- NETTY

精尽 Netty 源码分析 —— NIO 基础（四）之 Selector

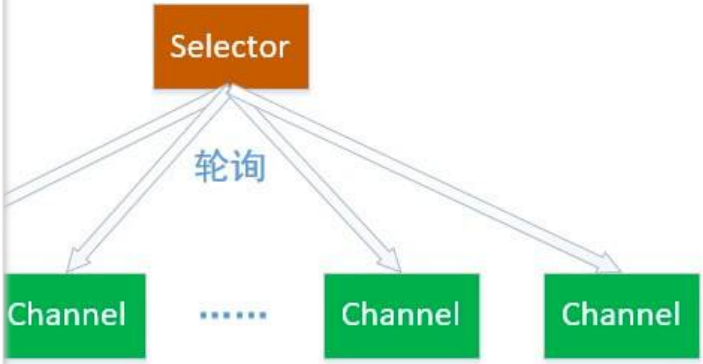
1. 概述

Selector ， 一般称为**选择器**。它是 Java NIO 核心组件中的一个，用于轮询一个或多个 NIO Channel 的状态是否处于可读、可写。如此，一个线程就可以管理多个 Channel ， 也就说可以管理多个网络连接。也因此，Selector 也被称为**多路复用器**。

那么 Selector 是如何轮询的呢?

- 首先，需要将 Channel 注册到 Selector 中，这样 Selector 才知道哪些 Channel 是它需要管理的。
- 其上的 Channel 。如果某个 Channel 上面发生了读或者写事件，这个 Channel 就来，然后通过 SelectionKey 可以获取就绪 Channel 的集合，进行后续的 I/O 操

的示例：



文章目录

2. 优缺点

① **优点**

使用一个线程**能够**处理多个 Channel 的优点是，只需要更少的线程来处理 Channel 。事实上，可以使用一个线程处理所有的 Channel 。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系统的一些资源(例如 CPU、内存)。因此，使用的线程越少越好。

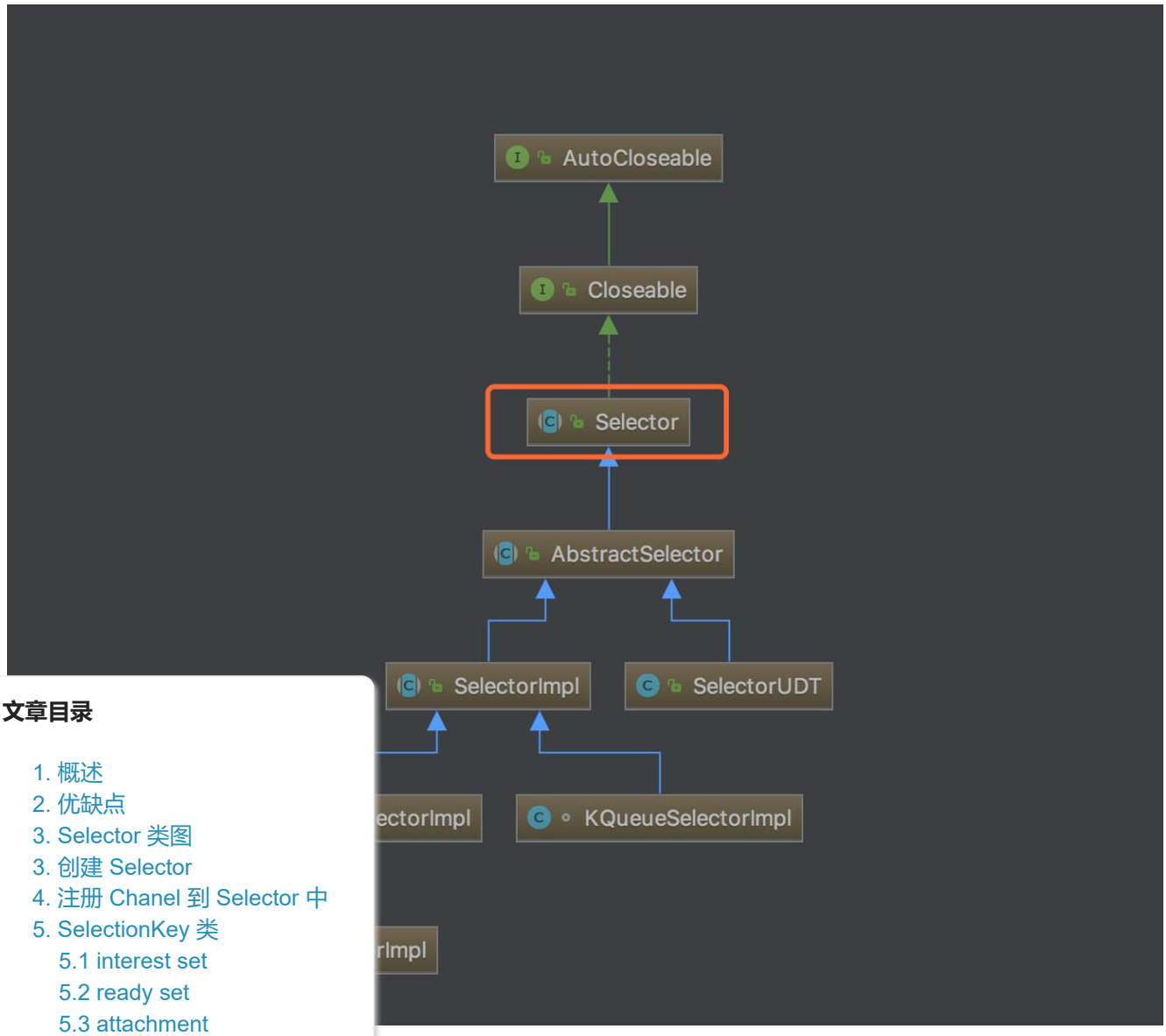
② **缺点**

因为在一个线程中使用了多个 Channel ， 因此会造成每个 Channel 处理效率的降低。

当然，Netty 在设计实现上，通过 n 个线程处理多个 Channel ， 从而很好的解决了这样的缺点。其中，n 的指的是有限的线程数，默认情况下为 CPU * 2 。

3. Selector 类图

Selector 在 `java.nio` 包中，被定义成**抽象类**，整体实现类图如下：



文章目录

兴趣的胖友可以看看占小狼的 《深入浅出NIO之Selector实现原理》 。

个 Selector 对象。代码如下：

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

4. 注册 Chanel 到 Selector 中

为了让 Selector 能够管理 Channel ，我们需要将 Channel 注册到 Selector 中。代码如下：

```
channel.configureBlocking(false); // <1>
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

- **注意**，如果一个 Channel 要注册到 Selector 中，那么该 Channel 必须是**非阻塞**，所以 `<1>` 处的 `channel.configureBlocking(false);` 代码块。也因此，FileChannel 是不能够注册到 Channel 中的，因为它是**阻**

塞的。

- 在 `#register(Selector selector, int interestSet)` 方法的**第二个参数**，表示一个“interest 集合”，意思是通过 Selector 监听 Channel 时，对**哪些**(可以是多个)事件感兴趣。可以监听四种不同类型的事件：
  - Connect ： 连接完成事件(TCP 连接)，仅适用于客户端，对应 `SelectionKey.OP_CONNECT` 。
  - Accept ： 接受新连接事件，仅适用于服务端，对应 `SelectionKey.OP_ACCEPT` 。
  - Read ： 读事件，适用于两端，对应 `SelectionKey.OP_READ` ，表示 Buffer 可读。
  - Write ： 写时间，适用于两端，对应 `SelectionKey.OP_WRITE` ，表示 Buffer 可写。

  Channel 触发了一个事件，意思是该事件已经就绪：

- 一个 Client Channel Channel 成功连接到另一个服务器，称为“连接就绪”。
- 一个 Server Socket Channel 准备好接收新进入的连接，称为“接收就绪”。
- 一个有数据可读的 Channel ， 可以说是“读就绪”。
- 一个等待写数据的 Channel ， 可以说是“写就绪”。

因为 Selector 可以对 Channel 的**多个**事件感兴趣，所以在我们想要注册 Channel 的多个事件到 Selector 中时，可以使用**或运算** `|` 来组合多个事件。示例代码如下：

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

Channel 感兴趣的事件集合，可以通过再次调用 `#register(Selector selector,` 示例代码如下：

```
tionKey.OP_READ);
tionKey.OP_READ | SelectionKey.OP_WRITE);
```

`electionKey.OP_READ` 事件感兴趣。
`electionKey.OP_READ` 和 `SelectionKey.OP_WRITE)` 事件**都**感兴趣。

`gister(...)` 方法，向 Selector 注册一个 Channel 后，会返回一个 SelectionKey 对
`tionKey` 在 `java.nio.channels` 包下，被定义成一个**抽象类**，表示一个 Channel
内容：

- interest set ： 感兴趣的事件集合。
- ready set ： 就绪的事件集合。
- Channel
- Selector
- attachment ： *可选的*附加对象。

5.1 interest set

通过调用 `#interestOps()` 方法，返回感兴趣的事件集合。示例代码如下：

```
int interestSet = selectionKey.interestOps();

// 判断对哪些事件感兴趣
boolean isInterestedInAccept  = interestSet & SelectionKey.OP_ACCEPT != 0;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT != 0;
```

```
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ != 0;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE != 0;
```

- 其中每个事件 Key 在 SelectionKey 中枚举，通过位(bit) 表示。代码如下：

```
//  SelectionKey.java

public static final int OP_READ = 1 << 0;
public static final int OP_WRITE = 1 << 2;
public static final int OP_CONNECT = 1 << 3;
public static final int OP_ACCEPT = 1 << 4;
```

- 所以，在上述示例的后半段的代码，可以通过与运算 & 来判断是否对指定事件感兴趣。

5.2 ready set

通过调用 #readyOps() 方法，返回就绪的事件集合。示例代码如下：

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

文章目录

经内置了判断事件的方法。代码如下：

```
le() {
EAD) != 0;

le() {
RITE) != 0;

table() {
ONNECT) != 0;
}
public final boolean isAcceptable() {
    return (readyOps() & OP_ACCEPT) != 0;
}
```

5.3 attachment

通过调用 #attach(Object ob) 方法，可以向 SelectionKey 添加附加对象；通过调用 #attachment() 方法，可以获得 SelectionKey 获得附加对象。示例代码如下：

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

又获得在注册时，直接添加附加对象。示例代码如下：

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

6. 通过 Selector 选择 Channel

在 Selector 中，提供三种类型的选择(select)方法，返回当前有感兴趣事件准备就绪的 Channel **数量**：

```
// Selector.java

// 阻塞到至少有一个 Channel 在你注册的事件上就绪了。
public abstract int select() throws IOException;

// 在 `#select()` 方法的基础上，增加超时机制。
public abstract int select(long timeout) throws IOException;

// 和 `#select()` 方法不同，立即返回数量，而不阻塞。
public abstract int selectNow() throws IOException;
```

- 有一点**非常需要注意**：select 方法返回的 `int` 值，表示有多少 Channel 已经就绪。亦即，**自上次调用 select 方法后**
  调用 select 方法，因为有一个 Channel 变成就绪状态则返回了 1 ； 若再次调用
  绪了，它会再次返回1。如果对第一个就绪的 Channel 没有做任何操作，现在就有
  ct 方法调用之间，只有一个 Channel 就绪了，所以才返回 1。

annel

用有一个或更多个 Channel 就绪了，然后可以通过调用Selector 的
择键集(selected key set)”中的**就绪** Channel 。示例代码所示：

```
ectedKeys();
```

要先调用 select 方法，才会添加到“已选择键集(selected key set)”中。否则，我们
是无法获得它们对应的 SelectionKey 们。

择

某个线程调用 `#select()` 方法后，发生阻塞了，即使没有通道已经就绪，也有办法让其从 `#select()` 方法返回。

- 只要让其它线程在第一个线程调用 `select()` 方法的那个 Selector 对象上，调用该 Selector 的 `#wakeup()` 方法，进行唤醒该 Selector 即可。
- 那么，阻塞在 `#select()` 方法上的线程，会立马返回。

> Selector 的 `#select(long timeout)` 方法，若未超时的情况下，也可以满足上述方式。

注意，如果有其它线程调用了 `#wakeup()` 方法，但当前没有线程阻塞在 `#select()` 方法上，下个调用 `#select()` 方法的线程会立即被唤醒。😈 有点神奇。

9. 关闭 Selector

当我们不再使用 Selector 时，可以调用 Selector 的 `#close()` 方法，将它进行关闭。

- Selector 相关的所有 SelectionKey 都**会失效**。
- Selector 相关的所有 Channel 并**不会关闭**。

注意，此时若有线程阻塞在 `#select()` 方法上，也会被唤醒返回。

10. 简单 Selector 示例

如下是一个简单的 Selector 示例，创建一个 Selector ，并将一个 Channel注册到这个 Selector上(Channel 的初始化过程略去)，然后持续轮询这个 Selector 的四种事件(接受，连接，读，写)是否就绪。代码如下：

> 老艿艿：本代码取自《Java NIO系列教程（六） Selector》提供的示例，实际代码。🙂 最佳的实践，我们将在 Netty 中看到。


```
n();

;
ter(selector, SelectionKey.OP_READ);

nel
ctor.select();
{


Set selectedKeys = selector.selectedKeys();
// 遍历 SelectionKey 数组
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
        }
        // 移除
```

```
                keyIterator.remove(); // <1>
            }
        }
```

- **注意**, 在每次迭代时, 我们都调用 `keyIterator.remove()` 代码块，将这个 key 从迭代器中删除。
  - 因为 `#select()` 方法仅仅是简单地将就绪的 Channel 对应的 SelectionKey 放到 selected keys 集合中。
  - 因此，如果我们从 selected keys 集合中，获取到一个 key ，但是没有将它删除，那么下一次 `#select` 时, 这个 SelectionKey 还在 selectedKeys 中.

666. 彩蛋

参考文章如下：

- 《Java NIO系列教程（六） Selector》
- 《Java NIO之Selector（选择器）》
- 《Java NIO 的前生今世 之四 NIO Selector 详解》

问量 次