http://wiki.jikexueyuan.com/project/java-nio-zh/java-nio-selector.html

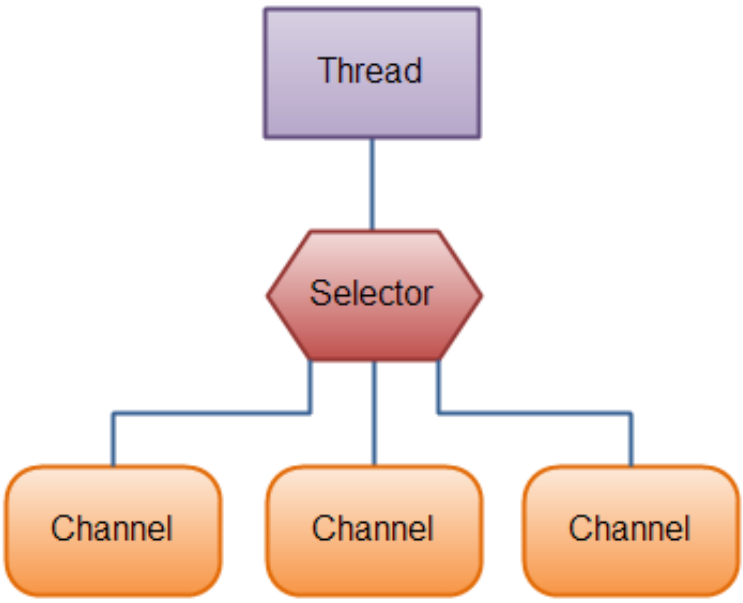
Selector是Java NIO中的一个组件，用于检查一个或多个NIO Channel的状态是否处于可读、可写。如此可以实现单线程管理多个channels,也就是可以管理多个网络链接。

为什么使用Selector（Why Use a Selector?）

用单线程处理多个channels的好处是我需要更少的线程来处理channel。实际上，你甚至可以用一个线程来处理所有的channels。从操作系统的角度来看，切换线程开销是比较昂贵的，并且每个线程都需要占用系统资源，因此暂用线程越少越好。

需要留意的是，现代操作系统和CPU在多任务处理上已经变得越来越好，所以多线程带来的影响也越来越小。如果一个CPU是多核的，如果不执行多任务反而是浪费了机器的性能。不过这些设计讨论是另外的话题了。简而言之，通过Selector我们可以实现单线程操作多个channel。

这有一幅示意图，描述了单线程处理三个channel的情况：



Java NIO: A Thread uses a Selector to handle 3 Channel's

一、 创建Selector(Creating a Selector)

创建一个Selector可以通过Selector.open()方法：

```
Selector selector = Selector.open();
```

二、 注册Channel到Selector上(Registering Channels with the Selector)

为了同Selector挂了Channel，我们必须先把Channel注册到Selector上，这个操作使用SelectableChannel。register()：

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector,
SelectionKey.OP_READ);
```

注意register的第二个参数，这个参数是一个“关注集合”，代表我们关注的channel状态，有四种基础类型可供监听：

1. Connect
2. Accept
3. Read
4. Write

一个channel触发了一个事件也可视作该事件处于就绪状态。因此当channel与server连接成功后，那么就是“连接就绪”状态。server channel接收请求连接时处于“可连接就绪”状态。channel有数据可读时处于“读就绪”状态。channel可以进行数据写入时处于“写就绪”状态。

上述的四种就绪状态用SelectionKey中的常量表示如下：

1. SelectionKey.OP_CONNECT
2. SelectionKey.OP_ACCEPT
3. SelectionKey.OP_READ
4. SelectionKey.OP_WRITE

如果对多个事件感兴趣可利用位的或运算结合多个常量，比如：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

三、SelectionKey's

在上一小节中，我们利用register方法把Channel注册到了Selectors上，这个方法的返回值是SelectionKeys，这个返回的对象包含了一些比较有价值的属性：

- The interest set

- The ready set
- The Channel
- The Selector
- An attached object (optional)

这5个属性都代表什么含义呢？下面会一一介绍。

1. Interest Set

这个"关注集合"实际上就是我们希望处理的事件的集合，它的值就是注册时传入的参数，我们可以用按为与运算把每个事件取出来：

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept = interestSet &
SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet &
SelectionKey.OP_CONNECT;
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

2. Ready Set

"就绪集合"中的值是当前channel处于就绪的值，一般来说在调用了select方法后都会需要用到就绪状态，select的介绍在后续文章中继续展开。

```
int readySet = selectionKey.readyOps();
```

从"就绪集合"中取值的操作类似月"关注集合"的操作，当然还有更简单的方法，SelectionKey提供了一系列返回值为boolean的的方法：

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

3. Channel + Selector

从SelectionKey操作Channel和Selector非常简单：

```
Channel  channel  = selectionKey.channel();
Selector selector = selectionKey.selector();
```

4. Attaching Objects

我们可以给一个SelectionKey附加一个Object，这样做一方面可以方便我们识别某个特定的channel，同时也增加了channel相关的附加信息。例如，可以把用于

channel的buffer附加到SelectionKey上：

```
selectionKey.attach(theObject);
Object attachedObj = selectionKey.attachment();
```

附加对象的操作也可以在register的时候就执行：

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ,
theObject);
```

5. 从Selector中选择channel(Selecting Channels via a Selector)

一旦我们向Selector注册了一个或多个channel后，就可以调用select来获取channel。select方法会返回所有处于就绪状态的channel。 select方法具体如下：

- int select()
- int select(long timeout)
- int selectNow()

select()方法在返回channel之前处于阻塞状态。 select(long timeout)和select做的事一样，不过他的阻塞有一个超时限制。

selectNow()不会阻塞，根据当前状态立刻返回合适的channel。

select()方法的返回值是一个int整型，代表有多少channel处于就绪了。也就是自上一次select后有多少channel进入就绪。举例来说，假设第一次调用select时正好有一个channel就绪，那么返回值是1，并且对这个channel做任何处理，接着再次调用select，此时恰好又有一个新的channel就绪，那么返回值还是1，现在我们一共有两个channel处于就绪，但是在每次调用select时只有一个channel是就绪的。

6. selectedKeys()

在调用select并返回了有channel就绪之后，可以通过选中的key集合来获取channel，这个操作通过调用selectedKeys()方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

还记得在register时的操作吧，我们register后的返回值就是SelectionKey实例，也就是我们现在通过selectedKeys()方法所返回的SelectionKey。

遍历这些SelectionKey可以通过如下方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
```

```
            // a connection was established with a remote server.
        } else if (key.isReadable()) {
            // a channel is ready for reading
        } else if (key.isWritable()) {
            // a channel is ready for writing
        }
        keyIterator.remove();
    }
```

上述循环会迭代key集合，针对每个key我们单独判断他是处于何种就绪状态。注意keyIterater.remove()方法的调用，Selector本身并不会移除SelectionKey对象，这个操作需要我们收到执行。当下次channel处于就绪是，Selector任然会把这些key再次加入进来。

SelectionKey.channel返回的channel实例需要强转为我们实际使用的具体的channel类型，例如ServerSocketChannel或SocketChannel.

7. wakeUp()

由于调用select而被阻塞的线程，可以通过调用Selector.wakeup()来唤醒即便此时已然没有channel处于就绪状态。具体操作是，在另外一个线程调用wakeup，被阻塞与select方法的线程就会立刻返回。

8. close()

当操作Selector完毕后，需要调用close方法。close的调用会关闭Selector并使相关的SelectionKey都无效。channel本身不管被关闭。

四、完整的Selector案例(Full Selector Example)

这有一个完整的案例，首先打开一个Selector,然后注册channel，最后监听Selector的状态：

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector,
SelectionKey.OP_READ);
while(true) {
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;
  Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

```
  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
  while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }
    keyIterator.remove();
  }
}
```