第A章　Javaによるネットワークプログラミング（基礎編）

C や Java 言語で TCP/IP による通信を行なうときは _____と呼ばれるものを用いる。ソケットは TCP/IP のポートに対するプログラムからのインタフェースである。

A.1　クライアントのプログラミング

例題 A.1.1 コネクション型（*TCP*）（受信のみ）

ファイル TCP_RO.java

```
import java.net.*;
import java.io.*;

public class TCP_RO {
  public static void main(String[] argv) {
    try {
      Socket readSocket = new Socket(argv[0], Integer.parseInt(argv[1]));
      InputStream instrm = readSocket.getInputStream();
      while(true) {
          int c = instrm.read();
          if (c==-1) break;
          System.out.write(c);
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

.....
.....

このプログラムは、サーバから *TCP* を用いてデータを受信し画面に出力するだけのプログラムである。

```
java TCP_RO サーバホスト名 ポート番号
```

という形で使用する。*Socket* クラスのオブジェクトを生成するときの引数は、通信先のホスト名（*IP* アドレスでも可）とポート番号である。

この *Socket* オブジェクトから _____メソッドで、*InputStream* オブジェクトを取り出すことができる。この *InputStream* に対しては、標準入力オブジェクト（*System.in*）と同じ方法で入力が可能である。

ただし、通常は

```
while(true) {
    int c = instrm.read();
    if (c==-1) break;
    System.out.write(c);
}
```

の部分は、一文字ずつ入出力を行なうことになって効率が悪いので、この部分は

```
byte[] buff = new byte[1024];
while(true) {
    int n = instrm.read(buff);
    if (n==-1) break;
    System.out.write(buff, 0, n);
}
```

のようにバッファを用いて、一度に大量の文字（この場合は 1024 文字）を読むようしている。（InputStream クラスの read() メソッド（無引数）は一文字を読む。read(byte[]) メソッドは引数として与えられた配列に文字を一度に読み込み、読み込んだ文字数を返す。）また、後で紹介するプログラムのように BufferedReader などのバッファつきの入力ストリームのクラスを使用する方法もある。

このプログラムの main メソッドでは全体を try 〜catch で囲んで、エラーが起ったときはメッセージを表示するようにしている。Exception の _____ メソッドはエラーが起った場所の情報を出力する。（実用的なプログラムでは、もっとユーザに親切なエラー処理を書くべきだが、ここでは簡略にしている。）

例題 A.1.2 コネクション型（*TCP*）（送受信）

ファイル TCP_RW.java

```
import java.net.*;
import java.io.*;

public class TCP_RW {
  public static void main(String[] argv) {
    byte[] buff = new byte[1024];
    try {
      Socket rwSocket = new Socket(argv[0], Integer.parseInt(argv[1]));
      InputStream  instrm = rwSocket.getInputStream();
      OutputStream outstr = rwSocket.getOutputStream();
      while(true) {     // 標準入力からソケットへ
        int n = System.in.read(buff);
        if (n==-1) break;
        outstr.write(buff, 0, n);
      }
      while(true) {     // ソケットから標準出力へ
        int n = instrm.read(buff);
        if (n==-1) break;
        System.out.write(buff, 0, n);
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

このプログラムはソケットに対して、まず送信を行なってから受信を行なう。送信を行なうために *Socket* クラスの _____メソッドを使って、*OutputStream* クラスのオブジェクト *outstr* を得ている。この *outstr* の *write(byte[], int, int)* メソッドを用いて、ソケットに出力する。*write(buff, i, n)* という呼出しは *buff* という配列の *i* 番目から *n* 文字を出力する。

このプログラムの使用法は次のとおりである。

```
java TCP_RW サーバホスト名 ポート番号
```

例えば *80* 番は *HTTP* のポートなので、*Web* サーバがあるマシンに対して、通信を行なうと次のようになる。

```
> java TCP_RW 133.92.XXX.XXX 80↩
GET /index.html HTTP/1.0↩
↩ （※ ← この改行の後に Ctrl-C または Ctrl-D）
HTTP1.1 200 OK
Date: Mon, XX Xxx 2XXX XX:XX:XX GMT
Server: Apache/X.X
 ...
```

立字体の部分が、ユーザが入力した部分、斜字体の部分がシステムの反応である。入力の終の印として *Ctrl-C*（*Windows* の場合）, *Ctrl-D*（*Unix* の場合）を入力すると “標準入力からソケットへ” のループを脱出して、“ソケットから標準出力へ” のループに移る。

問 A.1.3 *URL* をコマンドライン引数として受け取って、*HTTP* サーバと交信し、ページをファイルにダウンロードするプログラム *simpleGet* を書け。例えば、

```
simpleGet http://133.92.XXX.XXX/index.html
```

とすると *index.html* ファイルをダウンロードする。

問 A.1.4 *URL* をコマンドライン引数として受け取って、*HTTP* サーバと交信し、ページの中のリンク <a href=" ... ">を表示するプログラムを書け。（ヒント*:* *Java.lang.String* クラス[1]のメソッドを利用せよ。また、*2* 行にまたがる場合やコメントに含まれている場合などを完全に考慮すると難しくなるので、*100%*完全なプログラムでなくても良い。）

問 A.1.5 *simpleChmod* モード パスという形で実行すると、*FTP* サーバと交信して、ファイルを *chmod* するプログラム *simpleChmod* を書け。
実行例）

```
java simpleChmod 660 ftp://stfile/home/Report/ ...
```

[1] *(JDKDIR)*/docs/api/java/lang/String.html 参照

問 A.1.6 *Tenso* 転送元 転送先という形で実行すると、転送元のローカルファイルを *FTP* サーバ上の転送先に転送するプログラム *Tenso* を書け。（ヒント: *Java.io.File* クラス[2]のメソッドを利用せよ。）、さらにディレクトリ構造をコピーできるようにせよ。

問 A.1.7 *java.lang.Console* クラス、特に *readPassword* メソッドの使用方法を調べ、上の問の *simpleChmod* や *Tenso* で、パスワードを安全に（画面エコー無しで）入力できるようにせよ。

A.2　コネクションレス型のプログラミング

これまで紹介したソケットは＿＿＿を使った＿＿＿＿＿＿＿＿＿と呼ばれるものである。コネクション型では、最初にソケット間の接続を行ない、通信されるデータの順序が保存されるようになっている。プログラマはあたかも回線を独占しているかのようにプログラムを作成することができる。

一方、＿＿＿を用いる、最初に接続を行なわない＿＿＿＿＿＿＿＿＿＿＿＿のソケットもある。これは、送信のたびに宛先を指定する。コネクションレス型のソケットでは、データの順序は保存されないし、データが失われる場合もある。ただし管理のための処理が少なくて済むので高速である。

例題 A.2.1 コネクションレス型（*UDP*）（クライアント）

ファイル UdpClient.java

```
import java.net.*;
import java.io.*;

public class UdpClient {
  public static void main(String[] argv) {
    try {
      // 接続先の IP アドレスとポート番号
      InetAddress addr = InetAddress.getByName(argv[0]);
      int port = Integer.parseInt(argv[1]);

      // 適当な空いているポート番号にソケットを作る
      DatagramSocket dgSock = new DatagramSocket();
      while (true) {
        byte buff1[]= new byte[512];
        int n = System.in.read(buff1);
        // 送信パケットの作成
        DatagramPacket pa1 = new DatagramPacket(buff1, n, addr, port);
        dgSock.send(pa1);  // パケット送出
        System.out.println("Sent!");

        // 受信パケット用データ領域の作成
        byte buff2[] = new byte[512];
        DatagramPacket pa2 = new DatagramPacket(buff2, buff2.length);
        dgSock.receive(pa2); // パケット受信
        System.out.printf("received: %s", new String(pa2.getData()));
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

[2] *(JDKDIR)*/docs/api/java/io/File.html 参照

使用方法は、

```
java UdpClient サーバホスト名　ポート番号
```

である。

引数なしで *DatagramSocket* クラスのコンストラクタを呼び出すと、適当な空きポートに *UDP* ソケットを作る。また、パケットは *DatagramPacket* というクラスのオブジェクトとして表現される

```
DatagramPacket(byte[] data, int len, InetAddress addr, int port)
```

という形のコンストラクタは、長さ *len* のデータで、宛先の *IP* アドレス *addr*、ポート番号が *port* というパケットのためのデータを用意する。実際にパケットを送るのは *DatagramSocket* クラスの *send(DatagramPacket)* メソッドである。

```
DatagramPacket(byte[] data, int len)
```

という形の 2 引数のコンストラクタは受信したパケットのデータを受け取るためのオブジェクトを用意する。実際にパケットを受信するのは *DatagramSocket* クラスの *receive* メソッドである。*receive* メソッドに、*DatagramPacket* クラスのオブジェクトを引数として与える。*receive* の呼出し後には、このオブジェクトの内容が受信したデータに書き換えられている。

例題 A.2.2 コネクションレス型（*UDP*）（サーバ）

ファイル UdpServer.java

```
import java.net.*;
import java.io.*;

public class UdpServer {
  public static void main(String[] argv) {
    try {
      int port = Integer.parseInt(argv[0]);  // 使用するポート番号
      // 指定されたポート番号にソケットを作る
      DatagramSocket dgSock = new DatagramSocket(port);
      while (true) {          // 受信パケット用データ領域の作成
        byte buff1[]= new byte[512];
        DatagramPacket pa1 = new DatagramPacket(buff1, buff1.length);
        dgSock.receive(pa1);  // パケット受信
        System.out.println("Received!");
        System.out.print(new String(pa1.getData()));
        System.out.println("addr: %s%n port: %s%n",
                           pa1.getAddress(), pa1.getPort());

        // 送信パケットの作成
        DatagramPacket pa2 =
          new DatagramPacket(pa1.getData(), pa1.getLength(),
                             pa1.getAddress(), pa1.getPort());
        dgSock.send(pa2);
        System.out.println("Sent!");
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

サーバ側では、クライアントと違って固定したポート番号にソケットを作成する。そのためポート番号（*int* 型）を *DatagramSocket* のコンストラクタの引数として用いる。

このプログラムは、まず先にクライアントから送られてきたパケットを受信している。受信したパケット（*DatagramPacket*）から *getData* メソッドでデータの部分を取り出すことができる。また *getAddress*, *getPort* メソッドで送り元の *IP* アドレス、ポート番号を知ることができる。

このプログラムでは送られてきたデータを、そのまま何も変更せずにクライアントの送り元のポートに送り返している。

問 A.2.3（チャット）
チャットサーバとクライアントを作成せよ

問 A.2.4（電子ホワイトボード）
電子ホワイトボードサーバとクライアントを作成せよ

問 A.2.5（タートルグラフィックスサーバ）
タートルグラフィックスとは、画面上の仮想の亀に指令を与えて、線を描画することである。例えば、次のような指令は一辺が *100* の正三角形を描く。

```
FORWARD 100
RIGHT 120
FORWARD 100
RIGHT 120
FORWARD 100
RIGHT 120
```

FORWARD は前進する命令、*RIGHT* は右に回転する命令である。この “亀” をサーバとして実現して、複数のクライアントから指令を与えることができるようにせよ。サーバとクライアントの間はコネクションレス型で通信を行なうこと。クライアントはサーバから情報を得て、“亀” の軌跡を表示できるようにせよ。（タートルグラフィックスの命令は自由に拡張しても良い。）

キーワード　ソケット、Socket クラス、getInputStream メソッド、getOutputStream メソッド、DatagramSocket クラス、DatagramPacket クラス、send メソッド、receive メソッド、getData メソッド、getAddress メソッド、getPort メソッド