Java NIO Selector

Jakob Jenkov
Last update: 2014-06-23

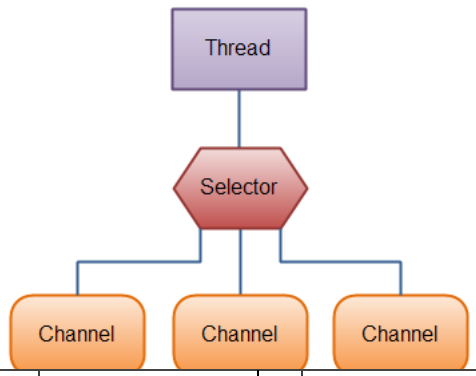
A `Selector` is a Java NIO component which can examine one or more NIO Channel's, and determine which channels are ready for e.g. reading or writing. This way a single thread can manage multiple channels, and thus multiple network connections.

Why Use a Selector?

The advantage of using just a single thread to handle multiple channels is that you need less threads handle the channels. Actually, you can use just one thread to handle all of your channels. Switching between threads is expensive for an operating system, and each thread takes up some resources (memory) in the operating system too. Therefore, the less threads you use, the better.

Keep in mind though, that modern operating systems and CPU's become better and better at multitas so the overheads of multithreading becomes smaller over time. In fact, if a CPU has multiple cores, yc might be wasting CPU power by **not** multitasking. Anyways, that design discussion belongs in a differ text. It suffices to say here, that you can handle multiple channels with a single thread, using a `Selecto`

Here is an illustration of a thread using a `Selector` to handle 3 `Channel`'s:



Creating a Selector

You create a `Selector` by calling the `Selector.open()` method, like this:

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

Registering Channels with the Selector

In order to use a `Channel` with a `Selector` you must register the `Channel` with the `Selector`. This is done u the `SelectableChannel.register()` method, like this:

```
channel.configureBlocking(false);

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

The `Channel` must be in non-blocking mode to be used with a `Selector`. This means that you cannot use `FileChannel`'s with a `Selector` since `FileChannel`'s cannot be switched into non-blocking mode. Socket channels will work fine though.

Notice the second parameter of the `register()` method. This is an "interest set", meaning what events are interested in listening for in the `Channel`, via the `Selector`. There are four different events you can lis for:

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

A channel that "fires an event" is also said to be "ready" for that event. So, a channel that has connect successfully to another server is "connect ready". A server socket channel which accepts an incoming connection is "accept" ready. A channel that has data ready to be read is "read" ready. A channel that ready for you to write data to it, is "write" ready.

These four events are represented by the four `SelectionKey` constants:

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

If you are interested in more than one event, OR the constants together, like this:

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

I'll return to the interest set a bit further down in this text.

SelectionKey's

As you saw in the previous section, when you register a `Channel` with a `Selector` the `register()` method returns a `SelectionKey` objects. This `SelectionKey` object contains a few interesting properties:

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

I'll describe these properties below.

Interest Set

The interest set is the set of events you are interested in "selecting", as described in the section "Registering Channels with the Selector". You can read and write that interest set via the `SelectionKey` this:

```
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;
```

As you can see, you can AND the interest set with the given `SelectionKey` constant to find out if a certa

Ready Set

The ready set is the set of operations the channel is ready for. You will primarily be accessing the reac set after a selection. Selection is explained in a later section. You access the ready set like this:

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

You can test in the same way as with the interest set, what events / operations the channel is ready fo But, you can also use these four methods instead, which all reaturn a boolean:

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

Channel + Selector

Accessing the channel + selector from the `SelectionKey` is trivial. Here is how it's done:

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

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

Attaching Objects

You can attach an object to a `SelectionKey` this is a handy way of recognizing a given channel, or attac further information to the channel. For instance, you may attach the `Buffer` you are using with the char or an object containing more aggregate data. Here is how you attach objects:

```
selectionKey.attach(theObject);

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

You can also attach an object already while registering the `Channel` with the `Selector`, in the `register()` method. Here is how that looks:

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

Selecting Channels via a Selector

Once you have register one or more channels with a `Selector` you can call one of the `select()` method These methods return the channels that are "ready" for the events you are interested in (connect, acc read or write). In other words, if you are interested in channels that are ready for reading, you will rece the channels that are ready for reading from the `select()` methods.

Here are the `select()` methods:

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

`select()` blocks until at least one channel is ready for the events you registered for.

`select(long timeout)` does the same as `select()` except it blocks for a maximum of `timeout` milliseconds (the parameter).

`selectNow()` doesn't block at all. It returns immediately with whatever channels are ready.

The `int` returned by the `select()` methods tells how many channels are ready. That is, how many char that became ready since last time you called `select()`. If you call `select()` and it returns 1 because one channel has become ready, and you call `select()` one more time, and one more channel has become ready, it will return 1 again. If you have done nothing with the first channel that was ready, you now ha ready channels, but only one channel had become ready between each `select()` call.

selectedKeys()

Once you have called one of the `select()` methods and its return value has indicated that one or more channels are ready, you can access the ready channels via the "selected key set", by calling the selec `selectedKeys()` method. Here is how that looks:

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

When you register a channel with a `Selector` the `Channel.register()` method returns a `SelectionKey` obje This key represents that channels registration with that selector. It is these keys you can access via th

```
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();
}
```

This loop iterates the keys in the selected key set. For each key it tests the key to determine what the channel referenced by the key is ready for.

Notice the `keyIterator.remove()` call at the end of each iteration. The `Selector` does not remove the `SelectionKey` instances from the selected key set itself. You have to do this, when you are done proces the channel. The next time the channel becomes "ready" the `Selector` will add it to the selected key se again.

The channel returned by the `SelectionKey.channel()` method should be cast to the channel you need tc work with, e.g a ServerSocketChannel or SocketChannel etc.

wakeUp()

A thread that has called the `select()` method which is blocked, can be made to leave the `select()` met even if no channels are yet ready. This is done by having a different thread call the `Selector.wakeup()` method on the `Selector` which the first thread has called `select()` on. The thread waiting inside `select(` then return immediately.

If a different thread calls `wakeup()` and no thread is currently blocked inside `select()`, the next thread th calls `select()` will "wake up" immediately.

close()

When you are finished with the `Selector` you call its `close()` method. This closes the `Selector` and invalidates all `SelectionKey` instances registered with this `Selector`. The channels themselves are not closed.

Full Selector Example

Here is a full example which opens a `Selector`, registers a channel with it (the channel instantiation is l out), and keeps monitoring the `Selector` for "readiness" of the four events (accept, connect, read, write

```
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.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
  }
}
```

Next: Java NIO FileChannel

Jakob Jenkov