Урок 2

Java NIO

Работа с файлами. Чтение и запись данных. Работа с буферами и FileChannel.

Java NIO

Работа с файлами

Для описания пути к файлу в **java.nio** используется класс **Path** (аналог класса **File** в **java.io**). Чтобы получить экземпляр **Path**, необходимо вызвать статический метод **Paths.get()**, в который можно передать абсолютный или относительный путь.

```
Path path1 = Paths.get("C:\\data\\1.txt");
Path path2 = Paths.get("C:\\data", "123\\22\\2.txt");
Path path3 = Paths.get(".\\22\\..\\44\\2.txt");
```

Символ «.» означает текущий каталог, «..» — подъем на уровень выше.

Для работы с файлами предназначен класс **Files**. Его методы:

Метод	Описание
boolean exists()	Проверяет существование файла или каталога
void move()	Перемещает файл
void copy()	Копирует файл
void delete()	Удаляет файл
void walkFileTree()	Рекурсивно обходит подкаталоги и файлы
List<String> readAllLines()	Считывает весь файл в лист строк
byte[] readAllBytes()	Считывает файл в массив байт

Files.exists() проверяет существование пути в системе, так как есть возможность создать экземпляр класса **Path**, даже если он указывает на несуществующий файл/каталог (например, если хотим создать новый). Пример:

```
Path path = Paths.get("data/1.txt");
boolean pathExists = Files.exists(path);
```

Files.createDirectory() создает каталог по указанному пути. Пример:

```
Path path = Paths.get("data/123");
try {
    Path newDir = Files.createDirectory(path);
} catch(FileAlreadyExistsException e){
    // Каталог уже существует
} catch (IOException e) {
    // Что-то пошло не так при чтении/записи
    e.printStackTrace();
}
```

Files.copy() копирует файл из одного пути в другой:

```
Path sourcePath      = Paths.get("data/1.txt");
Path destinationPath = Paths.get("data/2.txt");

try {
    Files.copy(sourcePath, destinationPath);
} catch(FileAlreadyExistsException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Для перезаписи файлов при копировании используется перегрузка метода **copy()**:

```
Path sourcePath      = Paths.get("data/1.txt");
Path destinationPath = Paths.get("data/2.txt");

try {
    Files.copy(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch(FileAlreadyExistsException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Третий аргумент **StandardCopyOption** указывает, что делать, если файл уже существует.

Files.move() перемещает файл из одного пути в другой, что равносильно переименованию файла (изменению его пути и имени):

```
Path sourcePath      = Paths.get("data/1.txt");
Path destinationPath = Paths.get("data/1/2.txt");

try {
    Files.move(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    e.printStackTrace();
}
```

Третий аргумент метода **move()** указывает, что делать, если файл с таким именем уже существует. Этот аргумент необязательный и в данном случае говорит о том, что при существовании файла его необходимо переписать. Если файл есть, и такой аргумент не указан, при вызове метода **move()** будет брошено **IOException**.

В отличие от **java.io**, **java.nio** позволяет перемещать файлы между разными файловыми системами.

Files.delete() удаляет файл или каталог:

```
Path path = Paths.get("data/files/1.txt");

try {
    Files.delete(path);
} catch (IOException e) {
    // Не удалось удалить файл
    e.printStackTrace();
}
```

Files.walkFileTree() позволяет рекурсивно обойти дерево каталогов. В качестве аргументов подается путь и экземпляр интерфейса **FileVisitor**. Он указывает, что необходимо делать с каталогами и файлами при обходе. Каждый его метод выполняется в определенные моменты обхода дерева: перед заходом в каталог, при посещении файла, а если это сделать не удалось — перед выходом из каталога. Пример:

```
Files.walkFileTree(path, new FileVisitor<Path>() {
  @Override
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
throws IOException {
    System.out.println("pre visit dir:" + dir);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
    System.out.println("visit file: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFileFailed(Path file, IOException exc) throws
IOException {
    System.out.println("visit file failed: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws
IOException {
    System.out.println("post visit directory: " + dir);
    return FileVisitResult.CONTINUE;
  }
});
```

Метод **preVisitDirectory()** вызывается перед посещением каталога, **postVisitDirectory()** — после. **visitFile()** вызывается только при посещении файла (каталоги не учитываются). Метод **visitFileFailed()** срабатывает, если файл посетить не удалось — например, нет прав доступа.

Каждый метод возвращает результат посещения в виде **enum FileVisitResult**, который содержит следующие элементы:

- CONTINUE;
- TERMINATE;
- SKIP_SIBLINGS;
- SKIP_SUBTREE.

Этот результат влияет на дальнейший обход дерева. **CONTINUE** означает, что обход должен продолжаться. **TERMINATE** — что обход необходимо закончить. **SKIP_SIBLINGS** — обход должен продолжаться без посещения соседних каталогов. **SKIP_SUBTREE** — пропустить обход элементов данного каталога (этот результат возвращается только из **preVisitDirectory()**; если его вернуть из других методов, он будет заменен на **CONTINUE**).

Пример поиска файлов

Ниже приведен код программы, которая ищет в каталоге и подкаталогах файл **README.txt**:

```
Path rootPath = Paths.get("data");
String fileToFind = File.separator + "README.txt";

try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
throws IOException {
      String fileString = file.toAbsolutePath().toString();
      //System.out.println("pathString = " + fileString);

      if(fileString.endsWith(fileToFind)){
        System.out.println("file found at path: " + file.toAbsolutePath());
        return FileVisitResult.TERMINATE;
      }
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
    e.printStackTrace();
}
```

Рекурсивное удаление каталогов

Files.walkFileTree() можно использовать для рекурсивного удаления файлов и подкаталогов из каталога. Метод **Files.delete()** удаляет только пустые каталоги, а для заполненных необходимо сделать рекурсивный обход и удалять файлы, начиная с нижнего уровня.

```
Path rootPath = Paths.get("data/to-delete");

try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
throws IOException {
      System.out.println("delete file: " + file.toString());
      Files.delete(file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws
IOException {
      Files.delete(dir);
      System.out.println("delete dir: " + dir.toString());
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
  e.printStackTrace();
}
```

Чтение и запись данных

Основные компоненты Java NIO: **Channels** (каналы), **Buffers** (буферы), **Selectors** (селекторы)

Каналы и буферы

Как правило, все операции ввода/вывода в NIO начинается с канала. Он похож на поток (**Stream**) из пакета **java.io**. Из канала можно считывать данные в буфер, а также записывать из буфера в канал.

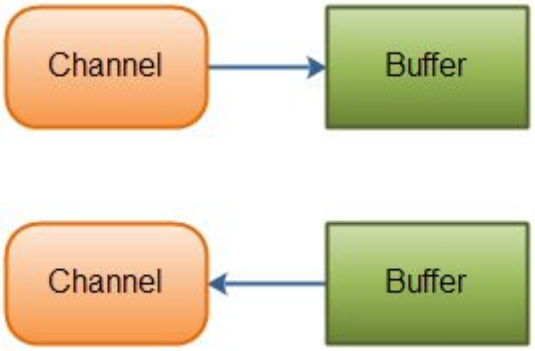

***Рисунок 1**. Каналы считывают данные в буферы, а буферы записывают данные в каналы*

Существует несколько типов каналов и буферов. Основные реализации канала в Java NIO:

- **FileChannel** — считывает данные из файлов и в них;
- **DatagramChannel** — может читать и записывать данные по сети через UDP;
- **SocketChannel** — может читать и записывать данные по сети через TCP;
- **ServerSocketChannel** — позволяет прослушивать входящие TCP-соединения, как веб-сервер. Для каждого входящего соединения создается **SocketChannel**.

Каналы Java NIO похожи на потоки, но есть отличия:

- Каналы можно читать и записывать. А потоки, как правило, являются односторонними (чтение или запись);
- Каналы можно читать и записывать асинхронно;
- Каналы всегда читают данные из буфера и записывают в него.

Селекторы

Селектор позволяет одному потоку обрабатывать несколько каналов. Это удобно, если у приложения множество подключений (каналов), но низкий трафик для каждого соединения — например, на чат-сервере.

Иллюстрация потока, использующего селектор для обработки трех каналов:

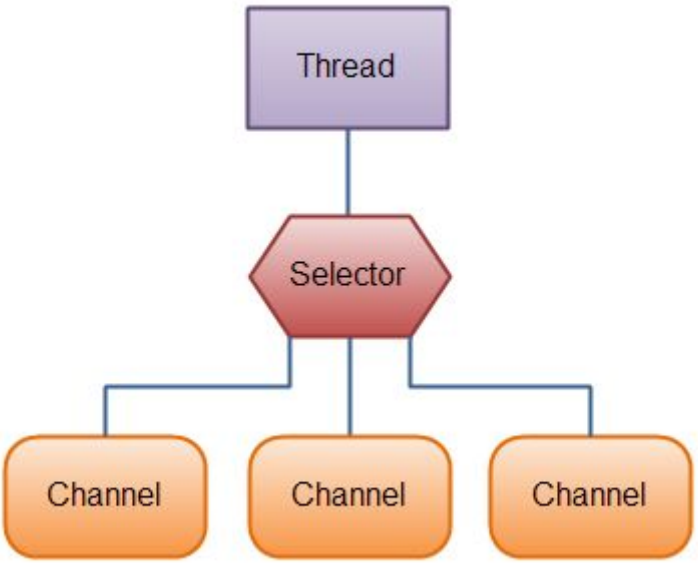

***Рисунок 2**. Один **Thread** использует селектор для обработки трех каналов*

Чтобы использовать селектор, регистрируется канал с ним. Затем надо назвать его методом **select()**, который будет блокироваться до тех пор, пока не подготовится событие для одного из зарегистрированных каналов. Как только метод возвращается, поток может обрабатывать эти события: входящие соединения, полученные данные и т. д.

Работа с буферами

Buffer — это фиксированный блок памяти, в который можно записывать данные и затем считывать их. Класс **java.nio.Buffer** позволяет работать с этим блоком памяти. Буферы хранят определенный тип данных. Основные реализации буфера в Java NIO: **ByteBuffer**, **CharBuffer**, **DoubleBuffer**, **FloatBuffer**, **IntBuffer**, **LongBuffer**, **ShortBuffer**. Эти классы покрывают основные типы данных, которые можно отправлять через IO: **byte**, **short**, **int**, **long**, **float**, **double** и символы.

Чтобы создать экземпляр класса Buffer, необходимо выделить под него память методом **allocate()**:

```
ByteBuffer byteBuf = ByteBuffer.allocate(48);
CharBuffer charBuf = CharBuffer.allocate(1024);
```

Структура буфера

У **Buffer** три основных параметра: **capacity**, **position**, **limit**. Последние два зависят от того, в каком режиме работает буфер — чтения или записи. Capacity не зависит от режима. Схема работы буфера:

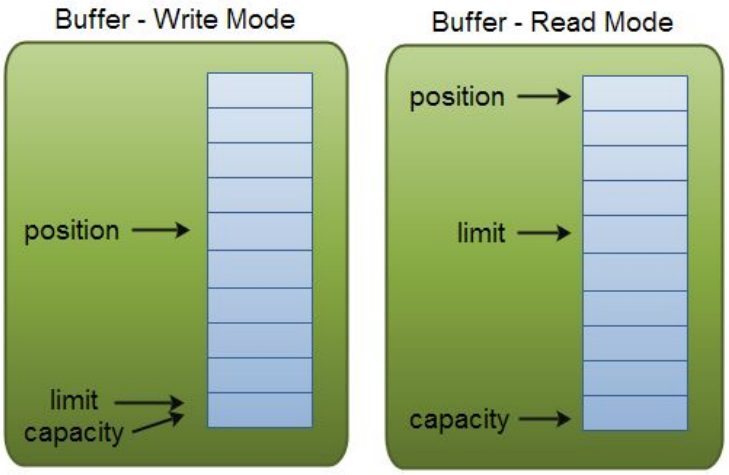

***Рисунок 3**. Схема работы буфера в режимах чтения/записи*

Емкость (Capacity) — у буфера есть фиксированный размер (емкость).

Позиция (Position) — позиция ячейки в буфере, куда производится запись. Максимально возможное значение равно (**capacity** – 1). При записи эта позиция увеличивается. В режиме чтения **position** указывает на ячейку, значение которой будет считано, при чтении позиция увеличивается. При переходе (**flip**) с режима записи на режим чтения позиция сбрасывается в 0.

Предел (Limit) — в режиме записи **limit** показывает, сколько данных можно записать в буфер, и предел равен емкости. При переключении в режим чтения **limit** показывает, до какой ячейки можно читать данные. При переключении из режима записи в режим чтения **limit** устанавливается в последнюю позицию записи.

Методы для работы с буферами

flip() — переключает буфер из режима записи в режим чтения. При вызове этого метода позиция сбрасывается в 0 и **limit** устанавливается в ячейку, где только что был **position**. При переключении в режим чтения позиция обнуляется, и можно читать буфер с начала. Учитывая, что **limit** ставится в последнюю записанную ячейку, мы не можем прочитать больше данных, чем только что записали.

rewind() — сбрасывает **position** в 0, так что прочитанные данные могут быть прочитаны снова.

mark() и **reset()** — первый запоминает позицию в буфере, второй возвращает позицию в ранее запомненную.

clear() — сбрасывает буфер. Позиция устанавливается на 0, **limit** уравнивается с **capacity**. Данные в буфере остаются, но из-за смещения маркеров они недоступны.

compact() — удаляет из буфера прочитанные данные, а непрочитанные переносятся в начало буфера.

Простой пример чтения канала

Использование буфера для чтения/записи данных обычно состоит из 4 шагов:

1. Запись данных в буфер.
2. Переключение буфера в режим чтения (**flip**).
3. Чтение данных из буфера, очистка буфера (**clear**).

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
   FileChannel inChannel = aFile.getChannel();
   ByteBuffer buf = ByteBuffer.allocate(48);
   int bytesRead = inChannel.read(buf);
   while (bytesRead != -1) {
     buf.flip();
     while(buf.hasRemaining()){
         System.out.print((char) buf.get());
     }
     buf.clear();
     bytesRead = inChannel.read(buf);
   }
   aFile.close();
```

Запись данных в буфер

В буфер данные можно записать двумя способами:

1. Записать данные из канала в буфер.
2. Записать данные в буфер вручную.

```
int bytesRead = inChannel.read(buf);              // 1
buf.put(127);                                     // 2
```

Чтение данных из буфера

Существует два способа чтения данных из буфера:

1. Прочитать данные из буфера и отправить в канал.
2. Прочитать данные из буфера вручную.

```
int bytesWritten = inChannel.write(buf);    // 1
byte aByte = buf.get();                     // 2
```

Работа с FileChannel

FileChannel всегда работает **только** в блокирующем режиме. Для начала работы необходимо открыть канал, но напрямую это сделать нельзя. Его можно получить через **InputStream**, **OutputStream** или **RandomAccessFile**. Пример работы через **RandomAccessFile**:

```
RandomAccessFile aFile     = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel      inChannel = aFile.getChannel();
```

Чтение данных из FileChannel

Для чтения данных из канала используется метод **read()**. Пример:

```
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
```

Сначала необходимо создать **Buffer**, после чего с помощью метода **FileChannel.read()** прочитать данные из файла в него. Метод **read()** возвращает **int**, который обозначает количество прочитанных байт. Если вернулось значение **-1**, был достигнут конец файла.

Запись данных в FileChannel

Запись данных в канал осуществляется через метод **write()**, в качестве аргумента принимает **Buffer**:

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

Как видим в коде, метод **FileChannel.write()** работает в цикле и не сообщает, сколько байт было записано в файл. Он работает до тех пор, пока в буфере есть данные.

Закрытие FileChannel

По завершении работы **FileChannel** необходимо закрыть:

```
channel.close();
```

FileChannel Position

При чтении или записи данных из/в **FileChannel** курсор установлен на определенной позиции файла. Положение курсора можно узнать с помощью метода **FileChannel.position()**. Через него же можно указать позицию курсора в файле:

```
long pos = channel.position();
channel.position(pos + 100);
```

Если указать позицию за пределами файла, при попытке чтения будет получено значение **–1**, означающее конец файла. Если же указать позицию за пределами файла в режиме записи, он будет расширен до этого значения и запись продолжится.

FileChannel Size

Метод **size()** позволяет узнать размер файла.

```
long fileSize = channel.size();
```

FileChannel Truncate

Метод **truncate()** позволяет обрезать файл до указанного размера.

```
channel.truncate(1024);
```

Практическое задание

1. Подготовить текстовый файл с описанием проделанной за неделю работы, вопросами по решению отдельных задач (если они возникли) и блоками кода, которые вызвали у вас затруднения (если такие есть);
2. * Реализовать передачу файла с клиента на сервер используя java.io или java.nio.

Дополнительные материалы

1. https://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html
2. https://habr.com/post/235585/

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. http://tutorials.jenkov.com/java-nio/index.html
2. https://docs.oracle.com/javase/tutorial/essential/io/fileio.html