ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

16: Channels

CHANNELS

Per completare la panoramica sui metodi di gestione dei `Socket` ci manca un'astrazione che ci permetta di ascoltare e reagire a più richieste di connessione.

Nella revisione dei metodi di I/O introdotta con il package `java.nio` in Java 1.4 nel 2002, viene introdotto un'intero albero di tipi dedicati alla gestione della comunicazione nel modo più generico.

```
/**
 * A nexus for I/O operations.
 *
 **/
public interface Channel extends Closeable
```

Un `Channel` rappresenta un canale di I/O, che può essere aperto o chiuso.

```
/**
 * A channel to a network socket.
 *
 **/
public interface NetworkChannel extends Closeable
```

Un `NetworkChannel` rappresenta una comunicazione su di una rete. Può:

- esssere legato (con l'operazione `bind`) ad un'indirizzo
- dichiarare le opzioni che supporta.

```
/**
 * An asynchronous channel for stream-oriented
 * listening sockets.
 *
 **/
public abstract class AsynchronousServerSocketChannel
  implements AsynchronousChannel, NetworkChannel
```

Un `AsynchronousServerSocketChannel` è un canale asincrono basato su di una server socket.

Ci permette, in modo asincrono, di accettare connessioni e gestirle.

```
/**
 * A handler for consuming the result of an asynchronous
 * I/O operation.
 *
 * @param V The result type of the I/O operation
 * @param A The type of the object attached to the
 *   I/O operation
 **/
interface CompletionHandler< V,A >
```

`CompletionHandler` è l'interfaccia che deve implementare un oggetto che gestisce la ricezione di un'operazione di I/O asincrona.

```
/**
 * Invoked when an operation has completed.
 *
 * @param result The result of the I/O operation
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void completed(V result, A attachment)
```

Il compito del metodo `completed` è gestire l'interazione relativa ai dati ricevuti, ed eventualmente predisporre l'operazione successiva.

```
/**
 * Invoked when an operation fails.
 *
 * @param exc The exception to indicate why the I/O
 *   operation failed
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void failed(Throwable exc, A attachment)
```

Il compito del metodo `failed` è, ovviamente, gestire il caso in cui un'interazione ha incontrato una eccezione.

Implementando un `CompletionHandler` possiamo esprimere il comportamento del server alla prossima interazione, in maniera asincrona.

Il parametro generico `attachment` ci permette di far circolare le informazioni di contesto riguardo allo stato della conversazione.

I vari handler potrebbero essere chiamati da `Thread` diversi, in momenti imprevedibili; da qui la necessità di gestire esplicitamente il passaggio del contesto.

La gestione delle operazioni di I/O richiede quindi di specificare sempre l'attachment da far circolare ed il `CompletitionHandler` che gestisce il completamento.

```
/**
 * (from AsynchronousServerSocketChannel)
 * Accepts a connection.
 *
 * @param A The type of the attachment
 * @param attachment The object to attach to the I/O
 *   operation; can be null
 * @param handler The handler for consuming the result
 **/
public abstract < A > void accept(A attachment,
  CompletionHandler< AsynchronousSocketChannel,? super A >
  handler)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Reads a sequence of bytes from this channel into the given
 * buffer.
 *
 * @param A The type of the attachment
 * @param dst The buffer into which bytes are to be
 *   transferred
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler
 **/
public final < A > void read(ByteBuffer dst, A attachment,
  CompletionHandler< Integer,? super A > handler)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Writes a sequence of bytes to this channel from the given
 * buffer.
 *
 * @param A The type of the attachment
 * @param src The buffer from which bytes are to be
 *   retrieved
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler object
 **/
public final < A > void write(ByteBuffer src, A attachment,
  CompletionHandler< Integer,? super A > handler)
```

Questo richiede di riorganizzare (pesantemente) il nostro codice, ma ci permette di gestire molte più connessioni.

Un'alternativa all'uso di un `CompletionHandler` è data dalla versione dei metodi che ritorna un `Future`.

```
/**
 * (from AsynchronousServerSocketChannel)
 * Accepts a connection.
 *
 * @return a Future object representing the pending result
 **/
public abstract Future< AsynchronousSocketChannel >
  accept()
```

```
/**
 * (from AsynchronousSocketChannel)
 * Reads a sequence of bytes from this channel into the given
 * buffer.
 *
 * @param dst The buffer into which bytes are to be
 *   transferred
 * @return A Future representing the result of the operation
 **/
public abstract Future< Integer > read(ByteBuffer dst)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Writes a sequence of bytes to this channel from the given
 * buffer.
 *
 * @param src The buffer from which bytes are to be
 *   retrieved
 * @return A Future representing the result of the operation
 **/
public abstract Future< Integer > write(ByteBuffer src)
```

La struttura a cui questo approccio porta è duale alla precedente: il contesto è dato dal blocco in cui viene eseguito gestito il `Future`.

La principale differenza è che in questo caso, se il blocco di codice è unico per tutta la conversazione, il thread che la gestisce è unico e rimane allocato per l'intera durata della conversazione.

ESEMPIO

pcd2018.channels.Server

```
ExecutorService pool = Executors.newFixedThreadPool(4);
AsynchronousChannelGroup group=
  AsynchronousChannelGroup.withThreadPool(pool);
AsynchronousServerSocketChannel serverSocket =
  AsynchronousServerSocketChannel.open()
  .bind(new InetSocketAddress("127.0.0.1", GAME_PORT), 16);

pool.submit(() -> {
  serverSocket.accept(
    new GameAttachment(1, new Game(), serverSocket, group),
    new AcceptPlayer0());
});
```

pcd2018.channels.AcceptPlayerO

```
@Override
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {
  System.out.println(Thread.currentThread().getName() +
    " : game " + attachment.id + " connected player O");
  attachment.server.accept(attachment.playerO(result),
    new WriteFirstStatus());
}
```

pcd2018.channels.WriteFirstStatus

```
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {

  attachment = attachment.playerX(result);
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

```
  // more games?
  if (attachment.id <= 5) {
    attachment.server.accept(new GameAttachment(attachment.id
      new Game(), attachment.server), new AcceptPlayerO());
  } else {
    attachment.group.shutdown();
  }
}
```

pcd2018.channels.ReadPlayer

```
public void completed(Integer result,
    GameAttachment attachment) {
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  attachment.readBuf.clear();
  socket.read(attachment.readBuf, attachment,
    new WriteStatus());
}
```

pcd2018.channels.WriteStatus

```
String input = new String(attachment.readBuf.array(),
  0, result).trim();
Integer move = Integer.parseInt(input);
GameResult initial = attachment.game.status();
GameResult status =
  attachment.game.move(initial.next, move);
```

```
if (!status.end) {
  // the game goes on
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

```
} else if (status.valid) {
  attachment.players[status.next].write(
    wrap("You won.".getBytes()), attachment,
    new CloseSocket(status.next));
  int loser = (status.next + 1) & 0x1;
  attachment.players[loser].write(
    wrap("You lost.".getBytes()), attachment,
    new CloseSocket(loser));
```

```
} else {
  // we have a tie
  attachment.players[0].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(0));
  attachment.players[1].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(1));
}
```

pcd2018.channels.CloseSocket

```
try {
  attachment.players[idx].close();
} catch (IOException e) {
  e.printStackTrace();
}
```