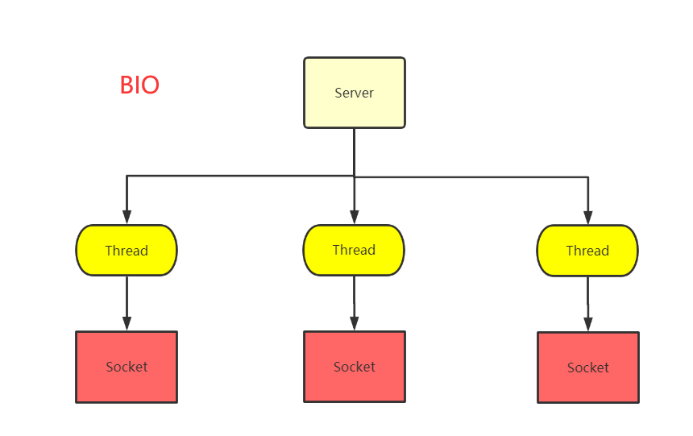
# IO

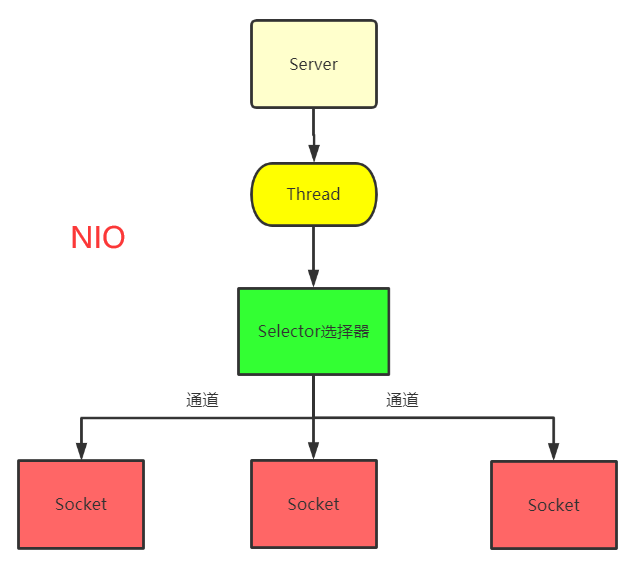
## I/O模型

### BIO



同步并阻塞(传统阻塞型)，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销

### NIO



同步非阻塞，服务器实现模式为一个线程处理多个请求(连接)，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求就进行处理

### AIO

（又称为NIO 2.0）异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，一般适用于连接数较多且连接时间较长的应用

### 适用场景分析

1、BlO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序简单易理解。

2、NIO方式适用于连接数目多且连接比较短(轻操作)的架构，比如聊天服务器，弹幕系统，服务器间通讯等。编程比较复杂，JDK1.4开始支持。

3、AlO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS（OutputStream）参与并发操作，编程比较复杂，JDK7开始支持。

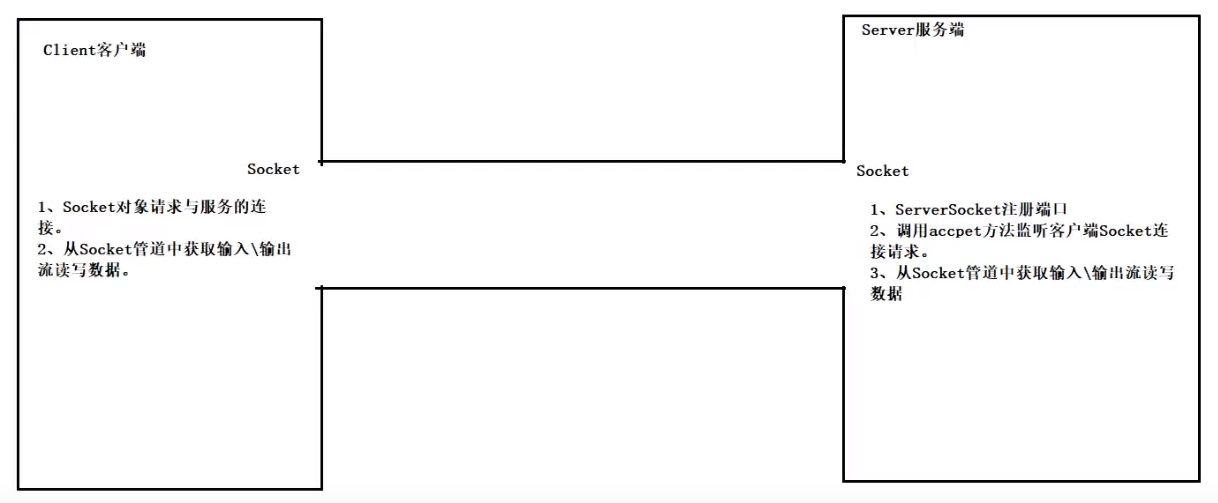
# BIO

## 简介

Java BIO就是传统的java io 编程，其相关的类和接口在java.io

BlO(blocking l/O):同步阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，可以通过线程池机制改善(实现多个客户连接服务器).

### BIO工作模式



## 单发机制

Server

|  |
| --- |
| **public class** Server {  **public static void** main(String[] args) {  **try** {  *//1、定义一个server socket 对象进行服务器端的端口注册* ServerSocket serverSocket =**new** ServerSocket(9999);  *//2、监听客户端的socket链接请求,这里会被阻塞，需要等待client端的链接。三次握手。* Socket socket = serverSocket.accept();  *//3、从socket管道中得到一个字节输入流对象* InputStream is = socket.getInputStream();  *//4、把字节输入流包装成一个缓冲字符输入流* BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(is));  String msg;  if((msg=bf.readLine())!=**null**){  System.***out***.println(**"服务端接口:"**+msg);  }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

Client

|  |
| --- |
| **public class** Client {  **public static void** main(String[] args) **throws** IOException {  *//1、创建socket对象请求服务端的链接* Socket socket = **new** Socket(**"127.0.0.1"**,9999);  *//2、从socket对象中获取一个字节输出流。* OutputStream os = socket.getOutputStream();  PrintStream ps = **new** PrintStream(os);  ps.println(**"hello,world"**);  ps.flush();  } } |

小结

在以上通信中，服务端会一致等待客户端的消息，如果客户端没有进行消息的发送江服务端将一直进入阻塞状态。

同时服务端是按照行获取消息的，这意味着客户端也必须按照行进行消息的发送，否则服务端将进入等待消息的阻塞状态!

可以考虑改为if判断

由于这种机制是端到端的，若连接的Client Socket断了，服务端的Socket也会出现异常机制。

## 多发和多收机制

在上一个demo的基础上作出一些修改，使得Client可以持续输入，然后后Server可以持续输出。保持两端之间通道的连接。

Server

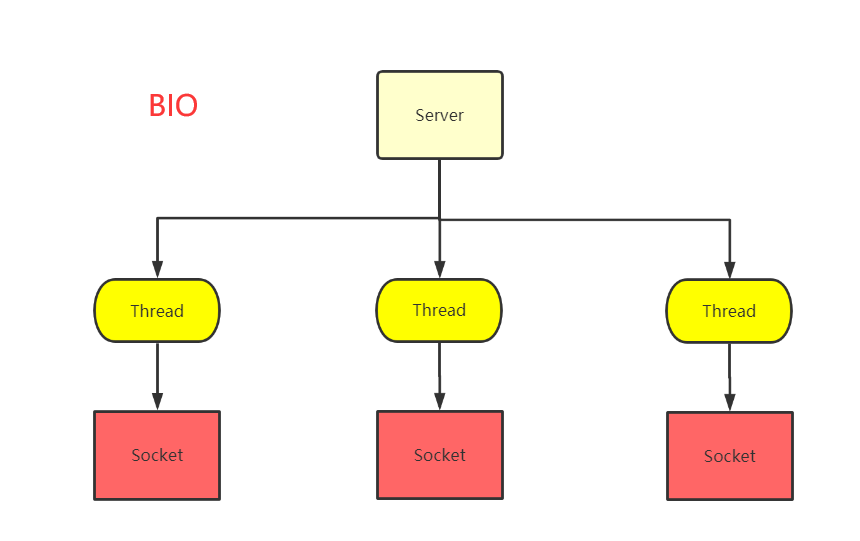
|  |
| --- |
| **public class** Server {  **public static void** main(String[] args) {  **try** {  *//1、定义一个server socket 对象进行服务器端的端口注册* ServerSocket serverSocket =**new** ServerSocket(9999);  *//2、监听客户端的socket链接请求,这里会被阻塞，需要等待client端的链接。三次握手。* Socket socket = serverSocket.accept();  *//3、从socket管道中得到一个字节输入流对象* InputStream is = socket.getInputStream();  *//4、把字节输入流包装成一个缓冲字符输入流* BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(is));  String msg;  **while** ((msg=bf.readLine())!=**null**){  System.***out***.println(**"服务端接口:"**+msg);  }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

Client

|  |
| --- |
| **public class** Client {  **public static void** main(String[] args) **throws** IOException {  *//1、创建socket对象请求服务端的链接* Socket socket = **new** Socket(**"127.0.0.1"**,9999);  *//2、从socket对象中获取一个字节输出流。* OutputStream os = socket.getOutputStream();  PrintStream ps = **new** PrintStream(os);  Scanner sc = **new** Scanner(System.***in***);  **while** (**true**){  System.***out***.print(**"发送"**);  String msg =sc.nextLine();  ps.println(msg);  ps.flush();  }  } } |

## 接收多个客户端

在上述的案例中，一个服务端只能接收一个客户端的通信请求，那么如果服务端需要处理很多个客户端的消息通信请求应该如何处理呢，此时我们就需要在服务端引入线程了，也就是说客户端每发起一个请求，服务端就创建一个新的线程来处理这个客户端的请求，这样就实现了一个客户端一个线程的模型，图解模式如下:



Client

|  |
| --- |
| **public class** Client {  **public static void** main(String[] args) **throws** IOException {  *//1、创建socket对象请求服务端的链接* Socket socket = **new** Socket(**"127.0.0.1"**,9999);  *//2、从socket对象中获取一个字节输出流。* OutputStream os = socket.getOutputStream();  PrintStream ps = **new** PrintStream(os);  Scanner sc = **new** Scanner(System.***in***);  **while** (**true**){  System.***out***.print(**"发送:"**);  String msg =sc.nextLine();  ps.println(msg);  ps.flush();  }  } } |

Server

|  |
| --- |
| **public class** Server {  **public static void** main(String[] args) {  **try** {  *//1、创建socket对象请求服务端的链接* ServerSocket ss =**new** ServerSocket(9999);  *//2、定义一个死循环，负责不断的接受客户端的socket链接请求* **while** (**true**){  Socket socket = ss.accept();  *//3、创建一个独立的线程俩处理这个客户端的socket通信需求* **new** ServerThread(socket).start();  }   } **catch** (IOException e) {  e.printStackTrace();  }  } } |

ServerThread

|  |
| --- |
| **public class** ServerThread **extends** Thread {   **private** Socket **socket**;   **public** ServerThread(Socket socket){  **this**.**socket** = socket;  }   @Override  **public void** run() {  **try** {  **int** port = **socket**.getPort();  *//从socket对象中得到一个字节输入流* InputStream is = **socket**.getInputStream();  BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(is));  String msg;  **while** ((msg=bf.readLine())!=**null**){  System.***out***.println(**"服务端接口:"**+port+**":"**+msg);  }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

每个Socket接收到，都会创建一个线程，线程的竞争、切换上下文影响性能

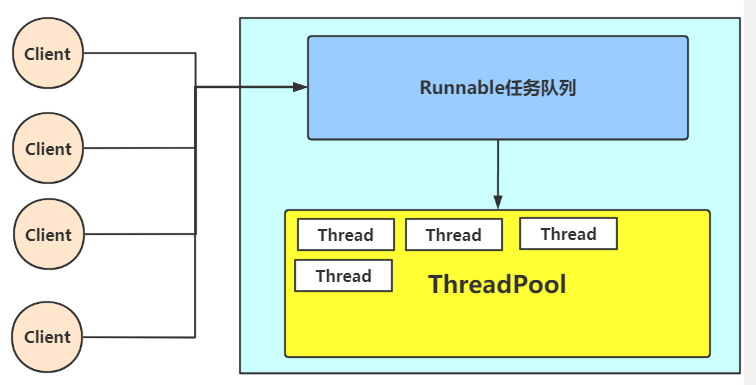
每个线程都会占用栈空间和CPU资源;

并不是每个socket都进行IO操作，无意义的线程处理（等待）

客户端的并发访问增加时。服务端将呈现1:1的线程开销，访问量越大，系统将发生线程栈溢出，线程创建失败，最终导致进程宕机或者僵死，从而不能对外提供服务。

## 伪异步I/O编程

### 概述



上述案例中:客户端的并发访问增加时。服务端将呈现1:1的线程开销，访问量越大，系统将发生线程栈溢出，线程创建失败，最终导致进程宕机或者僵死，从而不能对外提供服务。

接下来我们采用一个伪异步I/O的通信框架，采用线程池和任务队列实现，当客户端接入时，将客户端的Socket封装成一个Task(该任务实现java.lang.Runnable线程任务接口)交给后端的线程池中进行处理。JDK的线程池维护一个消息队列和N个活跃的线程，对消息队列中Socket任务进行处理，由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。’

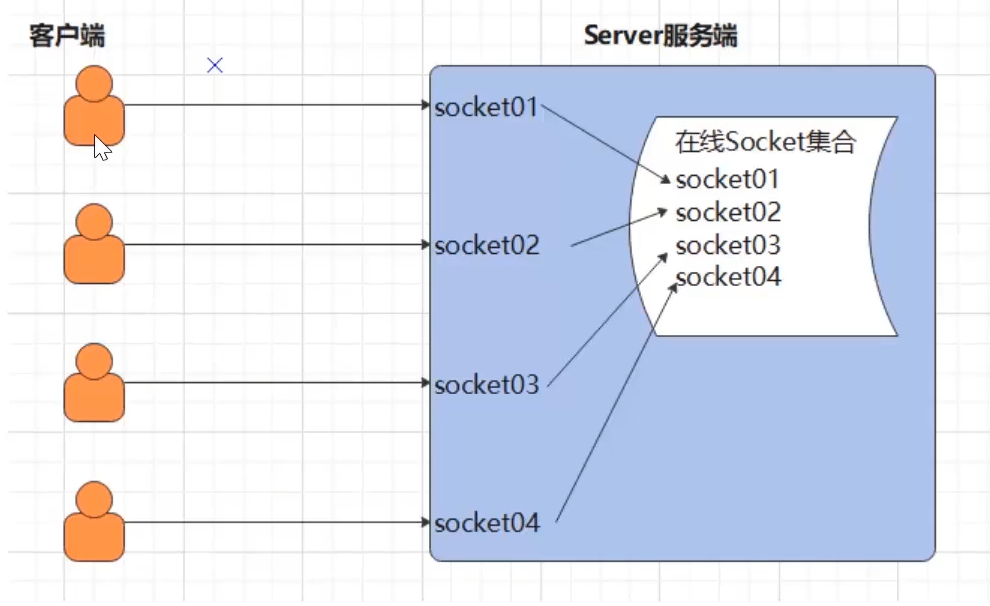
### 小结

伪异步io采用了线程池实现，因此避免了为每个请求创建一个独立线程造成线程资源耗尽的问题，但由于底层依然是采用的同步阻塞模型，因此无法从根本上解决问题。

如果单个消息处理的缓慢，或者服务器线程池中的全部线程都被阻塞，那么后续套接字的I/O消息都将在队列中排队。新的Socket请求将被拒绝，客户端会发生大量连接超时.

## BIO模式下的端口转发思想

需求：需要实现一个客户端的消息可以发送给所有的客户端去接收。



### 服务端

1、注册端口

2、接受客户端的Socket连接，交给一个独立的线程来处理

3、把当前连接的客户端Socket存入到一个所谓的在线Socket集合中

4、接受客户端的消息，然后推送给当前所有在线的Socket接收

## .基于BIO模式下的即时通讯

# NIO

## NIO

### 概述

NIO (New lO)也有人称之为java non-blocking lO是从Java 1.4版本开始引入的一个新的IO API，可以替代标准的Java lO API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的、基于通道的IO操作。NIO将以更加高效的方式进行文件的读写操作。NIO可以理解为非阻塞IO,传统的IO的read和write只能阻塞执行，线程在读写IO期间不能干其他事情，比如调用socket.read()时，如果服务器一直没有数据传输过来，线程就一直阻塞，而NIO中可以配置socket为非阻塞模式。

NIO相关类都被放在java.nio包及子包下，并且对原java.io包中的很多类进行改写。

NIO有三大核心部分:Channel(通道)，Buffer(缓冲区), Selector(选择器)

Java NlO的非阻塞模式，使一个线程从某通道发送请求或者读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此，一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。

通俗理解:NIO是可以做到用一个线程来处理多个操作的。假设有1000个请求过来,根据实际情况，可以分配20或者80个线程来处理。不像之前的阻塞IO那样，非得分配1000个。

### NIO 与 BIO的比较

BIO以流的方式处理数据,而NIO以块的方式处理数据,块I/O的效率比流IO高很多

BIO是阻塞的，NIO则是非阻塞的

BlO基于字节流和字符流进行操作，而NIO基于Channel(通道)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。Selector(选择器)用于监听多个通道的事件（比如:连接请求，数据到达等)，因此使用单个线程就可以监听多个客户端通道

|  |  |
| --- | --- |
| NIO | BIO |
| 面向缓冲区（Buffer） | 面向流（Stream） |
| 非阻塞（Non Blocking IO） | 阻塞IO（Blocking IO） |
| 选择器（Selectors） |  |

NIO可以先将数据写入到缓冲区，然后再有缓冲区写入通道，因此可以做到同步非阻塞。

BIO则是面向的流，读写数据都是单向的。因此是同步阻塞。

## NIO 三大核心原理

NIO有三大核心部分: Channel(通道)，Buffer(缓冲区)，Selector(选择器)

Buffer(缓冲区)

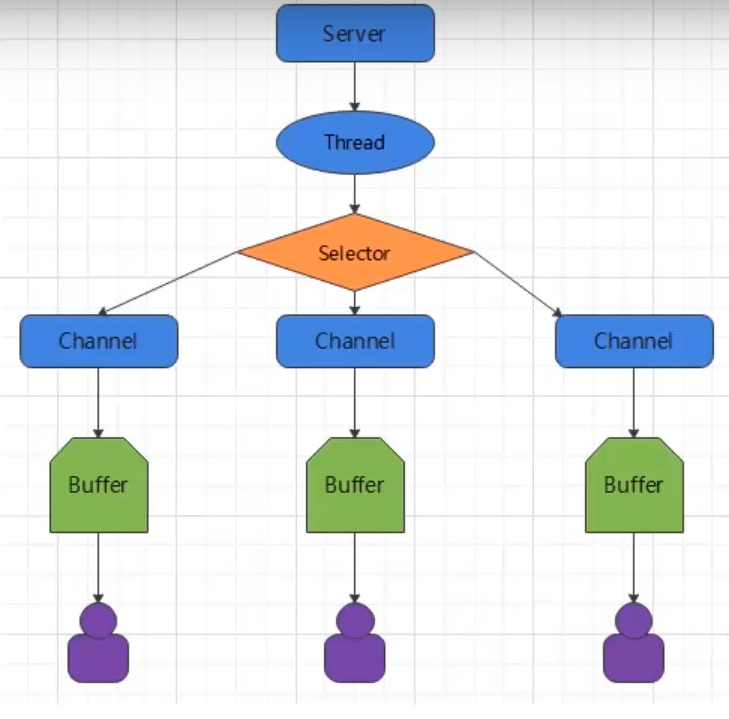
缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。相比较直接对数组的操作，Buffer APl更加容易操作和管理。

Channel(通道)

Java NIO的通道类似流，但又有些不同:既可以从通道中读取数据，又可以写数据到通道。但流的(input或output)读写通常是单向的。通道可以非阻塞读取和写入通道，通道可以支持读取或写入缓冲区，也支持异步地读写。

Selector(选择器)

Selector是一个java NIO组件，可以能够检查一个或多个NIO通道，并确定哪些通道已经准备好进行读取或写入。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接，提高效率



每个channel都会对应一个 Buffer

一个线程对应Selector ,一个Selector对应多个channel(连接)程序

切换到哪个channel是由事件决定的

Selector 会根据不同的事件，在各个通道上切换

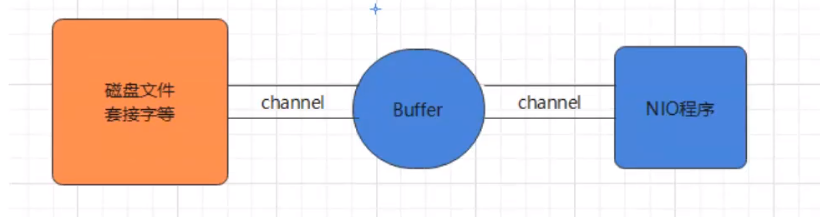
Buffer 就是一个内存块，底层是一个数组

数据的读取写入是通过 Buffer完成的，BlO中要么是输入流，或者是输出流,不能双向，但是NIO的Buffer是可以读也可以写。

Java NIO系统的核心在于:通道(Channel)和缓冲区(Buffer)。通道表示打开到lO设备(例如:文件、套接字)的连接。若需要使用NIO系统，需要获取用于连接IO设备的通道以及用于容纳数据的缓冲区。然后操作缓冲区，对数据进行处理。简而言之，Channel负责传输，Buffer负责存取数据

## 缓冲区Buffer

一个用于特定基本数据类型的容器。由java.nio包定义的，所有缓冲区都是Buffer抽象类的子类。Java NIO中的Buffer主要用于与NIO通道进行交互，数据是从通道读入缓冲区，从缓冲区写入通道中的



Buffer类以及其子类

Buffer就像一个数组，可以保存多个相同类型的数据。根据数据类型不同，有了如下子类

//初始化

IntBuffer intBuffer = IntBuffer.allocate(10);

ByteBuffer

CharBuffer，

DoubleBuffer

FloatBuffer

IntBuffer

LongBuffer

ShortBuffer

### 缓冲区的基本属性

容量(capacity):作为一个内存块，Buffer具有一定的固定大小，也称为"容量"，缓冲区容量不能为负，并且创建后不能更改。

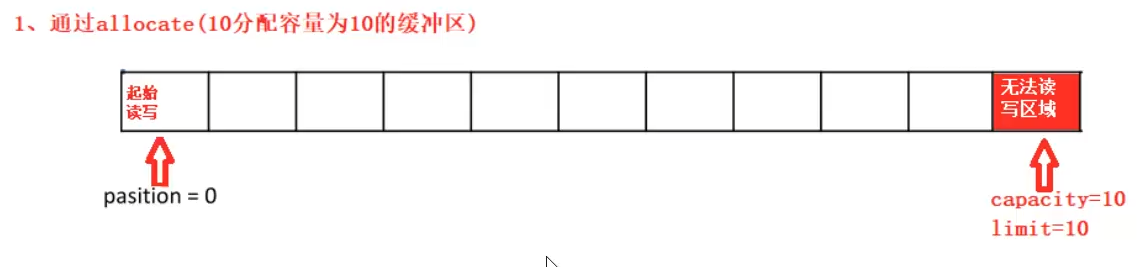
\*\*限制(limit)\*\*表示缓冲区中可以操作数据的大小(limit 后数据不能进行读写)。缓冲区的限制不能为负，并且不能大于其容量。写入模式，限制等于buffer的容量。读取模式下，limit等于写入的数据量。

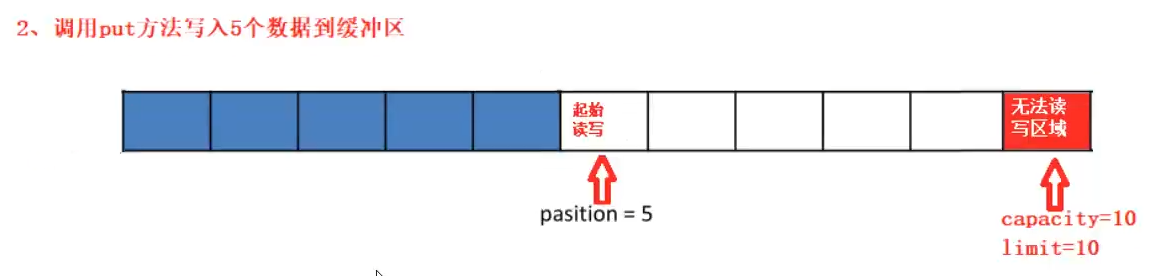
位置(position):下一个要读取或写入的数据的索引。缓冲区的位置不能为负，并且不能大于其限制

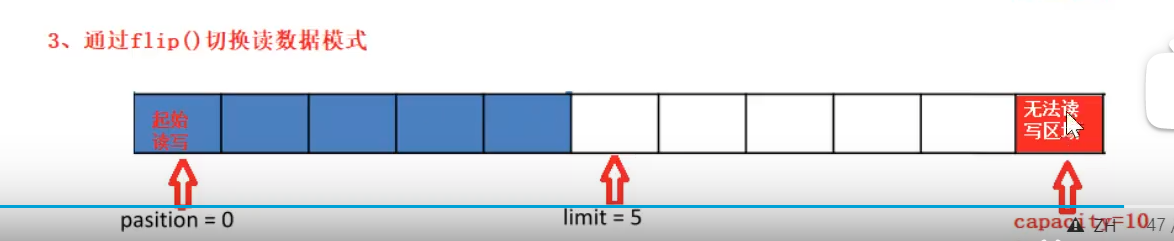
\*\*标记(mark)\*\*与重置(reset):标记是一个索引，通过Buffer中的mark()方法指定Buffer中一个特定的position，之后可以通过调用reset()方法恢复到这个position.

标记、位置、限制、容量遵守以下不变式: 0<=mark <= position <= limit <= capacity

例如图示:







### Buffer常用方法

|  |
| --- |
| abstract Object array()  返回数组，支持这个缓冲区 （可选操作）。    abstract int arrayOffset()  返回在缓冲 第一元这个缓冲区支持数组的偏移（可选操作）。    int capacity()  返回此缓冲区的容量。    Buffer clear()  清除此缓冲区。    Buffer flip()  翻转这个缓冲区。将position赋值给limit，然后position初始化为0    abstract boolean hasArray()  告诉是否这个缓冲区是由一个可访问的数组支持的。    boolean hasRemaining()  告诉当前位置和极限之间是否有任何元素。    abstract boolean isDirect()  告诉这是否是 direct缓冲。    abstract boolean isReadOnly()  告诉是否该缓冲区是只读的。    int limit()  返回此缓冲区的限制。    Buffer limit(int newLimit)  设置此缓冲区的限制。    Buffer mark()  设置此缓冲区的标记位置。    int position()  返回此缓冲区的位置。    Buffer position(int newPosition)  设置此缓冲区的位置。    int remaining()  返回当前位置和极限之间的元素的数目。    Buffer reset()  重置此缓冲区的位置之前标记的位置。    Buffer rewind()  将此缓冲区。 |

### 直接与非直接缓冲区

byte byffer可以是两种类型，一种是基于直接内存（也就是非堆内存)﹔另一种是非直接内存（也就是堆内存)。对于直接内存来说，JVM将会在IO操作上具有更高的性能，因为它直接作用于本地系统的IO操作。而非直接内存，也就是堆内存中的数据，如果要作IO操作，会先从本进程内存复制到直接内存，再利用本地IO处理。

从数据流的角度，非直接内存是下面这样的作用链：

本地IO-->直接内存-->非直接内存-->直接内存-->本地IO

而直接内存是：

本地IO-->直接内存-->本地IO

很明显，在做IO处理时，比如网络发送大量数据时，直接内存会具有更高的效率。直接内存使用allocateDirect创建，但是它比申请普通的堆内存需要耗费更高的性能。不过，这部分的数据是在JVM之外的，因此它不会占用应用的内存。所以呢，当你有很大的数据要缓存，并且它的生命周期又很长，那么就比较适合使用直接内存。只是一般来说，如果不是能带来很明显的性能提升，还是推荐直接使用堆内存。字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 isDirect() 方法来确定。

使用场景

有很大的数据需要存储，它的生命周期又很长

适合频繁的IO操作，比如网络并发场景

## 通道Channel

### 概述

通道（Channel）：由 java.nio.channels 包定义 的。Channel 表示 IO 源与目标打开的连接。 Channel 类似于传统的“流”。只不过 Channel 本身不能直接访问数据，Channel 只能与 Buffer 进行交互。

NIO 的通道类似于流，但有些区别如下：

通道可以同时进行读写，而流只能读或者只能写。

通道可以实现异步读写数据。

通道可以从缓冲读数据，也可以写数据到缓冲:

BIO 中的 stream 是单向的，例如 FileInputStream 对象只能进行读取数据的操作，而 NIO 中的通道(Channel)是双向的，可以读操作，也可以写操作。

3、Channel 在 NIO 中是一个接口

public interface Channel extends Closeable{}

### 常用的Channel实现类

FileChannel：用于读取、写入、映射和操作文件的通道。

DatagramChannel：通过 UDP 读写网络中的数据通道。

SocketChannel：通过 TCP 读写网络中的数据。

ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。 【ServerSocketChanne 类似 ServerSocket , SocketChannel 类似 Socket】

### FileChannel 类

获取通道的一种方式是对支持通道的对象调用getChannel() 方法。支持通道的类如下：

FileInputStream

FileOutputStream

RandomAccessFile

DatagramSocket

Socket

ServerSocket

获取通道的其他方式是使用 Files 类的静态方法 newByteChannel() 获取字节通道。或者通过通道的静态方法 open() 打开并返回指定通道

### FileChannel常用方法

int read(ByteBuffer dst) 从 从 Channel 到 中读取数据到 ByteBuffer

long read(ByteBuffer[] dsts) 将 将 Channel 到 中的数据“分散”到 ByteBuffer[]

int write(ByteBuffer src) 将 将 ByteBuffer 到 中的数据写入到 Channel

long write(ByteBuffer[] srcs) 将 将 ByteBuffer[] 到 中的数据“聚集”到 Channel

long position() 返回此通道的文件位置

FileChannel position(long p) 设置此通道的文件位置

long size() 返回此通道的文件的当前大小

FileChannel truncate(long s) 将此通道的文件截取为给定大小

void force(boolean metaData) 强制将所有对此通道的文件更新写入到存储设备中