

# 课程主题

IO模型讲解及IO多路复用详解

## 课程目标

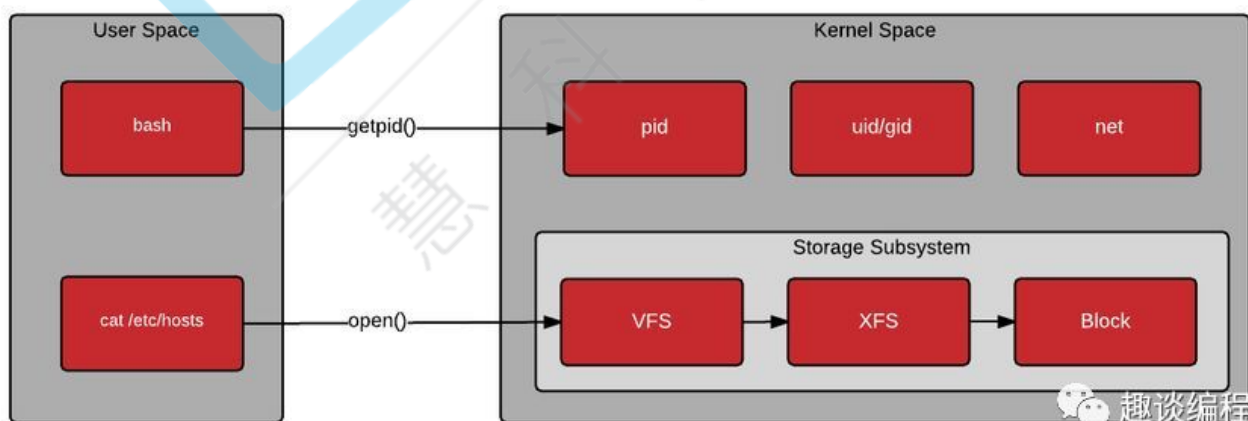
- 理解Linux中的用户空间和内核空间
- 理解内存与磁盘的PIO和DMA交互方式
- 理解缓存IO和直接IO的方式及区别
- 理解磁盘IO和网络IO的访问方式及区别
- 理解同步IO和异步IO的区别
- 理解堵塞IO和非堵塞IO的区别
- 回顾Socket网络编程中如何利用多线程提高并发能力
- 理解IO设计模式之Reactor和Proactor以及它们的区别
- 理解五种IO模型
- 理解Redis底层关于IO多路复用之epoll实现原理

## 用户空间和内核空间

学习 Linux 时，经常可以看到两个词：User space (用户空间) 和 Kernel space (内核空间)。

简单说，Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

虚拟内存被操作系统划分成两块：内核空间和用户空间，内核空间是内核代码运行的地方，用户空间是用户程序代码运行的地方。当进程运行在内核空间时就处于内核态，当进程运行在用户空间时就处于用户态。



Kernel space 可以执行任意命令，调用系统的一切资源；User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 `system call`），才能向内核发出指令。

通过系统接口，进程可以从用户空间切换到内核空间。

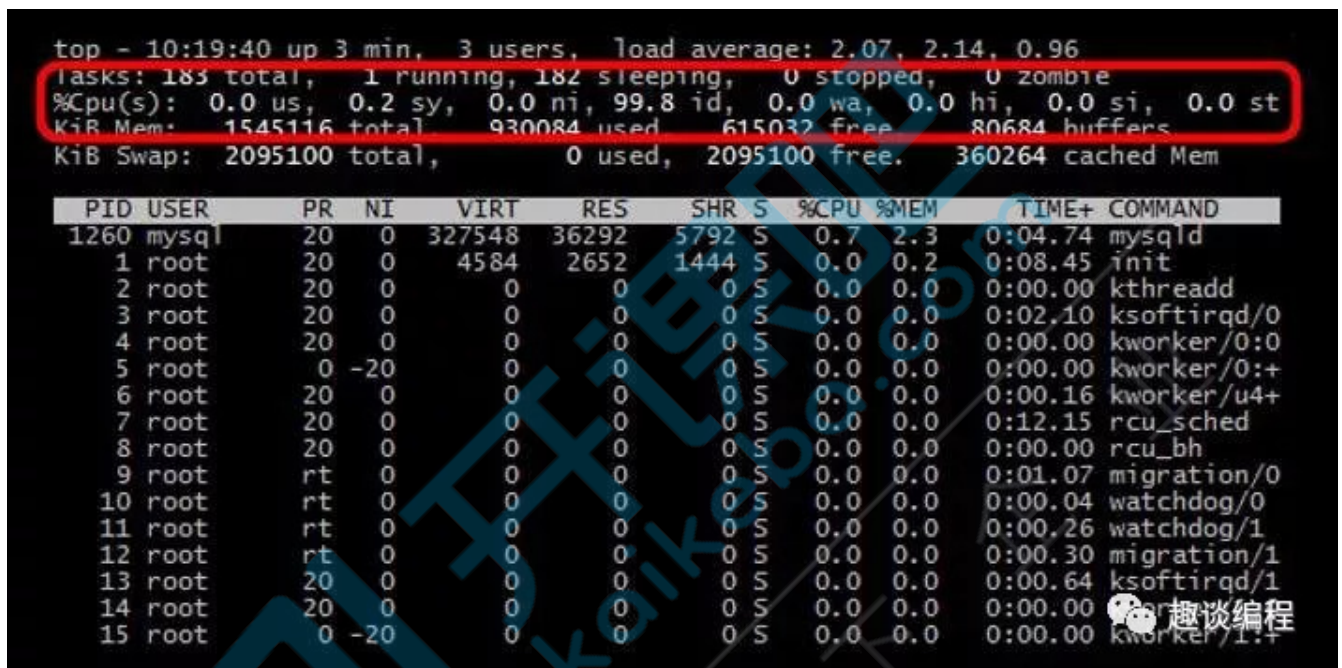
```

1 str = "my string" // 用户空间
2 x = x + 2
3 file.write(str) // 切换到内核空间
4 y = x + 4 // 切换回用户空间

```

上面代码中，第一行和第二行都是简单的赋值运算，在 User space 执行。第三行需要写入文件，就要切换到 Kernel space，因为用户不能直接写文件，必须通过内核安排。第四行又是赋值运算，就切换回 User space。

查看 CPU 时间在 User space 与 Kernel Space 之间的分配情况，可以使用top命令。它的第三行输出就是 CPU 时间分配统计。



The screenshot shows the output of the 'top' command. A red box highlights the third line: '%Cpu(s): 0.0 us, 0.2 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st'. Below this, a table lists system processes with columns for PID, USER, PR, NI, VIRT, RES, SHR, S, %CPU, %MEM, TIME+, and COMMAND.

| PID  | USER  | PR | NI  | VIRT   | RES   | SHR  | S | %CPU | %MEM | TIME+   | COMMAND     |
|------|-------|----|-----|--------|-------|------|---|------|------|---------|-------------|
| 1260 | mysql | 20 | 0   | 327548 | 36292 | 5792 | S | 0.7  | 2.3  | 0:04.74 | mysqld      |
| 1    | root  | 20 | 0   | 4584   | 2652  | 1444 | S | 0.0  | 0.2  | 0:08.45 | init        |
| 2    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | kthreadd    |
| 3    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:02.10 | ksoftirqd/0 |
| 4    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | kworker/0:0 |
| 5    | root  | 0  | -20 | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | kworker/0:+ |
| 6    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.16 | kworker/u4+ |
| 7    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:12.15 | rcu_sched   |
| 8    | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | rcu_bh      |
| 9    | root  | rt | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:01.07 | migration/0 |
| 10   | root  | rt | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.04 | watchdog/0  |
| 11   | root  | rt | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.26 | watchdog/1  |
| 12   | root  | rt | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.30 | migration/1 |
| 13   | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.64 | ksoftirqd/1 |
| 14   | root  | 20 | 0   | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | crond       |
| 15   | root  | 0  | -20 | 0      | 0     | 0    | S | 0.0  | 0.0  | 0:00.00 | kworker/1:+ |

这一行有 8 项统计指标。

```
%Cpu(s): 24.8 us, 0.5 sy, 0.0 ni, 73.6 id, 0.4 wa, 0.0 hi, 0.2 si, 0.0 st
```

其中，第一项24.8 us (user 的缩写) 就是 CPU 消耗在 User space 的时间百分比，第二项0.5 sy (system 的缩写) 是消耗在 Kernel space 的时间百分比。

随便也说一下其他 6 个指标的含义。

ni: niceness 的缩写，CPU 消耗在 nice 进程（低优先级）的时间百分比

id: idle 的缩写，CPU 消耗在闲置进程的时间百分比，这个值越低，表示 CPU 越忙

wa: wait 的缩写，CPU 等待外部 I/O 的时间百分比，这段时间 CPU 不能干其他事，但是也没有执行运算，这个值太高就说明外部设备有问题

hi: hardware interrupt 的缩写，CPU 响应硬件中断请求的时间百分比

si: software interrupt 的缩写，CPU 响应软件中断请求的时间百分比

st: stole time 的缩写，该项指标只对虚拟机有效，表示分配给当前虚拟机的 CPU 时间之中，被同一台物理机上的其他虚拟机偷走的时间百分比

# PIO与DMA

有必要简单地说说慢速I/O设备和内存之间的数据传输方式。

- PIO 我们拿磁盘来说，很早以前，磁盘和内存之间的数据传输是需要CPU控制的，也就是说如果我们读取磁盘文件到内存中，数据要经过CPU存储转发，这种方式称为PIO。显然这种方式非常不合理，需要占用大量的CPU时间来读取文件，造成文件访问时系统几乎停止响应。
- DMA 后来，DMA（直接内存访问，Direct Memory Access）取代了PIO，它可以不经过CPU而直接进行磁盘和内存（内核空间）的数据交换。在DMA模式下，CPU只需要向DMA控制器下达指令，让DMA控制器来处理数据的传送即可，DMA控制器通过系统总线来传输数据，传送完毕再通知CPU，这样就在很大程度上降低了CPU占有率，大大节省了系统资源，而它的传输速度与PIO的差异其实并不十分明显，因为这主要取决于慢速设备的速度。

可以肯定的是，PIO模式的计算机我们现在已经很少见到了。

## 缓存I/O和直接I/O

- 1 缓存I/O：数据从磁盘先通过DMA copy到内核空间，再从内核空间通过cpu copy到用户空间
- 2 直接I/O：数据从磁盘通过DMA copy到用户空间

### 缓存I/O

缓存I/O又被称作标准I/O，大多数文件系统的默认I/O操作都是缓存I/O。在Linux的缓存I/O机制中，数据先从磁盘复制到内核空间的缓冲区，然后从内核空间缓冲区复制到应用程序的地址空间。

- 读操作：  
操作系统检查内核的缓冲区有没有需要的数据，如果已经缓存了，那么就直接从缓存中返回；否则从磁盘中读取，然后缓存在操作系统的缓存中。
- 写操作：  
将数据从用户空间复制到内核空间的缓存中。这时对用户程序来说写操作就已经完成，至于什么时候再写到磁盘中由操作系统决定，除非显示地调用了sync同步命令（详情参考《[【珍藏】linux 同步IO: sync、fsync与fdatasync](#)》）。
- 缓存I/O的优点：
  - 1）在一定程度上分离了内核空间和用户空间，保护系统本身的运行安全；
  - 2）可以减少读盘的次数，从而提高性能。
- 缓存I/O的缺点：

在缓存 I/O 机制中，DMA 方式可以将数据直接从磁盘读到页缓存中，或者将数据从页缓存直接写回到磁盘上，而不能直接在应用程序地址空间和磁盘之间进行数据传输，这样，数据在传输过程中需要在应用程序地址空间（用户空间）和缓存（内核空间）之间进行多次数据拷贝操作，这些数据拷贝操作所带来的CPU以及内存开销是非常大的。

## 直接IO

直接IO就是应用程序直接访问磁盘数据，而不经过程序内核缓冲区，也就是绕过内核缓冲区，自己管理I/O缓存区，这样做的目的是减少一次从内核缓冲区到用户程序缓存的数据复制。

引入内核缓冲区的目的在于提高磁盘文件的访问性能，因为当进程需要读取磁盘文件时，如果文件内容已经在内核缓冲区中，那么就不需要再次访问磁盘；而当进程需要向文件中写入数据时，实际上只是写到了内核缓冲区便告诉进程已经写成功，而真正写入磁盘是通过一定的策略进行延迟的。

然而，对于一些较复杂的应用，比如数据库服务器，它们为了充分提高性能，希望绕过内核缓冲区，由自己在用户态空间实现并管理I/O缓冲区，包括缓存机制和写延迟机制等，以支持独特的查询机制，比如数据库可以根据更加合理的策略来提高查询缓存命中率。另一方面，绕过内核缓冲区也可以减少系统内存的开销，因为内核缓冲区本身就在系统内存中。

应用程序直接访问磁盘数据，不经过操作系统内核数据缓冲区，这样做的目的是减少一次从内核缓冲区到用户程序缓存的数据复制。这种方式通常是在对数据的缓存管理由应用程序实现的数据库管理系统中。

直接I/O的缺点就是如果访问的数据不在应用程序缓存中，那么每次数据都会直接从磁盘进行加载，这种直接加载会非常缓慢。通常直接I/O跟异步I/O结合使用会得到较好的性能。

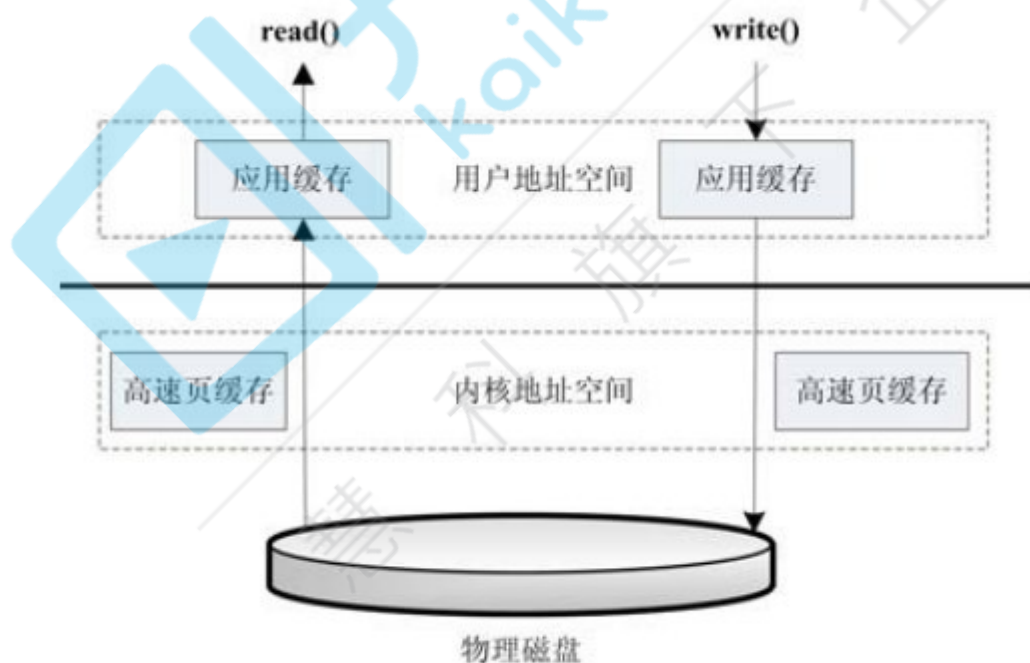
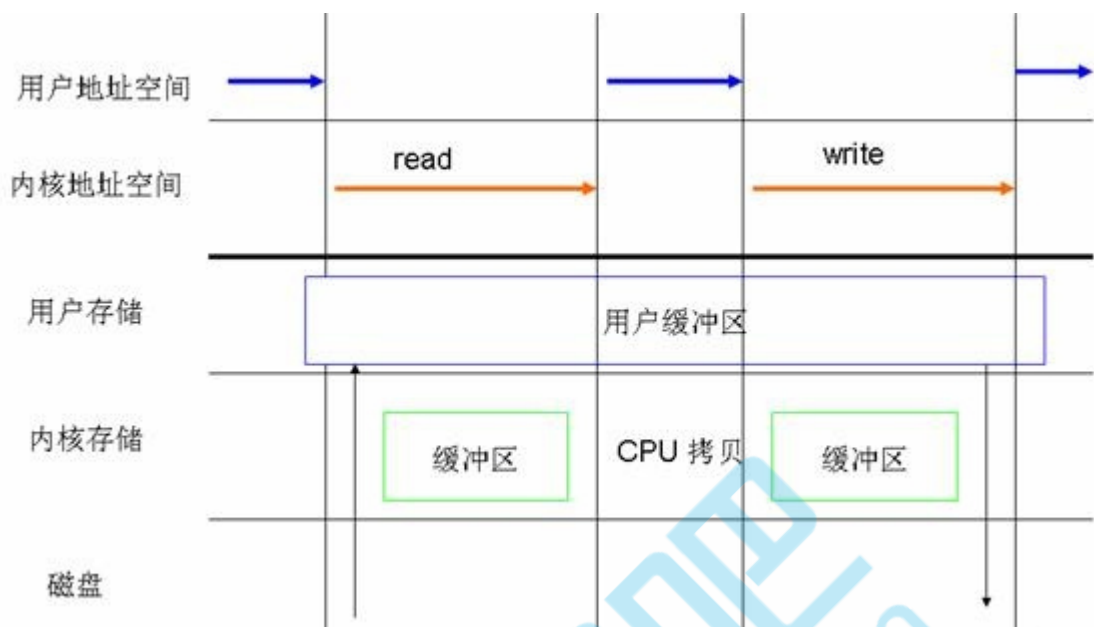


图 2-8 直接 I/O 方式

访问步骤：



Linux提供了对这种需求的支持，即在`open()`系统调用中增加参数选项`O_DIRECT`，用它打开的文件便可以绕过内核缓冲区的直接访问，这样便有效避免了CPU和内存的多余时间开销。

顺便提一下，与`O_DIRECT`类似的一个选项是`O_SYNC`，后者只对写数据有效，它将写入内核缓冲区的数据立即写入磁盘，将机器故障时数据的丢失减少到最小，但是它仍然要经过内核缓冲区。

## IO访问方式

### 磁盘IO

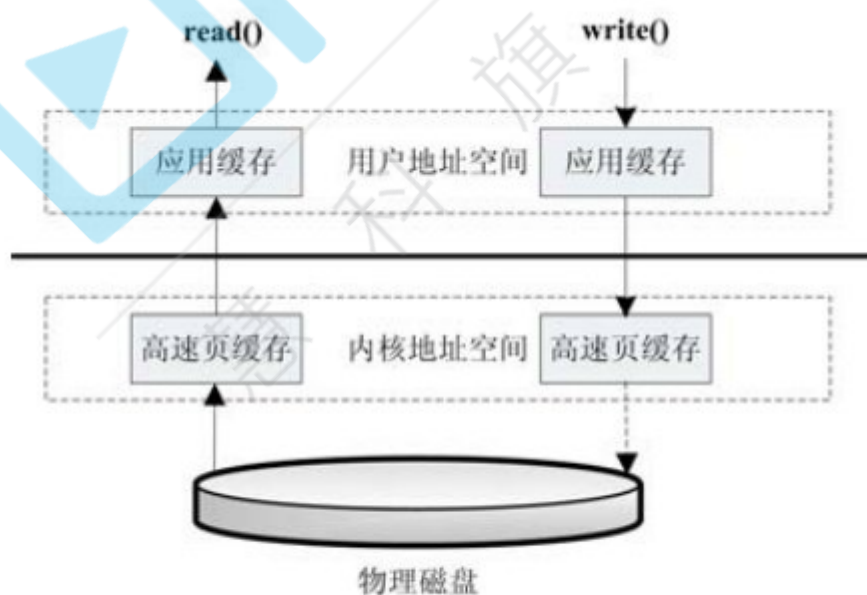


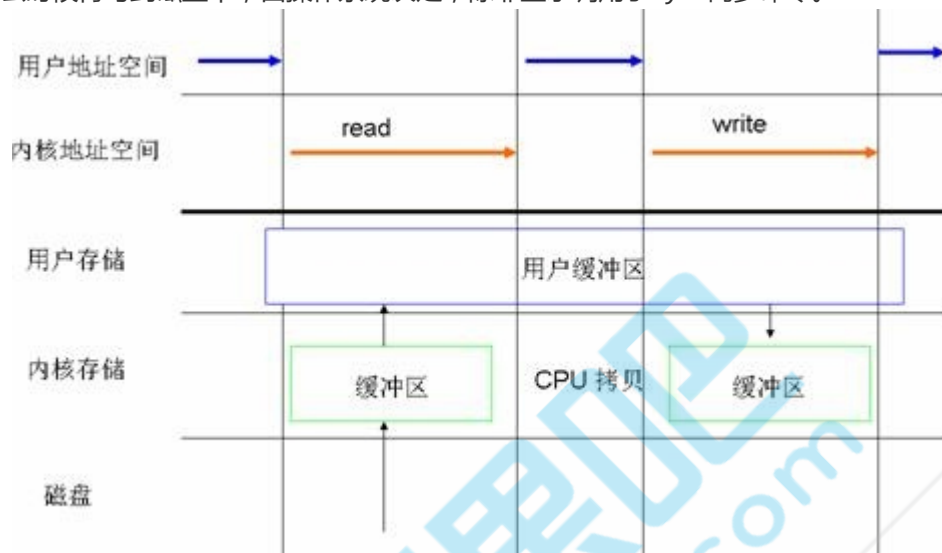
图 2-7 标准访问文件方式

具体步骤：



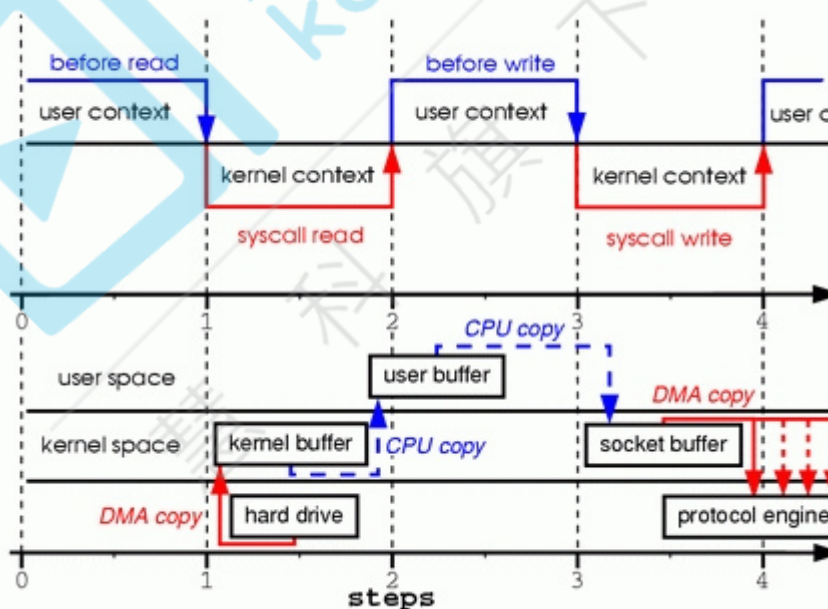
当应用程序调用read接口时，操作系统检查在内核的高速缓存有没有需要的数据，如果已经缓存了，那么就直接从缓存中返回，如果没有，则从磁盘中读取，然后缓存在操作系统的缓存中。

应用程序调用write接口时，将数据从用户地址空间复制到内核地址空间的缓存中，这时对用户程序来说，写操作已经完成，至于什么时候再写到磁盘中，由操作系统决定，除非显示调用了sync同步命令。



## 网络IO

1) 操作系统将数据从磁盘复制到操作系统内核的页缓存中 2) 应用将数据从内核缓存复制到应用的缓存中 3) 应用将数据写回内核的Socket缓存中 4) 操作系统将数据从Socket缓存区复制到网卡缓存，然后将其通过网络发出



1、当调用read系统调用时，通过DMA ( Direct Memory Access ) 将数据copy到内核模式 2、然后由CPU控制将内核模式数据copy到用户模式下的 buffer中 3、read调用完成后，write调用首先将用户模式下 buffer中的数据copy到内核模式下的socket buffer中 4、最后通过DMA copy将内核模式下的socket buffer中的数据copy到网卡设备中传送。

从上面的过程可以看出，数据白白从内核模式到用户模式走了一圈，浪费了两次的copy，而这两次copy都是CPU copy，即占用CPU资源。

## 磁盘IO和网络IO对比

首先，磁盘IO主要的延时是由（以15000rpm硬盘为例）：机械转动延时（机械磁盘的主要性能瓶颈，平均为2ms）+ 寻址延时（2~3ms）+ 块传输延时（一般4k每块，40m/s的传输速度，延时一般为0.1ms）决定。（平均为5ms）

而网络IO主要延是由：服务器响应延时 + 带宽限制 + 网络延时 + 跳转路由延时 + 本地接收延时 决定。（一般为几十到几千毫秒，受环境干扰极大）

所以两者一般来说网络IO延时要大于磁盘IO的延时。

## Socket网络编程

### 客户端

```
1 public class SocketClient {
2     public static void main(String args[]) throws Exception {
3         // 要连接的服务端IP地址和端口
4         String host = "127.0.0.1";
5         int port = 55533;
6         // 与服务端建立连接
7         Socket socket = new Socket(host, port);
8         // 建立连接后获得输出流
9         OutputStream outputStream = socket.getOutputStream();
10        String message="你好 yiwangzhibujian";
11        socket.getOutputStream().write(message.getBytes("UTF-8"));
12        outputStream.close();
13        socket.close();
14    }
15 }
```

### 服务端

```
1 public class SocketServer {
2     public static void main(String[] args) throws Exception {
3         // 监听指定的端口
4         int port = 55533;
5         ServerSocket server = new ServerSocket(port);
6
7         // server将一直等待连接的到来
8         System.out.println("server将一直等待连接的到来");
9         Socket socket = server.accept();
10        // 建立好连接后，从socket中获取输入流，并建立缓冲区进行读取
11        InputStream inputStream = socket.getInputStream();
12        byte[] bytes = new byte[1024];
13        int len;
```

```

14     StringBuilder sb = new StringBuilder();
15     while ((len = inputStream.read(bytes)) != -1) {
16         //注意指定编码格式，发送方和接收方一定要统一，建议使用UTF-8
17         sb.append(new String(bytes, 0, len, "UTF-8"));
18     }
19     System.out.println("get message from client: " + sb);
20     inputStream.close();
21     socket.close();
22     server.close();
23 }
24 }

```

```

1 public class SocketServer {
2     public static void main(String args[]) throws IOException {
3         // 监听指定的端口
4         int port = 55533;
5         ServerSocket server = new ServerSocket(port);
6         // server将一直等待连接的到来
7         System.out.println("server将一直等待连接的到来");
8
9         while(true){
10             Socket socket = server.accept();
11             // 建立好连接后，从socket中获取输入流，并建立缓冲区进行读取
12             InputStream inputStream = socket.getInputStream();
13             byte[] bytes = new byte[1024];
14             int len;
15             StringBuilder sb = new StringBuilder();
16             while ((len = inputStream.read(bytes)) != -1) {
17                 // 注意指定编码格式，发送方和接收方一定要统一，建议使用UTF-8
18                 sb.append(new String(bytes, 0, len, "UTF-8"));
19             }
20             System.out.println("get message from client: " + sb);
21             inputStream.close();
22             socket.close();
23         }
24     }
25 }

```

```

1 public class SocketServer {
2     public static void main(String args[]) throws Exception {
3         // 监听指定的端口
4         int port = 55533;
5         ServerSocket server = new ServerSocket(port);
6         // server将一直等待连接的到来
7         System.out.println("server将一直等待连接的到来");
8
9         //如果使用多线程，那就需要线程池，防止并发过高时创建过多线程耗尽资源
10        ExecutorService threadPool = Executors.newFixedThreadPool(100);
11
12        while (true) {

```



```

13     Socket socket = server.accept();
14
15     Runnable runnable = () -> {
16         try {
17             // 建立好连接后，从socket中获取输入流，并建立缓冲区进行读取
18             InputStream inputStream = socket.getInputStream();
19             byte[] bytes = new byte[1024];
20             int len;
21             StringBuilder sb = new StringBuilder();
22             while ((len = inputStream.read(bytes)) != -1) {
23                 // 注意指定编码格式，发送方和接收方一定要统一，建议使用UTF-8
24                 sb.append(new String(bytes, 0, len, "UTF-8"));
25             }
26             System.out.println("get message from client: " + sb);
27             inputStream.close();
28             socket.close();
29         } catch (Exception e) {
30             e.printStackTrace();
31         }
32     };
33     threadPool.submit(runnable);
34 }
35 }
36 }

```

## 同步IO和异步IO

同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。

- 1 指的是用户空间和内核空间数据交互的方式
- 2 同步：用户空间要的数据，必须等到内核空间给它才做其他事情
- 3 异步：用户空间要的数据，不需要等到内核空间给它，才做其他事情。内核空间会异步通知用户进程，并把数据直接给到用户空间。

## 阻塞IO和非阻塞IO

阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

- 1 指的是用户就和内核空间IO操作的方式
- 2 堵塞：用户空间通过系统调用（systemcall）和内核空间发送IO操作时，该调用是堵塞的
- 3 非堵塞：用户空间通过系统调用（systemcall）和内核空间发送IO操作时，该调用是不堵塞的，直接返回的，只是返回时，可能没有数据而已

# IO设计模式之Reactor和Proactor

平时接触的开源产品如Redis、ACE，事件模型都使用的Reactor模式；而同样做事件处理的Proactor，由于操作系统的原因，相关的开源产品也少；这里学习下其模型结构，重点对比下两者的异同点；

## 反应器Reactor

### 概述

反应器设计模式(Reactor pattern)是一种为处理并发服务请求，并将请求提交到一个或者多个服务处理程序的事件设计模式。当客户端请求抵达后，服务处理程序使用多路分配策略，由一个非阻塞的线程来接收所有的请求，然后派发这些请求至相关的工作线程进行处理。

Reactor模式主要包含下面几部分内容：

- **初始事件分发器(Initialization Dispatcher)**：用于管理Event Handler，定义注册、移除EventHandler等。它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法
- **同步（多路）事件分离器(Synchronous Event Demultiplexer)**：无限循环等待新事件的到来，一旦发现有新的事件到来，就会通知初始事件分发器去调取特定的事件处理器。
- **系统处理程序(Handles)**：操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接（Connection，在Java NIO中的Channel）。这个Channel注册到Synchronous Event Demultiplexer中，以监听Handle中发生的事件，对ServerSocketChannel可以是CONNECT事件，对SocketChannel可以是READ、WRITE、CLOSE事件等。
- **事件处理器(Event Handler)**：定义事件处理方法，以供Initialization Dispatcher回调使用。

对于Reactor模式，可以将其看做由两部分组成，一部分是由Boss组成，另一部分是由worker组成。Boss就像老板一样，主要是拉活儿、谈项目，一旦Boss接到活儿了，就下发给下面的work去处理。也可以看做是项目经理和程序员之间的关系。

### 为什么使用Reactor模式

并发系统常使用reactor模式代替常用的多线程的处理方式，节省系统的资源，提高系统的吞吐量。例如：在高并发的情况下，既可以使用多处理处理方式，也可以使用Reactor处理方式。

多线程的处理：

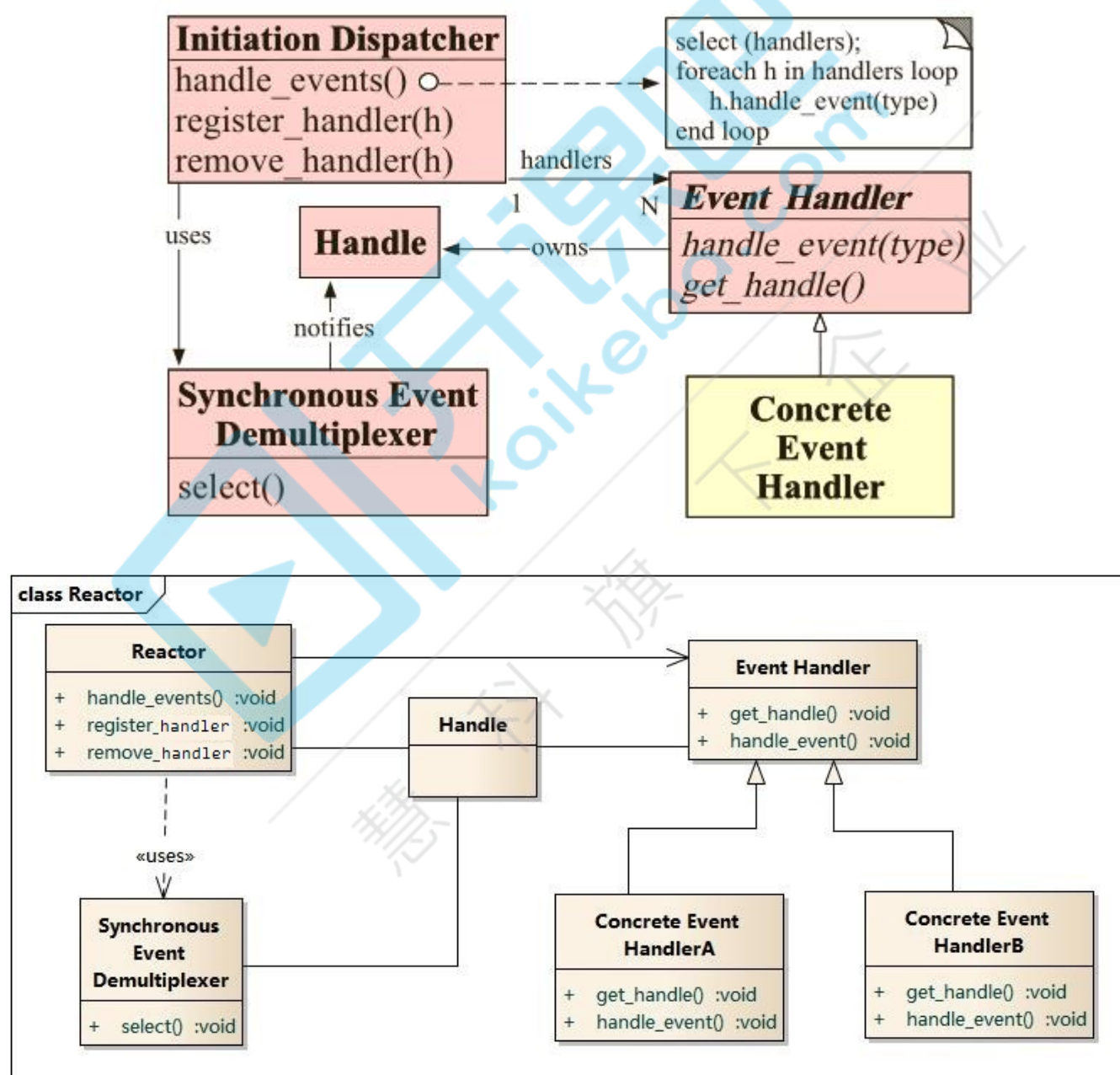
- 1 为每个单独到来的请求，专门启动一条线程，这样的话造成系统的开销很大，并且在单核的机上，多线程并不能提高系统的性能，除非在有一些阻塞的情况发生。否则线程切换的开销会使处理的速度变慢。

Reactor模式的处理：

- 1 服务器端启动一条单线程，用于轮询IO操作是否就绪，当有就绪的才进行相应的读写操作，这样的话就减少了服务器产生大量的线程，也不会出现线程之间的切换产生的性能消耗。（目前JAVA的NIO就采用的此种模式，这里引申出一个问题：在多核情况下NIO的扩展问题）

以上两种处理方式都是基于同步的，多线程的处理是我们传统模式下对高并发的处理方式，Reactor模式的处理是现今面对高并发和高性能一种主流的处理方式。

## Reactor模式结构

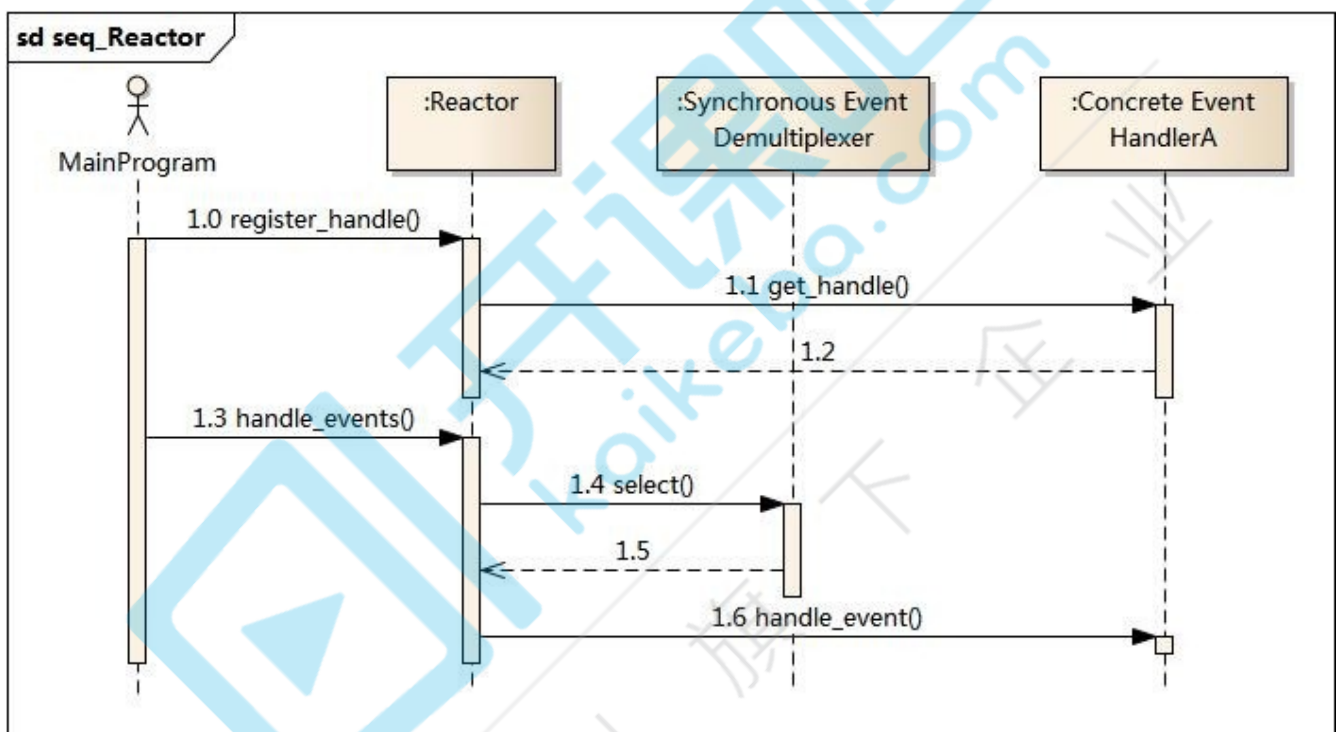


Reactor包含如下角色：

- Handle 句柄；用来标识socket连接或是打开文件；

- Synchronous Event Demultiplexer：同步事件多路分解器：由操作系统内核实现的一个函数；用于阻塞等待发生在句柄集合上的一个或多个事件；（如select/epoll；）
- Event Handler：事件处理接口
- Concrete Event HandlerA：实现应用程序所提供的特定事件处理逻辑；
- Reactor：反应器，定义一个接口，实现以下功能：1）供应用程序注册和删除关注的事件句柄；2）运行事件循环；3）有就绪事件到来时，分发事件到之前注册的回调函数上处理；
- Initiation Dispatcher：用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法。

## 业务流程及时序图



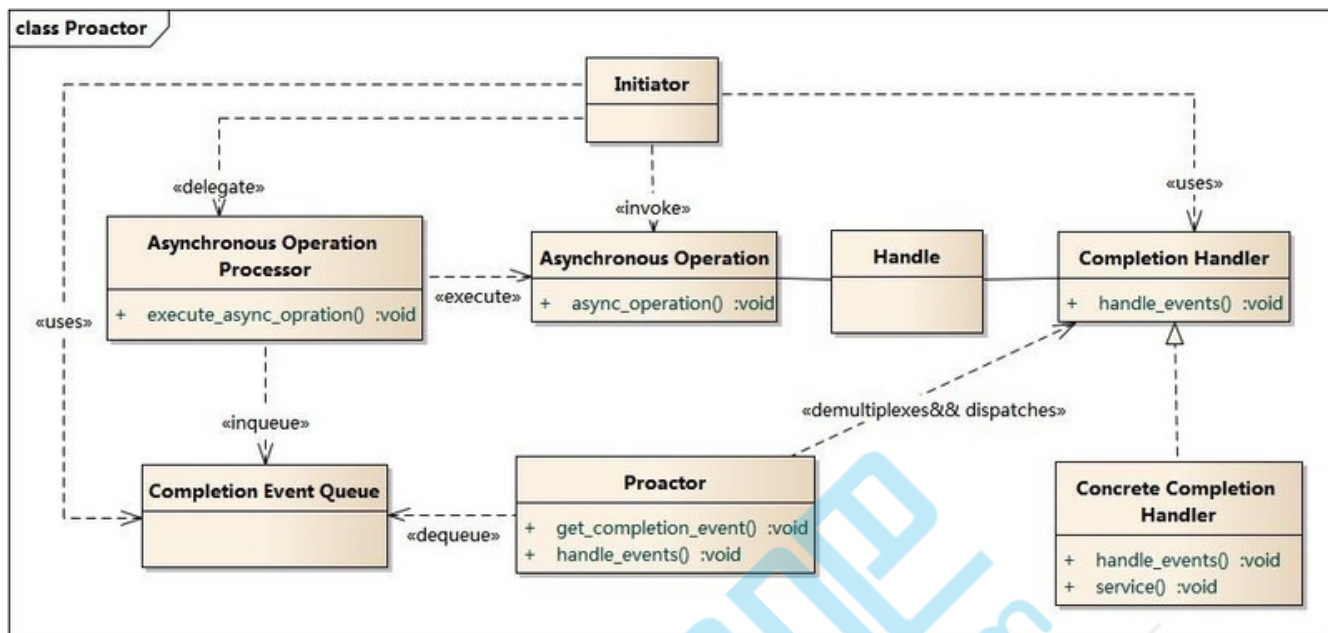
1. 应用启动，将关注的事件handle注册到Reactor中；
2. 调用Reactor，进入无限事件循环，等待注册的事件到来；
3. 事件到来，select返回，Reactor将事件分发到之前注册的回调函数中处理；

## Proactor模式

运用于异步I/O操作，Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的Io设备。

Proactor中写入操作和读取操作，只不过感兴趣的事件是写入完成事件。

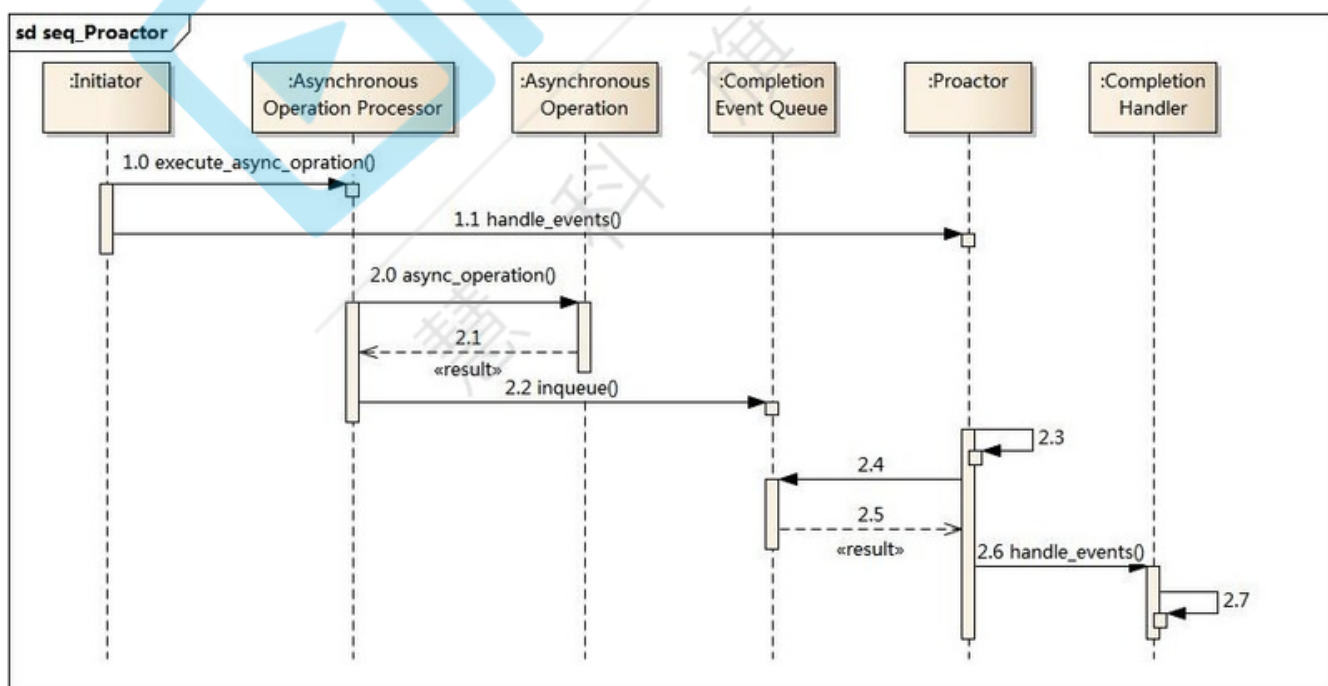
## Proactor模式结构



Proactor主动器模式包含如下角色

- **Handle** 句柄；用来标识socket连接或是打开文件；
- **Asynchronous Operation Processor**：异步操作处理器；负责执行异步操作，一般由操作系统内核实现；
- **Asynchronous Operation**：异步操作
- **Completion Event Queue**：完成事件队列；异步操作完成的结果放到队列中等待后续使用
- **Proactor**：主动器；为应用程序进程提供事件循环；从完成事件队列中取出异步操作的结果，分发调用相应的后续处理逻辑；
- **Completion Handler**：完成事件接口；一般是由回调函数组成的接口；
- **Concrete Completion Handler**：完成事件处理逻辑；实现接口定义特定的应用处理逻辑；

## 业务流程及时序图



1. 应用程序启动，调用异步操作处理器提供的异步操作接口函数，调用之后应用程序和异步操作处理就独立运行；应用程序可以调用新的异步操作，而其它操作可以并发进行；



2. 应用程序启动Proactor主动器，进行无限的事件循环，等待完成事件到来；
3. 异步操作处理器执行异步操作，完成后将结果放入到完成事件队列；
4. 主动器从完成事件队列中取出结果，分发到相应的完成事件回调函数处理逻辑中；

## 对比两者的区别

### 主动和被动

以主动写为例：

- Reactor将handle放到select()，等待可写就绪，然后调用write()写入数据；写完处理后续逻辑；
- Proactor调用aio\_write后立刻返回，由内核负责写操作，写完后调用相应的回调函数处理后续逻辑；

可以看出，Reactor被动的等待指示事件的到来并做出反应；它有一个等待的过程，做什么都要先放入到监听事件集合中等待handler可用时再进行操作；Proactor直接调用异步读写操作，调用完后立刻返回；

### 实现

Reactor实现了一个被动的事件分离和分发模型，服务等待请求事件的到来，再通过不受间断的同步处理事件，从而做出反应；

Proactor实现了一个主动的事件分离和分发模型；这种设计允许多个任务并发的执行，从而提高吞吐量；并可执行耗时长的任务（各个任务间互不影响）

### 优点

Reactor实现相对简单，对于耗时短的处理场景处理高效；操作系统可以在多个事件源上等待，并且避免了多线程编程相关的性能开销和编程复杂性；事件的串行化对应用是透明的，可以顺序的同步执行而不需要加锁；事务分离：将与应用无关的多路分解和分配机制和与应用相关的回调函数分离开来；

Proactor性能更高，能够处理耗时长的并发场景；

### 缺点

Reactor处理耗时长的操作会造成事件分发的阻塞，影响到后续事件的处理；

Proactor实现逻辑复杂；依赖操作系统对异步的支持，目前实现了纯异步操作的操作系统少，实现优秀的如windows IOCP，但由于其windows系统用于服务器的局限性，目前应用范围较小；而Unix/Linux系统对纯异步的支持有限，应用事件驱动的主流还是通过select/epoll来实现；

### 适用场景

Reactor：同时接收多个服务请求，并且依次同步的处理它们的事件驱动程序；Proactor：异步接收和同时处理多个服务请求的事件驱动程序；

## 漫谈五种IO模型

### 高性能IO模型浅析

服务器端编程经常需要构造高性能的IO模型，常见的IO模型有四种：

(1) 同步阻塞IO (Blocking IO)：即传统的IO模型。

(2) 同步非阻塞IO (Non-blocking IO)：默认创建的socket都是阻塞的，非阻塞IO要求socket被设置为NONBLOCK。注意这里所说的NIO并非Java的NIO (New IO) 库。

(3) IO多路复用 (IO Multiplexing)：即经典的Reactor设计模式，有时也称为异步阻塞IO，Java中的Selector和Linux中的epoll都是这种模型。

(4) 异步IO (Asynchronous IO)：即经典的Proactor设计模式，也称为异步非阻塞IO。

## IO模型举例理解1

1. 阻塞IO，给女神发一条短信，说我来找你了，然后就默默的一直等着女神下楼，这个期间除了等待你不会做其他事情，属于备胎做法。

1. 非阻塞IO，给女神发短信，如果不回，接着再发，一直发到女神下楼，这个期间你除了发短信等待不会做其他事情，属于专一做法。

1. IO多路复用，是找一个宿管大妈来帮你监视下楼的女生，这个期间你可以些其他的事情。例如可以顺便看看其他妹子，玩玩王者荣耀，上个厕所等等。IO复用又包括 select, poll, epoll 模式。那么它们的区别是什么？  
3.1 select大妈 每一个女生下楼，select大妈都不知道这个是不是你的女神，她需要一个一个询问，并且select大妈能力还有限，最多一次帮你监视1024个妹子  
3.2 poll大妈不限制盯着女生的数量，只要是经过宿舍楼门口的女生，都会帮你去问是不是你女神  
3.3 epoll大妈不限制盯着女生的数量，并且也不需要一个一个去问。那么如何做呢？epoll大妈会为每个进宿舍楼的女生脸上贴上一个大字条，上面写上女生自己的名字，只要女生下楼了，epoll大妈就知道这个是不是你女神了，然后大妈再通知你。

上面这些同步IO有一个共同点就是，当女神走出宿舍门口的时候，你已经站在宿舍门口等着女神的，此时你属于同步等待状态

接下来是异步IO的情况 你告诉女神我来了，然后你就去王者荣耀了，一直到女神下楼了，发现找不见你了，女神再给你打电话通知你，说我下楼了，你在哪呢？这时候你才来到宿舍门口。此时属于逆袭做法

## IO模型举例理解2

1. 阻塞I/O模型 老李去火车站买票，排队三天买到一张退票。 耗费：在车站吃喝拉撒睡 3天，其他事一件没干。

2. 非阻塞I/O模型 老李去火车站买票，隔12小时去火车站问有没有退票，三天后买到一张票。耗费：往返车站6次，路上6小时，其他时间做了好多事。

3. I/O复用模型  
1. select/poll 老李去火车站买票，委托黄牛，然后每隔6小时电话黄牛询问，黄牛三天内买到票，然后老李去火车站交钱领票。 耗费：往返车站2次，路上2小时，黄牛手续费100元，打电话17次  
2. epoll 老李去火车站买票，委托黄牛，黄牛买到后即通知老李去领，然后老李去火车站交钱领票。 耗费：往返车站2次，路上2小时，黄牛手续费100元，无需打电话

4. 信号驱动I/O模型 老李去火车站买票，给售票员留下电话，有票后，售票员电话通知老李，然后老李去火车站交钱领票。 耗费：往返车站2次，路上2小时，免黄牛费100元，无需打电话

5. 异步I/O模型 老李去火车站买票，给售票员留下电话，有票后，售票员电话通知老李并快递送票上门。 耗  
费：往返车站1次，路上1小时，免黄牛费100元，无需打电话

## 同步阻塞IO

同步阻塞IO模型是最简单的IO模型，用户线程在内核进行IO操作时被阻塞。

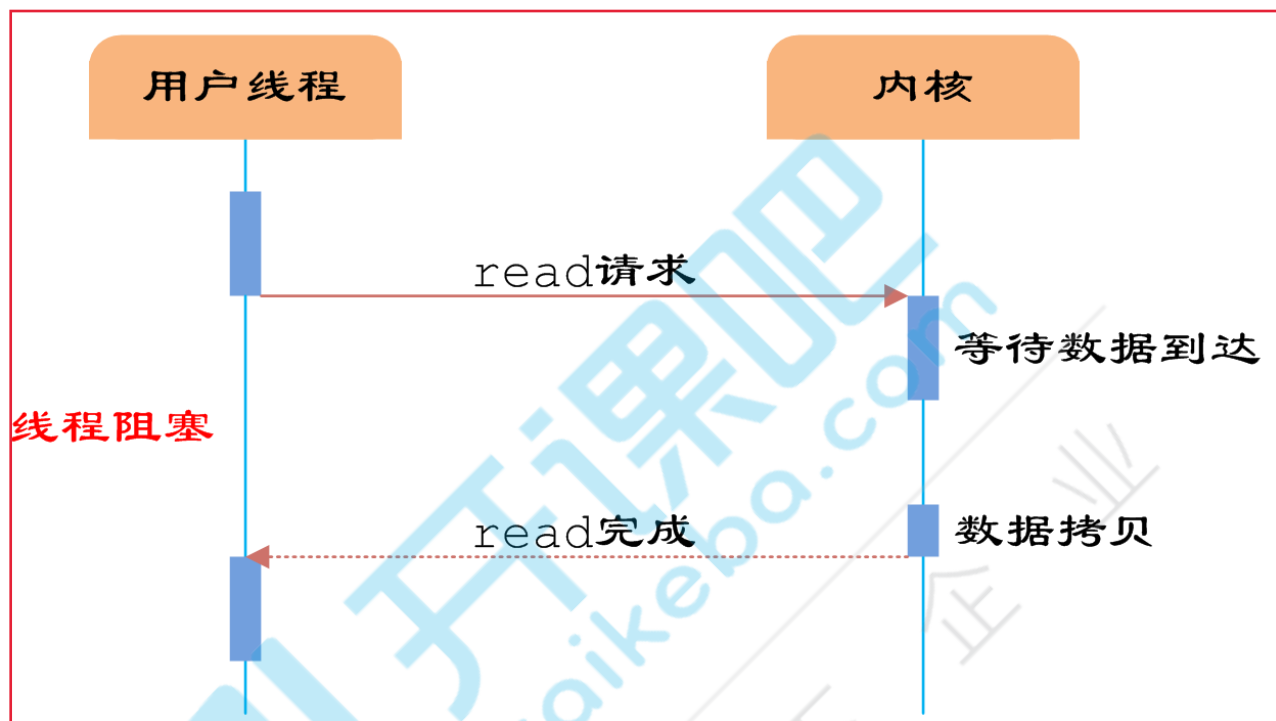


图1 同步阻塞IO

如图1所示，用户线程通过系统调用read发起IO读操作，由用户空间转到内核空间。内核等到数据包到达后，然后将接收的数据拷贝到用户空间，完成read操作。

用户线程使用同步阻塞IO模型的伪代码描述为：

```
1 {  
2  
3     read(socket, buffer);  
4  
5     process(buffer);  
6  
7 }
```

即用户需要等待read将socket中的数据读取到buffer后，才继续处理接收的数据。整个IO请求的过程中，用户线程是被阻塞的，这导致用户在发起IO请求时，不能做任何事情，对CPU的资源利用率不够。

## 同步非阻塞IO

同步非阻塞IO是在同步阻塞IO的基础上，将socket设置为NONBLOCK。这样做用户线程可以在发起IO请求后可以立即返回。

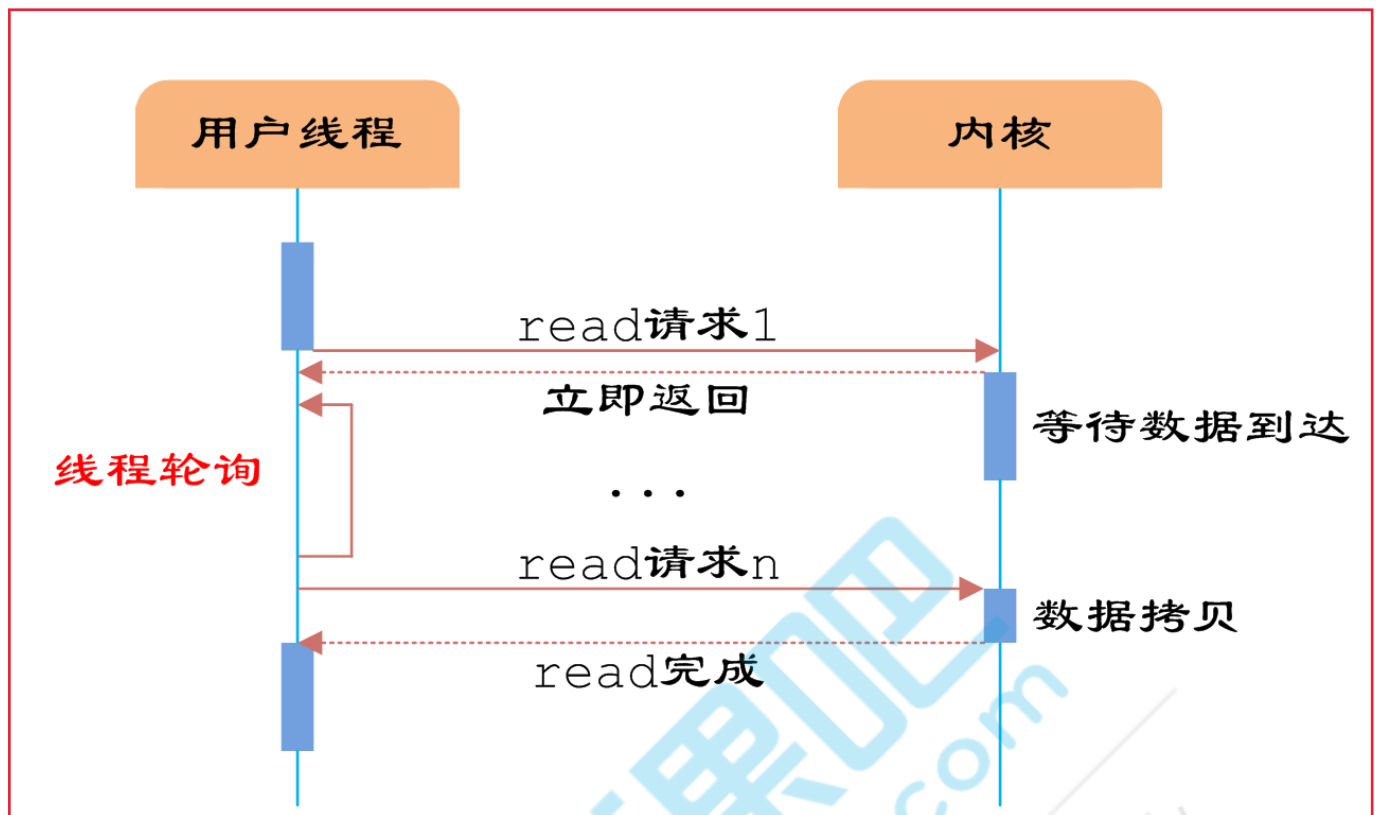


图2 同步非阻塞IO

如图2所示，由于socket是非阻塞的方式，因此用户线程发起IO请求时立即返回。但并未读取到任何数据，用户线程需要不断地发起IO请求，直到数据到达后，才真正读取到数据，继续执行。

用户线程使用同步非阻塞IO模型的伪代码描述为：

```
1 {  
2  
3     while(read(socket, buffer) != SUCCESS);  
4  
5     process(buffer);  
6 }
```

即用户需要不断地调用read，尝试读取socket中的数据，直到读取成功后，才继续处理接收的数据。整个IO请求的过程中，虽然用户线程每次发起IO请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的CPU的资源。一般很少直接使用这种模型，而是在其他IO模型中使用非阻塞IO这一特性。

## IO多路复用

IO多路复用模型是建立在内核提供的多路分离函数select基础之上的，使用select函数可以避免同步非阻塞IO模型中轮询等待的问题。

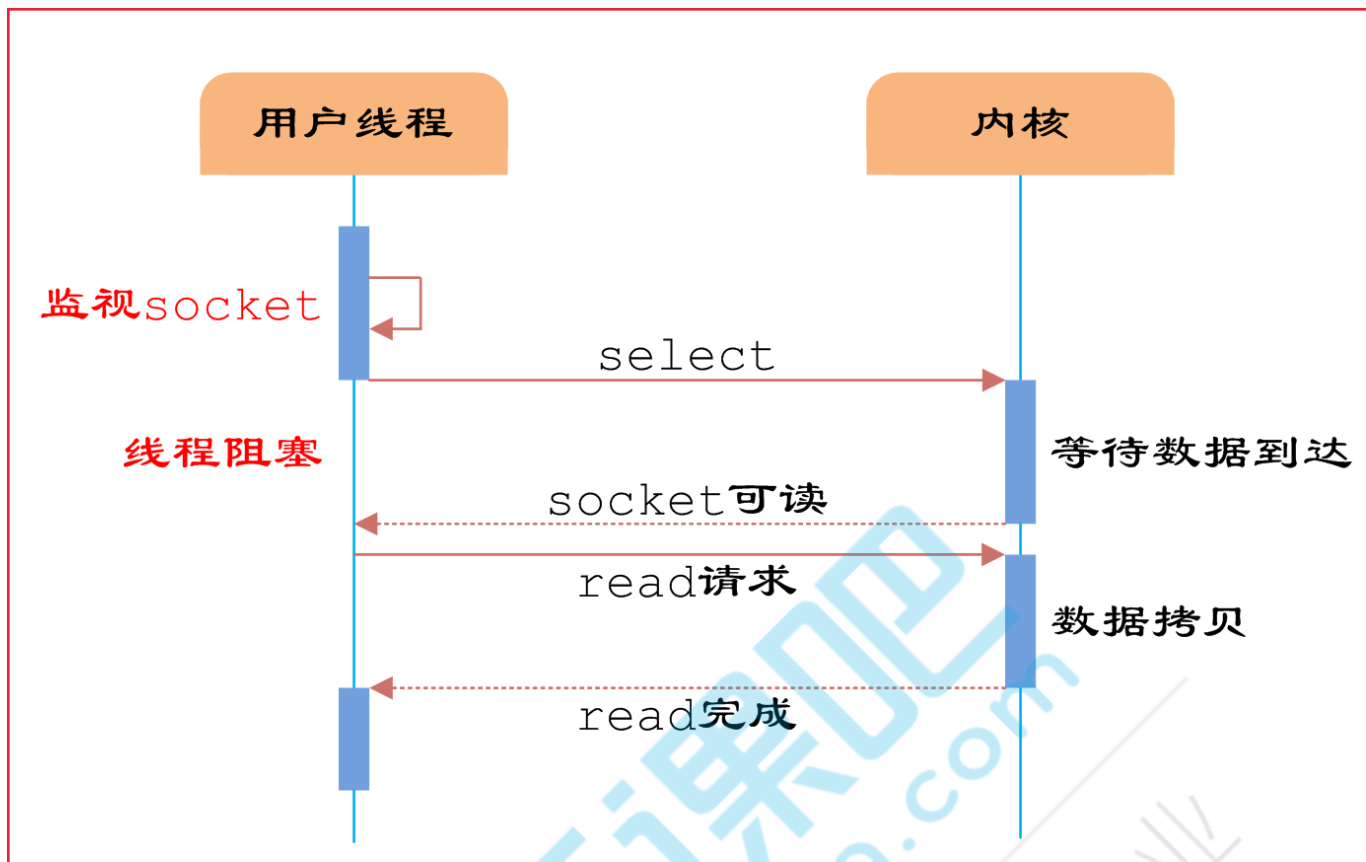


图3 多路分离函数select

如图3所示，用户首先将需要进行IO操作的socket添加到select中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。

从流程上来看，使用select函数进行IO请求和同步阻塞模型没有太大的区别，甚至还多了添加监视socket，以及调用select函数的额外操作，效率更差。但是，使用select以后最大的优势是用户可以在一个线程内同时处理多个socket的IO请求。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在同一个线程内同时处理多个IO请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

用户线程使用select函数的伪代码描述为：

```
1  {
2
3      select(socket);
4
5      while(1) {
6
7          sockets = select();
8
9          for(socket in sockets) {
10
11              if(can_read(socket)) {
12
13                  read(socket, buffer);
14
15                  process(buffer);
16
17              }
18
19          }
```



```

20
21     }
22
23 }

```

其中while循环前将socket添加到select监视中，然后在while内一直调用select获取被激活的socket，一旦socket可读，便调用read函数将socket中的数据读取出来。

然而，使用select函数的优点并不仅限于此。虽然上述方式允许单线程内处理多个IO请求，但是每个IO请求的过程还是阻塞的（在select函数上阻塞），平均时间甚至比同步阻塞IO模型还要长。**如果用户线程只注册自己感兴趣的socket或者IO请求，然后去做自己的事情，等到数据到来时再进行处理，则可以提高CPU的利用率。**

**IO多路复用模型使用了Reactor设计模式实现了这一机制。**

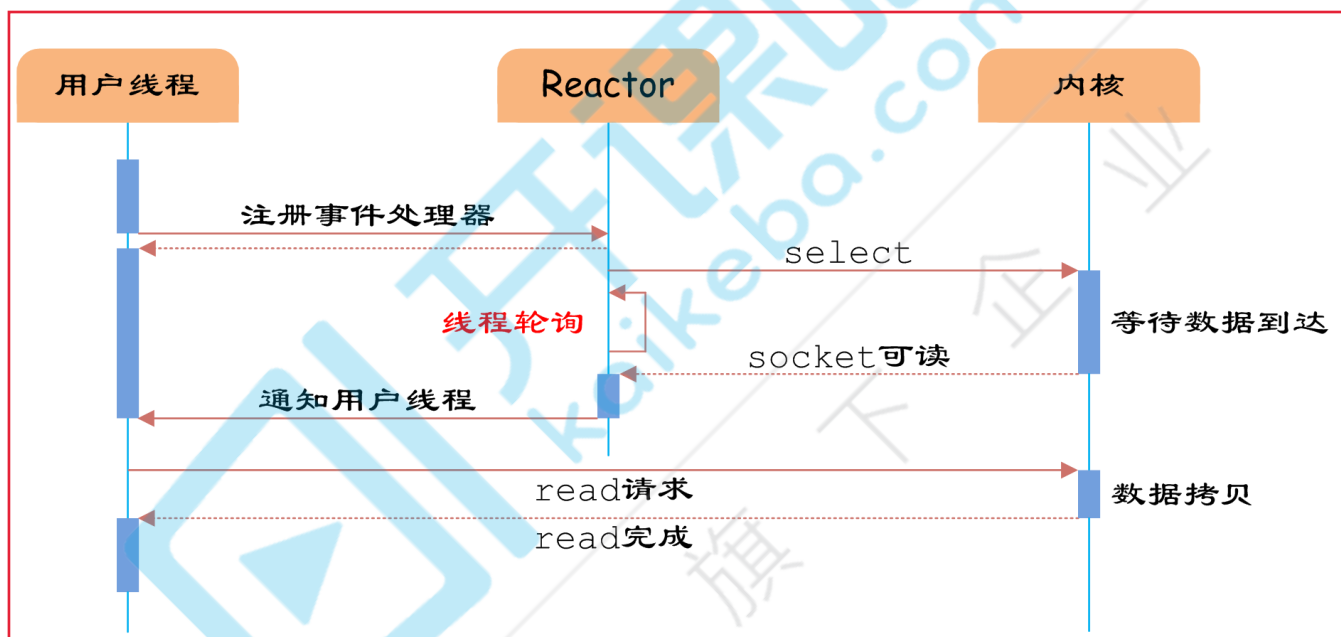


图5 IO多路复用

如图5所示，通过Reactor的方式，可以将用户线程轮询IO操作状态的工作统一交给handle\_events事件循环进行处理。用户线程注册事件处理器之后可以继续执行做其他的工作（异步），而Reactor线程负责调用内核的select函数检查socket状态。当有socket被激活时，则通知相应的用户线程（或执行用户线程的回调函数），执行handle\_event进行数据读取、处理的工作。由于select函数是阻塞的，因此多路IO复用模型也被称为异步阻塞IO模型。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。一般在使用IO多路复用模型时，socket都是设置为NONBLOCK的，不过这并不会产生影响，因为用户发起IO请求时，数据已经到达了，用户线程一定不会被阻塞。

用户线程使用IO多路复用模型的伪代码描述为：

```

1 void UserEventHandler::handle_event() {
2
3     if(can_read(socket)) {
4
5         read(socket, buffer);

```

```

6
7     process(buffer);
8
9 }
10
11 }
12
13 {
14
15     Reactor.register(new UserEventHandler(socket));
16
17 }

```

用户需要重写EventHandler的handle\_event函数进行读取数据、处理数据的工作，用户线程只需要将自己的EventHandler注册到Reactor即可。Reactor中handle\_events事件循环的伪代码大致如下。

```

1 Reactor::handle_events() {
2
3     while(1) {
4
5         sockets = select();
6
7         for(socket in sockets) {
8
9             get_event_handler(socket).handle_event();
10
11         }
12
13     }
14
15 }

```

事件循环不断地调用select获取被激活的socket，然后根据获取socket对应的EventHandler，执行器handle\_event函数即可。

IO多路复用是最常用的IO模型，但是其异步程度还不够“彻底”，因为它使用了会阻塞线程的select系统调用。因此IO多路复用只能称为异步阻塞IO，而非真正的异步IO。

## 异步IO

“真正”的异步IO需要操作系统更强的支持。在IO多路复用模型中，事件循环将文件句柄的状态事件通知给用户线程，由用户线程自行读取数据、处理数据。而在异步IO模型中，当用户线程收到通知时，数据已经被内核读取完毕，并放在了用户线程指定的缓冲区内，内核在IO完成后通知用户线程直接使用即可。

**异步IO模型使用了Proactor设计模式实现了这一机制。**

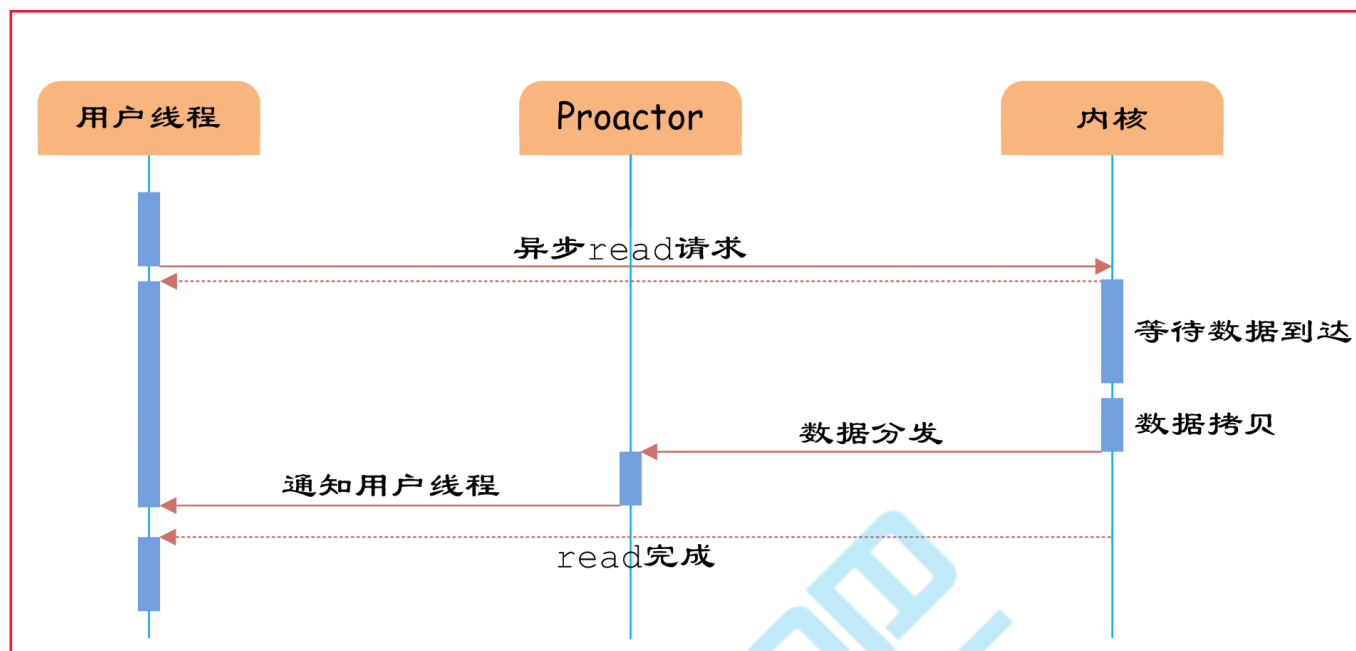


图7 异步IO

如图7所示，异步IO模型中，用户线程直接使用内核提供的异步IO API发起read请求，且发起后立即返回，继续执行用户线程代码。不过此时用户线程已经将调用的AsynchronousOperation和CompletionHandler注册到内核，然后操作系统开启独立的内核线程去处理IO操作。当read请求的数据到达时，由内核负责读取socket中的数据，并写入用户指定的缓冲区中。最后内核将read的数据和用户线程注册的CompletionHandler分发给内部Proactor，Proactor将IO完成的信息通知给用户线程（一般通过调用用户线程注册的完成事件处理函数），完成异步IO。

用户线程使用异步IO模型的伪代码描述为：

```
1 void UserCompletionHandler::handle_event(buffer) {
2     process(buffer);
3 }
4
5 {
6     aio_read(socket, new UserCompletionHandler);
7 }
8 }
```

用户需要重写CompletionHandler的handle\_event函数进行处理数据的工作，参数buffer表示Proactor已经准备好的数据，用户线程直接调用内核提供的异步IO API，并将重写的CompletionHandler注册即可。

相比于IO多路复用模型，异步IO并不十分常用，不少高性能并发服务程序使用IO多路复用模型+多线程任务处理的架构基本可以满足需求。况且目前操作系统对异步IO的支持并非特别完善，更多的是采用IO多路复用模型模拟异步IO的方式（IO事件触发时不直接通知用户线程，而是将数据读写完毕后放到用户指定的缓冲区中）。Java7之后已经支持了异步IO，感兴趣的读者可以尝试使用。

## Redis IO多路复用技术以及epoll实现原理

redis 是一个单线程却性能非常好的内存数据库，主要用来作为缓存系统。redis 采用网络IO多路复用技术来保证在多连接的时候，系统的高吞吐量。

### 为什么Redis中要使用I/O多路复用呢？

首先，Redis 是跑在单线程中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

select, poll, epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪，能够通知程序进行相应的操作。

redis的io模型主要是基于epoll实现的，不过它也提供了 select和kqueue的实现，默认采用epoll。

那么epoll到底是个什么东西呢？我们一起来看看

## epoll实现机制

设想一下如下场景：

有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的（事实上大部分场景都是这种情况）。如何实现这样的高并发？

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统（从用户态复制句柄数据结构到内核态），让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

如果没有I/O事件产生，我们的程序就会阻塞在select处。但是依然有个问题，我们从select那里仅仅知道了，有I/O事件发生了，但却并不知道是那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。

但是使用select，我们有O(n)的无差别轮询复杂度，同时处理的流越多，每一次无差别轮询时间就越长

总结：select和poll的缺点如下：

1. 每次调用select/poll，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大
2. 同时每次调用select/poll都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大
3. 针对select支持的文件描述符数量太小了，默认是1024
4. select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；
5. select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。
- 6.
7. 相比select模型，poll使用链表保存文件描述符，因此没有了监视文件数量的限制，但其他三个缺点依然存在。

epoll的设计和实现与select完全不同。epoll是poll的一种优化，返回后不需要对所有的fd进行遍历，在内核中维持了fd的列表。select和poll是将这个内核列表维持在用户态，然后传递到内核中。与poll/select不同，epoll不再是一个单独的系统调用，而是由epoll\_create/epoll\_ctl/epoll\_wait三个系统调用组成，后面将会看到这样做的好处。epoll在2.6以后的内核才支持。

epoll通过在Linux内核中申请一个简易的文件系统（文件系统一般用什么数据结构实现？B+树）。把原先的select/poll调用分成了3个部分：

- 1) 调用epoll\_create()建立一个epoll对象（在epoll文件系统中为这个句柄对象分配资源）

2) 调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

3) 调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

### 总结：epoll的优点

1. epoll 没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于 2048，\*\*一般来说这个数目和系统内存关系很大\*\*，具体数目可以 `cat /proc/sys/fs/file-max` 察看。
2. 效率提升，epoll 最大的优点就在于它\*\*只管你“活跃”的连接\*\*，而跟连接总数无关，因此在实际的网络环境中，epoll 的效率就会远远高于 select 和 poll。
3. 内存拷贝，epoll 在这点上使用了“共享内存”，这个内存拷贝也省略了。

## redis epoll底层实现

当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。

eventpoll结构体如下所示：

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root rbr;
    /*双链表中则存放着将通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

每一个epoll对象都有一个独立的事件poll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是 $\lg n$ ，其中 $n$ 为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：



```

struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head  rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}

```

当调用`epoll_wait`检查是否有事件发生时，只需要检查`eventpoll`对象中的`rdlist`双链表中是否有`epitem`元素即可。如果`rdlist`不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。

优势：

1. 不用重复传递。我们调用`epoll_wait`时就相当于以往调用`select/poll`，但是这时却不用传递`socket`句柄给内核，因为内核已经在`epoll_ctl`中拿到了要监控的句柄列表。
2. 在内核里，一切皆文件。所以，`epoll`向内核注册了一个文件系统，用于存储上述的被监控`socket`。当你调用`epoll_create`时，就会在这个虚拟的`epoll`文件系统里创建一个`file`结点。当然这个`file`不是普通文件，它只服务于`epoll`。

`epoll`在被内核初始化时（操作系统启动），同时会开辟出`epoll`自己的内核高速`cache`区，用于安置每一个我们想监控的`socket`，这些`socket`会以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。这个内核高速`cache`区，就是建立连续的物理内存页，然后在之上建立`slab`层，简单的说，就是物理上分配好你想要的`size`的内存对象，每次使用时都是使用空闲的已分配好的对象。

### 3. 极其高效的原因：

这是由于我们在调用`epoll_create`时，内核除了帮我们在`epoll`文件系统里建了个`file`结点，在内核`cache`里建了个红黑树用于存储以后`epoll_ctl`传来的`socket`外，还会再建立一个`list`链表，用于存储准备就绪的事件，当`epoll_wait`调用时，仅仅观察这个`list`链表里有没有数据即可。有数据就返回，没有数据就`sleep`，等到`timeout`时间到后即使链表没数据也返回。所以，`epoll_wait`非常高效。

这个准备就绪`list`链表是怎么维护的呢？

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。（注：好好理解这句话！）

从上面这句可以看出，`epoll`的基础就是回调呀！

如此，一颗红黑树，一张准备就绪句柄链表，少量的内核`cache`，就帮我们解决了大并发下的`socket`处理问题。执行

执行`epoll_create`时，创建了红黑树和就绪链表，执行`epoll_ctl`时，如果增加`socket`句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行`epoll_wait`时立刻返回准备就绪链表里的数据即可。

最后看看epoll独有的两种模式LT和ET。无论是LT和ET模式，都适用于以上所说的流程。区别是，LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时次次返回这个句柄，而ET模式仅在第一次返回。

关于LT，ET，有一端描述，LT和ET都是电子里面的术语，ET是边缘触发，LT是水平触发，一个表示只有在变化的边沿触发，一个表示在某个阶段都会触发。

LT，ET这件事怎么做到的呢？当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll\_wait每次都会返回这个句柄。（从上面这段，可以看出，LT还有个回放的过程，低效了）

