# 3年JAVA技能总结-网络通信篇

Socket  
Http  
WebService  
阻塞IO、非阻塞IO、多路复用IO、异步IO这四种IO模型

JDK 1.4后，Java提供了一个全新的IO API，即 Java New IO

NIO主要有三大核心部分：

Channel(通道):

Buffer(缓冲区):

Selector(选择器):用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道

传统IO基于字节流和字符流进行操作，而NIO基于Channel和Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中

NIO和传统IO（一下简称IO）之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的

Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream.而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作。

NIO中的Channel的主要实现有：

FileChannel

DatagramChannel

SocketChannel

ServerSocketChannel

package nio1;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public class NIO1 {

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

// 1. 获取数据源 和 目标传输地的输入输出流（此处以数据源 = 文件为例）

String infile = "";

String outfile = "";

FileInputStream fin = new FileInputStream(infile);

FileOutputStream fout = new FileOutputStream(outfile);

// 2. 获取数据源的输入输出通道

FileChannel fcin = fin.getChannel();

FileChannel fcout = fout.getChannel();

// 3. 创建 缓冲区 对象：Buffer（共有2种方法）

// 方法1：使用allocate()静态方法

ByteBuffer buff = ByteBuffer.allocate(256);

// 上述方法创建1个容量为256字节的ByteBuffer

// 注：若发现创建的缓冲区容量太小，则重新创建一个大小合适的缓冲区

// 方法2：通过包装一个已有的数组来创建

// 注：通过包装的方法创建的缓冲区保留了被包装数组内保存的数据

//ByteBuffer buff = ByteBuffer.wrap(byteArray);

// 额外：若需将1个字符串存入ByteBuffer，则如下

String sendString="你好,服务器. ";

ByteBuffer sendBuff = ByteBuffer.wrap(sendString.getBytes("UTF-16"));

// 4. 从通道读取数据 & 写入到缓冲区

// 注：若 以读取到该通道数据的末尾，则返回-1

fcin.read(buff);

// 5. 传出数据准备：将缓存区的写模式 转换->> 读模式

buff.flip();

// 6. 从 Buffer 中读取数据 & 传出数据到通道

fcout.write(buff);

// 7. 重置缓冲区

// 目的：重用现在的缓冲区,即 不必为了每次读写都创建新的缓冲区，在再次读取之前要重置缓冲区

// 注：不会改变缓冲区的数据，只是重置缓冲区的主要索引值

buff.clear();

}

}

NIO入门

public class NIO2 {

private static int port = 8081;

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

//创建selector对象

Selector selector = Selector.open();

// 2. 向Selector对象绑定通道

// a. 创建可选择通道，并配置为非阻塞模式

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.configureBlocking(false);

// b. 绑定通道到指定端口

ServerSocket serverSocket = serverSocketChannel.socket();

serverSocket.bind(new InetSocketAddress(port));

// c. 向Selector中注册感兴趣的事件

serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);

while (true) {

// 该调用会阻塞，直到至少有一个事件就绪、准备发生

int select = selector.select();

// 一旦上述方法返回，线程就可以处理这些事件

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

Iterator<SelectionKey> iterator = selectionKeys.iterator();

while (iterator.hasNext()) {

SelectionKey key = iterator.next();

iterator.remove();

// process(key);

}

}

}

}

nio复制文件  
public class Test {

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

// 设置输入源 & 输出地 = 文件

String infile = "C:\\copy.sql";

String outfile = "C:\\copy.txt";

// 1. 获取数据源 和 目标传输地的输入输出流（此处以数据源 = 文件为例）

FileInputStream fin = new FileInputStream(infile);

FileOutputStream fout = new FileOutputStream(outfile);

// 2. 获取数据源的输入输出通道

FileChannel fcin = fin.getChannel();

FileChannel fcout = fout.getChannel();

// 3. 创建缓冲区对象

ByteBuffer buff = ByteBuffer.allocate(1024);

while (true) {

// 4. 从通道读取数据 & 写入到缓冲区

// 注：若 以读取到该通道数据的末尾，则返回-1

int r = fcin.read(buff);

if (r == -1) {

break;

}

// 5. 传出数据准备：调用flip()方法

buff.flip();

// 6. 从 Buffer 中读取数据 & 传出数据到通道

fcout.write(buff);

// 7. 重置缓冲区

buff.clear();

}

}

}