停划算

停我享停 · 划算出行

技术开发部内部分享

中智华清（北京）科技有限公司

停划算
停我享停 · 划算出行

用户空间 / 内核空间

- 操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间

进程切换

- 为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的，并且进程切换是非常耗费资源的

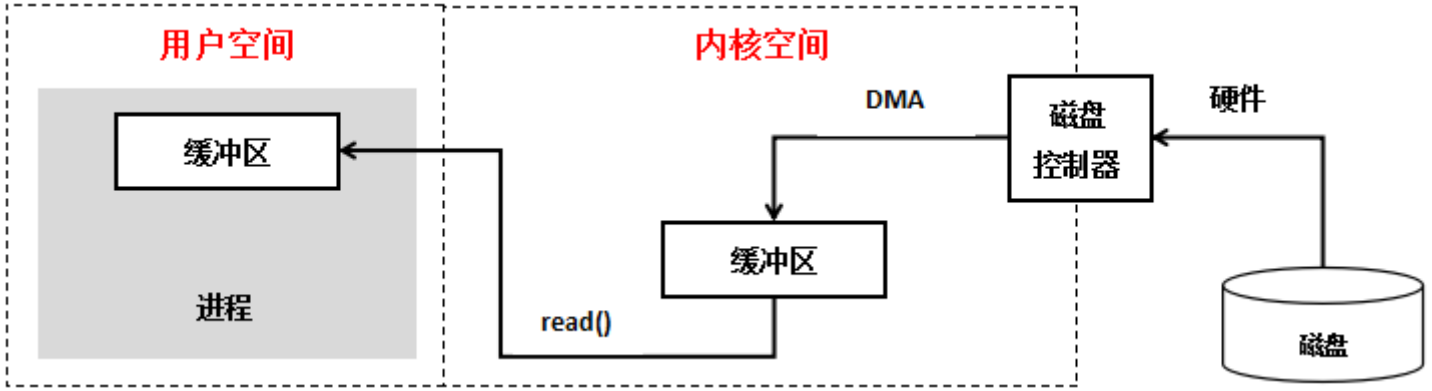
进程阻塞

- 正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态

文件描述符

- 指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统

linux系统IO分为内核准备数据和将数据从内核拷贝到用户空间两个阶段



缓存IO又称称为标准IO，大多数文件系统的默认IO操作都是缓存IO。在Linux的缓存IO机制中，操作系统会将IO的数据缓存在文件系统的页缓存（page cache）。也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓存区拷贝到应用程序的地址空间中。

这种做法的缺点就是，需要在应用程序地址空间和内核进行多次拷贝，这些拷贝动作所带来的CPU以及内存开销是非常大的

停划算
停我享停 · 划算出行

Richard Stevens 《Unix 网络编程》卷1 中描述的几种IO模型

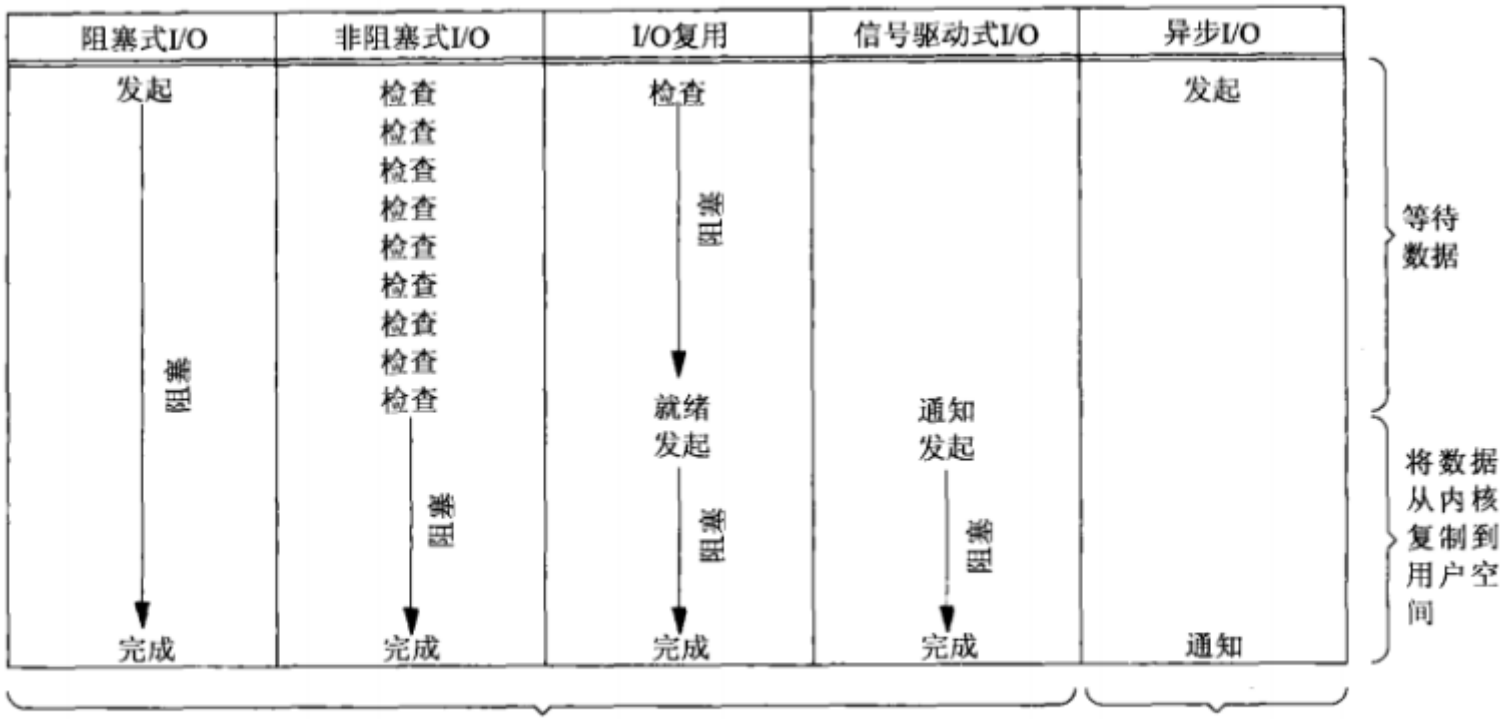


图6-6　5种I/O模型的比较

停划算
停我享停 · 划算出行

一个完整的IO读请求操作包括两个阶段:

1、查看数据是否就绪;
2、进行数据拷贝（内核将数据拷贝到用户线程）

区别

1、阻塞和非阻塞操作：是针对发起 IO 请求操作后，是否有立刻返回一个标志信息而不让请求线程等待。
2、同步和异步的区别：数据拷贝阶段是否需要完全由操作系统处理。同步IO向应用程序通知的是IO就绪事件，而异步IO向应用程序通知的是IO完成事件

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

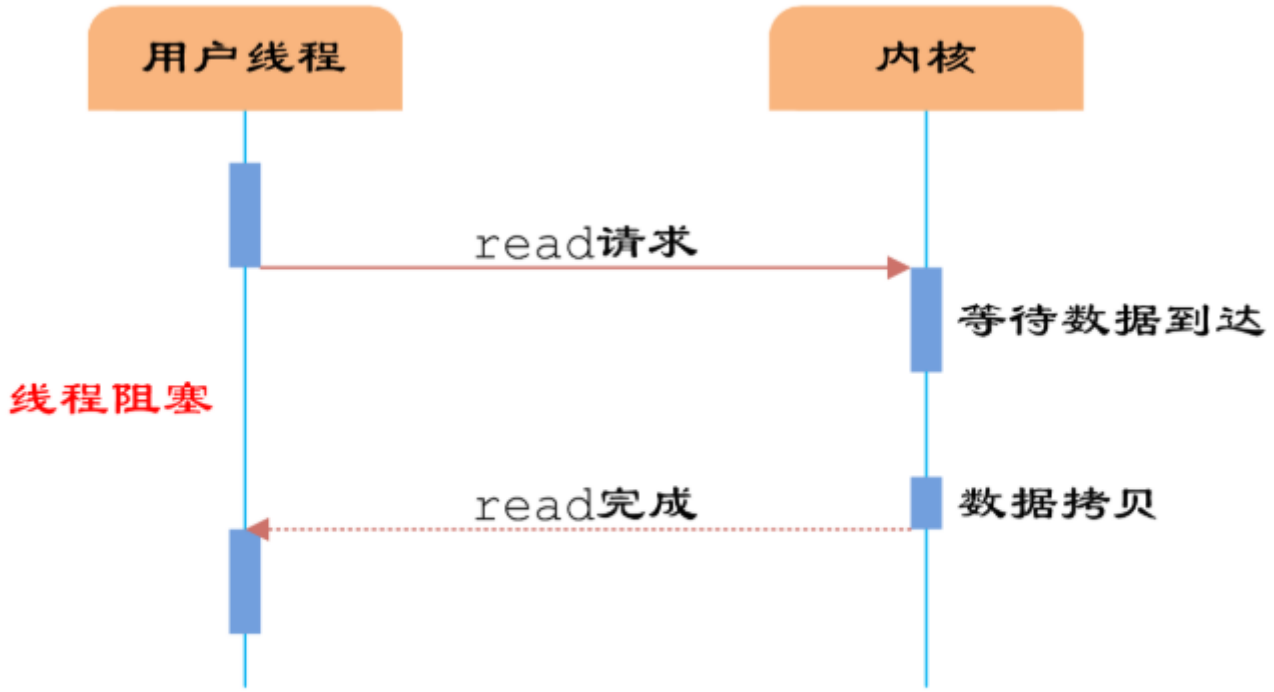


图1　同步阻塞I0

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

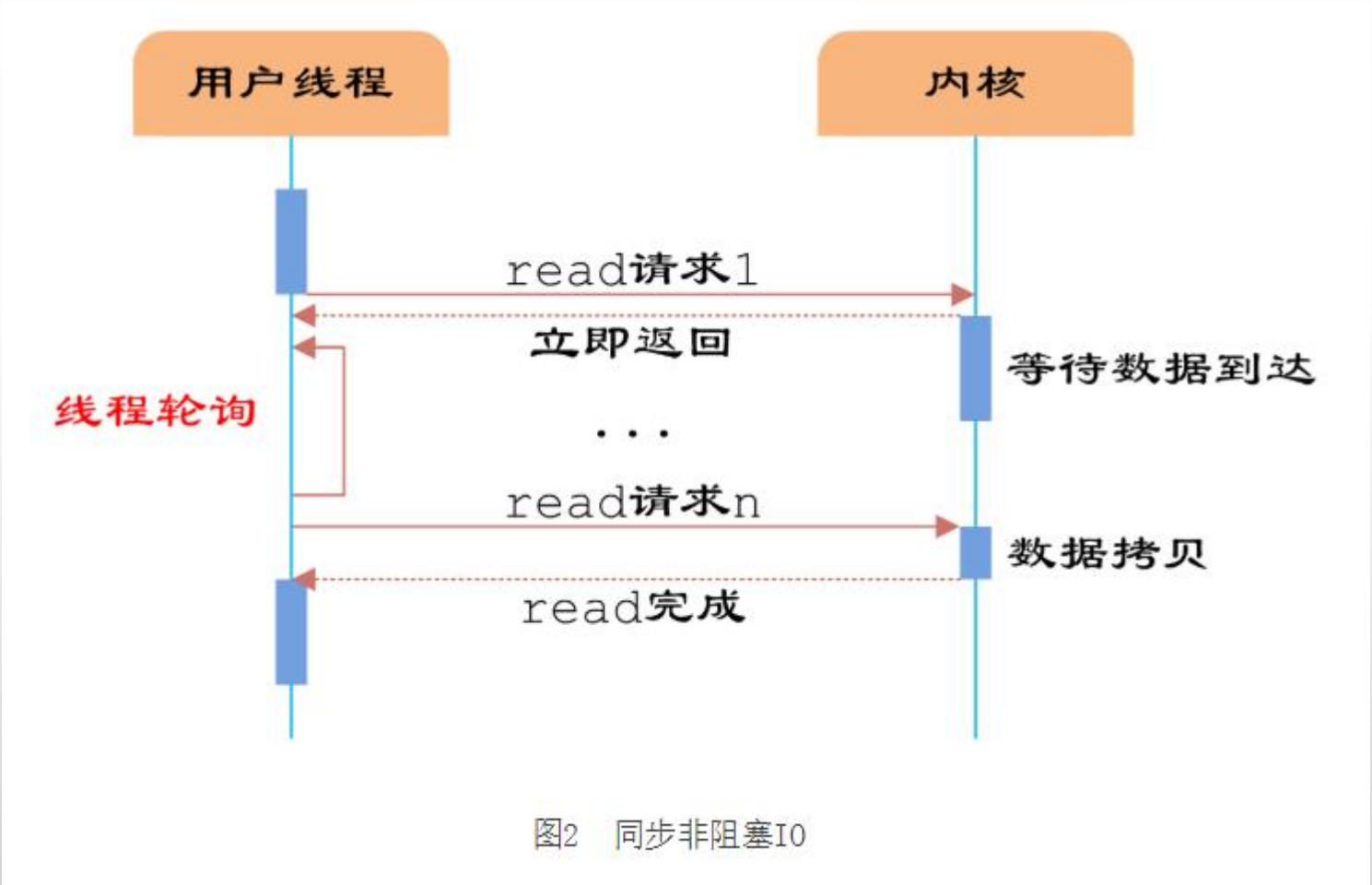


图2　同步非阻塞IO

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

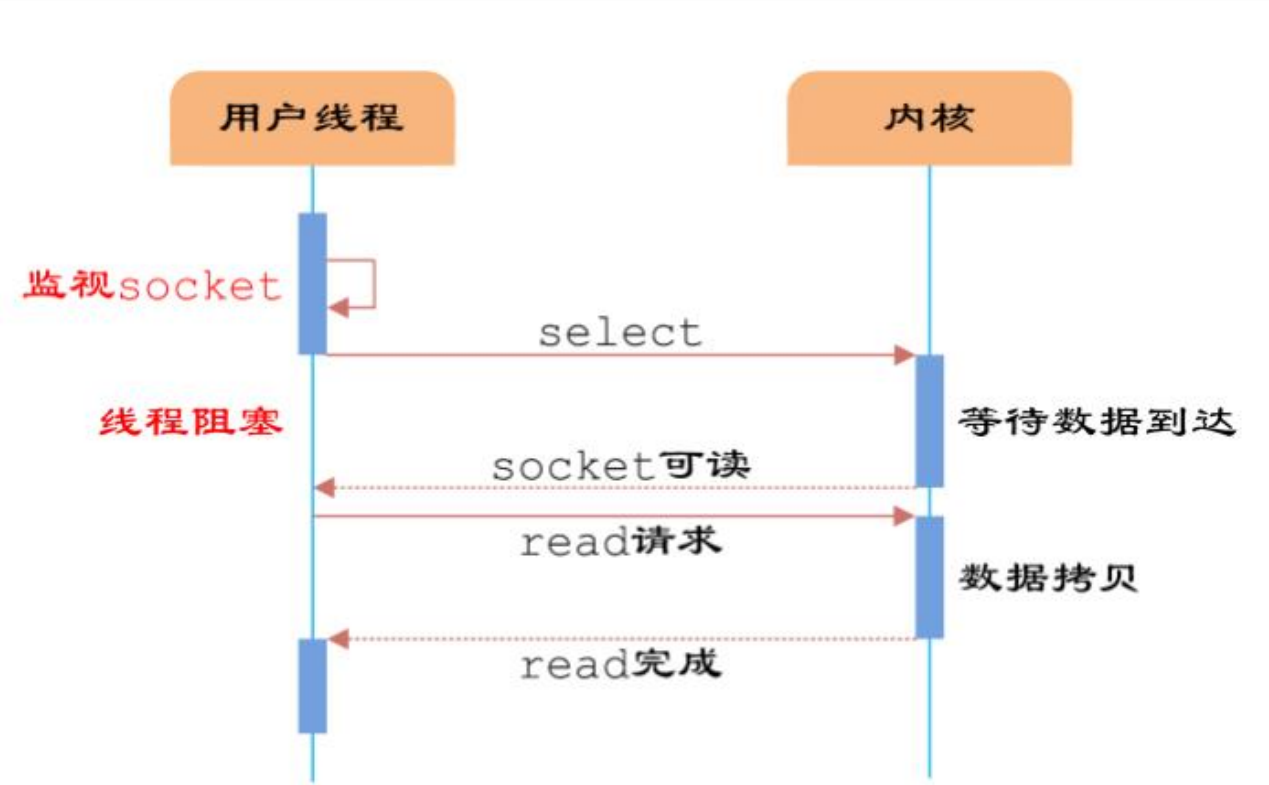
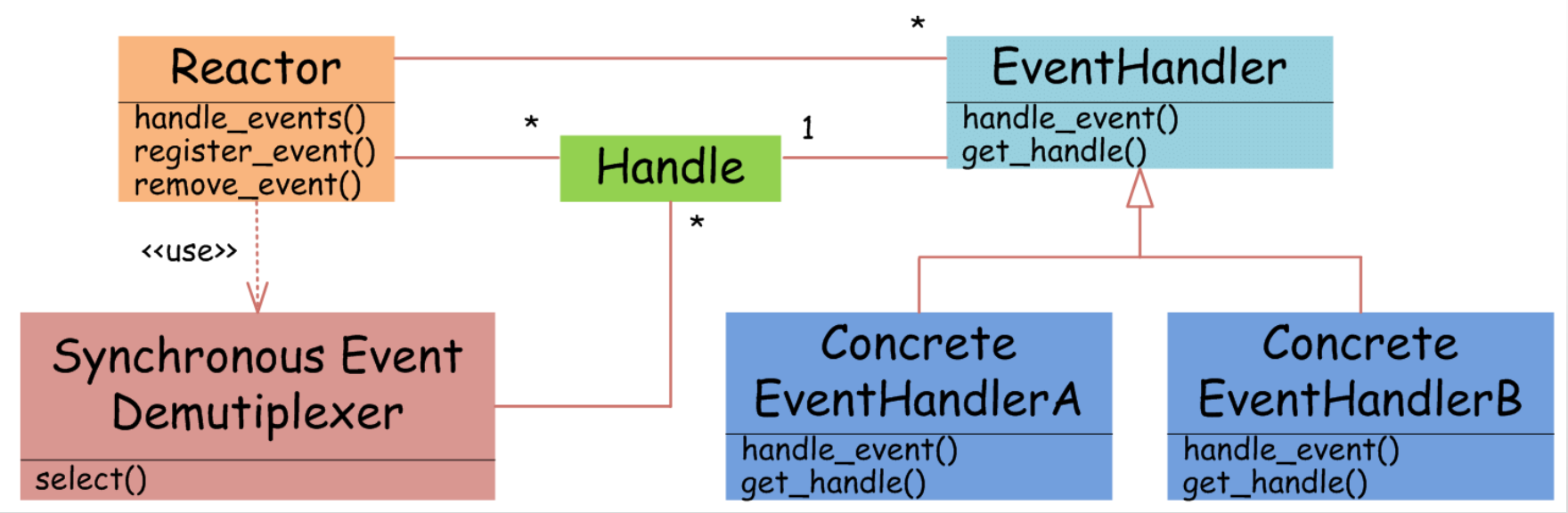


图3　多路分离函数select

Reactor 模型：基于事件驱动，实现多路复用的设计模式

模型中有三个重要的组件：
•多路复用器：由操作系统提供接口，Linux 提供的 I/O 复用接口有select、poll、epoll 。
•事件分离器：将多路复用器返回的就绪事件分发到事件处理器中。
•事件处理器：处理就绪事件处理函数

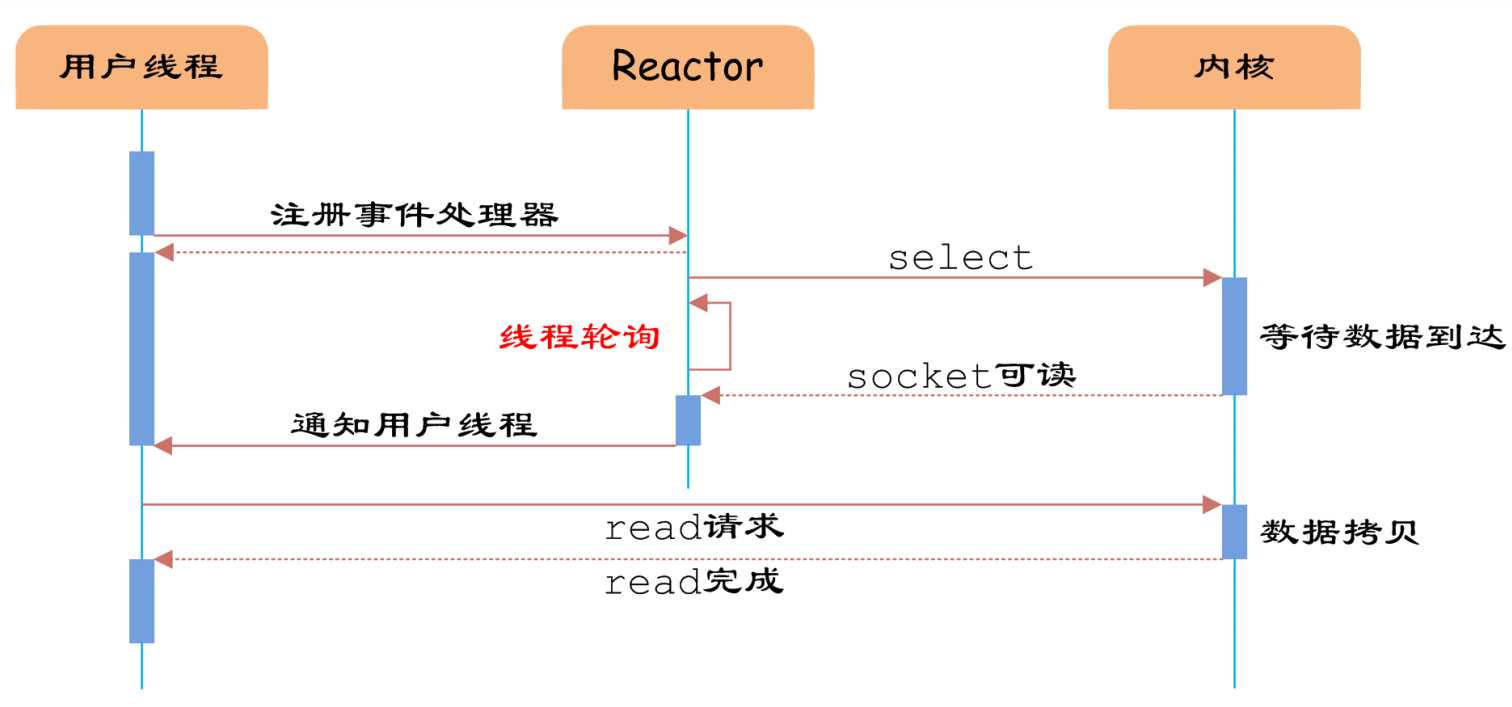


select ， poll，epoll的说明

网络IO本质上是对FD（文件描述符）的操作，用户代码需要先从操作系统获取到FD，进而执行IO操作。
上述将的几种实现，主要体现在FD遍历上的不同。
poll是对select的一次改进，但是遍历FD方式是一致的。
在用户调用selector.selectedKeys()的时候，操作系统扫描所有socket，从系统内核复制到用户的内存。
随着连接数的增长，遍历、复制的时间线性增长，并且消耗内存随之增大。
epoll的模式仅关心活跃的部分，减少遍历和复制操作。

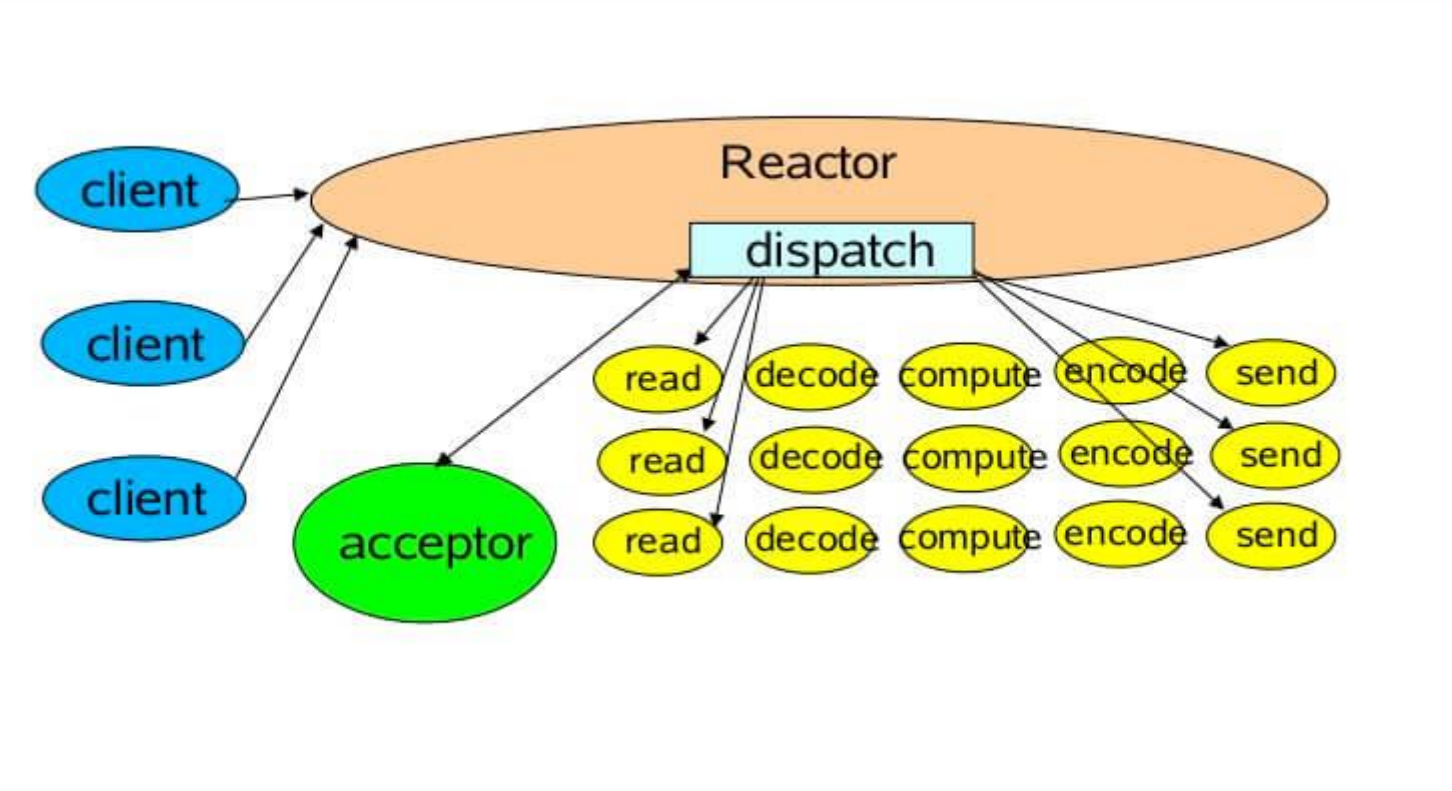
简单总结
epoll模式给高连接数，高并发的程序带来了性能的提高。

select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。



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

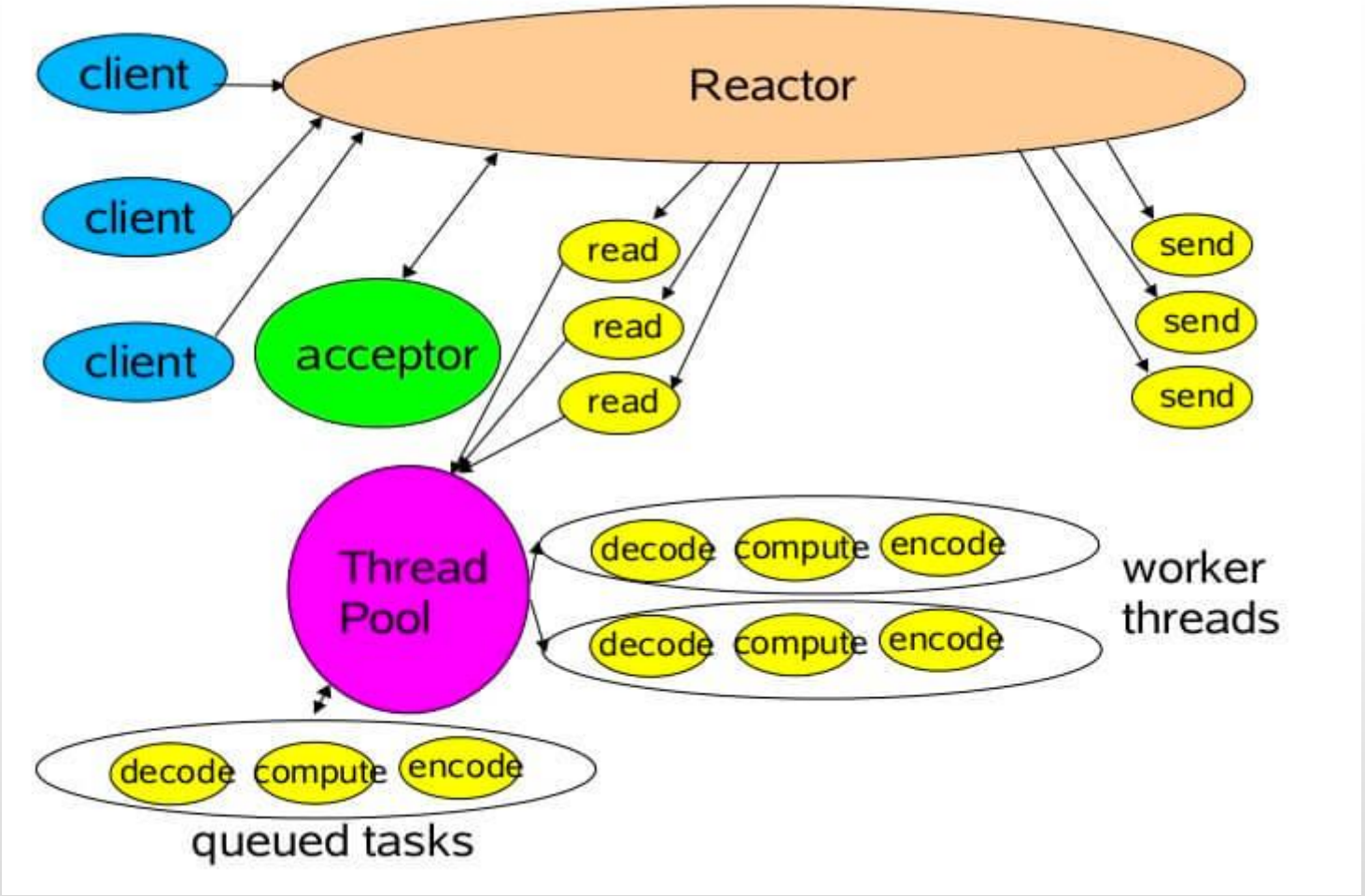
单 Reactor 单线程模型



所有的 IO 事件都绑定到 Selector 上，由 Reactor 统一分发

该模型适用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所以实际使用的不多

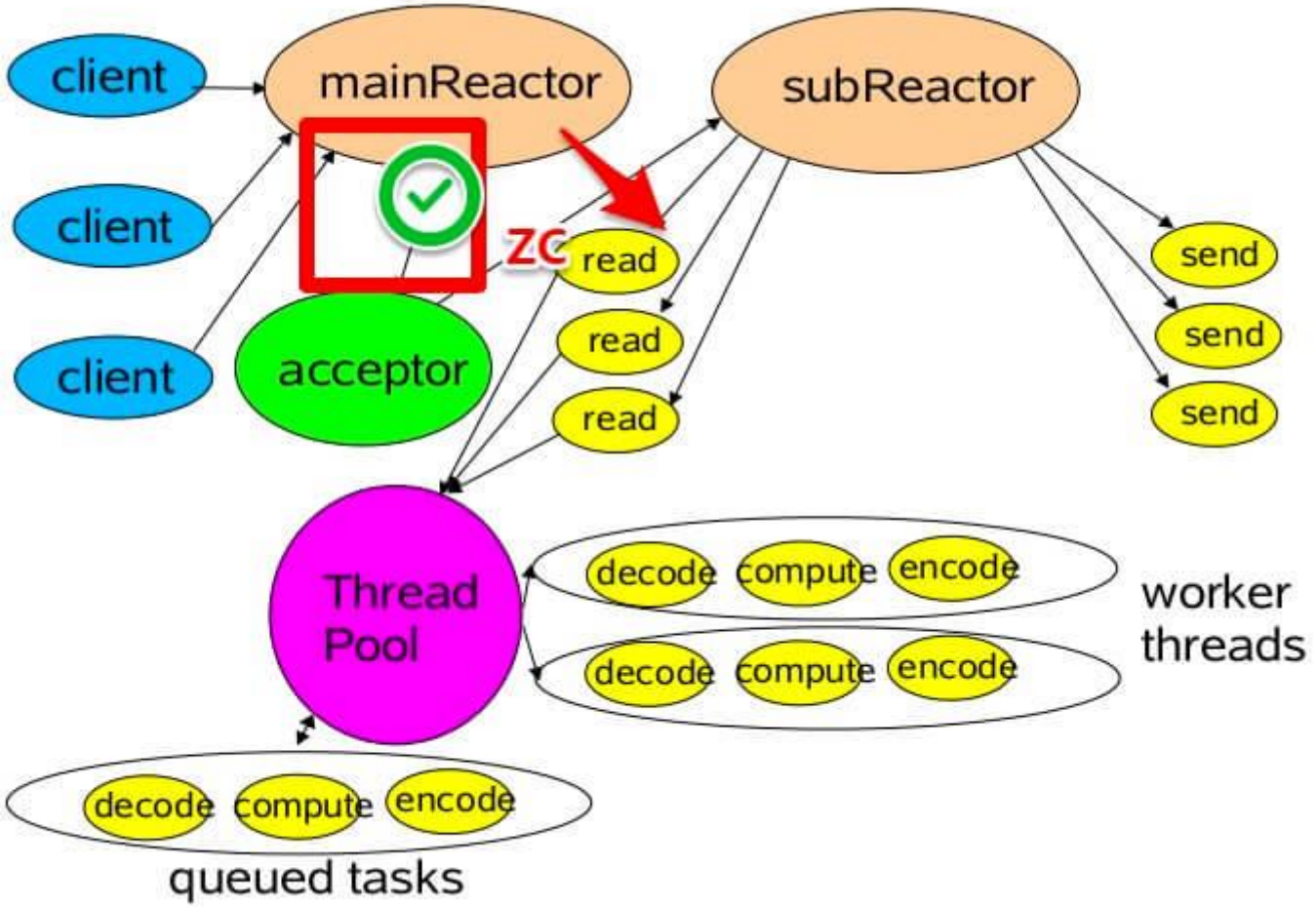
单 Reactor 多线程模型



停划算
停我享停 · 划算出行

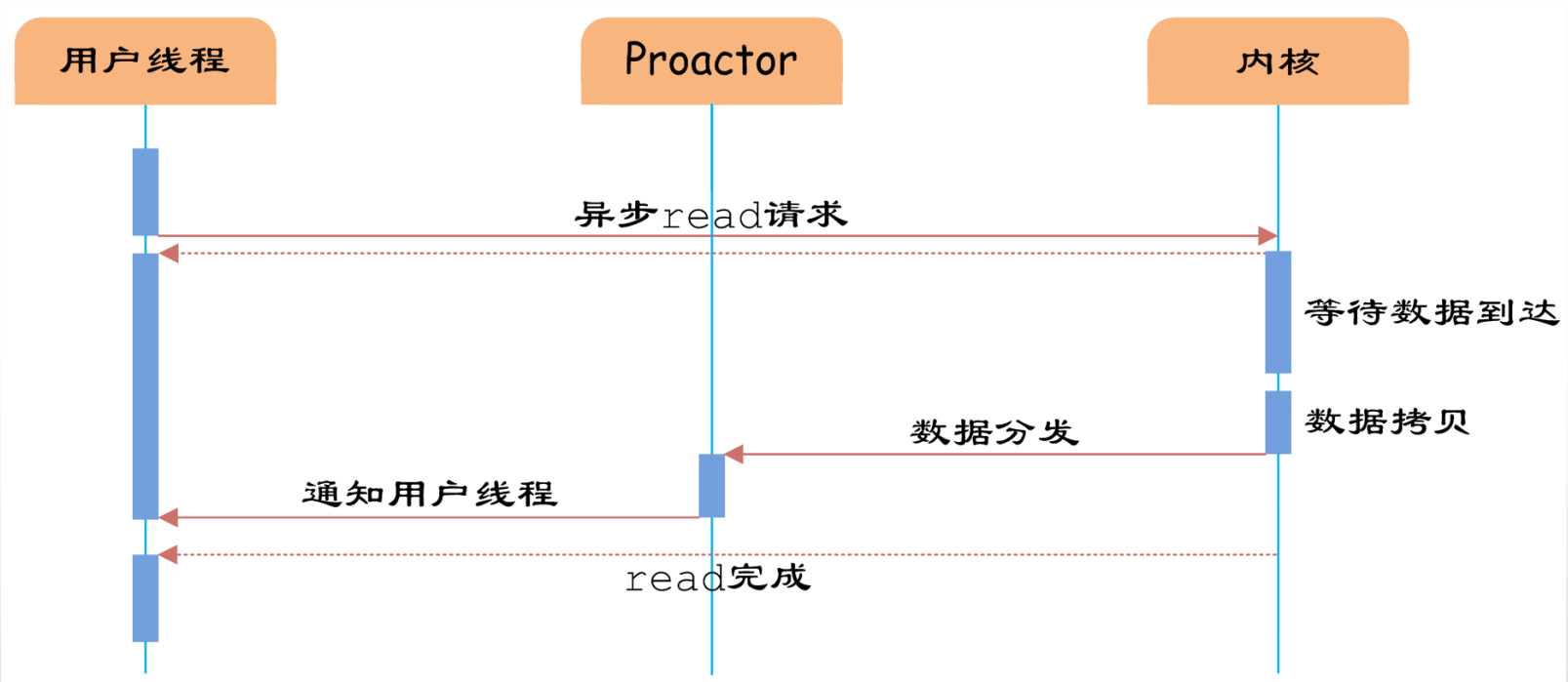
多Reactor 多线程模型

是将 Reactor 分成两部分：
1.mainReactor 负责监听 ServerSocketChannel ，用来处理客户端新连接的建立，并将建立的客户端的 SocketChannel 指定注册给 subReactor 。
2.subReactor 维护自己的 Selector ，基于 mainReactor 建立的客户端的 SocketChannel 多路分离 IO 读写事件，读写网络数据。对于业务处理的功能，另外扔给 worker 线程池来完成



此种模式的应用，目前有很多优秀的框架已经在应用，比如 Mina 和 Netty 等等。上述中去掉线程池的形式的变种，也是 Netty NIO 的默认模式

Proactor 模型：异步AIO的实现设计模式



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

Java NIO 简介

Buffer

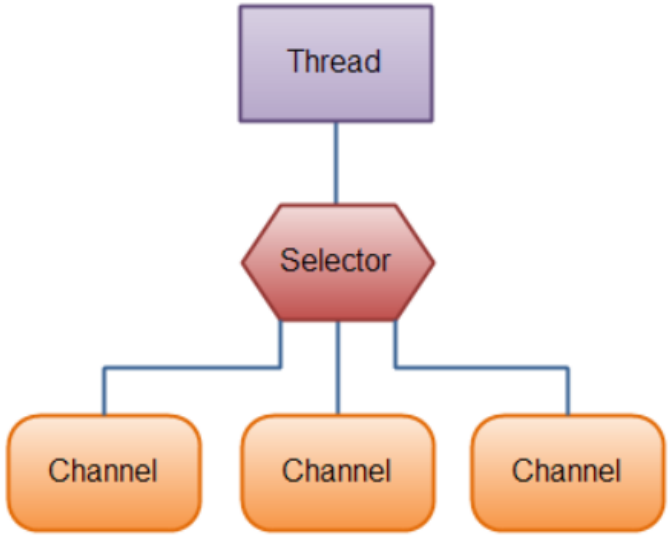
- 一个Buffer对象是固定数量的数据的容器。其作用是一个存储器，或者分段运输区，在这里数据可被存储并在之后用于检索

Channel

- 通道用于在字节缓冲区和位于通道另一边的实体(通常是一个文件或套接字)之间有效地传输数据

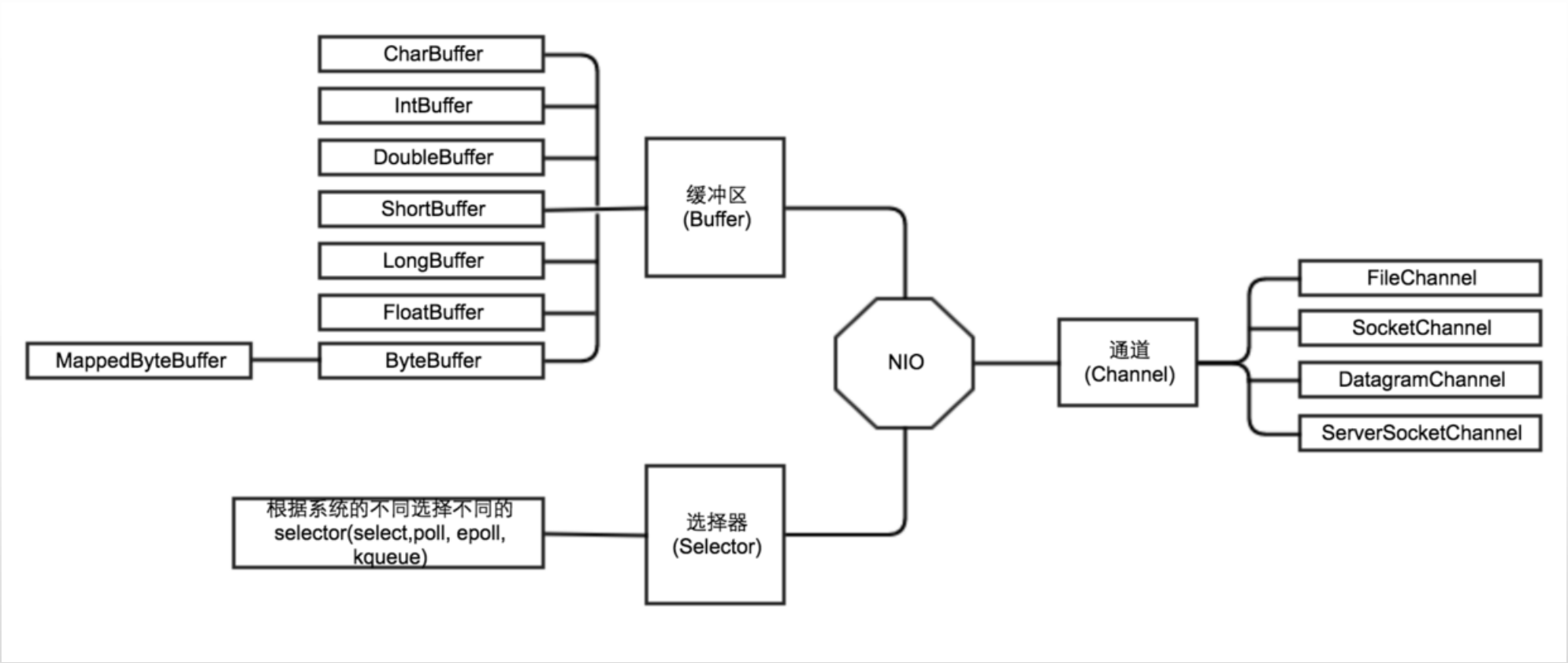
Selector

- 选择器类管理着一个被注册的通道集合的信息和它们的就绪状态。通道是和选择器一起被注册的，并且使用选择器来更新通道的就绪状态



停划算
停我享停 · 划算出行

Java NIO 组成



Buffer对象

容量Capacity

- 缓冲区能够容纳的数据元素的最大数量。容量在缓冲区创建时被设定，并且永远不能被改变。

上界Limit

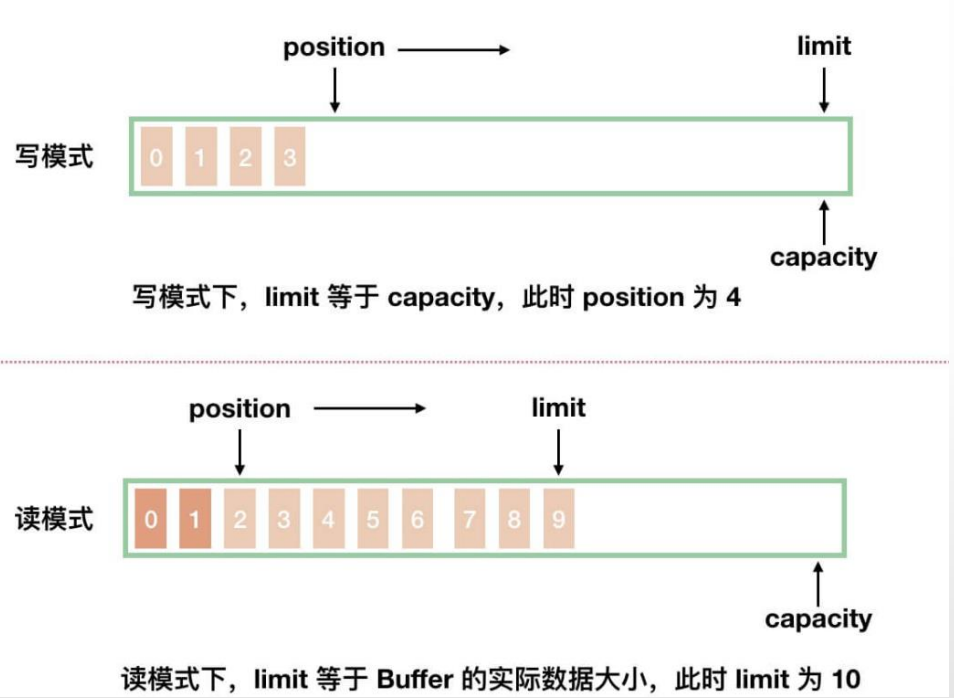
- 缓冲区里的数据的总数，代表了当前缓冲区中一共有多少数据

位置Position

- 下一个要被读或写的元素的位置。Position会自动由相应的 和 函数更新

标记Mark

- 一个备忘位置。用于记录上一次读写的位置



```
ByteBuffer byteBuffer = ByteBuffer.allocate(128);
byteBuffer.put(new byte[]{-26, -120, -111, -25, -120, -79, -28, -67, -96});
byteBuffer.flip();

/*对获取utf8的编解码器*/
Charset utf8 = Charset.forName("UTF-8");
CharBuffer charBuffer = utf8.decode(byteBuffer);/*对bytebuffer中的内容解码*/

/*array()返回的就是内部的数组引用，编码以后的有效长度是0~limit*/
char[] charArr = Arrays.copyOf(charBuffer.array(), charBuffer.limit());
System.out.println(charArr); /*运行结果：我爱你*/
```

```
/*通过Path对象创建文件通道*/
Path path = Paths.get("E:/noi_utf8.data");
FileChannel fc = FileChannel.open(path);

ByteBuffer bb = ByteBuffer.allocate((int) fc.size()+1);

Charset utf8 = Charset.forName("UTF-8");

/*阻塞模式，读取完成才能返回*/
fc.read(bb);

bb.flip();
CharBuffer cb = utf8.decode(bb);
System.out.print(cb.toString());
bb.clear();


fc.close();
```

```java
public NioServer() throws IOException {
    // 打开 Server Socket Channel
    serverSocketChannel = ServerSocketChannel.open();
    // 配置为非阻塞
    serverSocketChannel.configureBlocking(false);
    // 绑定 Server port
    serverSocketChannel.socket().bind(new InetSocketAddress(8080));
    // 创建 Selector
    selector = Selector.open();
    // 注册 Server Socket Channel 到 Selector
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
    System.out.println("Server 启动完成");

    handleKeys();
}

private void handleKeys() throws IOException {
    while (true) {
        // 通过 Selector 选择 Channel
        int selectNums = selector.select(30 * 1000L);
        if (selectNums == 0) {
            continue;
        }
        System.out.println("选择 Channel 数量: " + selectNums);

        // 遍历可选择的 Channel 的 SelectionKey 集合
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();
            iterator.remove(); // 移除下面要处理的 SelectionKey
            if (!key.isValid()) { // 忽略无效的 SelectionKey
                continue;
            }

            handleKey(key);
        }
    }
}
```

```java
public NioClient() throws IOException, InterruptedException {
    // 打开 Client Socket Channel
    clientSocketChannel = SocketChannel.open();
    // 配置为非阻塞
    clientSocketChannel.configureBlocking(false);
    // 创建 Selector
    selector = Selector.open();
    // 注册 Server Socket Channel 到 Selector
    clientSocketChannel.register(selector, SelectionKey.OP_CONNECT);
    // 连接服务器
    clientSocketChannel.connect(new InetSocketAddress(8080));

    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                handleKeys();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();

    if (connected.getCount() != 0) {
        connected.await();
    }
    System.out.println("Client 启动完成");
}

@SuppressWarnings("Duplicates")
private void handleKeys() throws IOException {
    while (true) {
        // 通过 Selector 选择 Channel
        int selectNums = selector.select(30 * 1000L);
        if (selectNums == 0) {
            continue;
        }

        // 遍历可选择的 Channel 的 SelectionKey 集合
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();
            iterator.remove(); // 移除下面要处理的 SelectionKey
            if (!key.isValid()) { // 忽略无效的 SelectionKey
                continue;
            }

            handleKey(key);
        }
    }
```

堆外内存

Direct Buffer

- 所分配的内存不在 JVM 堆上, 不受 GC 的管理.(但是 Direct Buffer 的 Java 对象是由 GC 管理的, 因此当发生 GC, 对象被回收时, Direct Buffer 也会被释放)。
- 因为 Direct Buffer 不在 JVM 堆上分配, 因此 Direct Buffer 对应用程序的内存占用的影响就不那么明显(实际上还是占用了这么多内存, 但是 JVM 不好统计到非 JVM 管理的内存.).
- 申请和释放 Direct Buffer 的开销比较大. 因此正确的使用 Direct Buffer 的方式是在初始化时申请一个 Buffer, 然后不断复用此 buffer, 在程序结束后才释放此 buffer.标记Mar
- 使用 Direct Buffer 时, 当进行一些底层的系统 IO 操作时, 效率会比较高, 因为此时 JVM 不需要拷贝 buffer 中的内存到中间临时缓冲区中.

Non-Direct Buffer

- 直接在 JVM 堆上进行内存的分配, 本质上是 byte[] 数组的封装.
- 因为 Non-Direct Buffer 在 JVM 堆中, 因此当进行操作系统底层 IO 操作中时, 会将此 buffer 的内存复制到中间临时缓冲区中. 因此 Non-Direct Buffer 的效率就较低.

停划算
停我享停 · 划算出行

谢谢！

共享百万车位、服务千万车主、构建万亿生态