通道的事件，首先要将通道注册到选择器。只需通道提前注册到Selector选择器即可，IO事件会被选择器查询到。

第2步：查询选择。在反应器模式中，一个反应器（或者SubReactor子反应器）会负责一个线程；不断地轮询，查询选择器中的IO事件（选择键）。

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

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

以上4步，就是整个反应器模式的IO处理器流程。其中，第1步和第2步，其实是Java NIO的功能，反应器模式仅仅是利用了Java NIO的优势而已。

题外话：上面的流程比较重要，是学习Netty的基础。如果这里看不懂，作为铺垫，请先回到反应器模式的详细介绍部分，回头再学习一下反应器模式。

## 6.2.2 Netty中的Channel通道组件

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

Netty中不直接使用Java NIO的Channel通道组件，对Channel通道组件进行了自己的封装。在Netty中，有一系列的Channel通道组件，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。

另外一点就是，除了Java的NIO，Netty还能处理Java的面向流的OIO（Old-IO，即传统的阻塞式IO）。

总结起来，Netty中的每一种协议的通道，都有NIO（异步IO）和OIO（阻塞式IO）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服务器端监听通道。
- 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服务器端监听通道。
- OioSocketChannel**: 同步阻塞式TCP Socket传输通道。
- OioServerSocketChannel**: 同步阻塞式TCP Socket服务器端监听通道。
- 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服务器端监听通道。

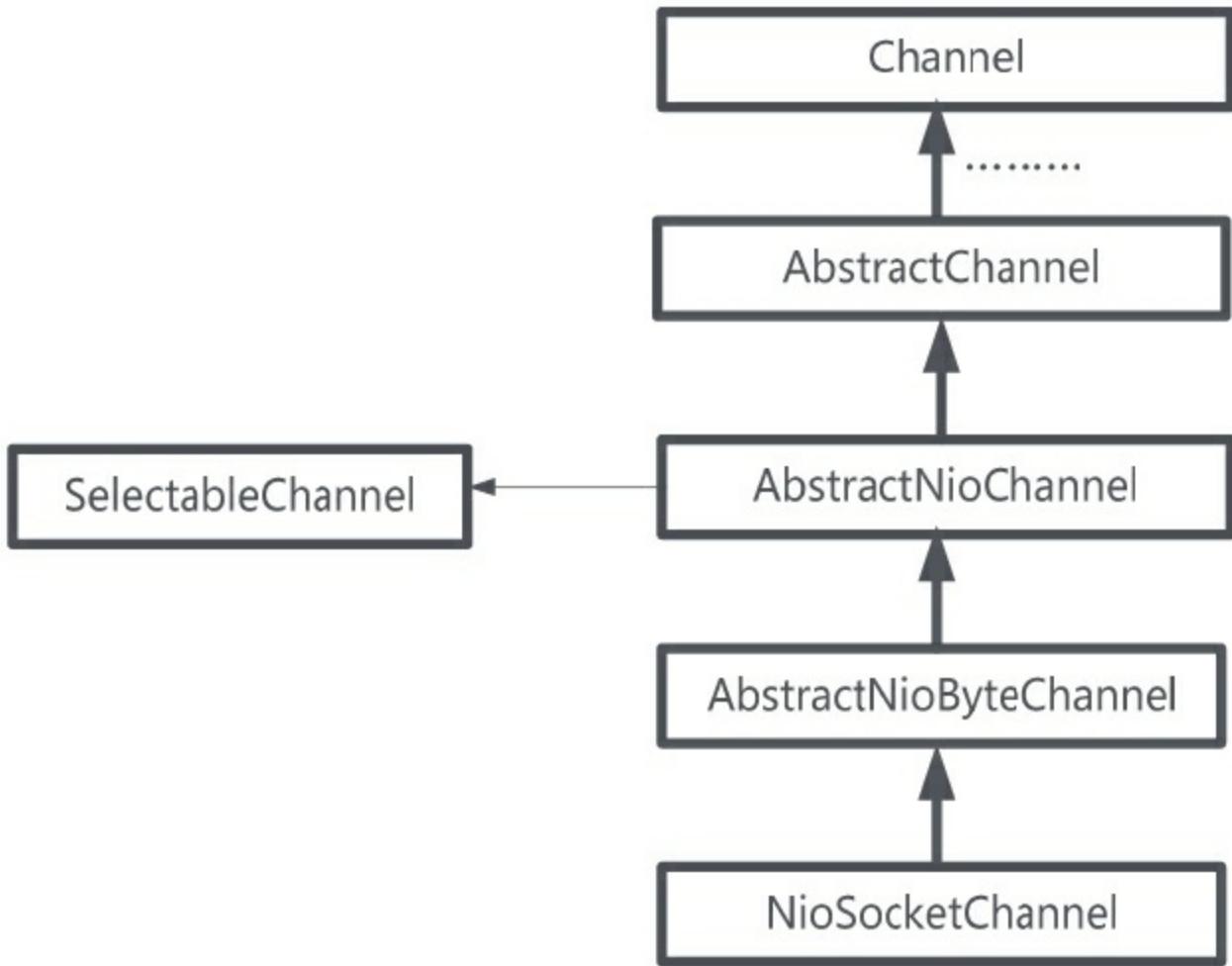


图6-2 NioSocketChannel的继承关系图

一般来说，服务器端编程用到最多的通信协议还是TCP协议。对应的传输通道类型为NioSocketChannel类，服务器监听类为NioServerSocketChannel。在主要使用的方法上，其他的通道类型和这个NioSocketChannel类在原理上基本是相通的，因此，本书的很多案例都以NioSocketChannel通道为主。

在Netty的NioSocketChannel内部封装了一个Java NIO的SelectableChannel成员。通过这个内部的Java NIO通道，Netty的NioSocketChannel通道上的IO操作，最终会落地到Java NIO的SelectableChannel底层通道。NioSocketChannel的继承关系图，如图6-2所示。

### 6.2.3 Netty中的Reactor反应器

在反应器模式中，一个反应器（或者SubReactor子反应器）会负责一个事件处理线程，不断地轮询，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，则分发给Handler业务处理器。

Netty中的反应器有多个实现类，与Channel通道类有关系。对于NioSocketChannel通道，Netty的反应器类为：NioEventLoop。

NioEventLoop类绑定了两个重要的Java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的继承关系和主要的成员属性，如下图6-3所示。

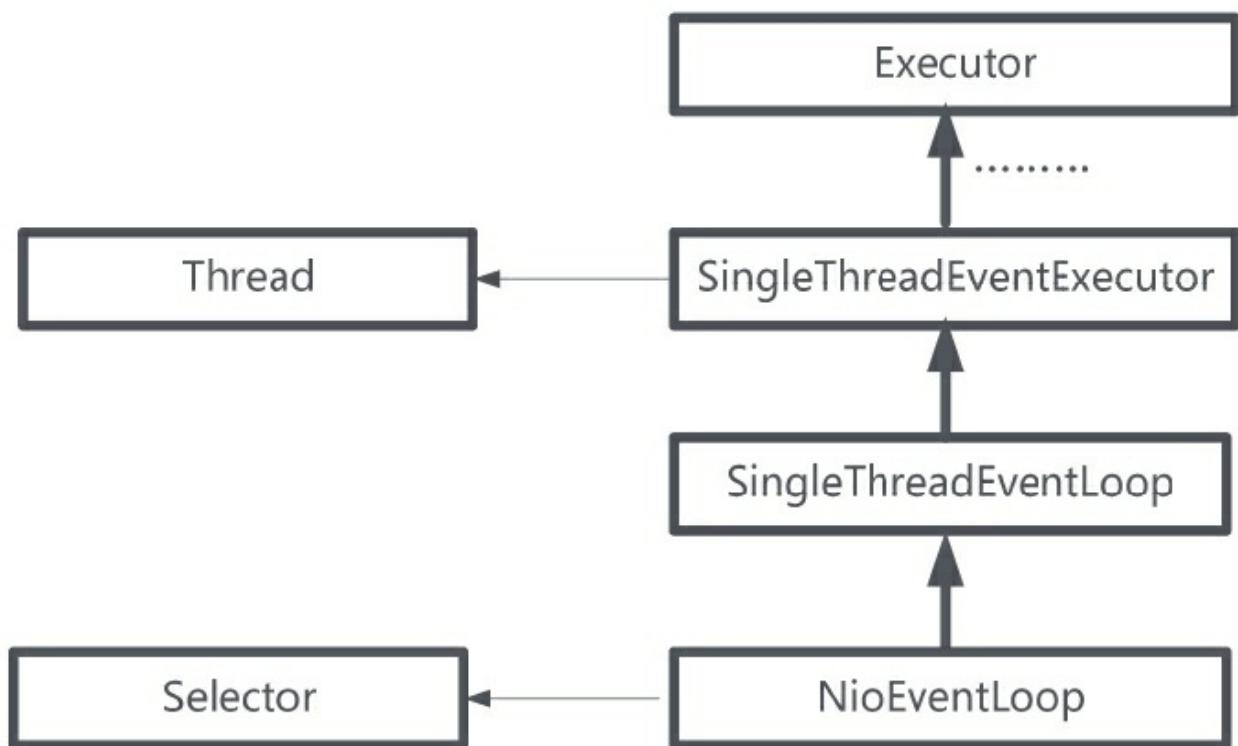


图6-3 NioEventLoop的继承关系和主要的成员

通过这个关系图，可以看出：NioEventLoop和前面章节讲到反应器，在思路上是一致的：一个NioEventLoop拥有一个Thread线程，负责一个Java NIO Selector选择器的IO事件轮询。

在Netty中，EventLoop反应器和Netty Channel通道，关系如何呢？理论上来讲，一个EventLoopNetty反应器和NettyChannel通道是一对多的关系：一个反应器可以注册成千上万的通道。

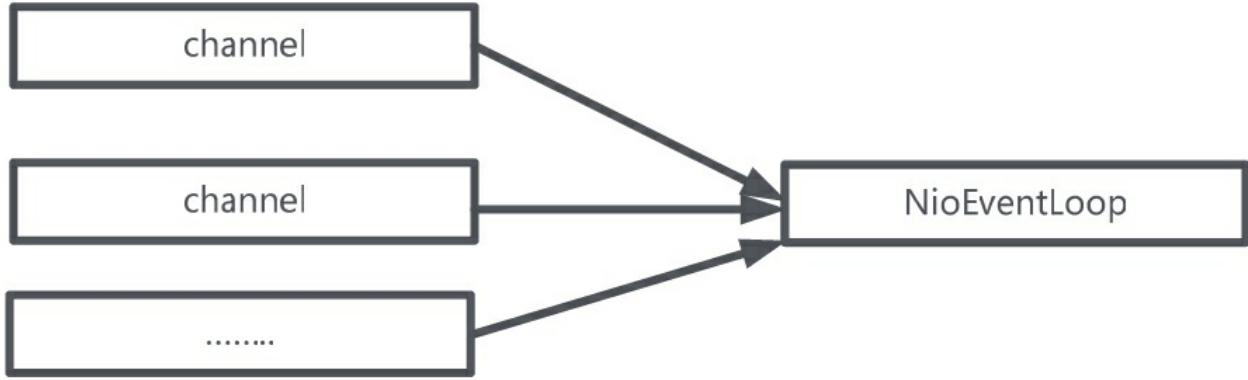


图6-4 EventLoop反应器和通道（Channel）的关系

## 6.2.4 Netty中的Handler处理器

在前面的章节，解读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Java NIO选择器成员执行以上事件的查询，然后进行对应的事件分发。事件分发（Dispatch）的目标就是Netty自己的Handler处理器。

Netty的Handler处理器分为两大类：第一类是ChannelInboundHandler通道入站处理器；第二类是ChannelOutboundHandler通道出站处理器。二者都继承了ChannelHandler处理器接口。Netty中的Handler处理器的接口与继承关系，如图6-5所示。

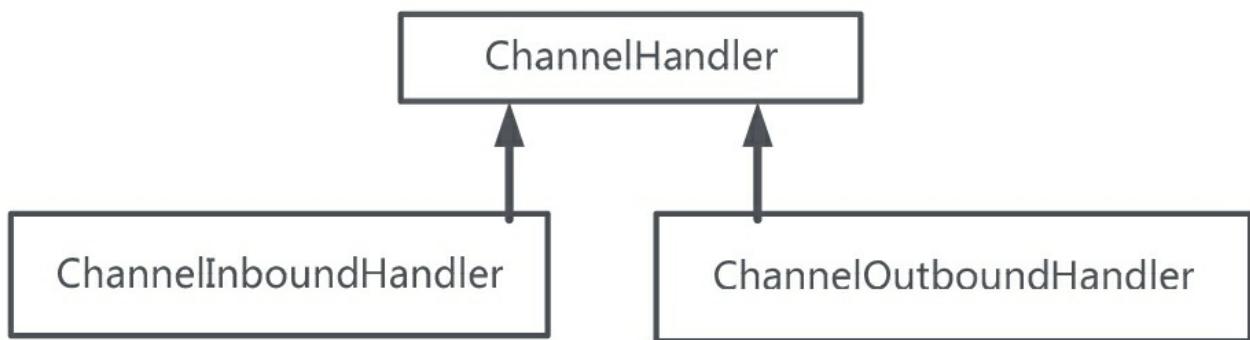


图6-5 Netty中的Handler处理器的接口与继承关系

Netty中的入站处理，不仅仅是OP\_READ输入事件的处理，还是从通道底层触发，由Netty通过层层传递，调用ChannelInboundHandler通道入站处理器进行的某个处理。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通道入站处理器，调用它的入站处理的方法read。在ChannelInboundHandler通道入站处理器内部的read方法可以从通道中读取数据。

Netty中的入站处理，触发的方向为：从通道到ChannelInboundHandler通道入站处理器。

Netty中的出站处理，本来就已经包括Java NIO的OP\_WRITE可写事件。注意，OP\_WRITE可写事件是Java NIO的底层概念，它和Netty的出站处理的概念不是一个维度，Netty的出站处理是应用层维度的。那么，Netty中的出站处理，具体指的是什么呢？指的是从ChannelOutboundHandler通道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通道出站处理器将处理的结果写入底层通道。它的最常用的一个方法就是write()方法，把数据写入到通道。

这两个业务处理接口都有各自的默认实现：ChannelInboundHandler的默认实现为ChannelInboundHandlerAdapter，叫作通道入站处理适配器。

ChannelOutboundHandler的默认实现为ChannelOutboundHandlerAdapter，叫作通道出站处理适配器。这两个默认的通道处理适配器，分别实现了入站操作和出站操作的基本功能。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。

## 6.2.5 Netty的流水线（Pipeline）

来梳理一下Netty的反应器模式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事件被多个的Handler实例处理；一个Handler处理器实例也能绑定到很多的通道，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，串在一起，形成一条流水线。ChannelPipeline（通道流水线）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了双向链表的节点，被加入到了ChannelPipeline（通道流水线）中。

重点申明：一个Netty通道拥有一条Handler处理器流水线，成员的名称叫作pipeline。

问题来了：这里为什么将pipeline翻译成流水线，而不是翻译成为管道呢？这是有原因的。具体来说，与流水线内部的Handler处理器之间处理IO事件的先后次序有关。

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通道流水线。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，这就意味着这个IO事件在此次流水线中的处理结束了。

(3) 如果在流水线中间需要终止流动，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。

为什么说Handler的处理是按照既定的次序，而不是从前到后的次序呢？Netty是

这样规定的：入站处理器Handler的执行次序，是从前到后；出站处理器Handler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6-6所示。

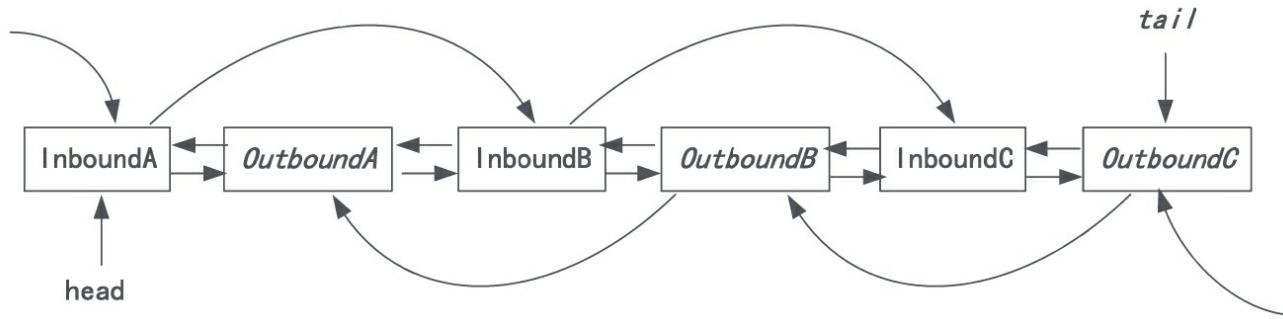


图6-6 流水线上入站处理器和出站处理器的执行次序

除了流动的方向与IO操作的类型有关之外，流动过程中经过的处理器节点的类型，也是与IO操作的类型有关。入站的IO操作只会且只能从Inbound入站处理器类型的Handler流过；出站的IO操作只会且只能从Outbound出站处理器类型的Handler流过。

总之，流水线是通道的“大管家”，为通道管理好了它的一大堆Handler“小弟”。

了解完了流水线之后，大家应该对Netty中的通道、EventLoop反应器、Handler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。至此，大家基本可以动手开发简单的Netty程序了。不过，为了方便开发者，Netty提供了一个类把上面的三个组件快速组装起来，这个系列的类叫作Bootstrap启动器。严格来说，不止一个类名字为Bootstrap，例如在服务器端的启动类叫作ServerBootstrap类。

下面，为大家详细介绍一下这个提升开发效率的Bootstrap启动器类。

## 6.3 详解Bootstrap启动器类

Bootstrap类是Netty提供的一个便利的工厂类，可以通过它来完成Netty的客户端或服务器端的Netty组件的组装，以及Netty程序的初始化。当然，Netty的官方解释是，完全可以不用这个Bootstrap启动器。但是，一点点去手动创建通道、完成各种设置和启动、并且注册到EventLoop，这个过程会非常麻烦。通常情况下，还是使用这个便利的Bootstrap工具类会效率更高。

在Netty中，有两个启动器类，分别用在服务器和客户端。如图6-7所示。

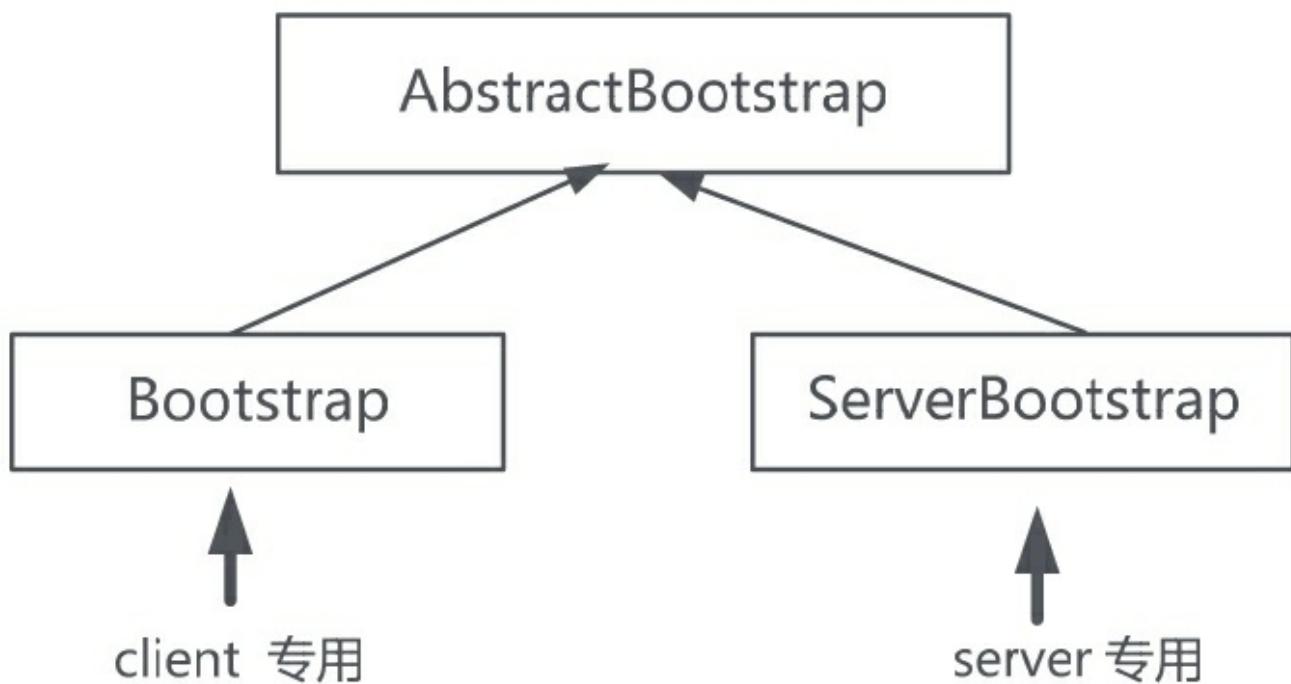


图6-7 Netty中的两个启动器类

这两个启动器仅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`服务器启动类作为重点的介绍对象。

在介绍`ServerBootstrap`的服务器启动流程之前，首先介绍一下涉及到的两个基础概念：父子通道、`EventLoopGroup`线程组（事件循环线程组）。

### 6.3.1 父子通道

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

- 连接监听类型。连接监听类型的socket描述符，放在服务器端，它负责接收客户端的套接字连接；在服务器端，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描述符。

- 传输数据类型。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描述符。

在Netty中，异步非阻塞的服务器端监听通道NioServerSocketChannel，封装在Linux底层的描述符，是“连接监听类型”socket描述符；而NioSocketChannel异步非阻塞TCP Socket传输通道，封装在底层Linux的描述符，是“数据传输类型”的socket描述符。

在Netty中，将有接收关系的NioServerSocketChannel和NioSocketChannel，叫作父子通道。其中，NioServerSocketChannel负责服务器连接监听和接收，也叫父通道（Parent Channel）。对于每一个接收到的NioSocketChannel传输类通道，也叫子通道（Child Channel）。

### 6.3.2 EventLoopGroup线程组

Netty中的Reactor反应器模式，肯定不是单线程版本的反应器模式，而是多线程版本的反应器模式。Netty的多线程版本的反应器模式是如何实现的呢？

在Netty中，一个EventLoop相当于一个子反应器（SubReactor）。大家已经知道，一个NioEventLoop子反应器拥有了一个线程，同时拥有一个Java NIO选择器。Netty如何组织外层的反应器呢？答案是使用EventLoopGroup线程组。多个EventLoop线程组成一个EventLoopGroup线程组。

反过来说，Netty的EventLoopGroup线程组就是一个多线程版本的反应器。而其中的单个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线程，而是使用EventLoopGroup线程组。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Thread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IO事件查询和分发。

如果使用EventLoopGroup的无参数的构造函数，没有传入线程数或者传入的线程数为0，那么EventLoopGroup内部的线程数到底是多少呢？默认的EventLoopGroup内部线程数为最大可用的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8个子反应器（SubReactor）实例。

从前文可知，为了及时接受（Accept）到新连接，在服务器端，一般有两个独立的反应器，一个反应器负责新连接的监听和接受，另一个反应器负责IO事件处理。对应到Netty服务器程序中，则是设置两个EventLoopGroup线程组，一个EventLoopGroup负责新连接的监听和接受，一个EventLoopGroup负责IO事件处理。

那么，两个反应器如何分工呢？负责新连接的监听和接受的EventLoopGroup线程组，查询父通道的IO事件，有点像负责招工的包工头，因此，可以形象地称为“包工头”（Boss）线程组。另一个EventLoopGroup线程组负责查询所有子通道的IO事件，并且执行Handler处理器中的业务处理——例如数据的输入和输出（有点儿像搬砖），这个线程组可以形象地称为“工人”（Worker）线程组。

至此，已经介绍了两个基础概念：父子通道、EventLoopGroup线程组。下一节将介绍ServerBootstrap的启动流程。

### 6.3.3 Bootstrap的启动流程

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，大致分成了8个步骤。本书仅仅演示的是服务器端启动器的使用，用到的启动器类为ServerBootstrap。正式使用前，首先创建一个服务器端的启动器实例。

```
//创建一个服务器端的启动器  
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：创建反应器线程组，并赋值给ServerBootstrap启动器实例

```
//创建反应器线程组  
//boss线程组  
EventLoopGroup bossLoopGroup = new NioEventLoopGroup(1);  
//worker线程组  
EventLoopGroup workerLoopGroup = new NioEventLoopGroup();  
//...  
//1 设置反应器线程组  
b.group(bossLoopGroup, workerLoopGroup);
```

在设置反应器线程组之前，创建了两个NioEventLoopGroup线程组，一个负责处理连接监听IO事件，名为bossLoopGroup；另一个负责数据IO事件和Handler业务处理，名为workerLoopGroup。

在线程组创建完成后，就可以配置给启动器实例，调用的方法是**b.group(bossGroup,workerGroup)**，它一次性地给启动器配置了两大线程组。

不一定非得配置两个线程组，可以仅配置一个EventLoopGroup反应器线程组。具体的配置方法是调用**b.group(workerGroup)**。在这种模式下，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：新连接的接受被更加耗时的数据传输或者业务处理所阻塞。

在服务器端，建议设置成两个线程组的工作模式。

#### 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（也叫BIO，Block-IO，即阻塞式IO）。下面配置的是Java NIO类型的通道类型，方法如下：

```
//2 设置nio类型的通道
```

```
b.channel(NioServerSocketChannel.class);
```

如果确实需要指定Bootstrap的IO模型为BIO，那么这里配置上Netty的OioServerSocketChannel.class类即可。由于NIO的优势巨大，通常不会在Netty中使用BIO。

### 第3步：设置监听端口

```
//3 设置监听端口  
b.localAddress(new InetSocketAddress(port));
```

这是最为简单的一步操作，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

```
//4 设置通道的参数  
b.option(ChannelOption.SO_KEEPALIVE, true);  
b.option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
```

这里用到了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）接收连接通道设置一些选项。

如果要给子通道（Child Channel）设置一些通道选项，则需要用另外一个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流水线

上一节介绍到，每一个通道的子通道，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实例加入双向链表中。

装配子通道的Handler流水线调用childHandler()方法，传递一个ChannelInitializer通道初始化类的实例。在父通道成功接收一个连接，并创建成功一个子通道后，就会初始化子通道，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到，向子通道流水线增加业务处理器。

```
//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的子通道，并初始化
    protected void initChannel(SocketChannel ch) throws Exception {
        // 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：父通道也就是NioServerSocketChannel连接接受通道，它的内部业务处理是固定的：接受新连接后，创建子通道，然后初始化子通道，所以不需要特别的配置。如果需要完成特殊的业务处理，可以使用ServerBootstrap的handler(ChannelHandler handler)方法，为父通道设置ChannelInitializer初始化器。

说明一下，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启动器中设置的通道类型，一一对应起来。

## 第6步：开始绑定服务器新连接的监听端口

```
// 6 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：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操作会立刻返回，在返回的时候，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），通过自我阻塞一直到ChannelFuture异步任务执行完成，或者为ChannelFuture增加事件监听器的两种方式，以获得Netty中的IO操作的真正结果。上面使用了第一种。

至此，服务器正式启动。

## 第7步：自我阻塞，直到通道关闭

```
// 7 等待通道关闭
// 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，可以使用通道的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

```
// 8关闭EventLoopGroup,  
// 释放掉所有资源，包括创建的反应器线程  
workerLoopGroup.shutdownGracefully();  
bossLoopGroup.shutdownGracefully();
```

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

### 6.3.4 ChannelOption通道选项

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选项。在ChannelOption类中，定义了一大票通道选项。下面介绍一些常见的选项。

#### 1.SO\_RCVBUF, SO\_SNDBUF

此为TCP参数。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连接的这两个缓冲区大小的。TCP的全双工的工作模式以及TCP的滑动窗口便是依赖于这两个独立的缓冲区及其填充的状态。

#### 2.TCP\_NODELAY

此为TCP参数。表示立即发送数据，默认值为True（Netty默认为True，而操作系统默认为False）。该值用于设置Nagle算法的启用，该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默认禁用该算法，从而最小化报文传输的延时。

说明一下：这个参数的值，与是否开启Nagle算法是相反的，设置为true表示关闭，设置为false表示开启。通俗地讲，如果要求高实时性，有数据发送时就立刻发送，就设置为true，如果需要减少发送次数和减少网络交互次数，就设置为false。

#### 3.SO\_KEEPALIVE

此为TCP参数。表示底层TCP协议的心跳机制。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。可以将此功能视为TCP的心跳机制，需要注意的是：默认的心跳间隔是7200s即2小时。Netty默认关闭该功能。

#### 4.SO\_REUSEADDR

此为TCP参数。设置为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：

- 当有一个有相同本地地址和端口的socket1处于TIME\_WAIT状态时，而我们希望启动的程序的socket2要占用该地址和端口。例如在重启服务且保持先前端口时。

- 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。

- 单个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。

## [5.SO\\_LINGER](#)

此为TCP参数。表示关闭socket的延迟时间，默认值为-1，表示禁用该功能。-1表示socket.close()方法立即返回，但操作系统底层会将发送缓冲区全部发送到对端。0表示socket.close()方法立即返回，操作系统放弃发送缓冲区的数据，直接向对端发送RST包，对端收到复位错误。非0整数值表示调用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。

## [6.SO\\_BACKLOG](#)

此为TCP参数。表示服务器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。默认值，在Windows中为200，其他操作系统为128。

如果连接建立频繁，服务器处理新连接较慢，可以适当调大这个参数。

## [7.SO\\_BROADCAST](#)

此为TCP参数。表示设置广播模式。

## 6.4 详解Channel通道

先介绍一下，在使用Channel通道的过程中所涉及的主要成员和方法。然后，为大家介绍一下Netty提供了一个专门的单元测试通道——EmbeddedChannel（嵌入式通道）。

## 6.4.1 Channel通道的主要成员和方法

在Netty中，通道是其中的核心概念之一，代表着网络连接。通道是通信的主题，由它负责同对端进行网络通信，可以写入数据到对端，也可以从对端读取数据。

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(); //底层的NIO 通道,完成实际的IO操作  
    pipeline = newChannelPipeline(); //一条通道,拥有一条流水线  
}
```

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。这段代码也表明，每个通道拥有一条ChannelPipeline处理器流水线。

AbstractChannel内部有一个parent属性，表示通道的父通道。对于连接监听通道（如NioServerSocketChannel实例）来说，其父亲通道为null；而对于每一条传输通道（如NioSocketChannel实例），其parent属性的值为接收到该连接的服务器连接监听通道。

几乎所有的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。

再来看一下，在通道接口中所定义的几个重要方法：

### 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，返回值为负责连接操作的异步任务ChannelFuture。此方法在客户端的传输通道使用。

### 方法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使用。

### 方法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。

#### 方法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。

#### 方法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Java NIO通道。此方法的返回值为出站处理的异步处理任务。

#### 方法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。并不是每一次write操作都是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到操作系统的缓冲区，操作系统会将根据缓冲区的情况，决定什么时候把数据写到对端。而执行flush()方法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。

## 6.4.2 EmbeddedChannel嵌入式通道

在Netty的实际开发中，通信的基础工作，Netty已经替大家完成。实际上，大量的工作是设计和开发ChannelHandler通道业务处理器，而不是开发Outbound出站处理器，换句话说就是开发Inbound入站处理器。开发完成后，需要投入单元测试。单元测试的大致流程是：需要将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。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，这整个的过程是非常的烦琐和浪费时间的。如何解决这种徒劳的、低效的重复工作呢？

Netty提供了一个专用通道——名字叫EmbeddedChannel（嵌入式通道）。

EmbeddedChannel仅仅是模拟入站与出站的操作，底层不进行实际的传输，不需要启动Netty服务器和客户端。除了不进行传输之外，EmbeddedChannel的其他的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，使用它，开发人员可以在开发的过程中方便、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6-1所示。

表6-1 EmbeddedChannel单元测试的辅助方法

名称	说明
writeInbound(...)	向通道写入 inbound 入站数据，模拟通道收到数据。也就是说，这些写入的数据会被流水线上的入站处理器处理
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 outbound 出站数据，模拟通道发送数据。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readOutbound方法。

### 方法1.writeInbound入站数据写到通道

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Inbound（入站）数据。可以调用writeInbound方法，向EmbeddedChannel写入一个入站二进制ByteBuf数据包，模拟底层的入站包。

### 方法2.readOutbound读取通道的出站数据

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），需要查看处理过的数据结果。可以调用**readOutbound**方法，读取通道的最终出站结果，它是经过流水线一系列的出站处理后，最终的出站数据包。比较绕口，重复一遍，通过**readOutbound**，可以读取完成**EmbeddedChannel**最后一个出站处理器，处理后的**ByteBuf**二进制出站包。

总之，这个**EmbeddedChannel**类，既具备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。它的具体用法，后面还会结合其他的Netty组件的实例反复提到。

## 6.5 详解Handler业务处理器

在Reactor反应器经典模型中，反应器查询到IO事件后，分发到Handler业务处理器，由Handler完成IO操作和业务处理。

整个的IO处理操作环节包括：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6-8所示。

前后两个环节，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。

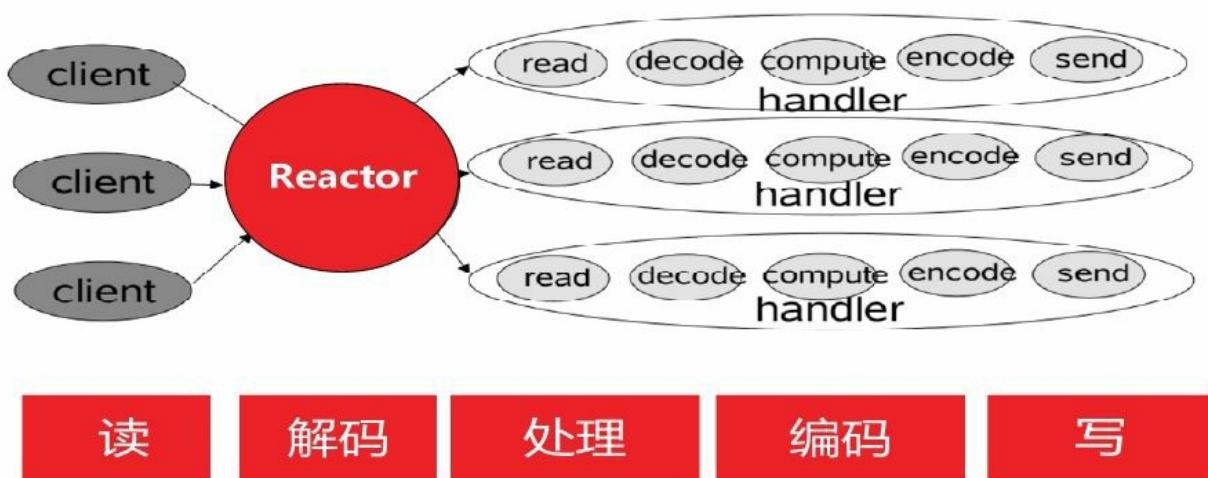


图6-8 整个的IO处理操作环节

用户程序主要在Handler业务处理器中，Handler涉及的环节为：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，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）。

按照这种方向来分，前面数据包解码、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。

### 6.5.1 ChannelInboundHandler通道入站处理器

当数据或者信息入站到Netty通道时，Netty将触发入站处理器ChannelInboundHandler所对应的入站API，进行入站操作处理。

ChannelInboundHandler的主要操作，如图6-9所示，具体的介绍如下：

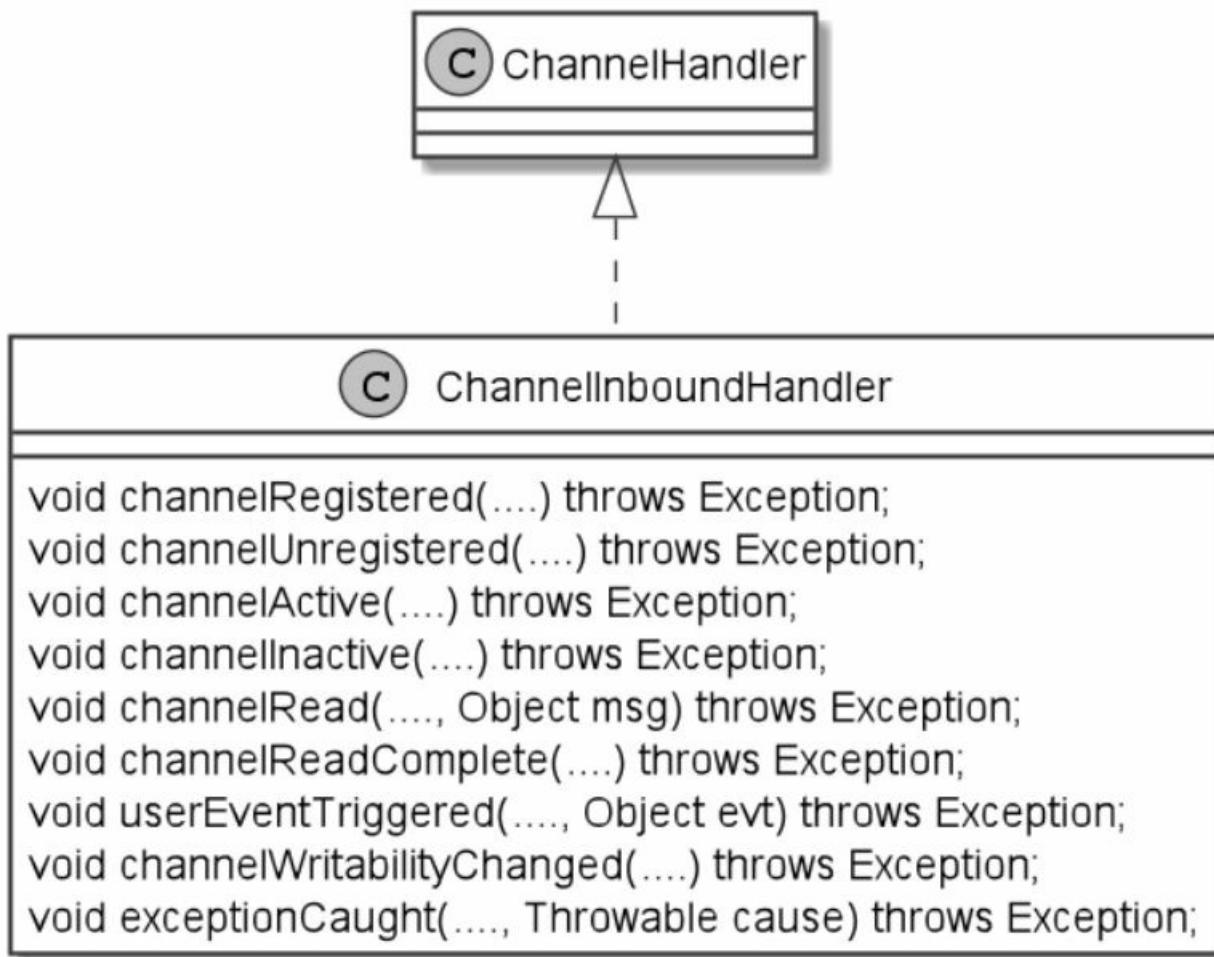


图6-9 ChannelInboundHandler的主要操作

#### 1.channelRegistered

当通道注册完成后，Netty会调用fireChannelRegistered，触发通道注册事件。通道会启动该入站操作的流水线处理，在通道注册过的入站处理器Handler的channelRegistered方法，会被调用到。

#### 2.channelActive

当通道激活完成后，Netty会调用fireChannelActive，触发通道激活事件。通道

会启动该入站操作的流水线处理，在通道注册过的入站处理器Handler的channelActive方法，会被调用到。

### 3.channelRead

当通道缓冲区可读，Netty会调用fireChannelRead，触发通道可读事件。通道会启动该入站操作的流水线处理，在通道注册过的入站处理器Handler的channelRead方法，会被调用到。

### 4.channelReadComplete

当通道缓冲区读完，Netty会调用fireChannelReadComplete，触发通道读完事件。通道会启动该入站操作的流水线处理，在通道注册过的入站处理器Handler的channelReadComplete方法，会被调用到。

### 5.channelInactive

当连接被断开或者不可用，Netty会调用fireChannelInactive，触发连接不可用事件。通道会启动对应的流水线处理，在通道注册过的入站处理器Handler的channelInactive方法，会被调用到。

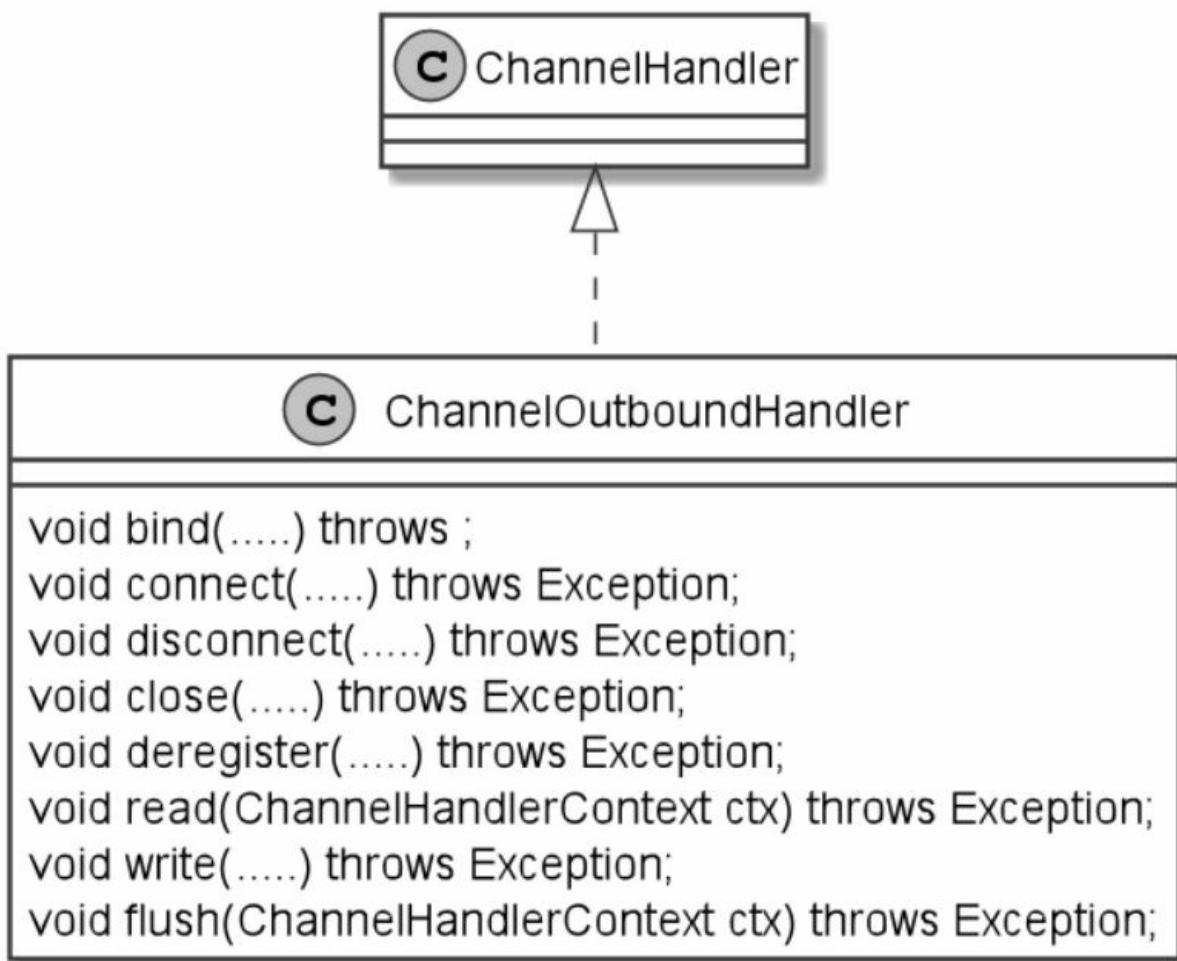
### 6.exceptionCaught

当通道处理过程发生异常时，Netty会调用fireExceptionCaught，触发异常捕获事件。通道会启动异常捕获的流水线处理，在通道注册过的处理器Handler的exceptionCaught方法，会被调用到。注意，这个方法是在通道处理器中ChannelHandler定义的方法，入站处理器、出站处理器接口都继承到了该方法。

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，它的默认实现为ChannelInboundHandlerAdapter，在实际开发中，只需要继承这个ChannelInboundHandlerAdapter默认实现，重写自己需要的方法即可。

## 6.5.2 ChannelOutboundHandler通道出站处理器

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通道出站处理器，完成Netty通道到底层通道的操作。比如说建立底层连接、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6-10所示，具体的介绍如下：



如图6-10 ChannelOutboundHandler的主要操作

再强调一下，出站处理的方向：是通过上层Netty通道，去操作底层Java IO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 IO通道的IP地址绑定。如果使用TCP传输协议，这个方法用于服务器端。

## [2.connect](#)

连接服务端：完成底层Java IO通道的服务器端的连接操作。如果使用TCP传输协议，这个方法用于客户端。

## [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而已，并不是完成实际的数据写入操作。

## [4.flush](#)

腾空缓冲区中的数据，把这些数据写到对端：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通道的服务器端连接。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，例如服务器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，只需要继承这个ChannelOutboundHandlerAdapter默认实现，重写自己需要的方法即可。

### 6.5.3 ChannelInitializer通道初始化处理器

在前面已经讲到，通道和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类——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，使用childHandler()方法设置了一个ChannelInitializer实例：

```
//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) throws Exception {
        // 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。它是何方神圣呢？

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。在父通道调用initChannel()方法时，会将新接收的通道作为参数，传递给initChannel()方法。initChannel()方法内部大致的业务代码是：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。

## 6.5.4 ChannelInboundHandler的生命周期的实践案例

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，它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) throws Exception {
        Logger.info("被调用: handlerAdded()");
        super.handlerAdded(ctx);
    }
    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        Logger.info("被调用: channelRegistered()");
        super.channelRegistered(ctx);
    }
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        Logger.info("被调用: channelActive()");
        super.channelActive(ctx);
    }
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        Logger.info("被调用: channelRead()");
        super.channelRead(ctx, msg);
    }
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        Logger.info("被调用: channelReadComplete()");
        super.channelReadComplete(ctx);
    }
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        Logger.info("被调用: channelInactive()");
        super.channelInactive(ctx);
    }
    @Override
    public void channelUnregistered(ChannelHandlerContext ctx) throws Exception {
        Logger.info("被调用: channelUnregistered()");
        super.channelUnregistered(ctx);
    }
    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
        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入站处理器加入到一个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，会处理从通道到流水线的入站报文——ByteBuf数据包。单元测试的代码如下：

```
package com.crazymakercircle.netty.handler;
//...
public class InHandlerDemoTester {
    @Test
    public void testInHandlerLifeCircle() {
        final InHandlerDemo inHandler = 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，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();
        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

运行上面的测试用例，输出的结果具体如下：

```
[main|InHandlerDemo:handlerAdded]: 被调用: handlerAdded()
[main|InHandlerDemo:channelRegistered]: 被调用: channelRegistered()
[main|InHandlerDemo:channelActive]: 被调用: channelActive()
[main|InHandlerDemo:channelRead]: 被调用: channelRead()
[main|InHandlerDemo:channelReadComplete]: 被调用: channelReadComplete()
[main|InHandlerDemo:channelRead]: 被调用: channelRead()
[main|InHandlerDemo:channelReadComplete]: 被调用: channelReadComplete()
[main|InHandlerDemo:channelInactive]: 被调用: channelInactive()
[main|InHandlerDemo:channelUnregistered]: 被调用: channelUnregistered()
[main|InHandlerDemo:handlerRemoved]: 被调用: handlerRemoved()
```

在讲解上面的方法之前，首先对方法进行分类：（1）生命周期方法，（2）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`中的回调方法的执行顺序为：  
`handlerAdded() → channelRegistered() → channelActive() → 入站方法回调  
→ channelInactive() → channelUnregistered() → handlerRemoved()`。其中，读数据的入站回调为：`channelRead() → channelReadComplete()`；入站方法会多次调用，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线程后，会通过流水线回调所有业务处理器的`channelRegistered()`方法。

(3) `channelActive()`: 当通道激活成功后，会通过流水线回调所有业务处理器的`channelActive()`方法。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NioEventLoop线程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线程解除绑定，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()`: 最后，Netty会移除掉通道上所有的业务处理器，并且回调所有的业务处理器的`handlerRemoved()`方法。

在上面的6个生命周期方法中，前面3个在通道创建的时候被先后回调，后面3个在通道关闭的时候会先后被回调。

除了生命周期的回调，就是入站和出站处理的回调。对于Inhandler入站处理器，有两个很重要的回调方法为：

(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，有一个非常清楚的了解。

上面的入站处理器实践案例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，与入站处理器是大致相同的。不同的是，出站处理器的业务处理方法。

在实践案例的Maven源代码工程中，有一个关于出站处理器的实践案例——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，这里就不再赘述。

## 6.6 详解Pipeline流水线

前面讲到，一条Netty通道需要很多的Handler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Handler业务处理器。首先看一下流水线的入站处理流程。

## 6.6.1 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，在ChannelInitializer通道初始化处理器的initChannel方法中把它们加入到流水线中。三个入站处理器分别为：SimpleInHandlerA、SimpleInHandlerB、SimpleInHandlerC，添加的顺序为A→B→C。实践的代码如下：

```
package com.crazymakercircle.netty.pipeline;
//...
public class InPipeline {
    static class SimpleInHandlerA extends ChannelInboundHandlerAdapter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            Logger.info("入站处理器 A: 被回调 ");
            super.channelRead(ctx, msg);
        }
    }
    static class SimpleInHandlerB extends ChannelInboundHandlerAdapter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            Logger.info("入站处理器 B: 被回调 ");
            super.channelRead(ctx, msg);
        }
    }
    static class SimpleInHandlerC extends ChannelInboundHandlerAdapter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            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);
        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

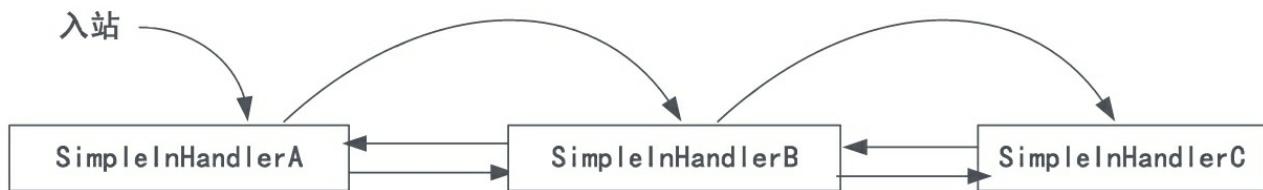
在channelRead()方法中，我们打印当前Handler业务处理器的信息，然后调用父类的channelRead()方法，而父类的channelRead()方法会自动调用下一个inBoundHandler的channelRead()方法，并且会把当前inBoundHandler入站处理器中处理完毕的对象传递到下一个inBoundHandler入站处理器，我们在示例程序中传递的对象都是同一个信息（msg）。

在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。

运行实践案例的代码，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。加在前面的，执行也在前面。具体如图6-11所示。



如图6-11 入站处理器的执行次序

## 6.6.2 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，在ChannelInitializer通道初始化处理器的initChannel方法中，把它们加入到流水线中。三个出站处理器分别为：SimpleOutHandlerA、SimpleOutHandlerB、SimpleOutHandlerC，添加的顺序为A→B→C。实践案例的代码如下：

```
package com.crazymakercircle.netty.pipeline;
//...
public class OutPipeline {
    public class SimpleOutHandlerA extends ChannelOutboundHandlerAdapter {
        @Override
        public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
            Logger.info("出站处理器 A: 被回调");
            super.write(ctx, msg, promise);
        }
    }
    public class SimpleOutHandlerB extends ChannelOutboundHandlerAdapter {
        @Override
        public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
            Logger.info("出站处理器 B: 被回调");
            super.write(ctx, msg, promise);
        }
    }
    public class SimpleOutHandlerC extends ChannelOutboundHandlerAdapter {
        @Override
        public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
            Logger.info("出站处理器 C: 被回调");
            super.write(ctx, msg, promise);
        }
    }
}
@Test
public void testPipelineOutBound() {
    ChannelInitializer<EmbeddedChannel> 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);
    //向通道写一个出站报文(或数据包)
    channel.writeOutbound(buf);
    try {
        Thread.sleep(Integer.MAX_VALUE);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

在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()方法会自动调用下一个outBoundHandler出站处理器的write()方法。这里有个小问题：在OutBoundHandler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。

运行上面的实践案例程序，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 $A \rightarrow B \rightarrow C$ 。从结果可以看出，出站流水处理次序为从后向前： $C \rightarrow B \rightarrow A$ 。最后加入的出站处理器，反而执行在最前面。这一点和`Inbound`入站处理次序是相反的，具体如图6-12所示。

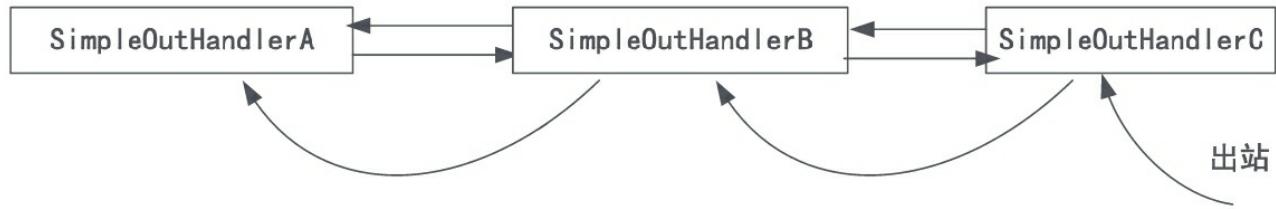


图6-12 出站处理器的执行次序

### 6.6.3 ChannelHandlerContext上下文

不管我们定义的是哪种类型的Handler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Handler业务处理器基类，而是一个新的Netty类型：ChannelHandlerContext通道处理器上下文类。ChannelHandlerContext又是何方神圣呢？

在Handler业务处理器被添加到流水线中时，会创建一个通道处理器上下文ChannelHandlerContext，它代表了ChannelHandler通道处理器和ChannelPipeline通道流水线之间的关联。

ChannelHandlerContext中包含了有许多方法，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，会有同样的出站和入站处理方法，同一个操作出现在不同的类中，功能有何不同呢？如果通过Channel或ChannelPipeline的实例来调用这些方法，它们就会在整条流水线中传播。然而，如果是通过ChannelHandlerContext通道处理器上下文进行调用，就只会从当前的节点开始执行Handler业务处理器，并传播到同类型处理器的下一站（节点）。

Channel、Handler、ChannelHandlerContext三者的关系为：Channel通道拥有一条ChannelPipeline通道流水线，每一个流水线节点为一个ChannelHandlerContext通道处理器上下文对象，每一个上下文中包裹了一个ChannelHandler通道处理器。在ChannelHandler通道处理器的入站/出站处理方法中，Netty都会传递一个Context上下文实例作为实际参数。通过Context实例的实参，在业务处理中，可以获取ChannelPipeline通道流水线的实例或者Channel通道的实例。

## 6.6.4 截断流水线的处理

在入站/出站的过程中，如果由于业务条件不满足，需要截断流水线的处理，不让处理进入下一站，怎么办呢？

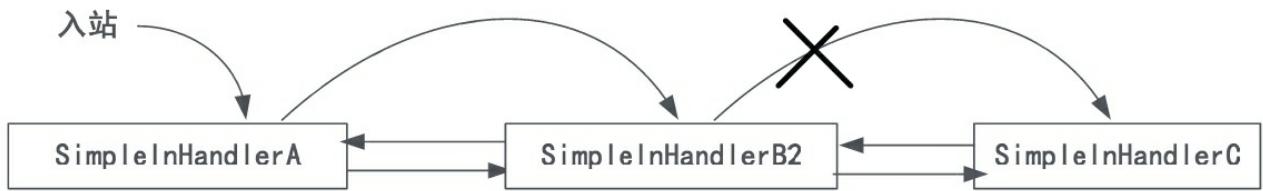
首先以channelRead通道读方法的流程为例，看看如何截断入站处理流程。这里办法是：在channelRead方法中，不再调用父类的channelRead入站方法，它的代码如下：

```
package com.crazymakercircle.netty.pipeline;
//...
public class InPipeline {
    //...省略SimpleInHandlerA、SimpleInHandlerC
    //定义 SimpleInHandlerB2
    static class SimpleInHandlerB2 extends ChannelInboundHandlerAdapter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            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 SimpleInHandlerB2());
                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);
        //....
    }
}
```

同样是3个业务处理器，只是中间的业务处理器SimpleInHandlerB2没有调用父类的super.channelRead方法了。运行的结果如下：

```
[T:main|C:InPipeline$SimpleInHandlerA|F:channelRead] |>入站处理器 A: 被回调
[T:main|C:InPipeline$SimpleInHandlerB2|F:channelRead] |>入站处理器 B: 被回调
```

从运行的结果看出，入站处理器C没有执行到，说明通过没用调用父类的super.channelRead方法，处理流水线被成功地截断了，如图6-13所示。



如图6-13 处理流水线的截断

在channelRead方法中，入站处理传入下一站还有一种方法：调用Context上下文的ctx.fireChannelRead(msg)方法。如果要截断流水线的处理，很显然，就不能调用ctx.fireChannelRead(msg)方法。

上面的channelRead通道读操作流程的截断，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) 也不调用ctx.fireChannelXxx()

如何截断出站处理流程呢？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，会看到截断后抛出的异常，这里就不再赘述。

## 6.6.5 Handler业务处理器的热拔插

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拔插：动态地增加、删除流水线上的业务处理器Handler。主要的Handler热拔插方法声明在ChannelPipeline接口中，如下：

```
package io.netty.channel;
//...
public interface ChannelPipeline
    extends Iterable<Entry<String, ChannelHandler>>
{
    //...
    //在头部增加一个业务处理器，名字由name指定
    ChannelPipelineaddFirst(String name, ChannelHandler handler);
    //在尾部增加一个业务处理器，名字由name指定
    ChannelPipelineaddLast(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
    ChannelPipelineaddBefore(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
    ChannelPipelineaddAfter(String baseName, String name, ChannelHandler handler);
    //删除一个业务处理器实例
    ChannelPipelineremove(ChannelHandler handler);
    //删除一个处理器实例
    ChannelHandlerremove(String handler);
    //删除第一个业务处理器
    ChannelHandlerremoveFirst();
    //删除最后一个业务处理器
    ChannelHandlerremoveLast();
    //...
}
```

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实例。代码如下：

```
package com.crazymakercircle.netty.pipeline;
//...
public class PipelineHotOperateTester {
    static class SimpleInHandlerA extends ChannelInboundHandlerAdapter {
        public void channelRead(ChannelHandlerContextctx, Object msg) throws Exception {
            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() {
        ChannelInitializeri = new ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannelch) {
                ch.pipeline().addLast(new SimpleInHandlerA());
                ch.pipeline().addLast(new SimpleInHandlerB());
                ch.pipeline().addLast(new SimpleInHandlerC());
            }
        };
        EmbeddedChannel channel = new EmbeddedChannel(i);
        ByteBufbuf = 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);
try {
    Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
```

---

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，SimpleInHandlerA已经不再被调用了。

回头看看，Netty的通道初始化处理器——ChannelInitializer，在它的注册回调channelRegistered方法中，就使用了ctx.pipeline().remove(this)，将自己从流水线中删除。ChannelInitializer的源代码，节选如下：

```
package io.netty.channel;
public abstract class ChannelInitializer extends ChannelInboundHandlerAdapter {
    //...
    //抽象方法：通道初始化
    protected abstract void initChannel(Channel var1) throws Exception;
    public final void channelRegistered(ChannelHandlerContextctx) throws Exception {
        //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，就是一条通道只需要做一次初始化的工作。

## 6.7 详解ByteBuf缓冲区

Netty提供了ByteBuf来替代Java NIO的ByteBuffer缓冲区，以操纵内存缓冲区。

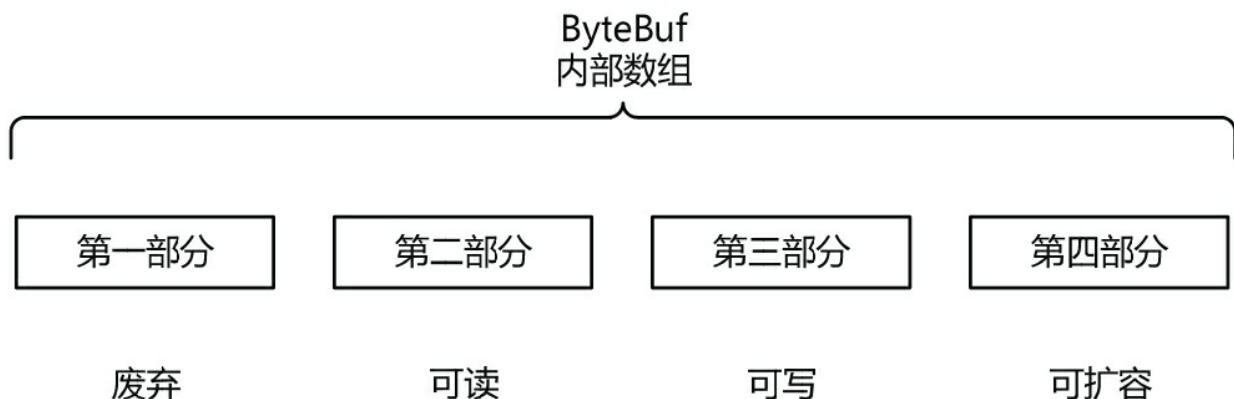
## 6.7.1 ByteBuf的优势

与Java NIO的ByteBuffer相比， ByteBuf的优势如下：

- Pooling（池化，这点减少了内存复制和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
- 扩展性好，例如StringBuffer
- 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

## 6.7.2 ByteBuf的逻辑部分

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6-14所示。



如图6-14 ByteBuf字节容器的内部字节数组

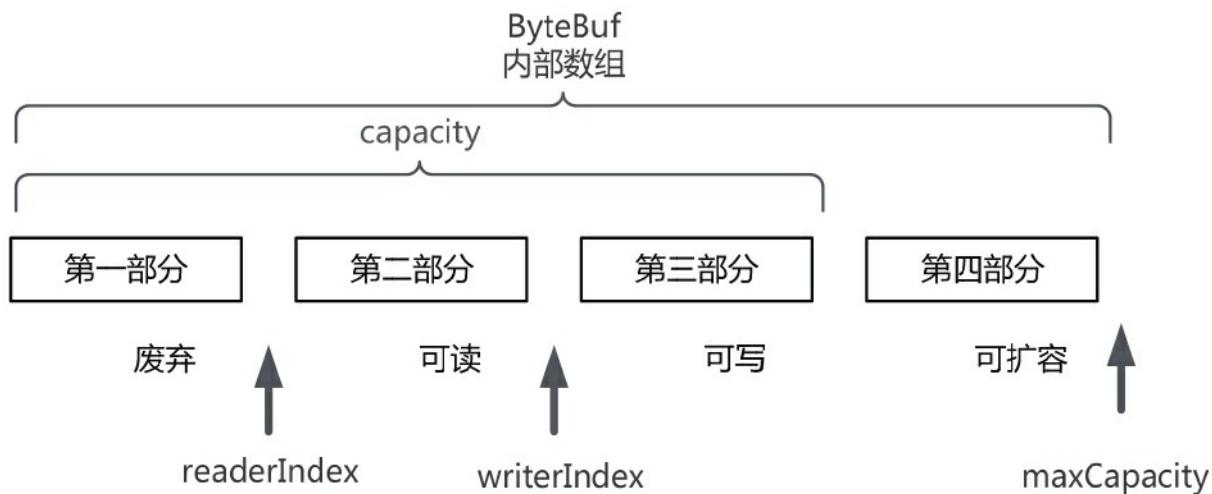
第一个部分是已用字节，表示已经使用完的废弃的无效字节；第二部分是可读字节，这部分数据是ByteBuf保存的有效数据，从ByteBuf中读取的数据都来自这一部分；第三部分是可写字节，写入到ByteBuf的数据都会写到这一部分中；第四部分是可扩容字节，表示的是该ByteBuf最多还能扩容的大小。

### 6.7.3 ByteBuf的重要属性

ByteBuf通过三个整型的属性有效地区分可读数据和可写数据，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）
- writerIndex（写指针）
- maxCapacity（最大容量）

ByteBuf的这三个重要属性，如图6-15所示。



如图6-15 ByteBuf内部的三个重要属性

这三个属性的详细介绍如下：

·`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已经不可写了。`capacity()`是一个成员方法，不是一个成员属性，它表示ByteBuf中可以写入的容量。注意，它不是最大容量`maxCapacity`。

·`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`的值来设定，超过`maxCapacity`就会报错。

## 6.7.4 ByteBuf的三组方法

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，它的值是以下三部分之和：废弃的字节数、可读字节数和可写字节数。

·maxCapacity(): 表示ByteBuf最大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，直到扩容到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：如果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中写数据写不进去的话，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，包含了8大基础数据类型。具体如下：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，包含了8大基础数据类型的设置。具体如下：setByte()、setBoolean()、setChar()、setShort()、setInt()、setLong()、setFloat()、setDouble()。setType系列与writeTYPE系列的不同：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(): 这两个方法一起介绍。前一个方法表示把当前的写指针writerIndex属性的值保存在markedWriterIndex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。

markedWriterIndex属性相当于一个暂存属性，也定义在AbstractByteBuf抽象基类

中。

### 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)`: 读取ByteBuf中的数据。将数据从ByteBuf读取到dst字节数组中，这里dst字节数组的大小，通常等于`readableBytes()`。这个方法也是最为常用的一个方法之一。

·`readType()`: 读取基础数据类型，可以读取8大基础数据类型。具体如下：  
`readByte()`、`readBoolean()`、`readChar()`、`readShort()`、`readInt()`、`readLong()`、  
`readFloat()`、`readDouble()`。

·`getTYPE(TYPE value)` : 读取基础数据类型，并且不改变指针值。具体如下：  
`getByte()`、`getBoolean()`、`getChar()`、`getShort()`、`getInt()`、`getLong()`、`getFloat()`、  
`getDouble()`。`getType`系列与`readTYPE`系列的不同：`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()`: 这两个方法一起介绍。前一个方法表示把当前的读指针ReaderIndex保存在markedReaderIndex属性中。后一个方法表示把保存在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。

## 6.7.5 ByteBuf基本使用的实践案例

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。

这里用了默认的分配器，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，稍候具体详细介绍。

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));
        }
    }
}
```

运行的结果，节选如下：

```
//...
[main|PrintAttribute:print]: after ======动作：分配ByteBuf(9, 100)=====
[main|PrintAttribute:print]: 1.0 isReadable(): false
[main|PrintAttribute:print]: 1.1 readerIndex(): 0
[main|PrintAttribute:print]: 1.2 readableBytes(): 0
[main|PrintAttribute:print]: 2.0 isWritable(): true
[main|PrintAttribute:print]: 2.1 writerIndex(): 0
[main|PrintAttribute:print]: 2.2 writableBytes(): 9
[main|PrintAttribute:print]: 3.0 capacity(): 9
```

```
[main|PrintAttribute:print]: 3.1 maxCapacity(): 100
[main|PrintAttribute:print]: 3.2 maxWritableBytes(): 100
//...
[main|PrintAttribute:print]: after =====动作: 写入4个字节 (1,2,3,4)=====
[main|PrintAttribute:print]: 1.0 isReadable(): true
[main|PrintAttribute:print]: 1.1 readerIndex(): 0
[main|PrintAttribute:print]: 1.2 readableBytes(): 4
[main|PrintAttribute:print]: 2.0 isWritable(): true
[main|PrintAttribute:print]: 2.1 writerIndex(): 4
[main|PrintAttribute:print]: 2.2 writableBytes(): 5
[main|PrintAttribute:print]: 3.0 capacity(): 9
[main|PrintAttribute:print]: 3.1 maxCapacity(): 100
[main|PrintAttribute:print]: 3.2 maxWritableBytes(): 96
//...
[main|PrintAttribute:print]: after =====动作: 取数据ByteBuf=====
[main|PrintAttribute:print]: 1.0 isReadable(): true
[main|PrintAttribute:print]: 1.1 readerIndex(): 0
[main|PrintAttribute:print]: 1.2 readableBytes(): 4
[main|PrintAttribute:print]: 2.0 isWritable(): true
[main|PrintAttribute:print]: 2.1 writerIndex(): 4
[main|PrintAttribute:print]: 2.2 writableBytes(): 5
[main|PrintAttribute:print]: 3.0 capacity(): 9
[main|PrintAttribute:print]: 3.1 maxCapacity(): 100
[main|PrintAttribute:print]: 3.2 maxWritableBytes(): 96
//...
[main|PrintAttribute:print]: after =====动作: 读完ByteBuf=====
[main|PrintAttribute:print]: 1.0 isReadable(): false
[main|PrintAttribute:print]: 1.1 readerIndex(): 4
[main|PrintAttribute:print]: 1.2 readableBytes(): 0
[main|PrintAttribute:print]: 2.0 isWritable(): true
[main|PrintAttribute:print]: 2.1 writerIndex(): 4
[main|PrintAttribute:print]: 2.2 writableBytes(): 5
[main|PrintAttribute:print]: 3.0 capacity(): 9
[main|PrintAttribute:print]: 3.1 maxCapacity(): 100
[main|PrintAttribute:print]: 3.2 maxWritableBytes(): 96
```

可以看到，使用get取数据是不会影响ByteBuf的指针属性值的。由于篇幅原因，这里不仅省略了很多的输出结果，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。

## 6.7.6 ByteBuf的引用计数

Netty的ByteBuf的内存回收工作是通过引用计数的方式管理的。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（注：GC是Garbage Collection的缩写，即Java中的垃圾回收机制），Netty也使用了这种手段来对ByteBuf的引用进行计数。Netty采用“计数器”来追踪ByteBuf的生命周期，一是对Pooled ByteBuf的支持，二是能够尽快地“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。

插个题外话：什么是Pooled（池化）的ByteBuf缓冲区呢？在通信程序的执行过程中，Buffer缓冲区实例会被频繁创建、使用、释放。大家都知道，频繁创建对象、内存分配、释放内存，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？从Netty4版本开始，新增了对象池化的机制。即创建一个Buffer对象池，将没有被引用的Buffer对象，放入对象缓存池中；当需要时，则重新从对象缓存池中取出，而不需要重新创建。

回到正题。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，它的引用为1；每次调用retain()方法，它的引用就加1；每次调用release()方法，就是将引用计数减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，它占用的内存需要回收。在下面的例子中，多次用到了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();
        Logger.info("after retain:" + buffer.refCnt());
        buffer.release();
        Logger.info("after release:" + buffer.refCnt());
        buffer.release();
        Logger.info("after release:" + buffer.refCnt());
        //错误:refCnt: 0, 不能再retain
        buffer.retain();
        Logger.info("after retain:" + buffer.refCnt());
    }
}
```

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。简单地说，在一个方法中，调用了retain，

就应该调用一次release。

```
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？则在缓冲区使用完成后，调用一次release，就是释放一次。例如在Netty流水线上，中间所有的Handler业务处理器处理完ByteBuf之后直接传递给下一个，由最后一个Handler负责调用release来释放缓冲区的内存空间。

当引用计数已经为0，Netty会进行ByteBuf的回收。分为两种情况：（1）Pooled池化的ByteBuf内存，回收方法是：放入可以重新分配的ByteBuf池子，等待下一次分配。（2）Unpooled未池化的ByteBuf缓冲区，回收分为两种情况：如果是堆（Heap）结构缓冲，会被JVM的垃圾回收机制回收；如果是Direct类型，调用本地方法释放外部内存（unsafe.freeMemory）。

## 6.7.7 ByteBuf的Allocator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ByteBufAllocator的两种实现：PoolByteBufAllocator和UnpooledByteBufAllocator。

PoolByteBufAllocator（池化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；这个池化分配器采用了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，它没有把ByteBuf放入池中，每次被调用时，返回一个新的ByteBuf实例；通过Java的垃圾回收机制回收。

为了验证两者的性能，大家可以做一下对比试验：

（1）使用UnpooledByteBufAllocator的方式分配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的情况。

实验的参考结果：在短时间内，可以看到占到10GB多的内存空间，但随着系统的运行，内存空间不断增长，直到整个系统内存被占满而导致内存溢出，最终系统宕机。

（2）把UnpooledByteBufAllocator换成PooledByteBufAllocator，再进行试验，看看服务器的内存使用量的情况。

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都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，可以通过Java系统参数（System Property）的选项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。而在Netty 4.1版本中，默认的分配器为PooledByteBufAllocator。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，可以在Netty程序中设置启动器Bootstrap的时候，将PooledByteBufAllocator设置为默认的分配器。

---

```
ServerBootstrap b = new ServerBootstrap()
//...
//4 设置通道的参数
b.option(ChannelOption.SO_KEEPALIVE, true);
b.option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
```

```
b.childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
//....
```

---

内存管理的策略可以灵活调整，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”！这主要是因为Netty用到了Java的Jemalloc内存管理库。

使用分配器分配ByteBuf的方法有多种。下面列出主要的几种：

```
package com.crazymakercircle.netty.bytebuf;
//...
public class AllocatorTest {
    @Test
    public void showAlloc() {
        ByteBuf buffer = null;
        //方法一： 分配器默认分配初始容量为9，最大容量100的缓冲区
        buffer = ByteBufAllocator.DEFAULT.buffer(9, 100);
        //方法二： 分配器默认分配初始容量为256，最大容量Integer.MAX_VALUE的缓冲区
        buffer = ByteBufAllocator.DEFAULT.buffer();
        //方法三： 非池化分配器，分配基于Java的堆（Heap）结构内存缓冲区
        buffer = UnpooledByteBufAllocator.DEFAULT.heapBuffer();
        //方法四： 池化分配器，分配基于操作系统管理的直接内存缓冲区
        buffer = PooledByteBufAllocator.DEFAULT.directBuffer();
        //....其他方法
    }
}
```

---

如果没有特别的要求，使用第一种或者第二种分配方法分配缓冲区即可。

## 6.7.8 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，如表6-2所示。根据内存的管理方不同，分为堆缓存区和直接缓存区，也就是Heap ByteBuf和Direct ByteBuf。另外，为了方便缓冲区进行组合，提供了一种组合缓存区。

表6-2 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，通过 hasArray 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（使用系统的方法）；在 Java 中操作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	

上面三种缓冲区的类型，无论哪一种，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。

下面对Direct Memory（直接内存）进行一下特别的介绍：

·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的本地内存堆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。

## 6.7.9 三类ByteBuf使用的实践案例

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：

- 创建的方法不同：Heap ByteBuf通过调用分配器的buffer()方法来创建；而Direct ByteBuf的创建，是通过调用分配器的directBuffer()方法。
- Heap ByteBuf缓冲区可以直接通过array()方法读取内部数组；而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Heap堆缓冲，否则就不是。
- Direct ByteBuf要读取缓冲数据进行业务处理，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。

Heap ByteBuf和Direct ByteBuf这两类缓冲区的使用对比，实践案例的代码如下：

```
package com.crazymakercircle.netty.bytebuf;
//...
public class BufferTypeTest {
    final static Charset UTF_8 = Charset.forName("UTF-8");
    //堆缓冲区
    @Test
    public void testHeapBuffer() {
        //取得堆内存
        ByteBuf heapBuf = ByteBufAllocator.DEFAULT.buffer();
        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
    @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
            directBuf.getBytes(directBuf.readerIndex(), array);
            Logger.info(new String(array, UTF_8));
        }
        directBuf.release();
    }
}
```

注意，如果hasArray()返回false，不一定代表缓冲区一定就是Direct ByteBuf直接缓冲区，也有可能是CompositeByteBuf缓冲区。

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）组成的。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。

下面演示一下通过CompositeByteBuf来复用Header，代码如下：

```
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，向CompositeByteBuf对象中增加ByteBuf对象实例，这里调用了addComponents方法。Heap ByteBuf和Direct ByteBuf两种类型都可以增加。如果内部只存在一个实例，则CompositeByteBuf中的hasArray()方法，将返回这个唯一实例的hasArray()方法的值；如果有多个实例，CompositeByteBuf中的hasArray()方法返回false。

调用nioBuffer()方法可以将CompositeByteBuf实例合并成一个新的Java NIO ByteBuffer缓冲区（注意：不是ByteBuf）。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}));
        //合并成一个的缓冲区
        ByteBuffernioBuffer = cbuf.nioBuffer(0, 6);
        byte[] bytes = nioBuffer.array();
        System.out.print("bytes = ");
        for (byte b : bytes) {
            System.out.print(b);
        }
        cbuf.release();
    }
}
```

---

在以上代码中，使用到了Netty中一个非常方便的类——Unpooled帮助类，用它来创建和使用非池化的缓冲区。另外，还可以在Netty程序之外独立使用Unpooled帮助类。

## 6.7.10 ByteBuf的自动释放

在入站处理时，Netty是何时自动创建入站的ByteBuf的呢？

查看Netty源代码，我们可以看到，Netty的Reactor反应器线程会在底层的Java NIO通道读数据时，也就是AbstractNioByteChannel.NioByteUnsafe.read()处，调用ByteBufAllocator方法，创建ByteBuf实例，从操作系统缓冲区把数据读取到Bytebuf实例中，然后调用pipeline.fireChannelRead(byteBuf)方法将读取到的数据包送入到入站处理流水线中。

再看看入站处理时，入站的ByteBuf是如何自动释放的。

### 方式一： TailHandler自动释放

Netty默认会在ChannelPipeline通道流水线的最后添加一个TailHandler末尾处理器，它实现了默认的处理方法，在这些方法中会帮助完成ByteBuf内存释放的工作。

在默认情况下，如果每个InboundHandler入站处理器，把最初的ByteBuf数据包一路往下传，那么TailHandler末尾处理器会自动释放掉入站的ByteBuf实例。

如何让ByteBuf数据包通过流水线一路向后传递呢？

如果自定义的InboundHandler入站处理器继承自ChannelInboundHandlerAdapter适配器，那么可以在InboundHandler的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 DomoHandler 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) throws Exception {  
        ByteBuf byteBuf = (ByteBuf) msg;  
        //...省略ByteBuf的业务处理  
        //自动释放ByteBuf的方法：调用父类的入站方法，将msg向后传递  
        // super.channelRead(ctx,msg);  
    }  
}
```

总体来说，如果自定义的InboundHandler入站处理器继承自ChannelInboundHandlerAdapter适配器，那么可以调用以下两种方法来释放ByteBuf内存：

- (1) 手动释放ByteBuf。具体的方式为调用byteBuf.release()。

(2) 调用父类的入站方法将msg向后传递，依赖后面的处理器释放ByteBuf。具体的方式为调用基类的入站处理方法super.channelRead(ctx,msg)。

```
public class DomoHandler 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) throws Exception {  
        ByteBuf byteBuf = (ByteBuf) msg;  
        //...省略ByteBuf的业务处理  
        //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);  
    }  
}
```

## 方式二： SimpleChannelInboundHandler自动释放

如果Handler业务处理器需要截断流水线的处理流程，不将ByteBuf数据包送入后边的InboundHandler入站处理器，这时，流水线末端的TailHandler末尾处理器自动释放缓冲区的工作自然就失效了。

在这种场景下， Handler业务处理器有两种选择：

- 手动释放ByteBuf实例。
- 继承SimpleChannelInboundHandler， 利用它的自动释放功能。

这里， 我们聚焦的是第二种选择： 看看SimpleChannelInboundHandler是如何自动释放的。

以入站读数据为例， Handler业务处理器必须继承自SimpleChannelInboundHandler基类。并且， 业务处理器的代码必须移动到重写的channelRead0(ctx,msg)方法中。 SimpleChannelInboundHandle类的channelRead等入站处理方法，会在调用完实际的channelRead0方法后， 帮忙释放ByteBuf实例。

如果大家好奇， 想看看SimpleChannelInboundHandler是如何释放ByteBuf的， 那么一起来看看Netty源代码。

截取部分的代码如下所示：

```
public abstract class SimpleChannelInboundHandler<I> extends ChannelInboundHandlerAdapter  
{  
    //基类的入站方法  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
```

```
boolean release = true;
try {
    if (acceptInboundMessage(msg)) {
        @SuppressWarnings("unchecked")
        I imsg = (I) msg;
        //调用实际的业务代码，必须由子类继承，并且提供实现
        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，执行完由子类继承的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，将被彻底释放掉。

再看看出站处理时，Netty是何时释放出站的ByteBuf的呢？

出站缓冲区的自动释放方式：HeadHandler自动释放。在出站处理流程中，申请分配到的ByteBuf主要是通过HeadHandler完成自动释放的。

出站处理用到的Bytebuf缓冲区，一般是要发送的消息，通常由Handler业务处理器所申请而分配的。例如，在write出站写入通道时，通过调用ctx.writeAndFlush(Bytebufmsg)，Bytebuf缓冲区进入出站处理的流水线。在每一个出站Handler业务处理器中的处理完成后，最后数据包（或消息）会来到出站的最后一棒HeadHandler，在数据输出完成后，Bytebuf会被释放一次，如果计数器为零，将被彻底释放掉。

在Netty开发中，必须密切关注Bytebuf缓冲区的释放，如果释放不及时，会造成Netty的内存泄露（Memory Leak），最终导致内存耗尽。

## 6.8 ByteBuf浅层复制的高级使用方式

首先说明一下，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，有切片（slice）浅层复制和整体（duplicate）浅层复制。

## 6.8.1 slice切片浅层复制

ByteBuf的slice方法可以获取到一个ByteBuf的一个切片。一个ByteBuf可以进行多次的切片浅层复制；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，在内部是调用了第二个带参数的slice方法，调用大致方式为：buf.slice(buf.readerIndex(), buf.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。

第二个带参数的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：

```
//...篇幅原因，省略了ByteBuf刚分配后的属性值输出
[main|SliceTest:print]: after =====动作：写入4个字节 (1,2,3,4)=====
[main|SliceTest:print]: 1.0 isReadable(): true
[main|SliceTest:print]: 1.1 readerIndex(): 0
[main|SliceTest:print]: 1.2 readableBytes(): 4
[main|SliceTest:print]: 2.0 isWritable(): true
[main|SliceTest:print]: 2.1 writerIndex(): 4
[main|SliceTest:print]: 2.2 writableBytes(): 5
[main|SliceTest:print]: 3.0 capacity(): 9
[main|SliceTest:print]: 3.1 maxCapacity(): 100
[main|SliceTest:print]: 3.2 maxWritableBytes(): 96
[main|SliceTest:print]: after =====动作：切片 slice=====
[main|SliceTest:print]: 1.0 isReadable(): true
[main|SliceTest:print]: 1.1 readerIndex(): 0
[main|SliceTest:print]: 1.2 readableBytes(): 4
```

```
[main|SliceTest:print]: 2.0 isWritable(): false
[main|SliceTest:print]: 2.1 writerIndex(): 4
[main|SliceTest:print]: 2.2 writableBytes(): 0
[main|SliceTest:print]: 3.0 capacity(): 4
[main|SliceTest:print]: 3.1 maxCapacity(): 4
[main|SliceTest:print]: 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）的值为0。
- writerIndex（写指针）的值为源Bytebuf的readableBytes()可读字节数。
- maxCapacity（最大容量）的值为源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：

- 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。

## 6.8.2 `duplicate`整体浅层复制

和slice切片不同，`duplicate()`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。

### 6.8.3 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，这就会导致一个问题：在源ByteBuf调用release()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，将引用计数减一，这样就不影响源ByteBuf的内存释放。

## 6.9 EchoServer回显服务器的实践案例

### 6.9.1 NettyEchoServer回显服务器的服务器端

前面实现过Java NIO版本的EchoServer回显服务器，在学习了Netty后，这里为大家设计和实现一个Netty版本的EchoServer回显服务器。功能很简单：从服务器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。

首先是服务器端的实践案例，目标为掌握以下知识：

- 服务器端ServerBootstrap的装配和使用。
- 服务器端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的引用计数的查看。

服务器端的ServerBootstrap装配和启动过程，它的代码如下：

```
package com.crazymakercircle.netty.echoServer;
//...
public class NettyEchoServer {
    //...
    public void runServer() {
        //创建反应器线程组
        EventLoopGroupbossLoopGroup = new NioEventLoopGroup(1);
        EventLoopGroupworkerLoopGroup = new NioEventLoopGroup();
        //....省略设置：1 反应器线程组/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(SocketChannelch) throws Exception {
                // 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一个Handler业务处理器
                ch.pipeline().addLast(NettyEchoServerHandler.INSTANCE);
            }
        });
        //.... 省略启动、等待、从容关闭（或称为优雅关闭）等
    }
    //...省略 main方法
}
```

## 6.9.2 共享NettyEchoServerHandler处理器

Netty版本的EchoServerHandler回显服务器处理器，继承自ChannelInboundHandlerAdapter，然后覆盖了channelRead方法，这个方法在可读IO事件到来时，被流水线回调。

这个回显服务器处理器的逻辑分为两步：

第一步，从channelRead方法的msg参数。

第二步，调用ctx.channel().writeAndFlush()把数据写回客户端。

先看第一步，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，为什么呢？实际上，msg的形参类型是由流水线的上一站决定的。大家知道，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，传入到第一个入站处理器；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，必须是Object类型。不过，可以肯定的是，第一个入站处理器的channelRead方法的msg实参类型，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直接内存。大家知道，Java不能直接访问Direct ByteBuf内部的数据，必须先通过getBytes、readBytes等方法，将数据读入Java数组中，然后才能继续在数组中进行处理。

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，如果上一步使用的readBytes，那么这一步就不能直接将msg写回了，因为数据已经被readBytes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。这一步调用了ctx.writeAndFlush，把msg数据写回客户端。也可调用ctx.channel().writeAndFlush()方法。这两个方法在这里的效果是一样的，因为这个流水线上没有任何的出站处理器。

服务器端的入站处理器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) throws Exception {
        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((ChannelFutureListener) -> {
            Logger.info("写回后, msg.refCnt:" + ((ByteBuf) msg).refCnt());
        });
    }
}
```

---

这里的NettyEchoServerHandler在前面加了一个特殊的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。什么叫作Handler共享呢？就是多个通道的流水线可以加入同一个Handler业务处理器实例。而这种操作，Netty默认是不允许的。

但是，很多应用场景需要Handler业务处理器实例能共享。例如，一个服务器处理十万以上的通道，如果一个通道都新建很多重复的Handler实例，就需要上十万以上重复的Handler实例，这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，建议设计成共享的模式：在前面加了一个Netty注解：`@ChannelHandler.Sharable`。

反过来，如果没有加`@ChannelHandler.Sharable`注解，试图将同一个Handler实例添加到多个ChannelPipeline通道流水线时，Netty将会抛出异常。

还有一个隐藏比较深的重点：同一个通道上的所有业务处理器，只能被同一个线程处理。所以，不是`@Sharable`共享类型的业务处理器，在线程的层面是安全的，不需要进行线程的同步控制。而不同的通道，可能绑定到多个不同的EventLoop反应器线程。因此，加上了`@ChannelHandler.Sharable`注解后的共享业务处理器的实例，可能被多个线程并发执行。这样，就会导致一个结果：`@Sharable`共享实例不是线程层面安全的。显而易见，`@Sharable`共享的业务处理器，如果需要操作的数据不仅仅是局部变量，则需要进行线程的同步控制，以保证操作是线程层面安全的。

如何判断一个Handler是否为`@Sharable`共享呢？`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通道流水线中。

NettyEchoServerHandler回显服务器处理器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，它不光是可以用来共享，而且不需要做任何的同步控制。在这里，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INSTANCE实例即可。

最后，揭示一个比较奇怪的问题。

运行程序，大家会看到在写入客户端的工作完成后，ByteBuf的引用计数的值变为0。在上面的代码中，既没有自动释放的代码，也没有手动释放的代码，为什么，引用计数没有了呢？

这个问题，比较有意思，留给大家自行思考。答案，就藏在上文之中，如果确实想不出来也没有找到，可以来疯狂创客圈社群，和大家一起交流，探讨最佳答案。

### 6.9.3 NettyEchoClient客户端代码

其次是客户端的实践案例，目标为掌握以下知识：

- 客户端Bootstrap的装配和使用。
- 客户端NettyEchoClientHandler入站处理器中，接受回写的数据，并且释放内存。
- 有多种方式用于释放ByteBuf，包括：自动释放、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() {
        //创建反应器线程组
        EventLoopGroup workerLoopGroup = new NioEventLoopGroup();
        try {
            //1 设置反应器 线程组
            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) throws Exception {
                    // 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一个Handler业务处理器
                    ch.pipeline().addLast(NettyEchoClientHandler.INSTANCE);
                }
            });
            ChannelFuture f = b.connect();
            f.addListener((ChannelFutureListener) e ->
            {
                if (e.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 tcfo("请输入发送内容:");
            while (scanner.hasNext()) {
                //获取输入的内容
                String next = scanner.next();
                byte[] bytes = (Dateutil.currentTimeMillis() + " >>" + next).getBytes();
                channel.writeAndFlush(bytes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        + next).getBytes("UTF-8");
    //发送ByteBuf
    ByteBuf buffer = channel.alloc().buffer();
    buffer.writeBytes(bytes);
    channel.writeAndFlush(buffer);
    Print.tcfo("请输入发送内容:");
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 从容关闭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，客户端在连接到服务器端成功后不断循环，获取控制台的输入，通过服务器端的通道发送到服务器。

## 6.9.4 NettyEchoClientHandler处理器

客户端的流水线不是空的，还需要装配一个回显处理器，功能很简单，就是接收服务器写过来的数据包，显示在Console控制台上。代码如下：

```
package com.crazymakercircle.netty.echoServer;
import com.crazymakercircle.util.Logger;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
/**
 * create by 尼恩 @ 疯狂创客圈
 */
@ChannelHandler.Sharable
public class NettyEchoClientHandlerAdapter extends ChannelInboundHandlerAdapter {
    public static final NettyEchoClientHandler INSTANCE
        = new NettyEchoClientHandler();
    /**
     * 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) throws Exception {
        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，从服务器端发送过来的ByteBuf，被手动方式强制释放掉了。当然，也可以使用前面介绍的自动释放方式来释放ByteBuf。

## 6.10 本章小结

本章详细介绍了Netty的基本原理：Reactor反应器模式在Netty中的应用，Netty中Reactor反应器、Handler业务处理器、Channel通道以及它们三者之间的相互关系。另外，Netty为了有效地管理通道和Handler业务处理器之间的关系，还引入了一个重要组件——Pipeline流水线。

如果第4章的Reactor反应器模式，大家理解得比较清晰了，那么掌握Netty的基本原理，其实就是一件非常简单的事情。

本章还介绍了Netty的ByteBuf缓冲区的使用，这也是使用Netty需要掌握的一项非常基础的知识。ByteBuf的入门不难，真正用好的话，还是有蛮多学问的，需要不断地积累经验。在疯狂创客圈社群中，就有不少的兄弟碰到过内存耗尽的问题，多半是由于ByteBuf使用不当引起的。

防止内存泄露（Memory Leak），不仅仅是Java的难题，也是Netty的难题，它不仅仅是个技术问题，也是一个经验问题。针对这个问题，疯狂创客圈社群通过博客或者视频的方式，总结了Netty的内存使用以及Netty内存泄露的排查方法。

## 第7章 Decoder与Encoder重要组件

大家知道，Netty从底层Java通道读到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。

本章专门为大家解读，什么是Netty的编码器和解码器。

## 7.1 Decoder原理与实践

什么叫做Netty的解码器呢？

首先，它是一个InBound入站处理器，解码器负责处理“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，进行数据的解码或者格式转换，然后输出（Output）到下一站Inbound入站处理器。

一个标准的解码器将输入类型为ByteBuf缓冲区的数据进行解码，输出一个一个的Java POJO对象。Netty内置了这个解码器，叫作ByteToMessageDecoder，位在Netty的io.netty.handler.codec包中。

强调一下，所有的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ChannelInboundHandler接口。

### 7.1.1 ByteToMessageDecoder解码器

ByteToMessageDecoder是一个非常重要的解码器基类，它是一个抽象类，实现了解码的基础逻辑和流程。ByteToMessageDecoder继承自ChannelInboundHandlerAdapter适配器，是一个入站处理器，实现了从ByteBuf到Java POJO对象的解码功能。

ByteToMessageDecoder解码的流程，大致如图7-1所示，具体可以描述为：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。

查看Netty源代码，我们会惊奇地发现：ByteToMessageDecoder仅仅是个抽象类，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的解码方法名为decode。通过源代码我们可以发现，decode方法只是提供了一个抽象方法，也就是说，decode方法中的具体解码过程，ByteToMessageDecoder没有具体的实现。换句话说，如何将Bytebuf数据变成Object数据，需要子类去完成，父类不管。

总之，作为解码器的父类，ByteToMessageDecoder仅仅提供了一个流程性质的框架：它仅仅将子类的decode方法解码之后的Object结果，放入自己内部的结果列表List<Object>中，最终，父类会负责将List<Object>中的元素，一个一个地传递给下一个站。哦，不对！父类还没有那么“勤快”，而是将子类的Object结果放入父类的List<Object>列表，也是交由子类的decode方法完成的。

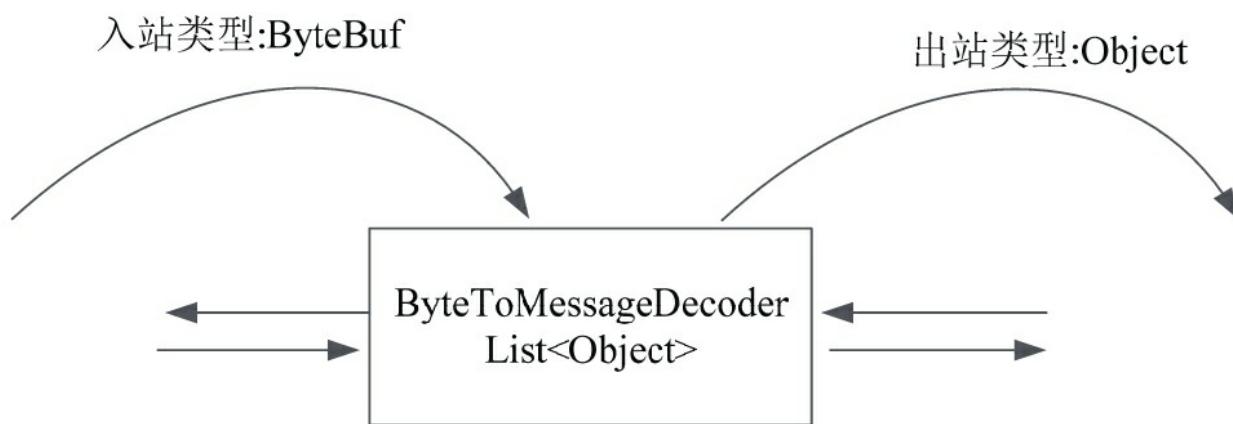


图7-1 ByteToMessageDecoder解码的流程

如果要实现一个自己的解码器，首先继承ByteToMessageDecoder抽象类。然

后，实现其基类的decode抽象方法。将解码的逻辑，写入此方法。总体来说，如果要实现一个自己的ByteBuf解码器，流程大致如下：

(1) 首先继承ByteToMessageDecoder抽象类。

(2) 然后实现其基类的decode抽象方法。将ByteBuf到POJO解码的逻辑写入此方法。将Bytebuf二进制数据，解码成一个一个的Java POJO对象。

(3) 在子类的decode方法中，需要将解码后的Java POJO对象，放入decode的List<Object>实参中。这个实参是ByteToMessageDecoder父类传入的，也就是父类的结果收集列表。

在流水线的过程中，ByteToMessageDecoder调用子类decode方法解码完成后，会将List<Object>中的结果，一个一个地分开传递到下一站的Inbound入站处理器。

## 7.1.2 自定义Byte2IntegerDecoder整数解码器的实践案例

下面是一个小小的实践案例：整数解码器。

其功能是，将ByteBuf缓冲区中的字节，解码成Integer整数类型。按照前面的流程，大致的步骤为：

(1) 定义一个新的整数解码器——Byte2IntegerDecoder类，让这个类继承Netty的ByteToMessageDecoder字节码解码抽象类。

(2) 实现父类的decode方法，将ByteBuf缓冲区数据，解码成以一个一个的Integer对象。

(3) 在decode方法中，将解码后得到的Integer整数，加入到父类传递过来的List<Object>实参中。

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);
        }
    }
}
```

上面实践案例程序的decode方法中的逻辑大致如下：

首先，Byte2IntegerDecoder解码器继承自ByteToMessageDecoder；

其次，在其实现的decode方法中，通过ByteBuf的readInt( )实例方法，从in缓冲区读取到整数，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基类所传入的List<Object>列表参数中；

最后，它不断地循环，不断地解码，并且不断地添加到List<Object>结果列表中。

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已经完成了。

如何使用这个自定义的Byte2IntegerDecoder解码器呢？

首先，需要将其加入到通道的流水线中。其次，由于解码器的功能仅仅是完成ByteBuf的解码，不做其他的业务处理，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对象后，完成具体的业务处理。

这里编写一个简单的业务处理器IntegerProcessHandler，用于处理Byte2IntegerDecoder解码之后的Java Integer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。实践案例的代码如下：

```
package com.crazymakercircle.netty.decoder;
//...
public class IntegerProcessHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        Integer integer = (Integer) msg;
        Logger.info("打印出一个整数: " + integer);
    }
}
```

至此，已经编写了Byte2IntegerDecoder和IntegerProcessHandler这两个自己的入站处理器：一个负责解码，另外一个负责业务处理。

最终，如何测试这两个入站处理器呢？使用EmbeddedChannel嵌入式通道，编写一个测试实例，完整的代码如下：

```
package com.crazymakercircle.netty.decoder;
//...
public class Byte2IntegerDecoderTester {
    /**
     * 整数解码器的使用实例
     */
    @Test
    public void testByteToIntegerDecoder() {
        ChannelInitializer<EmbeddedChannel> 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);
        }
        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在测试用例中，新建了一个EmbeddedChannel嵌入式通道实例，将两个自己的入站处理器Byte2IntegerDecoder和IntegerProcessHandler加入通道的流水线上。

请注意先后次序：Byte2IntegerDecoder解码器在前、IntegerProcessHandler整数处理器在后。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，向嵌入式通道EmbeddedChannel写入100次ByteBuf入站缓冲；每一次写入仅仅包含一个整数。

EmbeddedChannel的writeInbound方法模拟入站数据，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，然后逐个地输出到控制台上。运行测试实例，部分的输出结果如下：

```
//....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
//....省略部分输出
```

强烈建议大家参考以上代码，自行实践一下以上解码器，并仔细分析一下运行的结果。总之，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，应该有比较清楚地了解了。

还可以仿照这个例子，实现除了整数解码器之外，Java基本数据类型的解码器：Short、Char、Long、Float、Double等。

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。问题来了，ByteBuf缓冲区由谁负责进行引用计数和释放管理的呢？

其实，基类ByteToMessageDecoder负责解码器的ByteBuf缓冲区的释放工作，它会调用ReferenceCountUtil.release(in)方法，将之前的ByteBuf缓冲区的引用数减1。这个工作是自动完成的，大家不用担心。

也有同学会问：如果这个ByteBuf被释放了，在后面还需要用到，怎么办呢？可以在decode方法中调用一次ReferenceCountUtil.retain(in)来增加一次引用计数。

### 7.1.3 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，如果有足够的字节，才进行整数的读取。这种长度的判断，是否可以由Netty帮忙来完成呢？答案是：使用Netty的ReplayingDecoder类可以省去长度的判断。

ReplayingDecoder类是ByteToMessageDecoder的子类。其作用是：

- 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，如果没有足够的字节，则会停止解码。

使用ReplayingDecoder基类，编写整数解码器，则可以不用进行长度检测。

改写上一个的整数解码器，继承ReplayingDecoder类，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类实现一个解码器，就不用编写长度判断的代码。ReplayingDecoder进行长度判断的原理，其实很简单：它的内部定义了一个新的二进制缓冲区类，对ByteBuf缓冲区进行了装饰，这个类名为ReplayingDecoderBuffer。该装饰器的特点是：在缓冲区真正读数据之前，首先进行长度的判断：如果长度合格，则读取数据；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基类的关键技术就是偷梁换柱，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的值，它的直接类型并不是原始的ByteBuf类型，而是ReplayingDecoderBuffer类型。

这一点做得非常精彩，且看如下解释：

`ReplayingDecoderBuffer`类型，首先是一个内部的类，其次它继承了`ByteBuf`类型，包装了`ByteBuf`类型的大部分读取方法。`ReplayingDecoderBuffer`类型的读取方法与`ByteBuf`类型的读取方法相比，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，则抛出异常。这个异常会反过来被`ReplayingDecoder`基类所捕获，将解码工作停掉。

`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。

### 7.1.4 整数的分包解码器的实践案例

前面讲到，底层通信协议是分包传输的，一份数据可能分几次达到对端。发送端出去的包在传输过程中会进行多次的拆分和组装。接收端所收到的包和发送端所发送的包不是一模一样的，具体如图7-2所示：在发送端发出4个字符串，Netty NIO接收端可能只是接收到了3个ByteBuf数据缓冲。

在Java OIO流式传输中，不会出现这样的问题，因为它的策略是：不读到完整的信息，就一直阻塞程序，不向后执行。但是，在Java的NIO中，由于NIO的非阻塞性，就会出现图7-2这样的问题。怎样保证一次性读取到完整的数据，就成了一个大问题。

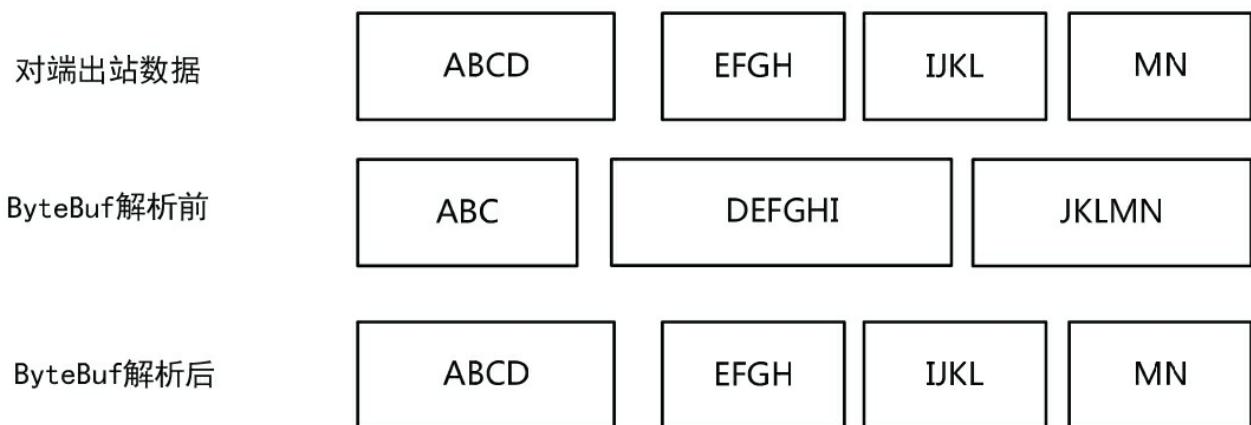


图7-2 通道接收到的ByteBuf数据包和发送端发送的数据包没有完全一致

我们知道，Netty接收到的数据都可以通过解码器进行解码。那么，Netty通过什么样的解码器对图7-2中的3个ByteBuf数据缓冲数据进行解码，而后得到和发送端一模一样的4个字符串，是怎么办到的呢？

对于上面这个问题，还是可以使用ReplayingDecoder来解决。前文讲到，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，ReplayingDecoder会结束解析，直到可读数据是足够的。这一切都是在ReplayingDecoder内部进行，它是通过和缓冲区装饰器类ReplayingDecoderBuffer相互配合完成的，根本就不需要用户程序来操心。

图7-2所展示的是字符串接收时的错乱问题，完全可以通过继承ReplayingDecoder基类来解决。但是，一上来就实现字符串的解码和纠正，相对比较复杂些。在此之前，作为铺垫，我们先看一个简单点的例子：解码整数序列，并且将它们两两一组进行相加。

要完成以上的例子，需要用到ReplayingDecoder一个很重要的属性——state成员属性。该成员属性的作用就是保存当前解码器在解码过程中的当前阶段。在Netty源

代码中，该属性的定义具体如下：

```
public abstract class ReplayingDecoder<S> extends ByteToMessageDecoder {  
    //...省略不相干的代码  
    //重要的成员属性，表示阶段，类型为泛型，默认为Object  
    private S state;  
    //默认的构造器  
    protected ReplayingDecoder() {  
        this((Object)null);  
    }  
    //重载的构造器  
    protected ReplayingDecoder(S initialState) {  
        //初始化内部的ByteBuf缓冲装饰器类  
        this.replayable = new ReplayingDecoderByteBuf();  
        //读断点指针，默认为-1  
        this.checkpoint = -1;  
        //状态state的默认值为null  
        this.state = initialState;  
    }  
    //...省略不相干的方法  
}
```

上一小节，在定义的整数解码实例中，使用的是默认的无参数构造器，也就是说，state的值为null，泛型实参类型，为默认的Object。总之，就是没有用到state属性。

这一小节，就需要用到state成员属性了。为什么呢？整个解码工作通过一次解码不能完成。要完成两个整数相加就需要解码两次，每一次解码只能解码出一个整数。只有两个整数都得到之后，然后求和，整个解码的工作才算完成。

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2个整数，并把这两个数据相加之和作为解码的结果。代码如下：

```
package com.crazymakercircle.netty.decoder;  
//...  
public class IntegerAddDecoder  
    extends ReplayingDecoder<IntegerAddDecoder.Status> {  
    enum Status {  
        PARSE_1, PARSE_2  
    }  
    private int first;  
    private int second;  
    public IntegerAddDecoder() {  
        //在构造函数中，需要初始化父类的state 属性，表示当前阶段  
        super(Status.PARSE_1);  
    }  
    @Override  
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,  
        List<Object> out) throws Exception {  
        switch (state()) {  
            case PARSE_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，并且设置“读断点指针”为当前的读取位置  
                checkpoint(Status.PARSE_2);  
                break;  
            case PARSE_2:  
                second = in.readInt();  
                Integer sum = first + second;  
                out.add(sum);  
        }  
    }  
}
```

```
        checkpoint(Status.PARSE_1);
        break;
    default:
        break;
    }
}
```

IntegerAddDecoder类继承了ReplayingDecoder<IntegerAddDecoder.Status>, 后面的泛型实参为自定义的状态类型, 是一个enum枚举类型, 这里的读取有两个阶段:

- (1) 第一个阶段读取前面的整数。
- (2) 第二个阶段读取后面的整数, 然后相加。

状态的值保持在父类的成员变量state中。前文说明过: 该成员变量保存当面解码的阶段, 需要在构造函数中进行初始化。在上面的子类构造函数中, 调用了super(Status.PARSE\_1)对state进行了初始化。

在IntegerAddDecoder类中, 每一次decode方法中的解码, 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 (Status) 方法, 把当前的状态设置为新的Status值。这个值保存在ReplayingDecoder的state属性中。

严格来说, checkpoint (Status) 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) 还有一个非常大的作用, 就是设置“读断点指针”。

什么是ReplayingDecoder的“读断点指针”呢?

“读断点指针”是ReplayingDecoder类的另一个重要的成员, 它保存着装饰器内部ReplayingDecoderBuffer成员的起始读指针, 有点儿类似于mark标记。当读数据时, 一旦可读数据不够, ReplayingDecoderBuffer在抛出ReplayError异常之前, ReplayingDecoder会把读指针的值还原到之前的checkpoint (IntegerAddDecoder.Status) 方法设置的“读断点指针” (checkpoint)。于是乎, 在ReplayingDecoder下一次读取时, 还会从之前设置的断点位置开始。

checkpoint (IntegerAddDecoder.Status) 方法, 仅仅从参数上去看比较奇怪, 参数为要设置的阶段, 但是它的功能却又包含了“读断点指针”的设置。

总结一下, 在这个IntegerAddDecoder的使用实例中, 解码器保持了以下状态信

息：

- (1) 当前通道的读取阶段，是Status.PARSE\_1或者Status.PARSE\_2。
- (2) 每一次读取，还要保持当前“读断点指针”，便于在可读数据不足时进行恢复。

因此，IntegerAddDecoder是有状态的，不能在不同的通道之间进行共享。更加进一步说，ReplayingDecoder类型和其所有的子类都需要保存状态信息，都有状态的，都不适合在不同的通道之间共享。

至此，IntegerAddDecoder已经介绍完了。

最后，如何使用上面的IntegerAddDecoder解码器呢？

具体的测试实例和前面的Byte2IntegerDecoder使用的实例大致相同，由于篇幅的限制，这里就不再赘述。大家可以在源代码包中查看，具体的类名为Byte2IntegerReplayDecoderTester。

## 7.1.5 字符串的分包解码器的实践案例

到目前为止，通过前面的整数分包传输，对ReplayingDecoder的分阶段解码，大家应该有了一个完整的了解。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。有所不同的是：整数的长度是固定的，目前在Java中是4个字节；而字符串的长度不是固定的，是可变长度的，这就是一个小小的难题。

如何获取字符串的长度信息呢？

这个问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Header-Content内容传输协议：

- (1) 在协议的Head部分放置字符串的字节长度。Head部分可以用一个整型int来描述即可。
- (2) 在协议的Content部分，放置的则是字符串的字节数组。

后面会专门介绍，在实际的传输过程中，一个Header-Content内容包，在发送端会被编码成为一个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，该如何解码成为最初的ByteBuf内容发送包，来获得Header-Content内容分包呢？

不用急，采用ReplayingDecoder解码器即可。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，它的代码大致如下：

---

```
package com.crazymakercircle.netty.decoder;
//...
public class StringReplayDecoder
    extends ReplayingDecoder<StringReplayDecoder.Status> {
    //每一个上层Header-Content内容的读取阶段
    enum Status {
        PARSE_1, PARSE_2
    }
    private int length;
    private byte[] inBytes;
    //在构造函数中，需要初始化父类的state 属性，表示当前阶段
    public StringReplayDecoder() {
        super(Status.PARSE_1);
    }
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                          List<Object> out) throws Exception {
        switch (state()) {
            case PARSE_1:
                //第一步，从装饰器ByteBuf中读取长度
                length = in.readInt();
                inBytes = new byte[length];
                // 进入第二步，读取内容
                // 并且设置“读断点指针”为当前的读取位置，同时设置下一个阶段
                checkpoint(Status.PARSE_2);
                break;
            case PARSE_2:
```

```
//第二步，从装饰器ByteBuf中读取内容数组
in.readBytes( inBytes, 0, length);
out.add(new String(inBytes, "UTF-8"));
// 第二步解析成功,
// 进入第一步，读取下一个字符串的长度
// 并且设置“读断点指针”为当前的读取位置，同时设置下一个阶段
checkpoint(Status.PARSE_1);
break;
default:
break;
}
}
}
```

---

在StringReplayDecoder类中，每一次decode方法中的解码分为两个步骤：

第1步骤，解码出一个字符串的长度。

第2步骤，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，每个阶段一完成，就通过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，并且输出到Console控制台上。新业务处理器名称为StringProcessHandler，具体如下：

```
package com.crazymakercircle.netty.decoder;
//...
public class StringProcessHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        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两个自己的入站处理器：一个负责字符串解码，另外一个负责字符串输出。

如何使用这两个入站处理器呢？编写一个测试实例，完整的代码如下：

```
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<EmbeddedChannel>i = new ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannel ch) {
                ch.pipeline().addLast(new StringReplayDecoder());
                ch.pipeline().addLast(new StringProcessHandler());
            }
        };
        EmbeddedChannel channel = new EmbeddedChannel(i);
        byte[] bytes =content.getBytes(Charset.forName("utf-8"));
        for (int j = 0; j < 100; j++) {

```

```
//1-3之间的随机数
int random = RandomUtil.randInMod(3);
ByteBuf buf = Unpooled.buffer();
buf.writeInt(bytes.length * random);
for (int k = 0; k < random; k++) {
    buf.writeBytes(bytes);
}
channel.writeInbound(buf);
}
try {
    Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
```

---

在测试用例中，新建了一个EmbeddedChannel嵌入式通道实例，将两个自己的入站处理器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；每一个ByteBuf缓冲仅仅包含一个字符串。EmbeddedChannel嵌入式通道接收到入站数据后，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：将接到的二进制字节解码成一个一个的字符串，然后逐个地输出到控制台上。

---

```
//...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！
//...部分输出省略
```

---

小结：通过ReplayingDecoder解码器，可以正确地解码分包后的ByteBuf数据包。但是，在实际的开发中，不太建议继承这个类，原因是：

- (1) 不是所有的ByteBuf操作都被ReplayingDecoderBuffer装饰类所支持，可能有些ByteBuf操作在ReplayingDecoder子类的decode实现方法中被使用时就会抛出ReplayError异常。
- (2) 在数据解析逻辑复杂的应用场景，ReplayingDecoder在解析速度上相对较差。

原因是什么呢？在ByteBuf中长度不够时，ReplayingDecoder会捕获一个ReplayError异常，这时会把ByteBuf中的读指针还原到之前的读断点指针（checkpoint），然后结束这次解析操作，等待下一次IO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，如果解析过程是一个消耗CPU的操作，那么这对CPU是个大负担。

所以，ReplayingDecoder更多的是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，建议使用在前文介绍的解码器ByteToMessageDecoder或者其子

类（后文介绍），它们会更加合适。

继承ByteToMessageDecoder基类，实现Header-Content协议传输分包的字符串内容解码器，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) throws Exception {
        //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()设置回滚点
        //回滚点为消息头的readIndex读指针位置
        buf.markReaderIndex();
        int length = buf.readInt();
        //从缓冲区中读出消息头的大小，这会使得readIndex读指针前移
        //剩余长度不够消息体，重置读指针
        if (buf.readableBytes() < length) {
            //读指针回滚到消息头的readIndex位置处，未进行状态的保存
            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()记录当前的位置指针，当可读内容不够，也就是buf.readableBytes()<length时，需要调用buf.resetReaderIndex()方法将读指针回滚到旧的读起始位置。

表面上，ByteToMessageDecoder基类是无状态的，它不像ReplayingDecoder，需要使用状态位来保存当前的读取阶段。但是，实际上，ByteToMessageDecoder也是有状态的。为什么呢？

在ByteToMessageDecoder的内部，有一个二进制字节的累积器cumulation，它用来保存没有解析完的二进制内容。所以，ByteToMessageDecoder及其子类是有状态的业务处理器，也就是说，它不能共享，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。

## 7.1.6 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。有一个问题：是否存在一些解码器，将一种POJO对象解码成另外一种POJO对象呢？答案是：存在的。

与前面不同的是，在这种应用场景下的Decoder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>。这个实参的作用就是指定入站消息Java POJO类型。

这里有个问题：为什么MessageToMessageDecoder<I>需要指定入站数据的类型，而在前面，使用ByteToMessageDecoder解码ByteBuf的时候，不需要指定泛型实参？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。但是，MessageToMessageDecoder<I>的入站消息的类型是不明确的，可以是任何的POJO类型，所以需要指定。

MessageToMessageDecoder类也是一个入站处理器，也有一个decode抽象方法。decode的具体解码的逻辑需要子类去实现。

下面通过实现一个整数Integer到字符串String的解码器，演示一下MessageToMessageDecoder的使用。此解码器的具体功能是将整数转成字符串，然后输出到下一站。代码很简单，如下所示：

```
package com.crazymakercircle.netty.decoder;
//...
public class Integer2StringDecoder extends
MessageToMessageDecoder<Integer> {
    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg,
                      List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

这里定义的Integer2StringDecoder新类，继承自MessageToMessageDecoder基类。基类泛型实参为Integer，明确了入站的数据类型为Integer。在decode方法中，将整数转成字符串，再加入到一个List列表参数中即可。这个List实参是由父类在调用时传递过来的。在子类decode方法处理完成后，父类会将这个List实例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。具体的测试实例和前面的StringReplayDecoder实例大致相同，由于篇幅的限制，这里就不再赘述。大家可以在源代码包中查看，具体的类名为Integer2StringDecoderTester。

## 7.2 开箱即用的Netty内置Decoder

Netty提供了不少开箱即用的Decoder解码器，在一般情况下，能满足很多编解码应用场景的需求，这为大家省去了开发Decoder的时间。下面将几个比较基础的解码器梳理一下，大致如下：

### （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个字节。

在这种场景下，只需要把这个解码器加到流水线中，它会把入站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。补充说明一下：这里所指的一个数据包，在Netty中就是一个ByteBuf实例。注：数据帧（Frame），本书也通称为数据包。

### （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）作为数据包的边界分割符。

如果每个接收到的数据包，都以换行符/回车换行符作为分隔。在这种场景下，只需要把这个解码器加到流水线中，Netty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。解码的时候，会按照这个长度进行原始数据包的提取。

这种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。

## 7.2.1 LineBasedFrameDecoder解码器

在前面字符串分包解码器中，内容是按照Header-Content协议进行传输的。如果不使用Header-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的、使用换行符分割字符串的解码器，它的名字为LineBasedFrameDecoder，它也是一个最为基础的Netty内置解码器。这个解码器的工作原理很简单，它依次遍历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。当然，这个ByteBuf数据包，也就是解码后的那行字符串的二进制字节码。

LineBasedFrameDecoder支持配置一个最大长度值，表示一行最大能包含的字节数。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() {
        try {
            ChannelInitializer<EmbeddedChannel> 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++) {
                //1-3之间的随机数
                int random = RandomUtil.randInMod(3);
                ByteBuf buf = Unpooled.buffer();
                for (int k = 0; k < random; k++) {
                    buf.writeBytes(content.getBytes("UTF-8"));
                }
                buf.writeBytes(spliter.getBytes("UTF-8"));
                channel.writeInbound(buf);
            }
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
}
```

在这个示例程序中，向通道写入100个入站数据包，每一个入站包都以“\r\n”回车换行符作为结束。通道的LineBasedFrameDecoder解码器会将“\r\n”作为分割符，分

割出一个一个的入站ByteBuf，然后发送给StringDecoder。StringDecoder的作用是将ByteBuf二进制数据转成字符串。最后，字符串被发送到StringProcessHandler业务处理器，由它负责将字符串展示出来。

至此，LineBasedFrameDecoder就演示完毕，它仅仅是Netty中的一个非常简单  
的数据包解码器。

## 7.2.2 DelimiterBasedFrameDecoder解码器

DelimiterBasedFrameDecoder解码器不仅可以使用换行符，还可以将其他的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,       //解码后数据包是否去掉分隔符，一般选择是
    ByteBuf delimiter)           //分隔符
{
    //.... 构造器的源代码，省略
}
```

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。下面是一个实践案例的小示例程序。

```
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(spliter2.getBytes("UTF-8"));
            ChannelInitializer
                = new ChannelInitializer<EmbeddedChannel>() {
                    protected void initChannel(EmbeddedChannelch) {
                        ch.pipeline().addLast(
                            new DelimiterBasedFrameDecoder(1024, true, delimiter));
                        ch.pipeline().addLast(new StringDecoder());
                        ch.pipeline().addLast(new StringProcessHandler());
                    }
                };
            EmbeddedChannel channel = new EmbeddedChannel(i);
            for (int j = 0; j < 100; j++) {
                //1-3之间的随机数
                int random = RandomUtil.randInMod(3);
                ByteBufbuf = Unpooled.buffer();
                for (int k = 0; k < random; k++) {
                    buf.writeBytes(content.getBytes("UTF-8"));
                }
                buf.writeBytes(spliter2.getBytes("UTF-8"));
                channel.writeInbound(buf);
            }
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
}
```

### 7.2.3 LengthFieldBasedFrameDecoder解码器

在Netty的开箱即用解码器中，最为复杂的是自定义长度数据包解码器——LengthFieldBasedFrameDecoder。它的难点在于参数比较多，也比较难以理解。同时它又比较常用，因而下面对它进行重点介绍。

LengthFieldBasedFrameDecoder解码器，它的名字可以翻译为“长度字段数据包解码器”。传输内容中的LengthField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Header-Content协议的内容传输，尽量用内置的LengthFieldBasedFrameDecoder来解码。

LengthFieldBasedFrameDecoder一个简单的使用示例如下：

```
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使用实例 1
     */
    @Test
    public void testLengthFieldBasedFrameDecoder1() {
        try {
            final LengthFieldBasedFrameDecoderspliter =
                new LengthFieldBasedFrameDecoder(1024, 0, 4, 0, 4);
            ChannelInitialzieri = new ChannelInitialzier<EmbeddedChannel>() {
                protected void initChannel(EmbeddedChannelch) {
                    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++) {
                ByteBufbuf = 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();
        }
    }
}
```

上面的示例程序用到了一个LengthFieldBasedFrameDecoder构造器，具体如下：

```
public LengthFieldBasedFrameDecoder(
    int maxFrameLength, //发送的数据包最大长度
```

```
int lengthFieldOffset,      //长度字段偏移量  
int lengthFieldLength,     //长度字段自己占用的字节数  
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个字节。

(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的字节数组中的下标值。

(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**: 长度的矫正值。这个参数最为难懂。在传输协议比较复杂的情况下，例如包含了长度字段、协议版本号、魔数等等。那么，解码时，就需要进行长度矫正。长度矫正值的计算公式为：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，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，上面的示例程序中，前面有4个节点的长度字段，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。

前面的示例程序中，自定义长度解码器的构造参数值如下：

```
LengthFieldBasedFrameDecoderspliter = new  
    LengthFieldBasedFrameDecoder(1024, 0, 4, 0, 4);
```

---

第1个参数maxFrameLength设置为1024，表示数据包的最大长度为1024。

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个字节，即表示内容长度的值占用数据包的4个字节。

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，在上面示例程序中实际的值为：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个字节的数据。

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，对应于第一个传输的数据包，下面给出一个简单的字节图：长度字段4个字节，内容Content字段52个字节，整个数据包56个字节。具体如图7-3所示。

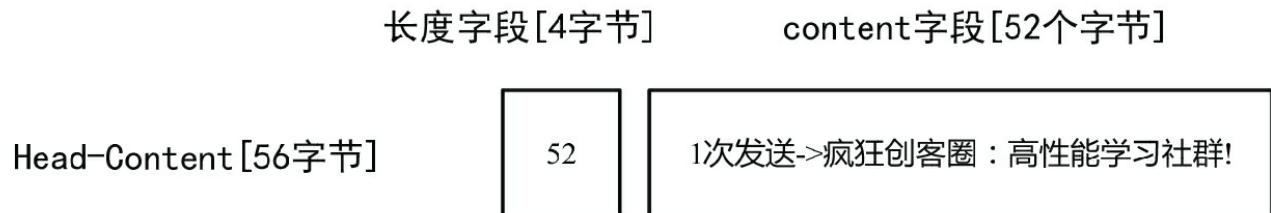


图7-3 Head-Content协议的示意图

## 7.2.4 多字段Head-Content协议数据帧解析的实践案例

Head-Content协议是最为简单的内容传输协议。而在实际使用过程中，则没有那么简单，除了长度和内容，在数据包中还可能包含了其他字段，例如，包含了协议版本号，如图7-4所示。

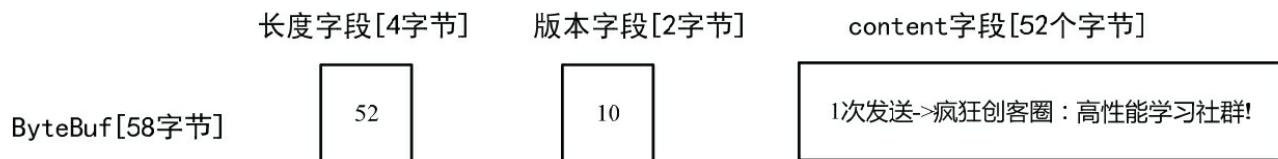


图7-4 包含协议版本号的Head-Content协议的示意图

那么，使用LengthFieldBasedFrameDecoder解码器，解析这一个协议版本号Head-Content协议，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个字节。

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实例中的值为4，表示长度字段的长度为4个字节。

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Content内容的字节数组时，抛弃最前面的6个字节数据。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使用实例 2
     */
    @Test
    public void testLengthFieldBasedFrameDecoder2() {
        try {
            final LengthFieldBasedFrameDecoderspliter =
                new LengthFieldBasedFrameDecoder(1024, 0, 4, 2, 6);
            ChannelInitializeli = new ChannelInitializer<EmbeddedChannel>() {
                protected void initChannel(EmbeddedChannelch) {
                    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++) {
    ByteBufbuf = 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();
}
}
```

运行实践案例，大家可以发现运行的结果和前一个实例一样，LengthFieldBasedFrameDecoder解码器可以正确地解析内容。这说明一个问题：参数的设置都是正确。

如果将协议设计得再复杂一点点：在长度字段前面加上两个字节的版本字段，在长度字段后面加上4个字节的魔数，用来对数据包做一些安全的认证。协议的数据包如图7-5所示。

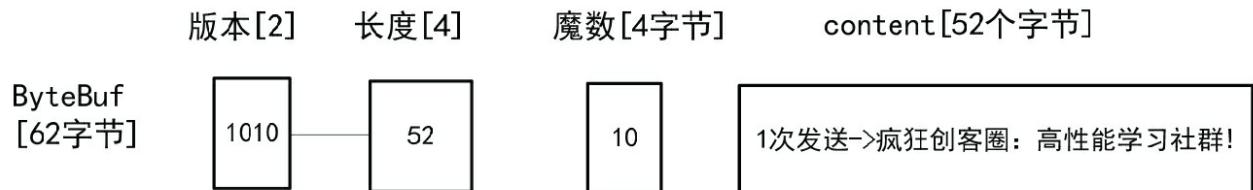


图7-5 包含协议版本号、魔数的Head-Content协议的示意图

那么使用LengthFieldBasedFrameDecoder解码器，解码图7-5中的Head-Content协议，构造器的参数又该如何计算呢？

参数的设置，大致可以如下：

第1个参数maxFrameLength可以设置为1024，表示数据包的最大长度为1024个字节。

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个字节。

第4个参数lengthAdjustment可以设置为4，长度调整的计算方法为：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。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个字节数据。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 LengthFieldBasedFrameDecoderspliter =
            new LengthFieldBasedFrameDecoder(1024, 2, 4, 4, 10);
        ChannelInitializeri = new ChannelInitializer<EmbeddedChannel>() {
            protected void initChannel(EmbeddedChannelch) {
                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++) {
            ByteBufbuf = 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();
    }
}
```

运行实践案例，大家可以发现运行的结果和前一个实例一样。这说明：参数计算是正确的，LengthFieldBasedFrameDecoder解码器可以正确地解析内容。

## 7.3 Encoder原理与实践

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。通过流水线写入到底层的Java通道。

在Netty中，什么叫编码器呢？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，或者编码成另一种Java POJO对象。

编码器是ChannelOutboundHandler出站处理器的实现类。一个编码器将出站对象编码之后，编码后数据将被传递到下一个ChannelOutboundHandler出站处理器，进行后面出站处理。

由于最后只有ByteBuf才能写入到通道中去，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。这里有个问题：为什么编码成的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：因为出站处理的顺序是从后向前的。

### 7.3.1 MessageToByteEncoder编码器

MessageToByteEncoder是一个非常重要的编码器基类，它位于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。但是，它的encode编码方法是一个抽象方法，没有具体的encode编码逻辑实现。而实现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)
        throws Exception {
        out.writeInt(msg);
        Logger.info("encoder Integer = " + msg);
    }
}
```

继承MessageToByteEncoder时，需要带上泛型实参，表示编码之前的Java POJO原类型。在这个示例程序中，编码之前的类型是Java整数类型（Integer）。

上面的encode方法实现很简单：将入站数据对象msg写入到Out实参即可（基类传入的ByteBuf类型的对象）。编码完成后，基类MessageToByteEncoder会将输出的ByteBuf数据包发送到下一站。

至此，编码器Integer2ByteEncoder已经完成，如何使用呢？这里编写了一个测试实例，代码如下：

```
package com.crazymakercircle.netty.encoder;
//...
public class Integer2ByteEncoderTester {
    /**
     * 测试整数编码器
     */
    @Test
    public void testIntegerToByteDecoder() {
        ChannelInitializer<EmbeddedChannel> 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);
        }
    }
}
```

```
channel.flush();
//取得通道的出站数据包
ByteBufbuf = (ByteBuf) channel.readOutbound();
while (null != buf) {
    System.out.println("o = " + buf.readInt());
    buf = (ByteBuf) channel.readOutbound();
}
try {
    Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
```

---

在上面的实例中，首先将Integer2ByteEncoder加入了嵌入式通道，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，将数据帧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贴出了。建议参考源代码工程，自行设计和实现一个整数编码器，以便加深理解。

### 7.3.2 MessageToMessageEncoder编码器

上一节的示例程序是将POJO对象编码成ByteBuf二进制对象。问题是：是否能够通过Netty的编码器将某一种POJO对象编码成另外一种的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类型的编码逻辑。在encode实现方法中，编码完成后，将解码后的目标对象加入到encode方法中的List实参列表中即可。

下面通过实现从字符串String到整数Integer的编码器，来演示一下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) throws Exception {
        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新类继承自MessageToMessageEncoder基类，并且明确了入站的数据类型为String。在encode方法中，将字符串的数字提取出来之后，放入到list列表，其他的字符直接略过。在子类的encode方法处理完成之后，基类会对这个List实例的所有元素进行迭代，将List列表的元素逐个发送给下一站。

编码器String2IntegerEncoder已经完成，代码也很简单。下面编写一个测试实例，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);
    }
    channel.flush();
    ByteBufbuf = (ByteBuf) channel.readOutbound();
    while (null != buf) {
        System.out.println("o = " + buf.readInt());
        buf = (ByteBuf) channel.readOutbound();
    }
    try {
        Thread.sleep(Integer.MAX_VALUE);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

---

除了需要使用String2IntegerEncoder，这里还需要用到前面那个编码器Integer2ByteEncoder。为什么呢？因为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。由于出站处理的过程是从后向前的次序，因此Integer2ByteEncoder先加入流水线的前面，String2IntegerEncoder加入流水线的后面。

此编码器的运行比较简单，运行的结果就不在书中贴出了。建议读者参考源代码工程，自行设计和实现一个String2IntegerEncoder编码器，以便加深理解。

## 7.4 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都是分开实现的。例如，通过继承 `ByteToMessageDecoder` 基类或者它的子类，完成 `ByteBuf` 数据包到 `POJO` 的解码工作；通过继承基类 `MessageToByteEncoder` 或其子类，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，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到通道的流水线时，常常需要分两次添加。

现在问题是：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：这就要用到 Netty 的新类型——`Codec` 类型。

## 7.4.1 ByteToMessageCodec编解码器

完成POJO到ByteBuf数据包的配套的编码器和解码器的基类，叫作ByteToMessageCodec<I>，它是一个抽象类。从功能上说，继承它，就等同于继承了ByteToMessageDecoder解码器和MessageToByteEncoder编码器这两个基类。

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)
        throws Exception {
        out.writeInt(msg);
        System.out.println("write Integer = " + msg);
    }
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
                      List<Object> out) throws Exception {
        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，简单通过继承的方式，将前面的编码器的encode方法和解码器的decode方法放在了同一个自定义的类中，这样在逻辑上更加紧密。当然在使用时，加入到流水线中，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，仅此而已，在技术上和功能上都没有增加太多的难度。

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，整合出一个新的基类——MessageToMessageCodec（编解码器）。这个基类同时包含了编码encode和解码decode两个抽象方法，用于完成pojo-to-pojo的双向转换。这仅仅是使用形式变得简化，在技术上并没有增加太多的难度。所以，本书不再展开介绍。

## 7.4.2 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。继承的方式有其不足，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IntegerFromByteDecoder整数解码器和它对应的IntegerToByteEncoder整数编码器组合起来。代码如下：

```
package com.crazymakercircle.netty.codec;
//...
public class IntegerDuplexHandler extends CombinedChannelDuplexHandler<
    Byte2IntegerDecoder, Integer2ByteEncoder>
{
    public IntegerDuplexHandler() {
        super(new Byte2IntegerDecoder(), new Integer2ByteEncoder());
    }
}
```

继承CombinedChannelDuplexHandler，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了，还是复用原来的编码器和解码器。

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。

## 7.5 本章小结

本章详细介绍了Netty的编码器和解码器。

在Netty中，解码器有ByteToMessageDecoder和MessageToMessageDecoder两大基类。如果要从ByteBuf到POJO解码，则可继承ByteToMessageDecoder基类；如果要从某一种POJO到另一种POJO解码，则可继承MessageToMessageDecoder基类。

Netty提供了不少开箱即用的Decoder解码器，能满足很多解码的场景需求，几个比较基础的解码器如下：

- 固定长度数据包解码器——FixedLengthFrameDecoder。
- 行分割数据包解码器——LineBasedFrameDecoder。
- 自定义分隔符数据包解码器——DelimiterBasedFrameDecoder。
- 自定义长度数据包解码器——LengthFieldBasedFrameDecoder。

在Netty中的编码器有MessageToByteEncoder和MessageToMessageEncoder两大重要的基类。如果要从POJO到ByteBuf编码，则可继承MessageToByteEncoder基类；如果要从某一种POJO到另一种POJO编码，则可继承MessageToMessageEncoder基类。

## 第8章 JSON和ProtoBuf序列化

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/反序列化的问题，例如一个“Person”对象从客户端通过TCP方式发送到服务器端；因为TCP协议（UDP等这种低层协议）只能发送字节流，所以需要应用层将Java POJO对象序列化成字节流，数据接收端再反序列化成Java POJO对象即可。“序列化”一定会涉及编码和格式化（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协议，在Web应用、移动开发方面等，这是常用的编码方式，因为JSON的可读性较强。但是它的性能稍差。

- 基于XML。和JSON一样，数据在序列化成字节流之前都转换成字符串。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）。

- 其他开源的序列化/反序列化框架，例如Apache Avro，Apache Thrift，这两个框架和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，客户端和服务器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，大概从两个方面着手：

(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，这会影响序列化/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系列的序列化框架；对于性能要求比较高的服务器程序，则应该选择传输效率更高的二进制序列化框架，目前的建议是Protobuf。

Protobuf是一个高性能、易扩展的序列化框架，与它的性能测试有关的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和Protobuf，还是其他的反序列化协议，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。

## 8.1 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。

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Handler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内容能一模一样。

现实总是那么残酷。或者说，理想很丰满，现实很骨感。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。

## 8.1.1 半包问题的实践案例

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——叫作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器端写ByteBuf。一直写到1000次，写入的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启动器配置和启动
        //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);
        }
        //...省略从容关闭客户端
    }
    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，首先要启动的是前面介绍过的NettyEchoServer回显服务器。然后启动的是新编写的客户端NettyDumpSendClient程序。客户端程序连接成功后，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8-1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8-1 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。

再仔细观察客户端的输出。可以看到，客户端和服务器端同样存在以上三种类型的输出。

对应于第1种情况，这里把接收端接收到的这种完整的ByteBuf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，就把这种读取的ByteBuf称为“粘包”。

对应于第3种情况，一个输入的ByteBuf被“拆”开读取，读取到一个破碎的包，就把这种读取的ByteBuf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”。

## 8.1.2 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，指接收端（Receiver）收到一个ByteBuf，包含了多个发送端（Sender）的ByteBuf，多个ByteBuf“粘”在了一起。

(2) 半包，就是接收端将一个发送端的ByteBuf“拆”开了，收到多个破碎的包。换句话说，一个接收端收到的ByteBuf是发送端的一个ByteBuf的一部分。

粘包和半包指的都是一次是不正常的ByteBuf缓存区接收，具体如图8-2所示。

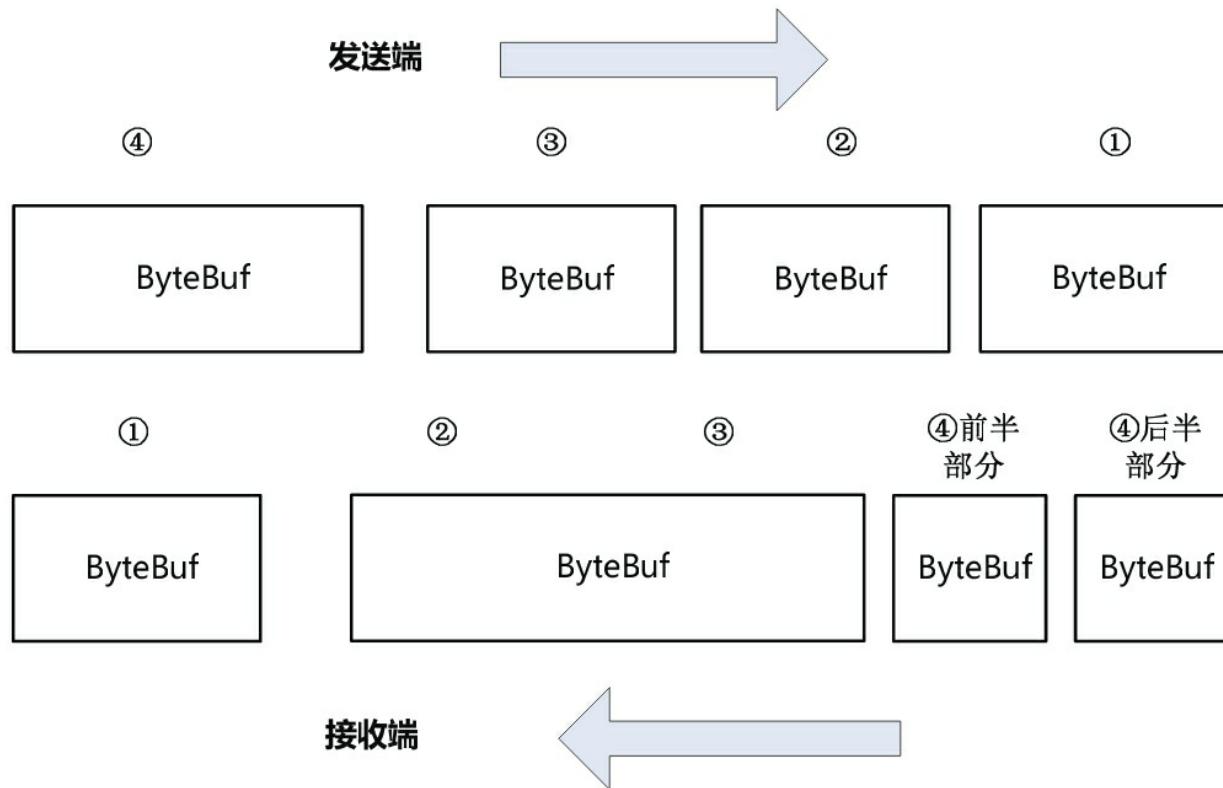


图8-2 粘包和半包现象：②③为粘包，④为半包

### 8.1.3 半包现象的原理

寻根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IO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：这中间编码器起作用，是将一个Java类型的数据转换成底层能够传输的二进制ByteBuf缓冲数据。解码器的作用与之相反，是将底层传递过来的二进制ByteBuf缓冲数据转换成Java能够处理的Java POJO对象。

在发送端Netty的应用层进程缓冲区，程序以ByteBuf为单位来发送数据，但是到了底层操作系统的内核缓冲区，底层会按照协议的规范对数据包进行二次拼装，拼装成传输层TCP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，首先保存在内核缓冲区，Netty读取ByteBuf时才复制到进程缓冲区。

在接收端，当Netty程序将数据从内核缓冲区复制到Netty进程缓冲区的ByteBuf时，问题就来了：

(1) 首先，每次读取底层缓冲的数据容量是有限制的，当TCP底层缓冲的数据包比较大时，会将一个底层包分成多次ByteBuf进行复制，进而造成进程缓冲区读到的是半包。

(2) 当TCP底层缓冲的数据包比较小时，一次复制的却不止一个内核缓冲区包，进而造成进程缓冲区读到的是粘包。

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，在应用层进行二次拼装，重新组装我们应用层的数据包。接收端的这个过程通常也称为分包，或者叫作拆包。

在Netty中，分包的方法，从第7章可知，主要有两种方法：

(1) 可以自定义解码器分包器：基于ByteToMessageDecoder或者ReplayingDecoder，定义自己的进程缓冲区分包器。

(2) 使用Netty内置的解码器。如，使用Netty内置的LengthFieldBasedFrameDecoder自定义分隔符数据包解码器，对进程缓冲区ByteBuf进行正确的分包。

在本章后面，这两种方法都会用到。

## 8.2 JSON协议通信

### 8.2.1 JSON序列化的通用类

Java处理JSON数据有三个比较流行的开源类库有：阿里的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、性能也还不错，另外Jackson社区相对比较活跃。Jackson开源库的缺点是：对于复杂POJO类型、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。

Google的Gson开源库是一个功能齐全的JSON解析库，起源于Google公司内部需求而由Google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，转换的能力非常强。

阿里巴巴的FastJson是一个高性能的JSON库。传闻说FastJson在复杂类型的POJO转换JSON时，可能会出现一些引用类型而导致JSON转换出错，需要进行引用的定制。顾名思义，从性能上说，FastJson库采用独创的算法，将JSON转成POJO的速度提升到极致，超过其他JSON开源库。

在实际开发中，目前主流的策略是：Google的Gson库和阿里的FastJson库两者结合使用。在POJO序列化成JSON字符串的应用场景，使用Google的Gson库；在JSON字符串反序列化成POJO的应用场景，使用阿里的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;
import com.alibaba.fastjson.JSONObject;
import com.google.gson.GsonBuilder;
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：使用谷歌Gson将 POJO 转成字符串
    public static String pojoToJson(java.lang.Object obj) {
        //String json = new Gson().toJson(obj);
        String json = gb.create().toJson(obj);
        return json;
    }
}
```

```
//反序列化：使用阿里Fastjson将字符串转成 POJO对象
public static <T>TjsonToPojo(String json, Class<T>tClass) {
    T t = JSONObject.parseObject(json, tClass);
    return t;
}
```

---

## 8.2.2 JSON序列化与反序列化的实践案例

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。

```
package com.crazymakercircle.netty.protocol;
import com.crazymakercircle.util.JsonUtil;
import lombok.Data;
@Data
public class JsonMsg {
    //id Field(字段)
    private int id;
    //content Field(字段)
    private String content;
    //序列化：调用通用方法，使用谷歌Gson转成字符串
    public String convertToJson() {
        return JsonUtil.pojoToJson(this);
    }
    //反序列化：使用阿里FastJson转成Java POJO对象
    public static JsonMsg parseFromJson(String json) {
        returnJsonUtil.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还加上了一个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），将JSON字符串反序列化成JsonMsg类型的对象实例。

使用POJO类JsonMsg实现从POJO对象到JSON的序列化、反序列化的实践案例演示，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;
    }
    //序列化 serialization & 反序列化 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);
        //JSON 字符串，反序列化成POJO对象
        JsonMsginMsg = JsonMsg.parseFromJson(json);
        Logger.info("id:=" + inMsg.getId());
        Logger.info("content:=" + inMsg.getContent());
    }
}
```

}



### 8.2.3 JSON传输的编码器和解码器之原理

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的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数据包的解码过程如图8-3所示，具体如下：

先使用LengthFieldBasedFrameDecoder（Netty内置的自定义长度数据包解码器）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JsonMsgDecoder解码器（一个自定义解码器）将JSON字符串解码成POJO对象。

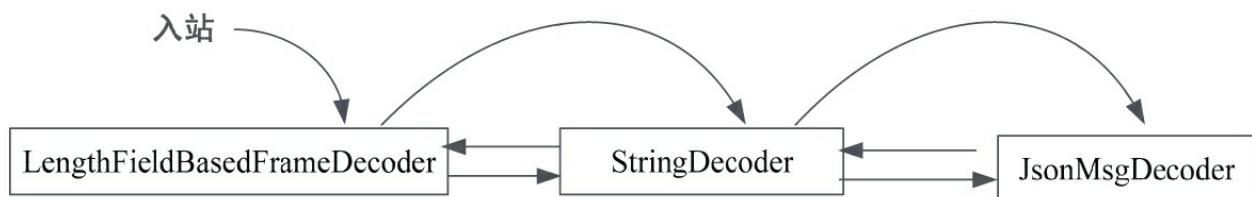


图8-3 JSON格式Head-Content数据包的解码过程

Head-Content数据包的编码过程如图8-4所示，具体如下：

先使用StringEncoder编码器（Netty内置）将JSON字符串编码成二进制字节数组。然后，使用LengthFieldPrepender编码器（Netty内置）将二进制字节数组编码成Head-Content二进制数据包。

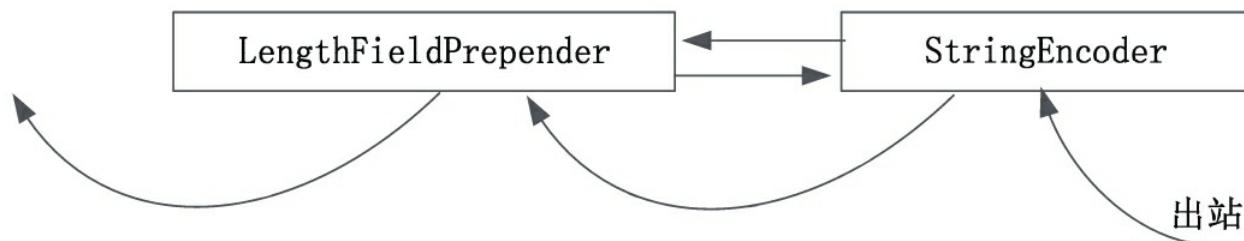


图8-4 JSON格式Head-Content数据包的编码过程

LengthFieldPrepender编码器的作用：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。另一个参数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true，表示长度字段的值（总长度）包含了自身的字节数。如果值为false，表示长度字段的值仅仅包含Content内容的二进制数据的长度。一般来说，lengthIncludesLengthFieldLength的默认值为false。

## 8.2.4 JSON传输之服务器端的实践案例

为了清晰地演示JSON传输，下面设计一个简单的客户端/服务器传输程序：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，编码后发送到服务器端。

为了简化流程，此服务器端的代码仅仅包含Inbound入站处理的流程，不包含OutBound出站处理的流程。也就是说，服务器端的程序仅仅读取客户端数据包并完成解码。服务器端的程序没有写出任何的输出数据包到对端（即客户端）。服务器端实践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
        EventLoopGroupbossLoopGroup = new NioEventLoopGroup(1);
        EventLoopGroupworkerLoopGroup = new NioEventLoopGroup();
        try {
            //...省略:启动器的反应器线程,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(SocketChannelch) throws Exception {
                    // 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 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,从容关闭
        }
        //服务器端业务处理器
        static class JsonMsgDecoderextends ChannelInboundHandlerAdapter {
            @Override
            public void channelRead(ChannelHandlerContextctx, Object msg) throws
                Exception {
                String json = (String) msg;
                JsonMsgjsonMsg = 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();
        }
    }
}
```

## 8.2.5 JSON传输之客户端的实践案例

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器端。客户端的程序并没有去处理从对端（即服务器端）过来的输入数据包。客户端的流程大致如下：

- (1) 通过谷歌的Gson框架，将POJO序列化成JSON字符串。
- (2) 然后，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) 最后，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。

客户端实践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 {
            //1 设置反应器线程组
            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 装配通道流水线
            b.handler(new ChannelInitializer<SocketChannel>() {
                //初始化客户端通道
                protected void initChannel(SocketChannel ch) throws Exception {
                    // 客户端通道流水线添加2个Handler业务处理器
                    ch.pipeline().addLast(new LengthFieldPrepender(4));
                    ch.pipeline().addLast(new
                        StringEncoder(CharsetUtil.UTF_8));
                }
            });
            ChannelFuture f = b.connect();
            f.addListener((ChannelFutureListener) listener) ->
            {
                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
    ChannelFuturecloseFuture = channel.closeFuture();
    closeFuture.sync();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 从容关闭EventLoopGroup,
    // 释放掉所有资源, 包括创建的线程
    workerLoopGroup.shutdownGracefully();
}
}
//构建Json对象
public JsonMsgbuild(int id, String content) {
    JsonMsg user = new JsonMsg();
    user.setId(id);
    user.setContent(content);
    return user;
}
public static void main(String[] args) throws InterruptedException {
    int port = NettyDemoConfig.SOCKET_SERVER_PORT;
    String ip = NettyDemoConfig.SOCKET_SERVER_IP;
    new JsonSendClient(ip, port).runClient();
}
}
```

---

执行次序是：先启动服务器端，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。

## 8.3 Protobuf协议通信

Protobuf是Google提出的一种数据交换的格式，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进行通信。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的解码过程则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。

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数据包是一种二进制的格式，相对于文本格式的数据交换（JSON、XML）来说，速度要快很多。由于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。

与JSON、XML相比，Protobuf算是后起之秀，是Google开源的一种数据格式。只是Protobuf更加适合于高性能、快速响应的数据传输应用场景。另外，JSON、XML是文本格式，数据具有可读性；而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，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，因为数据量很大，那么选择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。

### 8.3.1 一个简单的proto文件的实践案例

Protobuf使用proto文件来预先定义的消息格式。数据包是按照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“.proto”所使用的Protobuf协议版本，这里使用的是“proto3”。也可以使用旧一点的版本“proto2”，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声明选项，选项的前面有option关键字。“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Java代码放入指定的package中。“java\_outer\_classname”选项的作用为：在生成“proto”文件所对应Java代码时，所生产的Java外部类的名称。

在“proto”文件中，使用message这个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都对应于一个最终的Java POJO类。消息结构体的字段对应到POJO类的属性。也就是说，每定义一个“message”结构体相当于声明一个Java中的类。并且message中可以内嵌message，就像java的内部类一样。

每一个消息结构体可以有多个字段。定义一个字段的格式，简单来说就是“类型名称=编号”。例如“string content=2;”，表示该字段是string类型，名为content，序号为2。字段序号表示为：在Protobuf数据包的序列化、反序列化时，该字段的具体排序。

在每一个“.proto”文件中，可以声明多个“message”。大部分情况下，会把有依

赖关系或者包含关系的message消息结构体写入一个.proto文件。将那些没有关联关系的message消息结构体，分别写入不同的文件，这样便于管理。

### 8.3.2 控制台命令生成POJO和Builder

完成“.proto”文件定义后，下一步就是生成消息的POJO类和Builder（构造者）类。有两种方式生成Java类：一种是通过控制台命令的方式；另一种是使用Maven插件的方式。

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，对于在Linux或者Mac平台下，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的安装目录加入到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
```

---

上面的命令表示“proto”文件的名称为：./Msg.proto；所生产的POJO类和构造者类的输出文件夹为./src/main/java/。

### 8.3.3 Maven插件生成POJO和Builder

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，它可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plugin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>
</plugin>
```

protobuf-maven-plugin插件的配置项，具体介绍如下：

- protoSourceRoot：“proto”消息结构体文件的路径。
- outputDirectory：生成的POJO类和Builder类的目标路径。
- protocExecutable：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代码就利索生成了。或者在Maven的项目编译时，POJO类和Builder类也会自动生成。

### 8.3.4 消息POJO和Builder的使用之实践案例

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版本号的具体值为3.6.1。也就是说，Java运行时的protobuf依赖包的版本和“.proto”消息结构体文件中的syntax配置版本，以及编译“.proto”文件所使用的编译器“protoc3.6.1.exe”的版本，这三个版本需要配套一致。

#### 1. 使用Builder构造者，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消息结构体生成的Java类中，包含了一个POJO类、一个Builder类。构造POJO消息，首先需要使用POJO类的newBuilder静态方法获得一个Builder构造者。每一个POJO字段的值，需要通过Builder构造者的setter方法去设置。注意，消息POJO对象并没有setter方法。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serialization & 反序列化Deserialization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。下面是方式一：

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtobufDemo {
    //第1种方式:序列化 serialization & 反序列化 Deserialization
    @Test
    public void serAndDeser1() 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.MsginMsg = MsgProtos.Msg.parseFrom(data);
        Logger.info("id:=" + inMsg.getId());
        Logger.info("content:=" + inMsg.getContent());
    }
//...
}
```

---

这种方式通过调用POJO对象的toByteArray()方法将POJO对象序列化成字节数组。通过调用parseFrom (byte[] data) 方法，Protobuf也可以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层的应用场景。

### 3.序列化serialization & 反序列化Deserialization的方式二

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtobufDemo {
//...
    //第2种方式:序列化 serialization &反序列化 Deserialization
    @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.MsginMsg = MsgProtos.Msg.parseFrom(inputStream);
        Logger.info("id:=" + inMsg.getId());
        Logger.info("content:=" + inMsg.getContent());
    }
//...
}
```

---

这种方式通过调用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parseFrom (InputStream) 方法，Protobuf从输入流中读取二进制码流重新反序列化，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 OIO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，存在着粘包/半包的问题。

### 4.序列化serialization &反序列化Deserialization的方式三

```
package com.crazymakercircle.netty.protocol;
//...
public class ProtobufDemo {
//...
    //第3种方式:序列化 serialization &反序列化 Deserialization
```

---

```
//带字节长度: [字节长度][字节数据],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
    ByteArrayOutputStreamoutputStream = new ByteArrayOutputStream();
    message.writeDelimitedTo(outputStream);
    ByteArrayInputStreaminputStream
        = new ByteArrayInputStream(outputStream.toByteArray());
    //从二进码流反序列化成Protobuf对象
    MsgProtos.MsginMsg = MsgProtos.Msg.parseDelimitedFrom(inputStream);
    Logger.info("id:=" + inMsg.getId());
    Logger.info("content:=" + inMsg.getContent());
}
}
```

---

这种方式通过调用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。

反序列化时，调用parseDelimitedFrom（InputStream）方法。Protobuf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。

这种方式可以用于异步操作的NIO应用场景中，解决了粘包/半包的问题。

## 8.4 Protobuf编解码的实践案例

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。

## 8.4.1 Protobuf编码器和解码器的原理

Netty内置的Protobuf专用的基础编码器/解码器为：ProtobufEncoder编码器和ProtobufDecoder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，直接使用了message.toByteArray()方法将Protobuf的POJO消息对象编码成二进制字节，数据放入Netty的Bytebuf数据包中，然后交给了下一站的编码器。

### 2.ProtobufDecoder解码器

ProtobufDecoder解码器和ProtobufEncoder编码器相互对应。ProtobufDecoder需要指定一个POJO消息的prototype原型POJO实例，根据原型实例找到对应的Parser解析器，将二进制的字节解析为Protobuf POJO消息对象。

在Java NIO通信中，仅仅使用以上这组编码器和解码器会存在粘包/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，可以在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。

### 4.ProtobufVarint32FrameDecoder长度解码器

ProtobufVarint32FrameDecoder和ProtobufVarint32LengthFieldPrepender相对应。其作用是，根据数据包中varint32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，为什么不用int这种固定类型的长度呢？

varint32是一种紧凑的表示数字的方法，它不是一种具体的数据类型。varint32它用一个或多个字节来表示一个数字，值越小的数字使用越少的字节数，值越大使用的字节数越多。varint32根据值的大小自动进行长度的收缩，这能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。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的内置的ProtoBuf的编码器和解码器已经初步介绍完了。可以通过

这两组编码器/解码器完成Length+Protobuf Data(Head-Content)协议的数据传输。但是，在更加复杂的传输应用场景，Netty的内置编码器和解码器是不够用的。例如，在Head部分加上魔数字段进行安全验证；或者还需要对Protobuf Data的内容进行加密和解密等。也就是说，在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。

## 8.4.2 Protobuf传输之服务器端的实践案例

为了清晰地演示Protobuf传输，下面设计了一个简单的客户端/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，再发送到服务器端。

在服务器端，Protobuf协议的解码过程如下：

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，可以使用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ProtobufBussinessDecoder解码器来处理Protobuf POJO对象。

服务端的实践案例程序代码如下：

```
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
        EventLoopGroupbossLoopGroup = new NioEventLoopGroup(1);
        EventLoopGroupworkerLoopGroup = new NioEventLoopGroup();
        try {
            //...省略:启动器的反应器线程,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(SocketChannelch) throws Exception {
                    // 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(newProtobufVarint32FrameDecoder());
                    ch.pipeline().addLast(
                        newProtobufDecoder(MsgProtos.Msg.getDefaultInstance()));
                    ch.pipeline().addLast(new ProtobufBussinessDecoder());
                }
            });
            //....省略端口绑定,服务监听,从容关闭
        }
        //服务器端的业务处理器
        static class ProtobufBussinessDecoderextends ChannelInboundHandlerAdapter
        {
            @Override
            public void channelRead(ChannelHandlerContextctx, Object msg) throws Exception {
                MsgProtos.MsgprotoMsg = (MsgProtos.Msg) msg;
                //经过流水线的各个解码器，到此Person类型已经可以断定
                Logger.info("收到一个MsgProtos.Msg数据包 =» ");
                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();
        }
    }
}
```

### 8.4.3 Protobuf传输之客户端的实践案例

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协议的编码如8-5所示，过程如下：

先使用Netty内置的ProtobufEncoder，将Protobuf POJO对象编码成二进制的字节数组；然后，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器端。

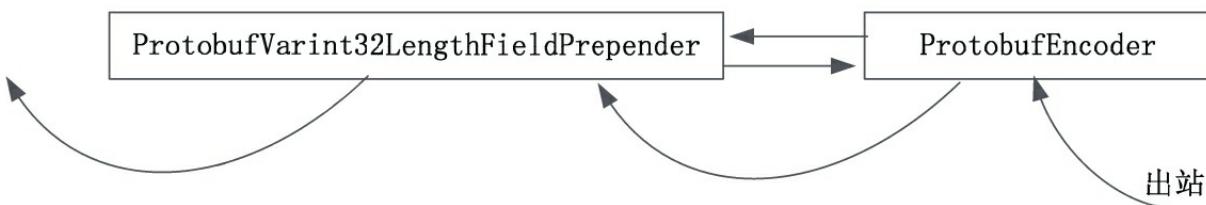


图8-5 Protobuf协议的解码过程

客户端的实践案例程序代码如下：

```
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 {
            //1 设置反应器线程组
            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 装配通道流水线
            b.handler(new ChannelInitializer<SocketChannel>() {
                //初始化客户端通道
                protected void initChannel(SocketChannel ch) throws Exception {
                    // 客户端太多流水线添加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,从容关闭
}
//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();
}
}
```

---

执行次序是：先启动服务器端，然后启动客户端。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，说明程序运行是正确的。

## 8.5 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，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。这些代码可以操作在.proto文件中定义的消息类型，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）。

## 8.5.1 proto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]
```

### 1.syntax版本号

对于一个“.proto”文件而言，文件首个非空、非注释的行必须注明Protobuf的语言版本，即syntax="proto3"，否则默认版本是proto2。

### 2.package包

和Java语言类似，通过package指定包名，用来避免信息（message）名字冲突。如果两个信息的名称相同，但是package包名不同，则它们可以共同存在。

如果第一个“.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，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.MsgcrazyMsg = 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Msg结构体。这一点和Java中package的使用方法是一样的。

另外，package指定包名后，会对生成的消息POJO代码产生影响。在Java语言中，会以package指定的包名作为生成的POJO类的包名。

### 3.选项

选项是否生效与“.proto”文件使用的一些特定的语言场景有关。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，生成类所在的Java包名。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。有两种方式：

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`，也即是方式2，表示使用外部类打包的方式。如果设置option `java_multiple_files=true`，则使用方式1生成Java类，则一个消息对应一个POJO Java类。

选项option `java_outer_classname`表示Protobuf编译器在生成Java POJO消息类时，如果“.proto”定义的全部POJO消息类都作为内部类打包在同一个外部类中，则以此作为外部类的类名。

## 8.5.2 消息结构体与消息字段

定义Protobuf消息结构体的关键字为message。一个信息结构体由一个或者多个消息字段组合而成。

```
// [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

### 1.限定修饰符

**repeated**限定修饰符：表示该字段可以包含0~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0~1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：用来保留字段名称（Field Name）和分配标识号（Assigning Tags），用于将来的扩展。

```
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.数据类型

详见下一个节。

### 3.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割，例如使用first\_name形式，而不是的驼峰式firstName。

### 4.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 $1 \sim 2^{32}$ （4294967296）。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。为什么呢？[1, 15]之内的标识号，在编码的时候只会占用一个字节。[16, 2047]之内的标识号则要占用2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Google Protobuf系统的内部保留值，建议不要在自己的项目中使用。

一个消息结构体中的标识号是无须连续的。另外，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。

### 8.5.3 字段的数据类型

Protobuf定义了一套基本数据类型。几乎都可以映射到C++\Java等语言的基本数据类型。

.proto Type	Notes	Java Type
double		double
Float		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，请使用 sint64 替代	int
uint32	使用变长编码	int
uint64	使用变长编码	long
sint32	使用变长编码，这些编码在负值时比 int32 高效得多	int
sint64	使用变长编码，有符号的整型值。编码时比通常的 int64 高效。	long
fixed32	总是 4 个字节，如果数值总是比 $2^{28}$ 大的话，这个类型会比 uint32 高效。	int
fixed64	总是 8 个字节，如果数值总是比 $2^{56}$ 大的话，这个类型会比 uint64 高效。	long
sfixed32	总是 4 个字节	int
sfixed64	总是 8 个字节	long
Bool		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

变长编码的类型表示打包的字节并不是固定，而是根据数据的大小或者长度来定。例如int32，如果数值比较小，在0~127时，使用一个字节打包。

关于fixed32和int32的区别。fixed32的打包效率比int32的效率高，但是使用的空间一般比int32多。因此一个属于时间效率高，一个属于空间效率高。根据项目的实际情况，一般选择fixed32，如果遇到对传输数据量要求比较苛刻的环境，则可以选择int32。

## 8.5.4 其他的语法规规范

### 1.import声明

在需要多个消息结构体时，“.proto”文件可以像Java语言的类文件一样分离为多个，在需要的时候通过import导入需要的文件。导入的操作和Java的import的操作大致相同。

### 2.嵌套消息

“.proto”文件支持嵌套消息，消息中可以包含另一个消息作为其字段，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
            int64ival = 1;
            bool booly = 2;
        }
    }
    message MiddleB{   // Level 1
        message Inner { // Level 2
            int32ival = 1;
            bool booly = 2;
        }
    }
}
```

如果你想在它的父消息类型的外部重复使用这些内部的消息类型，可以使用Parent.Type的形式来使用它，例如：

```
message SomeOtherMessage {
    Outer.MiddleA.Inner ref = 1;
}
```

### 3.enum枚举

枚举的定义和Java相同，但是有一些限制。枚举值必须大于等于0的整数。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;
}
```

## 8.6 本章小结

JSON格式是直观的文本化序列化方式，在实际的开发中，尤其是在基于RESTful进行远程交互的应用开发中使用得非常多。一般来说，在实际开发中使用较多的JSON开发包是阿里的FastJson、谷歌的Gson。

Protobuf格式是非直观的二进制序列化方式，效率比较高，主要用于高性能的通信开发。

# 第9章 基于Netty的单体IM系统的开发实践

本章是Netty应用的综合实践篇：将综合使用前面学到的编码器、解码器、业务处理器等知识，完成一个聊天系统的设计和实现。

## 9.1 自定义ProtoBuf编解码器

前面已经详细介绍过：在Netty中，内置了一组ProtoBuf编/解码器——ProtobufDecoder解码器和ProtobufEncoder编码器，它们负责Protobuf POJO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Protobuf半包处理器：可变长度ProtobufVarint32FrameDecoder解码器、ProtobufVarint32LengthFieldPrepender编码器。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，可以解决简单的Protobuf协议的传输问题，不过，面对复杂Head-Content协议的解析，例如，在数据包Head部分，加上魔数、版本号字段，具体如图9-1所示，内置Protobuf系列编解码器，就显得无能为力了。

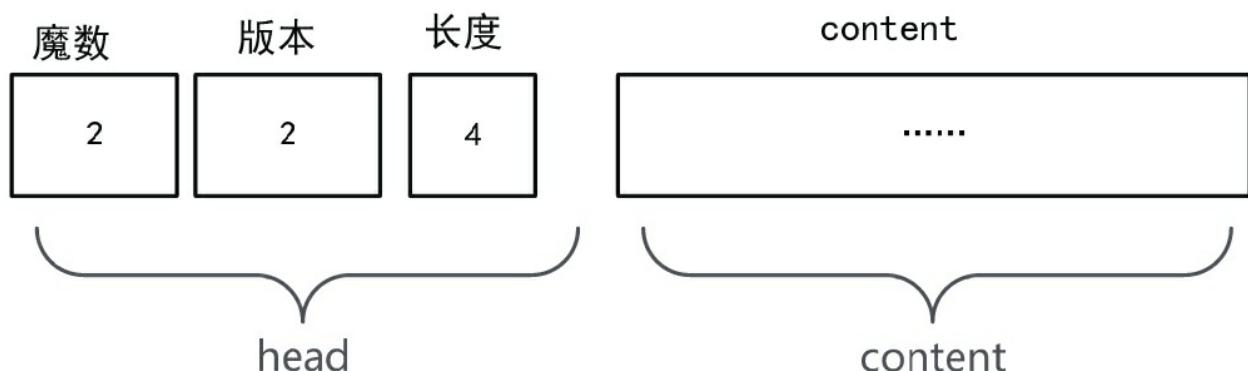


图9-1 复杂Head-Content协议的数据包

在通信数据包中，魔数是什么呢？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，魔数和土匪的接头暗号在原理上是一样的。无论是服务器端还是客户端，通信之前首先是对口令，如果口令不对，就不是符合自定义协议规范的数据包，也不是安全的数据包。通过魔数对比，服务器端能够在第一时间识别出不符合规范的数据包，为了安全考虑，可以直接关闭连接。

在通信数据包中，版本号是什么呢？如果在程序中有协议升级的需求，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，旧版本还需要使用。

解析复杂的Head-Content协议就需要自定义Protobuf编/解码器，需要开发者自己去解决半包问题，包括以下两个方面：

(1) 继承netty提供的MessageToByteEncoder编码器，完成Head-Content协议的复杂数据包的编码，将Protobuf POJO编码成Head-Content协议的二进制ByteBuf数据

包。

(2) 继承netty提供的ByteToMessageDecoder解码器，完成Head-Content协议的复杂数据包的解码，将二进制ByteBuf数据包最终解码出Protobuf POJO实例。

### 9.1.1 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到目标ByteBuf：

- 写入Protobuf的POJO的字节码长。
- 写入其他的字段，如魔数、版本号。
- 写入Protobuf的POJO的字节码内容。

按照上面的步骤，自定义一个ProtobufEncoder，代码如下：

```
package com.crazymakercircle.im.common.codec;
/**
 * 编码器
 */
@Slf4j
public class ProtobufEncoder extends MessageToByteEncoder<ProtoMsg.Message>
{
    @Override
    protected void encode(ChannelHandlerContext ctx, ProtoMsg.Message msg, ByteBuf out) throws Exception
    {
        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，也就是消息头，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，它们和长度的写入是类似的
        // 消息体中包含我们要发送的数据
        out.writeBytes(msg.toByteArray());
    }
}
```

#### 注意

这里写入的消息长度，调用了writeShort (length) 方法，仅仅两个字节。这就存在一个问题，数据包最大的净内容长度为32767个字节（有符号的短整型）。如果一个数据包需要传输更多的内容，可以调用writeInt (length) 方法。

## 9.1.2 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，将ByteBuf字节码解码成Protobuf的POJO对象，大致的过程如下：

- (1) 首先读取长度，如果长度位数不够，则终止读取。
- (2) 然后读取魔数、版本号等其他的字段。
- (3) 最后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，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
            return;
        }
        // 读取传送过来的消息的长度。
        int length = in.readShort();
        // 长度如果小于0
        if (length < 0)
        { // 非法数据，关闭连接
            ctx.close();
        }
        if (length > in.readableBytes())
        { // 读到的消息体长度如果小于传送过来的消息长度
            // 重置读取位置
            in.resetReaderIndex();
            return;
        }
        //省略：读取魔数、版本号等其他的数据
        //省略：读取内容
        byte[] array ;
        if (in.hasArray())
        {
            //堆缓冲
            ByteBuf slice=in.slice();
            array=slice.array();
        }
        else
        {
            //直接缓冲
            array = new byte[length];
            in.readBytes( array, 0, length);
        }
        // 字节转成Protobuf的POJO对象
        ProtoMsg.Message outmsg = ProtoMsg.Message.parseFrom(array);
        if (outmsg != null)
```

```
{  
    // 获取业务消息  
    out.add(outmsg);  
}  
}  


---


```

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：只需读相应的字节数，进行值的比较即可。

### 9.1.3 IM系统中Protobuf消息格式的设计

一般来说，网络通信涉及消息的定义，不管是直接使用二进制格式，还是XML、JSON等字符串格式，大体都可以分为3大消息类型：

(1) 请求消息

(2) 应答消息

(3) 命令消息

每个消息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。

Protobuf消息格式，大致有以下几个可供参考的原则：

#### 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，具体的例子如下：

```
enumHeadType {
    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
}
```

#### 原则二：使用一个Protobufmessage结构定义一类消息

例如，对应于登录请求类型——LOGIN\_REQUEST类型的消息，Protobuf message消息结构如下：

```
Protobuf message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
}
```

### 原则三：建议给应答消息加上成功标记和应答序号

对于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。

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呢？如果一个请求有多个响应，则发送端可以设计为：每一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。接收端在处理的时候，根据应答序号和结束标记，可以合并所有的响应消息。

对应于聊天响应类型——MESSAGE\_RESPONSE类型的消息，Protobuf message消息结构如下：

```
/*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
}
```

### 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。在通信的时候，可以从外层消息开始编码或者解码。

对应于聊天器中的外层消息，Protobuf message消息结构大致如下：

```
/*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
}
```

什么是序列号（sequence）呢？序列号主要用于请求数据包Request和响应数据包Response的配套，Response的值必须和Request相同，使得发送端可以进行请求-响应的匹配处理。

下面是一份完整的聊天器的“.proto”协议文件的定义：

```
syntax = "proto3";
package com.crazymakercircle.im.common.bean.msg;
message ProtoMsg {
    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
    }
    /*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
    }
    message LoginResponse {
        bool result = 1;
        uint32 code = 2;
        string info = 3;
        uint32 expose = 4;
    }
    message MessageRequest {
        uint64 msgId = 1;
        string from = 2;
        string to = 3;
        uint64 time = 4;
        uint32 msgType = 5;
        string content = 6;
        string url = 8;
        string property = 9;
        string fromNick = 10;
        string json = 11;
    }
    /*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
    }
    message MessageNotification {
        uint32 msgType = 1;
        bytes sender = 2;
        string json = 3;
        string timestamp = 4;
    }
    /*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
    }
}
```

以上的“.proto”文件在源代码工程的目录下：  
chatcommon\proto\protoConfig\ProtoMsg.proto。大家可以用Maven插件生成对应的  
Protobuf Builder和POJO类，以供后续使用。

## 9.2 概述IM的登录流程

单体IM系统中，首先需要登录。登录的流程，从端到端（End to End）的角度来说，包括以下环节：

- (1) 客户端发送登录数据包。
- (2) 服务器端进行用户信息验证。
- (3) 服务器端创建Session会话。
- (4) 服务器端返回登录结果的信息给客户端，包括成功标志、Session ID等。

在端到端（End to End）的流程中总计需要完成4次编/解码：

- 客户端编码：编码登录请求的Protobuf数据包编码。
- 服务器端解码：解码登录请求的Protobuf数据包解码。
- 服务器端编码：编码登录响应的Protobuf数据包编码。
- 客户端解码：解码登录响应的Protobuf数据包解码。

### 9.2.1 图解登录/响应流程的9个环节

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9-2所示。

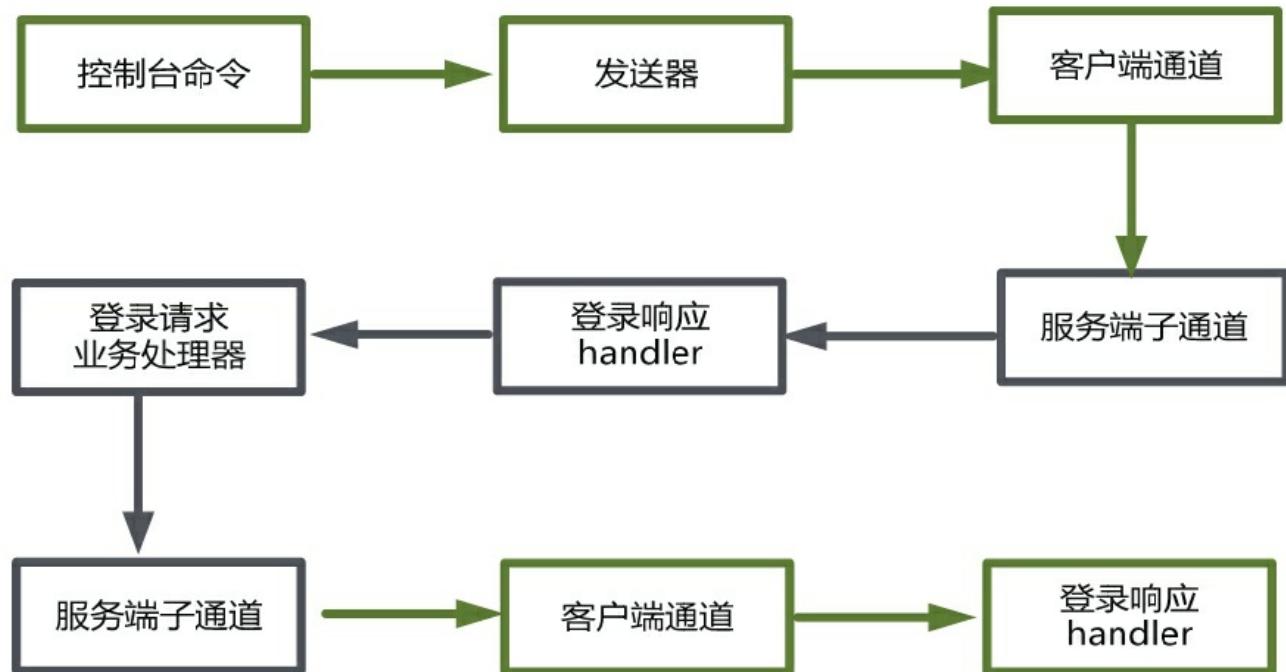


图9-2 登录/响应的流程

从客户端到服务器端再到客户端，9个环节的介绍如下：

- (1) 首先，客户端收集用户ID和密码，这一步需要使用到 LoginConsoleCommand 控制台命令类。
- (2) 然后，客户端发送Protobuf数据包到客户端通道，这一步需要通过 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，这一步需要通过Netty底层来完成。
- (4) 服务器子通道收到Protobuf数据包，这一步需要通过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，就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的Protobuf数据帧写入到对端，这一步需要通过Netty底层来完成。
- (8) 客户端通道收到Protobuf登录响应数据包，这一步需要通过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等等。

## 9.2.2 客户端涉及的主要模块

在具体的开发实践中，客户端所涉及的主要模块大致如下：

- ClientCommand模块：控制台命令收集器。
- ProtobufBuilder模块：Protobuf数据包构造者。
- Sender模块：数据包发送器。
- Handler模块：服务器响应处理器。

在具体的实现中，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：

- LoginConsoleCommand类：属于ClientCommand模块，它负责收集用户在控制台输入的用户ID和密码。
- CommandController类：属于ClientCommand模块，它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和收集相应的信息。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。
- LoginMsgBuilder类：属于ProtobufBuilder模块，它负责将User类组装成Protobuf登录请求数据包。
- LoginSender类：属于Sender模块，它负责将组装好Protobuf登录数据包发送到服务器端。
- LoginResponseHandler类：属于Handler模块，它负责处理服务器端的登录响应。

### 9.2.3 服务器端涉及的主要模块

在具体的开发实践中，服务器端所涉及的主要模块如下：

- Handler模块：客户端请求的处理。
- Processer模块：以异步方式完成请求的业务逻辑处理。
- 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。在服务器端的登录处理流程中，上面模块中对应的类为：

- 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。
- LoginProcesser类：属于Processer模块，完成服务器端的用户校验，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。
- ServerSession类：属于Session模块，如果校验成功，设置相应的会话状态（`isLoggedIn=true;`）；然后，将会话加入到服务器端的SessionMap映射中，可以接受其他用户发送的聊天消息。

这里有一个疑问：为什么在服务器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？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，稍后揭晓。

## 9.3 客户端的登录处理的实践案例

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：

```
//.....省略其他的输出  
2019-03-16 11:20:59 INFO (NettyClient.java:102) - 客户端开始连接 [疯狂创客圈IM]  
2019-03-16 11:20:59 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->show 所有命令 | 1->登录 | 2->聊天 | 10->退出 |  
//.....省略其他的输出
```

从上面的输出可以看出，ClientCommandMenu菜单展示类展示了4个选项：

- (1) 登录
- (2) 聊天
- (3) 退出
- (4) 查看全部命令

每一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) 登出命令的信息收集类： LogoutConsoleCommand
- (4) 命令的类型收集类： ClientCommandMenu

以上4个客户端主要的命令收集类都组合在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），CommandClient类会通过  
LoginConsoleCommand类完成用户ID和密码的收集。

### 9.3.1 LoginConsoleCommand和User POJO

登录命令的信息收集类LoginConsoleCommand负责从Scanner控制台实例收集客户端登录的用户ID和密码，代码如下：

```
package com.crazymakercircle.imClient.clientCommand;
//....
@Data
@Service("LoginConsoleCommand")
public class LoginConsoleCommand implements BaseCommand {
    public static final String KEY = "1";
    private String userName;
    private String password;
    @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 {
                break;
            }
        }
        userName=info[0];
        password = info[1];
    }
    @Override
    public String getKey() {
        return KEY;
    }
    @Override
    public String getTip() {
        return "登录";
    }
}
```

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，然后通过客户端的发送器loginSender开始向服务器端发送登录请求。

```
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);
        startLogin((LoginConsoleCommand) command);
        break;
    //...其他命令
}
}
}
}
//开始发送登录请求
private void startLogin(LoginConsoleCommand command) {
    //登录
    if (!isConnectFlag()) {
        log.info("连接异常, 请重新建立连接");
        return;
    }
    User user = new User();
    user.setUid(command.getUserName());
    user.setToken(command.getPassword());
    user.setDevId("1111");
    loginSender.setUser(user);
    loginSender.setSession(session);
    loginSender.sendLoginMsg();
}
}
//...
}
```

---

### 9.3.2 LoginSender发送器

LoginSender消息发送器的sendLoginMsg方法用于在第一步生成Protobuf登录数据包，在第二步则是调用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() {
        if (!isConnected()) {
            log.info("还没有建立连接!");
            return;
        }
        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);
    }
}
```

首先使用LoginMsgBuilder消息构造者来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。

```
package com.crazymakercircle.imClient.sender;
//...
@Data
@Slf4j
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)
                throws Exception {
                // 回调
                if (future.isSuccess()) {
                    sendSucced(message);
                } else {
                    sendfailed(message);
                }
            }
        });
    }
    //...
}
protected void sendSucced(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)/writeAndFlush (pkg) 这一组方法来发送数据包。前面多次反复讲到，发送方法channel.writeAndFlush (pkg) 是立即返回的，返回的类型是一个ChannelFuture异步任务实例。问题是：当channel.writeAndFlush (pkg) 函数返回时，如何判断是否已经成功将数据包写入到了底层的TCP连接呢？

答案是没有。在writeAndFlush (pkg) 方法返回时，真正的TCP写入的操作其实还没有执行。为什么呢？

由于这个知识点确实比较重要，因此在这里再一次强调一遍吧。在Netty中，无论是出站操作，还是出站操作，都有两个的特点：

(1) 同一条通道的所有出/入站处理都是串行的，而不是并行的。换句话说，同一条通道上的所有出/入站处理都会在它所绑定的EventLoop线程上执行。既然只有一个线程负责，那就只有串行的可能。

Netty是如何保障这一点的呢？

假如某个出/入站的处理在最开始执行的时候，会对当前的执行线程进行判断：如果当前线程不是通道的EventLoop线程，则当前出/入站的处理暂时不执行；Netty会将当前出/入站的处理，通过建立一个新的异步可执行任务，加入到通道的EventLoop线程的任务队列中。

EventLoop线程的任务队列是一个MPSC队列（即多生产者单消费者队列）。什么是MPSC队列呢？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，确保了EventLoop线程能做到：同一个通道上所有的IO操作是串行的，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，这点也能大大提升IO的性能。

(2) Netty的一个出/入站操作不是一次的单一Handler业务处理器操作，而是流水线上的一系列的出/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，通过为ChannelFuture异步任务增加

GenericFutureListener监听器的方式来判断writeAndFlush()是否已经执行完毕。当GenericFutureListener监听器的operationComplete方法被回调时，表示writeAndFlush()方法已经执行完毕了。而具体的回调业务逻辑，可以放在operationComplete监听器的方法中。

在上面的代码中，设计了两个sendSucced/sendfailed业务回调方法，放置在operationComplete监听器方法中，在发送完成后进行回调，并且将sendSucced和sendfailed方法封装在发送器的BaseSender基类中。如果需要改变默认的回调处理逻辑，可以在发送器子类重写基类的sendSucced和sendfailed方法即可。

再看另外一个话题：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。什么是会话呢？会话的作用是什么呢？什么时候创建会话呢？欲知后事如何，请看下回分解。

### 9.3.3 ClientSession客户端会话

ClientSession是一个很重要的胶水类，有两个成员：一个是user，代表用户，另一个是channel，代表了连接的通道。在实际开发中，这两个成员的作用是：

- (1) 通过user，可以获得当前的用户信息。
- (2) 通过channel，可以向服务器端发送消息。

ClientSession会话“左拥右抱”，左手“拥有”用户消息，右手“抱有”服务器端的连接。通过user成员可以获取当前的用户信息。借助channel通道，ClientSession可以写入Protobuf数据包，或者关闭Netty连接。

其次，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isLoggedIn

第三：ClientSession绑定在通道上，因而可以在入站处理器中通过通道反向取得绑定的ClientSession。

---

```
package com.crazymakercircle.imClient.client;
//...
@Slf4j
@Data
public class ClientSession {
    public static final AttributeKey<ClientSession> SESSION_KEY =
        AttributeKey.valueOf("SESSION_KEY");
    /**
     * 用户实现客户端会话管理的核心
     */
    private Channel channel;
    private User user;
    /**
     * 保存登录后的服务端sessionid
     */
    private String sessionId;
    private boolean isConnected = false;
    private boolean isLoggedIn = false;
    //绑定通道
    public ClientSession(Channel channel) {
        this.channel = channel;
        this.sessionId = String.valueOf(-1);
        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;
}
public String getRemoteAddress() {
    return channel.remoteAddress().toString();
}
//把protobuf数据包写入通道
public ChannelFuture writeAndFlush(Object pkg) {
    ChannelFuture f = channel.writeAndFlush(pkg);
    return f;
}
public void writeAndClose(Object pkg) {
    ChannelFuture future = channel.writeAndFlush(pkg);
    future.addListener(ChannelFutureListener.CLOSE);
}
//关闭通道
public void close() {
    isConnected = false;
    ChannelFuture future = channel.close();
    future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws
            Exception {
            if (future.isSuccess()) {
                log.error("连接顺利断开");
            }
        }
    });
}
}

```

什么时候创建客户端会话呢？在Netty客户端连接建立之后，并增加一个监听异步任务完成的监听器，代码如下：

```

package com.crazymakercircle.imClient.client;
//...
@Slf4j
@Data
@Service("CommandController")
public class CommandController {
    //...
    GenericFutureListener<ChannelFuture> connectedListener =
        (ChannelFuture f) -> {
        final EventLoop eventLoop = f.channel().eventLoop();
        if (!f.isSuccess()) {
            log.info("连接失败!在10s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();
        }
    };
    //...
}

```

### 9.3.4 LoginResponceHandler登录响应处理器

LoginResponceHandler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。在完成登录成功之后，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;
//...
@Slf4j
@ChannelHandler.Sharable
@Service("LoginResponceHandler")
public class LoginResponceHandler extends ChannelInboundHandlerAdapter {
    /**
     * 登录响应业务逻辑处理
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        //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.HeadingType headType = ((ProtoMsg.Message) msg).getType();
        if (!headType.equals(ProtoMsg.HeadingType.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LoginResponceHandler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，后面会有单独的小节，专门详细介绍客户端和服务器之间的心跳。

### 9.3.5 客户端流水线的装配

对于客户端的业务处理器流水线（Pipeline），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——LoginResponceHandler登录响应处理器的实例。

一般来说，在流水线最末端还需要装配一个ExceptionHandler异常处理器，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;
//....
@Slf4j
@Data
@Service("NettyClient")
public class NettyClient {
    @Autowired
    private ChatMsgHandlerchatMsgHandler;
    @Autowired
    private LoginResponceHandlerloginResponceHandler;
    //连接异步监听
    private GenericFutureListener<ChannelFuture>connectedListener;
    private Bootstrap b;
    private EventLoopGroup g;
    //....
    /**
     * 重连
     */
    public void doConnect() {
        try {
            b = new Bootstrap();
            //.... 设置通道初始化参数
            b.handler(new ChannelInitializer<SocketChannel>() {
                public void initChannel(SocketChannelch) {
                    ch.pipeline().addLast("decoder", new ProtobufDecoder());
                    ch.pipeline().addLast("encoder", new ProtobufEncoder());
                    ch.pipeline().addLast(loginResponceHandler);
                    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());
            }
        }
    public void close() {
        g.shutdownGracefully();
    }
}
```

关于业务处理器执行次序，再强调一下：登录响应处理器，必须装配在ProtobufDecoder解码器之后。具体的原因是：Netty客户端读到二进制Bytebuf数据包之后，首先需要通过ProtobufDecoder完成解码操作。解码后组装好Protobuf消息POJO，再进入loginResponceHandler登录响应处理器。

## 9.4 服务器端的登录响应的实践案例

服务器端的登录响应流程是：

- (1) ProtobufDecoder解码器把请求Bytebuf数据包解码成Protobuf数据包。
- (2) UserLoginRequestHandler登录处理器用于处理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，开始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/或者失败的登录响应报文，并发送给到客户端。

## 9.4.1 服务器流水线的装配

与客户端一样，对于服务器端流水线的，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登录业务处理器的实例。在流水线的最末端，还需要装配一个serverExceptionHandler异常处理器实例。

```
package com.crazymakercircle.imServer.server;
//...
@Data
@Slf4j
@Service("ChatServer")
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通道
                protected void initChannel(SocketChannel ch)
                    throws Exception {
                    // 管理子通道流水线中的Handler业务处理器
                    ch.pipeline().addLast(new ProtobufDecoder());
                    ch.pipeline().addLast(new ProtobufEncoder());
                    // 在流水线中添加Handler来处理登录，登录后删除
                    ch.pipeline().addLast(loginRequestHandler);
                    ch.pipeline().addLast(serverExceptionHandler);
                }
            });
            //省略：启动Bootstrap
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 8 从容关闭EventLoopGroup,
            // 释放掉所有资源，包括创建的线程
            wg.shutdownGracefully();
            bg.shutdownGracefully();
        }
    }
}
```

在服务器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登录请求处理器，由loginRequestHandler登录请求处理器完成实际的登录处理。

## 9.4.2 LoginRequestHandler登录请求处理器

这是个入站处理器，它继承自ChannelInboundHandlerAdapter入站适配器，重写了channelRead方法，主要的工作如下：

- (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站。
- (2) 创建一个ServerSession，即为客户建立一个服务器端的会话。
- (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)
        throws Exception {
        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.HeadingType headType = pkg.getType();
        if (!headType.equals(loginProcesser.type())) {
            super.channelRead(ctx, msg);
            return;
        }
        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() throws Exception {
                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());
            }
        });
    }
}
```

---

### 9.4.3 LoginProcesser用户验证逻辑

LoginProcesser用户验证逻辑主要包括：密码验证、将验证的结果写入到通道。如果登录验证成功，还需要绑定服务器端的会话，并且加入到在线用户列表中。

```
package com.crazymakercircle.imServer.processor;
//...
@Slf4j
@Service("LoginProcesser")
public class LoginProcesser extends AbstractServerProcessor {
    @Autowired
    LoginResponceBuilder loginResponceBuilder;
    @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
        boolean isValidUser = checkUser(user);
        if (!isValidUser) {
            ProtoInstant.ResultCodeEnum resultCode =
                ProtoInstant.ResultCodeEnum.NO_TOKEN;
            // 生成登录失败的报文
            ProtoMsg.Message response =
                loginResponceBuilder.loginResponce(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 =
        loginResponceBuilder.loginResponce(
        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100 ms以上的时间
        // 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
- 通过验证（Auth）服务器去验证用户

总之，验证用户是一个耗时间的操作。为了尽量地简化流程，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，也就是所有的登录都是成功的。

服务器端校验通过之后，可以完成服务器端会话（Session）的绑定工作。

服务器端的`ServerSession`会话也是一个胶水类，与客户端的`ClientSession`会话类似。每一个`ServerSession`对应一个客户端连接。一个`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，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`loginResponceBuilder`，构造一个登录响应POJO，设置好校验成功的标志位，调用会话（Session）的`writeAndFlush()`方法写到客户端。

#### 9.4.4 EventLoop线程和业务线程相互隔离

在前面的章节中，已经埋下一个疑问：为什么在服务器端整个登录处理需要分成两个模块：一个是Netty Handler业务处理器处理器，另一个是Processer业务逻辑处理器；而不是像客户端一样，在入站处理器Handler中统一完成呢？

答案是：在服务器端需要隔离EventLoop（Reactor）线程和业务线程。基本的方法是，使用独立的、异步的线程任务去执行用户验证的逻辑；而不在EventLoop线程中去执行用户验证的逻辑。

实际上，Reactor反应器线程和业务线程相互隔离，在服务器端非常重要。为什么呢？

首先，以读通道channelRead为例，看下一次普通的入站处理的基本步骤：

```
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 把结果写入到连接通道
}
```

其中的第三步，通常会涉及到一些比较耗时的业务处理操作，例如：

- (1) 数据库操作，百毫秒级，一般查询在100ms以上。
- (2) 远程接口调用，百毫秒级，一般在200ms以上，慢的在1000ms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线程的应用场景中，一旦耗时的业务处理操作也执行在EventLoop线程上，就会导致其他子通道的IO操作发生严重的性能问题。为什么说会是严重的性能问题呢？

大家知道，在默认情况下，Netty的一个EventLoop实例会开启2倍CPU核数的内部线程。通常情况下，一个Netty服务器端会有几万或者几十万的连接通道。也就是说，一个EventLoop内部线程会负责处理着几万个或者上十万个通道连接的IO处理。

在一个EventLoop内部线程上任务是串行的。如果一个Handler业务处理器中的channelRead()入站处理方法执行1000ms或者几秒钟，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1000ms或者

几秒钟。而耗时的入站/出站处理越多，就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。

就这样，严重的性能问题就出来了。咋办呢？解决办法是：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队列。对于耗时的业务操作封装成异步任务，并放入异步任务队列中去处理。这样的话，服务器端的性能会提升很多。

## 9.5 详解ServerSession服务器会话

无论是客户端还是服务器端，为了让通道（Channel）——连接和用户（User）状态的管理和使用变得方便，引入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，只是在实现上比较加单。

由于客户端和服务器分别都有各自的通道，并且相关的参数有一些也不一致，因此这里使用了两个会话类型：客户端会话ClientSession、服务端会话ServerSession。

会话和通道直接的导航关系有两个方向，一个是正向导航：通过会话导航到通道，主要用于出站处理的场景，通过会话将数据包写出到通道；另一个是反向导航，通过通道导航到会话，主要用于入站处理的场景，通过通道获取会话，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。

### 9.5.1 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“key-value”键值对的形式来保存任何Java Object类型的值。一般来说，可以存放一些与通道实例相關的属性，比如说服务器端的ServerSession会话实例。

除了Channel通道类，Netty中的HandlerContext处理器上下文类，也具备了类似的容器功能，可以绑定key-value键值对。

问题是：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（原理如图9-3），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key，返回一个Attribute类型的值。现特别说明一下：

(1) 这里的AttributeKey也不是原始的key（例如，放在Map中的key），而是一个key的包装类。AttributeKey确保了key的唯一性，在单个Netty应用中，key值必须唯一。

(2) 这里的值Attribute不是原始的value，也是value的包装类。原始的value就放置在Attribute包装类中，可以通过Attribute包装类实现value的读取（get）和设置（set），取值和设值。

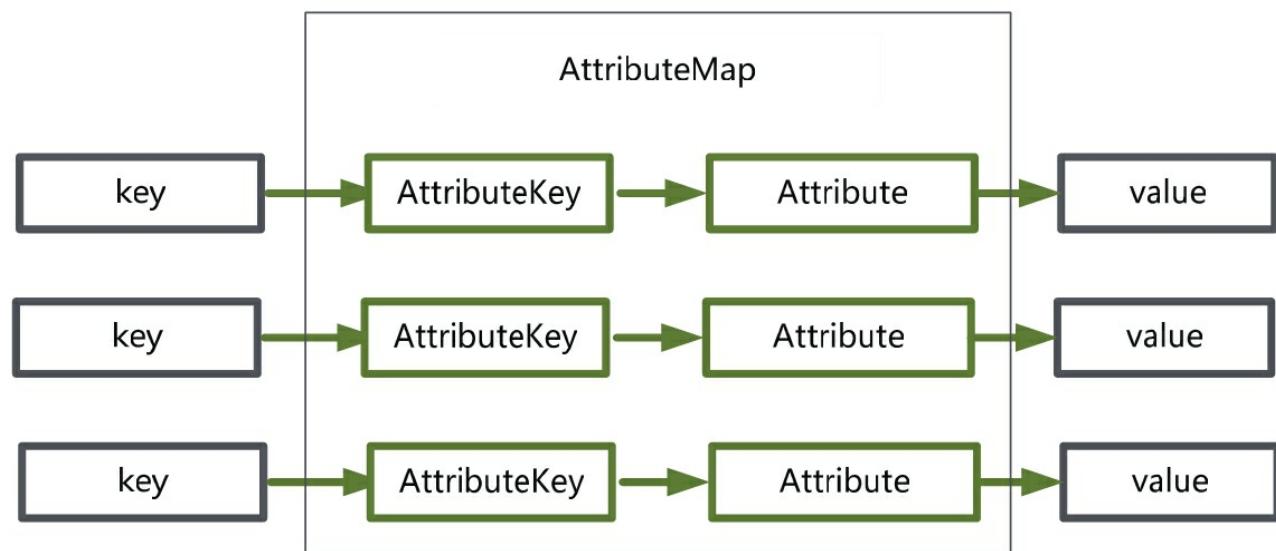


图9-3 AttributeMap原理图

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。可以通过AttributeMap的attr (...) 方法，根据key取得Attribute类型的value；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key
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，它的泛型类型为实际上的key-value键值对中value的实际类型。这里的value实际是ServerSession类型，所以返回值类型为AttributeKey<ServerSession>。

创建完AttributeKey后，就可以通过通道完成key-value的设值和取值了。常常是链式调用，首先通过通道的attr（AttributeKey）方法，取得value的Attribute包装类实例。然后通过Attribute的set()方法，设置真正的value值。在例子中，value值是一个会话（Session）实例。

再强调一句：这里的AttributeKey一般定义为一个常量，需要提前定义；它的那个泛型的参数是Attribute最终的包装值value的数据类型。

```
//key 的泛型形参是设置的value的类型
public static final AttributeKey<ServerSession> SESSION_KEY =
    AttributeKey.valueOf("SESSION_KEY");
```

## 2.Attribute取值

Attribute取值的方法，举例如下：

```
ServerSession session = ctx.channel().attr(SESSION_KEY).get();
```

还是使用了链式调用，首先通过通道的attr（AttributeKey）方法，取得键

(key) 所对应的值 (value) 的包装类Attribute实例。然后通过Attribute的get()方法，设置真正的值——前面所设置的会话session实例。

## 9.5.2 ServerSession服务器端会话类

在登录成功之后，服务器端会为每一个新连接通道创建一个ServerSession实例，表示用户与服务器端的会话信息。

每个ServerSession实例都拥有一个唯一标识，为sessionId。注意，唯一标识ServerSession实例的不是userId。为什么呢？主要原因是：同一个用户可能从网页端、手机端、电脑桌面，同时登录IM服务器端。微信、QQ都是典型的案例。另外，同一个用户的的消息需要在手机端、网页端、桌面端进行同步，也就是说同时接收消息、同时发送消息。

```
package com.crazymakercircle.imServer.server;
//...
@Data
@Slf4j
public class ServerSession {
    public static final AttributeKey<ServerSession> SESSION_KEY =
        AttributeKey.valueOf("SESSION_KEY");
    //通道
    private Channel channel;
    //用户
    private User user;
    //会话唯一标识
    private final String sessionId;
    //登录状态
    private boolean isLogin = false;
    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();
    }
    //关闭连接
    public static void closeSession(ChannelHandlerContext ctx) {
        ServerSession session =
            ctx.channel().attr(ServerSession.SESSION_KEY).get();
        if (null != session && session.isValid()) {
            session.close();
            SessionMap.inst().removeSession(session.getSessionId());
        }
    }
    //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("-", "");
    }
    //把Protobuf数据包写入通道
    public synchronized void writeAndFlush(Object pkg) {
        channel.writeAndFlush(pkg);
    }
    //关闭连接
    public synchronized void close() {
```

```
    ChannelFuture future = channel.close();
    future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws
            Exception {
            if (!future.isSuccess()) {
                log.error("CHANNEL_CLOSED error ");
            }
        }
    });
}
//....省略不是太重要的方法
```

---

从功能上说，`ServerSession`类与`ClientSession`类似，也是一个很重要的胶水类：

- (1) 通过`ServerSession`实例可以导航到通道，以发送消息。
- (2) 在通道收到消息时，通过`ServerSession`实例，反向导航到用户，以处理业务逻辑。

需要注意的是`sessionId`的类型，它是`String`类型而不是数字类型，为什么呢？主要是为了方便后期的分布式扩展。在后期，通信服务器需要从单体服务器扩展到多节点服务集群。为了在分布式集群的应用场景下，在后期`sessionId`可以确保全局唯一，使用`String`类型就比较方便。

### 9.5.3 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，服务器需要对这些大量的ServerSession实例进行管理。

SessionMap负责管理服务器端所有的ServerSession。它有一个线程安全的 ConcurrentHashMap类型的成员map，保持sessionId到服务器端ServerSession的映射。

```
package com.crazymakercircle.imServer.server;
//...
@Slf4j
@Data
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;
        }
    }
    //删除会话
    public void removeSession(String sessionId) {
        if (!map.containsKey(sessionId)) {
            return;
        }
        ServerSession s = map.get(sessionId);
        map.remove(sessionId);
        Print.tcfo("用户下线:id= " + s.getUser().getUid()
                + " 在线总数: " + map.size());
    }
    //....省略不是太重要的方法
}
```

在调用ServerSession.bind()绑定时，在SessionMap中增加一次映射：

```
public ServerSession bind() {
    //...
    SessionMap.inst().addSession(getSessionId(), this);
    //...
}
```

在调用ServerSession.unbind() 解除绑定时，在SessionMap中减少一次映射：

```
public ServerSession unbind() {
    //...
    SessionMap.inst().removeSession(getSessionId());
```

```
    } //...
```

---

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，消息需要通过服务器端，在不同的用户之间进行转发，这时也需要用到SessionMap。

## 9.6 点对点单聊的实践案例

单聊的业务非常简单，就像微信的聊天功能。这里强调一下，对单聊的业务流程进行具体介绍：

（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。

这里的消息格式设定为——`userId:content`。其中的`userId`，就是消息接收方目标用户B的`userId`；其中的`content`，表示聊天的内容。

（2）服务器端收到消息后，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。

理论上来说，单聊业务是十分的简单。不过，在这里有问题，为什么服务器端的路由转发不是根据`sessionID`，而是根据`userID`呢？

原因是：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。

### 9.6.1 简述单聊的端到端流程

单聊的端到端流程，从大的角度来说，包括以下环节（见图9-4）：

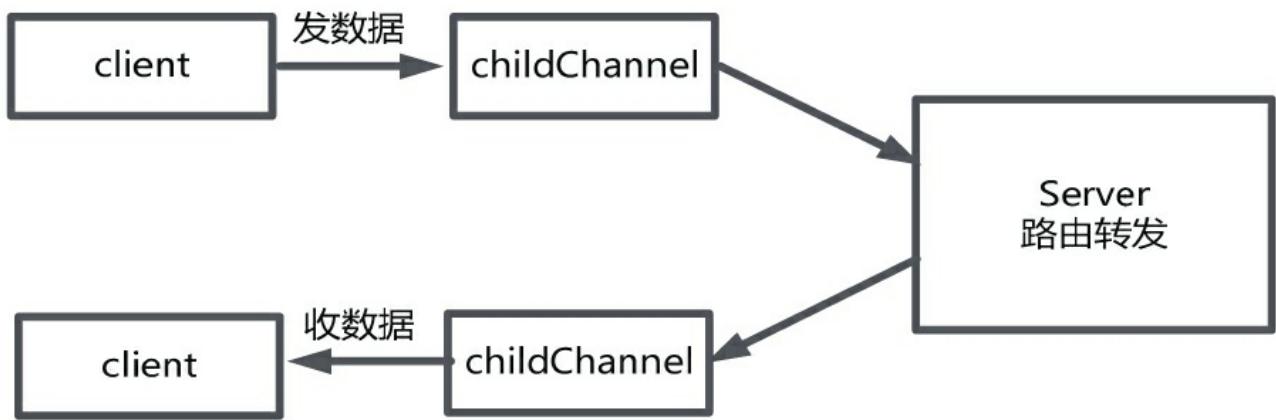


图9-4 单聊的端到端流程

- (1) 用户A发送单聊Protobuf数据包。
- (2) 服务器端接收到用户A的单聊数据包。
- (3) 服务器端转发单聊数据包到用户B。
- (4) 最终用户B接收到来自用户A的单聊数据包。

## 9.6.2 客户端的ChatConsoleCommand收集聊天内容

ChatConsoleCommand命令收集类负责从Scanner控制台实例获取聊天的消息（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;
    private String message;
    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];
    }
    //...
}
```

### 9.6.3 客户端的CommandController发送POJO

ChatConsoleCommand的调用者是CommandController命令控制类。CommandController在完成聊天内容和目标用户信息的收集后，在自己的startOneChat (...) 方法中调用chatSender发送器实例，将聊天消息组装成Protobuf数据包，通过客户端的通道发往服务器端。

```
package com.crazymakercircle.imClient.client;
//...
@Slf4j
@Data
@Service("CommandController")
public class CommandController {
    //聊天命令收集类
    @Autowired
    ChatConsoleCommand chatConsoleCommand;
    //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;
                    //省略其他的命令
                }
            }
        }
    }
    //发送单聊消息
    private void startOneChat(ChatConsoleCommand c) {
        //登录
        if (!isLogin()) {
            log.info("还没有登录，请先登录");
            return;
        }
        chatSender.setSession(session);
        chatSender.setUser(user);
        chatSender.sendChatMsg(c.getUserId(), c.getMessage());
    }
    //省略其他的命令处理
}
```

## 9.6.4 服务器端的ChatRedirectHandler消息转发

服务器端的消息转发处理器ChatRedirectHandler类，主要的工作如下：

- (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super.channelRead(ctx,msg)将消息交给流水线的下一站。
- (2) 对用户登录进行判断：如果没有登录，则不能发送消息。
- (3) 开启异步的消息转发，由负责转发的chatRedirectProcesser实例完成消息转发。

```
package com.crazymakercircle.imServer.handler;
//...
@sif4j
@Service("ChatRedirectHandler")
@ChannelHandler.Sharable
public class ChatRedirectHandler extends ChannelInboundHandlerAdapter {
    @Autowired
    ChatRedirectProcesser chatRedirectProcesser;
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        //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).getHeadType();
        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);
        });
    }
}
```

## 9.6.5 服务器端的ChatRedirectProcesser异步处理

编写负责异步消息转发的ChatRedirectProcesser类，功能如下：

- (1) 根据目标用户ID，找出所有的服务器端的会话列表。
- (2) 然后为每一个会话转发一份消息。

大致的代码如下：

```
package com.crazymakercircle.imServer.processor;
//...
@Slf4j
@Service("ChatRedirectProcessor")
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();
        Print.tcf("chatMsg | from=" +
                  + msg.getFrom() +
                  ", to=" + msg.getTo() +
                  ", content=" + msg.getContent());
        // 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
            Print.tcf("[ " + to + " ] 不在线，发送失败!");
        } else {
            toSessions.forEach((session) -> {
                // 将IM消息发送到接收方
                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(to)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 {
    //会话集合
    private ConcurrentHashMap<String, ServerSession> map =
        new ConcurrentHashMap<String, ServerSession>();
    // 根据用户id，获取会话对象
    public List<ServerSession> getSessionsBy(String userId) {
        List<ServerSession> list = map.values()
            .stream()
            .filter(s ->s.getUser().getUid().equals(userId));
    }
}
```

```
        .collect(Collectors.toList());  
    return list;  
}  
//...
```

---

## 9.6.6 客户端的ChatMsgHandler接收POJO

客户端的ChatMsgHandler聊天消息处理器很简单，主要的工作如下：

(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super.channelRead(ctx,msg)将消息交给流水线的下一站。

(2) 如果是聊天消息，则将聊天消息显示在控制台。

```
package com.crazymakercircle.imClient.handler;
//...
@ChannelHandler.Sharable
@Service("ChatMsgHandler")
public class ChatMsgHandler extends ChannelInboundHandlerAdapter {
    /**
     * 业务逻辑处理
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        //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.HeadingType headType = pkg.getType();
        if (!headType.equals(ProtoMsg.HeadingType.MESSAGE_REQUEST)) {
            super.channelRead(ctx, msg);
            return;
        }
        ProtoMsg.MessageRequest req = pkg.getMessageRequest();
        String content = req.getContent();
        String uid = req.getFrom();
        System.out.println(" 收到消息 from uid:" + uid + " -> " + content);
    }
}
```

## 9.7 详解心跳检测

客户端的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说起。

## 9.7.1 网络连接的假死现象

什么是连接假死呢？如果底层的TCP连接已经断开，但是服务器端并没有正常地关闭套接字，服务器端认为这条TCP连接仍然是存在的。

连接假死的具体表现如下：

- (1) 在服务器端，会有一些处于TCP\_ESTABLISHED状态的“正常”连接。
- (2) 但在客户端，TCP客户端已经显示连接已经断开。
- (3) 客户端此时虽然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器端认为有效，并且服务器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。

连接假死的情况虽然不常见，但是确实存在。服务器端长时间运行后，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，服务器端定时进行空闲检测。

## 9.7.2 服务器端的空闲检测

想解决假死问题，服务器端的有效手段是——空闲检测。

何为空闲检测？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。

服务器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;
//...
@Slf4j
public class HeartBeatServerHandler extends IdleStateHandler {
    private static final int READ_IDLE_GAP = 150; //最大空闲，单位s
    public HeartBeatServerHandler() {
        super(READ_IDLE_GAP, 0, 0, TimeUnit.SECONDS);
    }
    @Override
    protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt)
        throws Exception {
        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)
        throws Exception {
        //判断消息实例
        if (null == msg || !(msg instanceof ProtoMsg.Message)) {
            super.channelRead(ctx, msg);
            return;
        }
        ProtoMsg.Message pkg = (ProtoMsg.Message) msg;
        //判断消息类型
        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空闲检测时长，指的是一段时间内如果没有数据入

站，就判定连接假死；第二个参数是出站空闲检测时长，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。

判定假死之后，`IdleStateHandler`类会回调自己的`channelIdle()`方法。在这个子类的重写版本中，重写了这个空闲回调方法，手动关闭连接。

```
@Override
protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt)
    throws Exception {
    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，需要客户端定时发送心跳数据包（或报文、消息）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器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，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，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。

### 提示

如果`HeartBeatServerHandler`要重写`channelRead`方法，一定要记得调用基类的“`super.channelRead(ctx, msg);`”方法，不然`IdleStateHandler`的入站空闲检测会无效。

### 9.7.3 客户端的心跳报文

与服务器端的空闲检测相配合，客户端需要定期发送数据包到服务器端，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器端。

```
package com.crazymakercircle.imClient.handler;
//...
@Slf4j
@ChannelHandler.Sharable
@Service("HeartBeatClientHandler")
public class HeartBeatClientHandler extends ChannelInboundHandlerAdapter {
    //心跳的时间间隔，单位为s
    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) throws Exception {
        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) {
        ctx.executor().schedule(() -> {
            if (ctx.channel().isActive()) {
                log.info("发送HEART_BEAT 消息 to server");
                ctx.writeAndFlush(heartbeatMsg);
                //递归调用，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)
        throws Exception {
        //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.HeartType headType = pkg.getType();
        if (headType.equals(ProtoMsg.HeartType.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，它的递归调用的方式比较特别：使用了ctx.executor()获取当前通道绑定的Reactor反应器NIO线程，然后通过NIO线程的schedule()定时调度方法，每隔一段时间（50s）执行一次回调，向服务器端发送一个心跳数据包。

客户端的心跳间隔，要比服务器端的空闲检测时间间隔要短，一般来说，要比它的一半要短一些，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。

```
package com.crazymakercircle.imClient.clientHandler;
//...
@Slf4j
@ChannelHandler.Sharable
@Service("LoginResponceHandler")
public class LoginResponceHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        //...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",newHeartBeatClientHandler());
        }
    }
}
```

在登录成功之后，在ChannelPipeline通道流水线上，HeartBeatClientHandler心跳客户端处理器实例被动态插入到了解码器之后。

服务器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了简单的处理。这个地方还藏有另外的一个玄机，那就是HeartBeatClientHandler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，继承IdleStateHandler类，在客户端进行空闲检测。这样，客户端也可以对服务器进行假死判定，在服务器端假死的情况下，客户端可以发起重连。客户端的空闲检测的实战就留给大家去自行实验。

## 9.8 本章小结

本章内容是Netty学习的一次综合性的检验，覆盖了非常全面的Netty知识：包括自定义编解码器的开发、半包的处理、流水线的装配、会话的使用等。

## 第10章 ZooKeeper分布式协调

ZooKeeper（本书也简称ZK）是Hadoop的正式子项目，它是一个针对大型分布式的可靠协调系统，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，例如SOA的服务监控系统、Hadoop、Spark的分布式调度系统。

## 10.1 ZooKeeper伪集群安装和配置

现在，我们开始使用三台机器来搭建一个ZooKeeper集群。在学习环境中，由于没有多余的服务器，这里就将三个ZooKeeper节点都安装到本地机器上，故称之为伪集群模式。

虽然伪集群模式只是便于开发、普通测试，但是不能用于生产环境。实际上，如果了解了伪集群模式下的安装和配置，那么在生产环境下的配置也就大致差不多了。

首先是下载ZooKeeper。在Apache的官方网站提供了好多镜像下载地址，找到对应的版本，目前最新的版本是3.4.13。

下载地址为：

<http://mirrors.cnnic.cn/apache/ZooKeeper/ZooKeeper-3.4.13/ZooKeeper-3.4.13.tar.gz>

在Windows下安装时，需要把下载的ZooKeeper压缩文件解压缩到指定的目录，例如：

C:\devtools\ZooKeeper-3.4.13\>

接下来的文字描述中，将用以上的目录作为默认的安装目录。

### 10.1.1 创建数据目录和日志目录：

安装ZooKeeper之前，需要规划一下节点的个数，ZooKeeper节点数有以下要求：

(1) ZooKeeper集群节点数必须是奇数。

为什么呢？在ZooKeeper集群中，需要一个主节点，也称为Leader节点。主节点是集群通过选举的规则从所有节点中选举出来的。在选举的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，则可能会出现不满足这个规则的情况。

(2) ZooKeeper集群至少是3个。

ZooKeeper可以通过一个节点，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，仅仅作为学习使用尚可。在正常情况下，搭建ZooKeeper集群，至少需要3个节点。

这里作为学习案例，在本地机器上规划搭建一个具有3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：日志目录和数据目录。

具体来说，先创建日志目录。为3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，具体如图10-1所示。

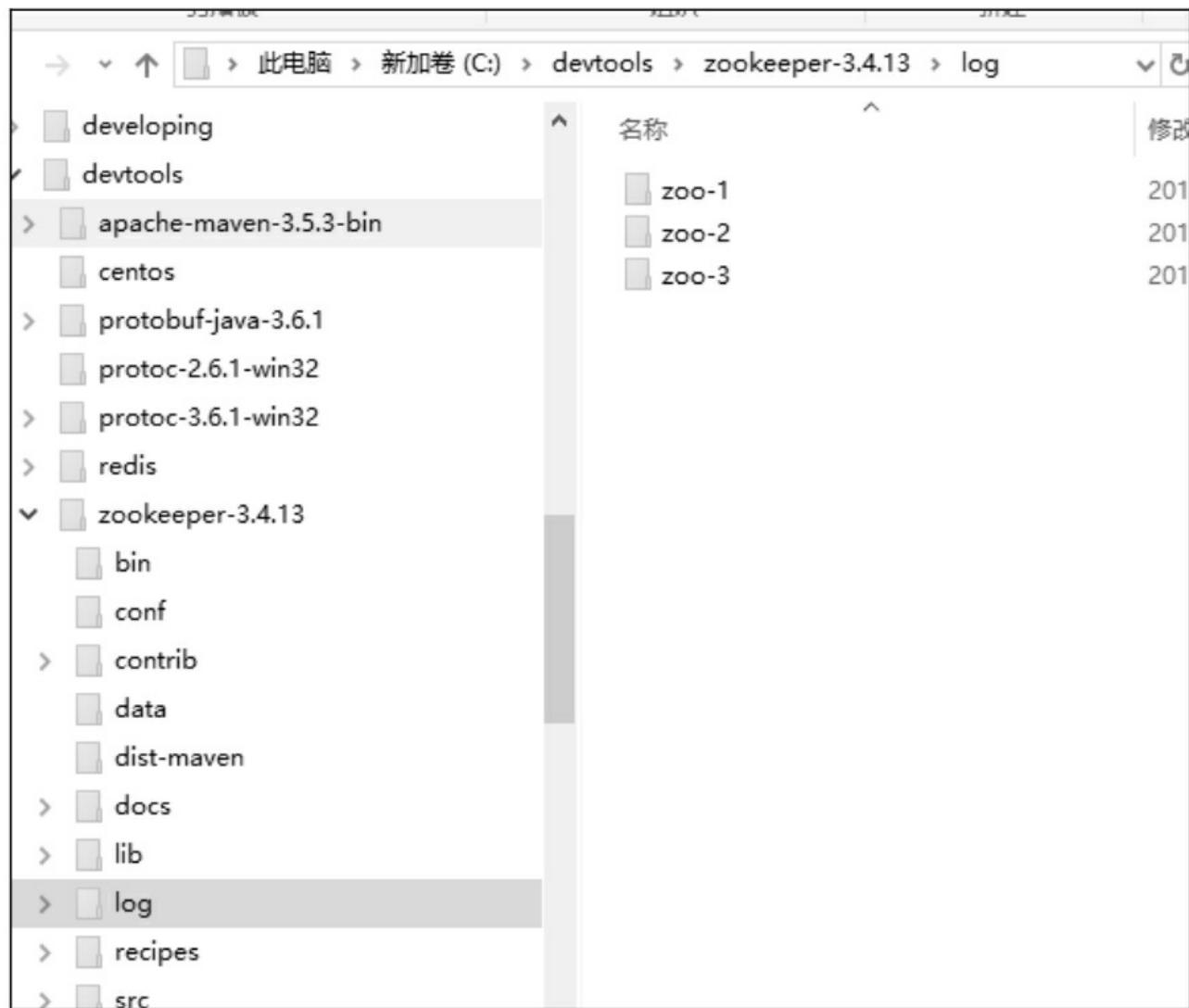


图10-1 集群中3个伪节点的日志目录

其次，创建数据目录。在安装目录下，为伪集群3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。

## 10.1.2 创建myid文件

安装集群的第二步，为每一个节点创建一个id文件。

每一个节点需要有一个记录节点id的文本文件，文件名为myid。myid文件的特点如下：

- myid文件的唯一作用是记录（伪）节点的编号。
- myid文件是一个文本文件，文件名称为myid。
- myid文件内容为一个数字，表示节点的编号。
- 在myid文件中只能有一个数字，不能有其他的内容。
- myid文件的存放位置，默认处于数据目录下。

下面分别为3个节点，创建3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1。

ZooKeeper对id的值有何要求呢？首先，myid文件中id的值只能是一个数字，即一个节点的编号ID；其次，id的范围是1~255，表示集群最多的节点个数为255个。

### 10.1.3 创建和修改配置文件

安装集群的第三步，为每一个节点创建一个配置文件。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本——zoo\_sample.cfg。复制这个样本，修改其中的某些配置项即可。

下面分别为三个节点，创建3个“.cfg”配置文件，具体的步骤为：

- (1) 将配置文件的样本zoo\_sample.cfg文件复制3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，对应于3个节点。
- (2) 需要修改每一个节点的配置文件，将前面准备的日志目录和数据目录，配置到“.cfg”中的正确选项中。

```
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：

·dataDir：数据目录选项，配置为前面准备的数据目录。myid文件处于此目录下。

·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。

首先是端口选项的配置，示例如下：

```
clientPort = 2181
```

clientPort：表示客户端连接ZooKeeper集群中的节点的端口号。在生成环境的集群中，不同的节点处于不同的机器，端口号一般都相同，这样便于记忆和使用。由于这里是伪集群模式，三个节点集中在一台机器上，因此3个端口号配置为不一样的。

clientPort：一般设置为2181。而伪集群下不同的节点，clientPort是不能相同的，可以按照编号进行累加。

其次是节点信息的配置，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
```

节点信息需要配置集群中所有节点的（id）编号、IP地址和端口号。格式为：

```
server.id=host:port:port
```

在“.cfg”配置文件中，可以按照这样的格式进行配置，每一行都代表一个节点。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，所以，每一个配置文件都需要配置全部的节点。

总体来说，配置节点的时候，需要注意四点：

- (1) 不能有相同id的节点，需要确保每个节点的myid文件中的id值不同；
- (2) 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文件中的id值保持一致；
- (3) 每一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那份，还需要配置所有节点的id、ip、端口。
- (4) 在每一行“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，后一个端口（如示例中的3888）用于选举主节点。

## 注意

在伪集群的模式下，每一行记录中的端口号必须修改成不一样的，主要是避免端口冲突。在实际的分布式集群模式下，由于不同节点的ip不同，每一行记录中可以配置相同的端口。

最后是时间相关选项的配置，示例如下：

```
tickTime=4000  
initLimit = 10  
syncLimit = 5
```

**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。如果不配置，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）的启动，并完成

与Leader（主节点）进行数据同步的时间。Follower节点在启动过程中，会与Leader节点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节点在initLimit时间内完成这项工作。该参数默认值为10，表示是参数tickTime值的10倍，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节点和Follower节点之间进行心跳检测的最大延时时间。在ZK集群运行的过程中，Leader节点会通过心跳检测来确定Follower节点是否存活。如果Leader节点在syncLimit时间内无法获取到Follower节点的心跳检测响应，那么Leader节点就会认为该Follower节点已经脱离了自己的同步。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。

## 10.1.4 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”文件的准备后，伪集群的安装工作就算基本完成了。

其中，“.cfg”配置文件是最为关键的环节。下面给出三份配置文件实际的代码。

第一个节点，配置文件zoo-1.conf的内容如下：

```
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，配置文件zoo-2.conf的内容如下：

```
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，配置文件zoo-3.conf的内容如下：

```
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，我们可以看出，对于不同的节点，“.cfg”配置文件中的配置项的内容大部分相同。这里需要强调一下，每个节点的集群节点信息的配置都是全量的；相对而言，每个节点的数据目录dataDir、日志目录dataLogDir和对外服务端口clientPort，则仅需配置自己的那份。

## 10.1.5 启动ZooKeeper伪集群

为了很方便地启动每一个节点，我们需要为每一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样本文件，为每一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，主要为每一个节点增加配置文件选项（ZOOCFG）。

修改之后，第一个节点的启动命令文件zkServer-1.cmd中的代码如下：

```
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 "";
endlocal
```

启动一个Windows的“命令提示符”窗口，进入到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，才能对外提供服务。所以，至少需要启动两个节点。

启动另外一个Windows的“命令提示符”窗口，进入到bin目录，并且启动zkServer-2.cmd，这个脚本中会启动第2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“命令提示符”窗口在服务期间，不能关闭。

启动之后，如何验证集群的启动是否成功呢？有两种方法。

方法一，可以通过执行jps命令，我们可以看到QuorumPeerMain进程的数量。

---

```
C:\devtools\ZooKeeper-3.4.13\bin >jps
1344 QuorumPeerMain
13380 QuorumPeerMain
9740 Jps
```

---

方法二，启动ZooKeeper客户端，运行并查看一下是否能连接集群。

如果最后显示出“CONNECTED”连接状态，则表示已经成功连接。当然，这个时候ZooKeeper已经启动成功了。

---

```
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
2019-05-05 21:17:36,886 [myid:] - INFO  [main-SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1:
WATCHER:::
WatchedEventstate:SyncConnectedtype:Nonepath:null
[zk: 127.0.0.1:2181(CONNECTED) 0]
```

---

在连接成功后，可以输入ZooKeeper的客户端命令，操作“ZNode”树的节点。

### 提示

在Windows下，ZooKeeper是通过.cmd命令文件中的批处理命令来运行的，默认没有提供Windows后台服务方案。为了避免每次关闭后，再一次启动ZooKeeper时还需要使用cmd带来的不便，可以通过prunsrv第三方工具来将ZooKeeper做成Windows服务，将ZooKeeper变成后台服务来运行。

## 10.2 使用ZooKeeper进行分布式存储

下面介绍一下ZooKeeper存储模型，然后介绍如何使用客户端命令来操作ZooKeeper的存储模型。

## 10.2.1 详解ZooKeeper存储模型

ZooKeeper的存储模型非常简单，它和Linux的文件系统非常的类似。简单地说，ZooKeeper的存储模型是一棵以"/"为根节点的树。ZooKeeper的存储模型中的每一个节点，叫作ZNode（ZooKeeper Node）节点。所有的ZNode节点通过树形的目录结构，按照层次关系组织在一起，构成一棵ZNode树。

每个ZNode节点都用一个以"/"（斜杠）分隔的完整路径来唯一标识，而且每个ZNode节点都有父节点（根节点除外）。例如：" /foo/bar"就表示一个ZNode节点，它的父节点为"/foo"节点，它的祖父节点的路径为"/"。"/"节点是ZNode树的根节点，它没有父节点。

通过ZNode树，ZooKeeper提供了一个多层级的树形命名空间。与文件的目录系统中的目录有所不同的是，这些ZNode节点可以保存二进制有效负载数据（Payload）。而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。

ZooKeeper为了保证高吞吐和低延迟，整个树形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，计算机的内存比较有限，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，每个节点存放的有效负载数据（Payload）的上限仅为1MB。

## 10.2.2 zkCli客户端命令清单

用zkCli.cmd（zkCli.sh）连接上ZooKeeper服务后，用help命令可以列出ZooKeeper的所有命令，如表10-1所示。

表10-1 zk的客户端常用命令介绍

zk的客户端常用命令	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

例如，使用get命令查看ZNode树的根节点“/”，具体的信息如下：

```
[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
```

所返回的节点信息主要有事务id、时间戳、版本号、数据长度、子节点的数量等。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。例如，有两个Zxid存在着zxid1<zxid2，那么说明Zxid1事务变化发生在Zxid2事务变化之前。

一个ZNode的创建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，分别是：

·cZxid：ZNode节点创建时的事务id（Transaction id）。

·mZxid：ZNode节点修改时的事务id，与子节点无关。

**·pZxid:** ZNode节点的子节点的最后一次创建或者修改的时间，与孙子节点无关。

所返回的节点信息，包含的时间戳有两个：

**·ctime:** ZNode节点创建时的时间戳。

**·mtime:** ZNode节点最新一次更新时的时间戳。

所返回的节点信息，包含的版本号有三个：

**·dataversion:** 数据版本号。

**·cversion:** 子节点版本号。

**·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的相应版本号增加。

对于ZNode节点信息的主要属性，如表10-2所示。

表10-2 ZNode节点信息的主要属性介绍

属性名称	说明
cZxid	创建节点时的 Zxid 事务 id
ctime	创建节点时的时间戳
mZxid	最后修改节点时的事务 id
mtime	最后修改节点时的时间戳
pZxid	表示该节点的子节点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，该版本号就加 1
dataversion	数据版本号，数据每次修改，该版本号就加 1
aclversion	权限版本号，权限每次修改，该版本号就加 1
dataLength	该节点的数据长度
numChildren	该节点拥有子节点的数量

## 10.3 ZooKeeper应用开发的实践

ZooKeeper应用的开发主要通过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：

- ZooKeeper官方的Java客户端API。

- 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。例如，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。不过，对于实际开发来说，ZooKeeper官方API有一些不足之处，具体如下：

- 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。

- 会话超时之后没有实现重连机制。

- 异常处理繁琐，ZooKeeper提供了很多异常，对于开发人员来说可能根本不知道应该如何处理这些抛出的异常。

- 仅提供了简单的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。

- 创建节点时如果抛出异常，需要自行检查节点是否存在。

- 无法实现级联删除。

总之，ZooKeeper官方API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。可供选择的Java客户端API之二，即第三方开源客户端API，主要有ZkClient和Curator。

### 10.3.1 ZkClient开源客户端介绍

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，例如，阿里的分布式Dubbo框架对它进行了无缝集成。

ZkClient解决了ZooKeeper原生API接口的很多问题。例如，ZkClient提供了更加简洁的API，实现了会话超时重连、反复注册Watcher等问题。虽然ZkClient对原生API进行了封装，但也有它自身的不足之处，具体如下：

- ZkClient社区不活跃，文档不够完善，几乎没有参考文档。
- 异常处理简化（抛出RuntimeException）。
- 重试机制比较难用。
- 没有提供各种使用场景的参考实现。

### 10.3.2 Curator开源客户端介绍

Curator是Netflix公司开源的一套ZooKeeper客户端框架，和ZkClient一样它解决了非常底层的细节开发工作，包括连接、重连、反复注册Watcher的问题以及NodeExistsException异常等。

Curator是Apache基金会的顶级项目之一，Curator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还为ZooKeeper客户端框架提供了一些比较普遍的、开箱即用的、分布式开发用的解决方案，例如Recipe、共享锁服务、Master选举机制和分布式计算器等，帮助开发者避免了“重复造轮子”的无效开发工作。

另外，Curator还提供了一套非常优雅的链式调用API，与ZkClient客户端API相比，Curator的API优雅太多了，以创建ZNode节点为例，让大家实际对比一下。

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ZNode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一书的作者，鼎鼎大名的Patrick Hunt都对Curator给予了高度评价，他的评语是：“Guava is to Java that 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集群操作的需求。对于ZooKeeper的客户端，我们这里只学习和研究Curator的使用，疯狂

创客圈社群的高并发项目——“IM实战项目”，最终也是通过Curator客户端来操作ZooKeeper集群的。

### 10.3.3 Curator开发的环境准备

打开Curator的官网，我们可以看到，Curator包含了以下几个包：

- curator-framework是对ZooKeeper的底层API的一些封装。
- curator-client提供了一些客户端的操作，例如重试策略等。
- 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。

以上的三个包，在使用之前，首先要在Maven的pom文件中加上包的Maven依赖。这里使用Curator的版本为4.0.0，与之对应ZooKeeper的版本为3.4.x。在pom文件中与依赖设置相关的代码如下：

```
<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的失败。如何确保Curator与ZooKeeper的具体版本是匹配的呢？可以去Curator的官网查看。

### 10.3.4 Curator客户端实例的创建

在使用curator-framework包操作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：

- 使用工厂类CuratorFrameworkFactory的静态newClient()方法。
- 使用工厂类CuratorFrameworkFactory的静态builder构造者方法。

下面实现一个通用的类，分别使用以上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
{
    /**
     * @param connectionStringzk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1s, 第二次重试等待2s, 第三次重试等待4s
        // 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);
        // 获取CuratorFramework实例的最简单方式
        // 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);
    }

    /**
     * @param connectionStringzk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 retryPolicy,
        int connectionTimeoutMs, int sessionTimeoutMs)
    {
        // 用builder方法创建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，前一个是通过newClient函数去创建，相当于是一个简化版本，只需要设置ZK集群的连接地址和重试策略。

后一个是通过CuratorFrameworkFactory.builder()函数去创建，相当于是一个复杂的版本，可以设置连接超时connectionTimeoutMs、会话超时sessionTimeoutMs等其他与会话创建相关的选项。

上面的示例程序将两种创建客户端的方式封装成了一个通用的ClientFactory连接工具类，大家可以直接使用。

### 10.3.5 通过Curator创建ZNode节点

通过Curator框架创建ZNode节点，可使用create()方法。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，最后再通过构造者实例的forPath(String znodePath,byte[] payload)方法来完成真正的节点创建。

总之，一般使用链式调用来完成节点的创建。在链式调用的最后，需要使用forPath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() {  
    //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();  
        // 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4种类型：

- (1) PERSISTENT 持久化节点
- (2) PERSISTENT\_SEQUENTIAL 持久化顺序节点
- (3) PHEMERAL 临时节

#### (4) EPHEMERAL\_SEQUENTIAL 临时顺序节点

这4种节点类型的定义和联系具体如下：

##### (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。持久化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，在ZooKeeper中，每个父节点会它的第一级子节点维护一份顺序编码，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，那么在创建节点过程中，ZooKeeper会自动为给定节点名加上一个表示顺序的数字后缀来作为新的节点名。这个顺序后缀数字的数值上限是整型的最大值。

例如，在创建节点时只需要传入节点“/test\_”，ZooKeeper自动会在“test\_”后面补充数字顺序。

##### (3) 临时节点 (EPHEMERAL)

和持久节点不同的是，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了，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。还要注意一件事，就是当客户端会话失效后，所产生的节点也不是一下子就消失了，也要过一段时间，大概是10秒以内。大家可以试一下，本机操作生成节点，在服务器端用命令来查看当前的节点数目，我们会发现有些客户端的状态已经是stop（中止），但是产生的节点还在。

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)

此节点是属于临时节点，不过带有顺序编号，客户端会话结束，所产生的节点就会消失。

### 10.3.6 在Curator中读取节点

在Curator框架中，与节点读取的有关的方法主要有三个：

- (1) 首先是判断节点是否存在，调用checkExists方法。
- (2) 其次是获取节点的数据，调用getData方法。
- (3) 最后是获取子节点列表，调用getChildren方法。

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) {
                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：

- 返回构造者实例，不会立即执行。
- 通过链式调用，在最末端调用forPath（String znodePath）方法执行实际的操作。

### 10.3.7 在Curator中更新节点

节点的更新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为止。

调用setData()方法进行同步更新，代码如下：

```
/*
 * 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，通过调用setData()方法返回一个SetDataBuilder构造者实例，执行该实例的forPath(zkPath,payload)方法即可完成同步更新操作。

如果是异步更新呢？

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。异步执行完成之后，SetDataBuilder构造者实例会再执行 AsyncCallback实例的processResult (...) 方法中的回调逻辑，即可完成更新后的其他操作。

异步更新的代码如下：

```
package com.crazymakercircle.zk.basicOperate;
import com.crazymakercircle.zk.ClientFactory;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.utils.CloseableUtils;
import org.apache.ZooKeeper.AsyncCallback;
import org.apache.ZooKeeper.CreateMode;
import org.apache.ZooKeeper.data.Stat;
import org.junit.Test;
import java.util.List;
```

```
/*
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
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);
        }
    }
}
```

### 10.3.8 在Curator中删除节点

删除节点非常简单，只需调用delete方法，实例代码如下：

```
package com.crazymakercircle.zk.basicOperate;
import com.crazymakercircle.zk.ClientFactory;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.utils.CloseableUtils;
import org.apache.ZooKeeper.AsyncCallback;
import org.apache.ZooKeeper.CreateMode;
import org.apache.ZooKeeper.data.Stat;
import org.junit.Test;
import java.util.List;
/**
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class CRUD {
    private static final String ZK_ADDRESS = "127.0.0.1:2181";
    //.....省略其他
    /**
     * 删除节点
     */
    @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，通过调用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，即可完成同步删除操作。

删除和更新操作一样，也可以异步进行。如何异步删除呢？思路是一样的：异步删除同样需要用到DeleteBuilder构造者实例的inBackground(AsyncCallback,asyncCallback)方法去设置回调，实际操作很简单，这里就不再赘述。

至此，Curator的CRUD基本操作就介绍完了。下面介绍基于Curator的基本操作，完成一些基础的分布式应用。注：CRUD是指Create（创建），Retrieve（查

询），Update（更新）和Delete（删除）。

## 10.4 分布式命名服务的实践

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顺序维护能力，来为分布式系统中的资源命名。

哪些应用场景需要用到分布式命名服务呢？典型的有：

### 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）中的文件系统的功能。借助于ZooKeeper的树形分层结构就能提供分布式的API调用功能。

著名的Dubbo分布式框架就是应用了ZooKeeper的分布式的JNDI功能。

在Dubbo中，使用ZooKeeper维护的全局服务接口API的地址列表。大致的思路为：

- 服务提供者（Service Provider）在启动的时候，向ZooKeeper上的指定节点/dubbo/\${serviceName}/providers写入自己的API地址，这个操作就相当于服务的公开。

- 服务消费者（Consumer）启动的时候，订阅节点/dubbo/{serviceName}/providers下的服务提供者的URL地址，获得所有服务提供者的API。

### 2. 分布式的ID生成器

在分布式系统中，为每一个数据资源提供唯一性的ID标识功能。

在单体服务环境下，通常来说，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，则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的节点组成，节点的数量不是固定的，而是不断动态变化的。比如说，当业务不断膨胀和流量洪峰到来时，大量的节点可能会动态加入到集群中。而一旦流量洪峰过去了，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？一种简单的办法是可以通过配置文件，手动为每一个节点命名。但是，如果节点数据量太大，或者说变动频繁，手动命名则是不现实的，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项目——“IM实战项目”，也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功能的场景远远不止这些，这里只是抛砖引玉。

### 10.4.1 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之多：

- 大量的数据记录，需要分布式ID。
- 大量的系统消息，需要分布式ID。
- 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。
- 分布式节点的命名服务，往往也需要分布式ID。
- 诸如此类.....

可以肯定的是，传统的数据库自增主键或者单体的自增主键，已经不能满足需求。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生成系统是基础系统，被许多关键系统调用，一旦宕机，就会造成严重影响。

有哪些分布式的ID生成器方案呢？简单的梳理一下，大致如下：

- Java的UUID。
- 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。
- Twitter的SnowFlake算法。
- ZooKeeper生成ID：利用ZooKeeper的顺序节点，生成全局唯一的ID。
- MongoDb的ObjectId：MongoDB是一个分布式的非结构化NoSQL数据库，每插入一条记录会自动生成全局唯一的一个“\_id”字段值，它是一个12字节的字符串，可以作为分布式系统中全局唯一的ID。

以上方案有哪些利弊呢？首先，分析一下Java语言中的UUID方案。UUID是Universally Unique Identifier的缩写，它是在一定的范围内（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是经由一定的算法机器生成的，为了保证UUID的唯一性，规范定义了包括网卡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。UUID只能由计算机生成。

一个UUID是16字节长的数字，一共128位。转成字符串之后，它会变成一个36字节的字符串，例如：3F2504E0-4F89-11D3-9A0C-0305E82C3301。使用的时候，可以把中间的4个连字符去掉，剩下32字节的字符串。

UUID的优点是本地生成ID，不需要进行远程调用，时延低，性能高。

UUID的缺点是UUID过长，16字节共128位，通常以36字节长的字符串来表示，在很多应用场景不适用，例如，由于UUID没有排序，无法保证趋势递增，因此用于数据库索引字段的效率就很低，添加记录存储入库时性能差。

所以，对于高并发和大数据量的系统，不建议使用UUID。

## 10.4.2 ZooKeeper分布式ID生成器的实践案例

大家知道，在ZooKeeper节点的四种类型中，其中有以下两种具备自动编号的能力：

- PERSISTENT\_SEQUENTIAL 持久化顺序节点。
- EPHEMERAL\_SEQUENTIAL 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，这个顺序编号是分布式同步的，也是全局唯一的。

在创建子节点时，如果设置为上面的类型，ZooKeeper会自动为创建后的节点路径在末尾加上一个数字，用来表示顺序。这个顺序值的最大上限就是整型的最大值。

例如，在创建节点的时候只需要传入节点“/test\_”，ZooKeeper自动会在“test\_”后面补充数字顺序，例如“/test\_0000000010”。

通过创建ZooKeeper的临时顺序节点的方法，生成全局唯一的ID，演示代码如下：

---

```
package com.crazymakercircle.zk.NameService;
import com.crazymakercircle.zk.ClientFactory;
import org.apache.curator.framework.CuratorFramework;
import org.apache.ZooKeeper.CreateMode;
/**
 * 生成分布式ID
 * create by 尼恩 @ 疯狂创客圈
 */
public class IDMaker {
    //...省略其他的方法
    /**
     * 创建临时顺序节点
     * @param pathPrefix 节点路径
     * @return 创建后的完整路径名称
     */
    private String createSeqNode(String pathPrefix) {
        try {
            // 创建一个ZNode顺序节点
            // 为了避免ZooKeeper的顺序节点暴增，建议创建之后，删除创建的节点
            String destPath = client.create()
                .creatingParentsIfNeeded()
                .withMode(CreateMode.EPHEMERAL_SEQUENTIAL)
                .forPath(pathPrefix);
            return destPath;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
    //创建id
    public String makeId(String nodeName) {
        String str = createSeqNode(nodeName);
        if (null == str) {
            return null;
        }
    }
}
```

```
        }
        //取得ZooKeeper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;
}

```

---

节点创建完成之后，会返回节点的完整路径，生成的序号放置在路径的末尾，一般为10位数字字符。可以通过截取路径末尾的数字作为新生成的ID。

基于自定义的IDMaker，编写单元测试用例来生成ID，代码如下：

```
/**
 * create by 尼恩 @ 疯狂创客圈
 */
@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();
    }
    //...省略其他的用例
}
```

---

下面是这个测试用例程序的部分运行结果：

```
第0个创建的id为:0000000010
第1个创建的id为:0000000011
//....省略其他的输出
```

---

### 10.4.3 集群节点的命名服务之实践案例

节点的命名主要是为节点进行唯一编号。主要的诉求是不同节点的编号是绝对不能重复的。一旦编号重复，就会有不同的节点发生碰撞而导致集群异常。

前面讲到在分布式集群中，可能需要部署大量的机器节点。在节点少的时候，节点的命名，可以手工完成。然而，在节点数量大的场景下，手工命名的维护成本高，还需要考虑到自动部署、运维等等问题，因此手工去命名不现实。总之，节点的命名最好由系统自动维护。

有以下两个方案可用于生成集群节点的编号：

- (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持久顺序节点的顺序特性来维护节点的NodeId编号。

下面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：

·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，就创建系统的根节点。

·在根节点下创建一个临时顺序ZNode节点，取回ZNode的编号把它作为分布式系统中节点的NODEID。  
·如果临时节点太多，可以根据需要删除临时顺序ZNode节点。

集群节点命名服务的主要实现代码如下：

```
/*
 * 集群节点的命名服务
 * create by 尼恩 @ 疯狂创客圈
 */
public class PeerNode {
    //ZooKeeper客户端
    private CuratorFramework client = null;
    private String pathRegistered = null;
    private Node node = null;
    private static PeerNodesingleInstance = null;
    //唯一实例模式
    public static PeerNodegetInst() {
        if (null == singleInstance) {
            singleInstance = new PeerNode();
            singleInstance.client =
                ZKclient.instance.getClient();
            singleInstance.init();
        }
        return singleInstance;
    }
}
```

```
    }
    private PeerNode() {
    }
    // 初始化，在ZooKeeper中创建当前的分布式节点
    public void init() {
        //使用标准的前缀，创建父节点，父节点是持久化的
        createParentIfNeeded(ServerUtils.MANAGE_PATH);
        // 创建一个ZNode节点
        try {
            pathRegistered = client.create()
                .creatingParentsIfNeeded()
                //创建一个非持久化的临时节点
                //临时节点的前缀，也需要提前定义
                .withMode(CreateMode.EPHEMERAL_SEQUENTIAL)
                .forPath(ServerUtils.pathPrefix);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //获取节点的编号
    public long getId() {
        String sid = null;
        if (null == pathRegistered) {
            throw new RuntimeException("节点注册失败");
        }
        int index = pathRegistered.lastIndexOf(ServerUtils.pathPrefix);
        if (index >= 0) {
            index += ServerUtils.pathPrefix.length();
            sid = index <= pathRegistered.length() ?
                pathRegistered.substring(index) : null;
        }
        if (null == sid) {
            throw new RuntimeException("分布式节点错误");
        }
        return Long.parseLong(sid);
    }
}
```

---

#### 10.4.4 使用ZK实现SnowFlakeID算法的实践案例

Twitter（推特）的SnowFlake算法是一种著名的分布式服务器用户ID生成算法。SnowFlake算法所生成的ID是一个64bit的长整型数字，如图10-2所示。这个64bit被划分成四个部分，其中后面三个部分分别表示时间戳、工作机器ID、序列号。

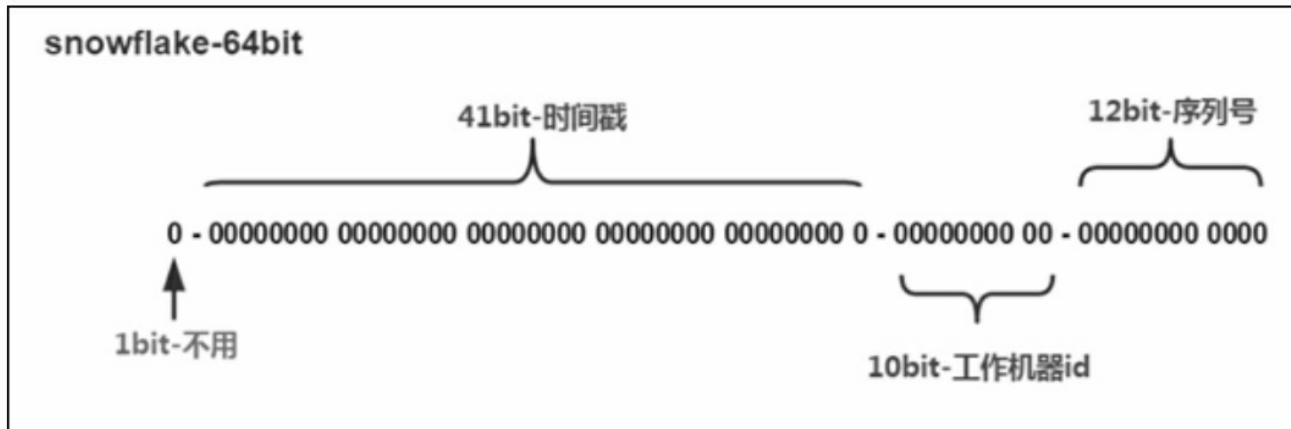


图10-2 SnowFlakeID的四个部分

SnowFlakeID的四个部分，具体介绍如下：

- (1) 第一位 占用1 bit，其值始终是0，没有实际作用。
- (2) 时间戳 占用41 bit，精确到毫秒，总共可以容纳约69年的时间。
- (3) 工作机器id 占用10 bit，最多可以容纳1024个节点。
- (4) 序列号 占用12 bit，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多少个全局唯一ID呢？同一毫秒的ID数量大致为：  
 $1024 * 4096 = 4194304$ ，总计400多万个ID，也就是说，在绝大多数并发场景下都是够用的。

在SnowFlake算法中，第三个部分是工作机器ID，可以结合上一节的命名方法，并通过ZooKeeper管理workId，免去了手动频繁修改集群节点去配置机器ID的麻烦。

上面的bit数分配只是一个官方的推荐，是可以微调的。比方说，如果1024的节点数不够，可以增加3个bit，扩大到8192个节点；再比方说，如果每毫秒生成4096

个ID比较多，可以从12 bit减小到使用10 bit，则每毫秒生成1024个ID。这样，单个节点1秒可以生成 $1024 \times 1000$ ，也就是100多万个ID，数量也是巨大的；剩下的位数为剩余时间，还剩下40 bit的时间戳，比原来少1位，则可以持续32年。按照以上的方式来实现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 {
    /**
     * 唯一实例
     */
    public static SnowflakeIdGenerator instance =
        new SnowflakeIdGenerator();
    /**
     * 初始化唯一实例
     *
     * @param workerId 节点Id, 最大8191
     * @return 这个唯一实例
     */
    public synchronized void init(long workerId) {
        if (workerId > MAX_WORKER_ID) {
            // ZooKeeper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bit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数为1024
     */
    private final static int SEQUENCE_BITS = 10;
    /**
     * 最大的 worker id, 8091
     * -1 的补码 (二进制数的所有位均为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 (二进制数的所有位均为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号
 */
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
        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，那么阻塞到下一个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个毫秒
        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
    // 时间戳右移动23位
    long time = (current - START_TIME) << TIMESTAMP_LEFT_SHIFT;
    // workerId右移动10位
    long workerId = this.workerId << APP_HOST_ID_SHIFT;
    return time | workerId | sequence;
}
/**
 * 阻塞到下一个毫秒
 */
private long nextMs(long timeStamp) {
    long current = System.currentTimeMillis();
    while (current <= timeStamp) {
        current = System.currentTimeMillis();
    }
    return current;
}
}

```

---

上面的代码中大量使用到了二进制位运算，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位二进制补码为1111 1111，也就是二进制位全为1。为什么呢？因为在8位二进制的应用场景下，-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，计算后的结果是：-1的二进制补码的各个位全为1。16位、32位、64位的-1，与8位二进制相同，它们的二进制补码也是各个位全为1。

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，可以去查看有关Java的基础书籍。

上面的代码是一个相对比较简单的SnowFlake实现版本，现在对其中的关键算法解释如下：

·在单节点上获得下一个ID，使用Synchronized控制并发，没有使用CAS（Compare And Swap，比较并交换）的方式，是因为CAS不适合并发量非常高的场景；

·如果在一台机器上当前毫秒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；

·如果当前时间小于记录的上一个毫秒值，则说明这台机器的时间回拨了，于是阻塞，一直等到下一毫秒。

编写一个测试用例，测试一下SnowflakeIdGenerator，代码如下：

```
package com.crazymakercircle.zk.NameService;
import lombok.extern.slf4j.Slf4j;
import org.junit.Test;
import java.util.Collections;
import java.util.HashSet;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
/*
 * 测试SnowflakeIdWorker、SnowflakeIdGenerator
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class SnowflakeIdTest {
    /**
     * 测试用例
     */
    @Test
    public void snowflakeIdTest() {
        //或者节点的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("开始生产 *");
        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("* 耗时: " + (end - start) + " ms!");
    }
}
```

```
    }
```

---

在测试用例中，用到了SnowflakeIdWorker节点的命令服务，并通过它取得了节点workerId。

SnowFlake算法的优点：

- 生成ID时不依赖于数据库，完全在内存生成，高性能和高可用性。
- 容量大，每秒可生成几百万个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：

- 依赖于系统时钟的一致性，如果某台机器的系统时钟回拨了，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，那么有可能出现ID重复的危险。

## 10.5 分布式事件监听的重点

实现对ZooKeeper服务器端的事件监听，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：

- (1) 一种是标准的观察者模式；
- (2) 另一种是缓存监听模式。

第一种标准的观察者模式是通过Watcher监听器实现的；第二种缓存监听模式是通过引入了一种本地缓存视图Cache机制去实现的。

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Zk集群的ZNode状态变化时，会触发事件，注册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实现实际的监听。另外，Cache机制提供了反复注册的能力，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3种：Path Cache，Node Cache和Tree Cache。

### 10.5.1 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);
    }
};
```

如何使用Watcher监听器实例呢？可以过GetDataBuilder、GetChildrenBuilder和ExistsBuilder等这类实现了Watchable<T>接口的构造者，然后使用构造者的usingWatcher(Watcher w)方法，为构造者设置Watcher监听器实例。

在Curator中，Watchable<T>接口的源代码如下：

```
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和ExistsBuilder构造者，分别通过getData()、getChildren()和checkExists()三个方法返回，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监听器在向服务器端完成注册后，当服务器端的一些事件触发了这个Watcher，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客户端收到服务器端的通知后，会封装一个WatchedEvent事件实例，再传递给监听器的process（WatchedEvent e）回调方法。

来看一下通知事件WatchedEvent实例的类型，WatchedEvent包含了三个基本属性：

- 通知状态（keeperState）
- 事件类型（EventType）
- 节点路径（path）

### 说明

WatchedEvent并不是从ZooKeeper集群直接传递过来的事件实例，而是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。从ZooKeeper服务器端直接通过网络传输传递过来的事件实例其实是一个WatcherEvent类型的实例，WatchedEvent传输实例和Curator的WatchedEvent封装实例，在名称上基本上一样，只有一个字母之差，而且功能也是一样的，都表示的是同一个服务器端事件。

这里聚焦一下，只讲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。Watcher接口中定义的通知状态和事件类型，具体如表10-3所示。

表10-3 Watcher接口中定义的通知状态和事件类型

KeeperState	EventType	触发条件	说明
	None (-1)	客户端与服务器端成功建立连接	
SyncConnected (0)	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，1：使用错误的 schema 进行权限检查 2：SASL 权限检查失败	通常同时也会收到 AuthFailedException 异常

利用Watcher来对节点事件进行监听，来看一个简单的实例程序：

```

package com.crazymakercircle.zk.publishSubscribe;
import com.crazymakercircle.zk.ZKclient;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.cache.*;
import org.apache.ZooKeeper.WatchedEvent;
import org.apache.ZooKeeper.Watcher;
import org.junit.Test;
import java.io.UnsupportedEncodingException;
/*
 * 客户端监听实践
 * 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-";
    //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 event) {

```

```
        System.out.println("监听到的变化watchedEvent = " +  
                           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1次更改内容".getBytes());  
// 第二次变更节点数据  
client.setData().forPath(workerPath, "第2次更改内容".getBytes());  
Thread.sleep(Integer.MAX_VALUE);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}  
}  
}
```

---

运行上面的程序代码，输出的结果如下：

```
//....省略其他的输出  
监听到的变化watchedEvent = WatchedEventstate:SyncConnectedtype:NodeDataChanged path:/test/listener,
```

---

在这个程序中，节点路径“/test/listener/node”注册了一个Watcher监听器实例，随后调用setData方法改变节点内容，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，监听器的注册是一次性的，当第二次改变节点内容时，注册已经失效，无法再次捕获节点的变动事件。

既然Watcher监听器是一次性的，如果要反复使用，怎么办呢？需要反复地通过构造者的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，例如会话超时、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服务器端的事件。Cache机制对ZooKeeper事件监听进行了封装，能够自动处理反复注册监听。

## 10.5.2 NodeCache节点缓存的监听

Curator引入的Cache缓存机制的实现拥有一个系列的类型，包括了Node Cache、Path Cache和Tree Cache三组类。其中：

- (1) Node Cache节点缓存可用于ZNode节点的监听；
- (2) Path Cache子节点缓存可用于ZNode的子节点的监听；
- (3) Tree Cache树缓存是Path Cache的增强，不光能监听子节点，还能监听ZNode节点自身。

先看Node Cache，用于监控节点的增加，删除和更新。

Node Cache使用的第一步，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)
```

第一个参数就是传入创建的Curator的框架客户端，第二个参数就是监听节点的路径，第三个重载参数dataIsCompressed表示是否对数据进行压缩。

NodeCache使用的第二步，就是构造一个NodeCacheListener监听器实例。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。实例代码如下：

```
NodeCacheListener listener = new NodeCacheListener() {
    @Override
    public void nodeChanged() throws Exception {
        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());
    }
};
```

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，使用缓存实例的addListener方法，然后使用缓存实例nodeCache的start方法来启动节点的事件监听。

```
//启动节点的事件监听  
nodeCache.getListenable().addListener(listener);  
nodeCache.start();
```

再强调一下，需要调用nodeCache的start方法进行缓存和事件监听，这个方法有两个版本：

```
void start()//Start the cache.  
void start(boolean buildInitial) //true代表缓存当前节点
```

唯一的一个参数buildInitial代表着是否将该节点的数据立即进行缓存。如果设置为true的话，在start启动时立即调用NodeCache的getCurrentData方法就能够得到对应节点的信息ChildData类，如果设置为false，就得不到对应的信息。

使用NodeCache来监听节点的事件，完整的实例代码如下：

```
package com.crazymakercircle.zk.publishSubscribe;  
import com.crazymakercircle.zk.ZKclient;  
import lombok.Data;  
import lombok.extern.slf4j.Slf4j;  
import org.apache.curator.framework.CuratorFramework;  
import org.apache.curator.framework.recipes.cache.*;  
import org.apache.ZooKeeper.WatchedEvent;  
import org.apache.ZooKeeper.Watcher;  
import org.junit.Test;  
import java.io.UnsupportedEncodingException;  
/**  
 * 客户端监听实践  
 * 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() throws Exception {  
                    ChildData childData = nodeCache.getCurrentData();  
                    log.info("ZNode节点状态改变, path={}", childData.getPath());  
                }  
            };  
            nodeCache.getListenable().addListener(listener);  
            nodeCache.start();  
        } catch (Exception e) {  
            log.error("启动失败", e);  
        }  
    }  
}
```

```
        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1次变更节点数据
    client.setData().forPath(workerPath, "第1次更改内容".getBytes());
    Thread.sleep(1000);
    // 第2次变更节点数据
    client.setData().forPath(workerPath, "第2次更改内容".getBytes());
    Thread.sleep(1000);
    // 第3次变更节点数据
    client.setData().forPath(workerPath, "第3次更改内容".getBytes());
    Thread.sleep(1000);
    // 第4次变更节点数据
    client.delete().forPath(workerPath);
    Thread.sleep(Integer.MAX_VALUE);
} catch (Exception e) {
    log.error("创建NodeCache监听失败, path={}", workerPath);
}
}
}
```

---

通过运行的结果我们可以看到，NodeCache节点缓存能够重复地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3次更改内容
- ZNode节点状态改变, stat=17179869191,
...
```

---

最后说明一下，如果在监听的时候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。

### 10.5.3 PathChildrenCache子节点监听

PathChildrenCache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：

- 只能监听子节点，监听不到当前节点。
- 不能递归监听，子节点下的子节点不能递归监控。

PathChildrenCache子节点缓存使用的第一步就是构造一个缓存实例。PathChildrenCache有多个重载版本的构造方法，下面选择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)  
//重载版本三  
public PathChildrenCache(CuratorFramework client, String path, boolean cacheData, boolean dataIsCompressed)  
//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。

- 第一个参数就是传入创建的Curator框架的客户端。
- 第二个参数就是监听节点的路径。
- 第三个重载参数cacheData表示是否把节点的内容缓存起来。如果cacheData为true，那么接收到节点列表变更事件的同时会将获得节点内容。

除了上边的三个参数，其他参数的说明如下：

- dataIsCompressed参数，表示是否对节点数据进行压缩。
- threadFactory参数表示线程池工厂，当PathChildrenCache内部需要启动新的线程执行时，使用该线程池工厂来创建线程。
- executorService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。

构造完缓冲实例之后，PathChildrenCache缓存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;
}
```

---

在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的使用实例如下：

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

---

在创建完PathChildrenCacheListener的实例之后，需要将这个实例注册到PathChildrenCache缓存实例，在调用缓存实例的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。

start方法可以传入启动的模式，定义在StartMode枚举中，具体如下：

- NORMAL——异步初始化cache
- BUILD\_INITIAL\_CACHE——同步初始化cache
- POST\_INITIALIZED\_EVENT——异步初始化cache，并触发完成事件

StartMode枚举的三种启动方式，详细说明如下：

(1) BUILD\_INITIAL\_CACHE模式：启动时同步初始化cache，表示创建cache后就从服务器提取对应的数据；

(2) POST\_INITIALIZED\_EVENT模式：启动时异步初始化cache，表示创建cache后从服务器提取对应的数据，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。

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;
import com.crazymakercircle.zk.ZKclient;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.cache.*;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.junit.Test;
import java.io.UnsupportedEncodingException;
/**
 * 客户端监听实践
 * 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-";
    /**
     * 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 =
                new PathChildrenCacheListener() {
                    @Override
                    public void childEvent(CuratorFramework client,
                        PathChildrenCacheEvent event) {
                        try {
                            ChildData data = event.getData();
                            switch (event.getType()) {
                                //子节点增加
                                case CHILD_ADDED:
                                    log.info("子节点增加, path={}, data={}",
                                        data.getPath(),
                                        new String(data.getData(), "UTF-8"));
                                    break;
                                //子节点更新
                                case CHILD_UPDATED:
                                    log.info("子节点更新, path={}, data={}",
                                        data.getPath(),
                                        new String(data.getData(), "UTF-8"));
                                    break;
                                //子节点删除
                                case CHILD_REMOVED:
                                    log.info("子节点删除, path={}, data={}",
                                        data.getPath(),
                                        new String(data.getData(), "UTF-8"));
                            }
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                };
            cache.getListenable().addListener(listener);
            client.start();
            Thread.sleep(10000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        break;
    default:
        break;
    }
} catch (
    UnsupportedEncodingException e) {
    e.printStackTrace();
}
}
};

//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3个子节点
for (int i = 0; i< 3; i++) {
    ZKclient.instance.createNode(subWorkerPath + i, null);
}
Thread.sleep(1000);
//删除3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能够反复地监听到节点的增加和删除。

Curator监听的原理：无论是PathChildrenCache，还是TreeCache，所谓的监听都是在进行Curator本地缓存视图和ZooKeeper服务器远程的数据节点的对比，并且在进行数据同步时会触发相应的事件。

这里，以NODE\_ADDED（节点增加事件）的触发为例来进行说明。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。为什么呢？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，也会触发本地缓存的NODE\_ADDED事件。

至此，已经讲完了两个系列的缓存监听。简单回顾一下：

- (1) Node Cache用来观察ZNode自身，如果ZNode节点本身被创建，更新或者删除，那么Node Cache会更新缓存，并触发事件给注册的监听器。Node Cache是通

过NodeCache类来实现的，监听器对应的接口为NodeCacheListener。

(3) Path Cache子节点缓存用来观察ZNode的子节点、并缓存子节点的状态，如果ZNode的子节点被创建，更新或者删除，那么Path Cache会更新缓存，并且触发事件给注册的监听器。Path Cache是通过PathChildrenCache类来实现的，监听器注册是通过PathChildrenCacheListener。

## 10.5.4 Tree Cache节点树缓存

最后的一个系列是Tree Cache。

Tree Cache可以看作是Node Cache和Path Cache的合体。Tree Cache不光能监听子节点，还能监听节点自身。

Tree Cache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)
```

第一个参数就是传入创建的Curator框架的客户端，第二个参数就是监听节点的路径，其他参数的简单说明如下：

- `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，在一般情况下，使用TreeCache的第一个构造函数即可。

TreeCache使用的第二步就是构造一个TreeCacheListener监听器实例。该接口的定义如下：

```
//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监听器接口中也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的使用实例如下：

```
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.getPath(), new String(data.getData(), "UTF-8"));
                    break;
                case NODE_UPDATED:
                    log.info("[TreeCache]节点更新, path={}, data={}",
                            data.getPath(), new String(data.getData(), "UTF-8"));
                    break;
                case NODE_REMOVED:
                    log.info("[TreeCache]节点删除, 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，调用缓存实例的addListener方法，将TreeCacheListener监听器实例注册到TreeCache缓存实例。然后调用缓存实例nodeCache的start方法来启动节点的事件监听。整个实例的代码如下：

```
package com.crazymakercircle.zk.publishSubscribe;
import com.crazymakercircle.zk.ZKclient;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.cache.*;
import org.apache.ZooKeeper.WatchedEvent;
import org.apache.ZooKeeper.Watcher;
import org.junit.Test;
import java.io.UnsupportedEncodingException;
/**
 * 客户端监听实践
 * 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-";
    /**
     * Tree Cache不光能监听子节点，还能监听节点自身
     */
    @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 = 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.getPath(), new String(data.getData(),
                                "UTF-8"));
                        break;
                    case NODE_UPDATED:
                        log.info("[TreeCache] 节点更新, path={}, data={},",
                                data.getPath(), new String(data.getData(),
                                "UTF-8"));
                        break;
                    case NODE_REMOVED:
                        log.info("[TreeCache] 节点删除, path={}, data={},",
                                data.getPath(), new String(data.getData(),
                                "UTF-8"));
                        break;
                    default:
                        break;
                }
            } catch (
                UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }
    };
    //设置监听器
    treeCache.getListenable().addListener(listener);
    //启动缓存视图
    treeCache.start();
    Thread.sleep(1000);
    //创建3个子节点
    for (int i = 0; i < 3; i++) {
        ZKclient.instance.createNode(subWorkerPath + i, null);
    }
    Thread.sleep(1000);
    //删除3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, 说明一下TreeCacheEvent的事件类型, 具体为:

- 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。

对比TreeCacheEvent的事件类型与Path Cache的事件类型是不同的。回忆一下, Path Cache的事件类型具体如下:

- 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。

## 10.6 分布式锁的原理与实践

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`（同步）或者其他同一个JVM内Lock机制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，就需要一种更加高级的锁机制来处理跨机器的进程之间的数据同步问题。这种跨机器的锁就是分布式锁。

### 10.6.1 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，还是从具体的实例入手吧！这里尽量用一个简单的故事来类比，估计就简单多了。

故事发生在一个没有自来水的古代，在一个村子里有一口井，水质非常的好，村民们都在抢着取井里的水。井就那么一口，村里的人很多，村民为争抢取水打架斗殴，甚至头破血流。

问题总是要解决，于是村长绞尽脑汁最终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) 号排在前面的人，就可以先取水。
- (3) 先到的人排在前面，那些后到的人，一个一个挨着在井边排成一队。取水示意图，如图10-3所示。

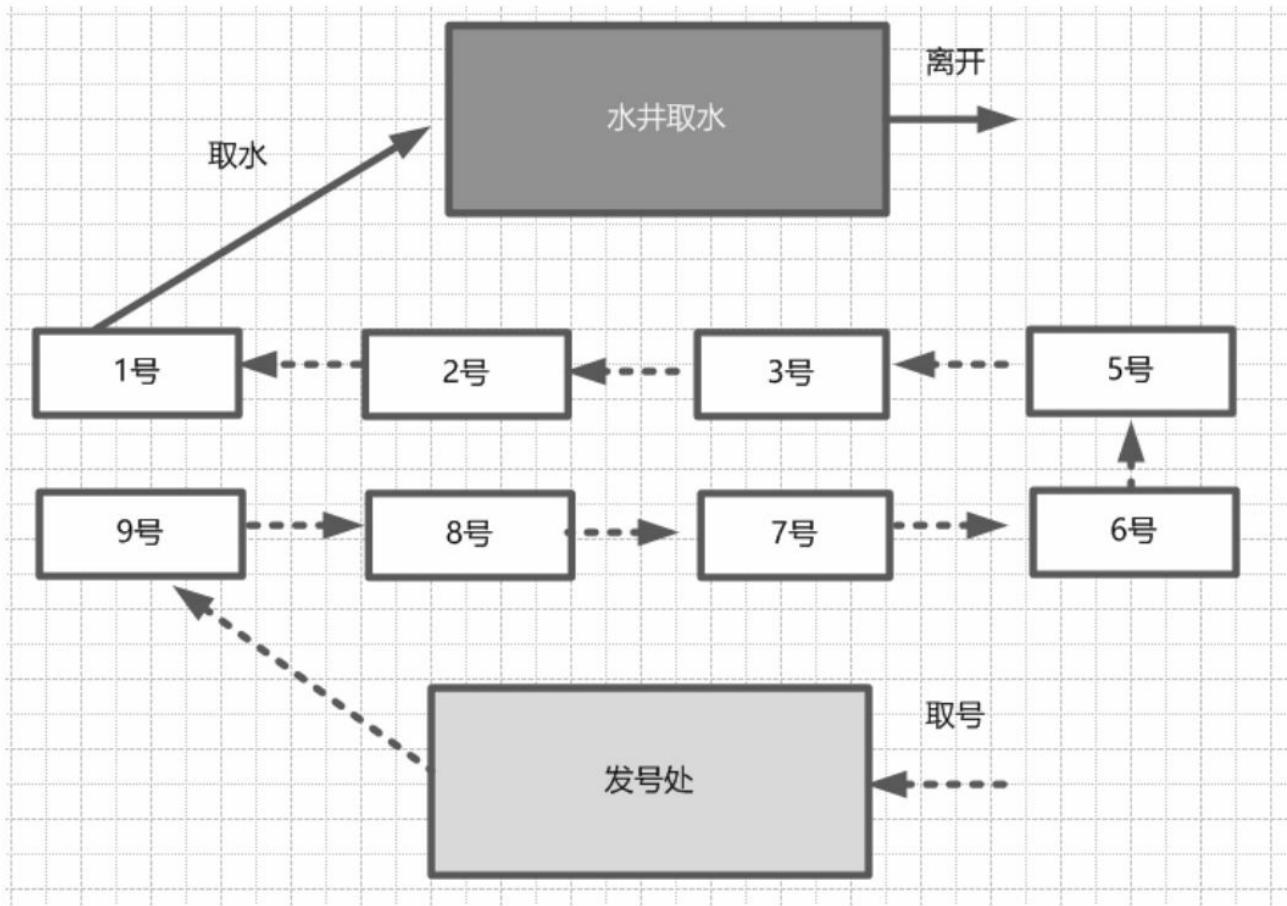


图10-3 排队取水示意图

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。

什么是可重入锁呢？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，这时候不用再取号，如图10-4所示。

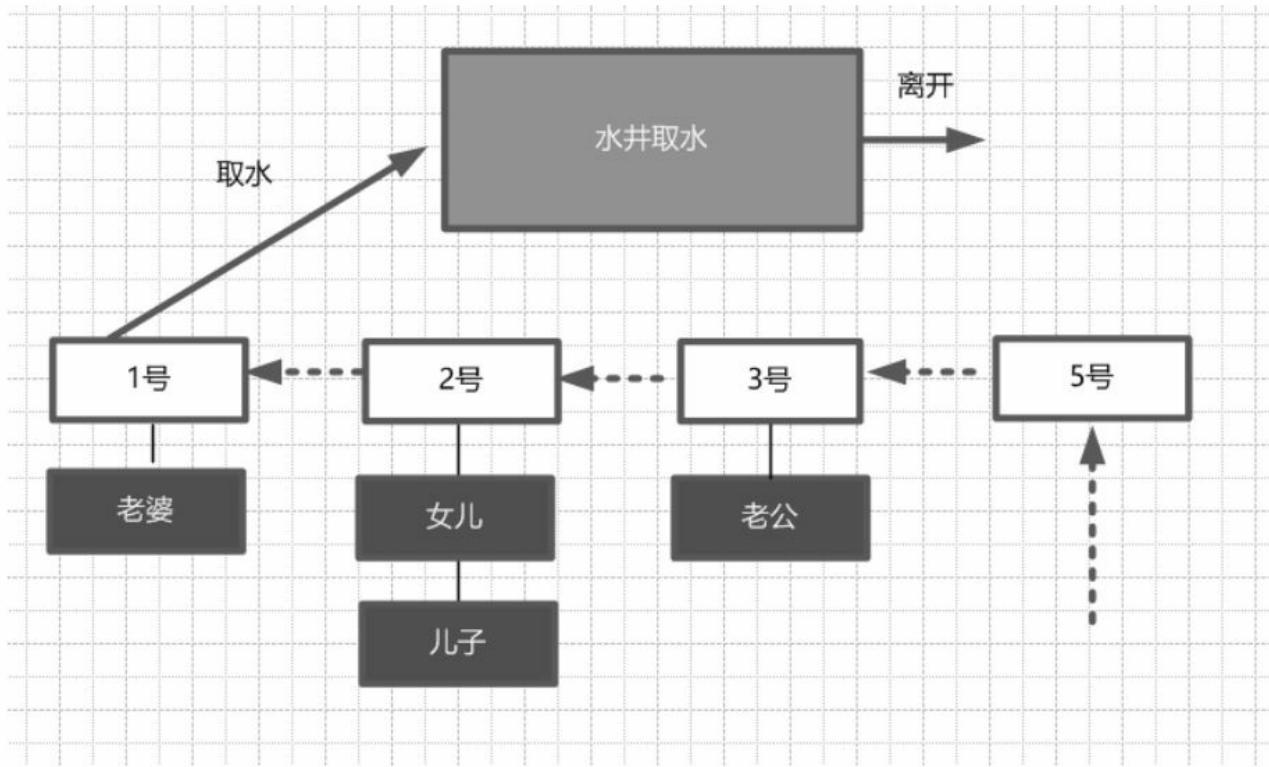


图10-4 同一家庭的人不需要重复排队

在图10-4中，排在1号的家庭，老公有号，他的老婆来了直接排第一个，“妻凭夫贵”。再看上图中的2号，父亲正在打水，他的儿子和女儿也到井边了，直接排第二个，这个叫作“子凭父贵”。总之，如果是同一个家庭，可以直接复用排号，不用重新取号从后面排起。

以上这个故事的模型，只要满足条件，同一个取水号，可以用来多次取水，相当于重入锁的模型。在重入锁的模型中，一把独占锁可以被多次锁定，这就叫作可重入锁。

## 10.6.2 ZooKeeper分布式锁的原理

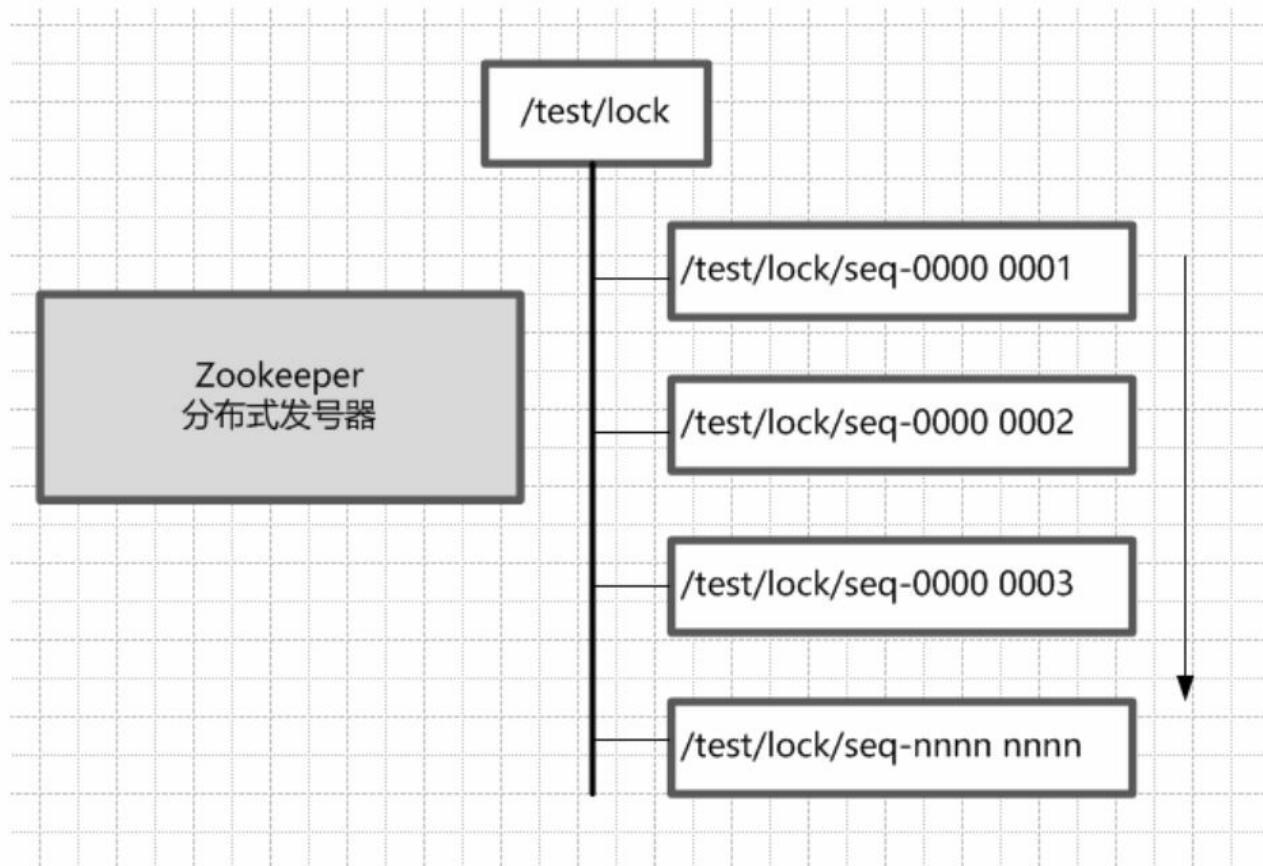
理解了经典的公平可重入锁的原理后，再来看在分布式场景下的公平可重入锁的原理。

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顺序编号。这个顺序编号是在上一个生成的顺序编号加1。

例如，用一个发号的节点“/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-”。如果是第一个创建的子节点，那么生成的子节点为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，依次类推，如图10-5所示。



## 图10-5 ZooKeeper临时顺序节点的发号器作用

### 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，首先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个临时顺序节点。由于Zk节点是按照创建的顺序依次递增的，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。因此，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的，如果是，则获取锁。

### 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，先抢号创建自己的ZNode。同样，释放锁的时候，就需要删除抢号的ZNode。在抢号成功之后，如果不是排号最小的节点，就处于等待通知的状态。等谁的通知呢？不需要其他人，只需要等前一个ZNode的通知即可。当前一个ZNode被删除的时候，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传递。

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，只需要监听或者监视排号在自己前面那个且紧挨在自己前面的那个节点。只要上一个节点被删除了，就再进行一次判断，看看自己是不是序号最小的那个节点，如果是，则获得锁。

ZooKeeper内部优越的机制，能保证由于网络异常或者其他原因造成集群中占用锁的客户端失联时，锁能够被有效释放。一旦占用锁的ZNode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。所以，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了，所有节点都去监听，然后作出反应，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，只有它后面的那个节点才作出反应。

### 10.6.3 分布式锁的基本流程

接下来就基于ZooKeeper实现一下分布式锁。

首先，定义了一个锁的接口Lock，很简单，仅仅两个抽象方法：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，流程大致如下：

(1) 一把锁，使用一个ZNode节点表示，如果锁对应的ZNode节点不存在，那么先创建ZNode节点。这里假设为“/test/lock”，代表了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，所有子节点尽量共用一个有意义的子节点前缀。

如果子节点前缀为“/test/lock/seq-”，则第一个客户端对应的子节点为“/test/lock/seq-000000000”，第二个为“/test/lock/seq-000000001”，以此类推。

如果子节点前缀为“/test/lock/”，则第一个客户端对应的子节点为“/test/lock/000000000”，第二个为“/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，需要进行判断：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，则认为加锁成功；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的变更通知，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，则认为加锁成功；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。在完成业务流程后，删除自己对应的子节点，完成释放锁的工作，以便后面的节点能捕获到节点的变更通知，获得分布式锁。

#### 10.6.4 加锁的实现

在Lock接口中，加锁的方法是lock()。lock()方法的大致流程是，首先尝试着去加锁，如果加锁失败就去等待，然后再重复。代码如下：

```
package com.crazymakercircle.zk.distributedLock;
import com.crazymakercircle.zk.ZKclient;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.ZooKeeper.WatchedEvent;
import org.apache.ZooKeeper.Watcher;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
/**
 * 分布式锁的实现
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
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;
}
//...省略其他的方法
}

```

---

尝试加锁的tryLock方法是关键，它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，就找到前一个ZNode节点，并且把它的路径保存到prior\_path。

tryLock()方法的代码如下：

```

package com.crazymakercircle.zk.distributedLock;
//.... 省略import
/**
 * 分布式锁的实现
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class ZkLock implements Lock {
    /**
     * 尝试加锁
     *
     * @return 是否加锁成功
     * @throws Exception 异常
     */
    private boolean tryLock() throws Exception {
        //创建临时ZNode节点
        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;
    }
    //...省略其他的方法
}
```

---

创建临时顺序节点后，它的完整路径存放在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？因为获取的远程子节点列表中的其他路径都是短路径，只有最后一层。为了方便进行比较，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，判断是否锁定成功。如果锁定成功，则返回true；如果自己没有获得锁，则要监听前一个节点。找出前一个节点的路径，保存在prior\_path成员中，供后面的await()等待方法用于监听。在介绍await()等待方法之前，先说下checkLocked锁定判断方法。

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，返回true，表示加锁成功。
- 如果不是，说明有其他线程早已先持有了锁，返回false。

checkLocked()方法的代码如下：

```
package com.crazymakercircle.zk.distributedLock;
//.... 省略import
/**
 * 分布式锁的实现
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
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;
    }
    //...省略其他的方法
}
```

---

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，代表自己已经获得了锁。如果不是，则会返回false。

如果checkLocked()为false，那么外层的调用方法一般会执行等待方法await()执行争夺锁失败以后的等待逻辑。

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
/**
 * 分布式锁的实现
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class ZkLock implements Lock {
    /**
     * 等待
     * 监听前一个节点的删除事件
     * @throws Exception 可能会有Zk异常、网络异常
     */
    private void await() throws Exception {
        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
        //订阅比自己编号小的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);
        //监听方式二： TreeCache订阅
        //订阅比自己编号小的顺序节点的删除事件
        /*
        TreeCachetreeCache = new TreeCache(client, prior_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:
                            log.debug("[TreeCache]节点删除, path={}, data={}",
                                    data.getPath(), data.getData());
                            latch.countDown();
                            break;
                        default:
                            break;
                    }
                }
            }
        };
        //开始监听
        treeCache.getListenable().addListener(l);
        treeCache.start();
        */
        //限时等待，最长加锁时间为3s
        latch.await(WAIT_TIME, TimeUnit.SECONDS);
```

```
    }
    //...省略其他的方法
}
```

---

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，以提升效率。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。

在上面的代码中，监听前一个节点的删除可以使用两种监听方式：

- Watcher订阅。
- TreeCache订阅。

两种方式的效果都差不多。但是，这里的删除事件只需要监听一次即可，不需要反复监听，所以，建议使用Watcher一次性订阅。而程序中有关TreeCache订阅的代码已经被注释，供大家参考。一旦前一个节点prior\_path被删除，那么就将线程从等待状态唤醒，重新开始一轮锁的争夺，直到获取锁，再完成业务处理。

至此，关于lock加锁的算法还差一点点就介绍完成。这一点，就是实现锁的可重入。什么是可重入呢？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，表示加锁成功。至此，lock()方法已经介绍完了，接下来，就是去释放锁。

## 10.6.5 释放锁的实现

Lock接口中的unLock()方法用于释放锁，释放锁主要有两个工作：

- 减少重入锁的计数，如果不是0，直接返回，表示成功地释放了一次。
- 如果计数器为0，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
/**
 * 分布式锁的实现
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
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，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;
    }
    //...省略其他的方法
}
```

在程序中，为了尽量保证线程的安全，可重入计数器的类型不是int类型，而是Java并发包中的原子类型——AtomicInteger。

## 10.6.6 分布式锁的使用

编写一个用例，测试一下ZLock的使用，代码如下：

```
package com.crazymakercircle.zk.distributedLock;
import com.crazymakercircle.concurrent.FutureTaskScheduler;
import com.crazymakercircle.zk.ZKclient;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.locks.InterProcessMutex;
import org.junit.Test;
/**
 * 测试分布式锁
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class ZkLockTester {
    //需要锁来保护的公共资源
    //变量
    int count = 0;
    /**
     * 测试自定义分布式锁
     *
     * @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);
    }
}
```

### 说明

ZLock仅仅对应一个ZNode路径，也就是说，仅仅代表一把锁。如果需要代表不同的ZNode路径，还需要进行简单改造。这种改造，留给各位自己去实现。

## 10.6.7 Curator的InterProcessMutex可重入锁

Zlock实现的主要价值是展示一下分布式锁的原理和基础开发。在实际的开发中，如果需要使用到分布式锁，不建议去自己“重复造轮子”，而建议直接使用Curator客户端中的各种官方实现的分布式锁，例如其中的InterProcessMutex可重入锁。

这里提供一个InterProcessMutex可重入锁的使用实例，代码如下：

```
package com.crazymakercircle.zk.distributedLock;
//...省略import
/**
 * 测试分布式锁
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
public class ZKLockTester {
    //需要锁来保护的公共资源
    //变量
    int count = 0;
    /**
     * 测试ZK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 InterProcessMutexzkMutex =
            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);
    }
}
```

最后，总结一下ZooKeeper分布式锁：

(1) 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, 使用起来也较为简单。

(2) 缺点: ZooKeeper实现的分布式锁, 性能并不太高。为什么呢? 因为每次在创建锁和释放锁的过程中, 都要动态创建、销毁暂时节点来实现锁功能。大家知道, Zk中创建和删除节点只能通过Leader(主)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(从)服务器上, 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、高并发的应用场景下, 不建议使用ZooKeeper的分布式锁。而由于ZooKeeper的高可用性, 因此在并发量不是太高的应用场景中, 还是推荐使用ZooKeeper的分布式锁。

目前分布式锁, 比较成熟、主流的方案有两种:

(1) 基于Redis的分布式锁。适用于并发量很大、性能要求很高而可靠性问题可以通过其他方案去弥补的场景。

(2) 基于ZooKeeper的分布式锁。适用于高可靠(高可用), 而并发量不是太高的场景。

所以, 这里没有谁好谁坏的问题, 而是谁更合适的问题。

## 10.7 本章小结

在分布式系统中，ZooKeeper是一个重要的协调工具。本章介绍了分布式命名服务、分布式锁的原理以及基于ZooKeeper的参考实现。

本章的那些实战案例，建议大家自己去动手掌握，无论是开始应用实际、还是大公司的面试，都是非常有用的。

## 第11章 分布式缓存Redis

缓存是一个很简单的问题，为什么要用缓存？主要原因是数据库的查询比较耗时，而使用缓存能大大节省数据访问的时间。举个例子，假如表中有2千万个用户信息，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，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。

## 11.1 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。

### 11.1.1 Redis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。Windows版本的下载地址为：<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。

无论在Linux还是在Windows下安装Redis，具体安装过程，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。这部分的安装过程将会以视频的形式，在疯狂创客圈的社群博客，为大家解读安装过程。

查看和修改Redis的配置项，有两种方式：

- (1) 是通过配置文件查看和修改；
- (2) 是通过配置命令查看和修改。

第一种方式，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，以自己的这份来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，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是查看命令，后面加配置项的名称；后一个命令CONFIG SET，是修改命令，后面加配置项的名称和要设置的新值。还要注意的是：Redis的客户端命令是不区分字母大小写的；另外CONFIG GET查看命令可以使用通配符。

举个例子，查看Redis的服务端口，使用config get port，具体如下：

```
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服务的端口为6379。

Redis的配置项比较多，大致的清单如下：

- (1) **port**: 端口配置项，查看和设置Redis监听端口，默认端口为6379。
- (2) **bind**: 主机地址配置项，查看和绑定的主机地址，默认地址的值为127.0.0.1。这个选项，在单网卡的机器上，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，表示时长不限制。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**: 指定保存缓存数据库的本地文件名，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**: 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变得巨大。
- (7) **save**: 指定在多长时间内，有多少次Key-Value更新操作，就将数据同步到本地数据库文件。**save**配置项的格式为**save<seconds><changes>**：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：

```
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900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10000个更改，三者满足一个条件，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，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**: 在主从复制的模式下，设置当前节点为**slave**（从）节点时，设置**master**（主）节点的IP地址及端口，在Redis启动时，它会自动从**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**: 在主从复制的模式下，当**master**（主）服务器节点设置了密码保护时，**slave**（从）服务器连接**master**（主）服务器的密码。

---

**master**服务器节点设置密码的格式为: `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数据库中，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。在默认情况下，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客户端命令将使用数据库3。

Redis存储的形式是Key-Value（键-值对），其中Key（键）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Key相冲突。在不同的数据库中，相同的Key可以分别取到各自的值。

当清除缓存数据时，使用`flushdb`命令，只会清除当前数据库中的数据，而不会影响到其他数据库；而`flushall`命令，则会清除这个Redis实例所有数据库（从0-15）的数据，因此在执行这个命令前要格外小心。

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`

---

举例如下：

---

`redis://testRedis:foobared@119.254.166.136:6379/1`

---

表示连接到第2个Redis缓存库，其中的用户名是可以随意填写的。

## 11.1.2 Redis客户端命令

通过安装目录下的redis-cli命令客户端，可以连接到Redis本地服务。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服务上。

一旦连接上Redis本地服务或者远程服务，即可以通过命令客户端，完成Redis的命令执行，包括了基础Redis的Key-Value缓存操作。

(1) set命令：根据Key，设置Value值。

(2) get命令：根据Key，获取Value值。当Key不存在时，会返回空结果。

set、get两个命令的使用很简单，与Java中Map数据类型的Key-Value设置与获取非常相似。如Key为“foo”、Value为“bar”的设置和获取，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Key。模式支持多种通配符，大致的规则如表11-1所示。

表11-1 Key的匹配模式支持的多种通配符

符号	含义
?	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"
\	转义符。使用\?，可以匹配"??"字符

(4) exists命令：判断一个Key是否存在。如果Key存在，则返回整数类型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：为指定的Key设置过期时间，以秒为单位。

(6) ttl命名：返回指定Key的剩余生存时间（ttl,time to live），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
```

如果没有指定剩余时间，默认的剩余生存时间为-1，表示永久存在。

(7) type命令：返回Key所存储的Value值的类型。最简单的类型为string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），后面会详细介绍。

(8) del命令：删除Key，可以删除一个或多个Key，返回值是删除的Key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) exists命令：检查指定的key是否存在。指定的Key存在，则返回1；Key不存在，则返回0。实例如下：

```
127.0.0.1:6379> exists foo
(integer) 0
```

(10) ping命令：检查客户端是否连接成功，如果连接成功，则返回pong。

### 11.1.3 Redis Key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，Key会有很多的层次间隔，就像一棵目录树一样。例如“疯狂创客圈”的CrazyIM系统中，有缓存用户的Key，也有缓存IM消息的Key。为了以示区分，方便统计、更新、清除，可以将Key的命令组织成一种目录树一样的层次关系。

很多人习惯用英文句号来作为层次关系的Key的分隔符，例如：

---

---

```
superkey.subkey.subsubkey.subsubsubkey....
```

---

---

而使用Redis，建议使用冒号作为superkey和subkey直接的分隔符，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Key，也有缓存IM消息的Key，使用上面的规范，进行命名的规则如下：

- 缓存用户的Key，命名规则为：CrazyIMKey:User:0001
- 缓存消息的Key，命名规则为：CrazyIMKey:ImMessage:0001

#### 提示

最后的部分（如0001），表示的是业务ID。

Key的命名规范使用冒号分割，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，方便快速查到要查阅的Redis Key对应的Value值。

(2) 方便删除与维护。可以对于某一层次下面的Key，使用通配符进行批量查询和批量删除。

## 11.2 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）。

## 11.2.1 String字符串

String类型是Redis中最简单的数据结构。它既可以存储文字（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]

将Key键设置成指定的Value值。如果Key键已经存在，并且保存了一个旧值的话，旧的值会被覆盖，不论旧的类型是否为String都会被忽略掉。如果Key键不存在，那么会在数据库中添加一个Key键，保存的Value值就是刚刚设置的新值。

[EXseconds]选项表示Key键过期的时间，单位为秒。如果不加设置，表示Key键永不过期。另外，SET命令还有一些选项，由于使用较少，这里就展开说明了。

### (2) 批量设值: MSET Key Value[Key Value...]

一次性设置多个Key-Value（键-值对）。相当于同时调用多次SET命令。不过要注意的是，这个操作是原子的。也就是说，所有的Key键都一次性设置的。如果同时运行两个MSET来设置相同的Key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Key-Value（键-值对）。如果任何一个Key键已经存在，那么这个操作都不会执行。所以，当使用MSETNX时，要么全部Key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Key键不存在的时候，才会设置Key键的Value值。

### (4) 获取: GET Key

使用GET命令，可以取得单个Key键所绑定的String值。

### (5) 批量获取: MGET Key[Key...]

在GET命令的前面增加了一个前缀M，表示多个（Multi）。使用MGET命令一次性获取多个Value值，这和多次使用GET命令取得单个值，有什么区别呢？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Key键对应的String的长度，如果Key键对应的不是String，则报错。如果Key键不存在，则返回0。

- (7) 为Key键对应的整数Value值增加1: INCR Key
- (8) 为Key键对应的整数Value值减少1: DECR Key
- (9) 为Key键对应的整数Value值增加increment: INCRBY Key increment
- (10) 为Key键对应的整数Value值减少decrement: DECRBY Key decrement
- (11) 为Key键对应的浮点数Value值增加increment: INCRBYFLOAT Key increment

说明一下: Redis并没有为浮点数Value值减少decrement的操作DECRBYFLOAT。如果要为浮点数Value值减少decrement, 只需要把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的值加上了10.01; 最后将INCRBYFLOAT命令的参数设置成负数, 为foo的值減少了5.0。

## 11.2.2 List列表

Redis的List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列表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列表最多可以包含232-1个元素（最多可超过40亿个元素）。

List列表的典型应用场景：网络社区中最新的发帖列表、简单的消息队列、最新新闻的分页列表、博客的评论列表、排队系统等等。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队列中，后台程序依次从队列中获取任务，处理并将结果返回到结果队列。换句话说，通过List队列，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队列的使用场景，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...]

也叫后推入。将一个或多个的Value值依次推入到列表的尾部（右端）。如果Key键不存在，那么RPUSH之前会先自动创建一个空的List列表。如果Key键的Value值不是一个List类型，则会返回一个错误。如果同时RPUSH多个Value值，则多个Value值会依次从尾部进入List列表。RPUSH命令的返回值为操作完成后List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...]

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List列表的头部（左侧）推入的。

### (3) 左弹出：LPOP Key

PUSH操作是增加元素；而POP操作，则是获取元素并删除。LPOP命令是从List队列的左边（前端），获取并移除一个元素，复杂度O(1)。如果List列表为空，则返回nil。

### (4) 右弹出：RPOP key

与LPOP功能基本相同，是从队列的右边（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:
  - (8) 设置指定索引上的元素:
- 

```
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列表的下标是从0开始的，index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。

### 11.2.3 Hash哈希表

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，Value值可以是文字、整数、浮点数或者二进制数据。在同一个Hash哈希表中，每个Field字段的名称必须是唯一的。这一点和Java中HashMap的Key键的规范要求也是八九不离十的。下面对Hash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Key哈希表中，给Field字段设置Value值。如果Field字段之前没有设置值，那么命令返回1；如果Field字段已经有关联值，那么命令用新值覆盖旧值，并返回0。

（2）获取字段-值： HGET Key Field；

返回Key哈希表中Field字段所关联的Value值。如果Field字段没有关联Value值，那么返回nil。

（3）检查字段是否存在： HEXISTS Key Field；

查看在Key哈希表中，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...]；

删除Key哈希表中，一个或多个指定Field字段，以及那些Field字段所关联的值。不存在的Field字段将被忽略。命令返回被成功删除的Field-Value对的数量。

（5）查看指定的Field字段是否存在： HEXISTS Key Field；

（6）获取所有的Field字段： HKEYS Key；

（7）获取所有的Value值： HVALS Key。

下面使用一个Hash哈希表来缓存系统的IP、端口等配置信息，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
```

总结一下，使用Hash哈希列表的好处：

(1) 将数据集中存放。通过Hash哈希表，可以将一些相关的信息存储在同一个缓存Key键中，不仅方便了数据管理，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Key的命名规范时，可以在命名键的时候，使用冒号分割符来避免命名冲突，但更好的避免冲突的办法是直接使用哈希键来存储“键-值对”数据。

(3) 减少Key键的内存占用。在一般情况下，保存相同数量的“键-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Key键都带有很多的附加管理信息（例如这个Key键的类型、最后一次被访问的时间等），所以缓存的Key键越多，耗费的内存就越多，花在管理数据库Key键上的CPU也会越多。

总之，应该尽量使用Hash哈希表而不是字符串键来缓存Key-Value“键-值对”数据，优势为：方便管理、能够避免键名冲突、并且还能够节约内存。

## 11.2.4 Set集合

Set集合也是一个列表，不过它的特殊之处在于它是可以自动去掉重复元素的。Set集合类型的使用场景是：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。并且Set类型拥有一个命令，它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集合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集合进行集合运算，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 user0002user0003 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"

---

## 11.2.5 Zset有序集合

Zset有序集合和Set集合的使用场景类似，区别是有序集合会根据提供的score参数来进行自动排序。当需要一个不重复的且有序的集合列表，那么就可以选择Zset有序集合类型。常用案例：游戏中的排行榜。

Zset有序集合和Set集合不同的是，有序集合的每个元素，都关联着一个分值（Score），这是一个浮点数格式的关联值。Zset有序集合会按照分值（Score），按从小到大的顺序来排列有序集合中的各个元素。

(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，类似与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，是闭区间范围，而不是开区间范围，即 $\text{min} \leq \text{score} \leq \text{max}$ 内的成员将被返回。另外，可以使用-inf和+inf分别表示有序集合中分数的最小值和最大值。

(7) 获取成员数：ZCARD Key

## (8) 区间计数: ZCOUNT Key min max

在有序集合Key中，计算指定区间分数的成员数。

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
zset
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"
```

## 11.3 Jedis基础编程的实践案例

Jedis是一个高性能的Java客户端，是Redis官方推荐的Java开发工具。要在Java开发中访问Redis缓存服务器，必须对Jedis熟悉才能编写出“漂亮”的代码。Jedis的项目地址：<https://github.com/alphazero/jredis>，大家可以去查看和下载。

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>
```

本实践示例所使用的依赖版本为2.9.0。

Jedis基本的使用十分简单，在每次使用时，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，需要调用Jedis.close()方法把连接关闭，否则会占用系统资源。

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
```

### 11.3.1 Jedis操作String字符串

Jedis的String字符串操作函数和Redis客户端操作String字符串的命令，基本上可以一比一的相互对应。正因为如此，本节不对Jedis的String字符串操作函数进行清单式的说明，只设计了一个比较全面的String字符串操作的示例程序，演示一下这些函数的使用。

Jedis操作String字符串具体的示例程序代码，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 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 no Val");
try {
    Thread.sleep(3000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
// 2 seconds is no Val
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key
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。

### 11.3.2 Jedis操作List列表

Jedis的List列表操作函数和Redis客户端操作List列表的命令，基本上可以一比一的相互对应。也正因为如此，本节也不对Jedis的List列表操作函数做一个清单式的说明，只设计了一个比较全面的List列表操作的示例程序，演示一下这些函数的使用。

具体的示例程序代码如下：

```
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列表尾部添加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列表的长度
        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。

### 11.3.3 Jedis操作Hash哈希表

Jedis的Hash哈希表操作函数和Redis客户端操作Hash哈希表的命令，基本上可以一比一的相互对应。

本节不再罗列Hash哈希表操作函数，仅设计了一个比较全面的Hash哈希表操作的示例程序，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string类型的field字段和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 键-值对（40多亿）
     */
    @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值
        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哈希表操作函数。

### 11.3.4 Jedis操作Set集合

Jedis的Set集合操作函数和Redis客户端操作Set集合的命令，基本上可以一对  
应。

本节不再罗列Jedis操作Set集合的函数，仅设计了一个比较简单的Set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;
//....省略import
public class SetDemo {
    /**
     * Redis 的 Set集合是 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集合操作函数。

### 11.3.5 Jedis操作Zset有序集合

Jedis的Zset有序集合操作函数和Redis客户端操作Zset集合的命令，基本上可以一一对应。

本节不再罗列Jedis操作zset集合的函数，仅设计了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，用于从小到大进行排序。
     * 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。
     * 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);
        //type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有序集合，Zset的Key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有序集合的操作函数。

## 11.4 JedisPool连接池的实践案例

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。为什么呢？大家知道，数据库连接的底层是一条Socket通道，创建和销毁很耗时。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（Connection Pool）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。

### 11.4.1 JedisPool的配置

在使用JedisPool类创建Jedis连接池之前，首先要了解一个很重要的配置类——JedisPoolConfig配置类，它也位于redis.clients.jedis包中。这个连接池的配置类负责配置JedisPool的参数。JedisPoolConfig配置类涉及到很多与连接管理和使用有关的参数，下面将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，如果空闲连接无效，就销毁。销毁连接后，连接数量就少了，如果小于minIdle数量，就新建连接，维护数量不少于minIdle的数量。minIdle确保了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，建议设置为true，确保连接可用；在业务量很大的应用场景，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，建议设置为false（默认值），少一次ping命令的开销。
- (8) **testWhileIdle**: 如果为true，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，如果有效性检测失败，即表示无效连接，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活动之间，要睡眠的毫秒数，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。

有个实际的问题：如何推算一个连接池的最大连接数maxTotal呢？

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 $100/5=200$ 。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，上面的估算仅仅是个理论值。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。

## 提示

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值呢？

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。

## 11.4.2 JedisPool创建和预热

创建JedisPool连接池的一般步骤为创建一个JedisPoolConfig配置实例；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);
            // 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);
            //new JedisPool(config, ADDR, PORT, TIMEOUT);
            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，池子没有Jedis连接资源在使用，在初次访问请求到来的时候，才开始创建新的连接，不过，这样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JredisPoolBuilder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()，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();
    }
}
```

---

### 11.4.3 JedisPool的使用

可以使用前面定义好的getJedis()方法，间接地通过pool.getResource()从连接池获取连接；也可以直接通过pool.getResource()方法获取Jedis连接。

主要的要求是Jedis连接使用完后，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的关闭最终都会被执行到。

```
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() {
        Jedisredis =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 (Jedisredis = 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的效果和使用try-finally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。

## 11.5 使用spring-data-redis完成CRUD的实践案例

无论是Jedis还是JedisPool，都只是完成对Redis操作的极为基础的API，在不依赖任何中间件的开发环境中，可以使用它们。但是，一般的Java开发，都会使用了Spring框架，可以使用spring-data-redis开源库来简化Redis操作的代码逻辑，做到最大程度的业务聚焦。

下面从缓存的应用场景入手，介绍spring-data-redis开源库的使用。

## 11.5.1 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）。

一般来说，在普通的CRUD场景中，大致涉及到的缓存操作为：

### 1. 创建缓存

在创建（Create）一个POJO实例的时候，对POJO实例进行分布式缓存，一般以“缓存前缀+ID”为缓存的Key键，POJO对象为缓存的Value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实例的时候，首先应该根据POJO缓存的Key键，从Redis缓存中返回结果。如果不存在，才去查询数据库，并且能够将数据库的结果缓存起来。

### 3. 更新缓存

在更新（Update）一个POJO实例的时候，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实例的时候，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的缓存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，如果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）、更新操作（U）。
- (2) `getUser`完成查询操作（R）。
- (3) `deleteUser`完成删除操作（D）。

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，接下来就是定义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？

## 11.5.2 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，即配置spring-data-redis库的连接池实例和RedisTemplate模板实例。这是两个spring bean，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。

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
.CacheOperationService">
<property name="redisTemplate" ref="redisTemplate"/>
</bean>
//省略其他的spring-redis.xml配置，具体参见源代码
```

---

spring-data-redis库在JedisPool提供连接池的基础上封装了自己的连接池——RedisConnectionFactory连接工厂；并且spring-data-redis封装了一个短期、非线程安全的连接类，名为RedisConnection连接类。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服务。

在使用spring-data-redis时，虽然没有直接用到Jedis库，但是spring-data-redis库底层对Redis服务的操作还是调用Jedis库完成的。也就是说，spring-data-redis库从一定程度上使大家更好地使用Jedis库。

RedisConnection的API命令操作的对象都是字节级别的Key键和Value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。

### 11.5.3 使用RedisTemplate模板API

RedisTemplate模板类位于核心包org.springframework.data.redis.core中，它封装了五大数据类型的命令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API的名称上并不完全一致，RedisTemplate的API名称更加人性化。例如，Redis客户端命令setNX——Key-Value不存在才设值，非常不直观，但是RedisTemplate的API名称为setIfAbsent，翻译过来就是——如果不存在，则设值。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，为了尽可能地减少第三方库的“入侵”，或者为了在不同的第三方库之间进行方便的切换，一般来说，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基础操作 -----
    /**
     * 取得指定格式的所有的key键
     *
     * @param patens 匹配的表达式
     * @return key 的集合
     */
    public Set 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;
    }
}
/***
 * 根据key 获取过期的时间
 * @param key 键不能为null
 * @return 时间(秒) 返回0代表为永久有效
 */
public long getExpire(String key) {
    return redisTemplate.getExpire(key, TimeUnit.SECONDS);
}

/**
 * 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);
}
/***
 * 设置String
 * @param key 键
 * @param value 值
 * @return true则成功, false则失败
 */
public boolean set(String key, Object value) {
    try {
        redisTemplate.opsForValue().set(key, value);
    }
}

```

```

        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
//...省略其他的String 操作, 请参见源代码
// -----RedisTemplate操作list列表 -----
/***
 * 获取list缓存的内容, start == 0 到 end == -1代表所有值
 * @param key 键
 * @param start开始,从0开始
 * @param end结束
 * @return
 */
public List<Object>lGet(String key, long start, long end) {
    try {
        return redisTemplate.opsForList().range(key, start, end);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
/***
 * 将list放入缓存, 从右边(后端)插入
 * @param key 键
 * @param value 值
 * @return
 */
public booleanlSet(String key, Object value) {
    try {
        redisTemplate.opsForList().rightPush(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
//...省略其他的list操作, 请参见源代码
// -----RedisTemplate操作 Hash列表 -----
/***
 * HashGet, 设置map中的一个field
 * @param key 键不能为null
 * @param field项不能为null
 * @return 值
 */
public Object hget(String key, String field) {
    return redisTemplate.opsForHash().get(key, field);
}
/***
 * HashSet
 * @param key 键
 * @param map 对应多个键值
 * @return true 成功 false 失败
 */
public booleanhmset(String key, Map<String, Object> map) {
    try {
        redisTemplate.opsForHash().putAll(key, map);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
//...省略其他的hash操作, 请参见源代码
// -----RedisTemplate操作 Set集合 -----
/***
 * 根据key获取Set中的所有值
 * @param key 键
 * @return
 */
public Set<Object>sGet(String key) {
    try {

```

```
        return redisTemplate.opsForSet().members(key);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
/***
 * 将数据放入set缓存
 * @param key 键
 * @param values 值可以是多个
 * @return 成功个数
 */
public long sSet(String key, Object... values) {
    try {
        return redisTemplate.opsForSet().add(key, values);
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}
//...省略其他的Set操作, 请参见源代码
}
```

---

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个方法，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。

## 11.5.4 使用RedisTemplate模板API完成CRUD的实践案例

封装完成了自己的CacheOperationService缓存管理服务之后，可以注入到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;
    private static final long CASHE_LONG = 60 * 4;//4分钟
    public void setCacheOperationService
        (CacheOperationService cacheOperationService) {
        this.cacheOperationService = cacheOperationService;
    }
    /**
     * 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.getUid();
        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
        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
              .UserServiceImplWithTemplate">
    <property name="cacheOperationService" ref="cacheOperationService"/>
</bean>
```

---

编写一个用例，测试一下 UserServiceImplWithTemplate，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，缓存机制已经成功生效，数据访问的时间可以从数据库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;
import com.crazymakercircle.im.common.bean.User;
import com.crazymakercircle.util.Logger;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
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);
    }
    //....省略其他的测试用例
}
```

---

## 11.5.5 使用RedisCallback回调完成CRUD的实践案例

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，它负责二进制层面的Redis操作，Key、Value都是二进制字节数组。而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，怎么办呢？可以使用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Redis的操作。

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;
//...
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, CASHE_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, 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
}
}

```

---

同样的，在使用UserServiceInTemplate之前，也需要在配置文件（这里为spring-redis.xml）配置好依赖关系：

---

```

<bean id="serviceImplInTemplate"
    class="com.crazymakercircle.redis.springJedis
        .UserServiceImplInTemplate">
    <property name="redisTemplate" ref="redisTemplate"/>
</bean>

```

---

## 11.6 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，通过合理的添加缓存注解，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。但是，这组注解不仅仅是针对Redis，它本质上并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。通过这组缓存注解，然后加上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。

下面先给大家展示一下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。

## 11.6.1 使用Spring缓存注解完成CRUD的实践案例

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，大致的作用如下：

(1) @CachePut作用是设置缓存。先执行方法，并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键是否在缓存中，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;
import com.crazymakercircle.im.common.bean.User;
import com.crazymakercircle.util.Logger;
import org.springframework.cache.annotation.CacheConfig;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
@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类中没有出现任何的缓存操作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，通过这个实例大家可以发现，使用注解实现缓存和使用API实现缓存的功能相比，前者的代码简化太多了。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。

## 11.6.2 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。而CacheManager接口是Spring定义的一个用来管理Cache缓存的通用接口。对于不同的缓存，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。但是，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，这个是必须的，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" 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，默认是使用proxy。当mode为proxy时，只有当有缓存注解的方法被对象外部的方法调用时，Spring Cache才会发生作用，反过来说，如果一个缓存方法，被其所在对象的内部方法调用时，Spring Cache是不会发生作用的。还有一点，当mode为proxy模式时，只有加在public类型方法上的@Cacheable等注解，才会让Spring Cache发生作用。而mode为aspectj模式时，就不会发生上面的情况，只有注解方法被内部调用，缓存才会生效；非public类型的方法上，也可以使用Spring Cache注解。

<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，默認為false。

大家知道，JDK的动态代理是利用反射机制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。而使用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，@Cacheable和@CacheInvalidate等注解，必须标记在具体类（Concrete Class）类上，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。

但是，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，但是原理是大致相同的，大家可以自行研究。

### 11.6.3 详解@CachePut和@Cacheable注解

简单来说，这两个注解都可以增加缓存，但是有细微的区别：

- (1) @CachePut负责增加缓存。
- (2) @Cacheable负责查询缓存，如果没有查到，则将执行方法，并将方法的结果增加到缓存。

下面我们将来详细介绍一下@CachePut和@Cacheable两个注解。

在支持Spring Cache的环境下，如果@CachePut加在方法上，每次执行方法后，会将结果存入指定缓存的Key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;  
}
```

大家知道，Redis的缓存都是键-值对（Key-Value Pair）。Redis缓存中的Key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Value值就是方法的返回结果，在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缓存的名称

value值表示当前Key键被缓存在哪个Cache上，对应于Spring配置文件中CacheManager缓存管理器的cacheNames属性中配置的某个Cache名称，如userCache。可以配置一个Cache，也可以是多个Cache，当配置多个Cache时，value值是一个数组，如value={userCache, otherCache1, otherCache2...}。

#### 注意

Value属性值中的Cache名称，相当于缓存Key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

Key。

### (2) key属性：指定Redis的Key属性值

key属性，是用来指定Spring缓存方法的Key键，该属性支持SpringEL表达式。当没有指定该属性时，Spring将使用默认策略生成Key键。有关SpringEL表达式，稍候再详细介绍。

### (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，表示将缓存所有的结果。可以通过SpringEL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，代码如下：

```
@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;
}
```

再来看一看@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Key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，才会执行方法，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不同之处的是：@Cacheable只有当Key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Key键在Redis缓存中存在，则直接返回缓存结果。而加了@CachePut注解的方法，则缺少了检查的环节：@CachePut在方法执行前不去进行缓存检查，无论之前是否有缓存，都会将新的执行结果加入到缓存中。

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的三个属性的配置方法也是一样的，这里就不再赘述。

@CachePut和@Cacheable注解也可以标注在类上，表示所有的方法都具缓存处理的功能。但是这种情况，用得比较少。

## 11.6.4 详解@CacheEvict注解

注解@CacheEvict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的名称）；key表示清除哪个Key键；condition表示清除的条件。下面主要看一下两个属性allEntries和beforeInvocation。

(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Key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名称属性所指向的Cache中所有的缓存，这时候，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，这比一个一个清除Key键，效率更高。

在下面的例子中，一次清除Cache名称为userCache中的所有的Redis缓存，代码如下：

```
package com.crazymakercircle.redis.springJedis;
//...
@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，是在对应方法成功执行之后，再触发清除操作。但是，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。

最后说明一下：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。

## 11.6.5 详解@Caching组合注解

@Caching注解，是一个缓存处理的组合注解。通过@Caching，可以一次指定多个Spring Cache注解的组合。@Caching注解拥有三个属性：cacheable、put和evict。

@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key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key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key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。如果外键都进行了缓存，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：用户信息、地址信息、用户的商品、等等。

使用@Caching注解，为各个方法的加上一大票缓存注解，具体如下：

```
/*
 * 一个方法上，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。注解中需要用到SpEL表达式，将在下一节为大家专门介绍一下SpringEL。

## 11.7 详解SpringEL（SpEL）

Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”。SpEL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，表达式的值可以动态装配到Spring Bean属性或构造函数中，表达式可以调用Java静态方法，可以访问Properties文件中的配置值等等，SpringEL能与Spring功能完美整合，给静态Java语言增加了动态功能。

大家知道，JSP页面的表达式使用\${}进行声明。而SpringEL表达式使用#{}}进行声明。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、三目运算及Elvis表达式、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、自定义函数等等。
- (3) 集合相关表达式：内联列表、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；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。

## 11.7.1 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。基础的运算符主要包括：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、类型运算符、变量引用符等。

(1) 算术运算符：SpEL提供了以下算术运算符：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：SpEL提供了以下关系运算符：等于（==）、不等于（!=）、大于（>）、大于等于（>=）、小于（<）、小于等于（<=），区间（between）运算等等。例如：#{2>3}值为false。

(3) 逻辑运算符：SpEL提供了以下逻辑运算符：与（and）、或（or）、非（!或NOT）。例如：#{2>3 or 4>3}值为true。与Java逻辑运算不同，SpEL不支持“&&”和“||”。

(4) 字符串运算符：SpEL提供了以下字符串运算符：连接（+）和截取（[ ]）。例如：#{'Hello'+ 'World!'}的结果为“Hello World!”。#{'Hello World!' [0]}截取第一个字符“H”，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：“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)”，“Type”表示某个Java类型，实际上对应于Java类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“#”，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EvaluationContext接口的setVariable(variableName,value)方法，可以定义“上下文变量”，这些变量在表达式中

采用“#variableName”的方式予以引用。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，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;
import com.crazymakercircle.util.Logger;
import lombok.Data;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpELExpressionParser;
import org.springframework.expression.spel.support
    .StandardEvaluationContext;
import org.springframework.stereotype.Component;
/**SpEL运算符的使用
 * create by 尼恩 @ 疯狂创客圈
 */
@Component
@Data
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:=", root);
    }
}
```

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();
}
```

## 注意

一般来说，SpringEL表达式使用#{ }进行声明。但是，不是所有注解中的SpringEL表达式都需要#{ }进行声明。例如，@Value注解中的SpringEL表达式需要#{ }进行声明；而ExpressionParser.parseExpression实例方法中的SpringEL表达式不需要#{ }进行声明；另外，@CachePut和@Cacheable等缓存注解中key属性值的SpringEL表达式，也不需要#{ }进行声明。

## 11.7.2 缓存注解中的SpringEL表达式

对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，这个类继承于基础的方法注解上下文MethodBasedEvaluationContext，而这个方法则继承于StandardEvaluationContext（大家熟悉的标准注解上下文）。

CacheEvaluationContext的构造器如下：

```
class CacheEvaluationContext extends MethodBasedEvaluationContext {  
    //构造器  
    CacheEvaluationContext(Object rootObject, //根对象  
                           Method method, //当前方法  
                           Object[] arguments, //当前方法的参数  
                           ParameterNameDiscoverer parameterNameDiscoverer)  
    {  
        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，可以获取到如表11-2所示的属性。

表11-2 通过CacheEvaluationContext的rootObject根对象能获取的属性

属性名称	说明	示例
methodName	当前被调用的方法名	获取当前被调用的方法名: #root.methodName
Method	当前被调用的方法	获取当前被调用的方法: #root.method.name
Target	当前被调用的目标对象	当前被调用的目标对象: #root.target
targetClass	当前被调用的目标对象类	当前被调用的目标对象类型: #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	当前方法调用使用的缓存之列表， 如：@Cacheable(value={"cache1", "cache2"})，则有两个 cache	当前被调用方法的第 0 个 cache 名称: #root.caches[0].name

在配置key属性时，如果用到SpEL表达式root对象的属性，也可以将“#root”省略，因为Spring默认使用的就是root对象的属性。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) 方式一：使用“#p参数index”形式访问方法的参数

展示两个使用“#p参数index”形式访问arguments参数的示例程序：

```
//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，这里是参数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) 方式二：使用“#参数名”形式访问方法的参数

可以使用“#参数名”的形式直接访问方法的参数。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”的形式访问第0个参数的属性值，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，使用方式二“#参数名”的形式，比方式一“#p参数index”的形式，更加的直接和直观。

## 11.8 本章小结

本章介绍了Redis的安装、客户端的使用、Jedis编程。对于如何使用spring-data-redis操作缓存进行了详细的介绍。同时介绍了如何使用Spring的缓存注解，节省编程使用缓存的编码工作量。

本章重点内容是缓存操作的一个全面的知识基础，对大家的实际开发，是有非常大的指导作用的，尤其在目前的市面上，还没有一本书具体的介绍spring-data-redis和缓存注解的使用。然而，由于篇幅的原因，本章的内容没有涉及到Redis的高端架构和开发。包括如何搭建高可用、高性能的Redis集群，以及如何在Java中操作高可用、高性能的Redis集群等等。后续“疯狂创客圈”将结合本书，提供一些更加高端的教学视频，将这方面的内容呈现给大家，尽可能为大家的开发、面试等尽一份绵薄之力。

## 第12章 亿级高并发IM架构的开发实践

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通信的高并发应用方案。并从学习和实战的角度出发，将联合“疯狂创客圈”社群的高性能发烧友们，一起持续迭代出一个支持亿级流量的IM项目，暂时命名为“CrazyIM”。

## 12.1 如何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、SpringCloud WEB服务集群、RocketMQ消息队列集群等等，具体如图12-1所示。

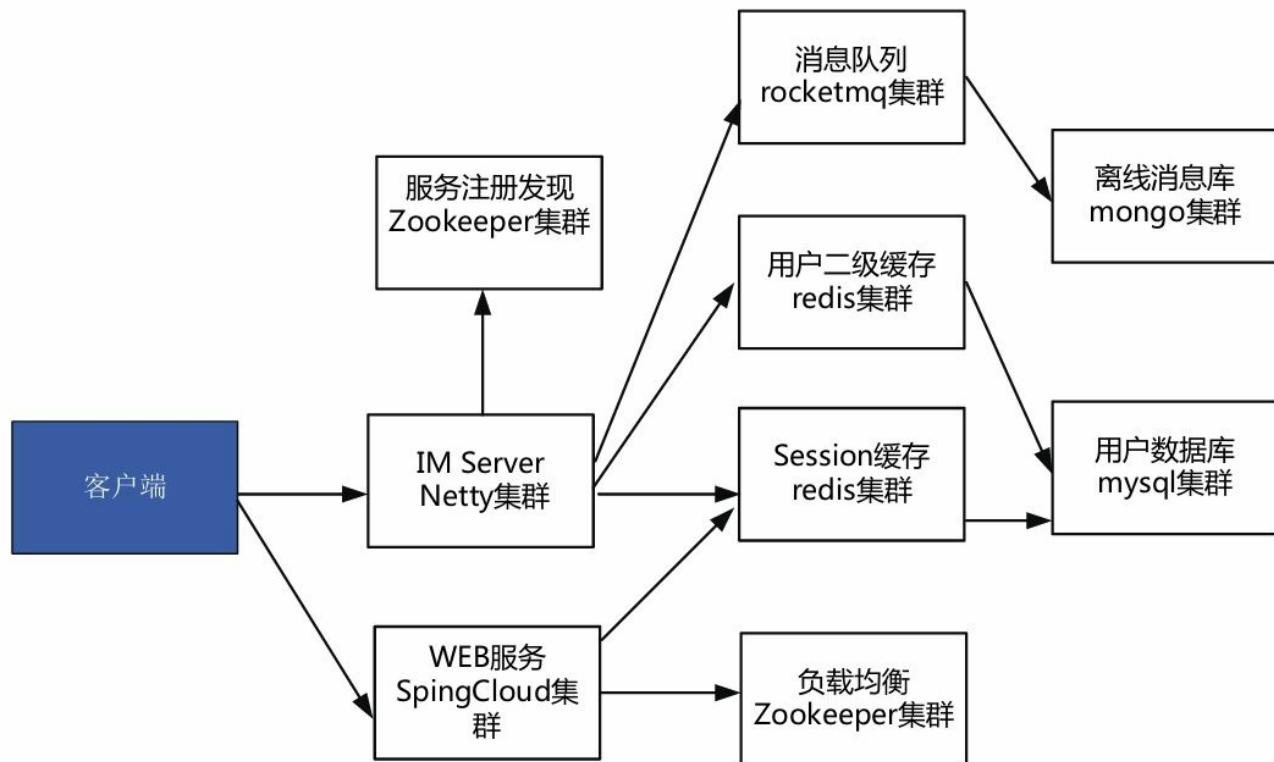


图12-1 支撑亿级流量的高并发IM架构

## 12.1.1 亿级流量的系统架构的开发实践

支撑亿级流量的高并发IM通信的几大集群中，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SpringCloud WEB服务集群、 MySql集群，完成海量用户的登录和存储，以及离线消息的存储。最后是RocketMQ消息队列集群，用于离线消息的保存。

主要的集群介绍如下：

### (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。主要在基于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的模式。例如，可以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，并且只是涉及核心功能，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，还有很多的完善的空间，“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，进行不断的更新和迭代，所以最终的架构图和实现，以最后的版本为准。

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，支撑亿级流量是没有任何问题的。为什么这么说呢？

单体的Netty服务器，远远不止支持10万个并发，在CPU、内存还不错的情况下，如果配置得当，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100万高并发，请查询疯狂创客圈社群的文章《Netty 100万级高并发服务器配置》。

## 12.1.2 高并发架构的技术选型

明确了架构之后，接下来就是平台的技术选型，大致如下：

(1) 核心

(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。

(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高速消队列。

(5) 数据库：mysql+mongodb

mysql用来存储结构化数据，如用户数据。mongodb很重要，用来存储非结构化的离线消息。

(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。

### 12.1.3 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，结构化数据到二进制数据的序列化和反序列化。序列化协议的类型：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。但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，但是劣势也非常的突出。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，很显然，这点不利于大家开发和调试。

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，建议使用文本协议，例如JSON；
- 对于并发度非常之高，QPS在千万级、亿级的通信系统，尽量选择二进制的协议。

“疯狂创客圈”社群持续迭代的“CrazyIM”项目，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。

#### 12.1.4 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。

大家知道，TCP协议的连接过程是比较烦琐的，建立连接是需要三次握手的，而释放则需要4次握手，所以说每个连接的建立都需要消耗资源和时间。

在高并发的IM系统中，客户端和服务器之间，需要大量的发送通信的消息，如果每次发送消息，都去建立连接，客户端的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，肯定是需要长连接。

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握手断开连接。

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等一些低频的请求，这些都使用短连接来实现。

综上所述，在这个高并发IM系统中，存在两类的服务器。一类短连接服务器和一个长连接服务器。

短连接服务器也叫Web服务器，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即时通信服务器，主要作用就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的转发。并且，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、自动重连等方面的工作。

短连接Web服务器和长连接IM服务器之间，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。

如果用户规模庞大，无论是短连接Web服务器，还是长连接IM服务器，都需要

进行横向的扩展，都需要扩展到上十台、百台、甚至上千台服务器。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，首先是代理大量的Web服务器，从而无感知地实现短连接的高并发。在客户端登录时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，目前非常成熟，可以使用SpringCloud或者Dubbo等分布式Web技术，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。

## 12.2 分布式IM的命名服务的实践案例

前面提到，一个高并发系统是由很多的节点所组成，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。

## 12.2.1 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、Netty服务端口，以及Netty的服务连接数。具体如下：

```
package com.crazymakercircle.imServer.distributed;
import lombok.Data;
import java.util.Objects;
import java.util.concurrent.atomic.AtomicInteger;
/**
 * IM节点的POJO类
 * create by 尼恩 @ 疯狂创客圈
 */
@Data
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 String toString() {
        return "ImNode{" +
            "id='" + id + '\'' +
            "host='" + host + '\'' +
            ", port='" + port + '\'' +
            ", balance=" + balance +
            '}';
    }
    @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。而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。

## 12.2.2 IM节点的ImWorker类

节点的命名服务的思路是，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，取得属于自己的那个后缀的编号。主要的代码如下：

```
package com.crazymakercircle.imServer.distributed;
import com.crazymakercircle.imServer.server.ServerConstants;
import com.crazymakercircle.imServer.zk.ZKclient;
import com.crazymakercircle.util.JsonUtil;
import org.apache.curator.framework.CuratorFramework;
import org.apache.ZooKeeper.CreateMode;
import org.apache.ZooKeeper.data.Stat;
<*/
 * IM 节点的Zk协调客户端
 * create by 尼恩 @ 疯狂创客圈
 */
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 ImWorkersingleInstance = null;
    //取得唯一的实例
    public static ImWorkergetInst() {
        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：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：

(1) MANAGE\_PATH

(2) pathPrefix

(3) 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，否则的话，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"/"分割符。也可是在工作路径的后面，加上"/"分割符和其他的前缀字符，如"/im/nodes/id-"、"/im/nodes/seq-"等等。

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());
```

## 12.3 Worker集群的负载均衡之实践案例

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，负载均衡就是需要将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，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。而每台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，去“/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，并且返回给客户端。

### 12.3.1 ImLoadBalance负载均衡器

短连接网关WebGate如何获得最佳的Netty服务器呢？需要通过查询ZooKeeper集群来实现。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，大致如下：

```
package com.crazymakercircle.Balance;
import com.crazymakercircle.util.JsonUtil;
import com.crazymakercircle.util.ImNode;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
/**
 * create by 尼恩 @ 疯狂创客圈
 */
@Data
@Slf4j
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 ImNode balance(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。而在这个方法中，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；二个是通过负载信息，计算最小负载的服务器。

代码中的getWorkers()方法，调用了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。

取到了工作节点的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。

### 12.3.2 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;
import com.crazymakercircle.Balance.ImLoadBalance;
import com.crazymakercircle.controller.utility.BaseController;
import com.crazymakercircle.mybatis.entity.LoginBack;
import com.crazymakercircle.mybatis.entity.User;
import com.crazymakercircle.service.UserService;
import com.crazymakercircle.util.ImNode;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiOperation;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import javax.annotation.Resource;
/**
 * WEB GATE
 * Created by 尼恩 at 疯狂创客圈
 */
@EnableAutoConfiguration
@RestController
@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
         */
        ImNode bestWorker = ImLoadBalance.INSTANCE.getBestWorker();
        back.setImNode(bestWorker);
        back.setUser(loginUser);
        back.setToken(loginUser.getUserId().toString());
        String r = super.getJsonResult(back);
        return r;
    }
    //...省略其他的方法
}
```

## 12.4 即时通信消息的路由和转发的实践案例

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，那么就需要在不同的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，找到用户的连接所在的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。

## 12.4.1 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，节选如下：

```
package com.crazymakercircle.imServer.distributed;
import com.crazymakercircle.imServer.server.ServerConstants;
import com.crazymakercircle.imServer.zk.ZKclient;
import com.crazymakercircle.util.JsonUtil;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.cache.ChildData;
import org.apache.curator.framework.recipes.cache.TreeCache;
import org.apache.curator.framework.recipes.cache.TreeCacheEvent;
import org.apache.curator.framework.recipes.cache.TreeCacheListener;
import java.util.concurrent.ConcurrentHashMap;
/**
 * IM路由器WorkerRouter
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
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, WorkerReSender> workerMap =
        new ConcurrentHashMap<>();
    public static WorkerRouter getInst() {
        if (null == singleInstance) {
            singleInstance = new WorkerRouter();
            singleInstance.client = ZKclient.instance.getClient();
            singleInstance.init();
        }
        return singleInstance;
    }
    private WorkerRouter() {
    }
    //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
}

```

---

在上一小节中，我们已经知道，一个节点上线时，首先要通过命名服务加入到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) {
    log.info("[TreeCache] 节点更新端口, path={}, data={}, data.getPath(),
             data.getData());
    byte[] payload = data.getData();
    String path = data.getPath();
    ImNode imNode = JsonUtil.JsonBytes2Object(payload, ImNode.class);
    long id = getId(path);
    imNode.setId(id);
    WorkerReSender reSender = workerMap.get(imNode.getId());
    //重复收到注册的事件
    if (null != reSender && reSender.getRemoteNode().equals(imNode))
    {
        return;
    }
    //创建一个消息转发器
    reSender = new WorkerReSender(imNode);
    //建立转发的连接
    reSender.doConnect();
    workerMap.put(id, reSender);
}

```

---

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路径中有新节点的ID，ZNode的payload负载中有新节点的Netty服务的IP地址和口号，这三个信息共同构成新节点的POJO信息——ImNode节点信息。WorkerRouter在检查完和确定本地不存在该节点的转发器后，添加一个转发器WorkerReSender，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WorkerReSender，而添加的远程Netty节点的POJO信息被封装在转发器中。

## 12.4.2 IM转发器WorkerReSender

IM转发器WorkerReSender封装了远程节点的IP地址、端口号以及ID。另外，WorkerReSender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 Channel通道，通过这个通道将消息转发给远程的节点。

IM转发器WorkerReSender的核心代码如下：

```
/*
 * IM转发器
 * create by 尼恩 @ 疯狂创客圈
 */
@Slf4j
@Data
public class WorkerReSender {
    //连接远程节点的Netty通道
    private Channel channel;
    //连接远程节点的POJO信息
    private ImNoderemoteNode;
    /**
     * 连接标记
     */
    private boolean connectFlag = false;
    private Bootstrap b;
    private EventLoopGroup g;
    public WorkerReSender(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();
        //服务器端口
        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());
            }
        } catch (Exception e) {
            log.error("连接分布式节点失败", e);
        }
    }
}
```

```
        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相关的代码，比较简单。严格来说，IM转发器是一个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 channel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);
}
```

---

## 12.5 Feign短连接RESTful调用

一般来说，短连接的服务接口都是基于应用层HTTP协议的HTTP API或者RESTful API实现的，通过JSON文本格式返回数据。如何在Java服务器端调用其他节点的HTTP API或者RESTful API呢？

至少有以下几种方式：

- JDK原生的URLConnection
- Apache的HttpClient/HttpComponents
- Netty的异步HttpClient
- Spring的RestTemplate

目前用的最多的，基本上是第二种，这也是在单体服务时代最为成熟和稳定的方式，也是效率较高的短连接方式。

首先作一个解释，什么是RESTful API。REST的全称是Representational State Transfer（表征状态转移，也有译成表述性状态转移），它是一种API接口的风格而不是标准，只是提供了一组调用的原则和约束条件。也就是说，在短连接服务的领域，它算是一种特殊格式的HTTP API。

言归正传，如果同一个HTTP API/RESTful API接口，倘若不止一个短连接服务器提供服务，而是有多个节点提供服务，那么简单使用HttpClient调用就无能为力了。

HttpClient/HttpComponents调用不能根据接口的负载或者其他条件，去判断哪一个接口应该调用，哪一个接口不应该调用。解决这个问题的方式如何呢？

可以使用Feign来调用多个服务器的同一个接口。Feign不仅可以进行同接口多服务器的负载均衡，一旦使用了Feign作为HTTP API的客户端，调用远程的HTTP接口就会变得像调用本地方法一样简单。

Feign是Netflix开发的一个声明式、模板化的HTTP客户端，Feign的目标是帮助Java工程师更快捷、优雅地调用HTTP API//RESTful API。Netflix Feign目前改名为OpenFeign，最新版本是2018.5发布的9.7.0。Feign在Java应用中负责处理与远程Web服务的请求响应，最大限度地降低了编码的复杂性。另外，Feign被无缝集成到了SpringCloud微服务框架，使用Feign后，可以非常方便地在项目中使用SpringCloud微服务技术。如果项目使用了SpringCloud技术，同样可以更方便地使用Feign。即便项目中没有使用SpringCloud，使用Feign也非常简单。总之，它是Java应用中调用

Web服务的客户端利器。

下面就看看在单独使用Feign的应用场景下是怎么调用远程的HTTP服务。

引入Feign依赖的jar包到pom.xml:

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-core</artifactId>
    <version>9.7.0</version>
</dependency>
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-gson</artifactId>
    <version>9.7.0</version>
</dependency>
```

接下来就可以开始使用Feign来调用远程的HTTP API了。

## 12.5.1 短连接API的接口准备

前面讲到，在高并发的IM系统中，用户的登录与认证、好友的更新与获取等等一些低频的请求都使用短连接来实现。

作为演示，这里仅仅列举两个短连接的API接口：

(1) `http://localhost:8080/user/{userid}`

这个接口的功能是获取用户信息，是一个典型的RESTful类型的接口。`{userid}`是一个占位符，在调用的时候需要替换成用户id。例如，如果用户id为1，Feign最终生成的实际调用的链接为：`http://localhost:8080/user/1`。

(2) `http://localhost:8080/login/{username}/{password}`

这个接口实现的功能是用户登录的认证。这里有两个占位符，占位符`{username}`表示用户名，占位符`{password}`表示用户密码。例如，如果用户名称为zhangsan，密码为123，那么Feign最终生成的实际调用的链接为：`http://localhost:8080/login/zhangsan/123`。

上面的接口很简单，仅仅是为了演示，不能用于生产场景。这些API的开发和实现可以使用Spring MVC/JSP Servlet等常见的WEB技术。

## 12.5.2 声明远程接口的本地代理

如何通过Feign技术来调用上面的这些HTTP API呢？

第一步，需要创建一个本地的API的代理接口。具体如下：

```
package com.crazymakercircle.imServer.feignClient;
import feign.Param;
import feign.RequestLine;
/**
 * 远程接口的本地代理
 * Created by 尼恩 at 疯狂创客圈
 */
public interface UserAction {
    /**
     * 登录代理
     * @param username 用户名
     * @param password 密码
     * @return 登录结果
     */
    @RequestLine("GET /login/{username}/{password}")
    public String loginAction(
        @Param("username") String username,
        @Param("password") String password);
    /**
     * 获取用户信息代理
     * @param userid 用户id
     * @return 用户信息
     */
    @RequestLine("GET /{userid}")
    public String getById(
        @Param("userid") Integer userid);
}
```

在代理接口中，为每一个远程HTTP API定义一个本地代理方法。

如何将方法对应到远程接口呢？在方法的前面加上一个@RequestLine注解，注明远程HTTP API的请求地址。这个地址不需要从域名和端口开始，只需要从URI的根目录“/”开始即可。例如，如果远程HTTP API的URL为：

<http://localhost:8080/user/{userid}>，@RequestLine声明的值只需要配成/user/{userid}即可。

如何给接口传递参数值呢？在方法的参数前面加上一个@Param注解即可。  
@Param内容为HTTP链接中参数占位符的名称。绑定好之后，实际这个Java接口中的参数值会替换到@Param注解中的占位符。例如，由于在getById的唯一参数userid的@Param注解中用到的占位符是userid，那么通过调用userAction.getById(100)，即userid的值为100，就会用来替换掉请求链接<http://localhost:8080/user/{userid}>中的占位符userid，最终得到的请求链接为：<http://localhost:8080/user/100>。

### 12.5.3 远程API的本地调用

在完成远程API的本地代理接口的定义之后，接下来的工作就是调用本地代理，这个工作也是非常简单的。

还是以疯狂创客圈的“CrazyIM”实战项目中获取用户信息和用户登录这两个API的代理接口的调用为例。实践案例的代码如下：

```
import com.crazymakercircle.imServer.feignClient.UserAction;
import feign.Feign;
import feign.codec.StringDecoder;
import org.junit.Test;
/**
 * 远程API的本地调用
 * Created by 尼恩 at 疯狂创客圈
 */
public class LoginActionTest {
    /**
     * 测试登录
     */
    @Test
    public void testLogin() {
        UserAction action = Feign.builder()
            // .decoder(new GsonDecoder())
            .decoder(new StringDecoder())
            .target(UserAction.class, "http://localhost:8080/user");
        String s = action.loginAction("zhangsan", "zhangsan");
        System.out.println("s = " + s);
    }
    /**
     * 测试获取用户信息
     */
    @Test
    public void test GetById() {
        UserAction action = Feign.builder()
            // .decoder(new GsonDecoder())
            .decoder(new StringDecoder())
            .target(UserAction.class, "http://localhost:8080/user");
        String s = action.getById(2);
        System.out.println("s = " + s);
    }
}
```

最为核心的就一步，构建一个远程代理接口的本地实例。调用Feign.builder()构造器模式的方法，带上一票配置方法的链式调用。主要的链式调用的配置方法介绍如下：

#### (1) options配置方法

options方法指定连接超时的时长以及响应超时的时长。

#### (2) retryer配置方法

retryer方法主要是指定重试策略。

### (3) decoder配置方法

decoder方法指定对象解码方式，这里用的是基于String字符串的解码方式。如果需要使用Jackson的解码方式，需要在pom.xml中添加Jackson的依赖。

### (4) client配置方法

此方法用于配置底层的请求客户端。Feign默认使用Java的HttpURLConnection作为HTTP请求客户端。Feign也可以直接使用现有的公共第三方HTTP客户端类库，如Apache HttpClient，OKHttp，来编写Java客户端以访问HTTP服务。

集成Apache HttpComponentsHttpClient的例子为：

```
Feign.builder().client(new ApacheHttpClient())
```

集成OKHttp的例子为：

```
Feign.builder().client(new OkHttpClient()).target(...)
```

集成Ribbon的例子为：

```
Feign.builder().client(RibbonClient.create()).target(...)
```

Feign集成了Ribbon后，利用Ribbon维护了API服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，Feign只需要定义服务绑定接口且以声明的方法，可以优雅而简单地实现服务调用。

### (5) target方法

这是构造器模式最后面的方法，通过它可以最终得到本地代理实例。它有两个参数，第一个是本地的代理接口的class类型，第二个是远程URL的根目录地址。第一个代理接口类很重要，最终Feign.builder()构造器返回的本地代理实例类型就这个接口的类型。代理接口类中每一个接口方法前用@RequestLine声明的URI链接，最终都会加上target方法的第二个参数的根目录值，来形成最终的URL。

target方法是最后面的一个方法，也就是说它的后面不能再链接调用其他的配置方法。

主要的构造器方法就介绍这些，具体使用的细节以及其他的方法，请参见官网的说明文档。

使用配置方法完成配置之后，再通过Feign.builder()构造完成代理实例，调用远程API，这就和调用Java函数一样简单了。

总之，如果是独立调用HTTP服务，那么尽量使用Feign。原因是一是简单；二是如果采用HttpClient或其他相对较“重”的框架，对初学者来说编码量与学习曲线都会是一个挑战；三是既可以独立使用Feign，又方便后续Spring Could微服务框架的继承。使用Feign代替HttpClient等其他方式，何乐而不为呢？

## 12.6 分布式的在线用户统计的实践案例

顾名思义，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。

## 12.6.1 Curator的分布式计数器

Curator有两个计数器，一个是用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;
import com.crazymakercircle.imServer.server.ServerConstants;
import com.crazymakercircle.imServer.zk.ZKclient;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.atomic.AtomicValue;
import org.apache.curator.framework.recipes.atomic.DistributedAtomicLong;
import org.apache.curator.retry.RetryNTimes;
/**
 * 分布式计数器
 * create by 尼恩 @ 疯狂创客圈
 */
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;
    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，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();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

```
        System.out.println("old cnt: " + val.preValue()
                           + "    new cnt : " + val.postValue()
                           + "    result:" + val.succeeded());
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}
=====
```

## 12.6.2 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：

```
/**  
 * 增加本地session  
 */  
public void addSession(String sessionId, SessionLocal s) {  
    localMap.put(sessionId, s);  
    String uid = s.getUser().getUid();  
    //增加用户数  
    OnlineCounter.getInst().increment();  
    //...  
}
```

当用户下线的时候，调用decrease方法分布式地减少一次计数：

```
/**  
 * 删除本地session  
 */  
public void removeLocalSession(String sessionId) {  
    if (!localMap.containsKey(sessionId)) {  
        return;  
    }  
    localMap.remove(sessionId);  
    //减少用户数  
    OnlineCounter.getInst().decrement();  
    //...  
}
```

## 12.7 本章小结

本章介绍了支撑亿级流量的高并发IM架构以及高并发架构下的技术选型。然后，集中介绍了Netty集群所涉及的分布式IM的命名服务、Worker集群的负载均衡、即时通信消息的路由和转发、分布式的在线用户统计等技术实现。

本章的实例代码来自于“疯狂创客圈”社群的高并发学习项目“CrazyIM”，由于项目在不断地迭代，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。

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。