Java URL处理 Java 发送邮件

Java 网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

java.net 包中 J2SE 的 API 包含有类和接口，它们提供低层次的通信细节。你可以直接使用这些类和接口，来专注于解决问题，而不用关注通信细节。

java.net 包中提供了两种常见的网络协议的支持：

- **TCP**：TCP 是传输控制协议的缩写，它保障了两个应用程序之间的可靠通信。通常用于互联网协议，被称 TCP / IP。
- **UDP**：UDP 是用户数据报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

本教程主要讲解以下两个主题。

- **Socket 编程**：这是使用最广泛的网络概念，它已被解释地非常详细。
- **URL 处理**：这部分会在另外的篇幅里讲，点击这里更详细地了解在 Java 语言中的 URL 处理。

Socket 编程

套接字使用TCP提供了两台计算机之间的通信机制。 客户端程序创建一个套接字，并尝试连接服务器的套接字。

当连接建立时，服务器会创建一个 Socket 对象。客户端和服务器现在可以通过对 Socket 对象的写入和读取来进行通信。

java.net.Socket 类代表一个套接字，并且 java.net.ServerSocket 类为服务器程序提供了一种来监听客户端，并与他们建立连接的机制。

以下步骤在两台计算机之间使用套接字建立TCP连接时会出现：

- 服务器实例化一个 ServerSocket 对象，表示通过服务器上的端口通信。
- 服务器调用 ServerSocket 类的 accept() 方法，该方法将一直等待，直到客户端连接到服务器上给定的端口。
- 服务器正在等待时，一个客户端实例化一个 Socket 对象，指定服务器名称和端口号来请求连接。
- Socket 类的构造函数试图将客户端连接到指定的服务器和端口号。如果通信被建立，则在客户端创建一个 Socket 对象能够与服务器进行通信。
- 在服务器端，accept() 方法返回服务器上一个新的 socket 引用，该 socket 连接到客户端的 socket。

连接建立后，通过使用 I/O 流在进行通信，每一个socket都有一个输出流和一个输入流，客户端的输出流连接到服务器端的输入流，而客户端的输入流连接到服务器端的输出流。

TCP 是一个双向的通信协议，因此数据可以通过两个数据流在同一时间发送.以下是一些类提供的一套完整的有用的方法来实现 socket。

ServerSocket 类的方法

服务器应用程序通过使用 java.net.ServerSocket 类以获取一个端口,并且侦听客户端请求。

ServerSocket 类有四个构造方法：

序号	方法描述
1	**public ServerSocket(int port) throws IOException** 创建绑定到特定端口的服务器套接字。
2	**public ServerSocket(int port, int backlog) throws IOException** 利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** 使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。
4	**public ServerSocket() throws IOException** 创建非绑定服务器套接字。

创建非绑定服务器套接字。 如果 ServerSocket 构造方法没有抛出异常，就意味着你的应用程序已经成功绑定到指定的端口，并且侦听客户端请求。

这里有一些 ServerSocket 类的常用方法：

序号	方法描述
1	**public int getLocalPort()** 返回此套接字在其上侦听的端口。
2	**public Socket accept() throws IOException** 侦听并接受到此套接字的连接。
3	**public void setSoTimeout(int timeout)** 通过指定超时值启用/禁用 SO_TIMEOUT，以毫秒为单位。
4	**public void bind(SocketAddress host, int backlog)** 将 ServerSocket 绑定到特定地址（IP 地址和端口号）。

Socket 类的方法

java.net.Socket 类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个 Socket 对象通过实例化 ，而 服务器获得一个 Socket 对象则通过 accept() 方法的返回值。

Socket 类有五个构造方法.

序号	方法描述
1	**public Socket(String host, int port) throws UnknownHostException, IOException.**

	创建一个流套接字并将其连接到指定主机上的指定端口号。
2	**public Socket(InetAddress host, int port) throws IOException** 创建一个流套接字并将其连接到指定 IP 地址的指定端口号。
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** 创建一个套接字并将其连接到指定远程主机上的指定远程端口。
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** 创建一个套接字并将其连接到指定远程地址上的指定远程端口。
5	**public Socket()** 通过系统默认类型的 SocketImpl 创建未连接套接字

当 Socket 构造方法返回，并没有简单的实例化了一个 Socket 对象，它实际上会尝试连接到指定的服务器和端口。

下面列出了一些感兴趣的方法，注意客户端和服务器端都有一个 Socket 对象，所以无论客户端还是服务端都能够调用这些方法。

序号	方法描述
1	**public void connect(SocketAddress host, int timeout) throws IOException** 将此套接字连接到服务器，并指定一个超时值。
2	**public InetAddress getInetAddress()** 返回套接字连接的地址。
3	**public int getPort()** 返回此套接字连接到的远程端口。
4	**public int getLocalPort()** 返回此套接字绑定到的本地端口。
5	**public SocketAddress getRemoteSocketAddress()** 返回此套接字连接的端点的地址，如果未连接则返回 null。
6	**public InputStream getInputStream() throws IOException** 返回此套接字的输入流。
7	**public OutputStream getOutputStream() throws IOException** 返回此套接字的输出流。
8	**public void close() throws IOException** 关闭此套接字。

InetAddress 类的方法

这个类表示互联网协议(IP)地址。下面列出了 Socket 编程时比较有用的方法：

序号	方法描述
1	**static InetAddress getByAddress(byte[] addr)** 在给定原始 IP 地址的情况下，返回 InetAddress 对象。
2	**static InetAddress getByAddress(String host, byte[] addr)** 根据提供的主机名和 IP 地址创建 InetAddress。
3	**static InetAddress getByName(String host)** 在给定主机名的情况下确定主机的 IP 地址。
4	**String getHostAddress()** 返回 IP 地址字符串（以文本表现形式）。
5	**String getHostName()** 获取此 IP 地址的主机名。
6	**static InetAddress getLocalHost()** 返回本地主机。
7	**String toString()** 将此 IP 地址转换为 String。

Socket 客户端实例

如下的 GreetingClient 是一个客户端程序，该程序通过 socket 连接到服务器并发送一个请求，然后等待一个响应。

GreetingClient.java 文件代码：

```
// 文件名 GreetingClient.java
import java.net.*;
import java.io.*;
public class GreetingClient
{
public static void main(String [] args)
{
String serverName = args[0];
int port = Integer.parseInt(args[1]);
try
{
System.out.println("连接到主机: " + serverName + " ，端口号: " + port);
Socket client = new Socket(serverName, port);
System.out.println("远程主机地址: " + client.getRemoteSocketAddress());
OutputStream outToServer = client.getOutputStream();
```

```
DataOutputStream out = new DataOutputStream(outToServer);
out.writeUTF("Hello from " + client.getLocalSocketAddress());
InputStream inFromServer = client.getInputStream();
DataInputStream in = new DataInputStream(inFromServer);
System.out.println("服务器响应： " + in.readUTF());
client.close();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

Socket 服务端实例

如下的GreetingServer 程序是一个服务器端应用程序，使用 Socket 来监听一个指定的端口。

GreetingServer.java 文件代码：

```
// 文件名 GreetingServer.java
import java.net.*;
import java.io.*;
public class GreetingServer extends Thread
{
private ServerSocket serverSocket;
public GreetingServer(int port) throws IOException
{
serverSocket = new ServerSocket(port);
serverSocket.setSoTimeout(10000);
}
public void run()
{
while(true)
{
try
{
System.out.println("等待远程连接，端口号为：" + serverSocket.getLocalPort() + "...");
Socket server = serverSocket.accept();
System.out.println("远程主机地址：" + server.getRemoteSocketAddress());
DataInputStream in = new DataInputStream(server.getInputStream());
System.out.println(in.readUTF());
DataOutputStream out = new DataOutputStream(server.getOutputStream());
out.writeUTF("谢谢连接我：" + server.getLocalSocketAddress() + "\nGoodbye!");
server.close();
}catch(SocketTimeoutException s)
{
System.out.println("Socket timed out!");
break;
}catch(IOException e)
{
e.printStackTrace();
break;
}
}
```

```
}
public static void main(String [] args)
{
int port = Integer.parseInt(args[0]);
try
{
Thread t = new GreetingServer(port);
t.run();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

编译以上两个 java 文件代码，并执行以下命令来启动服务，使用端口号为 6066：

```
$ javac GreetingServer.java
$ java GreetingServer 6066
等待远程连接，端口号为：6066...
```

新开一个命令窗口，执行以上命令来开启客户端：

```
$ javac GreetingClient.java
$ java GreetingClient localhost 6066
连接到主机：localhost ，端口号：6066
远程主机地址：localhost/127.0.0.1:6066
服务器响应： 谢谢连接我：/127.0.0.1:6066
Goodbye!
```

← Java URL处理 Java 发送邮件 →

2 篇笔记

写笔记

Socket的概念：上面已经解释了，不在复述。

同步和异步：同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO 操作并等待或者轮询的去查看IO 操作是否就绪，而异步是指用户进程触发IO 操作以后便开始做自己的事情，而当IO 操作已经完成的时候会得到IO 完成的通知。

以银行取款为例：

同步 ： 自己亲自出马持银行卡到银行取钱（使用同步 IO 时，Java 自己处理IO 读写）；

异步 ： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO 时，Java 将 IO 读写委托给OS 处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS 需要支持异步IO操作API）；

阻塞和非阻塞：阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作方法的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入方法会立即返回一个状态值。

以银行取款为例：

阻塞 ： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；

非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器通知可读写时再继续进行读写，不断循环直到读写完成）

1.BIO 编程

Blocking IO： 同步阻塞的编程方式。

BIO编程方式通常是在JDK1.4版本之前常用的编程方式。编程实现过程为：首先在服务端启动一个ServerSocket来监听网络请求，客户端启动Socket发起网络请求，默认情况下ServerSocket回建立一个线程来处理此请求，如果服务端没有线程可用，客户端则会阻塞等待或遭到拒绝。

且建立好的连接，在通讯过程中，是同步的。在并发处理效率上比较低。大致结构如下：

同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

使用线程池机制改善后的BIO模型图如下:

2.NIO 编程：Unblocking IO（New IO）： 同步非阻塞的编程方式。

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题，NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题

3.AIO编程：Asynchronous IO： 异步非阻塞的编程方式。

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：
AsynchronousSocketChannel、AsynchronousServerSocketChannel、AsynchronousFileChannel、AsynchronousDatagramChannel

bio示例

server示例：

```
public class Server {

    public static void main(String[] args) {
        int port = genPort(args);

        ServerSocket server = null;
        ExecutorService service = Executors.newFixedThreadPool(50);

        try{
            server = new ServerSocket(port);
            System.out.println("server started!");
            while(true){
                Socket socket = server.accept();

                service.execute(new Handler(socket));
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(server != null){
                try {
                    server.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            server = null;
        }
    }

    static class Handler implements Runnable{
        Socket socket = null;
        public Handler(Socket socket){
            this.socket = socket;
        }
        @Override
        public void run() {
            BufferedReader reader = null;
            PrintWriter writer = null;
            try{

                reader = new BufferedReader(
                        new InputStreamReader(socket.getInputStream(), "UTF-8"));
                writer = new PrintWriter(
                        new OutputStreamWriter(socket.getOutputStream(), "UTF-8"));
                String readMessage = null;
                while(true){
                    System.out.println("server reading... ");
```

```
                    if((readMessage = reader.readLine()) == null){
                        break;
                    }
                    System.out.println(readMessage);
                    writer.println("server recive : " + readMessage);
                    writer.flush();
                }
            }catch(Exception e){
                e.printStackTrace();
            }finally{
                if(socket != null){
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                socket = null;
                if(reader != null){
                    try {
                        reader.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                reader = null;
                if(writer != null){
                    writer.close();
                }
                writer = null;
            }
        }

    }

    private static int genPort(String[] args){
        if(args.length > 0){
            try{
                return Integer.parseInt(args[0]);
            }catch(NumberFormatException e){
                return 9999;
            }
        }else{
            return 9999;
        }
    }

}
```

2.client示例：

```
public class Client {
    public static void main(String[] args) {
        String host = null;
        int port = 0;
        if(args.length > 2){
            host = args[0];
            port = Integer.parseInt(args[1]);
        }else{
            host = "127.0.0.1";
            port = 9999;
        }

        Socket socket = null;
        BufferedReader reader = null;
        PrintWriter writer = null;
        Scanner s = new Scanner(System.in);
        try{
            socket = new Socket(host, port);
            String message = null;

            reader = new BufferedReader(
                    new InputStreamReader(socket.getInputStream(), "UTF-8"));
            writer = new PrintWriter(
                    socket.getOutputStream(), true);
            while(true){
                message = s.nextLine();
                if(message.equals("exit")){
                    break;
                }
                writer.println(message);
                writer.flush();
                System.out.println(reader.readLine());
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            socket = null;
            if(reader != null){
                try {
                    reader.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            reader = null;
            if(writer != null){
                writer.close();
            }
            writer = null;
        }
    }
}
```

以上只是简单示例，仅供参考！

JeoSaber 7个月前 (08-07)

DatagramSocket(UDP)简单示例

服务端：

```
public class Server {
    public static void main(String[] args) {
        try {
            DatagramSocket server = new DatagramSocket(5060);
            DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
            server.receive(packet);
            System.out.println(packet.getAddress().getHostName() + "(" + packet.getPort()
 + "):" + new String(packet.getData()));
            packet.setData("Hello Client".getBytes());
            packet.setPort(5070);
            packet.setAddress(InetAddress.getLocalHost());
            server.send(packet);
            server.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

客户端:

```
public class Client {
    public static void main(String[] args){
        try {
            DatagramSocket client = new DatagramSocket(5070);
            DatagramPacket packet = new DatagramPacket(new byte[1024],1024);
            packet.setPort(5060);
            packet.setAddress(InetAddress.getLocalHost());
            packet.setData("Hello Server".getBytes());
            client.send(packet);
```

```
            client.receive(packet);
            System.out.println(packet.getAddress().getHostName() + "(" + packet.getPort()
 + "):" + new String(packet.getData()));
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

一生默默守护你 7个月前 (08-30)