

I/O分类一：

第一种方法就是最传统的多进程并发模型(每进来一个新的I/O流会分配一个新的进程管理。)

第二种方法就是I/O多路复用

I/O多路复用：

一个进程（线程）掌管多个I/O,

常见方式为三种：epoll,poll,select

同步异步：

同步/异步：是消息通信机制

同步：发出调用后没有结果不返回

异步：发出调用后就返回,靠异步回调实现功能

阻塞和非阻塞：程序在等待调用结果（消息，返回值）时的状态.

阻塞调用：调用结果返回之前，当前线程会被**挂起**（移出内存）。

非阻塞调用：不能立刻得到结果之前，该调用不会阻塞当前线程。

Nio：同步非阻塞

Aio：异步非阻塞

Bio：同步阻塞

虽然概念不能混为一起，但是多进程并发模型和BIO是相关联的，多个链接多个进程争抢资源，争抢不到堵塞挂起（此处非常浪费性能）

而多路复用使用的select/poll/epoll本身都是nio的，所以简单来说也是强关联。

问题：

异步非阻塞有什么用？

虽然线程没有挂起，但是却没有返回结果有什么用？

解：I/O多路复用，一个线程管理多个I/O

例：简单select 模型实现服务器：

每个socket链接都是一个文件句柄,

**while循环内{**

**监听socket;**

**监听到的socketfd加入句柄集；**

**select（）//检查fd是否ready，**

**如果Fd ready，相应的socket读写**

**}**

//没有ready前没有返回，线程也没有堵塞

Select详解：<http://blog.csdn.net/turkeyzhou/article/details/8609360>

Select vs poll vs epoll

Select:

所有的文件句柄初始化放到一个数组里，

循环内Select遍历看是否可读，可读则执行对应io的逻辑，都不可读则continue。

Poll：

与select类似，poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，poll没有最大文件描述符数量的限制

Epoll

1. Epoll\_create建立线程内核缓冲区（红黑树，链表）
2. 将文件句柄放到红黑树上(fd红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除)，**并注册句柄的回调函数**
3. 当句柄（读/写）可读可写则将句柄放到**就绪列表**
4. Epoll\_wait监控就绪列表（同样需要轮询，本质非异步）

当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

0 epollfd = epoll\_create(FDSIZE);

1 //添加监听描述符事件

2 add\_event(epollfd,listenfd,EPOLLIN);

3 for ( ; ; )

4 {

5 //获取已经准备好的描述符事件

6 ret = epoll\_wait(epollfd,events,EPOLLEVENTS,-1);

7 handle\_events(epollfd,events,ret,listenfd,buf);

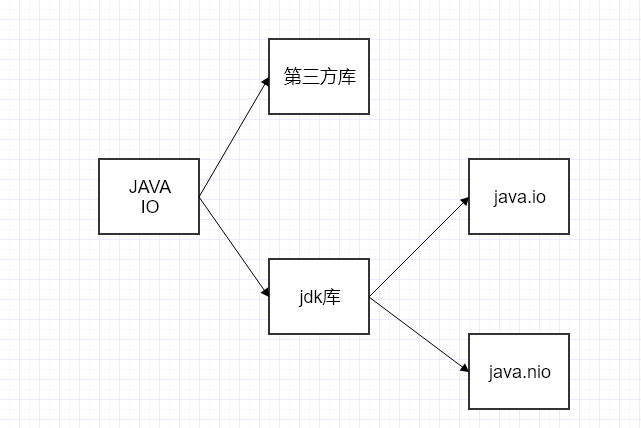
8 }

9 close(epollfd);

使用例子：

多线程并发模型：apache

Epoll：nginx

**java.nio**

Java NIO 由以下几个核心部分组成：

Channels （通道，类似于流）

Buffers

**Selectors（多路复用器）**

NIO的类库和API繁杂，使用麻烦，需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等

**简单使用：**

01 RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");

02 FileChannel inChannel = aFile.getChannel();

04 //create buffer with capacity of 48 bytes

05 ByteBuffer buf = ByteBuffer.allocate(48);

**07 int bytesRead = inChannel.read(buf); //read into buffer.**

08 while (bytesRead != -1) {

10 buf.flip(); //make buffer ready for read

11 while(buf.hasRemaining()){

13  **System.out.print((char) buf.get()); // read 1 byte at a time**

14 }

16 buf.clear(); //make buffer ready for writing

17 bytesRead = inChannel.read(buf);

18 }

19 aFile.close();

**Selector用于管理多个通道**。,

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器

(生成一个channel后)

Selector selector = Selector.open();

SelectionKey key=channel.register(selector, Selectionkey.OP\_READ); //生成一个channel

Set selectedKeys = selector.selectedKeys();//返回selector全部通道

Iterator keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

} keyIterator.remove();

}

**Channel之socketchannel：**

socketChannel.configureBlocking(false); socketChannel.connect(newInetSocketAddress("http://jenkov.com",80));

while(! socketChannel.finishConnect() ){ //wait, or do something else...

}

如果SocketChannel在非阻塞模式下，此时调用connect()，该方法可能在连接建立之前就返回了。为了确定连接是否建立，可以调用finishConnect()的方法。

Read(),write()同理

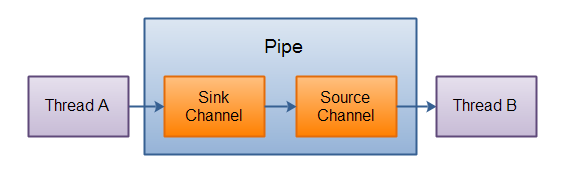
java NIO中的 **ServerSocketChannel** 是一个可以监听新进来的TCP连接的通道, 就像标准IO中的ServerSocket一样。ServerSocketChannel类在 java.nio.channels包中。

**Filechannel**不能非阻塞(jdk7.0后可以)

非阻塞模式与选择器搭配会工作的更好，通过将一或多个SocketChannel注册到Selector，可以询问选择器哪个通道已经准备好了读取，写入等。

**管道**

Pipe pipe = Pipe.open();



要向管道写数据，需要访问sink通道。

像这样：

1 Pipe.SinkChannel sinkChannel = pipe.sink();

2 sinkChannel.write(buf);//

从读取管道的数据，需要访问source通道，像这样：

Pipe.SourceChannel sourceChannel = pipe.source();

int bytesRead = sourceChannel.read(buf);

**IO**                **NIO**

面向流            面向缓冲

阻塞IO           非阻塞IO

无 选择器

面向：Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。ava NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。

阻塞非阻塞：

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，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。

唯一与通道交互的————buffer

例如：（调用read，资源被占用直接返回，reactor实现释放后自动写入buffer）

任何时候读取数据，**都不是直接从通道读取，而是从通道读取到缓冲区**。所以使用NIO读取数据可以分为下面三个步骤：(所以直接返回，待写入缓冲再读取)  
1. 从FileInputStream获取Channel   
2. 创建Buffer   
3. 将数据从Channel读取到Buffer中

**使用NIO写入数据**

使用NIO写入数据与读取数据的过程类似，同样数据不是直接写入通道，而是写入缓冲区，可以分为下面三个步骤：   
1. 从FileInputStream获取Channel   
2. 创建Buffer   
3. 将数据从Channel写入到Buffer中

第三方类库：

也可参考rpc章节netty相关内容

注释：Fd：FileDescriptor是文件描述符，一个socket句柄，可以看做是一个文件，在socket上收发数据，相当于对一个文件进行读写，所以一个socket句柄，通常也用表示文件句柄的fd来表示