Introduction to NIO.2 (Asynchronous I/O) and how you can benefit from being asynchronous!

Jeanfrancois Arcand
Senior Staff Engineer
Sun Microsystems

Goal

This session will introduce NIO.2 API and Concepts and demonstrate how Apache Projects like Tomcat can take advantage of the new API, both in term of code clarity and performance.

Agenda

- Introduction to Asynchronous I/O
- The Frightening Thread Pool
- Bye Bye Selector!
- The Evil AsynchronousSocketChannel.read()
- The Mysterious AsynchronousSocketChannel.write()
- Damned ByteBuffer
- New File I/O (finally)
- I’m Watching You - WatchService
- Conclusion

Introduction to AIO

- NIO.1 -> Get events through a Selector when there is some I/O ready to be processed, like read and write operations. As soon as you get notified that an event is ready to be processed, execute an operation:

```
If (key.isReadable(){
    channel.read(byteBuffer);
}
```

- Asynchronous I/O send you a notification when the I/O is **completed**. You get the notification ONLY when the operation has completed (or failed).

CompletionHandler

- With AIO, you wait for I/O operations using we a CompletionHandler:
  - completed(...): invoked when the I/O operation completes successfully.
  - failed(...): invoked if the I/O operations fails (like when the remote client close the connection).
  - cancelled(...): invoked when the I/O operation is cancelled by invoking the cancel method on a Future.
- As an example, you specify a completion handler when you want to execute an asynchronous read.

Future

- Almost all I/O operations returns a Future which can be used to monitor the I/O transaction and be used to block for the I/O transaction to complete/times out.

```
Future f =
  asynchronousSocketChannel.connect(…);
// Wait for the connect operation to complete.
f.get(30,TimeUnit.SECONDS);
```

AsynchronousChannelGroup

- A grouping of asynchronous channels for the purpose of resource sharing.
- Encapsulates the mechanics required to handle the completion of I/O operations initiated by asynchronous channels that are bound to the group.
- A group has an associated thread pool to which tasks are submitted to handle I/O events and dispatch to CompletionHandler that consume the result of asynchronous operations

AsynchronousChannelGroup

- In addition to handling I/O events, the pooled threads may also execute other tasks required to support the execution of asynchronous I/O operations.

The Frightening Thread Pool

- With AIO, you can configure the thread pool (ExecutorService) used by both the AIO kernel and your application

  AsynchronousChannelGroup.**withFixedThreadPool**
  (nThread, ThreadFactory)
  AsynchronousChannelGroup.**withCachedThreadPool**
  (ExecutorService, initialSize)
  AsynchronousChannelGroup.**withThreadPool**(ExecutorService)

- … or use the preconfigured/built in Thread Pool that comes by default (no control)

FixedThreadPool

- An asynchronous channel group associated with a fixed thread pool of size N creates N threads that are waiting for already processed I/O events.
- The kernel dispatch event directly to those threads:
    - Thread first complete the I/O operation (like filling a ByteBuffer during a read operation).
    - Next invoke the CompletionHandler.completed() that consumes the result.
    - When the CompletionHandler terminates normally then the thread returns to the thread pool and wait on a next event.

Booo my application freezes!

- What about if all threads "dead lock" inside a CompletionHandler?
    - Bang! your entire application can hangs until one thread becomes free to execute again.
    - The kernel is no longer able to EXECUTE anything!
- Hence this is **critically** important CompletionHandler complete in a timely manner and avoid blocking.
- If all completion handlers are blocked, any new event will be queued until one thread is 'delivered' from the lock.
- So try to avoid blocking operations inside a completion handler.

CachedThreadPool

- An asynchronous channel group associated with a cached thread pool submits events to the thread pool that simply invoke the user's completion handler.
- Internal kernel's I/O operations are handled by one or more internal threads that are not visible to the user application.
- That means you have one hidden thread pool that dispatch events to a cached thread pool, which in turn invokes completion handler
- Wait! you just win a price: a thread's context switch for free!!

CachedThreadPool

- Probability of suffering the hangs problem as with the FixedThreadPool is lower.
- Still might grows infinitely (those infinite thread pool should have never existed anyway!).
- At least you guarantee that the kernel will be able to complete its I/O operations (like reading bytes).
- Oups…CachedThreadPool must support unbounded queueing to works properly (grrr not sure I like that!!!).
- So you can possibly lock all the threads and feed the queue forever until OOM happens. Great!
- **Of course, nobody will ever do that!**

Build-in Kernel Thread Pool

- Hybrid of the above configurations:
    - Cached thread pool that creates threads on demand
    - N threads that dequeue events and dispatch directly to CompletionHandler
- N defaults to the number of hardware threads.
- In addition to N threads, there is one additional internal thread that dequeues events and submits tasks to the thread pool to invoke completion handlers.

Some code

AsynchronousChannelGroup:

org.apache.tomcat.util.net.AIOEndpoint

CompletionHandler

org.apache.tomcat.util.net.AIOEndpoint.SocketProcessor

AsynchronousServerSocketChannel

org.apache.tomcat.util.net.AIOEndpoint.Acceptor

Bye Bye Selector

- With NIO.1, the torture started with the Selector and its associated classes:
    - SelectionKey for determining I/O operations
    - Socket/DatagramChannel for I/O operations
    - Selector waked up when some I/O operations were ready to be executed
    - If OP_ACCEPT else IF OP_READ else IF OP_WRITE else IF OP_CONNECT

The nightmare is over

- Hence in order to perform any I/O operation, you needed to register your SelectionKey to one or more Selector, and wait to get notified once “ready”
- With AIO, no more Selector/SelectionKey. You just use the real objects:
    - AsynchrounousServerSocketChannel.accept(…)
    - AsynchronousSocketChannel.write (…)

Ex: Accepting connection

```
 asyncSocketServer.accept(null,
                          new CompletionHandler<AsynchronousSocketChannel,
Void>(){
                      public void completed(AsynchronousSocketChannel a, Void v) {
                          processConnection(a);
                          asyncSocketServer.accept(null,this);
                      }
                     // Client disconnected
                      public void failed(Throwable t, Void v) {}
                      // Future.cancel()
                      public void cancelled(Void v) {
                      }
                  });
```

The evil AsynchronousSocketChannel.read()

- Once a connection has been accepted, it is now time to read some bytes:

```
AsynchronousSocketChannel.read(ByteBuffer b,
                               Attachment a,
                               CompletionHandler c);
```

Hey Hey → You see the evil, right?

Who remember when I was scared by the
SelectionKey.attach()?

The evil AsynchronousSocketChannel.read()

- Trouble trouble trouble:
  - Let’s say you get 10 000 accepted connections
  - Hence 10 000 ByteBuffer created, and the read operations get invoked
  - Now we are waiting, waiting, waiting, waiting for the remote client(s) to send us bytes (slow clients/ network)
  - Another 10 000 requests comes in, and we are again creating 10 000 ByteBuffer and invoke the read() operations.
  - BOOM OOM!

The evil AsynchronousSocketChannel.read()

- Let’s not be too negative here. So far we have tested with more than 20 000 clients without any issues
- But this is still something you have to keep in mind!!
- Might want to throttle the read() operation to avoid the creation of too many ByteBuffer
- I strongly recommend the use of a ByteBuffer pool, specially if you are using Heap ByteBuffer (more on this later). Get a ByteBuffer before invoking the read() method, and return it to the pool once the read operations complete.

Blocking AsynchronousSocketChannel.read()

- Hein? Blocking? Not like NIO.1 PLEASE!!
- When invoking the read operation, the returned value is a Future:

```
Future readOp =
   AsynchronousSocketChannel.read(…);
readOp.get(30, TimeUnit.SECONDS);
```

- The Thread will blocks until the read operation complete or times out.
- Be careful as you might lock your ThreadPool (specially the fixedThreadPool)

Some code

AsynchronousSocketChannel.read()

org.apache.tomcat.util.net.AIOEndpoint.SocketProcessor

Blocking read()

org.apache.coyote.http11.InternalAioInputBuffer

The mysterious AsynchronousSocketChannel.write()

- Now let's execute some write operations:

  AsynchronousSocketChannel.write(ByteBuffer b,
  Attachement a,
  CompletionHandler c);

- Wait wait wait. Since we are asynchronous, invoking write(..) will not block, so the calling thread can continue its execution.
- What's happen when the calling thread invokes the write method again and the CompletionHandler has not yet been invoked by the previous write call?

The mysterious AsynchronousSocketChannel.write()

- Aille!! You get a WritePendingException
- Hence when invoking the write operation, make sure the CompletionHandler.complete() has been invoked before initiating another write.
- Better, store ByteBuffer inside a queue and execute write operations only when the previous one has completed (will show code soon)
- As for read, I strongly recommend the use of a ByteBuffer pool for executing write operations. Get one before writing, put it back to the pool after.

Some code

AsynchronousSocketChannel.write()

com.sun.grizzly.aio.OutputWriter

Blocking write()

com.sun.grizzly.aio.OutputWriter

Damned ByteBuffer!

- If you are using Heap ByteBuffer, be aware the kernel will copy the bytes into a direct ByteBuffer during every write operation:
    - Free byte copy ☺
- Direct ByteBuffer performance have significantly improved with JDK 7, so use them directly.
- Scattered ByteBuffer write operations still offer you free copy, using direct ByteBuffer or not!

The cool AsynchronousFileChannel.open()

- Before, the nightmare:

```
File f = new File();

FileOutputStream fis = new FileOutputStream(f);

FileChannel fc = fis.getChannel();

fc.write(…);
```

…… typing so much lines hurts ☺

Now let's execute some write operations:

```
AsynchronousFileChannel.open(Path file, Set<? extends OpenOption> options,
    ExecutorService executor, FileAttribute<?>... attrs);

afc.write(…);
```

New File I/O system API

- New packages
  - java.nio.file, java.nio.file.attribute
- Main classes:
  - **FileSystem:**
    - Interface to file system
    - Factory for objects to access files and other objects in file system
  - **FileRef**
    - Reference to file or directory
    - Defines methods to operate on file or directory
  - **Path**
    - A FileRef that locates a file by a system dependent path
    - Created by FileSystem by converting path string or URI
  - **FileStore**
    - Underlying storage pool, device, partition...

I’m watching you - WatchService!

- A watch service that watches registered objects for changes and events.

- A Watchable object is registered with a watch service by invoking its register method, returning a WatchKey (oh no a SelectionKey cousin ;-)) to represent the registration.

- Grrr -> The implementation that observes events from the file system is intended to map directly on to the native file event notification facility where available, **or to use a primitive mechanism, such as polling, when a native facility is not available.**

- Consequently, many of the details on how events are detected, their timeliness, and whether their ordering is preserved are highly **implementation specific.**

Conclusion

Asynchronous I/O significantly improve the way scalable I/O based application can be constructed.

Use AIO **NOW** by downloading JDK 7 (starting @ build 50)

Tomcat AIO implementation soon available (Grizzly already available…)

Q&A

- Open JDK NIO.2 page: http://openjdk.java.net/projects/nio/
- My Tricks and Tips series on AIO http://weblogs.java.net/jfarcand
- AIO docs http://openjdk.java.net/projects/nio/javadoc/
- http://twitter.com/jfarcand