

002.Java的基本网络支持

Java为网络编程提供了支持包`java.net`, 该包下提供了`InetAddress`、`URLDecoder`、`URLEncoder`、`URL`和`URLConnection`等等很多工具类, 这些类可以让我们通过编程的形式访问Web服务器或者其他的服务器。

InetAddress类及其常用方法

Internet 上的主机有两种方式表示地址, 分别为域名和 IP 地址。

`java.net` 包中的 `InetAddress` 类对象包含一个 Internet 主机地址的域名和 IP 地址。

`InetAddress` 类提供了操作 IP 地址的各种方法。

该类本身没有构造方法, 而是通过调用相关静态方法, 比如`getByName()`方法获取实例。

`InetAddress` 类中的常用方法如下表 所示。

方法名称	说明
<code>byte[] getAddress()</code>	返回此 <code>InetAddress</code> 对象的原始 IP 地址
<code>String getCanonicalHostName()</code>	获取此 IP 地址的完全限定域名
<code>String.getHostAddress()</code>	返回 IP 地址字符串 (以文本表现形式)
<code>String getHostName()</code>	返回此 IP 地址的主机名
<code>static InetAddress getLocalHost()</code>	返回本地主机

编写程序练习 InetAddress 类的基本使用方法:

(1) 创建一个类。

在 `main()` 方法中创建一个 `InetAddress` 对象, 调用 `getByName()` 方法并传递参数 “`www.qq.com`”

输出此对象的 IP 地址字符串和主机名, 代码如下所示。

```
public static void main(String[] args)
{
    try
    {
        InetAddress ial=InetAddress.getByName("www.qq.com");
        System.out.println(ial.getHostName());
        System.out.println(ial.getHostAddress());
    }
    catch (UnknownHostException e)
    {
        e.printStackTrace();
    }
}
```

(2) 在 `main()` 方法中添加代码, 创建一个 `InetAddress` 对象, 调用 `getByName()` 方法并传递参数 “`61.135.169.105`” 输出此对象的 IP 地址字符串和主机名, 代码如下所示:

```
try
{
    InetAddress ia2=InetAddress.getByName("61.135.169.105");
    System.out.println(ia2.getHostName());
    System.out.println(ia2.getHostAddress());
}
catch(UnknownHostException e)
{
    e.printStackTrace();
}
```

(3) 创建一个 InetAddress 对象用于获取本地主机的信息，输出此对象的 IP 地址字符串和主机名，代码如下所示：

```
try
{
    InetAddress ia3=InetAddress.getLocalHost();
    System.out.println("主机名: "+ia3.getHostName());
    System.out.println("本地ip地址: "+ia3.getHostAddress());
}
catch(UnknownHostException e)
{
    e.printStackTrace();
}
```

注意：在上述代码中包含互联网的地址，所以运行时需要连网，否则会出现异常。

URLDecoder和URLEncoder

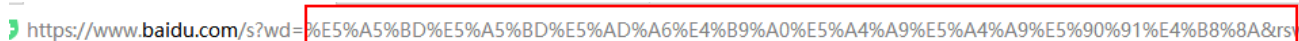
URLDecoder和URLEncoder的作用主要是实现：

普通字符串和application/x-www-form-urlencoded字符串相互转换的功能。

application/x-www-form-urlencoded： 是一种应用在网页提交数据的过程中，被提交的数据的编码格式。

这个编码，我们并不陌生，比如在百度上搜索一下，

网址中出现的“乱码”就是application/x-www-form-urlencoded格式的字符。

 <https://www.baidu.com/s?wd=%E5%A5%BD%E5%A5%BD%E5%AD%A6%E4%B9%A0%E5%A4%A9%E5%A4%A9%E5%90%91%E4%B8%8A&rsy>

它的基本格式：%XXXXX XX为十六进制的数字。

为什么会有这种转换呢？

网页的 URL **规定**只能包含合法的字符，这可以分成两类。

URL 元字符：分号 (;)，逗号 (','), 斜杠 (/)，问号 (?)，冒号 (:)，at (@)，&，等号 (=)，

加号 (+)，美元符号 (\$)，井号 (#)

语义字符：a-z，A-Z，0-9，连词号 (-)，下划线 (_)，点 (.)，感叹号 (!)，波浪线 (~)，

星号 (*)，单引号 (')，圆括号 (())

除了以上字符，其他字符出现在 URL 之中都必须转义，规则是根据操作系统的默认编码，将每个字节转为百分号（%）加上两个大写的十六进制字母。比如，UTF-8 的操作系统上，

`http://www.example.com/q=春节`这个 URL 之中，汉字“春节”不是 URL 的合法字符，所以被浏览器自动转成`http://www.example.com/q=%E6%98%A5%E8%8A%82`。

其中，“春”转成了`%E6%98%A5`，“节”转成了“`%E8%8A%82`”。

这是因为“春”和“节”的 UTF-8 编码分别是`E6 98 A5`和`E8 8A 82`，

将每个字节前面加上百分号，就构成了 `application/x-www-form-urlencoded`格式字符串。

```
try {  
    // 将application/x-www-form-urlencoded字符串转换成普通字符串  
    String keyWord = URLDecoder.decode("%C4%E3%BA%C3", "GBK");  
    System.out.println(keyWord); //输出你好  
    // 将普通字符转换成application/x-www-form-urlencoded字符串  
    String urlString = URLEncoder.encode("你好", "GBK"); //输出%C4%E3%BA%C3  
  
    System.out.println(urlString);  
} catch (UnsupportedEncodingException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}  
}
```

URL类和URLConnection类

URL类

在 `java.net` 包中包含专门用来处理 URL 的 URL类，可以获得 URL 的相关信息，例如 URL 的协议名和主机名等。下面分别对它的构造方法和常用方法进行介绍。

构造方法	说明
<code>public URL (String spec)</code>	通过一个表示 URL 地址的字符串可以构造一个 URL 对象。
<code>public URL(URL context,String spec)</code>	使用基本地址和相对 URL 构造一个 URL 对象。
<code>public URL(String protocol,String host,String file)</code>	使用指定的协议、主机名和文件名创建一个 URL 对象。
<code>public URL(String protocol,String host,int port,String file)</code>	使用指定的协议、主机名、端口号和文件名创建一个 URL 对象。

URL 的常用方法：

方法	说明
public String getProtocol()	获取该 URL 的协议名。
public String getHost()	获取该 URL 的主机名。
public int getPort()	获取该 URL 的端口号，如果没有设置端口，返回 -1。
public String getFile()	获取该 URL 的文件名。
public String getRef()	获取该 URL 在文件中的相对位置。
public String getQuery()	获取该 URL 的查询信息。
public String getPath()	获取该 URL 的路径。
public String getAuthority()	获取该 URL 的权限信息。
public String getUserInfo()	获得使用者的信息。
public String getRef()	获得该 URL 的锚点。

URLConnection 类

完成了 URL 的定义，接下来就可以获得 URL 的通信连接。
在 java.net 包中，定义了专门的 URLConnection 类来表示与 URL 建立的通信连接，URLConnection 类的对象使用 URL 类的 openConnection() 方法获得。
注意：我们在应用程序和URL链接的时候，使用URLConnection类，
当我们HTTP和URL链接的时候，使用URLConnection类的子类HttpURLConnection。

URLConnection 类的主要方法：

方法	说明
void addRequestProperty(String key,String value)	添加由键值对指定的一般请求属性。key 指的是用于识别请求的关键字（例如 accept），value 指的是与该键关联的值。
void connect()	打开到此 URL 所引用的资源的通信链接（如果尚未建立这样的链接）。
Object getConnection()	检索此 URL 链接的内容。
InputStream getInputStream()	返回从此打开的链接读取的输入流。
OutputStream getOutputStream()	返回写入到此链接的输出流。
URL getURL()	返回此 URLConnection 的 URL 字段的值。

代码案例：使用 URL 和 URLConnection 类获取与百度首页的链接并将其页面信息输出到控制台

(1) 创建一个类，编写 main() 方法，在该方法中创建一个 URL 对象，
然后传入参数“http://www.baidu.com/”，输出 URL 的相关信息，代码如下所示。

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;
public class URLEmo
{
    public static void main(String[] args)
    {
        try
        {
            URL url=new URL("http://www.baidu.com/");
```

```

        System.out.println("协议: " + url.getProtocol());
        System.out.println("主机: " + url.getHost());
        System.out.println("端口: " + url.getPort());
        InputStream in;
    }
    catch(IOException e)
    {
        //TODO 自动生成的 catch 块
        e.printStackTrace();
    }
}
}

```

(2) 在 main() 方法的 try 模块中继续添加代码, 获得 URLConnection 对象, 通过输入流读取页面源代码并将信息输出到控制台, 代码如下所示。

```

URLConnection uc=url.openConnection();
in=uc.getInputStream();
byte[] b=new byte[1024];
int len;
while((len=in.read(b))!=-1)
{
    System.out.println(new String(b,0,len));
}
in.close();

```

练习一个复杂一点的案例, 多线程下载文件的程序, 把前面的知识汇总应用一下:

```

public class DownUtil {
    //定义下载路径
    private String path;
    //制定锁下载文件的保存位置
    private String targetFile;
    //定义使用多少线程去下载资源
    private int threadNum;
    //定义下载的线程对象
    private DownThread[] threads;
    //定义下载文件的总大小
    private int fileSize;

    //构造函数初始化上面的属性
    public DownUtil(String path,String targetFile,int threadNum){
        this.path = path;
        this.targetFile = targetFile;
        this.threadNum = threadNum;
        //初始化线程对象数组
        threads = new DownThread[threadNum];
    }

    //实现下载的方法
    public void download() throws Exception {
        URL url = new URL(path);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setConnectTimeout(5000);
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "/*/*");
        conn.setRequestProperty("Accept-Language", "zh-CN");
        conn.setRequestProperty("Charset", "UTF-8");
        conn.setRequestProperty("Connection", "Keep-Alive");
    }
}

```

```

//得到文件大小
fileSize = conn.getContentLength();
conn.disconnect();

int currentPartSize = fileSize/threadNum + 1;

RandomAccessFile file = new RandomAccessFile(targetFile,"rw");
//设置本地文件大小
file.setLength(fileSize);
file.close();

for(int i=0;i<threadNum;i++){
    //计算每个线程下载的开始位置
    int startPos = i * currentPartSize;
    //每个线程使用一个Raf进行下载
    RandomAccessFile currentPart = new RandomAccessFile(targetFile,"rw");
    //定位该线程的下载位置
    currentPart.seek(startPos);
    //创建下载线程
    threads[i] = new DownThread(startPos, currentPartSize, currentPart);
    //启动下载线程
    threads[i].start();
}
}

//获取下载完成的百分百
public double getCompleteRate(){
    //统计多个线程已经下载的总大小
    int sumSize = 0;
    for(int i=0;i<threadNum;i++){
        sumSize += threads[i].length;
    }
    //返回已下载的百分比
    return sumSize * 1.0 / fileSize;
}

//下载线程对象的内部类
private class DownThread extends Thread{
    //当前线程下载的位置
    private int startPos;
    //当前线程负责下载的大小
    private int currentPartSize;
    //当前线程需要下载的文件块
    private RandomAccessFile currentPart;
    //定义当前线程已经下载的字节数
    public int length;

    //构造函数初始化
    public DownThread(int startPos,int currentPartSize,RandomAccessFile currentPart){
        this.startPos = startPos;
        this.currentPartSize = currentPartSize;
        this.currentPart = currentPart;
    }

    @Override
    public void run() {
        try {
            URL url = new URL(path);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setConnectTimeout(5000);
            conn.setRequestMethod("GET");

```

```

        conn.setRequestProperty("Accept", "*/*");
        conn.setRequestProperty("Accept-Language", "zh-CN");
        conn.setRequestProperty("Charset", "UTF-8");

        //建立实际的链接, 可以省略, 下面的getInputStream()方法会判断, 没链接, 自动调用。
        conn.connect();

        InputStream inStream = conn.getInputStream();
        //跳过startPos, 表示该线程只下载自己下载的那部分
        inStream.skip(this.startPos);
        byte[] buffer = new byte[1024];
        int hasRead = 0;
        while(length < currentPartSize && (hasRead=inStream.read(buffer))!=-1){
            currentPart.write(buffer, 0, hasRead);
            //累计线程下载的大小
            length += hasRead;
        }
        currentPart.close();
        inStream.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

执行测试类:

```

public class NetTest1 {
    public static void main(String[] args) throws Exception {
        final DownUtil downUtil = new DownUtil("https://timgsa.baidu.com/timg" +
            "?image&quality=80&size=b9999_10000&sec=" +
            "1575352177018&di=3ea9282c6a91f58b9ad1d4f035854a62" +
            "&imgtype=0&src=http%3A%2F%2Fimg07.mifile." +
            "cn%2Fv1%2FMI_55950AFBBEDCB%2FT1madQBXZT1RXrhCrK.jpg", "e:\\test.jpg", 5);

        downUtil.download();

        new Thread(new Runnable() {
            @Override
            public void run() {
                while (downUtil.getCompleteRate() < 1){
                    System.out.println("已完成: " + downUtil.getCompleteRate());
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
    }
}

```

如果在上面的实现基础上, 要实现断点下载, 只需要额外添加一个配置文件, 配置文件中记录一下每个线程下载到那个字节, 断网后再次下载的时候, 只需要继续从上次下载的位置继续下载就可以了。

再来一个案例：

前面案例是发送GET请求，然后这里我们再来实现一个POST请求：

```
public class SendPostDemo {

    public static void main(String[] args) throws Exception{
        String urlPath = new String("http://localhost:8080/Test1/HelloWorld");
        //String urlPath = new String("http://localhost:8080/Test1/HelloWorld?name=丁丁".getBytes("UTF-8"));

        String param="name="+URLEncoder.encode("丁丁", "UTF-8");

        //建立连接
        URL url=new URL(urlPath);
        HttpURLConnection httpConn=(HttpURLConnection)url.openConnection();

        //设置参数
        httpConn.setDoOutput(true);    //需要输出
        httpConn.setDoInput(true);    //需要输入
        httpConn.setUseCaches(false);  //不允许缓存
        httpConn.setRequestMethod("POST");    //设置POST方式连接

        //设置请求属性
        httpConn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
        httpConn.setRequestProperty("Connection", "Keep-Alive");// 维持长连接
        httpConn.setRequestProperty("Charset", "UTF-8");

        //连接,也可以不用明文connect, 使用下面的httpConn.getOutputStream()会自动connect
        httpConn.connect();

        //建立输入流, 向指向的URL传入参数
        DataOutputStream dos=new DataOutputStream(httpConn.getOutputStream());
        dos.writeBytes(param);
        dos.flush();
        dos.close();

        //获得响应状态
        int resultCode=httpConn.getResponseCode();
        if(HttpURLConnection.HTTP_OK==resultCode){
            StringBuffer sb=new StringBuffer();
            String readLine=new String();
            BufferedReader responseReader=new BufferedReader(new
InputStreamReader(httpConn.getInputStream(), "UTF-8"));
            while((readLine=responseReader.readLine())!=null){
                sb.append(readLine).append("\n");
            }
            responseReader.close();
            System.out.println(sb.toString());
        }
    }
}
```

**JAVA使用HttpURLConnection发送POST数据是依靠OutputStream流的形式发送
具体编码过程中，参数是以字符串“name=XXX”这种形式发送**

总结：

a: HttpURLConnection的connect()函数，实际上只是建立了一个与服务器的tcp连接，并没有实际发送http请求。

无论是post还是get，http请求实际上直到URLConnection的getInputStream() 这个函数里面才正式发送出去。

b:在用POST方式发送URL请求时，URL请求参数的设定顺序是重中之重，

对connection对象的一切配置（那一堆set函数）

都必须要在connect() 函数执行之前完成。

而对outputStream的写操作，又必须要在inputStream的读操作之前。

这些顺序实际上是由http请求的格式决定的。

如果inputStream读操作在outputStream的写操作之前，会抛出例外：

```
java.net.ProtocolException: Cannot write output after reading
input.....
```

c:http请求实际上由两部分组成，

一个是http头，所有关于此次http请求的配置都在http头里面定义，

一个是正文content。

connect() 函数会根据URLConnection对象的配置值生成http头部信息，

因此在调用connect函数之前，就必须把所有的配置准备好。

d: 在http头后面紧跟着的是http请求的正文，正文的内容是通过outputStream流写入的，实际上outputStream不是一个网络流，充其量是个字符串流，往里面写入的东西不会立即发送到网络，

而是存在于内存缓冲区中，待outputStream流关闭时(关闭的时候调用了flush方法)，根据输入的内容生成http正文。

至此，http请求的东西已经全部准备就绪。

在getInputStream() 函数调用的时候，就会把准备好的http请求正式发送到服务器了，

然后返回一个输入流，用于读取服务器对于此次http请求的返回信息。

由于http 请求在getInputStream的时候已经发送出去了（包括http头和正文），因此在getInputStream() 函数

之后对connection对象进行设置（对http头的信息进行修改）或者写入outputStream（对正文进行修改）

都是没有意义的了，执行这些操作会导致异常的发生。

