# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/037\_NIO核心原理剖析之如何通过Channel三次握手建立连接/笔记.doc**

每个channel在什么情况下会被selector感知到呢？

你在注册的时候设置了每个channel感兴趣的操作，SelectionKey来设置的

所以selector如果感知到某个channel可以进行你感兴趣的那个操作的话，此时就会把那个channel对应的key返回给你

跟一个客户端TCP三次握手之后，建立好TCP连接之后，就会有一个Socket，对应的就有一个SocketChannel，他底层肯定也是封装了操作系统级别的一个Socket，他就代表了TCP连接的两个端

默认是阻塞模式的

如果你当前设置的ServerSocketCHannel是非阻塞模式，那么如果调用accept()方法的时候发现没有人来发起连接，此时就会返回一个null；但是如果是阻塞模式，他就会在accept()方法这里卡住，阻塞住，一直等待别人发送一个连接请求过来

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/036\_NIO核心原理剖析之操作系统select反向通知机制/笔记.doc**

linux select机制，他可以让你一个线程可以监听多个socket是否有请求发送进来

从之前我们看过的源码已经知道了，每个channel都对应一个SelectionKey，大致可以表明的是说，我们对某一个channel感兴趣的操作，还有的话呢，通过这个key可以唯一反向获取到对应的channel

selector.select()机制，猜测一下都知道了

底层一定是基于操作系统的类库来实现对多个socket的一个监听，如果某个socket有对应的请求进来了，此时他对应的channel就可以执行某个操作了，每个channel是对应着一个key的

select()方法，他会感知到哪些channel底层的socket有请求进来了，可以执行IO操作来读取数据或者发送数据出去了，他就会把那些channel对应的key的集合，给你返回回来了，他会干这么一件事情

这个方法他是阻塞的，如果没感知到哪个channel对应的socket可以执行读写操作的话，此时他就会卡着，阻塞着，一直等待某个channel可以写数据，或者是可以读数据了

一些底层原理的分析，我觉得是可以帮助大家更好的理解NIO网络编程的底层的机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/025\_再次改造Socket网络通信程序支持长连接模式/笔记.doc**

什么是TCP长连接，什么是TCP短连接

每次建立一个连接，发送一个请求，获取一个响应，断开连接，这个就是所谓的短连接

长连接，每次建立一个连接，可以发送很多的请求，一直持续维持住这个连接，不断开，持续的通过这个链接跟服务端来通信，不停的发送数据和请求，服务端也长期维持了这个连接，不停的接受请求，返回响应

这个就是所谓的长连接，连接存在的时间很长的

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/030\_如何限制客户端的连接数来避免服务器负载过高/笔记.doc**

在bind方法中可以加一个参数

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/040\_NIO核心原理剖析之服务端与客户端程序底层梳理/笔记.doc**

再次来梳理一下客户端到服务端的整个流程

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/041\_课程预告：分布式存储系统项目实战NIO技术/笔记.doc**

一个大的项目，分布式存储系统，他整个我们会仿照hadoop hdfs的架构和原理来编写，hadoop源码里完全就没用netty，就是用原生的nio API自己构建了一套复杂的分布式存储系统，对文件的读写，对网络的通信，数据的传输，全部是基于NIO来写的

我们会完全仿照这个世界上最复杂额工业级的分布式存储系统来做，里面会大量的实战NIO技术，并发技术，初步的学习一下gRPC，做分布式系统的远程RPC调用的，用用比较简单，我们可以来学习一下

包含各种复杂架构和技术原理的一套分布式存储系统

NIO、并发、RPC，相关的技术，全都打磨的非常的熟练和精通了，完全你可以掌握如何基于底层技术来构建复杂的分布式系统，有了这套技术功底之后，后面去学习dubbo、zookeeper、mq、redis、mysql，底层，都有相似之处

在分布式存储系统的架构里，会涉及到大量的并发编程的一些东西，并发的技术会大量的来实战，而且我会尝试在这个真实项目里，引入几十种并发编程设计模式，把并发编程的艺术带着大家来提高一下

我们构建好的这套分布式存储系统，后面会替代掉那个不太活跃的fastdfs那个分布式文件系统，我们会集成自己的分布式存储系统到电商平台里去，用来存储海量的电商平台的商品的图片之类的一些东西

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/024\_将Server端改造为多线程模式应对大量客户端连接/笔记.doc**

现在的这个socket的程序有什么问题？

这么来写的话是有问题的，一般不能这么来写，所以说我们这里就应该改造一下，让server端的程序支持多个客户端来建立连接，接收人家的请求

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/023\_基于底层网络知识来透彻理解Socket网络编程/笔记.doc**

NIO来做网络编程这块，我们先体验一下传统的Socket网络通信，讲一下，带着大家来梳理一下，包括后面的最基本的NIO网络编程的代码，demo代码，很多高级的特性都没有展现出来

你要真正学会NIO文件读写、网络通信，其实不是通过写demo来学习的，一定要通过后面的项目实战来学习，而且要在项目中深刻的去体会NIO的各种高阶的特性，在NIO技术精讲的这个课里，先跟着我初步的来简单的过一过

NIO的用法、原理

我们就直接切入分布式文件系统的实战项目

public class Server {

public static void main(String[] args) {

ServerSocket serverSocket = new ServerSocket(9000);

Socket socket = serverSocket.accept();

char[] buf = new char[3];

InputStream in = socket.getInputStream();

InputStreamReader inReader = new InputStreamReader(in);

int len = inReader.read(buf);

while(len != -1) {

String data = new String(buf, 0, len);

System.out.println(data);

len = inReader.read(buf);

}

OutputStream out = socket.getOutputStream();

out.write(“收到你的消息了”.getBytes());

inReader.close();

in.close();

out.close();

socket.close();

}

}

public class Client {

public static void main(String[] args) {

try {

Socket socket = new Socket(“localhost”, 9000);

OutputStream out = socket.getOutputStream();

out.write(“大家好”.getBytes());

char[] buf = new char[3];

InputStream in = socket.getInputStream();

InputStreamReader inReader = new InputStreamReader(in);

int len = inReader.read(buf);

while(len != -1) {

String data = new String(buf, 0, len);

System.out.println(data);

len = inReader.read(buf);

}

inReader.close();

in.close();

out.close();

socket.close();

} catch(IOException e) {

e.printStackTrace();

} catch(InterruptedException e) {

e.printStackTrace();

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/035\_NIO核心原理剖析之Selector如何注册Channel/笔记.doc**

打开selector这块，其实底层都是基于操作系统级别的机制来实现的，select机制来监听注册到自己的channel有没有请求过来，每次你在注册的时候，都会指定一个SelectionKey，OP\_READ，OP\_ACCEPT

你要指定你比较感兴趣的这个channel接收到什么请求

我把这个channel交给你selector了，我告诉你，我就对这个channel接收到的连接的请求，ACCETP请求比较感兴趣，麻烦你，兄弟，如果你监听到我给你的这个channel他受到了ACCEPT类型的请求，麻烦告诉我一下

SelectionKey k = findKey(sel);

if (k != null) {

k.interestOps(ops);

k.attach(att);

}

if (k == null) {

// New registration

synchronized (keyLock) {

if (!isOpen())

throw new ClosedChannelException();

k = ((AbstractSelector)sel).register(this, ops, att);

addKey(k);

}

}

这个是你第一次来注册一个channel到selector上去，人家怎么可能提前就有了你这个channel对应的一个SelectionKey呢？不可能。你第一次找一个selector注册channel的时候，人家会反过来调用selector的register方法

把channel和感兴趣的SelectionKey注册到 selector上去，底层肯定是基于操作系统级别的库什么的来实现一个select机制，就是可以实现多路复用的效果，一个selector监听N多个channel对应的Socket接受到的请求

人家会把你的这个channel对应的SelectionKey给放到一个数组里去，初始默认数组大小是3，从index = 0开始放你的key。所以你同一个channel反复注册到selector，下次会直接发现你上一次的那个SelectionKey

他会直接改变你的channel对应的SelectionKey感兴趣的是什么操作

第一次注册：会走底层的类库，把channel和感兴趣的key注册到selector里去

keys数组：他会自动初始化，每个channel感兴趣的key，都会放到keys数组里面去

一个channel反复注册：本质就是改变keys数组里的那个感兴趣的操作而已

keys数组的扩容：如果keys数组越来越多的话，此时就会自动每次乘以2倍来进行扩容

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/026\_传统的Socket网络通信为何无法应对百万并发请求？/笔记.doc**

从代码层面已经demo的实践了一下socket网络编程

在这种模式之下，假如说你接受的客户端的请求是百万个客户端的话，此时在你的服务器端就需要构建百万个线程来支撑百万个客户端

一台普通的4核8G的服务器，虚拟机，100个工作线程，极限了，CPU负载就会很高了

4核8G的服务器，你需要1万台服务器，很恐怖很夸张的，庞大的资源

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/031\_基于NIO技术来彻底重构客户端程序/笔记.doc**

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

import java.util.Random;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.TimeUnit;

public class Client {

public static void main(String[] args) {

for(int i = 0; i < 10; i++){

new Worker().start();

}

}

static class Worker extends Thread {

@Override

public void run() {

SocketChannel channel = null;

Selector selector = null;

try {

channel = SocketChannel.open();

channel.configureBlocking(false);

channel.connect(new InetSocketAddress("localhost", 9000));

selector = Selector.open();

channel.register(selector, SelectionKey.OP\_CONNECT);

boolean isOver = false;

while(!isOver){

selector.select();

Iterator ite = selector.selectedKeys().iterator();

while(ite.hasNext()){

SelectionKey key = (SelectionKey) ite.next();

ite.remove();

if(key.isConnectable()){

if(channel.isConnectionPending()){

if(channel.finishConnect()){

key.interestOps(SelectionKey.OP\_READ);

channel.write(CharsetHelper.encode(CharBuffer.wrap("请求")));

sleep();

}

else{

key.cancel();

}

}

}

else if(key.isReadable()){

ByteBuffer byteBuffer = ByteBuffer.allocate(128);

channel.read(byteBuffer);

byteBuffer.flip();

CharBuffer charBuffer = CharsetHelper.decode(byteBuffer);

String response = charBuffer.toString();

String word = getWord();

if(word != null){

channel.write(CharsetHelper.encode(CharBuffer.wrap(response)));

}

else{

isOver = true;

}

sleep();

}

}

}

} catch (IOException e) {

e.printStackTrace();

}

finally{

if(channel != null){

try {

channel.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(selector != null){

try {

selector.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/027\_NIO是如何基于多路复用技术支持海量客户端的？/笔记.doc**

NIO里就三大核心组件：Buffer、Channel、Selector

其中Buffer就是封装数据的，主要是把数据写入Buffer，或者从Buffer里读数据

Channel就是一个数据管道，他是负责数据传输的，把Buffer给Channel，他可以负责传输数据，比如把数据写入文件，或者从文件读取数据；还可以把数据写入网络，或者从网络读取数据

最后就是一个Selector，他是多路复用组件，专门用在网络编程里的

不需要给每个Socket连接都创建一个线程，不需要每个Channel就创建一个线程，完全可以用一个Selector来多路复用监听N多个Channel是否有请求，这里是基于操作系统底层的select通知机制的，不是轮询各个channel

这样可以避免创建大量的线程

如果说是通过传统的socket编程来解决这个问题，可能比如说1000个客户端需要1000个线程，但是此时通过NIO网络编程模型来是实现的，线程池里就10个线程，就可以完美的处理1000个客户端的请求了

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/028\_基于NIO多路复用技术彻底重构服务端程序/笔记.doc**

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.ClosedChannelException;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

public class Server {

private static CharsetEncoder encoder = Charset.forName(UTF\_8).newEncoder();

private static CharsetDecoder decoder = Charset.forName(UTF\_8).newDecoder();

private ByteBuffer readBuffer;

private Selector selector;

public static void main(String[] args){

Server server = new Server();

server.init();

server.listen();

}

private void init(){

readBuffer = ByteBuffer.allocate(1024);

ServerSocketChannel servSocketChannel;

try {

servSocketChannel = ServerSocketChannel.open();

servSocketChannel.configureBlocking(false);

servSocketChannel.socket().bind(new InetSocketAddress(9000), 100);

selector = Selector.open();

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

} catch (IOException e) {

e.printStackTrace();

}

}

private void listen() {

while(true){

try{

selector.select();

Iterator keysIterator = selector.selectedKeys().iterator();

while(keysIterator.hasNext()){

SelectionKey key = (SelectionKey) keysIterator.next();

keysIterator.remove();

handleKey(key);

}

}

catch(Throwable t){

t.printStackTrace();

}

}

}

private void handleKey(SelectionKey key)

throws IOException, ClosedChannelException {

SocketChannel channel = null;

try{

if(key.isAcceptable()){

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

channel = serverChannel.accept();

channel.configureBlocking(false);

channel.register(selector, SelectionKey.OP\_READ);

}

else if(key.isReadable()){

channel = (SocketChannel) key.channel();

readBuffer.clear();

int count = channel.read(readBuffer);

if(count > 0){

readBuffer.flip();

CharBuffer charBuffer = decoder.decode(readBuffer);

String request = charBuffer.toString();

String response = "";

channel.write(encoder.encode(CharBuffer.wrap(response)));

}

else{

channel.close();

}

}

}

catch(Throwable t){

t.printStackTrace();

if(channel != null){

channel.close();

}

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/023~041资料/034\_NIO核心原理剖析之服务端初始化过程/笔记.doc**

他的底层其实也一定是基于操作系统层面的，网络底层的API，来实现的这么一个东西，NIO，BIO，AIO，怎么来封装，到底层，都是一样的，基于TCP协议，来尝试监听本机的一个端口号，看看谁发送了连接的请求过来

ServerSocketChannel，本质上底层还是基于原生态的TCP协议下的ServerSocket来做的一些事情，封装，来监听底层的某个端口的连接请求

TCP协议的监听某个端口号发送过来的连接请求的程序

被封装在了ServerSocketChannel里面，Channel，数据管道的意思，他就是一个管道，跟底层的网络连接的程序连通了，负责读取人家发送过来的网络连接的请求

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/021\_NIO核心原理剖析之force强制刷磁盘的实现/笔记.doc**

就是给大家讲一下，高阶的一个功能，write写磁盘是进os cache的，但是必须得要force一下可以强制从os cache刷入磁盘中，force的一些原理，源码没必要看，也不方面看，涉及的太底层，操作系统

接口里的一些注释的定义

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/012\_利用FileChannel对文件上共享锁限制文件只读/笔记.doc**

比如说在一个jvm内，是可以通过多个线程就使用一个FileChannel来写，是线程安全的，那如果是多个jvm呢？此时就没办法保证多线程按照顺序来写文件了，并发写文件，可能会有问题的

所以可能会导致文件里的数据出错，混乱

FileChannel给我们提供了一个功能，就是所谓的文件锁，你可以对文件上锁，共享锁，独占锁，如果对文件是上共享锁的话，此时你可以读文件，别人也可以读文件，别人也可以上共享锁

但是没人可以写文件，包括你在内

别人无法加独占锁，别人只能上共享锁，大家都持有对文件的共享锁

不能写数据到文件里去，只能读

现在因为有人加了锁，所以此时你读不到数据的，你必须也加一个文件共享锁，此时才能从里面去读取出来数据，但是写数据是肯定不行的，所有人都可以读文件里的数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/009\_如何利用FileChannel实现磁盘文件的随机写/笔记.doc**

顺序写磁盘文件，不停的基于FileChannel写数据，就是在文件末尾不停的追加数据。在大数据方向的kafka课程里，一个分析，当时讲了kafka如何利用磁盘文件顺序写的超高性能实现了高吞吐

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/013\_如果使用FileChannel对文件上独占锁有什么作用/笔记.doc**

如果有人上了独占锁，别人就不可以上独占锁了，也不能上共享锁，这个时候只能是上独占锁的人自己可以读和写，其他人应该就不可以读和写了

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/020\_NIO核心原理剖析之文件锁是如何实现的/笔记.doc**

FileChannel的高阶的一个用法，在读写磁盘文件的时候，为了避免一个磁盘文件被多个线程，或者是多个进程同时读写，可以对文件加锁，共享锁，别人就加不了独占锁来写了，如果你要写就加独占锁

加锁和释放锁的机制一定是涉及到了非常底层的操作系统的层面

他加锁是可以支持针对文件的position开始的size字节数，分段加锁，就对文件的某个段（region），进行加锁。这个东西是跟底层的操作系统是有关系的，如果操作系统不支持共享锁，就会自动转换为独占锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/004\_使用Direct模式创建的缓冲区有什么作用？/笔记.doc**

缓冲区比较特殊的，叫做direct模式的缓冲区

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/017\_NIO核心原理剖析之FileChannel写入数据到磁盘/笔记.doc**

FileChannel.write方法在干什么事情

FileChannelImpl是JDK内核的一个类，那个源码直接看是看不到的，也没必要去细看，并发底层，synchronized实现，volatile实现，IO流的实现，Socket的实现，其实底层很多都是基于c来写的，而且都是跟操作系统，硬件去打交道的

FileChannel是有一个position的概念，指针，是指向了当前文件的某个位置，所以说你写数据的时候是从当前position开始写数据的，你写入文件之后，文件会不断的变大，容纳你写入进去的一些数据

你写了多少bytes的数据，文件的position就会往前推移多少位

他一定是从buffer里读取数据（你需要调整好buffer里面的数据的标志位，position，limit），写入底层的文件，position，从position开始写，写了多少数据之后，就会顺便递增FileChannel的position

如果写文件的时候，RandomAccessFile设置的是append追加的模式，FileChannel的position刚开始就是在文件的末尾，大家应该知道，完全可以去百度一下看看

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/007\_NIO编程中Buffer和Channel之间的关系是什么/笔记.doc**

Buffer这个组件，我觉得说的还算是比较清楚了，Channel这个组件，数据通道，

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/019\_NIO核心原理剖析之多线程读写磁盘文件如何同步/笔记.doc**

多个线程并发的写磁盘的时候，通过FIleChannel并发写的时候，底层是如何实现同步的？FileChannelImpl这个类，是源码不太方便看，猜测一下大概都知道，在调用和执行FileChannel.write()方法的时候，这个里面一定是有多线程并发同步的代码的，他可以保证同时间只有一个线程在执行底层的文件写逻辑

NIO核心原理剖析

看的主要是一些接口源码和注释，去给分析一下他底层工作的一些原理，FileChannel的position，还有一个就是Buffer的position和limit，这些东西是如何配合起来使用的，你要自己好好去体会一下

底层一定是做了多线程同步的，只有一个线程可以写磁盘文件

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/003\_Buffer缓冲区的核心技术点实验以及图解剖析/笔记.doc**

如果学NIO技术，首当其冲就是要了解所谓的Buffer缓冲区，这个东西是NIO里比较核心的一个部分，一般来说，如果你要通过NIO写数据到文件或者网络，或者是从文件和网络读取数据出来

此时就需要通过Buffer缓冲区来进行

这里有4个概念：capacity、limit、position、mark

这个capacity大概可以认为是缓冲区的容量大小，就是里面包含的数据的大小，比如说下面的代码举个例子，用字节数组（这个是最常见的数据结构）封装了一个ByteBuffer，可以看看里面的capacity是多少

byte[] data = new byte[]{1, 2, 3};

ByteBuffer buffer = ByteBuffer.wrap(data);

System.out.println(buffer.capacity());

这里可以看到这个ByteBuffer的capacity是3，因为里面的字节数组的大小是3

limit这个概念，就是对Buffer缓冲区使用的一个限制，就是说从这个index开始就不能读写数据了，默认情况下limit是跟capacity一样，限制你最多读取capacity容量内的数据，比如上面有3个数据，此时用下面的代码可以进行限制

buffer.limit(1);

System.out.println(buffer.limit());

这就是说从index = 1的位置开始就不能读取和写入了，可以通过CharArray来演示一下，使用put方法即可

position，位置，就是代表了数组中可以开始读写的index，不能大于limit，可以通过CharArray来演示一下，使用put方法即可，他会随着读写一直自动推进，直到跟limit一样，就不让你读了，如果手动设置的position大于了limit，那么自动把limit设置为position

remaining，代表的是position到limit之间的距离

mark，是类似路标的东西，在某个position的时候，设置一下mark，此时就可以设置一个标记，后续调用reset()方法可以把position复位到当时设置的那个mark上去，把position或limit调整为小于mark的值时，就丢弃这个mark

其实说白了，大家会发现这个Buffer缓冲区是很简单的一个概念，说白了就是一个数据缓冲区，支持各种不同的类型，ByteBuffer、CharBuffer、LongBuffer、DoubleBuffer、FloatBuffer、IntBuffer，里面可以包裹不同类型的数组

然后提供的capacity、limit、position、mark等概念，都是让你对缓冲区里的数据读取用的，你主要可以灵活运用limit、position、mark几个标志位，来限定自己要读哪一段数据，哪些不能读，从哪里开始读

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/018\_NIO核心原理剖析之FileChannel从磁盘读取数据到内存/笔记.doc**

NIO读取数据是怎么来做的

源码，看不了多少，搞深了以后没太多东西可以看，结合源码里的接口和注释，搞清楚他基本的一些工作原理我觉得就可以了

对FileChannel底层有一个非常关键以及核心的概念：position

读数据，是从File当前的position开始读的，每次你刚开始比如说你要是初始化了一个FileChannel之后，position是从0开始的，所以你自然就可以读到从0开始的文件里的一些数据

如果你要随机读取文件里的某个位置，直接设置和定位FileChannel的position，就可以限定是从什么位置开始读数据，就可以随机读取文件里的某一段数据。你读取出来了多少字节的数据，FileChannel的position就会往前推移多少位

如果你用FileChannel写数据的话，刚开始position也是0，如果你直接写数据，很可能是会覆盖掉原来的老数据的

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/010\_利用多线程并发写测试FileChannel是线程安全的？/笔记.doc**

答案：多线程同步并发安全的

他一定是一个线程先执行完了写文件，hello world；下一个线程才能有机会去写；不可能多个线程同时基于一个FileChannel来写的

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/005\_如何分配一个Buffer缓冲区以及读写其中数据？/笔记.doc**

我把相关的一些API方法给大家简单介绍一下

NIO这块的硬核的实战

ByteBuffer.allocateDirect(100)：这可以分配一个Direct缓冲区，效率更高

ByteBuffer.wrap(byte[] array)：这就是把你已经有的一个byte数组，作为核心数据放到缓冲区里去

position = 0

capacity = 数组大小

limit = capacity

put(byte b)和get()：这两个就是说对当前position位置放入一个数据，或者读取一个数据

put(byte[] src, int offset, int length)和get(byte[] dst, int offset, int length)：类似上面的，把指定src数组里的一段数据写入缓冲区，或者是从缓冲区里读取数据到数组中

put(byte[] src)和get(byte[] dst)：类似上面那样，就是把数组全部写入缓冲区，以及从缓冲区读取全部数据到数组里去

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/014\_用FileChannel真可以立即写入数据到磁盘吗？/笔记.doc**

你通过FileChannel写数据到磁盘文件的时候，是立即数据可以写到磁盘上吗？不是的，刚写完以后，数据其实是停留在os cache上的，操作系统自己管理的一个内存区域，是一个读写的内存缓冲区

往磁盘文件写数据的时候，他实现落到os cache上的

如果说此时机器宕机，那么os cache里的数据可能就会丢失了，内存里面

如果写磁盘的时候，不是立马落地到磁盘上去，是先落地到os cache上去，此时最大的好处，就是性能高，让磁盘写的性能比较高

kafka、elasticsearch，写数据到磁盘的时候，都是先写入os cache的，后面的操作系统自己不定时的会决定把os cache里的数据写入到磁盘上去，希望数据立马可以写入磁盘上，就不能光是用FileChannel.write()

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/008\_基于FileChannel将数据写入磁盘文件/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/022\_NIO技术这么学就够了吗：不！必须在项目中实战才行！/笔记.doc**

NIO技术精讲以及原理剖析的一些内容，主要就是集中在这周和下周，快速的把NIO的一系诶技术，磁盘读写的原理，网络编程的例子，原理，基本上就完了，NIO学成这个样子，出去面试，基本上也就集中在这个里面的内容了

你即使看10本并发相关的书，你都达不到我们的并发课的里面带给你的并发技术提升的一半，理论的东西很简单，啃啃就啃下来了，一定要实战，并发课，完全手把手带着来一步一步从0开始来构建中间件的项目，那个项目还没结束

在真正的中间件项目的实战里，大量的去运用并发、NIO、网络编程、RPC、Netty，等等一系列的技术，通过分布式微服务注册中心 + 分布式文件系统，两个中间件项目，来实战所有的底层和基础的技术

后面去研究elasticsearch的源码，优化，redis的内核原理，RcoketMQ的原理，自己研发分布式KV存储系统，你就可以游刃有余了，基于这些中间件技术的深入的理解，再在此之上来做业务系统基于各种中间件构建起来的复杂的大规模的架构，高并发架构、高可用架构、海量数据架构、高性能架构

过程，一定是循序渐进的，你要是直接跟我学架构，没问题，1个月就给你讲完各种架构，你能落地吗？各种技术的底层和细节，你搞的定吗？复杂的架构涉及到大量的技术的细节，需要你有深厚逇内功来支撑的

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/002\_学习NIO编程应该从哪个点开始切入？/笔记.doc**

NIO -> Buffer、Channel、Selector，主要是一些API，有很多很多的用法，还有很多很多的高级特性，你没必要上来开始就把各种API，几百个API，几千个API，每个API都了解的很清晰，一些参数，高阶的特性

基本概念了解一下，自己写一些demo的程序体验一下，感受基于NIO读写文件，掌握一下基于NIO做网络编程是怎么玩儿的，有时间的话可以把NIO核心的API底层的一些源码初步的看一下

NIO，Buffer，Channel，文件读写，核心源码

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/006\_在编程中通常是如何调用Buffer API操作缓冲区的？/笔记.doc**

clear()，还原缓冲区里的状态，position设置为0，limit设置为capacity，丢弃mark，但是本质不是删除数据，就是还原那些标记位罢了，因为还原之后就可以复用缓冲区里的空间，覆盖老的数据了

flip()，准备读取刚写入的数据，就是将limit设置为当前position，将position设置为0，丢弃mark。一般就是先写入数据，接着准备从0开始读这段数据，就可以用flip

rewind()，将position设置为0，并且丢弃mark。一般先读取了一遍数据，接着想要再次重新读取一遍数据，这个时候可以用rewind，此时limit是不变的

所以其实一般其实比较少遇到说直接操作Buffer的position、limit和mark之类的，很少。通常都是上述几个方法，直接往里面写入数据，然后打算复用的时候，就clear()，重新可以写数据；如果写了一段数据打算要读取了，此时可以flip()；如果希望重新读取一遍数据，可以rewind()

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/001\_同学，BIO、NIO以及AIO分别是什么？/笔记.doc**

NIO -> 文件读写 -> 网络编程

BIO，NIO，AIO，这个东西其实是Java IO体系技术的发展的过程，简单的了解一下各种不同的IO模型是有什么区别

铺垫了网路的基本课程，大白话网络课程

NIO -> 网络编程，TCP，连接，到底是在干什么，HTTP

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/016\_NIO核心原理剖析之FileChannel的初始化/笔记.doc**

咱们初步的来看一下NIO核心源码的东西，稍微对底层的一些知识有一定的了解就可以

HYPERLINK "eclipse-open:%E2%98%82=demo-nio/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<sun.nio.ch(FileChannelImpl.class%E2%98%83FileChannelImpl" ??? HYPERLINK "eclipse-javadoc:%E2%98%82=demo-nio/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<sun" ?sun?.? HYPERLINK "eclipse-javadoc:%E2%98%82=demo-nio/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<sun.nio" ?nio?.? HYPERLINK "eclipse-javadoc:%E2%98%82=demo-nio/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<sun.nio.ch" ?ch?.FileChannelImpl

是非常底层的一个类，而且是JDK内核里的东西，所以对这块源码，我们就没必要深究他了，最最核心的一点，你只要知道这个FileChannel实例构造的时候，包含了哪些东西，我们大概就可以知道FileChannel在读写文件的时候，底层是有什么东西的

包含的就是文件的路径、还有文件是否支持读写模式

看源码，需要了解一点，就是说，他是基于synchronized做线程同步的，他就可以保证多线程调用同一个流的getChannel()方法的时候，他是可以保证多线程并发安全的

# **File Path: /Users/xiaotingting/Downloads/学习资料/10\_Java NIO技术精讲以及原理剖析unzip/001~022资料/011\_从磁盘文件中读取数据到Buffer缓冲区里/笔记.doc**

读取数据是怎么来做的