网络编程介绍

mindwind

2011-11-20

大 纲

- 网络通信基本概念
- I/O模型
- JAVA网络编程模型变迁及比较
- 实战：JVAV NIO 编程

基本概念

- 协议分层

  协议的独立性、简单性

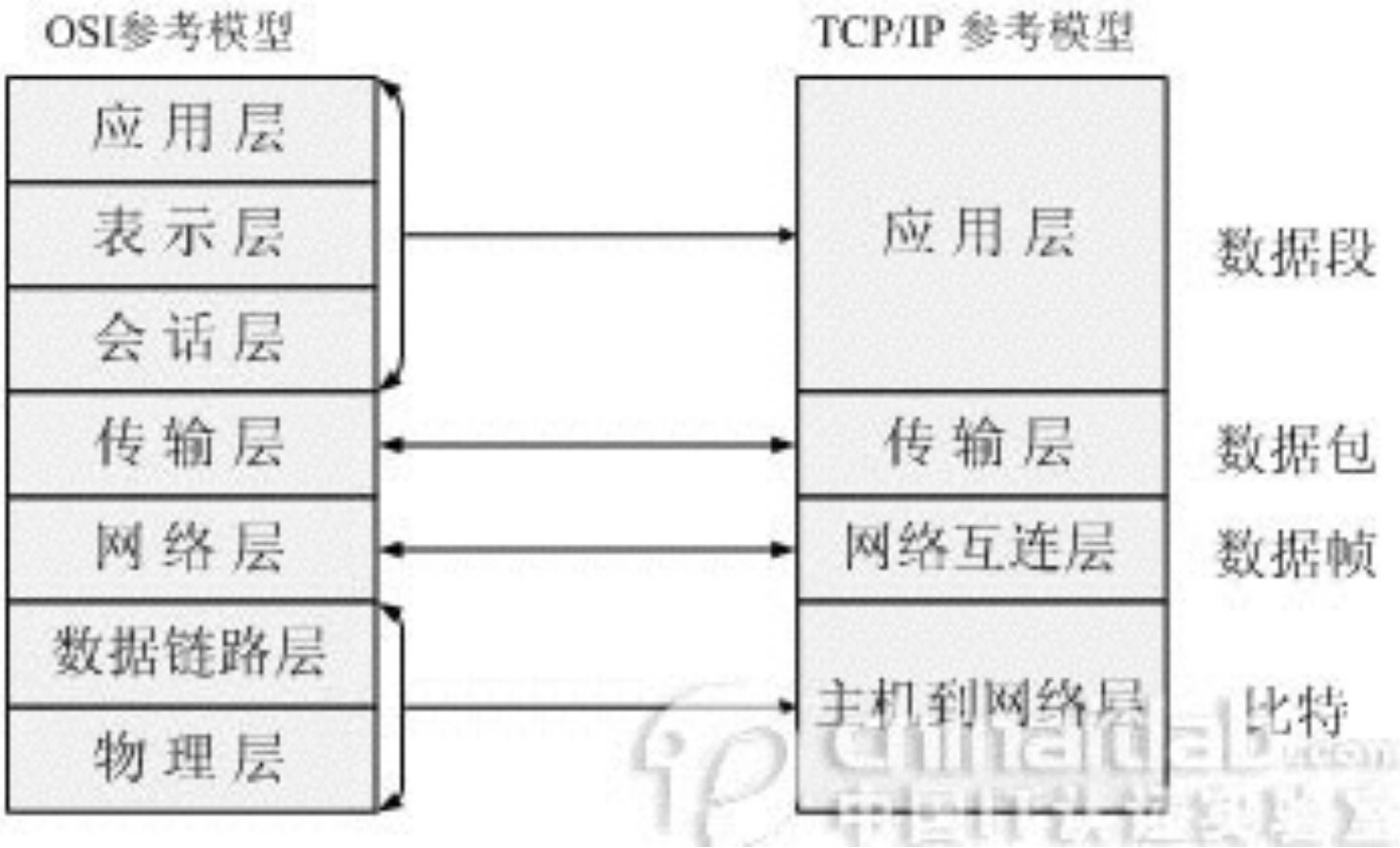
- 数据封装

  栈式推进

- 数据传输

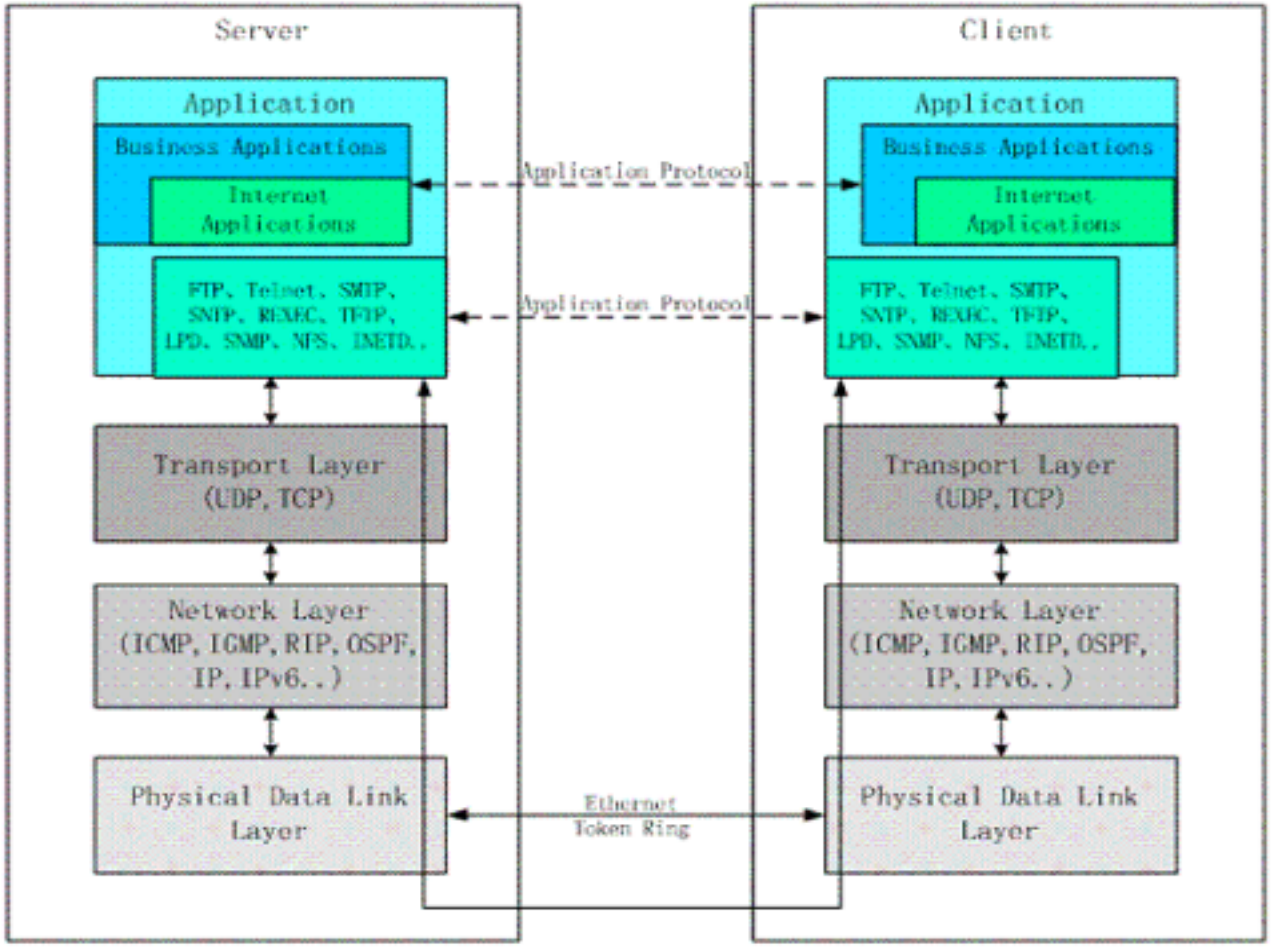
  面向连接或无连接

协议分层模型

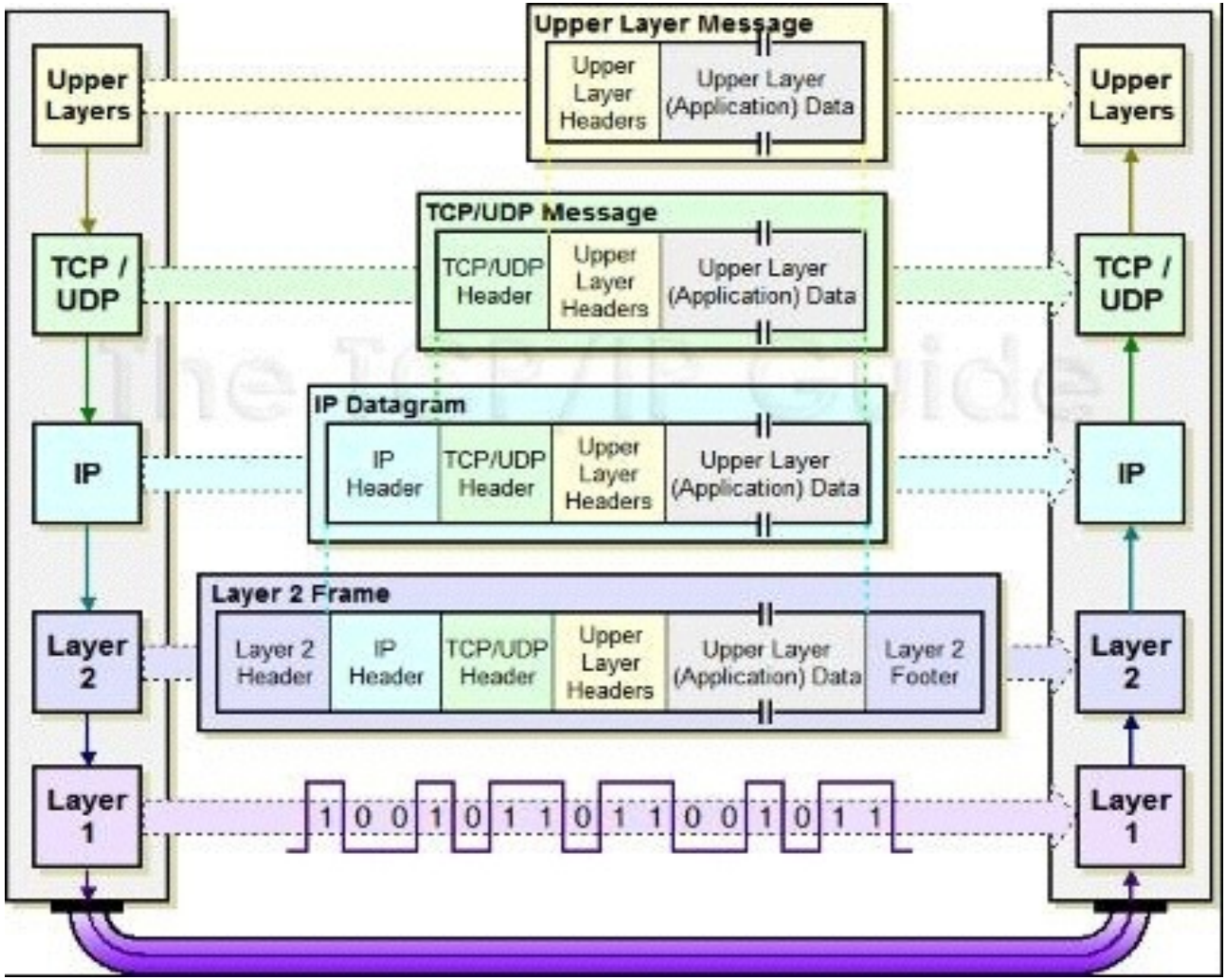


OSI模型：X.25协议（仅实现下三层）

TCP/IP协议通信层次

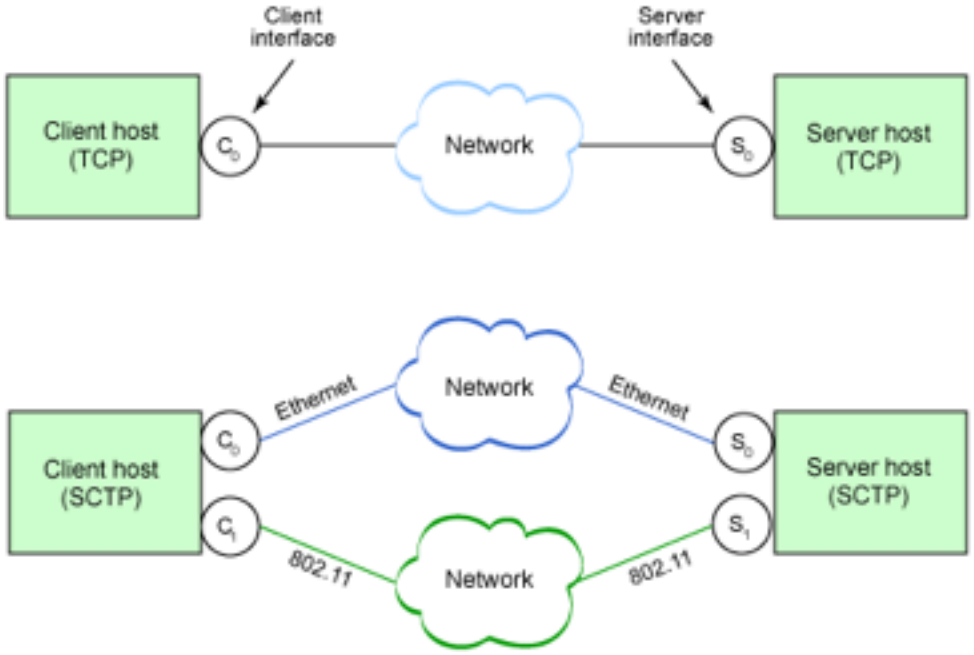


数据封装


Upper Layer Message
Upper Layer Headers
Upper Layer (Application) Data
TCP/UDP Message
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
IP Datagram
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Frame
Layer 2 Header
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Footer
Upper Layers
TCP / UDP
IP
Layer 2
Layer 1
1 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1

MTU

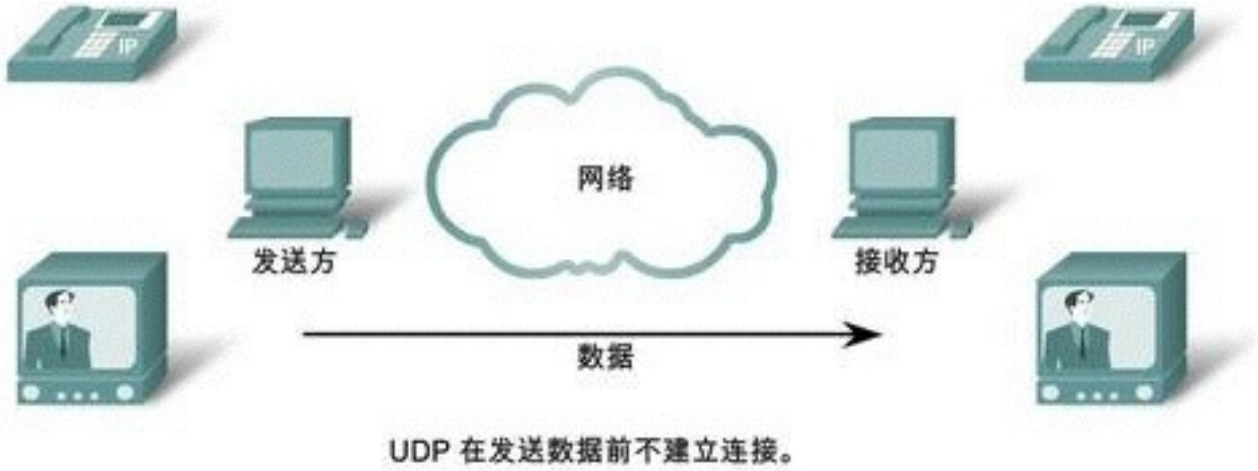
数据传输 - 有连接



TCP

- 虚电路连接（非物理）
- 无结构的数据流
- 可靠、有序、带缓冲的传输
- 开销大

数据传输 - 无连接



UDP

- 数据包传输
- 不可靠
- 开销低

I/O 模型

- I/O请求的两个阶段

  等待数据就绪

  从内核缓冲区拷贝到进程缓冲区

- I/O操作的两种类型

  同步IO

  异步IO

- 5种I/O模型

5种 I/O 模型

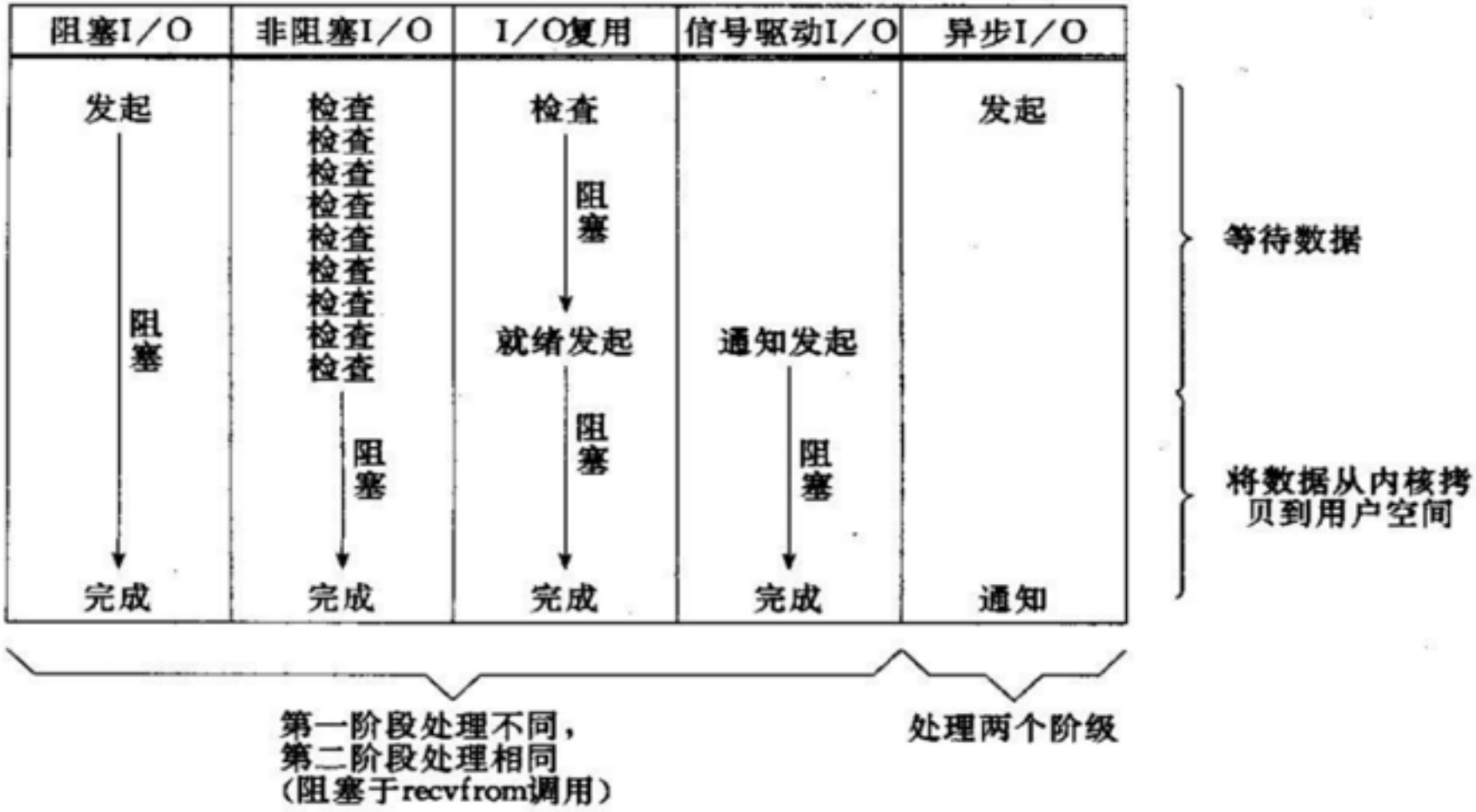
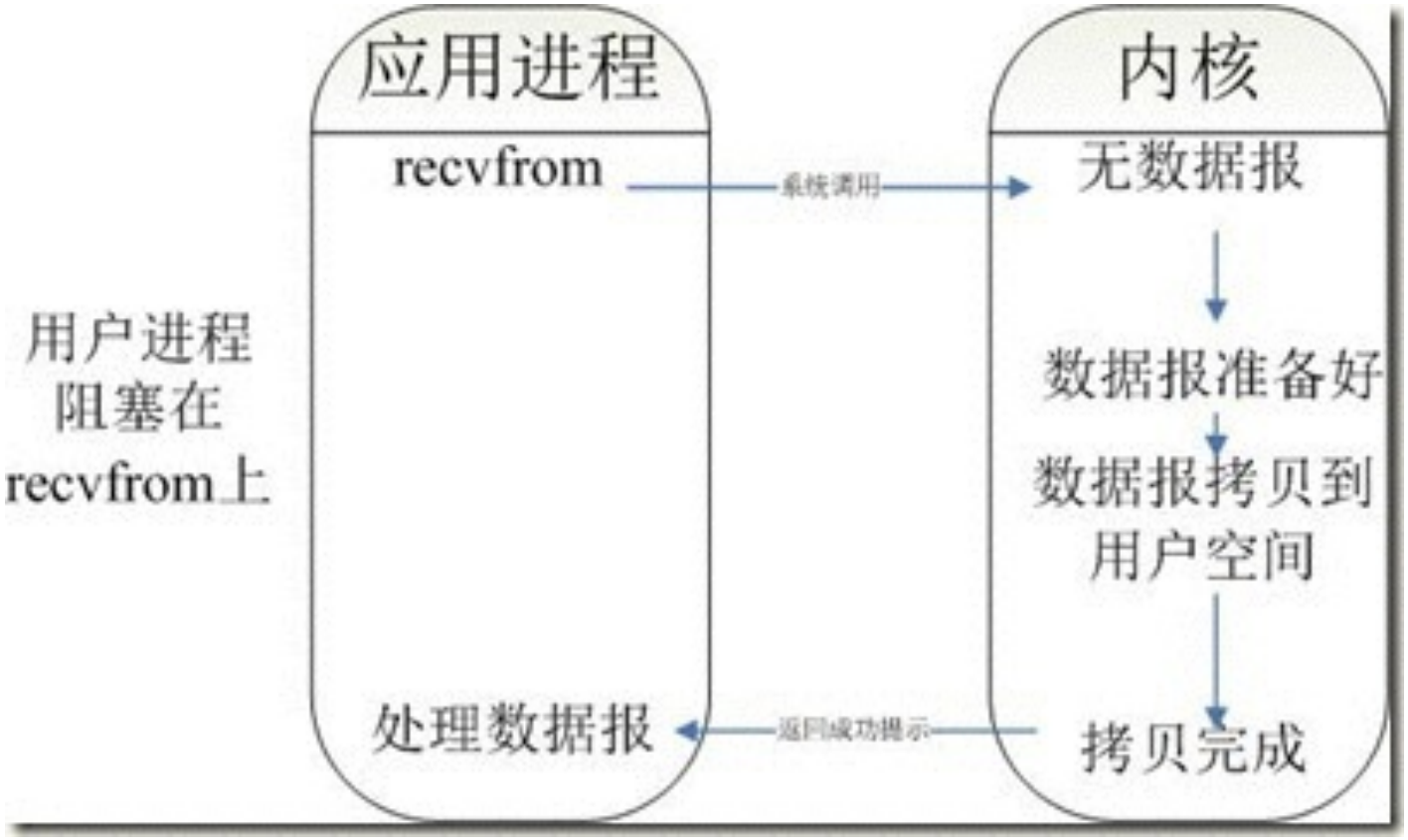
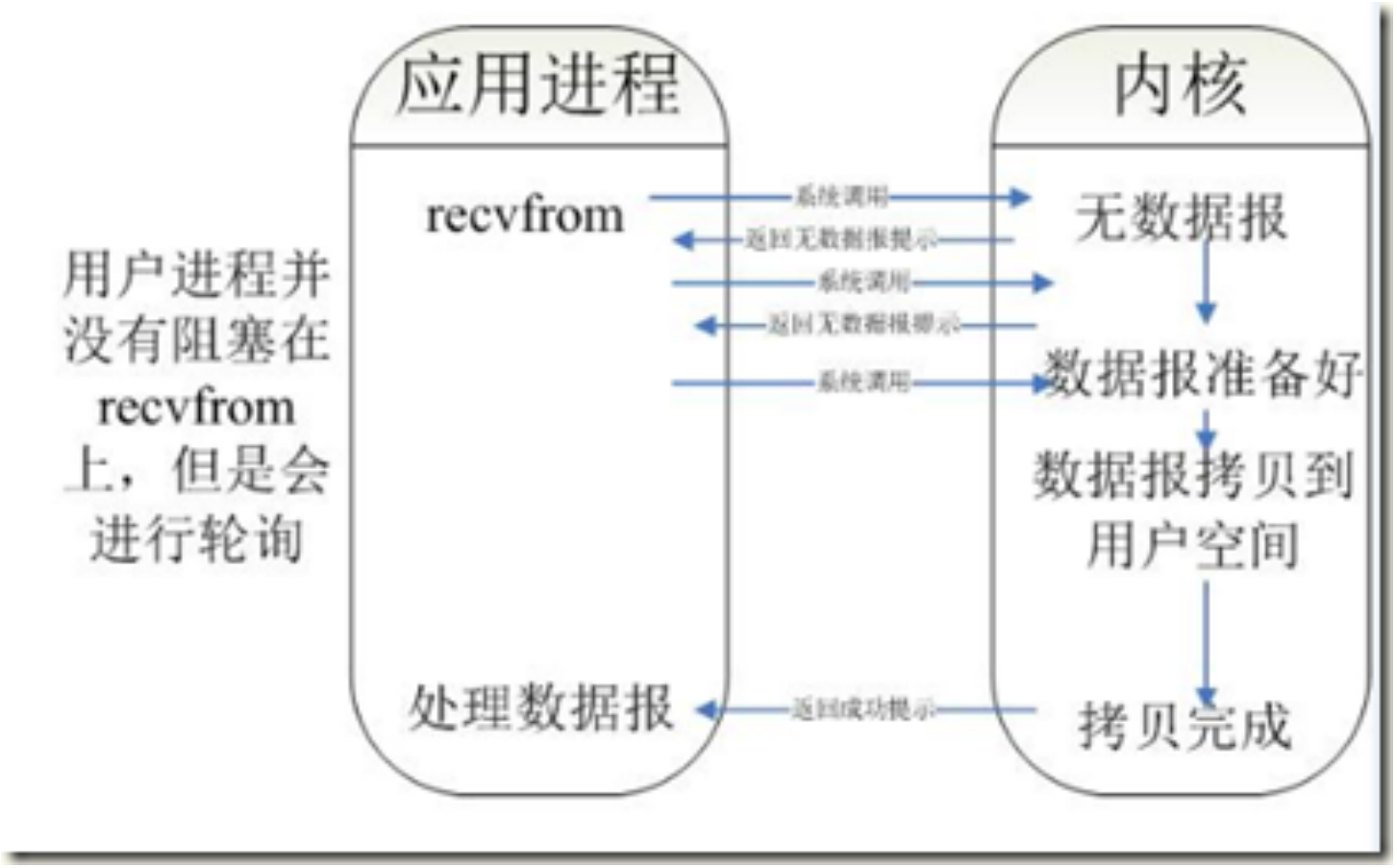


图 6.6　五个 I/O 模型的比较

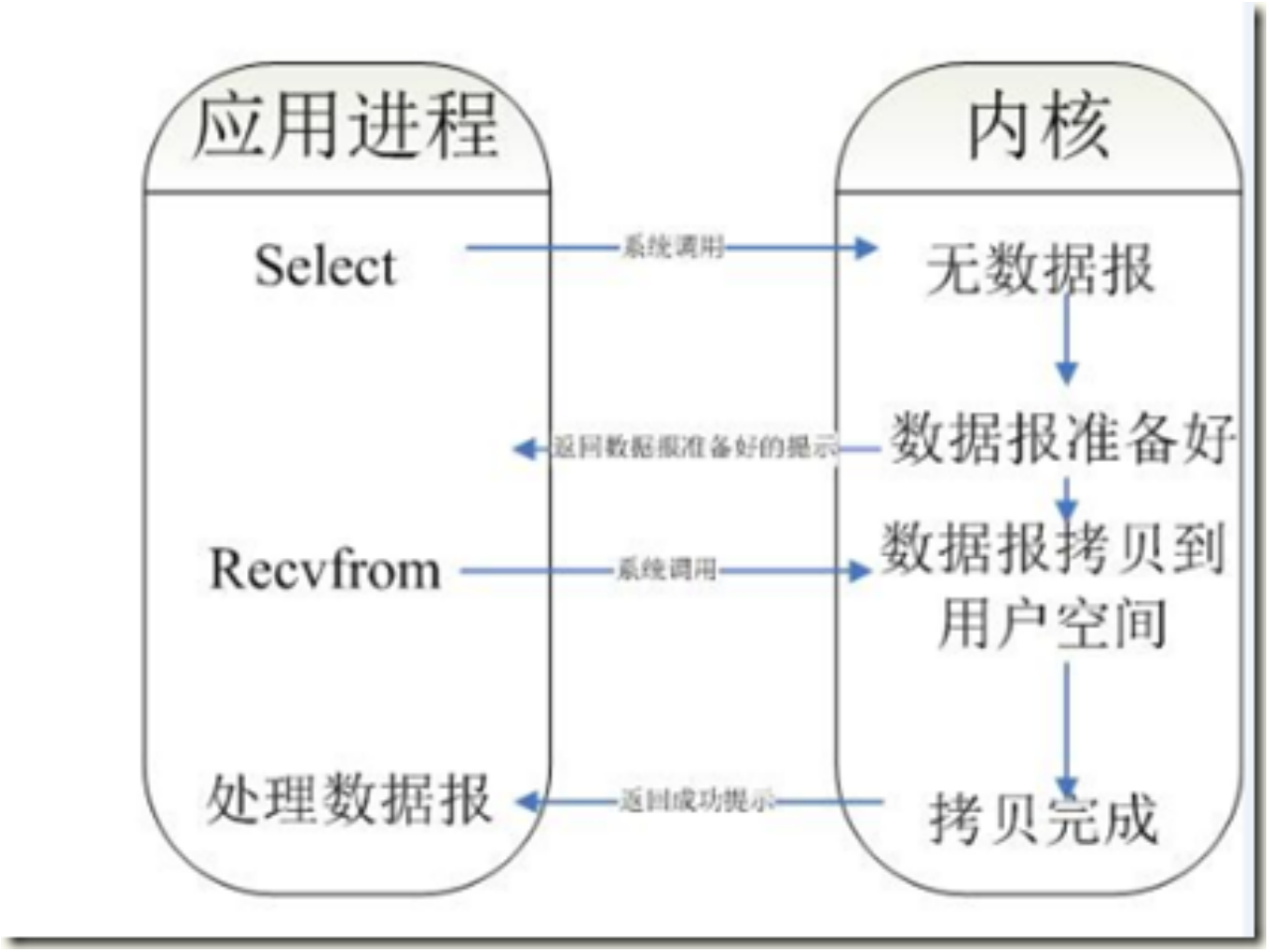
阻塞 I/O


应用进程
内核
recvfrom
系统调用
无数据报
用户进程
阻塞在
recvfrom上
数据报准备好
数据报拷贝到
用户空间
处理数据报
返回成功提示
拷贝完成

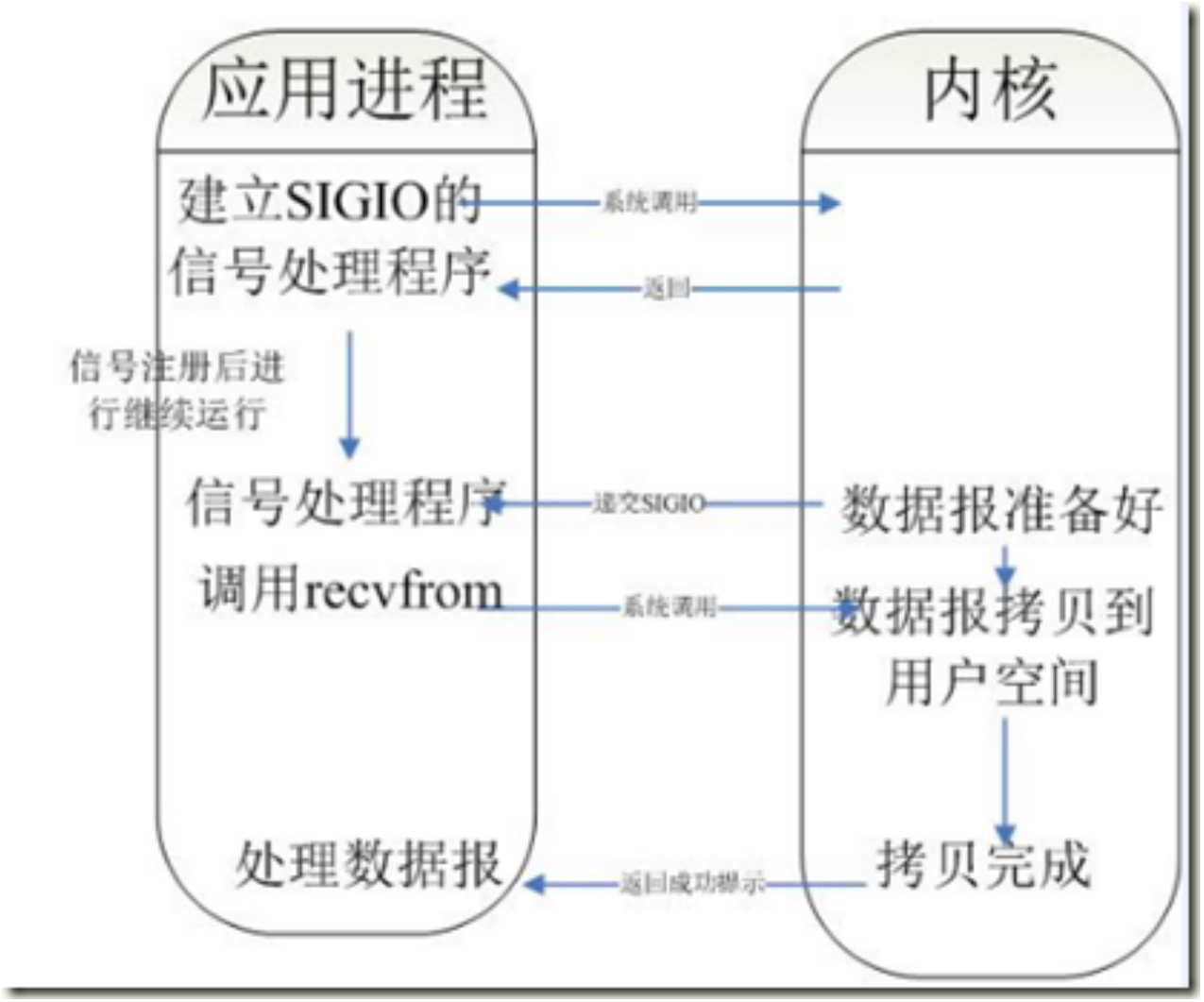
非阻塞 I/O


应用进程
内核
recvfrom
无数据报
系统调用
返回无数据报提示
系统调用
返回无数据报提示
系统调用
数据报准备好
数据报拷贝到用户空间
用户进程并没有阻塞在recvfrom上，但是会进行轮询
处理数据报
返回成功提示
拷贝完成

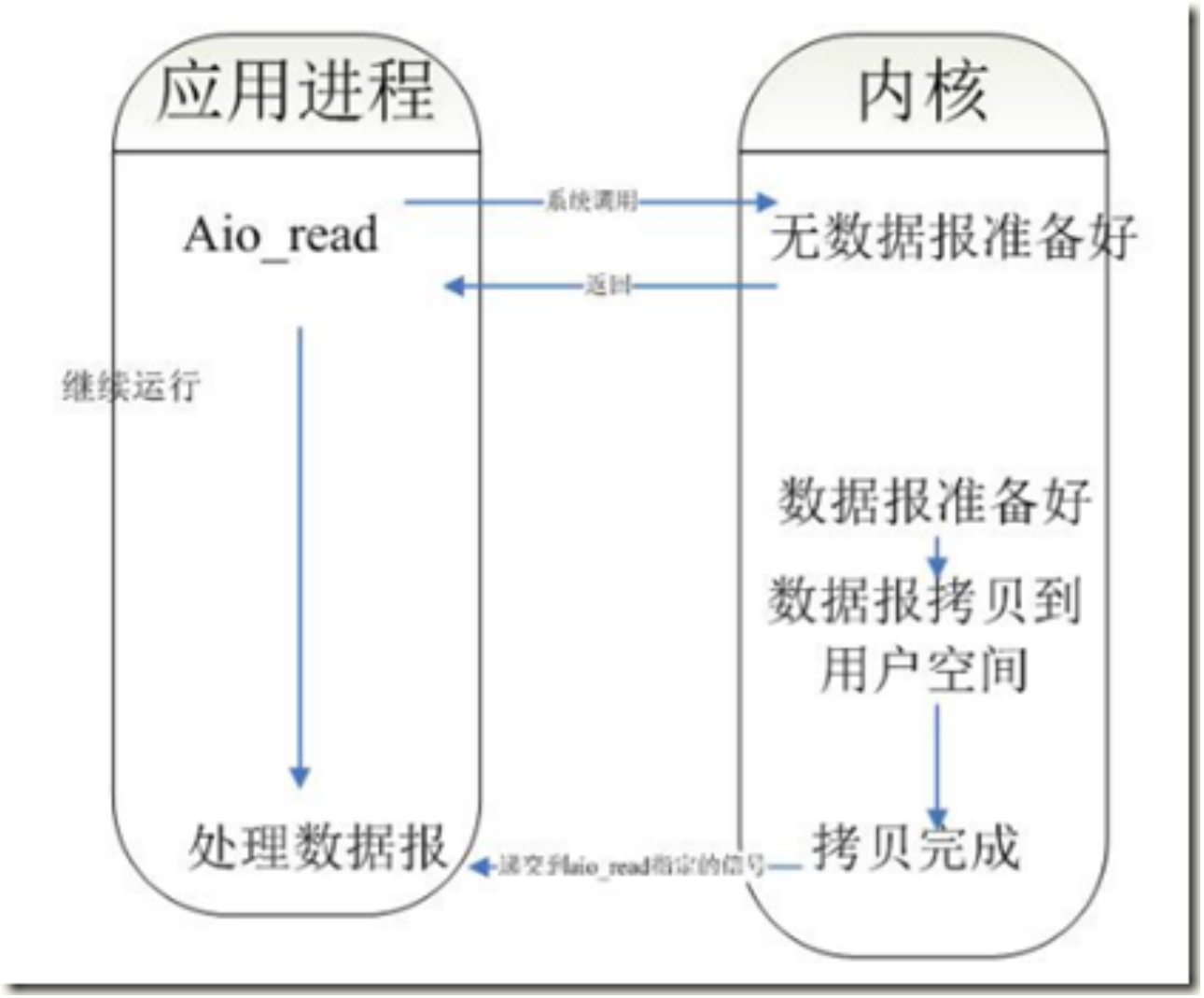
I/O 复用


应用进程
内核
Select
系统调用
无数据报
数据报准备好
返回数据报准备好的提示
Recvfrom
系统调用
数据报拷贝到用户空间
处理数据报
返回成功提示
拷贝完成

信号驱动I/O


应用进程
内核
建立SIGIO的
信号处理程序
系统调用
返回
信号注册后进
行继续运行
信号处理程序
递交SIGIO
数据报准备好
调用recvfrom
系统调用
数据报拷贝到
用户空间
处理数据报
返回成功指示
拷贝完成

异步I/O


应用进程
内核
Aio_read
系统调用
无数据报准备好
返回
继续运行
数据报准备好
数据报拷贝到
用户空间
处理数据报
拷贝完成

JAVA网络编程模型变迁

- BIO(Blocking I/O)

  JDK1.4前，每连接每线程，阻塞I/O模型

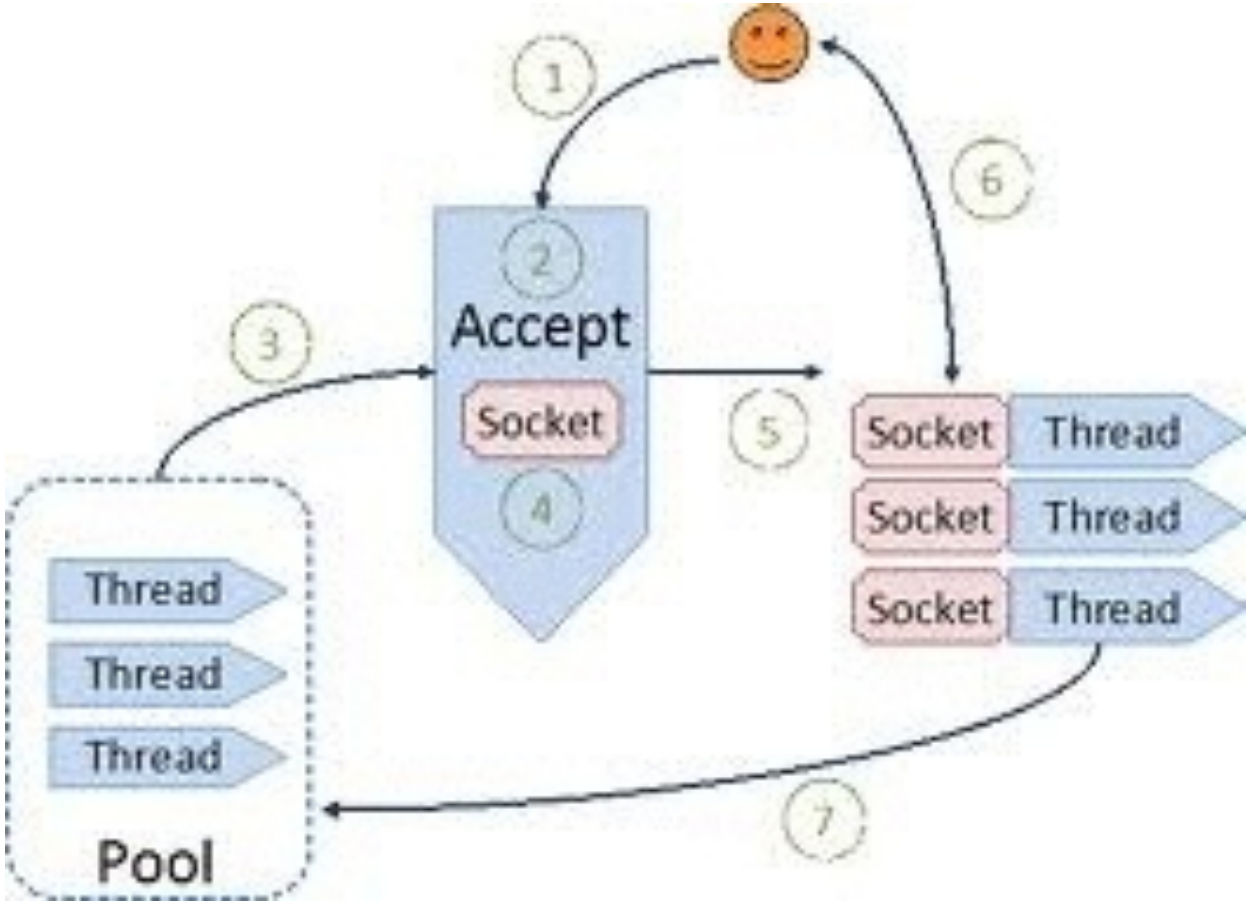
- NIO(New I/O)

  JDK1.4后，Reactor模式，I/O复用模型

- AIO(Asynchronous I/O)

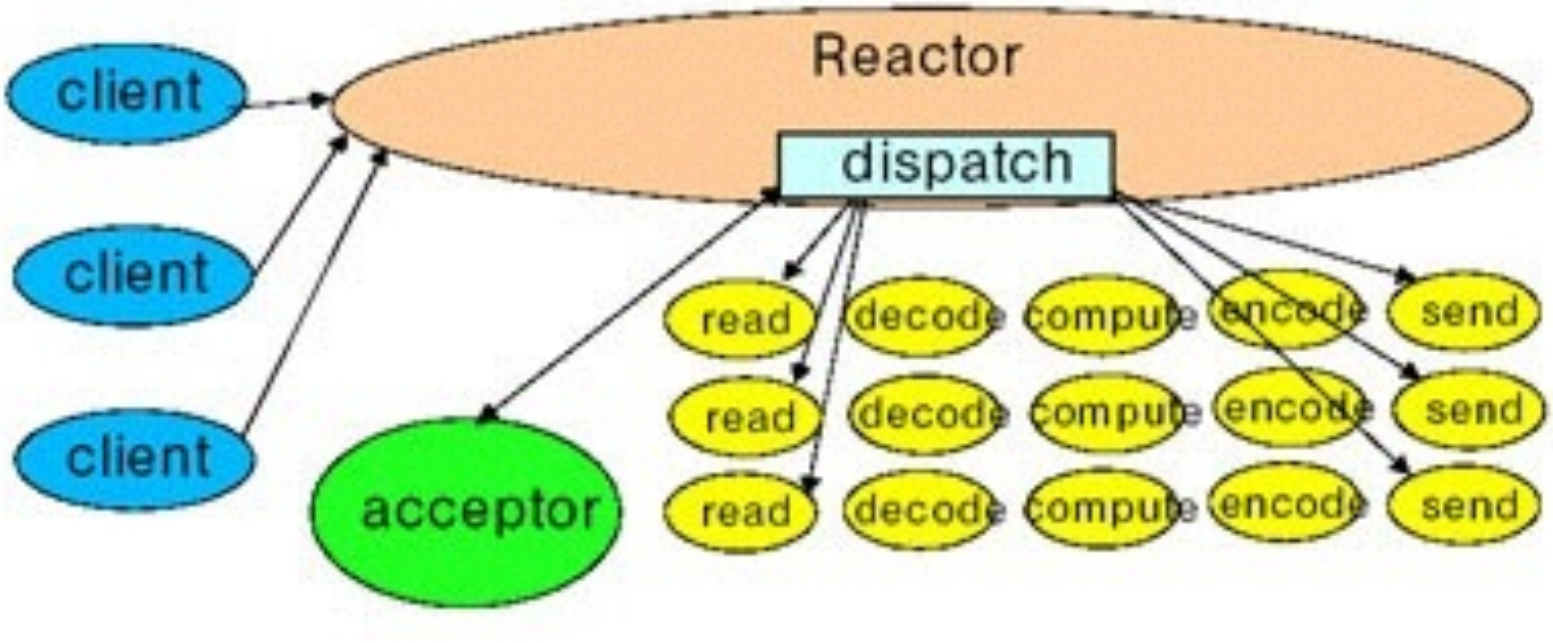
  JDK1.7，Proactor模式， 异步I/O模型

BIO



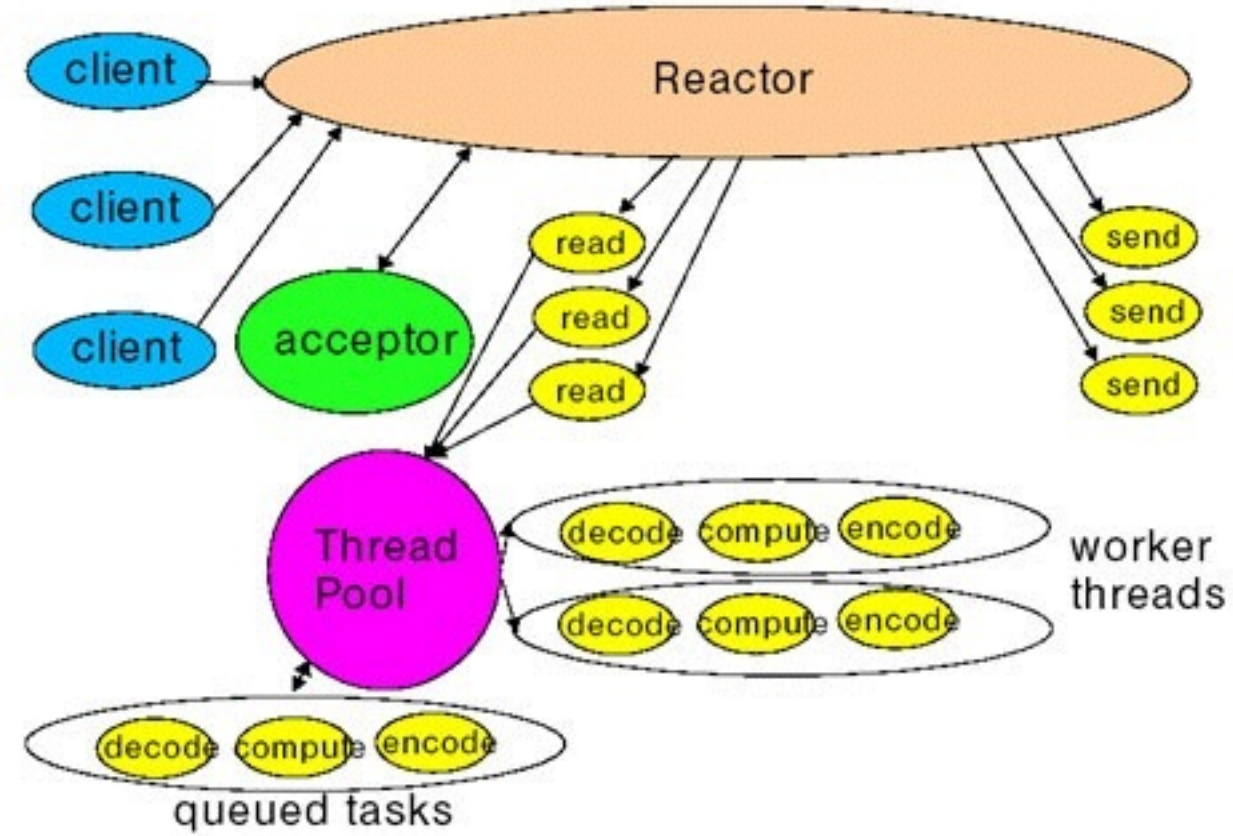
tomcat6之前的实现方式

NIO



Reactor模式单线程场景

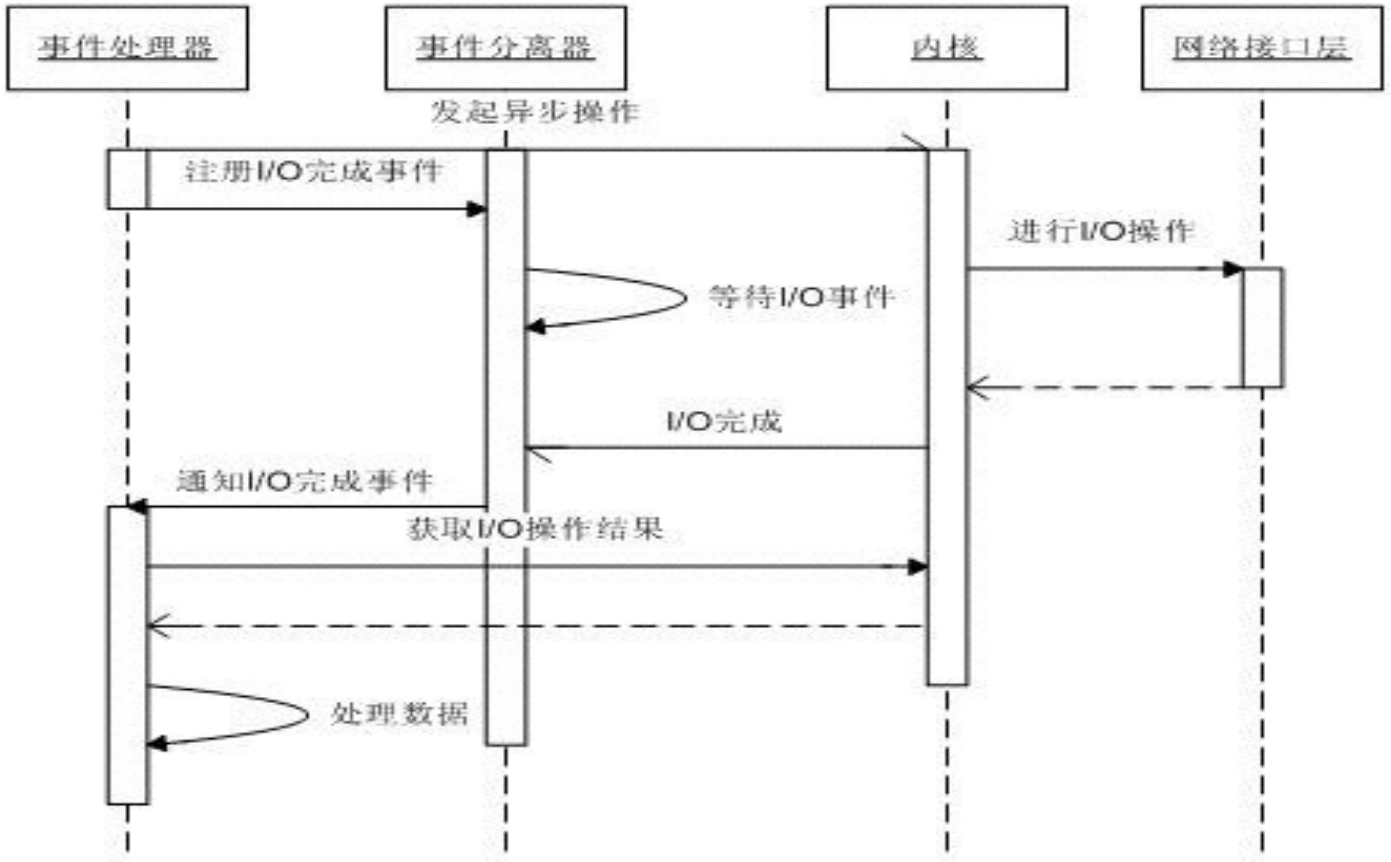
NIO



Reactor模式结合线程池场景

mina/netty/cindy/tomcat6+等开源产品都是此模式的实现者

AIO


事件处理器
事件分离器
内核
网络接口层
发起异步操作
注册I/O完成事件
进行I/O操作
等待I/O事件
I/O完成
通知I/O完成事件
获取I/O操作结果
处理数据

实战：JAVA NIO 编程

- NIO 核心构成

  Selector
  Channel
  Buffer
  SelectionKey

- NIO 天使还是魔鬼?

  NIO = 高性能 ?
  NIO使用技巧与陷阱

NIO Selector

- java.nio.channels.Selector
- 支持IO多路复用的抽象实体
- 不同平台不同实现，Linux2.6+，JDK1.6默认采用了Epoll
- 用于注册Channel进行监听

NIO Channel

- java.nio.channels.SocketChannel
- java.nio.channels.ServerSocketChannel
- 数据传输通道的抽象
- 与Buffer配合进行批量数据传输，块传输方式

NIO SelectionKey

- java.nio.channels.SelectionKey
- Selector和Channel注册关系的凭证，同时关联二者
- 保存Channel感兴趣的事件
- Selector.select()返回时更新所有SelectionKey的状态，返回事件就绪的channel个数

NIO Buffer

- java.nio.Buffer

  缓冲去抽象类包括两种实现

- HeapByteBuffer

  基于byte[]数组实现

  内存维护在JVM堆上

- DirectByteBuffer

  底层存储在非JVM堆上，通过native代码操作

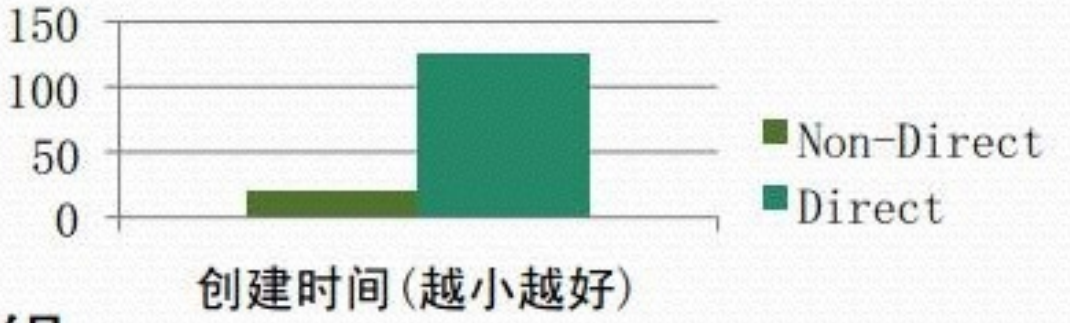
  无手工回收机制，gc时自动回收

HeapBuffer VS DirectBuffer

	DirectByteBuffer	HeapByteBuffer
创建开销	大	小
存储位置	Native heap	Java heap
数据拷贝	无需临时缓冲区做拷贝	拷贝到临时DirectByteBuffer，但临时缓冲区使用缓存。聚集写/发散读时没有缓存临时缓冲区。
GC影响	每次创建或者释放的时候都调用一次System.gc()	

HeapBuffer VS DirectBuffer

- 创建4K缓冲区



- 拷贝16K数组

Buffer的选择

场景	选择
不知道该用哪种buffer	Non-Direct
没有参与IO操作	Non-Direct
中小规模应用(<=1K并发连接)	Non-Direct
长生命周期，较大的缓冲区	Direct
测试证明Direct比Non-Direct更快	Direct
进程间数据共享(JNI)	Direct
一个Buffer发给多个Client	考虑使用View ByteBuffer共享数据(buffer.slice())

NIO 天使？魔鬼？

- NIO天使

  适用于高并发、高网络延时场景

  同样的并发吞吐，相对BIO更小的线程开销。

- NIO魔鬼

  离散且不易理解的事件驱动模型

  写出健壮、稳定的程序困难

  陷阱重重

NIO 技巧与陷阱

- 典型I/O编程模式

  open - read - write - close

- NIO 编程模式

  open - register - wait - notify - wakeup - read/write - close

NIO register 和 interest

- 陷阱

  一有连接接入时直接注册?

  channel.register(Selector sel, int ops, Object att)

  不可见的Selector内部锁

  不可避免的外部同步锁

  效率低

- 技巧

  异步注册，加入缓冲队列，唤醒Selector.select()

  reactor单线程循环处理

  注意：SelectionKey.interestOps(int ops) 在linux平台的实现也有内部锁竞争，编程方式类似注册处理。

NIO 正确处理写事件（OP_WRITE）

- 陷阱

  CPU 100%的噩梦，OP_WRITE事件处理不当导致

- 技巧

  仅在有数据写时才注册（注册了不代表立刻可写）

  channel.write()之后：

  一次循环没有写完，继续注册

  写完后立刻取消注册

NIO 连接空闲管理

- 陷阱

  每次IO读写记录时间戳

  另起独立监控线程扫描所有连接判断当前时间和记录的时间戳差值

  超过阈值触发idle事件通知（可能close连接）

  问题：低并发下没有问题，高并发下reactor循环线程事件派发不及时可能导致连接被提前关闭。

- 技巧

  select方式：Selector.select(timeout)而非Selector.select()

  在reactor线程中进行idle检查，无需额外线程

  注意：http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6403933

  关于select(timeout)的bug在jdk6u4后才彻底解决

总 结

- 分布式程序设计的基石
- 了解魔法背后的秘密

  webservice、ice、thrift、rmi、http

- 高效正确的使用网络程序API

谢 谢

Q & A