Java Avançado

Manipulação de Dados com a New I/O API

SoftBlue
www.softblue.com.br

Tópicos Abordados

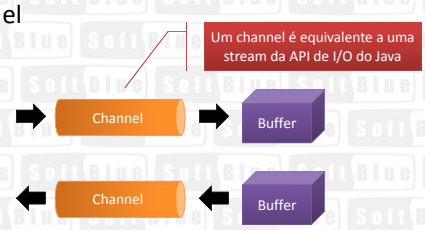
- Introdução à API NIO
- Channels
- Buffers
  - Funcionamento interno de um buffer
- Memory-mapped Files
- I/O Assíncrono
  - Selectors

A API NIO

- A **N**ew **I**/**O** API foi introduzida no Java 1.4
- É uma alternativa à API padrão de I/O do Java
- Possibilita realizar I/O com alta performance sem que haja a necessidade de implementar código nativo
- A diferença principal entre a API padrão e a NIO está na forma de tratamento dos dados
  - A API padrão utiliza *streams*
  - A NIO API utiliza blocos de dados

Channels e Buffers

- São conceitos importantes da NIO API
- As informações trafegam por um **channel**
- O **buffer** coloca/retira as informações no/do channel



Channels

- Diferente das streams, não é possível ler/escrever diretamente de/em um channel
- Um channel é bidirecional
- Tipos importantes de channels:

Classes	Descrição
FileChannel	Lê/escreve dados de/para arquivos
DatagramChannel	Lê/escreve dados via rede com o protocolo UDP
SocketChannel	Lê/escreve dados via rede com o protocolo TCP
ServerSocketChannel	Aceita conexões via protocolo TCP

Buffers

- Um buffer é um bloco de memória de tamanho fixo (array)
- Permite a leitura e escrita de dados
- A classe *Buffer* representa um buffer de dados
- Tipos importantes de buffers:

ByteBuffer	**LongBuffer**
CharBuffer	**FloatBuffer**
ShortBuffer	**DoubleBuffer**
IntBuffer	

Channels e Buffers na Prática

- Um *FileChannel* é obtido através de objetos *FileInputStream*, *FileOutputStream* ou *RandomAccessFile*

```
FileInputStream fis = new FileInputStream("arquivo.txt");
FileChannel channel = fis.getChannel();
```

- Um *Buffer* é criado através dos métodos *allocate()* ou *wrap()*

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

```
byte[] array = new byte[1024];
ByteBuffer buffer = ByteBuffer.wrap(array);
```

Channels e Buffers na Prática

- Escrevendo os dados em um buffer

```
channel.read(buffer);
```

Retorna o número de bytes lidos

- Escrevendo os dados em um channel

```
channel.write(buffer);
```

Retorna o número de bytes lidos

- Lendo/escrevendo dados de/em um buffer

```
buffer.get();
buffer.put(data);
```

Este métodos possuem diferentes assinaturas

Channels e Buffers na Prática

- Copiando dados de um channel para outro

```
RandomAccessFile inFile =
  new RandomAccessFile("arquivo1.txt", "r");
FileChannel inChannel = inFile.getChannel();

RandomAccessFile outFile =
  new RandomAccessFile("arquivo2.txt", "rw");
FileChannel outChannel = outFile.getChannel();

inChannel.transferTo(0, inChannel.size(), outChannel);
```

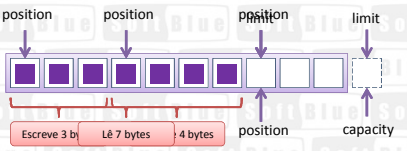
Existe também o método **transferFrom()**

Funcionamento do Buffer NIO

- Um buffer armazena três informações

Informação	Descrição
position	Posição do buffer onde os dados serão escritos ou lidos.
limit	Indica quantos dados ainda podem ser lidos do buffer ou gravados.
capacity	Capacidade total do buffer. Normalmente corresponde ao tamanho do array.

Funcionamento do Buffer NIO



O buffer volta ao estado inicial e está pronto para ser reutilizado

Memory-mapped Files

- Sistemas operacionais modernos têm a capacidade de mapear partes de arquivos em blocos de memória
- O resultado disto é que o arquivo, armazenado no sistema de arquivos, é visto como uma área de memória
- A NIO API tem a capacidade de explorar este recurso

Memory-mapped Files

```
RandomAccessFile file =
  new RandomAccessFile("arquivo.txt", "rw");
FileChannel channel = file.getChannel();

MappedByteBuffer buffer =
  channel.map(MapMode.READ_WRITE, 0, file.length());
```

O método **map()** cria um **MappedByteBuffer**

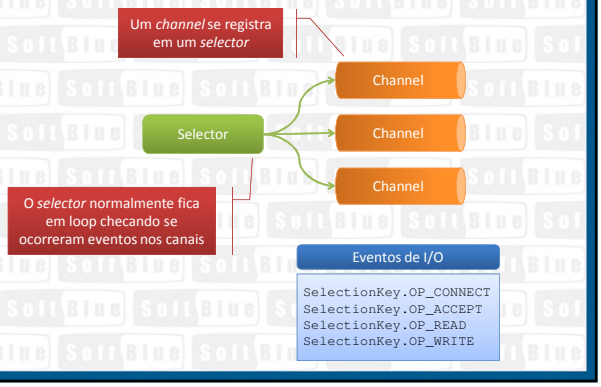
I/O Assíncrono

- Realiza a leitura e escrita de dados sem bloqueio
  - Ao invés do código ficar bloqueado aguardando o resultado, eventos de I/O são gerados e o código pode tomar determinada ação sobre eles
- Com o I/O assíncrono não é necessário disparar diversas threads
  - Uma thread pode fazer o processamento dos eventos de I/O em diversos channels diferentes

Selectors

- O selector desempenha um papel central no I/O assíncrono
- Ele "observa" um ou mais channels e determina se eles estão aptos para a leitura, escrita, etc.
- Um selector pode fazer o papel que várias threads fariam na API padrão de I/O do Java

Funcionamento de um Selector



Exemplo de Uso do Selector

- Cria um channel de socket do lado servidor

```
ServerSocketChannel channel = ServerSocketChannel.open();
channel.configureBlocking(false);
ServerSocket serverSocket = channel.socket();
serverSocket.bind(new InetSocketAddress(3000));
```

- Cria um selector

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

- Registra o channel no selector

```
channel.register(selector, SelectionKey.OP_ACCEPT);
```

Exemplo de Uso do Selector

- Loop de verificação de eventos

```
while (true) {
  selector.select();

  Set<SelectionKey> keys = selector.selectedKeys();
  Iterator<SelectionKey> iter = keys.iterator();

  while (iter.hasNext()) {
    SelectionKey key = iter.next();
    if (key.isAcceptable()) {
      ServerSocketChannel c = (ServerSocketChannel) key.channel();
      // o channel recebeu uma conexão
    }
    iter.remove();
}
```

Colocando em Prática...

Agora que você já aprendeu a teoria, acesse as vídeo-aulas práticas e pratique os assuntos abordados neste módulo!

Clique aqui para acessar as vídeo-aulas práticas