'**multipart/form-data**'에 해당되는 글 **1**건

1. 2010/07/30 [[Java] multipart/form-data 타입을 이용한 파일 전송지원 클래스 작성](http://sdw8001.tistory.com/25)

**http://ts.daumcdn.net/custom/blog/60/600708/skin/images/month07.gif**

**30**

[**[Java] multipart/form-data 타입을 이용한 파일 전송지원 클래스 작성**](http://sdw8001.tistory.com/25)[Programming/Java](http://sdw8001.tistory.com/category/Programming/Java)2010/07/30 16:25

파일 전송을 위해 사용되는 multipart/form-data MIME을 사용하여 HTTP 기반의 파일 전송 방법에 대해서 살펴보자.  
  
**multipart/form-data를 지원하는 HttpRequestor의 구현**  
  
예전에 *자바로 구현하는 Web-to-web 프로그래밍, Part 1*에서 필자는 HTTP 프로토콜의 GET/POST 방식을 사용하여 웹 기반의 데이터 송수신 방법에 대해서 살펴보았다. 이때 작성했던 클래스인 HttpMessage는 application/x-www-form-urlencoded 인코딩을 사용하는 GET/POST 방식만을 지원했었다. 따라서 multipart/form-data 인코딩을 사용해야 하는 파일 전송의 경우는 HttpMessage로는 해결할 수 없었다.  
  
이를 날카롭게 지적해주신 어떤 자바캔 회원님께서 필자에게 multipart/form-data 인코딩도 지원해주는 HttpMessage 클래스를 작성해줄 것을 요청했으며, 그에 따라 이번 기사에서는 multipart/form-data 인코딩까지 지원해주는 HttpRequestor 클래스에 대해서 살펴보도록 하겠다.  
  
**HttpRequestor의 구현**  
  
HTTP를 기반으로 하여 데이터를 송수신하기 위해서는 먼저 데이터를 전송할 대상 URL을 지정해주어야 한다. 이는 HttpRequestor의 생성자를 통해서 이루어지며, HttpRequestor의 생성자는 다음과 같이 정의되어 있다.

    public final class HttpRequestor {  
          
        public static final String CRLF = "\r\n";  
          
        /\*\*  
         \* 연결할 URL  
         \*/  
        private URL targetURL;  
          
        /\*\*  
         \* 파라미터 목록을 저장하고 있다.  
         \* 파라미터 이름과 값이 차례대로 저장된다.  
         \*/  
        private ArrayList list;  
          
        public HttpRequestor(URL target) {  
            this(target, 20);  
        }  
          
        /\*\*  
         \* HttpRequest를 생성한다.  
         \*   
         \* @param target HTTP 메시지를 전송할 대상 URL  
         \*/  
        public HttpRequestor(URL target, int initialCapicity) {  
            this.targetURL = target;  
            this.list = new ArrayList(initialCapicity);  
        }  
          
        ...  
    }

"\r\n"을 값으로 갖는 상수 CRLF를 정의하였다. 이는 줄 구분을 할 때 사용된다. (HTTP 프로토콜은 줄 구분을 "\r\n"으로 하도록 되어 있다.) 필드에는 targetURL과 list가 존재한다. targetURL은 연결할 URL을 나타낸다. 물론, HTTP 프로토콜을 위한 URL이 될 것이다. list는 파라미터 목록을 저장하기 위해서 사용된다. HttpRequestor는 HashMap이나 Properties와 같은 것을 사용하여 파라미터 이름과 값을 별도로 저장하지 않고 ArrayList에 모두 저장한다. 이때 list에는 *파라미터이름1, 값1, 파라미터이름2, 값2, ...*와 같이 파라미터 이름과 파라미터 값이 차례대로 저장된다.  
  
전달될 파라미터를 지정은 두 메소드를 통해서 이루어진다. 첫번째 메소드는 텍스트 파라미터를 지정해주는 addParameter(String parameterName, String parameterValue) 메소드이고 다른 하나는 파일 파라미터를 지정해주는 public void addFile(String parameterName, File parameterValue) 메소드이다. 이 두 메소드는 다음과 같다.

    /\*\*  
     \* 파라미터를 추가한다.  
     \* @param parameterName 파라미터 이름  
     \* @param parameterValue 파라미터 값  
     \* @exception IllegalArgumentException parameterValue가 null일 경우  
     \*/  
    public void addParameter(String parameterName, String parameterValue) {  
        if (parameterValue == null)   
        throw new IllegalArgumentException("parameterValue can't be null!");  
          
        list.add(parameterName);  
        list.add(parameterValue);  
    }  
      
    /\*\*  
     \* 파일 파라미터를 추가한다.  
     \* 만약 parameterValue가 null이면(즉, 전송할 파일을 지정하지 않는다면  
     \* 서버에 전송되는 filename 은 "" 이 된다.  
     \*   
     \* @param parameterName 파라미터 이름  
     \* @param parameterValue 전송할 파일  
     \* @exception IllegalArgumentException parameterValue가 null일 경우  
     \*/  
    public void addFile(String parameterName, File parameterValue) {  
        // paramterValue가 null일 경우 NullFile을 삽입한다.  
        if (parameterValue == null) {  
            list.add(parameterName);  
            list.add(new NullFile());  
        } else {  
            list.add(parameterName);  
            list.add(parameterValue);  
        }  
    }

위 코드에서 addFile() 메소드를 살펴보자. parameterValue가 null일 경우 값 부분에 NullFile을 전달해주는 것을 알 수 있다. 이는 전송할 파일을 지정하지 않은 것과 같은 효과를 주기 위한 것이다. 예를 들어, 게시판과 같은 곳에서 업로드할 파일을 지정하지 않은 것과 같이 전송할 파일을 지정하지 않을 때에는 parameterValue를 null로 지정해주면 된다. 참고적으로 NullFile 클래스는 HttpRequestor의 이너 클래스로 정의하였으며 다음과 같다.

    private class NullFile {  
        NullFile() {  
        }  
        public String toString() {  
            return "";  
        }  
    }

application/x-www-form-urlencoded 인코딩을 사용하는 GET/POST 방식은 파라미터 값을 인코딩해서 보내야 한다. 이는 [***자바로 구현하는 Web-to-web 프로그래밍, Part 1***](http://www.javacan.com/jsp/contents/read/article_35.html)에서 사용한 방식을 그대로 사용하였다. HttpRequestor 클래스는 인코딩을 손쉽게 할 수 있는 메소드인 encodeString 메소드를 정의하고 있으며 encodeString() 메소드는 다음과 같다.

    private static String encodeString(ArrayList parameters) {  
        StringBuffer sb = new StringBuffer(256);  
          
        Object[] obj = new Object[parameters.size()];  
        parameters.toArray(obj);  
          
        for (int i = 0 ; i < obj.length ; i += 2) {  
            **if ( obj[i+1] instanceof File || obj[i+1] instanceof NullFile ) continue;**              
            **sb.append(URLEncoder.encode((String)obj[i]) );**            **sb.append('=');**            **sb.append(URLEncoder.encode((String)obj[i+1]) );**              
            if (i + 2 < obj.length) sb.append('&');  
        }  
          
        return sb.toString();  
    }

encodeString() 메소드를 살펴보면 파라미터의 값이 File이나 NullFile인 경우에는 처리하지 않는 것을 알 수 있다. 이는 application/x-www-form-urlencoded 인코딩에서의 파일 전송은 의미가 없기 때문이다.  
  
**전송을 처리해주는 메소드의 구현**  
  
이제 실제로 파라미터의 값을 지정한 URL로 전송해주는 메소드를 살펴보도록 하자. 파라미터를 전송해주는 메소드는 sendGet(), sendPost() 그리고 sendMultipartPost() 이렇게 3가지가 존재한다. sendGet()과 sendPost() 메소드는 [***자바로 구현하는 Web-to-web 프로그래밍, Part 1***](http://www.javacan.com/jsp/contents/read/article_35.html)의 sendGet(), sendPost() 메소드와 거의 동일하므로 여기서는 간단하게 소스 코드만 보여주도록 하겠다. 이 두 메소드의 코드는 다음과 같다.

    /\*\*  
     \* GET 방식으로 대상 URL에 파라미터를 전송한 후  
     \* 응답을 InputStream으로 리턴한다.  
     \* @return InputStream  
     \*/  
    public InputStream **sendGet()** throws IOException {  
        String paramString = null;  
        if (list.size() > 0)  
            paramString = "?" + encodeString(list);  
        else  
            paramString = "";  
          
        URL url = new URL(targetURL.toExternalForm() + paramString);  
          
        URLConnection conn = url.openConnection();  
          
        return conn.getInputStream();  
    }  
      
    /\*\*  
     \* POST 방식으로 대상 URL에 파라미터를 전송한 후  
     \* 응답을 InputStream으로 리턴한다.  
     \* @return InputStream  
     \*/  
    public InputStream **sendPost()** throws IOException {  
        String paramString = null;  
        if (list.size() > 0)  
            paramString = encodeString(list);  
        else  
            paramString = "";  
          
        HttpURLConnection conn = (HttpURLConnection)targetURL.openConnection();  
        conn.setRequestMethod("POST");  
        conn.setRequestProperty("Content-Type",  
                                "application/x-www-form-urlencoded");  
        conn.setDoInput(true);  
        conn.setDoOutput(true);  
        conn.setUseCaches(false);  
          
        DataOutputStream out = null;  
        try {  
            out = new DataOutputStream(conn.getOutputStream());  
            out.writeBytes(paramString);  
            out.flush();  
        } finally {  
            if (out != null) out.close();  
        }  
        return conn.getInputStream();  
    }

크게 어려운 부분이 없으므로 설명은 생략하기로 하겠다.  
  
이제 이 글의 핵심 부분인 sendMultipartPost() 메소드에서 대해서 살펴보자. 이 메소드를 살펴보기 위해서는 먼저 multipart/form-data 인코딩 방식의 데이터가 웹 서버에 어떤 형태로 전달되는 지 알아야만 한다. 이를 위해 80 포트로 들어오는 데이터를 그대로 출력해주는 TestServer.java를 작성해보았다. TestServer.java는 다음과 같다.

    import java.io.\*;  
    import java.net.\*;  
      
    public class TestServer {  
        public static void main(String[] args) throws IOException {  
            ServerSocket ss = new ServerSocket(80);  
              
            Socket socket = ss.accept();  
            InputStream is = socket.getInputStream();  
            BufferedReader br = new BufferedReader(  
                                    new InputStreamReader(is));  
            String line = null;  
            while ( (line = br.readLine()) != null) {  
                System.out.println(line);  
            }  
            br.close();  
            ss.close();  
        }  
    }

TestServer를 수행한 다음에 다음과 같은 HTML 폼을 사용하여 텍스트 파일을 전송해보았다.

    <html><body>  
    <form action="http://localhost/test.jsp" method="post"  
          enctype="multipart/form-data">  
    이름: <input type="text" name="name"> <br>  
    <input type="file" name="upload">  
    <br>  
    <input type="submit">  
    </form>  
    </body></html>

HTML 페이지에서 Submit 버튼을 누르면 TestServer는 다음과 비슷한 결과를 출력하게 된다.

    POST /test.jsp HTTP/1.1  
    Accept: image/gif, image/jpeg, image/pjpeg, \*/\*  
    Accept-Language: ko  
    **Content-Type: multipart/form-data; boundary=---------------------------7d1539170136**    Accept-Encoding: gzip, deflate  
    User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)  
    Host: localhost  
    Content-Length: 558  
    Connection: Keep-Alive  
    Cache-Control: no-cache  
    Cookie: JSESSIONID=aaa7qyaGZraPlm; id=admin; name=adminname  
      
    **-----------------------------7d1539170136**    Content-Disposition: form-data; name="name"  
      
    이름에 넣은 값  
    **-----------------------------7d1539170136**    **Content-Disposition: form-data; name="upload"; filename="C:\work\framework\test\**    formtest.html"  
    Content-Type: text/html  
      
    <html><body>  
    <form action="http://localhost/test.jsp" method="post"  
          enctype="multipart/form-data">  
    이름: <input type="text" name="name"> <br>  
    <input type="file" name="upload">  
    <br>  
    <input type="submit">  
    </form>  
    </body></html>  
    **-----------------------------7d1539170136--**

위에서 "Cookie:" 까지가 헤더에 해당하는 부분인데, 이 헤더 정보의 Content-Type을 살펴보면 컨텐츠 타입을 multipart/form-data로 지정한 것을 알 수 있으며, 또한 boundary를 지정한 것을 알 수 있다. 이 boundary는 각각의 파라미터를 구분할 때 사용되는 문자열로서 각 파라미터 사이의 경계라고 생각하면 된다.  
  
실제로 경계에서 사용되는 boundary는 앞에 '--'가 추가된다. 그리고 가장 마지막 boundary에는 '--'가 뒤에 추가된다. 실제로 위 전송 데이터에서 각 boundary를 추출해서 비교하면 다음과 같다.

* ---------------------------7d1539170136 : content-Type에서의 boundary의 값:
* -----------------------------7d1539170136 : 각 파라미터 사이의 boundary
* -----------------------------7d1539170136-- : 가장 마지막 boundary

위 전송 결과를 통해서 알 수 있는 또 하나의 사실은 multipart/form-data는 파라미터 값을 인코딩하지 않고 그대로 전송한다는 점이다. 따라서 application/x-www-form-urlencoded 인코딩을 위해서 사용한 encodeString() 메소드를 사용할 필요가 없다.  
  
지금까지의 내용을 통해서 multipart/form-data MIME 타입을 사용할 경우 어떻게 데이터를 전송해야 하는지에 대해서 어느 정도 감을 잡았을 것이다. Socket을 사용하여 직접 위와 같은 형태로 데이터를 전송해도 되지만 Java에는 java.net.URLConnection이라는 클래스가 있으며, 필자도 URLConnection을 사용하여 구현해보았다.  
  
sendMultipartPost() 메소드는 앞에서 살펴본 sendGet() 메소드와 sendPost() 메소드에 비해 코드가 길다. 하지만, 메소드를 부분부분 보여주는 것보다는 한번에 다 보여주는 것을 자바캔 회원들이 좋아할 것 같아서 먼저 sendMultipartPost() 메소드의 완전한 코드부터 시작하기로 하자.

    public InputStream sendMultipartPost() throws IOException {  
        HttpURLConnection conn = (HttpURLConnection)targetURL.openConnection();  
          
        // Delimeter 생성  
        **String delimeter = makeDelimeter();**          
        byte[] newLineBytes = CRLF.getBytes();  
        **byte[] delimeterBytes = delimeter.getBytes();**        **byte[] dispositionBytes = "Content-Disposition: form-data; name=".getBytes();**        **byte[] quotationBytes = "\"".getBytes();**        **byte[] contentTypeBytes = "Content-Type: application/octet-stream".getBytes();**        byte[] fileNameBytes = "; filename=".getBytes();  
        byte[] twoDashBytes = "--".getBytes();  
          
        **conn.setRequestMethod("POST");**        **conn.setRequestProperty("Content-Type",**                                **"multipart/form-data; boundary="+delimeter);**        conn.setDoInput(true);  
        conn.setDoOutput(true);  
        conn.setUseCaches(false);  
          
        BufferedOutputStream out = null;  
        try {  
            out = new BufferedOutputStream(conn.getOutputStream());  
              
            Object[] obj = new Object[list.size()];  
            list.toArray(obj);  
              
            for (int i = 0 ; i < obj.length ; i += 2) {  
                // Delimeter 전송  
                out.write(twoDashBytes);  
                out.write(delimeterBytes);  
                out.write(newLineBytes);  
                // 파라미터 이름 출력  
                out.write(dispositionBytes);  
                out.write(quotationBytes);  
                out.write( ((String)obj[i]).getBytes() );  
                out.write(quotationBytes);  
                **if ( obj[i+1] instanceof String) {**                    // String 이라면  
                    out.write(newLineBytes);  
                    out.write(newLineBytes);  
                    // 값 출력  
                    **out.write( ((String)obj[i+1]).getBytes() );**                    out.write(newLineBytes);  
                } else {  
                    // 파라미터의 값이 File 이나 NullFile인 경우  
                    **if ( obj[i+1] instanceof File) {**                        File file = (File)obj[i+1];  
                        // File이 존재하는 지 검사한다.  
                        **out.write(fileNameBytes);**                        out.write(quotationBytes);  
                        **out.write(file.getAbsolutePath().getBytes() );**                        out.write(quotationBytes);  
                    **} else {**                        // NullFile 인 경우  
                        **out.write(fileNameBytes);**                        out.write(quotationBytes);  
                        out.write(quotationBytes);  
                    }  
                    out.write(newLineBytes);  
                    **out.write(contentTypeBytes);**                    out.write(newLineBytes);  
                    out.write(newLineBytes);  
                    // File 데이터를 전송한다.  
                    **if (obj[i+1] instanceof File) {**                        File file = (File)obj[i+1];  
                        // file에 있는 내용을 전송한다.  
                        BufferedInputStream is = null;  
                        try {  
                            is = new BufferedInputStream(  
                                     new FileInputStream(file));  
                            byte[] fileBuffer = new byte[1024 \* 8]; // 8k  
                            int len = -1;  
                            while ( (len = is.read(fileBuffer)) != -1) {  
                                out.write(fileBuffer, 0, len);  
                            }  
                        } finally {  
                            if (is != null) try { is.close(); } catch(IOException ex) {}  
                        }  
                    }  
                    out.write(newLineBytes);  
                } // 파일 데이터의 전송 블럭 끝  
                if ( i + 2 == obj.length ) {  
                    **// 마지막 Delimeter 전송**                    out.write(twoDashBytes);  
                    out.write(delimeterBytes);  
                    out.write(twoDashBytes);  
                    out.write(newLineBytes);  
                }  
            } // for 루프의 끝  
              
            out.flush();  
        } finally {  
            if (out != null) out.close();  
        }  
        **return conn.getInputStream();**    }

먼저 sendMultipartPost() 메소드는 포스팅에서 사용되는 "Content-Disposition: form-data; name="과 같은 다양한 문자열의 byte[] 배열을 구한다. 이처럼 문자열을 byte로 변환하는 이유는 multipart/form-data가 텍스트 데이터만 사용되는 것이 아니라 바이트 기반의 데이터(예를 들어, 전송할 파일)도 전송하기 때문이다. 물론, sendMultipartPost() 메소드의 앞부분에서 정의한 byte[] 배열들은 필드로 정의하여도 무방하다. (실제로 필드로 정의하는 것이 더 좋을 것이다.)  
  
데이터 전송에서 사용되는 문자열의 byte 배열을 구한 이후에는 HttpURLConnection에 필요한 헤더 정보를 지정한다. 특히 중요한 부분은 Content-Type 헤더 값을 "multipart/form-data"로 지정하는 것이다. 물론, 이때 boundary도 함께 지정해준다. 헤더 정보에 대한 기본 설정이 완료되면 HttpURLConnection으로부터 출력 스트림을 구한다.  
  
출력 스트림을 구한 이후에는 파라미터값을 저장하고 있는 출력 스트림에 차례대로 출력하기 시작한다. 이때 String 파라미터냐 File 파라미터냐에 따라서 알맞게 데이터를 출력해준다. out.write() 메소드를 사용하여 출력하는 데이터의 순서를 잘 보면 앞에서 multipart/form-data로 데이터를 전송할 때의 규칙과 같다는 점을 알 수 있다. 파라미터 타입이 File일 경우에는 추가적으로 파일의 내용을 전송해준다.  
  
File을 전송할 때 "Content-Type: application/octet-stream" 문장을 추가해주는 것을 알 수 있는데 여기서 application/octet-stream은 전송할 파일이 바이너리 파일이라는 것을 나타낸다. 여기서 바이너리 파일로 단정지은 것은 텍스트 파일의 경우 File이 아닌 텍스트로도 충분히 전송할 수 있기 때문이며, 또한 파일 전송은 대부분 바이너리 파일이 대상이기 때문이다.  
  
boundary를 구할 때 makeDelimeter() 메소드를 사용하는데, 원칙적으로는 makeDelimeter() 메소드를 실행할 때 마다 다른 boundary 값을 리턴해야 하겠지만 필자는 매번 같은 boundary를 리턴하였다. 왜냐면 매번 같은 boundary를 리턴한다 해서 문제될 것이 없기 때문이다. 필자는 makeDelimeter() 메소드를 다음과 같이 구현하였다.

    private static String makeDelimeter() {  
        return "---------------------------7d115d2a20060c";  
    }

**HttpRequestor 클래스의 사용**  
  
HttpRequestor 클래스의 사용방법은 매우 간단하다. 간단히 예를 들면 다음과 같다.

    HttpRequestor requestor = new HttpRequestor(someURL);  
    requestor.addParameter("param1", "파라미터1");  
    requestor.addParameter("param2", "parameter2");  
    requestor.addFile("file1", null); // 전송할 파일을 지정하지 않는 경우  
    requestor.addFile("file2", new File("c:\\autoexec.bat"));  
      
    InputStream is = requestor.sendMultipartPost();  
    BufferedReader br = new BufferedReader(new InputStreamReader(is));  
    ...  
    ...  
    br.close();

**결론**  
  
이번 글에서는 업로드한 한 파일을 처리해주는 유틸리티 클래스와 정반대의 기능을 수행해주는 HttpRequestor 클래스에 대해서 살펴보았다. 비록 여러분이 이 클래스를 직접적으로 사용하게 될 경우는 많지 않겠지만 웹 기반의 통신을 해야 하는 경우, 특히 이미지 파일이나 문서 파일등을 서로 주고 받아야 하는 경우 multipart/form-data의 지원은 필수적으로 할 수 있겠다. (필자도 multipart/form-data를 지원하지 않는 HttpMessage 때문에 손해아닌 손해를 본적이 있다.) 또한 굳이 multipart/form-data 방식의 데이터 교환을 사용할 필요가 없다고 해도 어떤식으로 multipart/form-data가 전송되는 지 아는 것도 나름대로 의미가 있을 것이라 생각한다.  
  
관련링크:

* [**자바로 구현하는 Web-to-web 프로그래밍, Part 1**](http://javacan.tistory.com/entry/35)
* [**자바로 구현하는 Web-to-web 프로그래밍, Part 2**](http://javacan.tistory.com/entry/42)