

# 10

## Сетевые клиенты

**В** большинстве приложений, которые разрабатываются сегодня, необходимо использовать сетевые функции. Автономное приложение Java — сегодня уже редкое явление. Поэтому настоящая глава, посвященная теме сетевых клиентов, будет полезна большинству разработчиков, которые в настоящее время пишут приложения Java.

Сетевая программа подразумевает коммуникации (передачу информации) между клиентом и сервером. Клиент — это обычно приложение, делающее запросы на получение информации (контента) или выполнение определенного действия от службы, а сервер — это сетевое приложение, которое предоставляет контент и службы многим клиентам. Данная глава посвящена теме клиентов. А в главе 11, “Сетевые серверы”, мы рассмотрим примеры кодов, связанные с работой сервера.

За исключением кодов, связанных с чтением Web-страницы через протокол HTTP, остальные коды в данной главе относятся к программированию на уровне сокетов. *Сокеты* — это реализации сетевого взаимодействия на низком уровне. Для многих задач нужно использовать протокол, который находится выше уровня сокетов, такой как HTTP, SMTP или POP. Эти сетевые протоколы высокого уровня реализованы в дополнительных классах Java или сторонних API.

В пакете `java.net` содержатся функции для работы с сетевыми клиентами, которые мы будем использовать в данной главе.

В J2EE, рассмотрение которого выходит за рамки данной книги, предусмотрено намного больше сетевых служб, включая полную поддержку серверной части Web-приложений на Java. К сетевым технологиям, включенным в J2EE, относятся сервлеты, EJB и JMS.

### Подключение к серверу

```
String serverName = "www.timothyfisher.com";  
Socket sock = new Socket(serverName, 80);
```

В этом фрагменте кода мы подключаемся к серверу по протоколу TCP/IP с помощью класса `Socket` языка Java. Во время создания объекта `sock` в данном коде осуществляется подключение сокета к серверу, заданному в переменной `serverName` — в данном случае, это `www.timothyfisher.com` и порт 80.

Создав `Socket`, вы обязательно должны закрыть сокет, когда закончите с ним работать. Для этого нужно вызвать метод `close()` для экземпляра класса `Socket`, с которым вы работаете.

В Java поддерживаются и другие способы подключения к серверу, подробное рассмотрение которых выходит за рамки данной книги. Например, можно использовать класс `URL`, чтобы открыть `URL` и прочитать его содержимое. Подробнее об использовании класса `URL` читайте в данной главе в разделе “Чтение Web-страницы через протокол HTTP”.

### Нахождение IP-адресов и доменных имен

```
// найти IP-адрес для заданного домена
String hostName = "www.timothyfisher.com";
String ip =
    InetAddress.getByName(hostName).getHostAddress();

// найти доменное имя для IP-адреса
String ipAddress = "66.43.127.5";
String hostName =
    InetAddress.getByName(ipAddress).getHostName();
```

В данном фрагменте кода мы получаем имя хоста с помощью IP-адреса удаленного хоста, а также получаем IP-адрес с помощью имени удаленного хоста. Для выполнения обеих этих задач используется класс `InetAddress`.

Для создания экземпляра класса `InetAddress` используется статический метод `getByName()` класса `InetAddress`. При создании экземпляра класса `InetAddress` методу `getByName()` можно передать либо IP-адрес, либо имя хоста. Создав экземпляр класса `InetAddress`, мы можем вызвать метод `getHostAddress()`, который возвращает IP-адрес в виде объекта `String`. Если нам уже известен IP-адрес клиента, то определить его имя можно с помощью вызова метода `getHostName()`, которое возвращается в виде объекта класса `String`. Если имя хоста нельзя определить, то метод `getHostName()` возвращает IP-адрес.

### Обработка сетевых ошибок

```
try {
    // Соединиться с сетевым хостом для
    // выполнения сетевого ввода-вывода
}
```

```
catch (UnknownHostException ex) {
    System.err.println("Неизвестный хост.");
}
catch (NoRouteToHostException ex) {
    System.err.println("Недоступный хост.");
}
catch (ConnectException ex) {
    System.err.println("Соединение отвергнуто.");
}
catch (IOException ex) {
    System.err.println(ex.getMessage());
}
```

В данном фрагменте кода иллюстрируется ряд исключений, которые нужно обработать при выполнении сетевых операций.

Первое исключение, которое мы пытаемся обработать в коде, — это `UnknownHostException`, относящееся к подклассу `IOException`. Данное исключение генерируется, если нельзя определить IP-адрес.

Исключения `NoRouteToHostException` и `ConnectException` являются подклассами `SocketException`. Исключение `NoRouteToHostException` говорит о том, что ошибка произошла при попытке соединения сокета с удаленным адресом и портом. Обычно удаленный хост бывает недоступен по причине проблем с промежуточным брандмауэром или маршрутизатором. Исключение `ConnectException` генерируется в случае, если соединение с удаленным хостом отвергнуто. Исключение `IOException` генерируется при возникновении других распространенных сетевых ошибок.

В примерах кодов в данной главе и в главе 11 не включены коды обработки ошибок. Но в своих приложениях, в которых используются сетевые функции, вы обязательно должны стараться обработать эти исключения.

### Чтение текста

```
BufferedReader in =  
    new BufferedReader(new InputStreamReader(  
        socket.getInputStream()));  
String text = in.readLine();
```

В этом фрагменте кода предполагается, что мы уже предварительно создали сокет и подключили его к серверу, с которого хотим прочитать текст. Более подробно о том, как создавать экземпляр сокета, говорится в разделе “Подключение к серверу” данной главы. Получив экземпляр сокета, мы вызываем метод `getInputStream()`, чтобы получить ссылку на входной поток данных сокета. Затем создаем объект `InputStreamReader` и используем его для создания экземпляра `BufferedReader`, чтобы считывать текст из сети с помощью метода `readLine()` класса `BufferedReader`.

Использование объекта `BufferedReader`, как в данном примере, позволяет эффективно считывать символы, массивы и строки. Если бы нас интересовало чтение только очень небольших объемов данных, мы могли бы делать это непосредственно из объекта `InputStreamReader`, не используя `BufferedReader`. Ниже показано, как считывать данные в символьный массив с помощью только объекта `InputStreamReader`:

```
InputStreamReader in =  
    new InputStreamReader(socket.getInputStream());  
String text = in.read(charArray, offset, length);
```

В данном примере данные считываются из входного потока данных в массив символов, заданный параметром `charArray`. Символы будут помещаться в массиве, начиная с точки смещения, которая определяется параметром `offset`. Максимальное количество считываемых символов определяется параметром `length`.

### Запись текста

```
PrintWriter out =  
    new PrintWriter(socket.getOutputStream(), true);  
out.print(msg);  
out.flush();
```

В этом фрагменте кода предполагается, что мы уже предварительно создали сокет и подключили его к серверу, которому мы хотим передать текст. Более подробно о том, как создавать экземпляр сокета, говорится в разделе “Подключение к серверу” данной главы. Получив экземпляр сокета, мы вызываем метод `getOutputStream()`, чтобы получить ссылку на выходной поток данных сокета. Затем мы создаем экземпляр объекта `PrintWriter` для записи текста по сети на сервер, к которому мы подключились. Вторым параметром, передаваемым конструктору `PrintWriter` в данном примере кода, задает опцию автоочистки. Определение для этого параметра истинного значения приводит к тому, что вызов методов `println()`, `printf()` и `format()` автоматически приводит к очистке буфера вывода и передачи его по сети серверу. В приведенном выше фрагменте кода мы используем метод `print()`, поэтому после него мы должны вызвать метод `flush()`, чтобы очистить выходной буфер и передать данные по сети.

### Чтение двоичных данных

```
DataInputStream in =  
    new DataInputStream(socket.getInputStream());  
in.readUnsignedByte();
```

В данном фрагменте кода показано, как читать двоичные данные из сети. В приведенном примере подразумевается, что мы предварительно создали сокет и подклю-

чили его к серверу, с которого хотим считывать данные. Подробнее о создании экземпляра сокета читайте в разделе “Подключение к серверу” данной главы.

В данном примере мы вызываем метод `getInputStream()` экземпляра сокета, чтобы получить ссылку на входной поток данных сокета. Передавая входной поток данных в качестве параметра, мы создаем экземпляр класса `DataInputStream`, который можно использовать для считывания двоичных данных из сети. Мы используем метод `readUnsignedByte()` для считывания единственного беззнакового байта из сети.

Если нужно считать большой объем данных, то эффективнее поместить входной поток данных сокета в экземпляр объекта `BufferedInputStream`. В следующем примере показано, как это делается.

```
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        socket.getInputStream()));
```

Здесь, вместо того чтобы непосредственно передать входной поток данных сокета конструктору объекта `DataInputStream`, мы сначала создаем экземпляр класса `BufferedInputStream` и передаем его конструктору `DataInputStream`.

В приведенном фрагменте кода мы использовали метод `readUnsignedByte()`, но у класса `DataInputStream` есть много других методов для считывания данных в любой примитивный тип данных Java. К методам считывания двоичных данных относятся: `read()`, `readBoolean()`, `readByte()`, `readChar()`, `readDouble()`, `readFloat()`, `readInt()`, `readLong()`, `readShort()`, `readUnsignedByte()` и `readUnsignedShort()`. Подробнее об использовании этих и других методов класса `DataInputStream` читайте в документации JavaDoc по адресу <http://java.sun.com/javase/6/docs/api/java/io/DataInputStream.html>.

### Запись двоичных данных

```
DataOutputStream out =  
    new DataOutputStream(socket.getOutputStream());  
out.write(byteArray, 0, 10);
```

В одном из предыдущих разделов данной главы “Запись текста” мы выяснили, как передавать текстовые данные по сети. В данном фрагменте кода показано, как записывать двоичные данные в сеть. В приведенном примере предполагается, что мы предварительно создали сокет и подключили его к серверу, которому хотим передавать данные. Подробнее о том, как создавать экземпляр сокета, читайте в разделе “Подключение к серверу” данной главы.

В данном примере мы вызываем метод `getOutputStream()` экземпляра сокета, чтобы получить ссылку на выходной поток сокета. Затем мы создаем экземпляр класса `DataOutputStream`, который можно использовать для записи двоичных данных в сеть. Мы используем метод `write()` для записи массива байтов в сеть. Методу `write()` передается три параметра. Первый параметр — типа `byte[]`, указывает на байтовый массив, содержащий информацию, которую нужно передать по сети. Второй параметр — смещение в байтовом массиве, определяющее позицию, с которой нужно начать запись данных. Третий параметр — это количество байтов, которое нужно записать. Так, в данном фрагменте кода, мы записываем 10 байтов из массива `byteArray`, начиная со смещения 0.

Если записывается большой объем данных, то более эффективно заключить выходной поток сокета в экземпляр класса `BufferedOutputStream`. Вот как это делается.

```
DataOutputStream out = new DataOutputStream(  
    new BufferedOutputStream(  
        socket.getOutputStream()));
```

Здесь, вместо того чтобы непосредственно передавать выходной поток данных сокета конструктору класса `DataOutputStream`, мы сначала создаем экземпляр объекта `BufferedOutputStream` и передаем его конструктору класса `DataOutputStream`.

В данном примере мы использовали метод `write()`, но у класса `DataOutputStream` есть много других методов, позволяющих записывать данные из любого примитивного типа Java. К имеющимся методам записи двоичных данных относятся: `write()`, `writeBoolean()`, `writeByte()`, `writeBytes()`, `writeChar()`, `writeChars()`, `writeDouble()`, `writeFloat()`, `writeInt()`, `writeLong()` и `writeShort()`. За более подробной документацией об использовании этих и других методов класса `DataOutputStream` обратитесь к документации JavaDoc по адресу <http://java.sun.com/javase/6/docs/api/java/io/DataOutputStream.html>.

### Чтение сериализованных данных

```
ObjectInputStream in =
    new ObjectInputStream(socket.getInputStream());
Object o = in.readObject();
```

В языке Java есть возможность сериализовать экземпляры объектов и записать их либо в файл, либо в сеть. В приведенном примере показано, как считать из сетевого сокета сериализованный объект. В этом фрагменте кода предполагается, что мы предварительно создали сокет и подключили его к серверу, с которым нужно обмениваться информацией. Более подробную информацию о создании экземпляра сокета читайте в разделе “Подключение к серверу” данной главы.

В данном фрагменте кода мы вызываем метод `getInputStream()` экземпляра сокета, чтобы получить ссылку на входной поток данных сокета. Затем можно создать экземп-

ляр класса `ObjectInputStream`. Класс `ObjectInputStream` используется для десериализации примитивных данных и объектов, предварительно записанных с помощью класса `ObjectOutputStream`. Мы используем метод `readObject()` класса `ObjectInputStream` для считывания объекта из потока данных, полученного по сети. Затем можно привести объект к нужному типу. Например, если бы мы считывали объект типа `Date` из потока данных, то использовали бы для считывания следующую строку кода:

```
Date aDate = (Date)in.readObject();
```

Все поля данных объекта, не являющиеся временными и статическими, будут восстановлены со значениями, которые они имели, когда объект был сериализован.

Из потока данных можно прочитать только объекты, поддерживающие интерфейс `java.io.Serializable` или `java.io.Externalizable`. При реализации сериализуемого класса настоятельно рекомендуется объявлять свойство класса (член данных класса) `serialVersionUID`. В данное поле помещается информация о номере версии, который используется во время десериализации. Это позволяет убедиться в том, что отправитель и получатель сериализованного объекта располагают совместимыми классами, используемыми при сериализации-десериализации. Если вы явно не объявите это поле, то будет вычислено стандартное значение поля `serialVersionUID`. Стандартное значение `serialVersionUID` очень чувствительно ко всем параметрам класса. Предположим, вы незначительно изменили класс и хотели бы сохранить тот же номер версии, поскольку считаете, что определение класса по-прежнему совместимо с текущей версией. Значит, в подобных случаях лучше объявить собственный номер `serialVersionUID`.

### Запись сериализованных данных

```
ObjectOutputStream out =  
    new ObjectOutputStream(socket.getOutputStream());  
out.writeObject(myObject);
```

В Java можно сериализовать и записывать экземпляры объектов либо в файл, либо в сеть. В приведенном фрагменте кода показано, как можно записать сериализованный объект в сетевой сокет. В этом коде предполагается, что мы предварительно создали сокет и подключили его к серверу, с которым хотим обмениваться информацией. Более подробную информацию о создании экземпляра сокета читайте в разделе “Подключение к серверу” выше в данной главе.

В данном примере мы вызываем метод `getOutputStream()` экземпляра сокета, чтобы получить ссылку на выходной поток данных сокета. Затем мы создаем экземпляр класса `ObjectOutputStream`. Класс `ObjectOutputStream` используется для сериализации примитивных данных и объектов. Мы используем метод `writeObject()` класса `ObjectOutputStream` для записи объекта в поток данных.

Все поля данных, не являющиеся временными и статическими, будут сохранены в процессе сериализации и восстановлены, когда объект будет десериализован. В потоки данных можно записать только те объекты, которые поддерживают интерфейс `java.io.Serializable`.

### Чтение Web-страницы через протокол HTTP

```
URL url = new  
    URL("http://www.timothyfisher.com");  
URLConnection http = new  
    URLConnection(url);  
InputStream in = http.getInputStream();
```

В данном фрагменте кода мы выходим за рамки сетевого программирования на уровне сокетов и показываем дополнительный способ чтения данных из сети. В Java поддерживается связь по протоколу HTTP через URL с помощью класса `URLConnection`. Мы создаем экземпляр объекта URL, передавая его конструктору корректную строку URL-адреса. Затем мы создаем экземпляр класса `URLConnection`, передавая экземпляр `url` конструктору класса `URLConnection`. Мы вызываем метод `getInputStream()` для получения ссылки на входной поток данных, чтобы прочитать данные из соединения, открытого с помощью URL. С помощью входного потока данных мы затем можем прочитать содержимое Web-страницы.

Можно также прочитать содержимое URL непосредственно с помощью класса URL. Вот как это делается с помощью только класса URL:

```
URL url =
    new URL("http://www.timothyfisher.com");
url.getContent();
```

Метод `getContent()` возвращает объект типа `Object`. Возвращаемый объект может принадлежать классу `InputStream` либо содержать данные. Чаще всего используется метод `getContent()`, возвращающий объект типа `String`, в котором находится содержимое URL. Метод `getContent()`, который мы здесь использовали, на самом деле представляет сокращенную запись следующего кода:

```
url.openConnection().getContent();
```

Метод `openConnection()` класса URL возвращает объект `URLConnection`. Это объект, в котором на самом деле реализован метод `getContent()`.

В классе `URLConnection` предусмотрены специальные методы, ориентированные на протокол HTTP, ко-

торых нет в более общих классах `URL` или `URLConnection`. Например, мы используем метод `getResponseCode()`, чтобы получить код состояния из ответного сообщения протокола HTTP. HTTP также определяет протокол для перенаправления запроса другому серверу. У класса `URLConnection` есть методы, которые также “понимают” эту функцию. Например, если вы хотите сделать запрос серверу и следовать любым перенаправлениям, которые сервер вернет, можете использовать следующий код, чтобы установить эту опцию:

```
URL url =
    new URL("http://www.timothyfisher.com");
URLConnection http =
    new HttpURLConnection(url);
http.setFollowRedirects(true);
```

На самом деле для этой опции по умолчанию устанавливается истинное значение. Поэтому, возможно, более полезен сценарий, устанавливающий для опции перенаправления ложное значение, если по некоторой причине вы не хотите, чтобы вас автоматически перенаправили на сервер, отличный от того, к которому вы первоначально сделали запрос. Например, эта функция может быть полезна для безопасных приложений, которые “доверяют” только определенным серверам.

Web-страницы, содержащие секретные данные, обычно защищены безопасным протоколом под названием `Secure Sockets Layer`, или сокращенно `SSL`. При обращении к `SSL`-защищенной странице в строке `URL` используется запись `https`, а не `http`. В стандартном пакете `Java JDK` реализация `SSL` включена как составная часть `Java Secure Socket Extension (JSSE)`. Чтобы извлечь `SSL`-защищенную страницу, нужно использовать класс `HttpsURLConnection`, а не `URLConnection`. Класс `HttpsURLConnection` явно и про-

зрочно оперирует всеми деталями SSL-протокола. Более подробно об использовании SSL и других функций безопасности JSSE говорится в справочном руководстве от компании *Sun* по адресу <http://java.sun.com/javase/6/docs/technotes/guides/security/index.html>.