ГЛАВА

21 Система ввода-вывода NIO

Начиная с версии 1.4 в Java предоставляется вторая система ввода-вывода под названием NIO (сокращение от New I/O — новый ввод-вывод). В этой системе поддерживается канальный подход к операциям ввода-вывода, ориентированный на применение буферов. А в версии JDK 7 система ввода-вывода NIO была существенно расширена, и теперь она оказывает улучшенную поддержку средств обработки файлов и файловых систем. На самом деле изменения в этой системе настолько значительны, что она нередко обозначается термином *NIO.2*. Благодаря возможностям, предоставляемым новыми классами файлов из системы ввода-вывода NIO, ожидается, что значение этой системы в обработке файлов будет только возрастать. В этой главе рассматривается ряд основных характеристик системы ввода-вывода NIO.

Классы системы ввода-вывода NIO

В табл. 21.1 перечислены пакеты, в которых содержатся классы системы ввода-вывода NIO.

Таблица 21.1. Пакеты, содержащие классы системы ввода-вывода NIO

Пакет	Назначение
`java.nio`	Это пакет верхнего уровня в системе ввода-вывода NIO. Он инкапсулирует различные типы буферов, содержащих данные, которыми оперирует система ввода-вывода NIO
`java.nio.channels`	Поддерживает каналы, открывающие соединения для ввода-вывода
`java.nio.channels.spi`	Поддерживает поставщики услуг для каналов
`java.nio.charset`	Инкапсулирует наборы символов. Поддерживает также функционирование кодеров и декодеров для взаимного преобразования символов и байтов
`java.nio.charset.spi`	Поддерживает поставщики услуг для наборов символов
`java.nio.file`	Поддерживает ввод-вывод в файлы
`java.nio.file.attribute`	Поддерживает атрибуты файлов
`java.nio.file.spi`	Поддерживает поставщики услуг для файловых систем

Прежде чем приступить к рассмотрению системы ввода-вывода NIO, следует заметить, что эта система не предназначена для замены классов ввода-вывода, входящих в состав пакета `java.io` и представленных в главе 20. Напротив, практические знания о классах из этого пакета помогают легче усвоить систему ввода-вывода NIO.

На заметку! В этой главе предполагается, что вы уже проработали материал глав 13 и 20, посвященный принципам ввода-вывода вообще и потокового ввода-вывода в частности.

Основные положения о системе ввода-вывода NIO

Система ввода-вывода NIO построена на двух основополагающих элементах: буферах и каналах. В *буфере* хранятся данные, а *канал* предоставляет открытое соединение с устройством ввода-вывода, например файлом или сокетом. В общем, для применения системы ввода-вывода NIO требуется получить канал для устройства ввода-вывода и буфер для хранения данных. После этого можно обращаться с буфером, вводя или выводя данные по мере необходимости. Поэтому в последующих разделах буфера и каналы будут рассмотрены подробно.

Буфера

Буфера определяются в пакете `java.nio`. Все буфера являются подклассами, производными от класса `Buffer`, в котором определяются основные функциональные возможности, характерные для каждого буфера, в том числе текущая позиция, предел и емкость. *Текущая позиция* определяет индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Предел* определяет значение индекса за позицией последней доступной ячейки в буфере. *Емкость* определяет количество элементов, которые можно хранить в буфере. Зачастую предел равен емкости буфера. В классе `Buffer` поддерживается также отметка и очистка буфера. В нем определяется ряд методов, перечисленных в табл. 21.2.

Таблица 21.2. Методы из классы `Buffer`

Метод	Описание
`abstract Object array()`	Возвращает ссылку на массив, если вызывающий буфер поддерживается массивом, иначе генерирует исключение типа **`UnsupportedOperationException`**. Если же массив доступен только для чтения, то генерируется исключение типа **`ReadOnlyBufferException`**
`abstract int arrayOffset()`	Возвращает индекс первого элемента массива, если вызывающий буфер поддерживается массивом, а иначе генерируется исключение типа **`UnsupportedOperationException`**. Если же массив доступен только для чтения, то генерируется исключение типа **`ReadOnlyBufferException`**

Окончание табл. 21.2

Метод	Описание
`final int capacity()`	Возвращает количество элементов, которые можно хранить в вызывающем буфере
`final Buffer clear()`	Очищает вызывающий буфер и возвращает ссылку на него
`final Buffer flip()`	Задает текущую позицию в качестве предела для вызывающего буфера и затем устанавливает текущую позицию в нуль. Возвращает ссылку на буфер
`abstract boolean has Array()`	Возвращает логическое значение **true**, если вызывающий буфер поддерживается массивом, доступным для чтения и записи, а иначе — логическое значение **false**
`final boolean hasRemaining()`	Возвращает логическое значение **true**, если в вызывающем буфере еще остались какие-нибудь элементы, а иначе — логическое значение **false**
`abstract boolean isDirect()`	Возвращает логическое значение **true**, если вызывающий буфер оказывается прямым. Иными словами, операции ввода-вывода выполняются над ним напрямую. В противном случае возвращается логическое значение **false**
`abstract boolean isReadOnly()`	Возвращает логическое значение **true**, если вызывающий буфер является буфером только для чтения, а иначе — логическое значение **false**
`final int limit()`	Возвращает предел для вызывающего буфера
`final Buffer limit(int n)`	Задает предел ***n*** для вызывающего буфера. Возвращает ссылку на буфер
`final Buffer mark()`	Устанавливает метку и возвращает ссылку на вызывающий буфер
`final int position()`	Возвращает текущую позицию
`final Buffer position(int n)`	Задает текущую позицию буфера равной ***n***. Возвращает ссылку на буфер
`int remaining()`	Возвращает количество элементов, доступных до того, как будет достигнут предел. Иными словами, возвращается предел минус текущая позиция
`final Buffer reset()`	Устанавливает текущую позицию в вызывающем буфере на установленной ранее метке. Возвращает ссылку на буфер
`final Buffer rewind()`	Устанавливает текущую позицию в вызывающем буфере в нуль. Возвращает ссылку на буфер

От класса `Buffer` происходят приведенные ниже классы конкретных буферов, где тип хранимых данных можно определить по их именам. Класс `MappedByteBuffer` является производным от класса `ByteBuffer` и используется для сопоставления файла с буфером.

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

Все упомянутые выше буфера предоставляют различные методы `get()` и `put()`, которые позволяют получать данные из буфера или вносить их в него. (Разумеется, метод `put()` недоступен, если буфер предназначен только для чтения.) В табл. 21.3 перечислены методы `get()` и `put()`, определенные в классе `ByteBuffer`. Другие классы буферов имеют похожие методы. Во всех классах буферов поддерживают-

ся также методы, выполняющие различные операции с буфером. Например, с помощью метода `allocate()` можно вручную выделить оперативную память под буфер, с помощью метода `wrap()` — организовать массив в пределах буфера, а с помощью метода `slice()` — создать подпоследовательность в буфере.

Таблица 21.3. Методы `get()` и `put()` из класса `ByteBuffer`

Метод	Описание
`abstract byte get()`	Возвращает байт на текущей позиции
`ByteBuffer get(byte` *`значения`*`[])`	Копирует вызывающий буфер в заданный массив ***значения***. Возвращает ссылку на буфер. Если же в буфере не осталось больше элементов, количество которых равно ***значения*`.length`**, то генерируется исключение типа **`BufferUnderflowException`**
`ByteBuffer get(byte` *`значения`*`[], int` *`начало`*`, int` *`количество`*`)`	Копирует заданное **`количество`** элементов из вызывающего буфера в указанный массив ***значения***, начиная с позиции по индексу ***начало***. Возвращает ссылку на буфер. Если в буфере больше не осталось заданное ***количество*** элементов, то генерируется исключение типа **`BufferUnderflowException`**
`abstract byte get(int` *`индекс`*`)`	Возвращает из вызывающего буфера байт по указанному ***индексу***
`abstract ByteBufferput(byte` *`b`*`)`	Копирует заданный байт ***b*** на текущую позицию в вызывающем буфере. Возвращает ссылку на буфер. Если буфер заполнен, то генерируется исключение типа **`BufferOverflowException`**
`final ByteBufferput(byte` *`значения`*`[])`	Копирует все элементы из указанного массива ***значения*** в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер. Если буфер не может вместить все элементы, то генерируется исключение типа **`BufferOverflowException`**
`ByteBuffer put(byte` *`значения`*`[], int` *`начало`*`, int` *`количество`*`)`	Копирует в вызывающий буфер заданное ***количество*** элементов из указанного массива ***значения***, начиная с указанной позиции ***начало***. Возвращает ссылку на буфер. Если буфер не может хранить все элементы, то генерируется исключение типа **`BufferOverflowException`**
`ByteBufferput(ByteBuffer` *`bb`*`)`	Копирует элементы из заданного буфера ***bb*** в вызывающий буфер, начиная с текущей позиции. Если буфер не может хранить все элементы, то генерируется исключение типа **`BufferOverflowException`**. Возвращает ссылку на буфер
`abstract ByteBufferput(int` *`индекс`*`, byte` *`b`*`)`	Копирует байт ***b*** на позицию по указанному ***индексу*** в вызывающем буфере. Возвращает ссылку на буфер

Каналы

Каналы определены в пакете `java.nio.channels`. *Канал* представляет открытое соединение с источником или адресатом ввода-вывода. Классы каналов реализуют ин-

терфейс Channel, расширяющий интерфейс Closeable, а начиная с JDK 7 — интерфейс AutoCloseable. При реализации интерфейса AutoCloseable каналами можно управлять в блоке оператора try с ресурсами, где канал закрывается автоматически, когда он больше не нужен. (Подробнее об операторе try с ресурсами см. в главе 13.)

Один из способов получения канала подразумевает вызов метода getChannel() для объекта, поддерживающего каналы. Например, метод getChannel() поддерживается в следующих классах ввода-вывода:

DatagramSocket	**FileInputStream**	**FileOutputStream**
RandomAccessFile	**ServerSocket**	**Socket**

Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод getChannel(). Например, когда метод getChannel() вызывается для объекта типа FileInputStream, FileOutputStream или RandomAccessFile, он возвращает канал типа FileChannel. А если этот метод вызывается для объекта типа Socket, то он возвращает канал типа SocketChannel.

Еще один способ получения канала подразумевает использование одного из статических методов, определенных в классе Files, который был введен в версии JDK 7. Например, используя класс Files, можно получить байтовый канал при вызове метода newByteChannel(). Он возвращает канал типа SeekableByteChannel, т.е. интерфейса, реализуемого классом FileChannel. (Более подробно класс Files рассматривается далее в этой главе.)

В каналах типа FileChannel и SocketChannel поддерживаются различные методы read() и write(), которые позволяют выполнять операции ввода-вывода через канал. Например, в табл. 21.4 перечислен ряд методов read() и write(), определенных в классе FileChannel.

Таблица 21.4. Методы read() и write() из класса FileChannel

Метод	Описание
abstract int read(ByteBuffer *bb*) throws IOException	Считывает байты из вызывающего канала в указанный буфер ***bb*** до тех пор, пока буфер не будет заполнен или же не исчерпаются вводимые данные. Возвращает количество прочитанных байтов
abstract int read(ByteBuffer *bb*, long *начало*) throws IOException	Считывает байты из вызывающего канала в указанный буфер **bb**, начиная с позиции ***начало*** и до тех пор, пока буфер не будет заполнен или же не исчерпаются вводимые данные. Текущая позиция не изменяется. Возвращает количество прочитанных байтов или значение **-1**, если позиция ***начало*** окажется за пределами файла
abstract int write(ByteBuffer bb) throws IOException	Записывает содержимое байтового буфера в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов
abstract int write(ByteBuffer *bb*, long *начало*) throws IOException	Записывает содержимое байтового буфера в вызывающий канал, начиная с позиции ***начало*** в файле. Текущая позиция не изменяется. Возвращает количество записанных байтов

Все каналы поддерживают дополнительные методы, предоставляющие доступ к каналу и позволяющие управлять им. Например, канал типа `FileChannel` поддерживает среди прочего методы для получения и установки текущей позиции, передачи данных между файловыми каналами, получения текущего размера канала и его блокировки. В классе `FileChannel` предоставляется статический метод `open()`, который открывает файл и возвращает для него канал. Такой результат достигается другим способом получения канала. В классе `FileChannel` предоставляется также метод `map()`, с помощью которого можно сопоставить файл с буфером.

Наборы символов и селекторы

В системе ввода-вывода NIO применяются наборы символов и селекторы. *Набор символов* определяет способ сопоставления байтов с символами. С помощью *кодера* можно закодировать последовательность символов в виде байтов. Процесс декодирования производится с помощью *декодера*. Наборы символов, кодеры и декодеры поддерживаются в классах, определяемых в пакете `java.nio.charset`. Кодеры и декодеры предоставляются по умолчанию, и поэтому обращаться непосредственно к наборам символов приходится крайне редко.

Селектор обеспечивает возможность многоканального ввода-вывода по ключам, не прибегая к блокировке. Иными словами, с помощью селекторов можно выполнять операции ввода-вывода через несколько каналов. Селекторы поддерживаются классами, определяемыми в пакете `java.nio.channels`. Они чаще всего применяются в каналах, опирающихся на сокеты. В примерах, представленных в этой главе, наборы символов и селекторы не применяются, тем не менее они могут оказаться полезными в ряде приложений.

Усовершенствования в системе NIO, начиная с версии JDK 7

В версии JDK 7 система ввода-вывода NIO была значительно расширена и усовершенствована. Помимо поддержки оператора `try` с ресурсами, который обеспечивает автоматическое управление ресурсами, усовершенствования включают три новых пакета (`java.nio.file`, `java.nio.file.attribute` и `java.nio.file.spi`), несколько новых классов, интерфейсов и методов, а также прямую поддержку потокового ввода-вывода. Эти усовершенствования существенно расширили возможности для применения системы ввода-вывода NIO, особенно в файлы. В последующих разделах описывается ряд ключевых дополнений данной системы.

Интерфейс `Path`

Возможно, одним из наиболее важных дополнений системы ввода-вывода NIO является интерфейс `Path`, поскольку он инкапсулирует путь к файлу. Как будет показано далее, интерфейс `Path` служит связующим звеном для большинства новых файловых средств в системе ввода-вывода NIO.2. Он описывает расположе-

ние файла в структуре каталогов. Интерфейс `Path` находится в пакете `java.nio.file` и наследует интерфейсы `Watchable`, `Iterable<Path>` и `Comparable<Path>`. Интерфейс `Watchable` описывает объект, который можно наблюдать и изменять. Интерфейсы `Iterable` и `Comparable` были представлены ранее в данной книге.

В интерфейсе `Path` объявляется немало методов для манипулирования путями к файлам. Некоторые из них приведены в табл. 21.5. Обратите особое внимание на метод `getName()`. Он служит для получения элемента пути. С этой целью в данном методе применяется индекс. Нулевому значению индекса соответствует ближайшая к корневому каталогу часть пути, являющаяся его крайним слева элементом. Последующие индексы определяют элементы вправо от корневого каталога. Количество элементов в пути может быть получено в результате вызова метода `getNameCount()`. Если же требуется получить строковое представление всего пути, достаточно вызвать метод `toString()`. Следует также заметить, что для распознавания относительного и абсолютного пути достаточно вызвать метод `resolve()`.

Таблица 21.5. Избранные методы из интерфейса `Path`

Метод	Описание
`boolean endsWith(String` *`путь`*`)`	Возвращает логическое значение **`true`**, если вызывающий объект типа **`Path`** оканчивается путем, определяемым параметром ***путь***, а иначе — логическое значение **`false`**
`boolean endsWith(Path` *`путь`*`)`	Возвращает логическое значение **`true`**, если вызывающий объект типа **`Path`** оканчивается путем, определяемым параметром ***путь***, а иначе — логическое значение **`false`**
`Path getFileName()`	Возвращает имя файла, связанное с вызывающим объектом типа **`Path`**
`Path getName(int` *`индекс`*`)`	Возвращает объект типа **`Path`**, содержащий имя элемента пути по указанному ***индексу*** в вызывающем объекте. Крайний слева элемент имеет нулевой индекс и находится ближе всего к корневому каталогу. А крайний справа элемент имеет индекс **`getNameCount() - 1`**
`int getNameCount()`	Возвращает количество элементов (кроме корневого) в вызывающем объекте типа **`Path`**
`Path getParent()`	Возвращает объект типа **`Path`**, который содержит весь путь, кроме имени файла, определяемого вызывающим объектом типа **`Path`**
`Path getRoot()`	Возвращает корневой каталог из вызывающего объекта типа **`Path`**
`boolean isAbsolute()`	Возвращает логическое значение **`true`**, если вызывающий объект типа **`Path`** обозначает абсолютный путь, а иначе — логическое значение **`false`**

Окончание табл. 21.5

Метод	Описание
`Path resolve(Path` ***путь*****`)`**	Если указанный ***путь*** является абсолютным, то возвращается именно он. А если указанный ***путь*** не содержит корневой каталог, то этот ***путь*** предваряется корневым каталогом из вызывающего объекта типа **`Path`**, а затем возвращается полученный результат. Если же указанный ***путь*** пуст, то возвращается вызывающий объект типа **`Path`**. В противном случае поведение данного метода не определено
`Path resolve(String` ***путь*****`)`**	Если указанный ***путь*** является абсолютным, возвращается именно этот ***путь***. А если указанный ***путь*** не содержит корневой каталог, то этот ***путь*** предваряется корневым каталогом из вызывающего объекта типа **`Path`**, а затем возвращается полученный результат. Если же указанный ***путь*** пуст, то возвращается вызывающий объект типа **`Path`**. В противном случае поведение данного метода не определено
`boolean startsWith(String` ***путь*****`)`**	Возвращает логическое значение **`true`**, если вызывающий объект типа **`Path`** начинается с указанного ***пути***, а иначе — логическое значение **`false`**
`boolean startsWith(Path` ***путь*****`)`**	Возвращает логическое значение **`true`**, если вызывающий объект типа **`Path`** начинается с указанного ***пути***, а иначе — логическое значение **`false`**
`Path toAbsolutePath()`	Возвращает вызывающий объект типа **`Path`** в виде абсолютного пути
`String toString()`	Возвращает строковое представление вызывающего объекта типа **`Path`**

Следует также иметь в виду, что при обновлении унаследованного кода, в котором используется класс `File`, определенный в пакете `java.io`, экземпляр класса `File` можно преобразовать в экземпляр интерфейса `Path`, вызвав метод `toPath()` для объекта типа `File`. Этот метод был введен в класс `File` в версии JDK 7. Кроме того, экземпляр класса `File` можно получить, вызвав метод `toFile()`, определяемый в интерфейсе `Path`.

Класс `Files`

Большинство действий, которые выполняются над файлами, предоставляются статическими методами из класса `Files`. Путь к файлу, над которым выполняются определенные действия, задает объект типа `Path`. Таким образом, методы из класса `Files` используют объект типа `Path`, чтобы указать используемый файл. Класс `Files` обладает обширным рядом функциональных возможностей. Так, в нем имеются методы, позволяющие открывать или создавать файл по указанному пути. Кроме того, из объекта типа `Path` можно получить следующие сведения о файле: является ли он исполняемым, скрытым или доступным только для чтения. В классе

Files предоставляются также методы, позволяющие копировать или перемещать файлы. Некоторые методы, определенные в этом классе, перечислены в табл. 21.6. Помимо исключения типа IOException, возможны и другие исключения. В версии JDK 8 класс Files дополнен следующими четырьмя методами: list(), walk(), lines() и find(). Все эти методы возвращают объект типа Stream. Они способствуют интеграции системы ввода-вывода NIO с новым прикладным программным интерфейсом API потоков ввода-вывода, определенным в версии JDK 8 и описываемым в главе 29.

Таблица 21.6. Избранные методы из класса Files

Метод	Описание
static Path copy(Path *источник*, **Path** *адресат* **CopyOption ... *способ*) throws IOException**	Копирует файл из ***источника*** по указанному ***адресату*** заданным ***способом***
static Path createDirectory(Path *путь*, **FileAttribute<?> ... *атрибуты*) throws IOException**	Создает каталог по указанному ***пути***. Атрибуты каталога определяются параметром ***атрибуты***
static Path createFile(Path *путь*, **FileAttribute<?> ... *атрибуты*) throws IOException**	Создает файл по указанному ***пути***. Атрибуты файла определяются параметром ***атрибуты***
static void delete(Path *путь*) **throws IOException**	Удаляет файл по указанному ***пути***
static boolean exists(Path *путь*, **LinkOptions ... *параметры*)**	Возвращает логическое значение **true**, если файл существует по указанному ***пути***, а иначе — логическое значение **false**. Если же аргумент ***параметры*** не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент ***параметры*** должен принимать значение **NOFOLLOW_LINKS**
static boolean isDirectory(Path *путь*, **LinkOptions ... *параметры*)**	Возвращает логическое значение **true**, если параметр ***путь*** определяет каталог, а иначе — логическое значение **false**. Если же аргумент ***параметры*** не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам заданный аргумент ***параметры*** должен принимать значение **NOFOLLOW_LINKS**
static boolean isExecutable(Path *путь*)	Возвращает логическое значение **true**, если файл по указанному ***пути*** является исполняемым, а иначе — логическое значение **false**
static boolean isHidden(Path *путь*) **throws IOException**	Возвращает логическое значение **true**, если файл по указанному ***пути*** является скрытым, а иначе — логическое значение **false**
static boolean isReadable(Path *путь*)	Возвращает логическое значение **true**, если файл по указанному ***пути*** доступен для чтения, а иначе — логическое значение **false**

Окончание табл. 21.6

Метод	Описание
`static boolean isRegularFile(Path` *`путь`*`, LinkOptions ...` *`параметры`*`)`	Возвращает логическое значение **`true`**, если параметр ***путь*** определяет файл, а иначе — логическое значение **`false`**. Если же аргумент ***параметры*** не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент ***параметры*** должен принимать значение **`NOFOLLOW_LINKS`**
`static boolean isWritable(Path` *`путь`*`)`	Возвращает логическое значение **`true`**, если файл по указанному ***пути*** доступен для записи, а иначе — логическое значение **`false`**
`static Path move(Path` *`источник`*`, Path` *`адресат`*`, CopyOption ...` *`способ`*`) throws IOException`	Копирует файл из ***источника*** по указанному ***адресату*** заданным ***способом***
`static SeekableByteChannel newByteChannel(Path` *`путь`*`, OpenOption ...` *`способ`*`) throws IOException`	Открывает файл по указанному ***пути*** заданным ***способом***. Возвращает для файла байтовый канал типа **`SeekableByteChannel`**. Текущая позиция в этом канале может быть изменена. Интерфейс **`SeekableByteChannel`** реализуется классом **`FileChannel`**
`static DirectoryStream<Path> newDirectoryStream(Path` *`путь`*`) throws IOException`	Открывает каталог по указанному ***пути***. Возвращает поток ввода каталога типа **`DirectoryStream`**, связанный с каталогом
`static InputStream newInputStream(Path` *`путь`*`, OpenOption ...` *`способ`*`) throws IOException`	Открывает файл по указанному ***пути*** заданным ***способом***. Возвращает поток ввода типа **`InputStream`**, связанный с файлом
`static OutputStream newOutputStream(Path` *`путь`*`, OpenOption ...` *`способ`*`) throws IOException`	Открывает файл по указанному ***пути*** заданным ***способом***. Возвращает поток вывода типа **`OutputStream`**, связанный с файлом
`static boolean notExists(Path` *`путь`*`, LinkOption ...` *`параметры`*`)`	Возвращает логическое значение **`true`**, если файл по указанному ***пути*** *не существует*, а иначе — логическое значение **`false`**. Если же аргумент ***параметры*** не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент ***параметры*** должен принимать значение **`NOFOLLOW_LINKS`**
`static <A extends BasicFileAttributes> A readAttributes(Path` *`путь`*`, Class<A>` *`тип_атрибута`*`, LinkOption ...` *`параметры`*`) throws IOExceptioьn`	Получает атрибуты, связанные с файлом. Тип передаваемых атрибутов определяется параметром ***тип_атрибута***. Если же аргумент ***параметры*** не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент ***параметры*** должен принимать значение **`NOFOLLOW_LINKS`**
`static long size(Path` *`путь`*`) throws IOException`	Возвращает размер файла по указанному ***пути***

Обратите внимание на то, что некоторые методы, перечисленные в табл. 21.6, получают аргумент типа `OpenOption`. Это интерфейс, описывающий способ открытия файла. Он реализуется классом `StandardOpenOption`, где определяется перечисление, значения которого представлены в табл. 21.7.

Таблица 21.7. Стандартные значения параметров открытия файлов

Значение	Назначение
APPEND	Присоединить выводимые данные в конце файла
CREATE	Создать файл, если он еще не существует
CREATE_NEW	Создать файл только в том случае, если он еще не существует
DELETE_ON_CLOSE	Удалить файл, когда он закрывается
DSYNC	Немедленно записать вносимые изменения в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
READ	Открыть файл для операций ввода
SPARSE	Указать файловой системе, что файл разрежен, а следовательно, не может быть полностью заполнен данными. Если файловая система не поддерживает разреженные файлы, это значение параметра игнорируется
SYNC	Немедленно записать вносимые изменения в файл или его метаданные в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
TRUNCATE_EXISTING	Укоротить до нуля длину уже существующего файла, открываемого для вывода
WRITE	Открыть файл для операций вывода

Класс `Paths`

Экземпляр типа `Path` нельзя создать непосредственно с помощью конструктора, поскольку это интерфейс, а не класс. Вместо этого можно получить объект типа `Path`, вызвав метод, который возвращает этот объект. Как правило, для этой цели служит метод `get()`, определяемый в классе `Paths`. Существуют две формы метода `get()`. Ниже приведена та его форма, которая употребляется в примерах этой главы.

```
static Path get(String имя_пути, String ... части)
```

Этот метод возвращает объект, инкапсулирующий определенный путь. Путь может быть задан двумя способами. Если параметр *части* не указан, то путь должен полностью определяться параметром *имя_пути*. В качестве альтернативы путь можно передать по частям, причем первую часть в качестве параметра *имя_пути*, а остальные части — в качестве параметра *части* переменной длины. Но в любом случае метод `get()` сгенерирует исключение типа `InvalidPathException`, если указанный путь синтаксически недостоверен.

Во второй форме метода `get()` объект типа `Path` создается из URI. Эта форма выглядит следующим образом:

```
static Path get(URI uri)
```

В итоге возвращается объект типа `Path`, соответствующий заданному параметру *`uri`*. Следует, однако, иметь в виду, что создание объекта типа `Path` не приводит к открытию или созданию файла. Вместо этого лишь создается объект, инкапсулирующий путь к каталогу, в котором находится файл.

Интерфейсы атрибутов файлов

С файлами связан ряд атрибутов, обозначающих время создания файла, время его последней модификации, размер файла или каталог. Система ввода-вывода NIO организует атрибуты файлов в виде иерархии различных интерфейсов, определенных в пакете `java.nio.file.attribute`. На вершине этой иерархии находится интерфейс `BasicFileAttributes`, инкапсулирующий ряд атрибутов, которые обычно применяются в большинстве файловых систем. Методы, определенные в интерфейсе `BasicFileAttributes`, перечислены в табл. 21.8.

Таблица 21.8. Методы из интерфейса `BasicFileAttributes`

Метод	Описание
`FileTime creationTime()`	Возвращает время создания файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
`Object fileKey()`	Возвращает файловый ключ. Если этот атрибут не поддерживается файловой системой, возвращается пустое значение `null`
`boolean isDirectory()`	Возвращает логическое значение `true`, если файл является каталогом
`boolean isOther()`	Возвращает логическое значение `true`, если файл является символической ссылкой или каталогом, а не файлом
`boolean isRegularFile()`	Возвращает логическое значение `true`, если файл является обычным файлом, а не каталогом или символической ссылкой
`boolean isSymbolicLink()`	Возвращает логическое значение `true`, если файл является символической ссылкой
`FileTime lastAccessTime()`	Возвращает время последнего обращения к файлу. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
`FileTime lastModifiedTime()`	Возвращает время последней модификации файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
`long size()`	Возвращает размер файла

Производными от интерфейса BasicFileAttributes являются следующие два интерфейса: DosFileAttributes и PosixFileAttributes. В частности, интерфейс DosFileAttributes описывает атрибуты, связанные с файловой системой FAT, которые были первоначально определены в файловой системе DOS. В этом интерфейсе определяются методы, перечисленные в табл. 21.9.

Таблица 21.9. Методы из интерфейса DosFileAttributes

Метод	Описание
`boolean isArchive()`	Возвращает логическое значение **true**, если файл помечен как архивный, а иначе — логическое значение **false**
`boolean isHidden()`	Возвращает логическое значение **true**, если файл помечен как скрытый, а иначе — логическое значение **false**
`boolean isReadOnly()`	Возвращает логическое значение **true**, если файл помечен как доступный только для чтения, а иначе — логическое значение **false**
`boolean isSystem()`	Возвращает логическое значение **true**, если файл помечается как системный, а иначе — логическое значение **false**

Интерфейс PosixFileAttributes инкапсулирует атрибуты, определенные по стандартам POSIX (Portable Operating System Interface — переносимый интерфейс операционных систем). В этом интерфейсе определяются методы, перечисленные в табл. 21.10.

Таблица 21.10. Методы из интерфейса PosixFileAttributes

Метод	Описание
`GroupPrincipal group()`	Возвращает группового владельца файла
`UserPrincipal owner()`	Возвращает отдельного владельца файла
`Set<PosixFilePermission> permissions()`	Возвращает полномочия доступа к файлу

Имеются разные способы доступа к атрибутам файлов. В частности, вызвав статический метод readAttributes(), определенный в классе Files, можно получить объект, инкапсулирующий атрибуты файла. Ниже приведена одна из общих форм объявления этого метода.

```
static <A extends BasicFileAttributes>
        A readAttributes(Path путь, Class<A> тип_атрибута,
        LinkOption... параметры) throws IOException
```

Этот метод возвращает ссылку на объект, обозначающий атрибуты файла по указанному *пути*. Конкретный тип атрибутов указывается в виде объекта типа Class с помощью параметра *тип_атрибута*. Например, для получения основных атрибутов файла следует передать в качестве параметра *тип_атрибута* объект типа BasicFileAttributes.class, для получения атрибутов DOS — объект типа DosFileAttributes.class, а для получения атрибутов POSIX — объект типа PosixFileAttributes.class. Дополнительные, но необязательные параметры ссылок передаются как аргумент *параметры*. Если же аргумент *параметры* не опре-

делен, то следуют символические ссылки. Метод `readAttributes()` возвращает ссылку на требуемый атрибут. Если же тип требуемого атрибута недоступен, то генерируется исключение типа `UnsupportedOperationException`. Используя объект, возвращаемый этим методом, можно обратиться к атрибутам файла.

Еще один способ доступа к атрибутам файла состоит в том, чтобы вызвать метод `getFileAttributeView()`, определенный в классе `Files`. В системе ввода-вывода NIO определяется несколько интерфейсов для представлений атрибутов, в том числе `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView` и `PosixFileAttributeView`. В примерах из этой главы представления атрибутов не употребляются, но в некоторых случаях это средство может оказаться полезным.

Иногда можно и не прибегать непосредственно к интерфейсам атрибутов файлов, поскольку в классе `Files` предоставляются служебные статические методы, позволяющие обращаться к некоторым атрибутам. Для этой цели в классе `Files` имеются такие методы, как `isHidden()` и `isWritable()`.

Следует, однако, иметь в виду, что все допустимые атрибуты файлов поддерживаются не во всех файловых системах. Например, атрибуты файлов DOS относятся к файловой системе FAT, хотя первоначально они были определены в файловой системе DOS. Те атрибуты, которые применяются в обширном ряде файловых систем, описаны в интерфейсе `BasicFileAttributes`. Поэтому именно они и употребляются в примерах из этой главы.

Классы `FileSystem`, `FileSystems` и `FileStore`

Для упрощения доступа к файловой системе в пакете `java.nio.file` предоставляются классы `FileSystem` и `FileSystems`. В действительности, используя метод `newFileSystem()`, определенный в классе `FileSystems`, можно даже получить новую файловую систему. А класс `FileStore` инкапсулирует систему хранения файлов. И хотя упоминаемые здесь классы не употребляются в примерах из этой главы, им можно найти применение в своих прикладных программах.

Применение системы ввода-вывода NIO

В этом разделе демонстрируется применение системы ввода-вывода NIO для решения самых разных задач. Но прежде следует подчеркнуть, что в версии JDK 7 была значительно расширена как сама система ввода-вывода NIO, так и область ее применения. Как упоминалось ранее, усовершенствованную версию этой системы иногда называют NIO.2. Вследствие столь значительных усовершенствований в системе ввода-вывода NIO.2 изменился и способ написания кода на ее основе, а также расширился круг задач, для решения которых можно ее применять. В силу этого обстоятельства в большей части примеров из этой главы применяются средства из системы ввода-вывода NIO.2, и поэтому для проработки этих примеров потребуется комплект JDK 7, JDK 8 или более поздняя его версия. Тем не менее в конце этой главы дается краткое описание кода, написанного до версии JDK 7, в помощь тем программистам, которые пользуются системой ввода-вывода до версии JDK 7 или сопровождают унаследованный код.

Помните! Для компиляции большинства примеров из этой главы требуется комплект JDK 7, JDK 8 или более поздняя его версия.

Раньше система NIO предназначалась в основном для канального ввода-вывода, и эта разновидность ввода-вывода по-прежнему остается важнейшей областью ее применения. Но теперь систему ввода-вывода NIO можно также использовать для потокового ввода-вывода и выполнения операций в файловой системе. Таким образом, обсуждение областей применения системы ввода-вывода NIO можно разделить на три части:

- канальный ввод-вывод;
- потоковый ввод-вывод;
- операции в файловой системе.

Самым распространенным средством ввода-вывода является файл на диске, поэтому в примерах, представленных далее в этой главе, употребляются файлы на диске. А поскольку все канальные операции ввода-вывода в файлы основываются на передаче байтов, то для их выполнения будут использоваться буфера типа `ByteBuffer`.

Прежде чем открыть файл для доступа к нему средствами системы ввода-вывода NIO, следует получить объект типа `Path`, описывающий этот файл. Это можно, в частности, сделать, вызвав упоминавшийся ранее фабричный метод `Paths.get()`. В приведенных далее примерах употребляется следующая общая форма объявления метода `get()`:

```
static Path get(String имя_пути, String ... части)
```

Напомним, что путь к файлу можно указать двумя способами. Во-первых, передать первую его часть в качестве параметра *имя_пути*, а остальные части — в качестве параметра *части* переменной длины. И во-вторых, указать весь путь в качестве параметра *имя_пути*, а параметр *части* опустить. Именно такой способ и применяется в рассматриваемых далее примерах.

Применение системы NIO для канального ввода-вывода

Важнейшей областью применения системы ввода-вывода NIO является получение доступа к файлу через каналы и буфера. В последующих разделах демонстрируются некоторые способы применения канала для чтения и записи данных в файл.

Чтение файла через канал

Имеется несколько способов чтения данных из файла через канал. Наиболее распространенный из них, вероятно, состоит в том, чтобы сначала выделить оперативную память под буфер вручную, а затем выполнить явным образом операцию чтения для загрузки этого буфера данными из файла. Поэтому рассмотрим этот способ в первую очередь.

Прежде чем прочитать данные из файла, его нужно открыть. Для этого сначала создается объект типа `Path`, описывающий файл, а затем он используется для открытия файла. Имеются разные способы открыть файл в зависимости от того, как он будет использоваться. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Метод `newByteChannel()` имеет следующую общую форму:

```
static SeekableByteChannel newByteChannel(
       Path путь, OpenOption ... способ)
       throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Объект типа `Path`, описывающий файл, передается в качестве параметра *путь*. А параметр *способ* определяет порядок открытия файла. Это параметр переменной длины, и поэтому в качестве его можно указать любое количество аргументов через запятую. (Допустимые значения параметров данного метода обсуждались ранее и приведены в табл. 21.7.) Если же никаких аргументов не указано, то файл открывается для операций ввода. Интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Он реализуется классом `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его. Классы всех каналов, включая и класс `FileChannel`, реализуют интерфейс `AutoCloseable`, поэтому для автоматического закрытия файла вместо явного вызова метода `close()` можно воспользоваться оператором `try` с ресурсами. Именно такой подход и применяется в примерах из этой главы.

Затем следует получить буфер, который будет использоваться каналом, заключив буфер в оболочку существующего массива или динамически выделив оперативную память под буфер. В примерах из этой главы применяется выделение оперативной памяти под буфер, но вы вольны выбрать любой из этих двух способов. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах из этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`. Ниже приведена его общая форма, где *емкость* обозначает конкретную емкость буфера, а в итоге возвращается ссылка на буфер.

```
static ByteBuffer allocate(int емкость)
```

После создания буфера для канала вызывается метод `read()`, которому передается ссылка на буфер. Ниже приведена общая форма метода `read()`, которая употребляется в примерах, представленных далее в главе.

```
int read(ByteBuffer буфер) throws IOException
```

При каждом вызове метода `read()` указанный *буфер* заполняется данными из файла. Чтение осуществляется последовательно, а следовательно, при каждом вызове метода `read()` из файла в буфер читается следующая порция байтов. Метод `read()` возвращает количество фактически прочитанных байтов. При попытке прочитать данные по достижении конца файла возвращается значение **`-1`**.

В приведенном ниже примере программы все изложенное выше демонстрируется на практике. В этой программе данные из файла `test.txt` читаются через канал посредством явных операций ввода.

```
// Использовать канал ввода-вывода для чтения файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;
        Path filepath = null;

        // сначала получить путь к файлу
        try {
            filepath = Paths.get("test.txt");
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
            return;
        }

        // затем получить канал к этому файлу в
        // блоке оператора try с ресурсами
        try (SeekableByteChannel fChan = Files.newByteChannel(filepath))
        {

            // выделить память под буфер
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // читать данные из файла в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

            System.out.println();
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Эта программа действует следующим образом. Сначала создается объект типа `Path`, содержащий относительный путь к файлу `test.txt`. Ссылка на этот объект присваивается переменной `filepath`. Затем для создания канала, связанного с фай-

лом, вызывается метод newByteChannel(), которому передается ссылка на файл в переменной filepath. А поскольку никаких параметров открытия файла не указано, то файл по умолчанию открывается для чтения. Обратите внимание на то, что созданный в итоге канал является объектом, управляемым оператором try с ресурсами. Таким образом, канал автоматически закрывается в конце блока этого оператора. Далее в данной программе вызывается метод allocate() из класса ByteBuffer, чтобы выделить оперативную память под буфер для хранения содержимого файла во время чтения. Ссылка на этот буфер хранится в переменной экземпляра mBuf. После этого вызывается метод read(), и содержимое файла читается по очереди в буфер mBuf. Количество прочитанных байтов сохраняется в переменной count. Затем вызывается метод rewind(), чтобы подготовить буфер к чтению из него данных. Этот метод нужно вызвать потому, что после вызова метода read() текущая позиция находится в конце буфера. Ее следует возвратить в начало буфера, чтобы при вызове метода get() можно было прочитать байты данных из буфера mBuf. (Напомним, что метод get() определяется в классе ByteBuffer.) Буфер mBuf может содержать только байты данных, поэтому из метода get() возвращаются байты. Они приводятся к типу char, чтобы выводить на экран содержимое файла в текстовом виде. (В качестве альтернативы можно создать буфер, автоматически преобразующий байты в символы, а затем прочитать их из этого буфера.) По достижении конца файла метод read() возвращает значение **–1**. В таком случае программа завершается и канал автоматически закрывается.

Обратите внимание на следующий интересный момент: программа получает объект типа Path в пределах одного блока оператора try, а затем использует его в другом блоке оператора try для получения и манипулирования каналом, связанным с файлом по пути, который задается объектом типа Path. И хотя в таком подходе нет ничего плохого, во многих случаях его можно упростить, чтобы организовать ввод-вывод данных только в одном блоке try. В этом случае вызовы методов Paths.get() и newByteChannel() следуют друг за другом. В качестве примера ниже приведена переделанная версия программы из предыдущего примера, где применяется данный подход.

```
// Более компактный способ открытия канала.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;

        // Здесь канал открывается по пути, возвращаемому
        // методом Paths.get() в виде объекта типа Path.
        // Переменная filepath больше не нужна
        try (SeekableByteChannel fChan =
             Files.newByteChannel(Paths.get("test.txt")) )
        {
            // выделить память под буфер
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
```

```
                // читать данные из файла в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);
            System.out.println();
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

В этой версии программы переменная `filepath` больше не требуется, и оба исключения обрабатываются в одном и том же блоке оператора `try`. Такой подход компактнее, поэтому именно он и применяется в остальных примерах из этой главы. Разумеется, при написании прикладного кода возможны случаи, когда создание объекта типа `Path` должно быть отделено от получения канала. В подобных случаях применяется предыдущий подход.

Другой способ чтения файла подразумевает его сопоставление с буфером. Преимущество такого способа состоит в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения не требуется. Сопоставление и чтение содержимого файла осуществляется в ходе следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, как описано ранее. Затем получается канал к этому файлу. С этой целью вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`, а тип возвращаемого из него объекта приводится к типу `FileChannel`. Как упоминалось ранее, метод `newByteChannel()` возвращает объект типа `SeekableByteChannel`. Если используется выбираемая по умолчанию файловая система, этот объект может быть приведен к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить этот канал с буфером. Метод `map()` определяется в классе `FileChannel`, поэтому и требуется приведение к типу `FileChannel`. Ниже приведена общая форма объявления метода `map()`.

```
MappedByteBuffer map(
          FileChannel.MapMode способ, long позиция, long размер)
          throws IOException
```

В методе `map()` данные из файла сопоставляются с буфером в оперативной памяти. Значение параметра *способ* определяет вид разрешенной операции. Этот параметр может принимать одно из следующих допустимых значений:

MapMode.READ_ONLY	**MapMode.READ_WRITE**	**MapMode.PRIVATE**

Для чтения данных из файла следует указать значение MapMode.READ_ONLY, а для чтения и записи данных в файл — значение MapMode.READ_WRITE. Выбор значения MapMode.PRIVATE приводит к созданию закрытой копии файла, чтобы внесенные в буфере изменения не повлияли на основной файл. Место для начала сопоставления в пределах файла определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. Ссылка на буфер возвращается в виде объекта класса MappedByteBuffer, производного от класса ByteBuffer. Как только файл будет сопоставлен с буфером, его содержимое можно прочитать в файл из буфера. Такой способ демонстрируется в следующем примере программы:

```
// Использовать сопоставление для чтения данных из файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelRead {
    public static void main(String args[]) {

        // получить канал к файлу в блоке оператора try с ресурсами
        try ( FileChannel fChan =
            (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
        {

            // получить размер файла
            long fSize = fChan.size();

            // а теперь сопоставить файл с буфером
            MappedByteBuffer mBuf =
                 fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // читать байты из буфера и выводить их на экран
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());

            System.out.println();

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

В данной программе сначала создается путь к файлу, обозначаемый объектом типа Path, а затем открывается файл с помощью метода newByteChannel(). Получаемый в итоге канал приводится к типу FileChannel и сохраняется в переменной экземпляра fChan. Затем в результате вызова метода size() для канала получается размер файла. Далее вызывается метод map() по ссылке fChan на объект канала, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на буфер сохраняется в переменной экземпляра mBuf. Обратите внимание на то, что переменная mBuf объявляется как ссылка на объект типа MappedByteBuffer. Байты из буфера в переменной mBuf читаются непосредственно методом get().

Запись данных в файл через канал

Как и при чтении данных из файла, для записи данных в файл через канал имеется несколько способов. Рассмотрим сначала один из наиболее распространенных способов. Он предполагает выделение оперативной памяти под буфер вручную, запись в него данных, а затем выполнение явной операции записи этих данных в файл.

Прежде чем записать данные в файл, его следует открыть. Для этого нужно получить сначала объект типа `Path`, обозначающий путь к файлу, а затем использовать этот путь, чтобы открыть файл. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Как показано в предыдущем разделе, общая форма метода `newByteChannel()` такова:

```
static SeekableByteChannel newByteChannel(
       Path путь, OpenOption ... способ)
       throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Чтобы открыть файл для вывода, в качестве параметра *способ* следует передать значение `StandardOpenOption.WRITE`. Если файл еще не существует и его нужно создать, то следует указать также значение `StandardOpenOption.CREATE`. (Другие доступные значения стандартных параметров открытия файлов перечислены в табл. 21.7.) Как пояснялось в предыдущем разделе, интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Его реализует класс `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его.

Один из способов записи данных в файл через канал подразумевает явные вызовы метода `write()`. Сначала получается объект типа `Path`, обозначающий путь к файлу, а затем для открытия этого файла вызывается метод `newByteChannel()` и возвращаемый результат приводится к типу `FileChannel`. Далее выделяется оперативная память под буфер байтов, в который записываются выводимые в файл данные. Прежде чем данные будут записаны в файл, для буфера следует вызвать метод `rewind()`, чтобы обнулить его текущую позицию. (Каждая операция вывода в буфер увеличивает его текущую позицию. Поэтому перед записью в файл ее следует возвратить в исходное положение.) Далее для канала вызывается метод `write()`, которому передается буфер. Вся эта процедура демонстрируется в следующем примере программы, где весь английский алфавит записывается в файл `test.txt`:

```
// Записать данные в файл средствами системы ввода-вывода NIO.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelWrite {
```

```
    public static void main(String args[]) {

        // получить канал к файлу в блоке оператора try с ресурсами
        try ( FileChannel fChan = (FileChannel)
                Files.newByteChannel(Paths.get("test.txt"),
                                     StandardOpenOption.WRITE,
                                     StandardOpenOption.CREATE) )
        {
            // создать буфер
            ByteBuffer mBuf = ByteBuffer.allocate(26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // подготовить буфер к записи данных
            mBuf.rewind();

            // записать данные из буфера в выходной файл
            fChan.write(mBuf);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
            System.exit(1);
        }
    }
}
```

Следует отметить одну важную особенность данной программы. Как упоминалось ранее, после записи данных в буфер байтов `mBuf`, но перед их записью в файл для буфера `mBuf` вызывается метод `rewind()`. Это требуется для обнуления текущей позиции после записи данных в буфер `mBuf`. Не следует забывать, что после каждого вызова метода `put()` для буфера `mBuf` текущая позиция смещается. Поэтому текущую позицию необходимо возвратить в начало буфера, прежде чем вызывать метод `write()`. Если не сделать этого, метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Еще один способ обнуления буфера между операциями ввода и вывода подразумевает вызов метода `flip()` вместо метода `rewind()`. Метод `flip()` устанавливает для текущей позиции нулевое значение, а для предела — значение предыдущей текущей позиции. В приведенном выше примере емкость буфера совпадает с его пределом, поэтому метод `flip()` можно использовать вместо метода `rewind()`. Но эти два метода взаимозаменяемы далеко не всегда.

Как правило, буфер следует обнулять между любыми операциями чтения и записи. Например, в результате выполнения приведенного ниже цикла, составленного на основе предыдущего примера, английский алфавит будет записан в файл три раза. Обратите особое внимание на то, что метод `rewind()` вызывается каждый раз в промежутке между операциям чтения и записи.

```
int h=0; h<3; h++) {
    // записать заданное количество байтов в буфер
    for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));
```

```
    // подготовить буфер к записи данных
    mBuf.rewind();

    // записать данные из буфера в выходной файл
    fChan.write(mBuf);

    // снова подготовить буфер к записи данных
    mBuf.rewind();
}
```

В отношении рассматриваемой здесь программы следует также иметь в виду, что в процессе записи данных из буфера в файл первые 26 байт в файле будут содержать выводимые данные. Если файл `test.txt` существовал ранее, то после выполнения программы первые 26 байт в файле `test.txt` будут содержать алфавит, а остальная часть файла останется без изменения.

Еще один способ записи данных в файл подразумевает его сопоставление с буфером. Преимущество такого подхода заключается в том, что занесенные в буфер данные будут автоматически записаны в файл. Никаких явных операций записи не требуется. Для сопоставления и записи содержимого буфера в файла необходимо придерживаться следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, а затем создается канал к этому файлу, для чего вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`. Ссылку, возвращаемую методом `newByteChannel()`, следует привести к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить канал с буфером. Метод `map()` был подробно описан в предыдущем разделе, а здесь он упоминается ради удобства изложения. Ниже приведена его общая форма.

```
MappedByteBuffer map(
          FileChannel.MapMode способ, long позиция, long размер)
          throws IOException
```

Метод `map()` сопоставляет данные из файла с буфером в памяти. Значение параметра *способ* определяет разрешенные операции. Чтобы записать данные в файл, в качестве параметра *способ* следует указать значение `MapMode.READ_WRITE`. Место для начала сопоставления в файле определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. В итоге возвращается ссылка на буфер. Как только файл будет сопоставлен с буфером, в буфер можно вывести данные, которые будут автоматически записываться в файл. Поэтому никаких явных операций записи в канал не требуется.

Ниже приведена новая версия программы из предыдущего примера, переделанная таким образом, чтобы использовать сопоставление для записи данных в файл. Обратите внимание на то, что при вызове метода `newByteChannel()` в качестве параметра указывается значение `StandardOpenOption.READ`. Дело в том, что сопоставляемый буфер может использоваться только для чтения или же для чтения и записи. Таким образом, для записи в сопоставляемый буфер канал должен быть открыт как для чтения, так и для записи.

```
// Записать данные в сопоставляемый файл
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
```

```
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelWrite {
    public static void main(String args[]) {

        // получить канал к файлу в блоке try с ресурсами
        try ( FileChannel fChan = (FileChannel)
              Files.newByteChannel(Paths.get("test.txt"),
                                   StandardOpenOption.WRITE,
                                   StandardOpenOption.READ,
                                   StandardOpenOption.CREATE) )
        {

            // затем сопоставить файл с буфером
            MappedByteBuffer mBuf =
                fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать заданное количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Как видите, в данном примере отсутствуют явные операции записи непосредственно в канал. Буфер `mBuf` сопоставляется с файлом, поэтому изменения в буфере автоматически отражаются в основном файле.

Копирование файлов средствами системы ввода-вывода NIO

Система ввода-вывода NIO упрощает несколько видов файловых операций. Хотя здесь недостаточно места, чтобы рассмотреть все эти операции, приведенный ниже пример программы дает общее представление о доступных средствах. В этой программе файл копируется единственным методом `copy()` — статическим методом из класса `Files` в системе ввода-вывода NIO. У этого метода имеется несколько общих форм. Ниже приведена та общая форма, которая будет использоваться в примерах, представленных далее.

```
static Path copy(Path источник, Path адресат, CopyOption ... способ)
                 throws IOException
```

Файл, определяемый параметром *источник*, копируется в файл, обозначаемый параметром *адресат*. А порядок копирования определяется параметром *способ*. Это параметр переменной длины, поэтому он может отсутствовать. Если же он определен, то позволяет передать одно или несколько приведенных ниже значений, допустимых для всех файловых систем. В зависимости от конкретной реализации могут поддерживаться и другие значения.

StandardCopyOption.COPY_ATTRIBUTES	Запросить копирование атрибутов файла
StandardLinkOption.NOFOLLOW_LINKS	Не следовать по символическим ссылкам
StandardCopyOption.REPLACE_EXISTING	Перезаписать прежний файл

В приведенном ниже примере программы демонстрируется применение метода copy(). Исходный и результирующие файлы указываются в командной строке, причем исходный файл указывается первым. Обратите внимание на краткость программы. Если сравнить эту версию программы копирования файла с ее аналогом из главы 13, то окажется, что та часть программы, которая фактически копирует файл, существенно короче в представленной здесь версии на основе системы ввода-вывода NIO.

```
// Скопировать файл средствами системы ввода-вывода NIO.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

    public static void main(String args[]) {

        if(args.length != 2) {
            System.out.println("Применение: откуда и куда копировать");
            return;
        }

        try {
            Path source = Paths.get(args[0]);
            Path target = Paths.get(args[1]);

            // скопировать файл
            Files.copy(source, target,
                       StandardCopyOption.REPLACE_EXISTING);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Применение системы NIO для потокового ввода-вывода

Начиная с версии NIO.2, систему NIO можно использовать для открытия потока ввода-вывода. Получив объект типа Path, следует открыть файл, вызвав статический метод newInputStream() или newOutputStream(), определенный в классе Files. Эти методы возвращают поток ввода-вывода, связанный с указанным файлом. В любом случае поток ввода-вывода может быть затем использован так, как описано в главе 20, и для этого пригодны те же самые способы. Преимущество использования объекта типа Path для открытия файла заключается в том, что доступны все средства системы ввода-вывода NIO.

Для открытия файла с целью потокового ввода служит метод Files.newInputStream(). Он имеет следующую общую форму:

```
static InputStream newInputStream(Path путь, OpenOption ... способ)
       throws IOException
```

где параметр *путь* обозначает открываемый файл, а параметр *способ* — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе `StandardOpenOption`. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку ввода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра передано значение `StandardOpenOption.READ`. После открытия файла можно использовать любой из методов, определенных в классе `InputStream`. Например, метод `read()` можно использовать для чтения байтов из файла.

В приведенном ниже примере программы демонстрируется применение потокового ввода-вывода на основе системы NIO. Этот пример содержит версию программы `ShowFile` из главы 13, переделанную таким образом, чтобы для открытия файла и получения потока ввода-вывода использовались средства системы NIO. Нетрудно заметить, что эта версия программы очень похожа на первоначальный ее вариант, за исключением используемого интерфейса `Path` и метода `newInputStream()`.

```
/* Эта программа выводит текстовый файл, используя код
   потокового ввода-вывода на основе системы NIO.
   Требуется установка комплекта JDK, начиная с версии 7

   Чтобы воспользоваться этой программой, укажите имя файла,
   который требуется просмотреть. Например, чтобы просмотреть
   файл TEST.TXT, введите в режиме командной строки
   следующую команду:

   java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // сначала удостовериться, что указано имя файла
        if(args.length != 1) {
            System.out.println("Применение: ShowFile имя_файла");
            return;
        }

        // открыть файл и получить связанный с ним поток ввода-вывода
        try (InputStream fin = Files.newInputStream(Paths.get(args[0])))
        {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Поток ввода-вывода, возвращаемый методом newInputStream(), является обычным, и поэтому он применяется как и любой другой поток ввода-вывода. Например, один поток ввода можно заключить в оболочку другого, буферизованного потока ввода, например, типа BufferedInputStream, чтобы обеспечить буферизацию так, как показано ниже. В итоге все операции чтения будут автоматически буферизованы.

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Для открытия файла с целью вывода служит метод Files.newOutputStream(), который имеет следующую общую форму:

```
static OutputStream newOutputStream(Path путь, OpenOption ... способ)
       throws IOException
```

где параметр *путь* обозначает открываемый файл, а параметр способ — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе StandardOpenOption. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку вывода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра переданы значения StandardOpenOption.WRITE, StandardOpenOption.CREATE и StandardOpenOption.TRUNCATE_EXISTING.

Метод newOutputStream() применяется таким же способом, как и описанный ранее метод newInputStream(). После открытия файла можно вызвать любой метод, определенный в классе OutputStream. Например, вызвать метод write() для записи байтов в файл. Кроме того, поток вывода можно заключить в поток вывода байтов типа BufferedOutputStream, чтобы буферизовать его.

В приведенном ниже примере программы демонстрируется применение метода newOutputStream(). В этой программе английский алфавит записывается в файл test.txt. Обратите внимание на использование буферизованного ввода-вывода.

```
// Продемонстрировать потоковый вывод на основе системы NIO
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
    public static void main(String args[])
    {
        // открыть файл и получить связанный с ним поток вывода
        try (OutputStream fout =
                 new BufferedOutputStream(
                     Files.newOutputStream(Paths.get("test.txt"))) )
        {
            // вывести в поток заданное количество байтов
            for(int i=0; i < 26; i++)
                fout.write((byte)('A' + i));

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Применение системы ввода-вывода NIO для операций в файловой системе

В начале главы 20 был представлен класс `File`, входящий в пакет `java.io`. Как упоминалось ранее, класс `File` обращается к файловой системе и оперирует различными атрибутами файлов, обозначающими, например, доступ только для чтения, скрытый файл и т.д. Он служит также для получения сведений о пути к файлу. Начиная с версии JDK 7 интерфейсы и классы, определенные в системе ввода-вывода NIO.2, предоставляют лучший способ выполнения этих операций. К их преимуществам относятся улучшенная поддержка символических ссылок, обхода дерева каталогов, усовершенствованная обработка метаданных и многое другое. В последующих подразделах приведены примеры двух наиболее распространенных операций в файловой системе: получения сведений о пути к файлу и самом файле, а также сведений о содержимом каталога.

Помните! Если требуется обновить устаревший код, в котором применяется класс `java.io.File`, новым кодом, в котором применяется интерфейс `Path`, воспользуйтесь методом `toPath()`, чтобы получить экземпляр интерфейса `Path` из экземпляра класса `File`.

Получение сведений о пути к файлу и самом файле

Сведения о пути к файлу могут быть получены методами, определенными в интерфейсе `Path`. Некоторые атрибуты файлов, описываемые в интерфейсе `Path` (например, скрытый файл), получаются методами, определенными в классе `Files`. В рассматриваемом здесь примере употребляются такие методы из интерфейса `Path`, как `getName()`, `getParent()` и `toAbsolutePath()`, а также методы `isExecutable()`, `isHidden()`, `isReadable()`, `isWritable()` и `exists()` из класса `Files` (они представлены в табл. 21.5 и 21.6).

Внимание! Такими методами, как `isExecutable()`, `isReadable()`, `isWritable()` и `exists()`, следует пользоваться осторожно, поскольку состояние файловой системы после их вызова может измениться, что может привести к нарушению нормальной работы программы и отрицательно сказаться на состоянии защиты системы.

Другие атрибуты файлов получаются по запросу из списка, создаваемого при вызове метода `Files.readAttributes()`. В рассматриваемом здесь примере программы этот метод вызывается для получения связанного с файлом объекта типа `BasicFileAttributes`, но аналогичный общий подход можно применить и к другим типам атрибутов.

В приведенном ниже примере программы демонстрируется применение некоторых методов из интерфейса `Path` и класса `Files` наряду с методами из интерфейса `BasicFileAttributes`. В этой программе подразумевается, что файл `test.txt` находится в каталоге `examples`, входящем в текущий каталог.

```
// Получить сведения о пути к файлу и самом файле
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
```

```
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\\test.txt");

        System.out.println("Имя файла: " + filepath.getName(1));
        System.out.println("Путь к файлу: " + filepath);
        System.out.println("Абсолютный путь к файлу: " +
                           filepath.toAbsolutePath());
        System.out.println(
              "Родительский каталог: " + filepath.getParent());
        if(Files.exists(filepath))
            System.out.println("Файл существует");
        else
            System.out.println("Файл не существует");

        try {
            if(Files.isHidden(filepath))
                System.out.println("Файл скрыт");
            else
                System.out.println("Файл не скрыт");
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }

        Files.isWritable(filepath);
        System.out.println("Файл доступен для записи");

        Files.isReadable(filepath);
        System.out.println("Файл доступен для чтения");

        try {
            BasicFileAttributes attribs =
               Files.readAttributes(filepath, BasicFileAttributes.class);

            if(attribs.isDirectory())
                System.out.println("Это каталог");
            else
                System.out.println("Это не каталог");

            if(attribs.isRegularFile())
                System.out.println("Это обычный файл");
            else
                System.out.println("Это не обычный файл");

            if(attribs.isSymbolicLink())
                System.out.println("Это символическая ссылка");
            else
                System.out.println("Это не символическая ссылка");

            System.out.println("Время последней модификации файла: " +
                               attribs.lastModifiedTime());
            System.out.println("Размер файла: " + attribs.size() +
                               " байтов");
        } catch(IOException e) {
            System.out.println("Ошибка чтения атрибутов: " + e);
        }
    }
}
```

Если запустить эту программу на выполнение из каталога MyDir, в котором имеется каталог examples, содержащий файл test.txt, то в конечном итоге будет выведен результат, аналогичный приведенному ниже. (Разумеется, размер файла и его временны*е* характеристики будут иными.)

```
Имя файла: test.txt
Путь к файлу: examples\test.txt
Абсолютный путь к файлу: C:\MyDir\examples\test.txt
Родительский каталог: examples
Файл существует
Файл не скрыт
Файл доступен для записи
Файл доступен для чтения
Это не каталог
Это обычный файл
Это не символическая ссылка
Время последней модификации файла: 2014-01-01T18:20:46.380445Z
Размер файла: 18 байтов
```

Если вы пользуетесь компьютером с файловой системой FAT (т.е. файловой системой DOS), то попытайтесь применить методы, определенные в интерфейсе DosFileAttributes. А если вы пользуетесь системой, совместимой с POSIX, то попробуйте применить методы, определенные в интерфейсе PosixFileAttributes.

Получение содержимого каталога

Если путь описывает каталог, можно прочитать содержимое этого каталога, используя статические методы, определенные в классе Files. Для этого следует сначала получить поток ввода из каталога, вызвав метод newDirectoryStream() и передав ему объект типа Path, обозначающий каталог. Ниже приведена одна из форм метода newDirectoryStream().

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу)
       throws IOException
```

Здесь параметр *путь_к_каталогу* инкапсулирует путь к конкретному каталогу. Этот метод возвращает объект типа DirectoryStream<Path>, применяемый для получения содержимого каталога. Он генерирует исключение типа IOException при возникновении ошибки ввода-вывода, а также исключение типа NotDirectoryException (его класс является производным от класса IOException), если указанный путь не приводит к каталогу. Кроме того, может быть сгенерировано исключение типа SecurityException, если доступ к каталогу запрещен.

Класс DirectoryStream<Path> реализует интерфейс AutoCloseable, поэтому объектом этого класса можно управлять в блоке оператора try с ресурсами. Этот класс реализует также интерфейс Iterable<Path>. Это означает, что содержимое каталога можно получить, перебрав содержимое объекта типа DirectoryStream. При переборе каждая запись каталога представлена экземпляром интерфейса Path. Простейший способ перебрать объект типа DirectoryStream — организовать цикл for в стиле for each. Следует, однако, иметь в виду, что итератор, реализуемый в классе DirectoryStream<Path>, может быть получен только один раз для каждого экземпляра. Следовательно, метод iterator() может быть вызван, а цикл for в стиле for each — выполнен только один раз.

В следующем примере программы выводится содержимое каталога MyDir:

```
// Вывести содержимое каталога. Требуется установка комплекта JDK,
// начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // получить и обработать поток ввода каталога
        // в блоке оператора try
        try ( DirectoryStream<Path> dirstrm =
              Files.newDirectoryStream(Paths.get(dirname)) )
        {
            System.out.println("Каталог " + dirname);

            // Класс DirectoryStream реализует интерфейс Iterable,
            // поэтому для вывода содержимого каталога можно
            // организовать цикл for в стиле for each
            for(Path entry : dirstrm) {
                BasicFileAttributes attribs =
                  Files.readAttributes(entry, BasicFileAttributes.class);

                if(attribs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " не является каталогом.");
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Эта программа выводит следующий результат:

```
Каталог \MyDir
      DirList.class
      DirList.java
<DIR> examples
      Test.txt
```

Содержимое каталога можно отфильтровать двумя способами. Самый простой из них — воспользоваться следующей общей формой метода newDirectory Stream():

```
static DirectoryStream<Path> newDirectoryStream(
       Path путь_к_каталогу, String шаблон)
       throws IOException
```

В этой форме получаются только те файлы, имена которых совпадают с заданным *шаблоном*. В качестве параметра *шаблон* можно указать полное имя файла или маску. *Маска* — это символьная строка, определяющая глобальный или общий шаблон, с которым будет совпадать один или несколько файлов, и содержащая общеупотребительные метасимволы ***** и **?**. Они соответствуют любому количеству символов и любому одиночному символу соответственно. Ниже приведены другие шаблоны для сопоставления с маской.

******	Совпадает с любым количеством различных символов в каталогах
[*символы*]	Совпадает с любым из указанных ***символов***. Символы ***** и **?** среди указанных ***символов*** будут рассматриваться как обычные символы, а не как метасимволы. Через дефис можно указать пределы сопоставления с шаблоном, например, **[x-z]**
{*список_масок*}	Совпадает с любой из масок, задаваемых в ***списке_масок*** через запятую

Метасимволы ***** и **?** можно указать, используя последовательности символов ***** и **\?**, а для того чтобы указать знак **** — последовательность символов ****. Можете поэкспериментировать с маской, подставив ее в вызов метода `newDirectoryStream()` из предыдущего примера программы следующим образом:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

Еще один способ отфильтровать каталог — воспользоваться приведенной ниже общей формой метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу,
       DirectoryStream.Filter<? super Path> фильтр_файлов)
                throws IOException
```

Здесь `DirectoryStream.Filter` — это интерфейс, в котором определяется следующий метод:

```
boolean accept(T элемент) throws IOException
```

В данном случае типом **T** будет `Path`. Если требуется включить указанный *элемент* в список, возвращается логическое значение `true`, а иначе — логическое значение `false`. Эта форма метода `newDirectoryStream()` предоставляет возможность отфильтровать каталог по другому критерию, кроме имени файла. В частности, каталог можно отфильтровать по размеру, дате создания, дате модификации или атрибуту.

Этот процесс демонстрируется в приведенном ниже примере программы. В этой программе перечисляются только те файлы, которые доступны для записи.

```
// Вывести только те файлы из каталога,
// которые доступны для записи

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // создать фильтр, возвращающий логическое значение true
```

```
        // только в отношении доступных для записи файлов
        DirectoryStream.Filter<Path> how =
                    new DirectoryStream.Filter<Path>() {
            public boolean accept(Path filename) throws IOException {
                if(Files.isWritable(filename)) return true;
                return false;
            }
        };

        // получить и использовать поток ввода из каталога
        // только доступных для записи файлов
        try (DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname), how) )
        {
            System.out.println("Каталог " + dirname);

            for(Path entry : dirstrm) {
              BasicFileAttributes attribs =
                Files.readAttributes(entry, BasicFileAttributes.class);

              if(attribs.isDirectory())
                System.out.print("<DIR> ");
              else
                System.out.print("      ");

              System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " не является каталогом.");
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Обход дерева каталогов с помощью метода `walkFileTree()`

В предыдущих примерах получалось содержимое только одного каталога. Но иногда требуется получить список файлов из дерева каталогов. В прошлом решить подобную задачу было нелегко, но система ввода-вывода NIO.2 значительно упрощает ее решение, поскольку теперь из класса `Files` можно вызвать метод `walkFileTree()`, способный обработать дерево каталогов. Этот метод имеет две общие формы объявления. В примерах, приведенных в этой главе, употребляется следующая форма этого метода:

```
static Path walkFileTree(Path корень, FileVisitor<? extends Path> fv)
        throws IOException
```

Исходная точка обхода дерева каталогов передается в качестве параметра *корень*. Экземпляр интерфейса `FileVisitor` передается в качестве параметра *fv*. Реализация интерфейса `FileVisitor` определяет способ обхода дерева каталогов, а также позволяет обращаться к сведениям о каталоге. При возникновении ошибки ввода-вывода генерируется исключение типа `IOException`. Кроме того, может быть сгенерировано исключение типа `SecurityException`.

В интерфейсе `FileVisitor` определяется, каким образом посещаются файлы при обходе дерева каталогов. Этот обобщенный интерфейс объявляется следующим образом:

```
interface FileVisitor<T>
```

При вызове метода `walkFileTree()` в качестве параметра типа **`T`** указывается интерфейс `Path` (или любой производный от него тип). В интерфейсе `FileVisitor` определены методы, перечисленные в табл. 21.11.

Таблица 21.11. Методы из интерфейса `FileVisitor`

Метод	Описание
`FileVisitResult postVisitDirectory(T` ***каталог*****`, IOException`** ***исключение*****`) throws IOException`**	Вызывается после посещения каталога. Каталог передается в качестве параметра ***каталог***, а любое исключение типа **`IOException`** — в качестве параметра ***исключение***. Если параметр ***исключение*** принимает пустое значение **`null`**, то никакого исключения не происходит. Возвращает полученный результат
`FileVisitResult preVisitDirectory(T` ***каталог*****`, BasicFileAttributes`** ***атрибуты*****`) throws IOException`**	Вызывается перед посещением каталога. Каталог передается в качестве параметра ***каталог***, а связанные с ним атрибуты — в качестве параметра ***атрибуты***. Возвращает полученный результат. Чтобы исследовать каталог, следует возвратить значение **`FileVisitResult.CONTINUE`**
`FileVisitResult visitFile(T` ***файл*****`, BasicFileAttributes`** ***атрибуты*****`) throws IOException`**	Вызывается при посещении файла. Файл передается в качестве параметра ***файл***, а связанные с ним атрибуты — в качестве параметра ***атрибуты***. Возвращает полученный результат
`FileVisitResult visitFileFailed(T` ***файл*****`, IOException`** ***исключение*****`) throws IOException`**	Вызывается при неудачной попытке посетить файл. Файл, который не удалось посетить, передается в качестве параметра ***файл***, а исключение типа **`IOException`** — в качестве параметра ***исключение***. Возвращает полученный результат

Обратите внимание на то, что каждый метод возвращает значение из перечисления `FileVisitResult`. В этом перечислении определяются следующие значения:

`CONTINUE`	**`SKIP_SIBLINGS`**	**`SKIP_SUBTREE`**	**`TERMINATE`**

В общем, для продолжения обхода каталога и находящихся в нем каталогов метод должен возвратить значение `CONTINUE`. Для того чтобы пропустить каталог и его содержимое, а также предотвратить вызов метода `postVisitDirectory()`, из метода `preVisitDirectory()` должно быть возвращено значение `SKIP_SIBLINGS`, чтобы пропустить только каталог и подкаталоги — значение `SKIP_SUBTREE`, а для того чтобы остановить обход каталога — значение `TERMINATE`.

Безусловно, можно создать собственный класс для обхода каталогов и реализовать в нем методы, определенные в интерфейсе `FileVisitor`, но обычно так не поступают, поскольку предоставляется простая их реализация в классе `SimpleFileVisitor`. В этом случае достаточно переопределить реализацию

по умолчанию одного или нескольких нужных методов. Ниже приведен краткий пример программы, демонстрирующий этот процесс. В этой программе выводятся все файлы из дерева каталогов, в корне которого находится каталог \MyDir. Обратите внимание на краткость этой программы.

```
// Простой пример применения метода walkFileTree()
// для вывода дерева каталогов. Требуется установка
// комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// создать специальную версию класса SimpleFileVisitor,
// в которой переопределяется метод visitFile()
class MyFileVisitor extends SimpleFileVisitor<Path> {
      public FileVisitResult visitFile(
                            Path path, BasicFileAttributes attribs)
      throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
}

class DirTreeList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        System.out.println("Дерево каталогов, начиная с каталога " +
                           dirname + ":\n");
        try {
          Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
        } catch (IOException exc) {
           System.out.println("Ошибка ввода-вывода");
        }
    }
}
```

Ниже приведен примерный результат, выводимый данной программой для обхода того же каталога MyDir, что и прежде. В данном примере подкаталог examples содержит только один файл MyProgram.java.

```
Дерево каталогов, начиная с каталога \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

В данной программе класс MyFileVisitor расширяет класс SimpleFileVisitor, переопределяя только метод visitFile(), который просто выводит их, хотя совсем не трудно достичь и более сложных функциональных возможностей. Например, можно было бы отфильтровать файлы или выполнить над ними такие действия, как копирование на резервное устройство. Ради простоты в данном примере для переопределения метода visitFile() выбран именованный класс, но вместо него ничто не мешает воспользоваться анонимным внутренним классом. И последнее замечание: средствами класса java.nio.file.WatchService можно отследить изменения в каталоге.

Примеры организации канального ввода-вывода до версии JDK 7

Прежде чем завершить эту главу, следует рассмотреть еще одну особенность системы ввода-вывода NIO. В приведенных выше примерах употреблялись некоторые из новых средств, внедренных в систему ввода-вывода NIO, начиная с версии JDK 7. Но ведь имеется еще немало кода, который написан до версии JDK 7 и требует надлежащего сопровождения, а возможно, и преобразования для применения новых средств. Поэтому в последующих разделах будет показано, как организовать чтение и запись в файлы средствами системы ввода-вывода NIO, доступными до версии JDK 7. Некоторые из приведенных выше примеров переделаны таким образом, чтобы использовать предыдущие средства системы ввода-вывода NIO, а не новые средства, поддерживаемые в системе ввода-вывода NIO.2. Это означает, что приведенные далее примеры пригодны для версий Java, выпущенных до версии JDK 7.

Главное отличие прежнего кода от нового, в котором применяется система ввода-вывода NIO, заключается в интерфейсе `Path`, который был внедрен в версии JDK 7. Следовательно, для описания файла или открытия канала к нему в прежнем коде не применяется интерфейс `Path`. Кроме того, в прежнем коде не применяются операторы `try` с ресурсами, поскольку автоматическое управление ресурсами также было внедрено только в версии JDK 7.

Помните! В примерах программ из этого раздела описывается действие унаследованного кода, в котором применяется система ввода-вывода NIO. Материал этого раздела предназначен для тех программистов, которые продолжают работать с унаследованным кодом или пользуются компилятором, выпущенным до версии JDK 7. В новом коде должны применяться средства системы ввода-вывода NIO, внедренные в версии JDK 7.

Чтение из файла до версии JDK 7

В этом разделе рассматриваются два предыдущих примера канального ввода из файла, переделанных для использования средств системы ввода-вывода NIO, доступных только до версии JDK 7. В первом примере для чтения данных из файла сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция чтения. Во втором примере производится сопоставление файла с буфером, автоматизирующее весь процесс чтения данных из файла.

Если для чтения данных из файла через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока ввода типа `FileInputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока ввода типа `FileInputStream`, чтобы получить канал к открытому файлу. Ниже приведена общая форма этого метода.

```
FileChannel getChannel()
```

В этой форме возвращается объект типа `FileChannel`, инкапсулирующий канал для файловых операций. Далее вызывается метод `allocate()` для выделения оперативной памяти под буфер. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`, как пояснялось ранее.

В следующем примере программы демонстрируется чтение и вывод содержимого файла `test.txt` через канал с использованием явных операций ввода для версий Java, выпущенных до JDK 7:

```
// Использовать каналы для чтения данных из файла.
// Версия до JDK 7
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // сначала открыть файл для ввода
            fIn = new FileInputStream("test.txt");

            // затем получить канал к этому файлу
            fChan = fIn.getChannel();

            // выделить оперативную память под буфер
            mBuf = ByteBuffer.allocate(128);

            do {
            // читать данные в буфер
            count = fChan.read(mBuf);

            // прекратить чтение по достижении конца файла
            if(count != -1) {

                // подготовить буфер к чтению из него данных
                mBuf.rewind();

                // читать байты данных из буфера и
                // выводить их на экран как символы
                for(int i=0; i < count; i++)
                    System.out.print((char)mBuf.get());
            }
        } while(count != -1);

        System.out.println();

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
```

```
            }
            try {
                if(fIn != null) fIn.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла.");
            }
        }
    }
}
```

Обратите внимание на то, что в данной программе файл открывается с помощью конструктора класса `FileInputStream`, а ссылка на создаваемый объект этого класса присваивается переменной экземпляра `fIn`. А когда вызывается метод `getChannel()` по ссылке на этот объект в переменной экземпляра `fIn`, то создается канал, подключаемый к открытому файлу. После этого программа работает таким же образом, как и в рассмотренной ранее версии для JDK 7. Напомним вкратце, что сначала вызывается метод `allocate()` из класса `ByteBuffer`, чтобы выделить оперативную память под буфер для хранения содержимого, прочитанного из файла. Для этого создается буфер байтов, поскольку класс `FileChannel` оперирует байтами. Ссылка на этот буфер хранится в переменной экземпляра `mBuf`. Затем содержимое файла читается по очереди в буфер `mBuf` методом `read()`. Количество прочитанных байтов хранится в переменной `count`. Далее вызывается метод `rewind()`, чтобы подготовить буфер к чтению данных. Этот метод приходится вызывать потому, что после вызова метода `read()` текущая позиция находится в конце буфера, а ее следует возвратить в начало буфера, чтобы все байты данных могли быть прочитаны из буфера `mBuf` при вызове метода `get()`. По достижении конца файла метод `read()` возвратит значение **`-1`**. Когда это произойдет, программа завершится, закрыв канал и файл явным образом.

Другой способ чтения данных из файла состоит в его сопоставлении с буфером. Как пояснялось ранее, основное преимущество такого подхода заключается в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения для этого не требуется. Чтобы сопоставить и прочитать содержимое файла, используя средства системы ввода-вывода NIO, доступные до версии JDK 7, следует открыть сначала файл с помощью потока ввода типа `FileInputStream`, а затем получить канал к этому файлу, вызвав метод `getChannel()` для файлового объекта. После этого вызывается метод `map()` для объекта типа `FileChannel`, чтобы сопоставить полученный канал с буфером, как описано ранее.

Ниже приведена версия программы из предыдущего примера, переделанная таким образом, чтобы для сопоставления файла с буфером использовались только средства системы ввода-вывода NIO, доступные до версии JDK 7.

```
// Использовать сопоставление для чтения данных из файла.
// Версия до JDK 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
```

```
        FileChannel fChan = null;
        long fSize;
        MappedByteBuffer mBuf;

        try {
            // сначала открыть файл для ввода
            fIn = new FileInputStream("test.txt");

            // затем получить канал к этому файлу
            fChan = fIn.getChannel();

            // получить размер файла
            fSize = fChan.size();

            // а теперь сопоставить файл с буфером
            mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // читать байты из буфера и выводить их на экран
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
            try {
                if(fIn != null) fIn.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла.");
            }
        }
    }
}
```

В данной программе файл открывается с помощью конструктора класса `FileInputStream`, а ссылка на создаваемый объект этого класса присваивается переменной экземпляра `fIn`. Затем вызывается метод `getChannel()` по ссылке на этот объект в переменной экземпляра `fIn`, чтобы создать канал, подключаемый к открытому файлу. Далее определяется размер файла. После этого вызывается метод `map()`, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на этот буфер сохраняется в переменной экземпляра `mBuf`. И наконец, вызывается метод `get()`, чтобы прочитать байты из буфера `mBuf`.

Запись в файл до версии JDK 7

В этом разделе представлены два предыдущих примера канального вывода данных в файл, переделанные таким образом, чтобы использовать только средства системы ввода-вывода NIO, доступные до версии JDK 7. В первом примере для записи данных в файл сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция записи. А во втором примере производится сопоставление файла с буфером, автоматизирующее весь

процесс записи данных в файл. Но в обоих случаях ни интерфейс `Path`, ни оператор `try` с ресурсами не применяются, поскольку они доступны лишь с версии JDK 7.

Если для записи данных в файл через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока вывода типа `FileOutputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока вывода типа `FileOutputStream`, чтобы получить канал к открытому файлу, а после этого — метод `allocate()`, чтобы выделить оперативную память под буфер, как описано в предыдущем разделе. Далее записываемые в файл данные размещаются в этом буфере и вызывается метод `write()` для открытого канала. Вся эта процедура демонстрируется в приведенном ниже примере программы. В этой программе английский алфавит записывается в файл `test.txt`.

```
// Записать данные в файл средствами системы ввода-вывода NIO.
// Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // сначала открыть файл для вывода данных
            fOut = new FileOutputStream("test.txt");

            // затем получить канал к файлу для вывода данных
            fChan = fOut.getChannel();

            // создать буфер
            mBuf = ByteBuffer.allocate(26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // подготовить буфер к записи данных
            mBuf.rewind();

            // записать данные из буфера в выходной файл
            fChan.write(mBuf);

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
            try {
                if(fOut != null) fOut.close(); // закрыть файл
```

```
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла.");
            }
        }
    }
}
```

Вызов метода `rewind()` по ссылке на буфер в переменной экземпляра `mBuf` требуется для возврата текущей позиции в начало буфера `mBuf` после записи данных. Напомним, что при каждом вызове метода `put()` текущая позиция продвигается к концу буфера. Поэтому, прежде чем вызвать метод `write()`, следует установить текущую позицию в начало буфера. В противном случае метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Если для записи данных в файл производится сопоставление файла с буфером и для этой цели используются версии Java, выпущенные до версии JDK 7, то сначала для выполнения операций чтения и записи создается объект класса `RandomAccessFile`, чтобы открыть файл. Для открываемого файла требуется разрешение на чтение и запись. Затем для данного объекта вызывается метод `map()`, чтобы сопоставить открытый файл с буфером. Далее записываемые данные размещаются в буфере. А поскольку буфер сопоставлен с файлом, то любые изменения в нем автоматически отражаются в файле. Таким образом, никаких явных операций записи в канал не требуется.

Ниже приведена версия программы из предыдущего примера, переделанная для сопоставления файла с буфером.

```
// Записать данные в сопоставленный файл. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // получить канал к открытому файлу
            fChan = fOut.getChannel();

            // затем сопоставить файл с буфером
            mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
```

```
                System.out.println("Ошибка закрытия канала.");
            }
            try {
                if(fOut != null) fOut.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла.");
            }
        }
    }
}
```

Как видите, в данном примере нет никаких явных операций записи непосредственно в канал. Благодаря тому что буфер `mBuf` сопоставляется с файлом, изменения в буфере автоматически отражаются в базовом файле.