非阻塞通信的用法

信息与通信工程学院　胡 铮

huzheng@bupt.edu.cn

主要内容

- 非阻塞的概念
- Java的非阻塞模式的支持类
- 编程范例
  - 服务器
  - 客户端

线程阻塞的概念（1）

- 线程在运行中会因为某些原因而阻塞，共同特征：
  - 放弃CPU，暂停运行
  - 只有等到导致阻塞的原因消除，才能恢复运行。
  - 或者被其他线程中断，退出阻塞状态，抛出InterruptedException

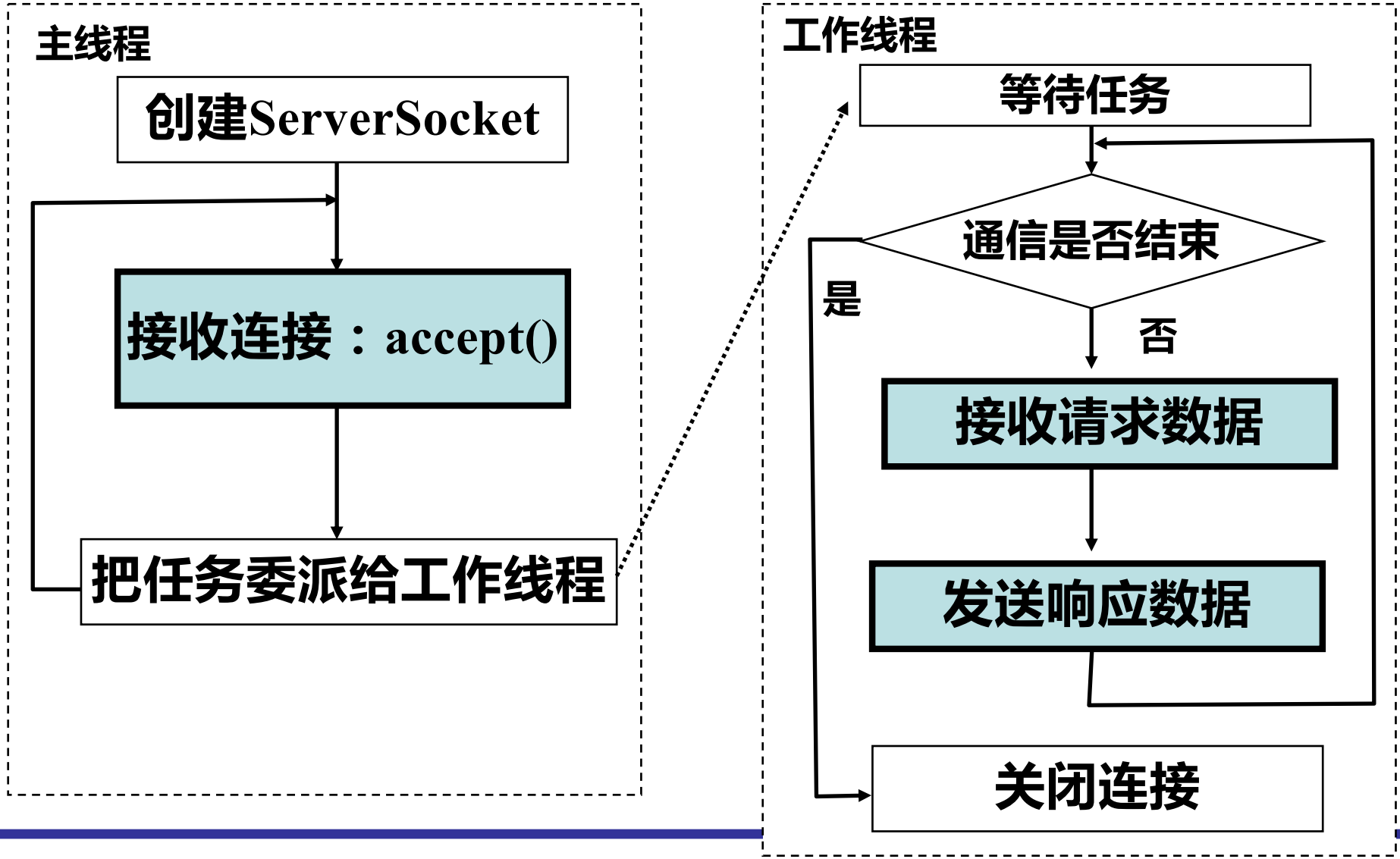
线程阻塞的概念（2）

- 线程阻塞的原因：
  - 执行了Thread.sleep(int n)方法
  - 要执行一段同步代码，无法获得相关的同步锁
  - 执行了一个对象的wait()方法，等待notify()或notifyAll()方法，才可能将其唤醒
  - 执行I/O操作或进行远程通信时，会因为等待相关的资源而阻塞。如：System.in.read()

线程阻塞的概念（3）

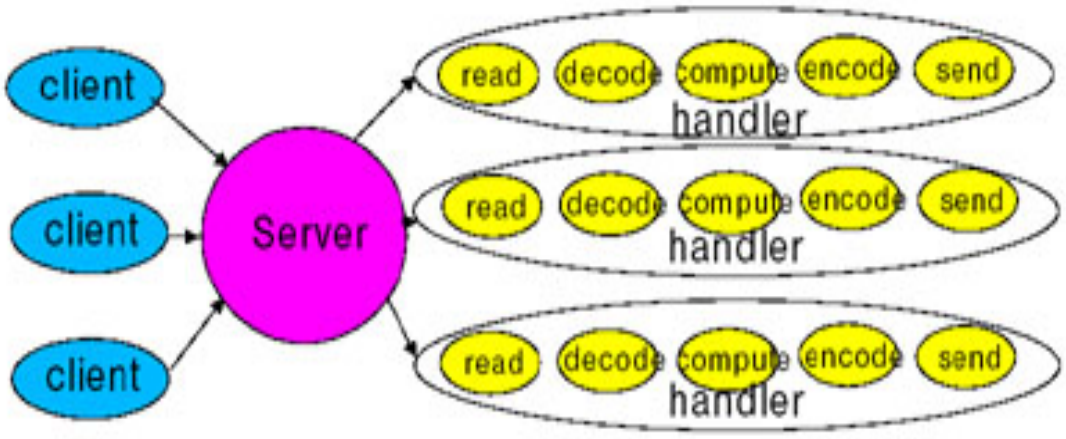
- 客户通信程序
  - 调用构造函数、或者connect()方法，直到连接成功
  - 从Socket的输入流读入数据
  - 向Socket的输出流写数据
  - setSoLinger方法设置了关闭Socket的延迟时间后，close()方法会进入阻塞
- 服务器程序
  - ServerSocket的accept()方法
  - 从Socket输入流输入，以及从输出流写数据

多线程处理阻塞通信


主线程
创建ServerSocket
接收连接：accept()
把任务委派给工作线程
工作线程
等待任务
通信是否结束
是
否
接收请求数据
发送响应数据
关闭连接

- 这些网络服务的底层都离不开对socket的操作。他们都有一个共同的结构：

1. Read request
2. Decode request
3. Process service
4. Encode reply
5. Send reply



局限分析

- 工作线程并不是越多越好
  - Java虚拟机会为每个线程分配独立的堆栈空间，工作线程数目越多，开销越大，增加JVM调度线程的负担，增加线程同步的复杂性，提高了线程死锁的可能性
  - 工作线程的许多时间都浪费在阻塞I/O操作上，Java虚拟机需要频繁地转让CPU的使用权，使进入阻塞状态的线程放弃CPU，再把CPU分配给处于可运行状态的线程

非阻塞通信的基本思想

- 举例
  - 烧开水

锅里放水，打开煤气炉
等待水烧开　　//阻塞
关闭煤气炉，把开水灌到壶里

煮粥

锅里放水和米，打开煤气炉
等待粥烧开　　//阻塞
调小煤气炉火力
等待粥烧熟　　//阻塞
关闭煤气炉

```
锅里放水，打开煤气炉                    //开始烧开水
锅里放水和米，打开煤气炉                //开始烧粥
while(一直等待，直到有水烧开、粥烧开或粥烧熟事件发生) {      //阻塞
    if (水烧开){
        关闭煤气炉，把开水灌到壶里；
    }
    if (粥烧开){
        调小煤气炉火力；
    }
    if (粥烧熟){
        关闭煤气炉；
    }
    if (水烧开且粥已经烧熟){
        退出循环；
    }
}
```

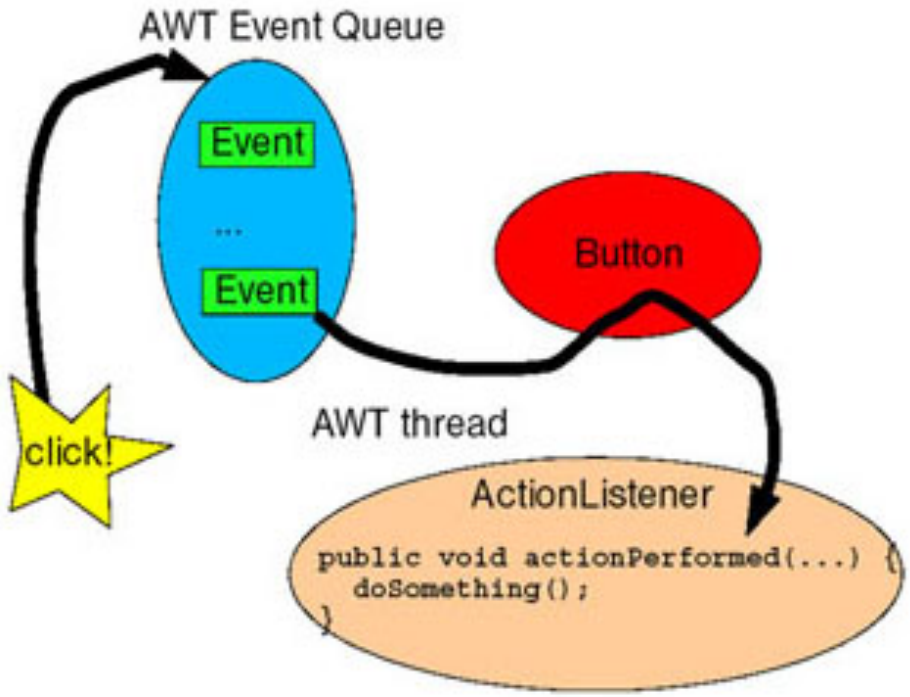
非阻塞服务器模式

- 只需要一个线程就能同时负责接收客户的连接、接收各个客户发送的数据，以及向各个客户发送响应数据

```
while (一直等待，直到有接收连接就绪事件、读就绪事件或写就绪事件发生)        //阻塞
{
    if (有客户连接)
        接受客户的连接；                                                    //非阻塞
     if (某个socket的输入流有可读数据)
        从输入流中读数据；                                                  //非阻塞
    if (某个Socket的输出流可以写数据)
        向输出流写数据；                                                    //非阻塞
}
```

设计背后的基石

- 反应器模式，REACTOR pattern ，类似于 Observer 模式
  - 用于事件多路分离和分派的体系结构模式。
  - 反应器模式的核心功能
    - 事件多路分用
    - 将事件分派到各自相应的事件处理程序
- Reactor模式类似于AWT中的Event处理
  - 事件触发机制是最好的解决办法，当有事件发生时，会触动handler，然后开始数据的处理。



题外话

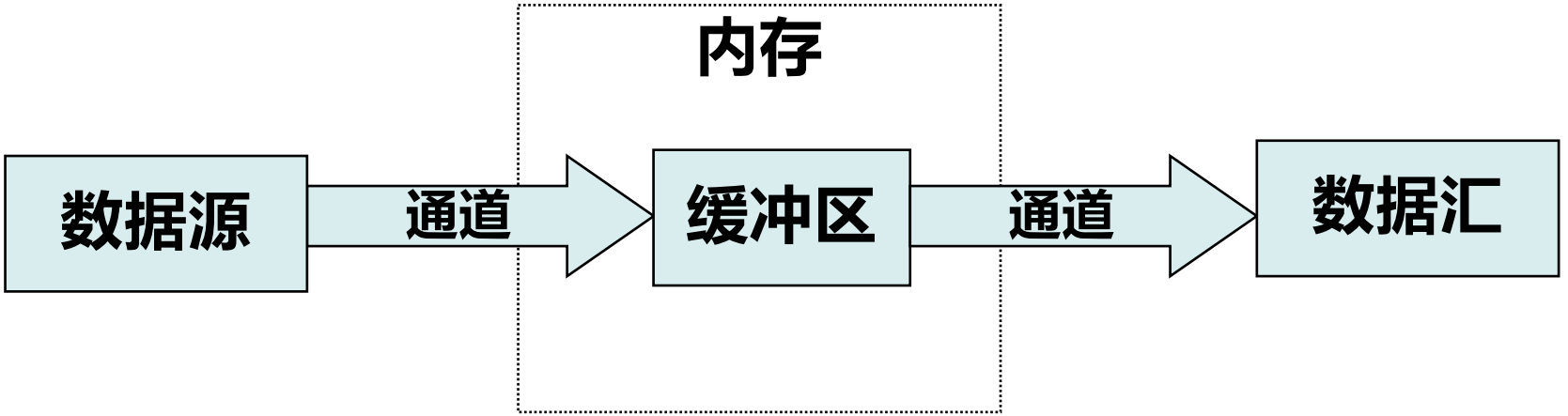
- 设计模式
  - 偏向于架构的编程模式，使用什么样的结构和用法来完成功能
  - 了解模式的用意、结构，以及这一模式适合于什么样的情况等
  - 推荐书籍
    - Java与模式 —— 阎宏

Java非阻塞通信支持

- java.nio包提供了支持非阻塞通信的类
  - ServerSocketChannel: 替代ServerSocket，支持阻塞通信和非阻塞通信；
  - SocketChannel: 替代Socket，支持阻塞通信和非阻塞通信
  - Selector: 为ServerSocketChannel监控接收连接就绪事件，为SocketChannel监控连接就绪、读就绪和写就绪事件
  - SelectionKey: 代表ServerSocketChannel及SocketChannel向Selector注册事件的句柄。当一个SelectionKey对象位于Selector对象的selected-keys集合中时，就表示与这个SelectionKey对象相关的事件发生了。

Channel（通道）

- 用来连接缓冲区与数据源/汇
- 支持非阻塞读写



- NIO 的非阻塞 I/O 机制是围绕 选择器和通道构建的。
  - Channel 类表示服务器和客户机之间的一种通信机制。
  - Selector 类是 Channel 的事件多路分离器(Event Demultiplexer) ，将传入客户机请求多路分用并将它们分派到各自的请求处理程序。
- NIO是一个基于事件的IO架构，最基本的思想就是：
  - 有事件我通知你，你再去做你的事情．而且NIO的主线程只有一个，不像传统的模型，需要多个线程以应对客户端请求，也减轻了JVM的工作量。
  - 当Channel注册至Selector以后，经典的调用方法如下：

```
while (somecondition) {
    int n = selector.select(TIMEOUT);  //取得时间通知
    if (n == 0)
        continue;     //如果大于零，即有事件发生
    for (Iterator iter = selector.selectedKeys().iterator(); iter
        .hasNext();) {  //依次取得事件，并判断各事件类型，根据key中的表示事件，来做相应
的处理
            if (key.isAcceptable())
                doAcceptable(key);
            if (key.isConnectable())
                doConnectable(key);
            if (key.isValid() && key.isReadable())
                doReadable(key);
            if (key.isValid() && key.isWritable())
                doWritable(key);
            iter.remove();
    } }
```

- 异步socket的核心，即异步socket不过是将多个socket的调度（或者还有他们的线程调度）全部交给操作系统自己去完成。Selector，不过是将这些调度收集、分发而已。

SelectableChannel

- 支持阻塞I/O和非阻塞I/O的通道
- 方法
  - 设置为阻塞/非阻塞：

    configureBlocking(boolean block)
  - 注册事件

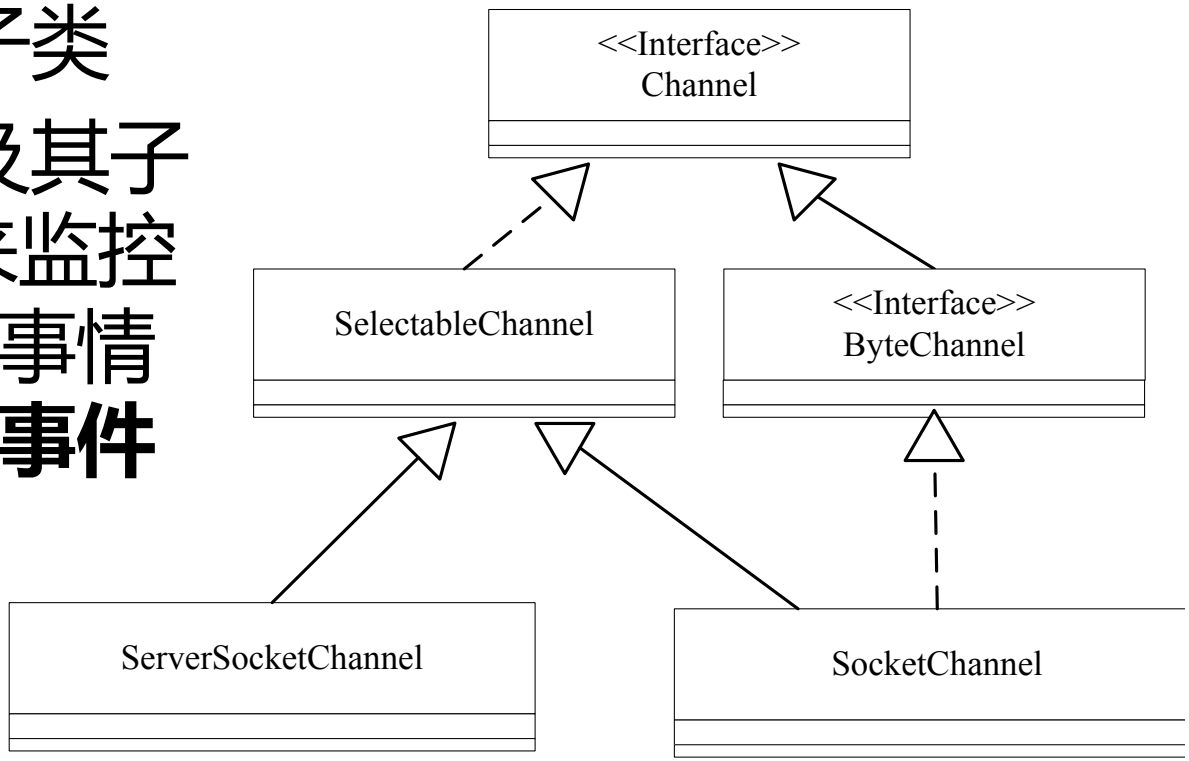
    register(Selector sel, int ops)

    register(Selector sel, int ops, Object attachment)

  返回一个SelectionKey对象，用来跟踪被注册的事件；attachment用于为SelectionKey关联一个附件，从中可以得到包含处理与这个事件有关的信息

SelectableChannel

- ServerSocketChannel与SocketChannel都是SelectableChannel的子类
- SelectableChannel类及其子类都能委托Selector来监控它们可能发生的一些事情，这个过程称为**注册事件过程。**



```
MyHandler handler = new MyHandler();
SelectionKey key=socketChannel.register(selector,
    SelectionKey.OP_READ|SelectionKey.OP_WRITE,
    handler);
```

等价于

```
SelectionKey key=socketChannel.register(selector,
    SelectionKey.OP_READ|SelectionKey.OP_WRITE);
MyHandler handler = new MyHandler();
key.attatch(handler);
```

ServerSocketChannel

- 除了继承的，主要是

SocketChannel accept(); 阻塞或非阻塞，返回的SocketChannel对象是处于阻塞模式的

ServerSocketChannel open()： 静态方法创建对象

ServerScoket socket(): 返回关联的ServerSocket对象

SocketChannel

静态方法 SocketChannel open() //返回阻塞模式的SocketChannel

Socket socket() //返回关联的Socket对象

boolean isConnectionPending()

boolean connect()

boolean finishConnect()

int read(ByteBuffer dst)

int write(ByteBuffer src)

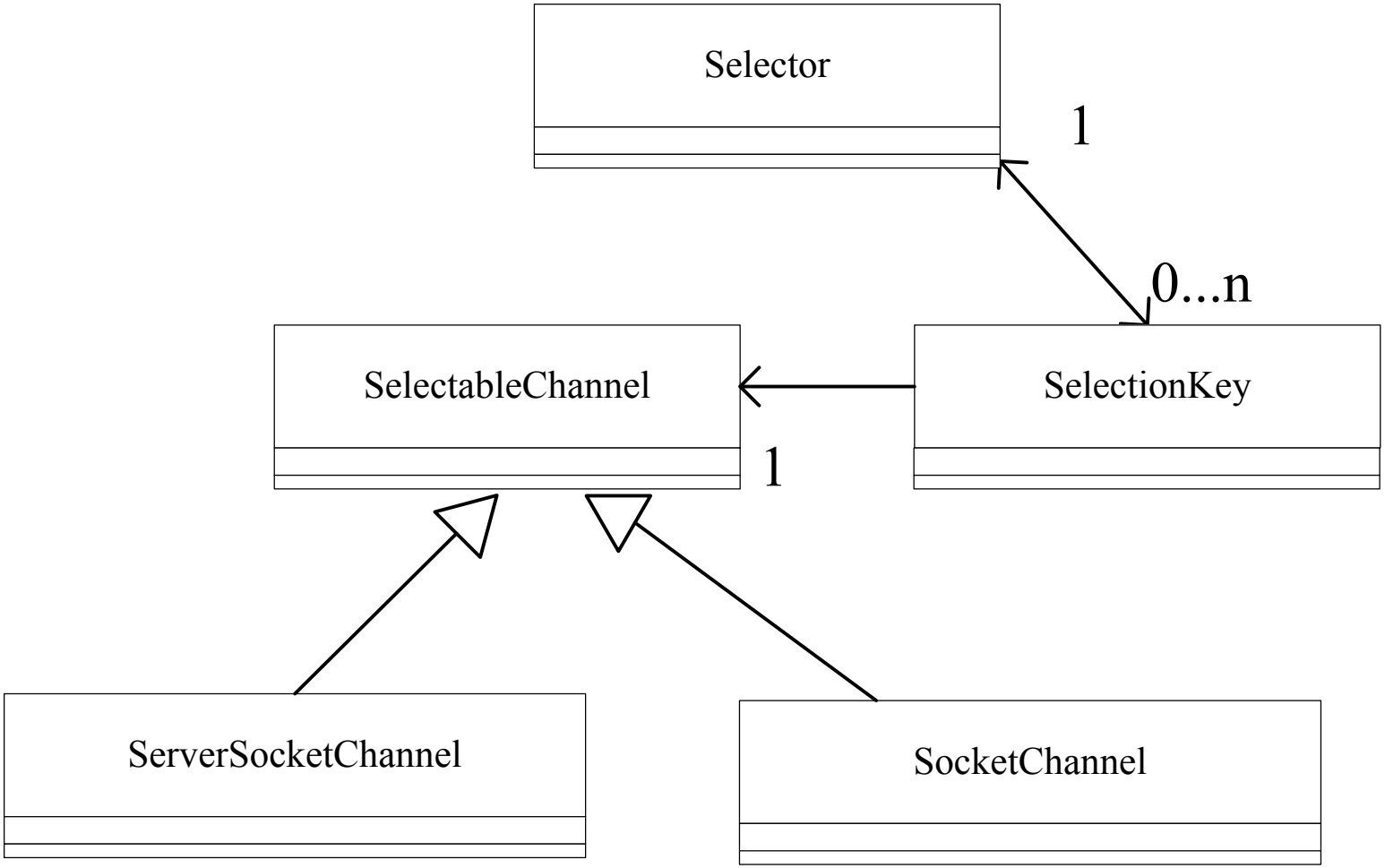
Selector类

- all-keys集合
  - 子集：selected-keys集合, cancelled-keys集合
- 主要参见api参考
- int select()
- int select(long timeout)
- Selector wakeup()
- void close()

SelctionKey类

- 在对象有效期间，会一直监控与SelectionKey对象相关的事件
- 失效：
  - 程序调用SelectionKey的cancel()方法；
  - 关闭与SelectionKey关联的Channel；
  - 与Selectionkey关联的Selector被关闭
- 事件
  - ServerSocketChannel的事件：
    - SelectionKey.OP_ACCEPT: 接收连接就绪 ：常量16
  - SocketChannel的事件：
    - SelectionKey.OP_CONNECT：连接就绪事件 ：常量8
    - SelectionKey.OP_READ：读就绪事件 ： 常量1
    - SelectionKey.OP_WRITE：写就绪事件 ： 常量4

SelectionKey、Selector与SelectableChannel之间的关联关系



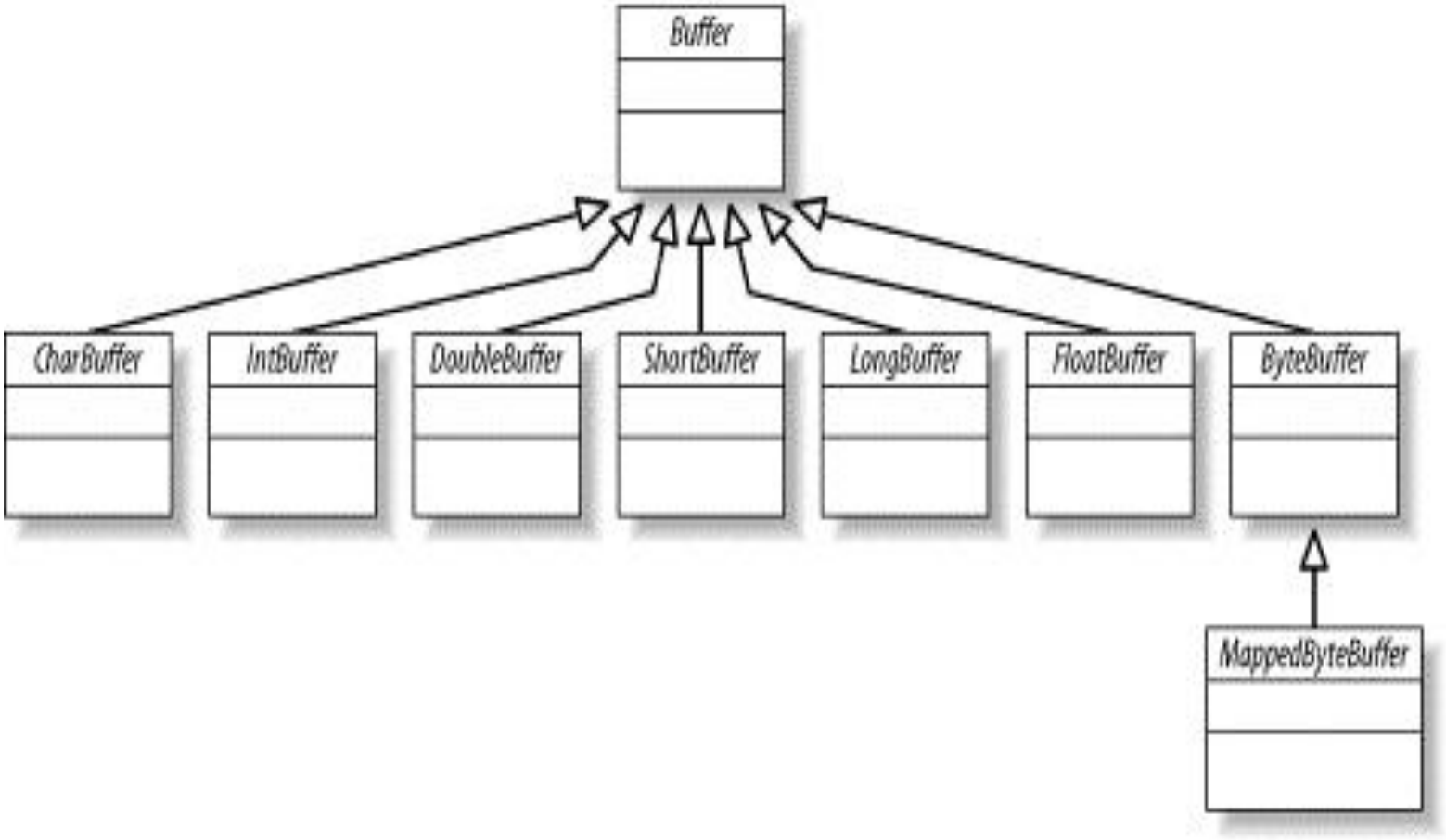
还需关注的（1）——Buffer

- 缓冲区Buffer的使用
  - 两个方面提高IO操作的效率（减少实际的物理读写次数；减少动态分配和回收内存区域的次数）
  - Java.nio包公开buffer类的API
- buffer的几个属性（都是非负值）
  - 容量（capacity）：表示该缓冲区可以保存多少数据
  - 极限（limit）：表示缓冲区目前的使用终点，使缓冲区便于重用
  - 位置（position）：表示缓冲区中下一个读写单元的位置
  - 容量>=极限>=位置>=0
- 改变3个属性的方法
  - clear(): 把极限设为容量，把位置设为0；
  - flip(): 把极限设为位置，再把位置设为0；
  - rewind(): 不改变极限，把位置设为0；
  - remaining(): 返回缓冲区的剩余容量，取值等于limit-position
  - compact(): 删除0到位置的内容，然后把当前位置到极限limit的内容复制到0到limit-positon的区域内。当前位置position和极限limit的取值也作相应的变化

ByteBuffer实例的方法

ByteBuffer方法	准备Buffer以实现	结果值	
		Position	Limit
ByteBuffer clear()	将数据read()/put() 进缓冲区	0	*capacity*
ByteBuffer flip()	从缓冲区write()/get()	0	*position*
ByteBuffer rewind()	从缓冲区rewrite()/get()	0	unchanged

Buffer类的层次结构



需要关注的(2)

- 字符编码Charset
  - java.nio.Charset
  - 解码：字符编码→字符串

    Charbuffer decode(ByteBuffer bb)
  - 编码：字符串→字符编码

    ByteBuffer encode(String str);

    ByteBuffer encode(CharBuffer cb);
  - 静态工厂方法（无构造函数）

    Charset.forName(String encode);

    Charset charset=Charset.forName(”GBK”);

    Charset.defaultCharset();

编程范例

- 服务器-阻塞+非阻塞
  - EchoServer
- 客户端-阻塞+非阻塞
  - EchoClient

- 小小的技巧，执行写操作之前，请取消掉你的写注册，否则你的cpu肯定是100%。

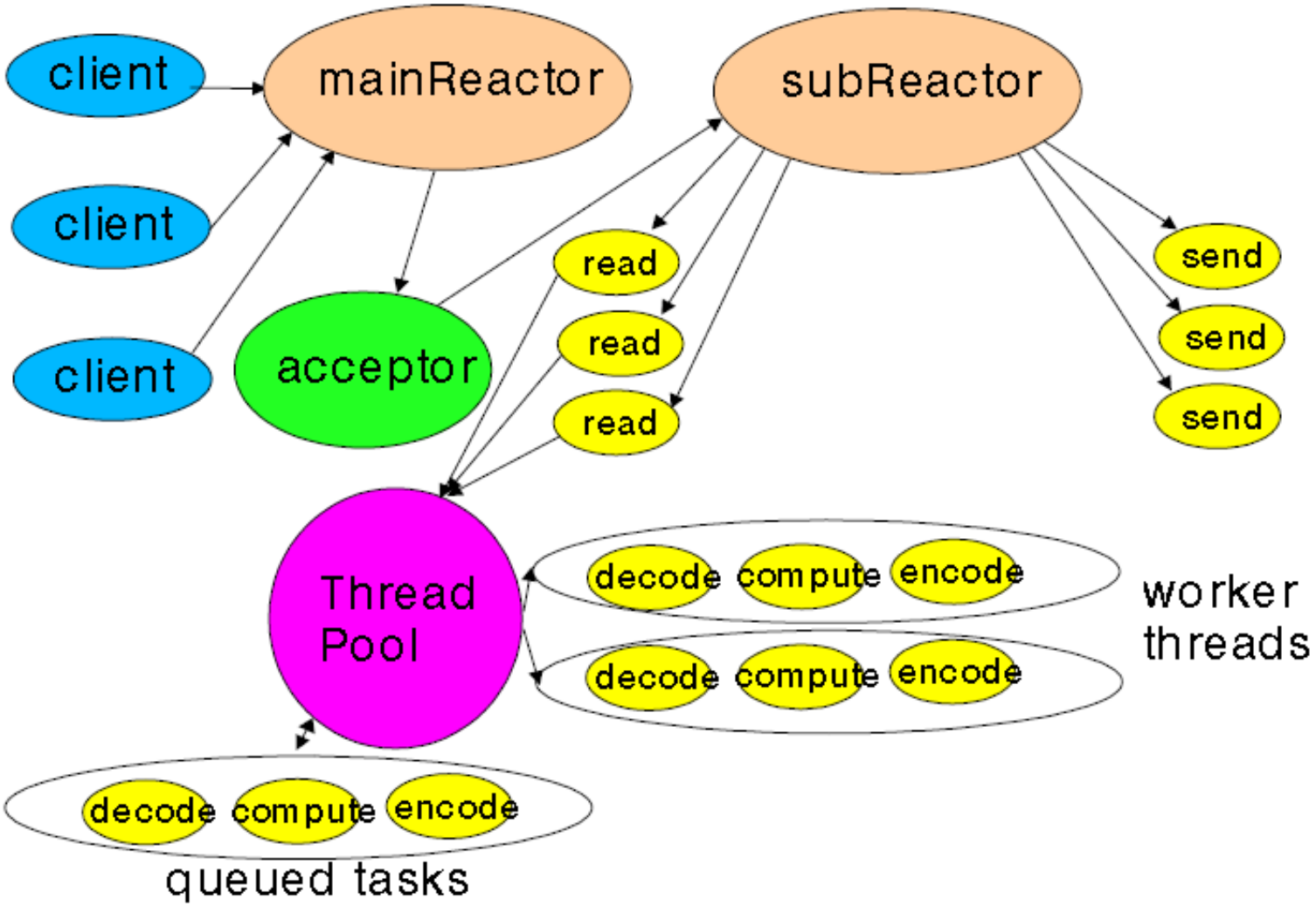
同步通信 vs 异步通信

- 客户与服务器之间的通信，按照它们收发数据的协调程度来区分，
  - 同步通信：甲方向乙方发送了一批数据后，必须等接收到了乙方的响应数据，再发送下一批数据
  - 异步通信：发送数据和接收数据互不干扰，各自独立进行
  - 不要求两端都得采用同样的通信方式
- NIO→AIO（java 7.0）

Echo小结

- 第一次出现
  - 同步、阻塞、单线程
- 第二次出现多线程
  - 对每个客户连接分配一个线程、同步、阻塞
- 第三次出现多线程池
  - 线程池、同步、阻塞
- 第四次出现
  - Channel模式的线程、同步、阻塞
- 第五次出现
  - Channel、异步、非阻塞、单线程
- 第六次出现
  - Channel、异步、阻塞+非阻塞、两个线程

优化方案模式——Using Multiple Reactors



有用的思考和尝试

- 建议去动手尝试！
  - java.nio+线程池, 如何处理死连接（即恶意的只是连接上，却不继续做什么后续操作，占据资源）
  - 结合多线程以及非阻塞，仔细考虑围绕Selector.select()阻塞引发死锁问题如何解决——一种设计模式。
- nio很通用。目前比较流行的框架，参考开源的netty、mina、grizzly
  - Netty: http://netty.io/
  - Mina: http://mina.apache.org/
  - grizzly: https://grizzly.dev.java.net

非阻塞连接多服务器的客户端

- 当客户程序需要与多个服务器建立连接（或者服务器同时建立多个连接），可采用非阻塞模式来建立。

  PingClient.java
  - 定义了两个外部类：Target 和 PingClient
  - PingClient有两个内部类：Connector 和 Printer
  - 三个线程：主线程、两个内部类的线程
  - 典型的生产者-消费者的编程模式

The End