

# 目录

---

- [IO概述](#)
  - [什么是Java IO流](#)
  - [IO文件](#)
  - [字符流和字节流](#)
  - [IO管道](#)
  - [Java IO：网络](#)
  - [字节和字符数组](#)
  - [System.in, System.out, System.err](#)
  - [字符流的Buffered和Filter](#)
- [JavaIO流面试题](#)
  - [什么是IO流？](#)
  - [字节流和字符流的区别。](#)
  - [Java中流类的超类主要由那些？](#)
  - [FileInputStream和FileOutputStream是什么？](#)
  - [System.out.println\(\)是什么？](#)
  - [什么是Filter流？](#)
  - [有哪些可用的Filter流？](#)
  - [在文件拷贝的时候，那一种流可用提升更多的性能？](#)

- 说说管道流(Piped Stream)
- 说说File类
- 说说RandomAccessFile?

---

本文参考

并发编程网 – ifeve.com

## IO概述

---

在这一小节，我会试着给出Java IO(java.io)包下所有类的概述。更具体地说，我会根据类的用途对类进行分组。这个分组将会使你在未来的工作中，进行类的用途判定，或者是为某个特定用途选择类时变得更加容易。

## 输入和输出

术语“输入”和“输出”有时候会有一点让人疑惑。一个应用程序的输入往往是另外一个应用程序的输出

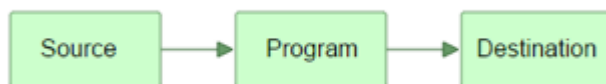
那么`OutputStream`流到底是一个输出到目的地的流呢，还是一个产生输出的流？`InputStream`流到底会不会输出它的数据给读取数据的程序呢？就我个人而言，在第一天学习Java IO的时候我就感觉到了一丝疑惑。

为了消除这个疑惑，我试着给输入和输出起一些不一样的别名，让它们从概念上与数据的来源和数据的流向相联系。

Java的IO包主要关注的是从原始数据源的读取以及输出原始数据到目标媒介。以下是最典型的数据源和目标媒介：

文件  
管道  
网络连接  
内存缓存  
`System.in`, `System.out`, `System.error`(注：Java标准输入、输出、错误输出)

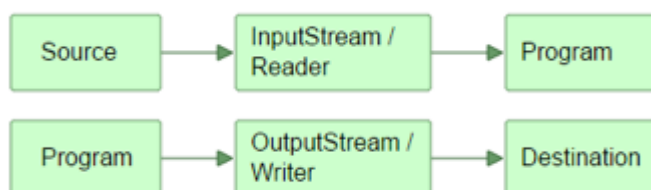
下面这张图描绘了一个程序从数据源读取数据，然后将数据输出到其他媒介的原理：



## 流

在Java IO中，流是一个核心的概念。流从概念上来说是一个连续的数据流。你既可以从流中读取数据，也可以往流中写数据。流与数据源或者数据流向的媒介相关联。在Java IO中流既可以是字节流(以字节为单位进行读写)，也可以是字符流(以字符为单位进行读写)。

类`InputStream`, `OutputStream`, `Reader` 和 `Writer` 一个程序需要`InputStream`或者`Reader`从数据源读取数据，需要`OutputStream`或者`Writer`将数据写入到目标媒介中。以下的图说明了这一点：



`InputStream`和`Reader`与数据源相关联，`OutputStream`和`writer`与目标媒介相关联。

## Java IO的用途和特征

Java IO中包含了许多InputStream、OutputStream、Reader、Writer的子类。这样设计的原因是让每一个类都负责不同的功能。这也就是为什么IO包中有这么多不同的类的缘故。各类用途汇总如下：

- 文件访问
- 网络访问
- 内存缓存访问
- 线程内部通信(管道)
- 缓冲
- 过滤
- 解析
- 读写文本 (Readers / Writers)
- 读写基本类型数据 (long, int etc.)
- 读写对象

当通读过Java IO类的源代码之后，我们很容易就能了解这些用途。这些用途或多或少让我们更加容易地理解，不同的类用于针对不同业务场景。

Java IO类概述表 已经讨论了数据源、目标媒介、输入、输出和各类不同用途的Java IO类，接下来是一张通过输入、输出、基于字节或者字符、以及其他比如缓冲、解析之类的特定用途划分的大部分Java IO类的表格。

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

## Java IO类图

## 什么是Java IO流

Java IO流是既可以从中读取，也可以写入到其中的数据流。正如这个系列教程之前提到过的，流通常会与数据源、数据流向目的地相关联，比如文件、网络等等。

流和数组不一样，不能通过索引读写数据。在流中，你也不能像数组那样前后移动读取数据，除非使用RandomAccessFile 处理文件。流仅仅只是一个连续的数据流。

某些类似PushbackInputStream 流的实现允许你将数据重新推回到流中，以便重新读取。然而你只能把有限的数据推回流中，并且你不能像操作数组那样随意读取数据。流中的数据只能顺序访问。

Java IO流通常是基于字节或者基于字符的。字节流通常以“stream”命名，比如InputStream和OutputStream。除了DataInputStream 和DataOutputStream 还能够读写int, long, float和double类型的值以外，其他流在一个操作时间内只能读取或者写入一个原始字节。

字符流通常以“Reader”或者“Writer”命名。字符流能够读写字符(比如Latin1或者Unicode字符)。可以浏览Java Readers and Writers获取更多关于字符流输入输出的信息。

## InputStream

java.io.InputStream类是所有Java IO输入流的基类。如果你正在开发一个从流中读取数据的组件，请尝试用InputStream替代任何它的子类(比如FileInputStream)进行开发。这么做能够让你的代码兼容任何类型而非某种确定类型的输入流。

## 组合流

你可以将流整合起来以便实现更高级的输入和输出操作。比如，一次读取一个字节是很慢的，所以可以从磁盘中一次读取一大块数据，然后从读到的数据块中获取字节。为了实现缓冲，可以把InputStream包装到BufferedInputStream中。

```
代码示例 InputStream input = new BufferedInputStream(new  
FileInputStream("c:\data\input-file.txt"));
```

缓冲同样可以应用到OutputStream中。你可以实现将大块数据批量地写入到磁盘(或者相应的流)中，这个功能由BufferedOutputStream实现。

缓冲只是通过流整合实现的其中一个效果。你可以把InputStream包装到PushbackInputStream中，之后可以将读取过的数据推回到流中重新读取，在解析过程中有时候这样做很方便。或者，你可以将两个InputStream整合成一个SequenceInputStream。

将不同的流整合到一个链中，可以实现更多种高级操作。通过编写包装了标准流的类，可以实现你想要的效果和过滤器。

## IO文件

在Java应用程序中，文件是一种常用的数据源或者存储数据的媒介。所以这一小节将会对Java中文件的使用做一个简短的概述。这篇文章不会对每一个技术细节都做出解释，而是会针对文件存取的方法提供给你一些必要的知识点。在之后的文章中，将会更加详细地描述这些方法或者类，包括方法示例等等。

## 通过Java IO读文件

如果你需要在不同端之间读取文件，你可以根据该文件是二进制文件还是文本文件来选择使用 `FileInputStream` 或者 `FileReader`。

这两个类允许你从文件开始到文件末尾一次读取一个字节或者字符，或者将读取到的字节写入到字节数组或者字符数组。你不必一次性读取整个文件，相反你可以按顺序地读取文件中的字节和字符。

如果你需要跳跃式地读取文件其中的某些部分，可以使用 `RandomAccessFile`。

## 通过Java IO写文件

如果你需要在不同端之间进行文件的写入，你可以根据你要写入的数据是二进制型数据还是字符型数据选用 `FileOutputStream` 或者 `FileWriter`。

你可以一次写入一个字节或者字符到文件中，也可以直接写入一个字节数组或者字符数据。数据按照写入的顺序存储在文件当中。

## 通过Java IO随机存取文件

正如我所提到的，你可以通过 `RandomAccessFile` 对文件进行随机存取。

随机存取并不意味着你可以在真正随机的位置进行读写操作，它只是意味着你可以跳过文件中某些部分进行操作，并且支持同时读写，不要求特定的存取顺序。

这使得 `RandomAccessFile` 可以覆盖一个文件的某些部分、或者追加内容到它的末尾、或者删除它的某些内容，当然它也可以从文件的任何位置开始读取文件。

下面是具体例子：

```
@Test
// 文件流范例，打开一个文件的输入流，读取到字节数组，再写入另一个文件的输出流
public void test1() {
    try {
        FileInputStream fileInputStream = new FileInputStream(new
        File("a.txt"));
        FileOutputStream fileOutputStream = new FileOutputStream(new
        File("b.txt"));
        byte []buffer = new byte[128];
        while (fileInputStream.read(buffer) != -1) {
            fileOutputStream.write(buffer);
        }
    }
}
```

```
//随机读写，通过mode参数来决定读或者写
RandomAccessFile randomAccessFile = new RandomAccessFile(new
File("c.txt"), "rw");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 字符流和字节流

Java IO的Reader和Writer除了基于字符之外，其他方面都与InputStream和OutputStream非常类似。他们被用于读写文本。InputStream和OutputStream是基于字节的，还记得吗？

Reader Reader类是Java IO中所有Reader的基类。子类包括BufferedReader, PushbackReader, InputStreamReader, StringReader和其他Reader。

Writer Writer类是Java IO中所有Writer的基类。子类包括BufferedWriter和PrintWriter等等。

这是一个简单的Java IO Reader的例子：

```
Reader reader = new FileReader("c:\\data\\myfile.txt");

int data = reader.read();

while(data != -1){

    char dataChar = (char) data;

    data = reader.read();

}
```

你通常会使用Reader的子类，而不会直接使用Reader。Reader的子类包括InputStreamReader, CharArrayReader, FileReader等等。可以查看Java IO概述浏览完整的Reader表格。

### 整合Reader与InputStream

一个Reader可以和一个InputStream相结合。如果你有一个InputStream输入流，并且想从其中读取字符，可以把这个InputStream包装到InputStreamReader中。把InputStream传递到InputStreamReader的构造函数中：

```
Reader reader = new InputStreamReader(inputStream);
```

在构造函数中可以指定解码方式。

## Writer

Writer类是Java IO中所有Writer的基类。子类包括BufferedWriter和PrintWriter等等。这是一个Java IO Writer的例子：

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");

writer.write("Hello World Writer");

writer.close();
```

同样，你最好使用Writer的子类，不需要直接使用Writer，因为子类的实现更加明确，更能表现你的意图。常用子类包括OutputStreamWriter, CharArrayWriter, FileWriter等。Writer的write(int c)方法，会将传入参数的低16位写入到Writer中，忽略高16位的数据。

## 整合Writer和OutputStream

与Reader和InputStream类似，一个Writer可以和一个OutputStream相结合。把OutputStream包装到OutputStreamWriter中，所有写入到OutputStreamWriter的字符都将会传递给OutputStream。这是一个OutputStreamWriter的例子：

```
Writer writer = new OutputStreamWriter(outputStream);
```

## IO管道

Java IO中的管道为运行在**同一个JVM中的两个线程提供了通信的能力**。所以管道也可以作为数据源以及目标媒介。

你不能利用管道与不同的JVM中的线程通信(不同的进程)。在概念上，Java的管道不同于Unix/Linux系统中的管道。在Unix/Linux中，运行在不同地址空间的两个进程可以通过管道通信。在Java中，**通信的双方应该是运行在同一进程中的不同线程**。

### 通过Java IO创建管道

可以通过Java IO中的PipedOutputStream和PipedInputStream创建管道。一个PipedInputStream流应该和一个PipedOutputStream流相关联。

一个线程通过PipedOutputStream写入的数据可以被另一个线程通过相关联的PipedInputStream读取出来。

Java IO管道示例 这是一个如何将PipedInputStream和PipedOutputStream关联起来的简单例子：



```
//使用管道来完成两个线程间的数据点对点传递
@Test
public void test2() throws IOException {
    PipedInputStream pipedInputStream = new PipedInputStream();
    PipedOutputStream pipedOutputStream = new
PipedOutputStream(pipedInputStream);
    new Thread(new Runnable() {
        @Override
        public void run() {
            try { //输出流线程向通过管道写数据
                pipedOutputStream.write("hello input".getBytes());
                pipedOutputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                byte []arr = new byte[128]; //输入流线程从pipe管道中读取数据，实现线程通信
                while (pipedInputStream.read(arr) != -1) {
                    System.out.println(Arrays.toString(arr));
                }
                pipedInputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

管道和线程 请记住，当使用两个相关联的管道流时，务必将它们分配给不同的线程。

`read()`方法和`write()`方法调用时会导致流阻塞，这意味着如果你尝试在一个线程中同时进行读和写，可能会导致线程死锁。

管道的替代 除了管道之外，一个JVM中不同线程之间还有许多通信的方式。实际上，线程在大多数情况下会传递完整的对象信息而非原始的字节数据。但是，如果你需要在线程之间传递字节数据，Java IO的管道是一个不错的选择。

## Java IO：网络

Java中网络的内容或多或少的超出了Java IO的范畴。关于Java网络更多的是在我的Java网络教程中探讨。但是既然网络是一个常见的数据来源以及数据流目的地，并且因为你使用Java IO的API通过网络连接进行通信，所以本文将简要的涉及网络应用。

当两个进程之间建立了网络连接之后，他们通信的方式如同操作文件一样：利用InputStream读取数据，利用OutputStream写入数据。换句话说，Java网络API用来在不同进程之间建立网络连接，而Java IO则用来在建立了连接之后的进程之间交换数据。

基本上意味着如果你有一份能够对文件进行写入某些数据的代码，那么这些数据也可以很容易地写入到网络连接中去。你所需要做的仅仅只是在代码中利用OutputStream替代FileOutputStream进行数据的写入。因为FileOutputStream是OutputStream的子类，所以这么做并没有什么问题。

```
//从网络中读取字节流也可以直接使用OutputStream
public void test3() {
    //读取网络进程的输出流
    OutputStream outputStream = new OutputStream() {
        @Override
        public void write(int b) throws IOException {
        }
    };
}
public void process(OutputStream ouput) throws IOException {
    //处理网络信息
    //do something with the OutputStream
}
```

## 字节和字符数组

从InputStream或者Reader中读入数组

从OutputStream或者Writer中写数组

在java中常用字节和字符数组在应用中临时存储数据。而这些数组又是通常的数据读取来源或者写入目的地。如果你需要在程序运行时需要大量读取文件里的内容，那么你也可以把一个文件加载到数组中。

前面的例子中，字符数组或字节数组是用来缓存数据的临时存储空间，不过它们同时也可以作为数据来源或者写入目的地。举个例子：

```
//字符数组和字节数组在io过程中的作用
public void test4() {
    //arr和brr分别作为数据源
    char []arr = {'a','c','d'};
    CharArrayReader charArrayReader = new CharArrayReader(arr);
    byte []brr = {1,2,3,4,5};
    ByteArrayInputStream byteArrayInputStream = new
    ByteArrayInputStream(brr);
}
```

## System.in, System.out, System.err

System.in, System.out, System.err这3个流同样是常见的数据来源和数据流目的地。使用最多的可能是在控制台程序里利用System.out将输出打印到控制台上。

JVM启动的时候通过Java运行时初始化这3个流，所以你不需要初始化它们(尽管你可以在运行时替换掉它们)。

#### System.in

**System.in**是一个典型的连接控制台程序和键盘输入的InputStream流。通常当数据通过命令行参数或者配置文件传递给命令行Java程序的时候，**System.in**并不是很常用。图形界面程序通过界面传递参数给程序，这是一块单独的Java IO输入机制。

#### System.out

**System.out**是一个PrintStream流。**System.out**一般会把你写到其中的数据输出到控制台上。**System.out**通常仅用在类似命令行工具的控制台程序上。**System.out**也经常用于打印程序的调试信息(尽管它可能并不是获取程序调试信息的最佳方式)。

#### System.err

**System.err**是一个PrintStream流。**System.err**与**System.out**的运行方式类似，但它更多的是用于打印错误文本。一些类似Eclipse的程序，为了让错误信息更加显眼，会将错误信息以红色文本的形式通过**System.err**输出到控制台上。

**System.out**和**System.err**的简单例子：这是一个**System.out**和**System.err**结合使用的简单示例：

```
//测试System.in, System.out, System.err
public static void main(String[] args) {
    int in = new Scanner(System.in).nextInt();
    System.out.println(in);
    System.out.println("out");
    System.err.println("err");
    //输入10，结果是
//      err（红色）
//      10
//      out
}
```

## 字符流的Buffered和Filter

BufferedReader能为字符输入流提供缓冲区，可以提高许多IO处理的速度。你可以一次读取一大块的数据，而不需要每次从网络或者磁盘中一次读取一个字节。特别是在访问大量磁盘数据时，缓冲通常会让IO快上许多。

BufferedReader和BufferedInputStream的主要区别在于，BufferedReader操作字符，而BufferedInputStream操作原始字节。只需要把Reader包装到BufferedReader中，就可以为Reader添加缓冲区(译者注：默认缓冲区大小为8192字节，即8KB)。代码如下：

```
Reader input = new BufferedReader(new FileReader("c:\\data\\input-file.txt"));
```

你也可以通过传递构造函数的第二个参数，指定缓冲区大小，代码如下：

```
Reader input = new BufferedReader(new FileReader("c:\\data\\input-file.txt"),
8 * 1024);
```

这个例子设置了8KB的缓冲区。最好把缓冲区大小设置成1024字节的整数倍，这样能更高效地利用内置缓冲区的磁盘。

除了能够为输入流提供缓冲区以外，其余方面BufferedReader基本与Reader类似。BufferedReader还有一个额外readLine()方法，可以方便地一次性读取一整行字符。

## BufferedWriter

与BufferedReader类似，BufferedWriter可以为输出流提供缓冲区。可以构造一个使用默认大小缓冲区的BufferedWriter(译者注：默认缓冲区大小8 \* 1024B)，代码如下：

```
Writer writer = new BufferedWriter(new FileWriter("c:\\data\\output-
file.txt"));
```

也可以手动设置缓冲区大小，代码如下：

```
Writer writer = new BufferedWriter(new FileWriter("c:\\data\\output-
file.txt"), 8 * 1024);
```

为了更好地使用内置缓冲区的磁盘，同样建议把缓冲区大小设置成1024的整数倍。除了能够为输出流提供缓冲区以外，其余方面BufferedWriter基本与Writer类似。类似地，BufferedWriter也提供了writeLine()方法，能够把一行字符写入到底层的字符输出流中。

**值得注意的是，你需要手动flush()方法确保写入到此输出流的数据真正写入到磁盘或者网络中。**

## FilterReader

与FilterInputStream类似，FilterReader是实现自定义过滤输入字符流的基类，基本上它仅仅只是简单覆盖了Reader中的所有方法。

就我自己而言，我没发现这个类明显的用途。除了构造函数取一个Reader变量作为参数之外，我没看到FilterReader任何对Reader新增或者修改的地方。如果你选择继承FilterReader实现自定义的类，同样也可以直接继承自Reader从而避免额外的类层级结构。

# JavaIO流面试题

---

## 什么是IO流？

它是一种数据的流从源头流到目的地。比如文件拷贝，输入流和输出流都包括了。输入流从文件中读取数据存储在进程(process)中，输出流从进程中读取数据然后写入到目标文件。

## 字节流和字符流的区别。

字节流在JDK1.0中就被引进了，用于操作包含ASCII字符的文件。JAVA也支持其他的字符如Unicode，为了读取包含Unicode字符的文件，JAVA语言设计者在JDK1.1中引入了字符流。ASCII作为Unicode的子集，对于英语字符的文件，可以使用字节流也可以使用字符流。

## Java中流类的超类主要由那些？

`java.io.InputStream` `java.io.OutputStream` `java.io.Reader` `java.io.Writer`

## FileInputStream和FileOutputStream是什么？

这是在拷贝文件操作的时候，经常用到的两个类。在处理小文件的时候，它们性能表现还不错，在大文件的时候，最好使用BufferedInputStream (或 BufferedReader) 和 BufferedOutputStream (或 BufferedWriter)

## System.out.println()是什么？

println是PrintStream的一个方法。out是一个静态PrintStream类型的成员变量，System是一个java.lang包中的类，用于和底层的操作系统进行交互。

## 什么是Filter流？

Filter Stream是一种IO流主要作用是用来对存在的流增加一些额外的功能，像给目标文件增加源文件中不存在的行数，或者增加拷贝的性能。

## 有哪些可用的Filter流？

在java.io包中主要由4个可用的filter Stream。两个字节filter stream，两个字符filter stream. 分别是FilterInputStream, FilterOutputStream, FilterReader and FilterWriter.这些类是抽象类，不能被实例化的。

## 在文件拷贝的时候，那一种流可用提升更多的性能？

在字节流的时候，使用BufferedInputStream和BufferedOutputStream。在字符流的时候，使用BufferedReader 和 BufferedWriter

## 说说管道流(Piped Stream)

有四种管道流， PipedInputStream, PipedOutputStream, PipedReader 和 PipedWriter.在多个线程或进程中传递数据的时候管道流非常有用。

## 说说File类

它不属于 IO流，也不是用于文件操作的，它主要用于知道一个文件的属性，读写权限，大小等信息。

## 说说RandomAccessFile?

它在java.io包中是一个特殊的类，既不是输入流也不是输出流，它两者都可以做到。他是Object的直接子类。通常来说，一个流只有一个功能，要么读，要么写。但是RandomAccessFile既可以读文件，也可以写文件。 DataInputStream 和 DataOutputStream 有的方法，在RandomAccessFile中都存在。

## 参考文章

---

<https://www.imooc.com/article/24305>

<https://www.cnblogs.com/UncleWang001/articles/10454685.html>

<https://www.cnblogs.com/Jixiangwei/p/Java.html>

[https://blog.csdn.net/baidu\\_37107022/article/details/76890019](https://blog.csdn.net/baidu_37107022/article/details/76890019)

---

从应用程序的视角来看的话，我们的应用程序对操作系统的内核发起 IO 调用（系统调用），操作系统负责的内核执行具体的 IO 操作。也就是说，我们的应用程序实际上只是发起了 IO 操作的调用而已，具体 IO 的执行是由操作系统的内核来完成的。

当应用程序发起 I/O 调用后，会经历两个步骤：

- 1. 内核等待 I/O 设备准备好数据
- 2. 内核将数据从内核空间拷贝到用户空间。

先来个例子理解一下概念，以银行取款为例：

**同步**：自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；  
**异步**：委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）；  
**阻塞**：ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；  
**非阻塞**：柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成

### 有哪些常见的 IO 模型？

UNIX 系统下，IO 模型一共有 5 种： 同步阻塞 I/O、同步非阻塞 I/O、I/O 多路复用、信号驱动 I/O 和 异步 I/O。

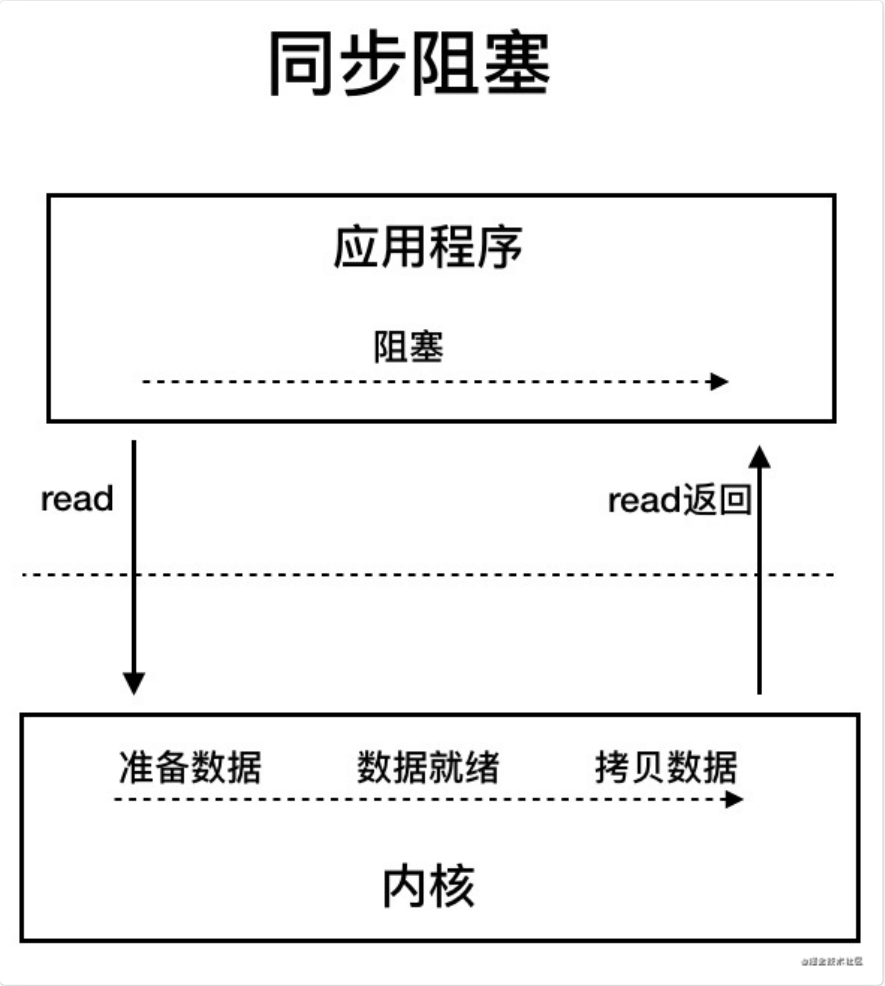
这也是我们经常提到的 5 种 IO 模型。

### Java 中 3 种常见 IO 模型

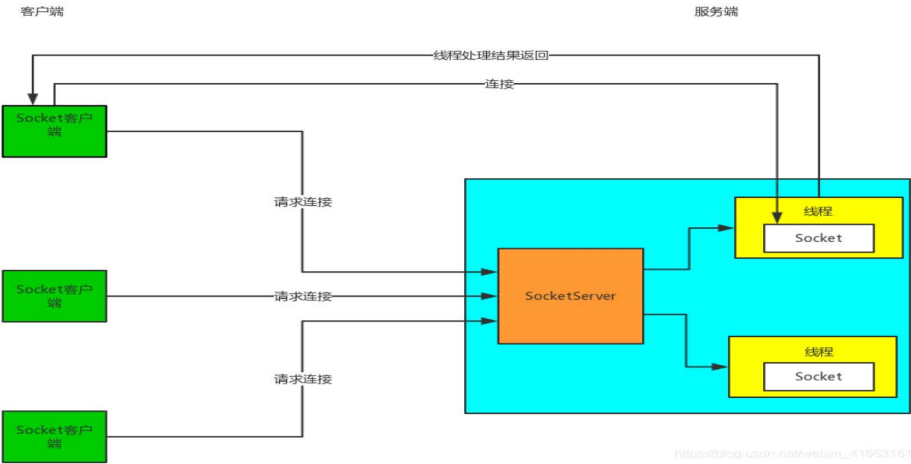
#### BIO (Blocking I/O)jdk1.4之前唯一IO选择

BIO 属于同步阻塞 IO 模型。

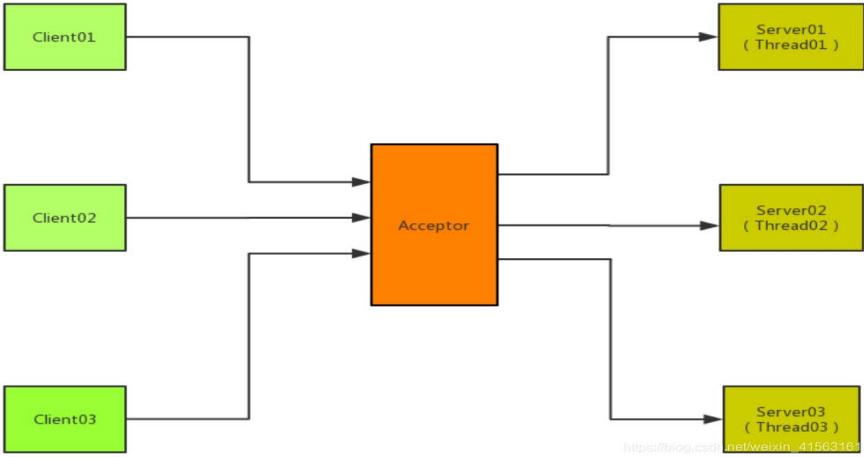
同步阻塞 IO 模型中，应用程序发起 read 调用后，会一直阻塞，直到在内核把数据拷贝到用户空间。



BIO：传统的网络通讯模型，就是BIO，同步阻塞IO，其实就是服务端创建一个ServerSocket，然后就是客户端用一个Socket去连接服务端的那个ServerSocket，ServerSocket接收到了一个的连接请求就创建一个Socket和一个线程去跟那个Socket进行通讯。接着客户端和服务端就进行阻塞式的通信，客户端发送一个请求，服务端Socket进行处理后返回响应，在响应返回前，客户端那边就阻塞等待，什么事情也做不了。这种方式的缺点，每次一个客户端接入，都需要在服务端创建一个线程来服务这个客户端，这样大量客户端来的时候，就会造成服务端的线程数量可能达到了几千甚至几万，这样就可能造成服务端过载过高，最后崩溃死掉。



传统的IO模型的网络服务的设计模式中有俩种比较经典的设计模式：一个是多线程，一种是依靠线程池来进行处理。如果是基于多线程的模式来的话，就是这样的模式，这种也是Acceptor线程模型。





在客户端连接数量不高的情况下，是没问题的。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

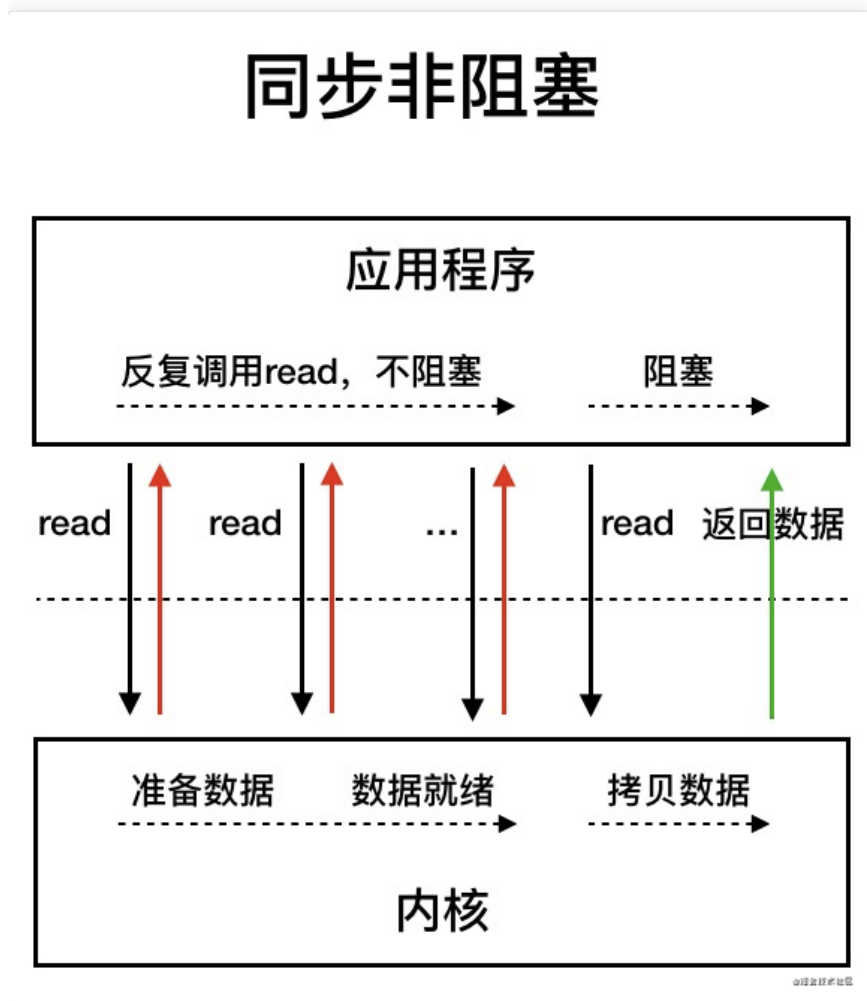
## NIO (Non-blocking/New I/O)

Java 中的 NIO 于 Java 1.4 中引入，对应 `java.nio` 包，提供了 `Channel`，`Selector`，`Buffer` 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。对于高负载、高并发的（网络）应用，应使用 NIO。

Java 中的 NIO 可以看作是 I/O 多路复用模型。也有很多人认为，Java 中的 NIO 属于同步非阻塞 IO 模型。

跟着我的思路往下看，相信你会得到答案！

我们先来看看 同步非阻塞 IO 模型。



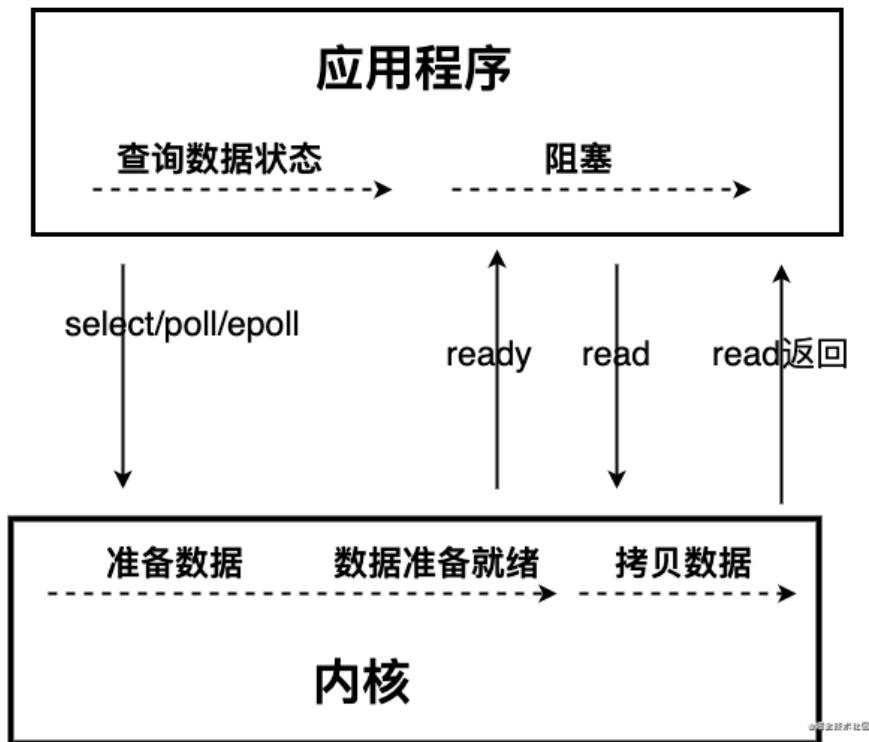
同步非阻塞 IO 模型中，应用程序会一直发起 `read` 调用，等待数据从内核空间拷贝到用户空间的这段时间里，线程依然是阻塞的，直到在内核把数据拷贝到用户空间。

相比于同步阻塞 IO 模型，同步非阻塞 IO 模型确实有了很大改进。通过轮询操作，避免了一直阻塞。

但是，这种 IO 模型同样存在问题：应用程序不断进行 I/O 系统调用轮询数据是否已经准备好的过程是十分消耗 CPU 资源的。

这个时候，I/O 多路复用模型 就上场了。

# I/O多路复用



IO 多路复用模型中，线程首先发起 **select 调用**，询问内核数据是否准备就绪，等内核把数据准备好了，用户线程再发起 **read 调用**。**read 调用**的过程（数据从内核空间->用户空间）还是阻塞的。

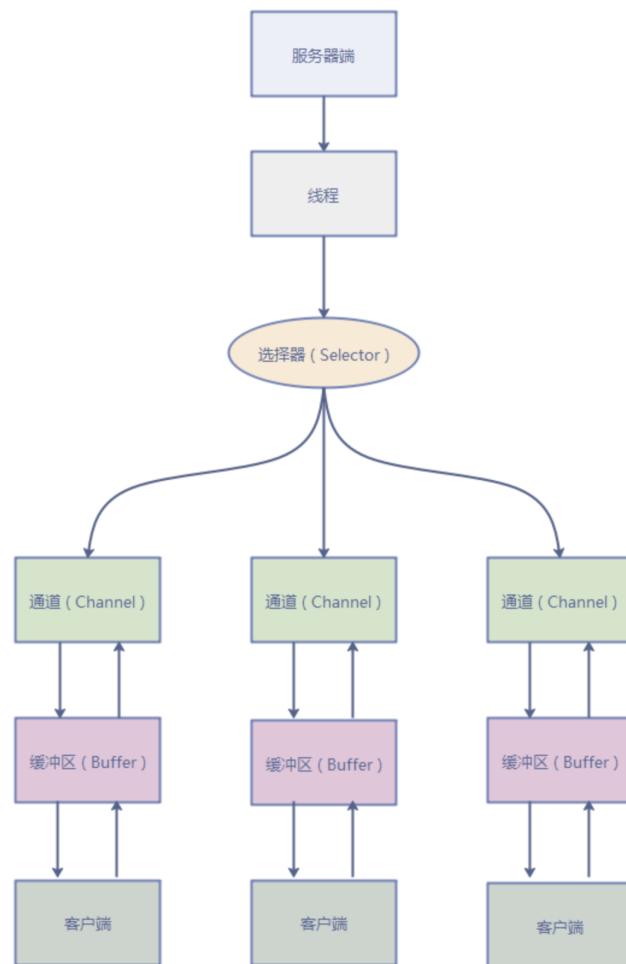
目前支持 IO 多路复用的**系统调用**，有 **select**，**epoll** 等等。**select** 系统调用，是目前几乎在所有的操作系统上都有支持

- **select 调用**：内核提供的系统调用，它支持一次查询多个系统调用的可用状态。几乎所有的操作系统都支持。
- **epoll 调用**：linux 2.6 内核，属于 **select 调用**的增强版本，优化了 IO 的执行效率。

**IO 多路复用模型**，通过减少无效的系统调用，减少了对 CPU 资源的消耗。 **优点**

Java 中的 NIO，有一个非常重要的**选择器 (Selector)**的概念，也可以被称为 **多路复用器**。通过它，只需要一个线程便可以管理多个客户端连接。当客户端数据到了之后，才会为其服务。

**缺点**：虽然减少了数据准备阶段的系统调用提升了性能，但是数据从内核拷贝到用户空间，进程需要阻塞



©掘金技术社区

掘金技术社区

## AIO (Asynchronous I/O)

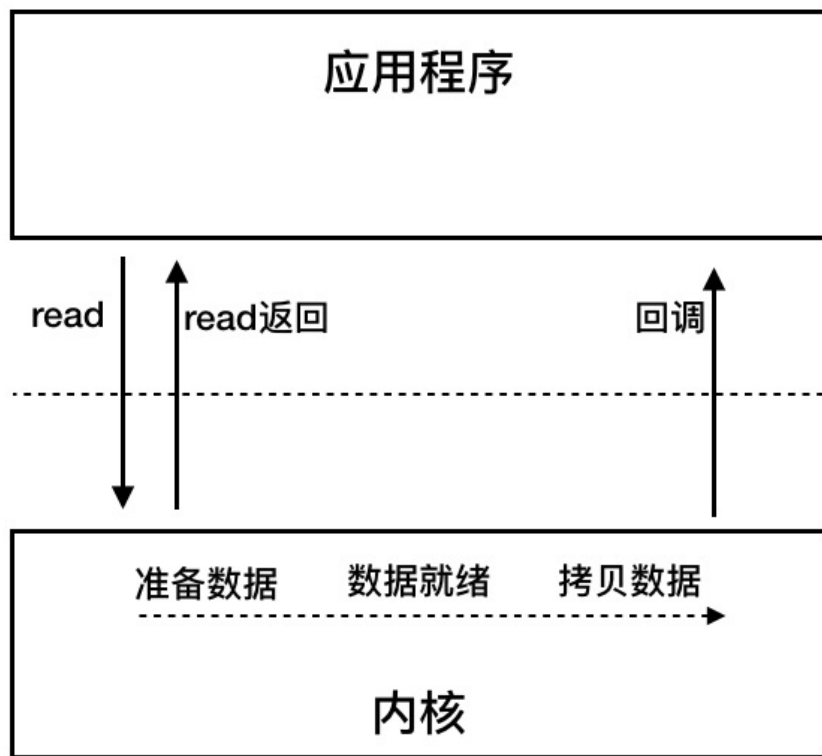
AIO 也就是 NIO 2。Java 7 中引入了 NIO 的改进版 NIO 2,它是异步 IO 模型。

异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

优点：从执行系统调用到数据从内核态拷贝到用户态，以及回调时候都不需要用户进程阻塞

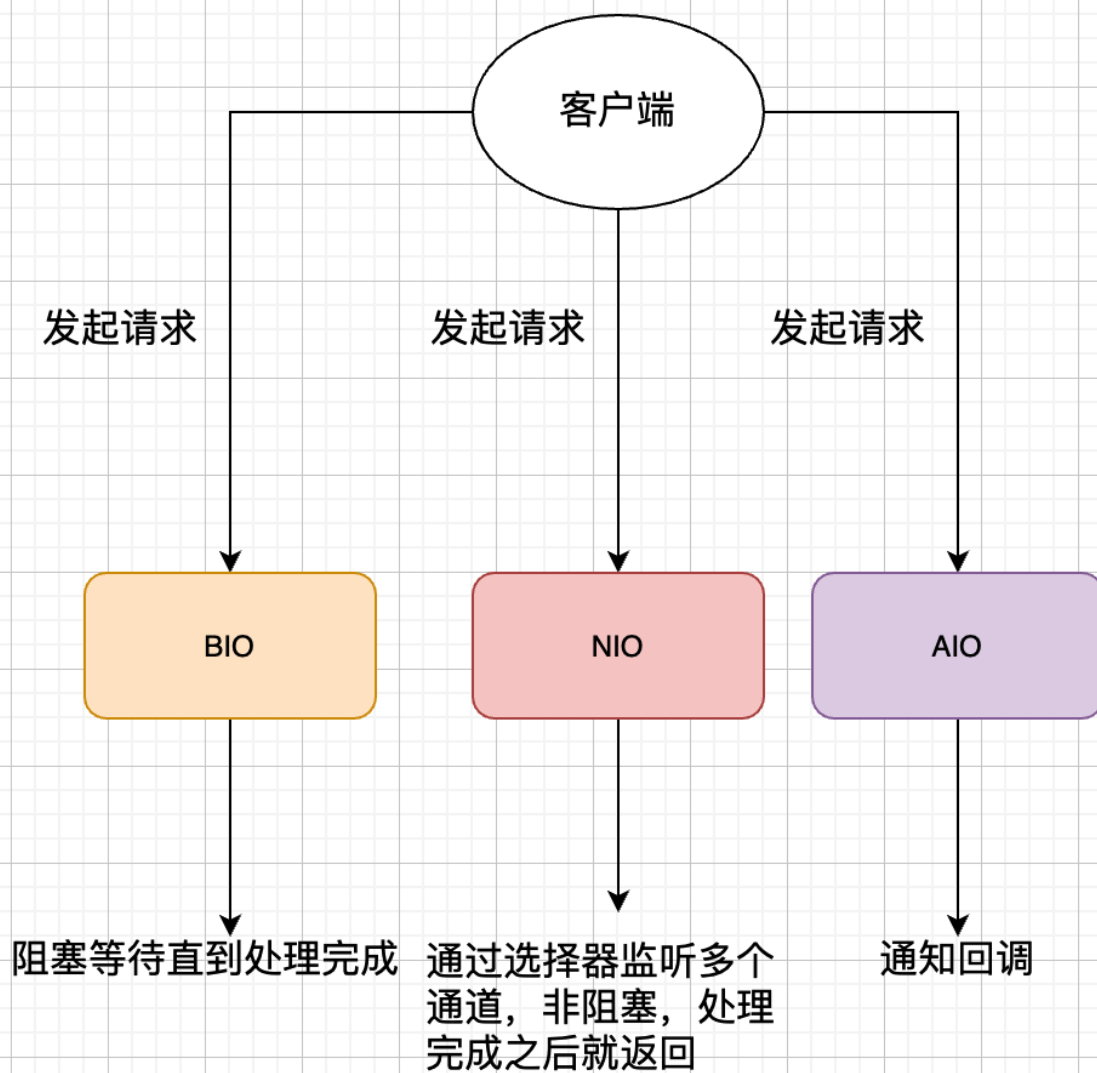


# 异步



目前来说 AIO 的应用还不是很广泛。Netty 之前也尝试使用过 AIO，不过又放弃了。这是因为，Netty 使用了 AIO 之后，在 Linux 系统上的性能并没有多少提升。

最后，来一张图，简单总结一下 Java 中的 BIO、NIO、AIO。



## 参考

- 《深入拆解 Tomcat & Jetty》
- 如何完成一次 IO: <https://lhc687.top/post/如何完成一次-io/>
- 程序员应该这样理解 IO: <https://www.jianshu.com/p/fa7bdc4f3de7>
- 10 分钟看懂，Java NIO 底层原理: <https://www.cnblogs.com/crazymakercircle/p/10225159.html>
- IO 模型知多少 | 理论篇: <https://www.cnblogs.com/sheng-jie/p/how-much-you-know-about-io-models.html>
- 《UNIX 网络编程 卷 1: 套接字联网 API》6.2 节 IO 模型

## 一、什么是NIO

J2SE1.4以上版本中发布了全新的I/O类库。

NIO包（`java.nio.*`）引入了四个关键的抽象数据类型，它们共同解决传统的I/O类中的一些问题。

1. `Buffer`：它是包含数据且用于读写的线形表结构。其中还提供了一个特殊类用于内存映射文件的I/O操作。
2. `Charset`：它提供 Unicode 字符串影射到字节序列以及逆影射的操作。
3. `Channels`：包含 `socket`，`file` 和 `pipe` 三种管道，它实际上是双向交流的通道。
4. `Selector`：它将多元异步I/O操作集中到一个或多个线程中（它可以被看成是 Unix 中 `select（）` 函数或 Win32 中 `WaitForSingleEvent（）` 函数的面向对象版本）。

**Channel，Buffer 和 Selector 构成了核心的 API。** 其它组件，如 `Pipe` 和 `FileLock`，只不过是与三个核心组件共同使用的工具类。因此，在概述中我将集中在这三个组件上。其它组件会在单独的章节中讲到。

## 二、NIO与IO的区别

IO——>面向流、阻塞IO

NIO——>面向缓冲、非阻塞IO、选择器

### 面向流与面向缓冲

Java NIO 和 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**。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。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java 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。**这就增加了处理过程中的灵活性**。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

```
InputStream input = ... ; // get the InputStream from the client socket
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine    = reader.readLine();
String ageLine     = reader.readLine();
String emailLine   = reader.readLine();
String phoneLine   = reader.readLine();
```

请注意处理状态由程序执行多久决定。换句话说，一旦reader.readLine()方法返回，你就知道肯定文本行就已读完，readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个readline()调用返回的时候，你知道这行包含年龄等。正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。

说明了这条原则：**（Java IO: 从一个阻塞的流中读数据）** 而一个NIO的实现会有所不同，下面是一个简单的例子：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
    bytesRead = inChannel.read(buffer);
}
```

`bufferFull()`方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

## 阻塞与非阻塞IO

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。线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，**所以一个单独的线程现在可以管理多个输入和输出通道（channel）。**

## 选择器 (Selectors)

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，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

## Channel 和 Buffer

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。这里有个图示：

overview-channels-buffers1.png

Channel 和 Buffer 有好几种类型。下面是 JAVA NIO 中的一些主要 Channel 的实现：

```
FileChannel
DatagramChannel
SocketChannel
ServerSocketChannel
```

正如你所看到的，这些通道涵盖了 UDP 和 TCP 网络 IO，以及文件 IO。

以下是Java NIO里关键的Buffer实现：



ByteBuffer  
CharBuffer  
DoubleBuffer  
FloatBuffer  
IntBuffer  
LongBuffer  
ShortBuffer




这些Buffer覆盖了你能通过IO发送的基本数据类型： `byte`，`short`，`int`，`long`，`float`，`double` 和 `char`。

## Selector

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：

overview-selectors.png

要使用Selector，得向Selector注册Channel，然后调用它的`select()`方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

## 前言

在上一篇 (<http://www.hollischuang.com/archives/4986>) 文章中，我们了解流的概念以及JavaIO流的基本用法，但JavaIO流的演化不仅是如此简单，有心的读者会发现，在JDK1.4之前的IO类都是基于阻塞的IO（可以从InputStream.read()方法实现中看到由synchronized修饰的代码块），发展到JDK1.4之后NIO提供了selector多路复用的机制以及channel和buffer，再到JDK1.7

的NIO升级提供了真正的异步api.....

Java网络IO涵盖的知识体系很广泛，本文将简单介绍Java网络IO的相关知识：

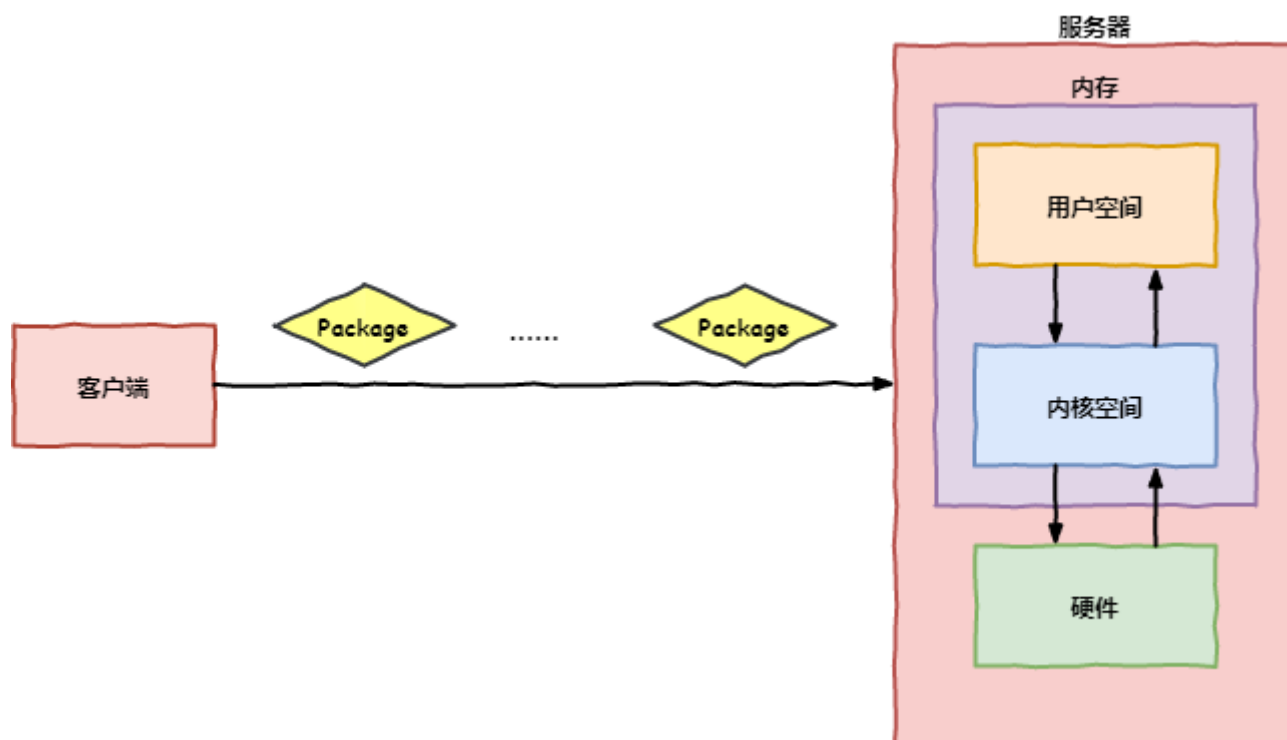
（若文章有不正之处，或难以理解的地方，请多多谅解，欢迎指正）

### 1. 从操作系统开始

为了保护操作系统的安全，会将内存分为**用户空间**和**内核空间**两个部分。**如果用户想要操作内核空间的数据，则需要把数据从内核空间拷贝到用户空间。**

举个栗子，如果服务器收到了从客户端过来的请求，并且想要进行处理，那么需要经过这几个步骤：

- 服务器的网络驱动接受到消息之后，向内核申请空间，并在收到完整的数据包（这个过程会产生延时，因为有可能是通过分组传送过来的）后，将其复制到内核空间；
- 数据从内核空间拷贝到用户空间；
- 用户程序进行处理。



因此我们可以将服务器接收消息理解为两个阶段：

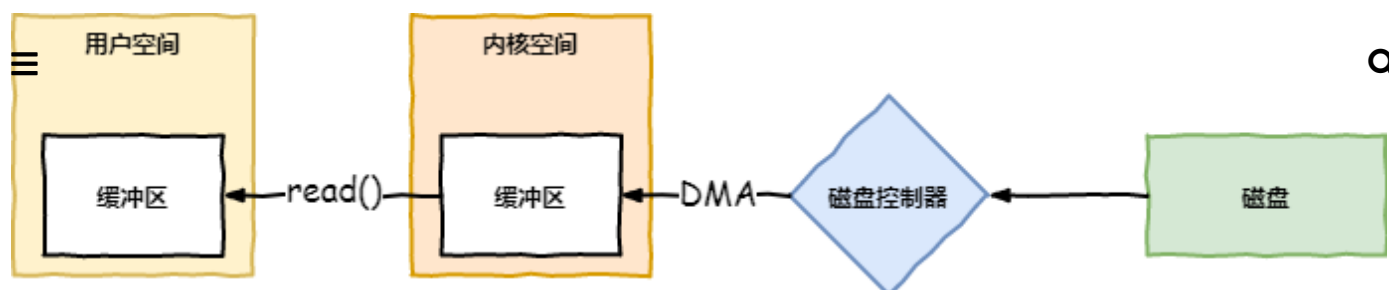
- 等待数据到达
- 将数据从内核空间拷贝到用户空间

## 2. 在操作系统中的IO

在此以Linux操作系统为例。Linux是一个将所有的外部设备都看作是文件来操作的操作系统，在它看来：**everything is a file**，那么我们就把对与外部设备的操作都看作是对文件进行操作。而且我们对一个文件进行读写，都需要通过调用内核提供的系统调用。

而在Linux中，一个基本的IO会涉及到两个系统对象：一个是调用这个IO的进程对象（用户进程），另一个是系统内核。也就是说，当一个read操作发生时，将会经历这些阶段：

- 通过read系统调用，向内核发送读请求；
- 内核向硬件发送读指令，并等待读就绪；
- DMA把将要读取的数据复制到指定的内核缓存区中；
- 内核将数据从内核缓存区拷贝到用户进程空间中。



在此期间会发生几种IO操作：

- **同步IO**：当用户发出IO请求操作后，内核会去查看要读取的数据是否就绪，如果没有，就一直等待。期间用户线程或内存会不断地轮询数据是否就绪。当数据就绪时，再把数据从内核拷贝到用户空间。
- **异步IO**：用户线程只需发出IO请求和接收IO操作完成通知，期间的IO操作由内核自动完成，并发送通知告知用户线程IO操作已经完成。也就是说，在异步IO中，并不会对用户线程产生任何阻塞。
- **阻塞IO**：当用户线程发起一个IO请求操作，而内核要操作的数据还没就绪，则当前线程被挂起，阻塞等待结果返回。
- **非阻塞IO**：如果数据没有就绪，就会返回一个标志信息告知用户线程，当前的数据还没有就绪。当前线程在获得此次请求结果的过程中，还可以做点其他事情。

可能会有读者觉得，怎么同步IO、异步IO和阻塞IO、非阻塞IO的操作好相似，为什么要它们都分出来呢？笔者认为，这同步、异步和阻塞、非阻塞是从不同角度来看待问题的。

### 3. 同步与异步

同步与异步主要是从等待B任务消息通知的角度来说的。

**同步**就是当一个任务A的完成需要依赖另一个任务B时，只有等到B任务完成后，A才能成功地进行，这是一种可靠的任务队列。要么都成功，要么都失败，两个任务的状态可以保持一致。

**异步**是不需要等待任务B完成，只是通知任务B要完成什么工作，任务A也立即执行，只要任务A自己执行完了那么整个任务就算完成了。至于任务B最终是否真正完成，A任务无法确定，所以这是不可靠的一种任务队列。

举个栗子，假如小J要去银行柜台办事，拿号排队。如果他只盯着号码提示牌，还时不时问是否到他了，这就是**同步**，A（客户）任务自己去检查B（柜员）任务是否完成；如果他拿了号之后就去打电话了，等到排到他的时候柜员通知他去办理业务，这就是**异步**，A任务等待B任务主动通知自己任务完成。他们之间的区别就在于，等待消息通知的方式不同。

### 4. 阻塞与非阻塞

**阻塞**就是指在调用结果返回之前，当前线程会被挂起，一直处于等待消息通知的状态，不能执行其他业务。只有当调用结果返回之后才能进行其他操作。

**非阻塞**与阻塞的概念相对应，就是指不能立即得到结果之前，该函数不会阻塞当前线程，而是会立即返回。虽然非阻塞的方式看上去可以明显提高CPU的利用率，但是也会使系统的线程切换增加，需要好好评估增加的CPU执行时间能不能步长系统的切换成本。

我们继续用上面的栗子，小J无论是在排队还是拿号等通知，如果在这个等待的过程中，小J除了等待消息通知之外就做不了其他的事情，那么该机制就是阻塞的。如果他可以一边打电话一边等待，这个状态就是非阻塞的。

## 5. 同步、异步与阻塞、非阻塞

其实可能会有其他读者把同步与阻塞等同起来，实际上这两个是不同的。对于同步来说，很多时候当前线程还是在激活状态，只是逻辑上当前函数没有返回而已，此时，线程也会去处理其他的信息。也就是说，同步、阻塞其实是在消息通知机制下从不同角度对当前线程状态的描述。

### 5.1 同步阻塞形式

这是效率最低的一种方式，拿上面的栗子来说，就是小J心无旁骛地排队，什么别的事都不做。

在这里，同步与阻塞体现在：

- **同步**：小J等待队伍排到他办理业务；
- **阻塞**：小J在等待队伍排到他的过程中，不做其他任务处理。

### 5.2 异步阻塞形式

如果小J在银行等待办理业务的时候，领了号，这时候就采用了异步的方式去等待消息被触发（通知），等着柜员喊他的号而不是时刻盯着是不是排到他了。但是在这段时间里，他还是不能离开银行去做其他的事情，那么很显然，他被阻塞在这个等待喊号的操作上了。

在这里，异步与阻塞体现在：

- **异步**：排到小J的话柜员会喊他的号码；
- **阻塞**：等待喊号的过程中，不能做其他事情。

### 5.3 同步非阻塞形式

实际上效率也是低下。小J在排队的时候可以打电话，但是要边打电话边看看还有多久才排到他。如果将打电话和观察排队情况看成是程序中的两个操作的话，这个程序需要在这两个不同的行为之间来回切换。

在这里，同步与非阻塞体现在：

- **同步**：排队等待轮到他办理业务；
- **非阻塞**：可以在排队过程中打电话，只不过要时不时看看还要多久才排到他办理业务。

## 5.4 异步非阻塞形式

这是一个效率更高的模式。小J在拿号之后可以去打电话，只要等待柜员喊号就可以了，在这里打电话是等待者的事情，而通知小J办理业务是柜员的事情。

在这里，异步和非阻塞体现在：

- **异步**：柜员喊小J去办理业务；
- **非阻塞**：在等待喊号的过程中，小J去打电话，只要接收到柜员喊号的通知即可，无需关注是否队伍的进度。

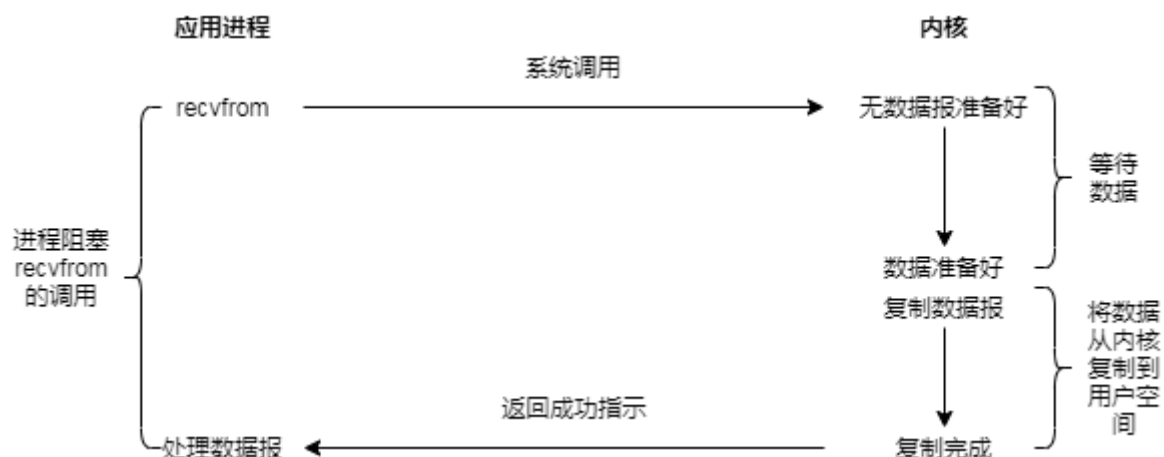
也就是说，同步和异步仅需关注消息如何通知的机制，而阻塞和非阻塞关注的是在等待消息通知的过程中能不能去做别的事。在同步情况下，是由处理者自己去等待消息是否被触发，而异步情况下是由触发机制来通知处理者处理业务。

## 6. Linux五种IO模型

在我们了解Linux操作系统的IO操作，以及同步与异步、阻塞与非阻塞的概念之后，我们来看看Linux系统中根据同步、异步、阻塞、非阻塞实现的五种IO模型。以Linux下的系统调用recv为例，是一个用于从套接字上接收一个消息，因为是系统调用，所以在调用的时候，会从用户空间切换到内核空间运行一段时间后，再切换回来。在默认情况下recv会等到网络数据到达并复制到用户空间或发生错误时返回。

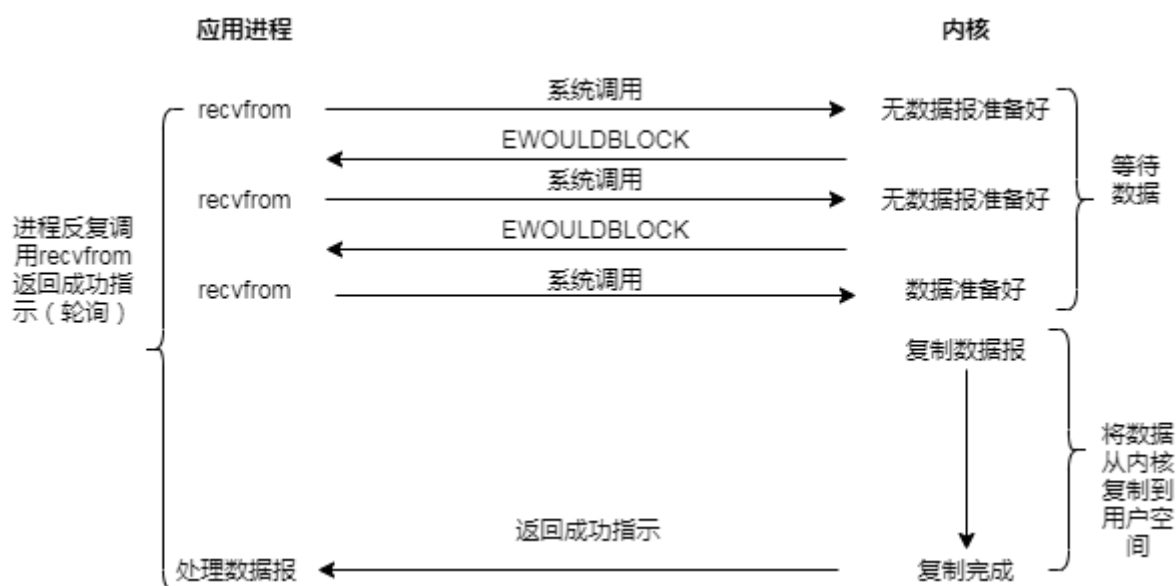
### 6.1 同步阻塞IO模型

从系统调用recv到将数据从内核复制到用户空间并返回，在这段时间内进程始终阻塞。就相当于，小J想去柜台办理业务，如果柜台业务繁忙，他也要排队，直到排到他办理完业务，才能去做别的事。显然，这个IO模型是同步且阻塞的。



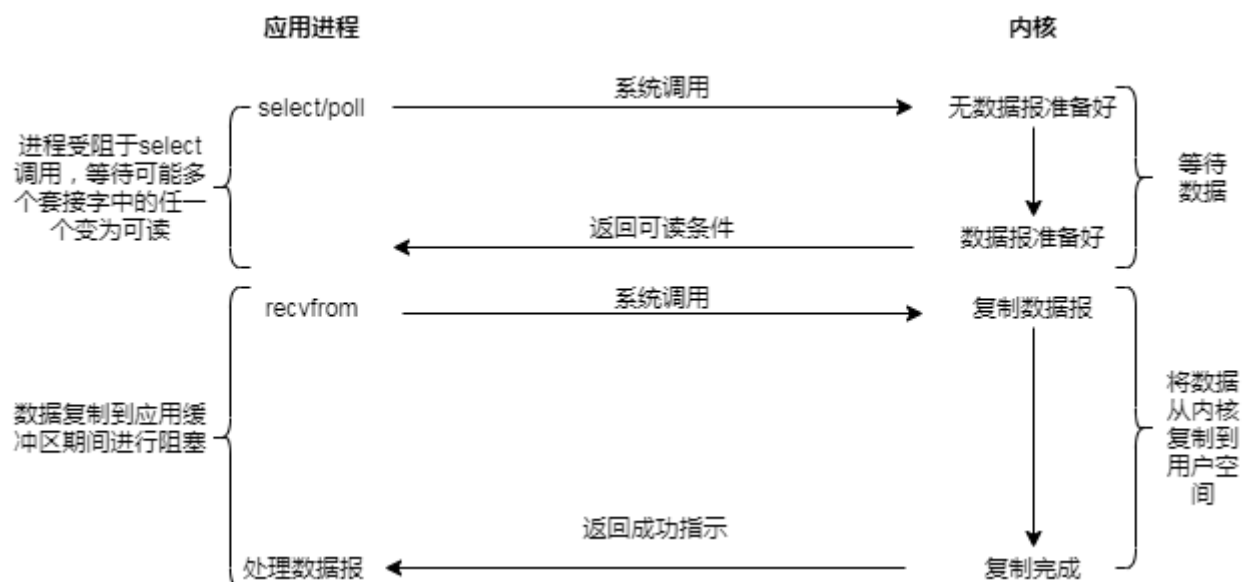
## 6.2 同步非阻塞IO模型

在这里recv不管有没有获得数据都返回，如果没有数据的话就过段时间再调用recv看看，如此循环。就像是小J来柜台办理业务，发现柜员休息，他离开了，过一会又过来看看营业了没，直到终于碰到柜员营业了，这才办理了业务。而小J在中间离开的时间，可以做他自己的事情。但是这个模型只有在检查无数据的时候是非阻塞的，在数据到达的时候依然要等待复制数据到用户空间（办理业务），因此它还是同步IO。



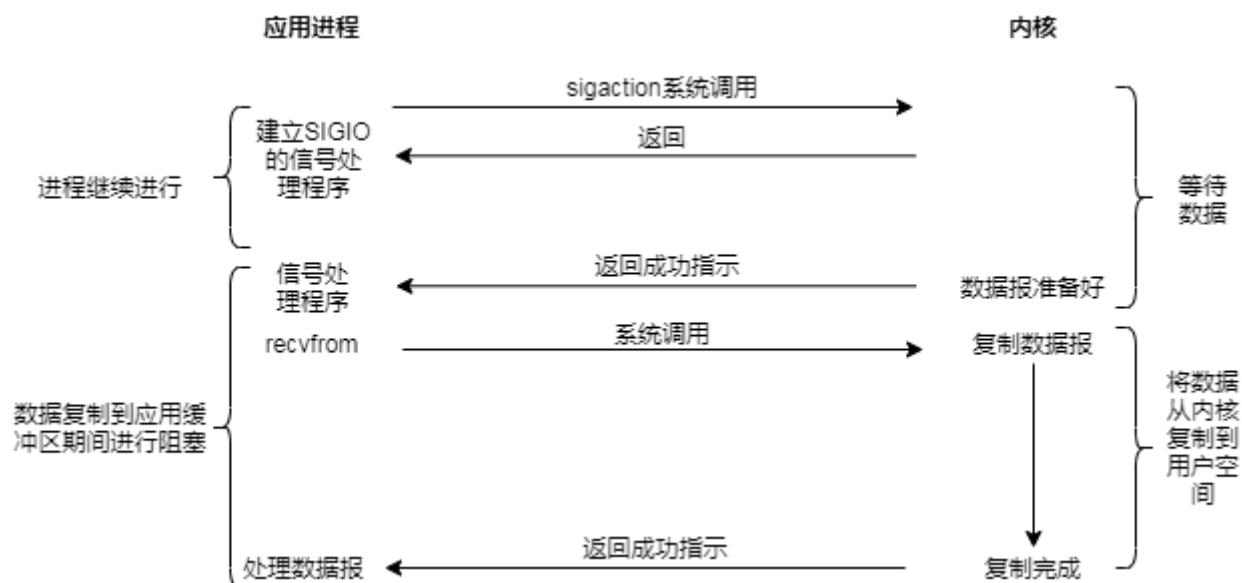
## 6.3 IO复用模型

在IO复用模型中，调用recv之前会先调用select或poll，这两个系统调用都可以在内核准备好数据（网络数据已经到达内核了）时告知用户进程，它准备好了，这时候再调用recv时是一定有数据的。因此在这一模型中，进程阻塞于select或poll，而没有阻塞在recv上。就相当于，小J来银行办理业务，大堂经理告诉他现在所有柜台都有人在办理业务，等有空位再告诉他。于是小J就等啊等（select或poll调用中），过了一会儿大堂经理告诉他有柜台空出来可以办理业务了，但是具体是几号柜台，你自己找下吧，于是小J就只能挨个柜台地找。



## 6.4 信号驱动IO模型

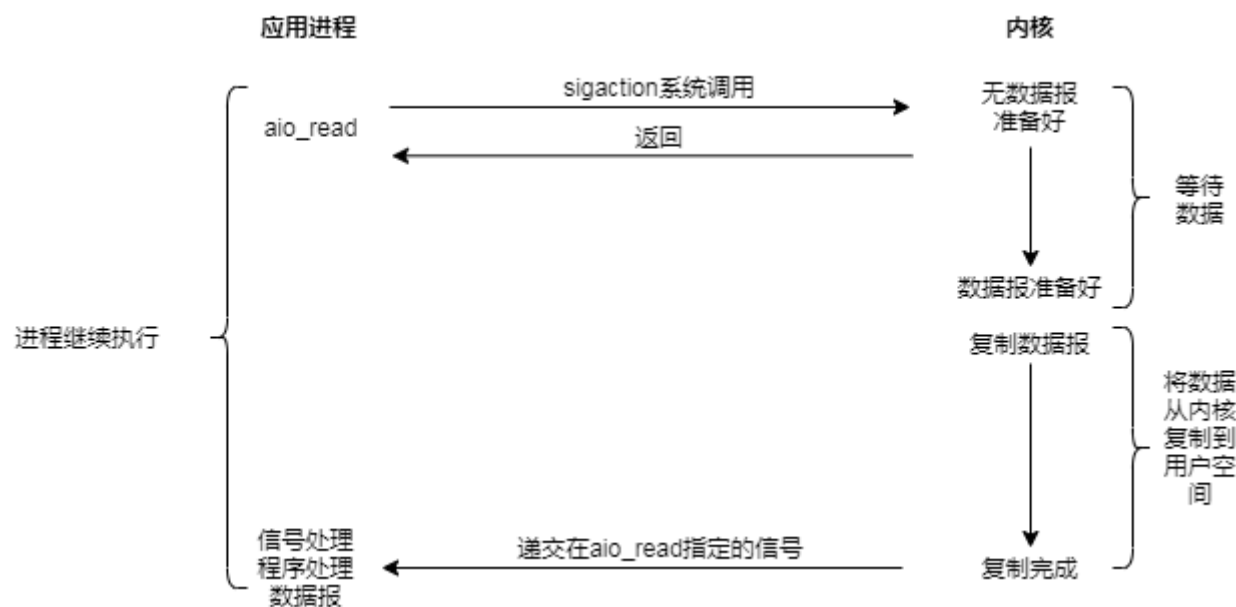
此处会通过调用 `sigaction` 注册信号函数，在内核数据准备好的时候系统就中断当前程序，执行信号函数（在这里调用 `recv`）。相当于，小J让大堂经理在柜台有空位的时候通知他（注册信号函数），等没多久大堂经理通知他，因为他是银行的VIPPP会员，所以专门给他开了一个柜台来办理业务，小J就去特席柜台办理业务了。但即使在等待的过程中是非阻塞的，但在办理业务的过程中依然是同步的。



## 6.5 异步IO模型

调用 `aio_read` 令内核把数据准备好，并且复制到用户进程空间后执行事先指定好的函数。就像是，小J交代大堂经理把业务给办理好了就通知他来验收，在这个过程中小J可以去做自己的事情。这就是真正的异步IO。





我们可以看到，前四种模型都是属于同步IO，因为在内核数据复制到用户空间的这一过程都是阻塞的。而最后一种异步IO，通过将IO操作交给操作系统处理，当前进程不关心具体IO的实现，后来再通过回调函数，或信号量通知当前进程直接对IO返回结果进行处理。

## 7. BIO、NIO、AIO的区别

上文谈到IO的四种模式：**同步阻塞IO**、**同步非阻塞IO**、**异步阻塞IO**、**异步非阻塞IO**，在JavaIO中提供了三种模式的实现：BIO（同步阻塞IO）、NIO（同步非阻塞IO）、AIO（异步非阻塞IO）。至于这四种模式之间的区别，上文已经有较为详细的介绍了，接下来笔者将对这三种JavaIO类型之间的区别进行介绍。

- **BIO**：同步并阻塞，在服务器中实现的模式为**一个连接一个线程**。也就是说，客户端有连接请求的时候，服务器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，当然这也可以通过**线程池机制**改善。BIO**一般适用于连接数目小且固定的架构**，这种方式对于服务器资源要求比较高，而且并发局限于应用中，是JDK1.4之前的唯一选择，但好在程序直观简单，易理解。
- **NIO**：同步并非阻塞，在服务器中实现的模式为**一个请求一个线程**，也就是说，客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到有连接IO请求时才会启动一个线程进行处理。NIO**一般适用于连接数目多且连接比较短（轻操作）的架构**，并发局限于应用中，编程比较复杂，从JDK1.4开始支持。比如聊天服务器
- **AIO**：异步并非阻塞，在服务器中实现的模式为**一个有效请求一个线程**，也就是说，客户端的IO请求都是通过操作系统先完成之后，再通知服务器应用去启动线程进行处理。AIO**一般适用于连接数目多且连接比较长（重操作）的架构**，充分调用操作系统参与并发操作，编程比较复杂，从JDK1.7开始支持。比如相册服务器

## 结语



本文从操作系统进行文件读写入手，对同步、异步、阻塞、非阻塞以及它们组合而成的IO模式进行了介绍，还了解Linux操作系统中的五种IO模型，以及重新回到JavaIO，看待BIO、NIO、AIO之间的区别。

如果本文对你有帮助，请给一个赞吧，这会是我最大的动力~

参考资料：

简述同步IO、异步IO、阻塞IO、非阻塞IO之间的联系与区别  
(<https://www.cnblogs.com/felixzh/p/10345929.html>).

BIO、NIO、AIO有什么区别？  
(<https://blog.csdn.net/meism5/article/details/89469101>).