File uploading and Java hash map data structure

Conference Paper · May 2011

2 authors:

Keivan Borna
Kharazmi University

45 PUBLICATIONS **61** CITATIONS

Zahra Nilforoushan
Kharazmi University, Faculty of Engineering

12 PUBLICATIONS **63** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Right now I'm working on three projects: "Finger print problem using split trees, "Approximation of Voronoi diagram due to decrease computations" and "A special case of Voronoi games" View project

Project   GBML System View project

The 2nd International Conference On

Contemporary Issues in Computer and Information Sciences

Conference Proceeding

CICIS 2011

May 31 - June 2 , 2011

Institute for Advanced Studies
in Basic Sciences
Gava Zang, Zanjan, Iran

DEBI

Zanjan University

IEEE
IRAN SECTIOON

File uploading and Java Hash map data structure

Keivan Borna and Zahra Nilforoushan

Faculty of Mathematical Sciences and Computer

Tarbiat Moallem University

Tehran, Iran

borna@tmu.ac.ir

Abstract— **The main objective of this paper is to provide a dependable, secure and strong method and a program for the file uploading using Java Servlets and JSP. We design and develop two servlets UploadHandler and MultipartRequest which are responsible for managing the uploading process. Our method and its performance is based on the high functionality of maps.**

Keywords-component; file uploading; hash map data structure; Java programming language; performance improvement.

I. INTRODUCTION

File upload is too rarely discussed by even respectable Java literature and with the growth of the Internet, file upload has now also played significant roles beyond email applications. Other Internet/Intranet applications such as Web-based document management systems and the likes of "Secure File Transfer via HTTP" require uploading files to the server extensively. For browsing some good survey on file/bean uploading with Java servlets see [1, 3, 4, 5, 2] where some framework for building scalable wide-area upload applications is provided. Furthermore uploads correspond to an important class of applications, whose examples include a large number of digital government applications. This paper discusses file uploading using Java hash map data structure. We need to understand the underlying theory, i.e., the map functionality and the HTTP request. This is done in Sections II and III. In Section IV we give Java pseudo-codes for developing our methods.

II. MAP FUNCTIONALITY

An *array list* allows one to select from a sequence of objects using a number, so in a sense it associates numbers to objects. But what if we like to select from a sequence of objects using some other criterion. A stack is an example: its selection criterion is "the last thing pushed on the stack". A powerful twist on this idea of "selecting from a sequence" is alternately termed a map, a dictionary, or an associative array. Conceptually, it seems like an array list, but instead of looking up objects using a number, we look them up using another object. This is a key technique in programming. The concept shows up in Java as the map interface.

Hash table based implementation of the map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The hash map class is roughly equivalent to hash table, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. This implementation provides constant-time performance for the basic operations "*get*" and "*put*"), assuming the hash function disperses the elements properly among the buckets. As an example of the use of a hash map, consider a program to check the randomness of Java's Random class. Ideally, it would produce a perfect distribution of random numbers, but to test this we need to generate a bunch of random numbers and count the ones that fall in the various ranges. A hash map is perfect for this, since it associates objects with objects (in this case, the value object contains the number produced by "*Math.random()*" along with the number of times that number appears).

An instance of hash map has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method. Iteration over collection views requires time proportional to the capacity of the hash map instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it is very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important. Performance can be adjusted via constructors that allow us to set the capacity and load factor of the hash table. As a general rule, the default load factor ($.75$) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the hash map class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. If many mappings are to be stored in a hash map instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

A big issue with maps is performance. If one look at what must be done for a "*get()*", it seems pretty slow to search

through (for example) an array list for the key. This is where hash map speeds things up. Instead of a slow search for the key, it uses a special value called a *hash code*. The hash code is a way to take some information in the object in question and turn it into a "relatively unique" int for that object. All Java objects can produce a hash code, and *hashCode()* is a method in the root class Object. A hash map takes the hash code of the object and uses it to quickly hunt for the key. This results in a dramatic performance improvement. Hashing is the most commonly-used way to store elements in a map.

The "*put(Object key, Object value)*" method adds a value (the thing we want), and associates it with a key (the thing we look it up with). "*get(Object key)*" produces the value given the corresponding key. One can also test a map to see if it contains a key or a value with "*containsKey()*" and "*containsValue()*".

The standard Java library contains different types of maps:

HashMap, *TreeMap*, *LinkedHashMap*, *WeakHashMap* and *IdentityHashMap*. They all have the same basic map interface, but they differ in behaviors including efficiency, order in which the pairs are held and presented, how long the objects are held by the map, and how key equality is determined. For the ease of reader we bring the detailed descriptions of each map type.

Map: This interface maintains key-value associations (pairs), so we can look up a value using a key.

HashMap: Implementation based on a hash table. (Hash map is used normally instead of hash table.) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow us to set the capacity and load factor of the hash table.

LinkedHashMap: (JDK 1.4) Like a hash map, but when we iterate through it we get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than a hash map, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.

TreeMap: Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order. The point of a TreeMap is that we get the results in sorted order. TreeMap is the only map with the subMap() method, which allows us to return a portion of the tree.

WeakHashMap: A map of weak keys that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.

IdentityHashMap: (JDK 1.4) A hash map that uses == instead of "*equals()*" to compare keys. Only for solving special types of problems; not for general use.

III.   THE HTTP REQUEST

Knowledge of the HTTP request is critical because when we process an uploaded file, we work with raw data not obtainable from an *HttpServletRequest* object's methods such as *getParameter*, *getParameterNames*, or *getParameterValues*.

Each HTTP request from the Web browser or other Web client applications consists of three parts:

A. A line containing the HTTP request method, the Uniform Resource Identifier (URI), and the protocol versions

B. HTTP request headers

C. The entity body

A.   The request method, URI and protocol versions

The first subpart of the first part, the HTTP request method, indicates the method used in the HTTP request. In HTTP 1.0, it could be one of the following three: get, head, or post. In HTTP 1.1, in addition to the three methods, there are four more methods: delete, put, trace, and options. Among the seven, the two methods that are most frequently used are *get* and *post*. *get* is the default method. We use it, for example, when you type a URL such as "http://www.onjava.com" in the Location or Address box of your browser to request a page. The post method is common too. We normally use this as the value of the form tag's method attribute. When uploading a file, we must use the post method. The second part of the first part, the URI, specifies an Internet resource. A URI is normally interpreted as being relative to the Web server's root directory. Thus, it starts with a forward slash (/) that is of the form /virtualRoot/pageName for example, in a typical JSP application the URI could be the following /eshop/login.jsp. More information about URI can be found in 2. The third component of the first part is the protocol and the protocol version understood by the requester (the browser). The protocol must be HTTP and the version could be 1.0 or 1.1. Most Web servers understand both versions 1.0 and 1.1 of HTTP. Therefore, this kind of Web server can serve HTTP requests in both versions as well. Combining the three sub-parts of the first component of an HTTP request, the first component would look like the following.

POST/virtualRoot/pageName HTTP/version

For instance, POST/eshop/login.jsp HTTP/1.1

B.   The HTTP request headers

The second component of an HTTP request consists of a number of HTTP headers. There are four types of HTTP headers: general, entity, request, and response. These headers are summarized in the following. The response headers are HTTP response specific, thus not relevant to be discussed here.

Pragma header: The Pragma general header is used to include implementation specific directives that may apply to any recipient along the request/response chain. This is to say that pragmas notify the servers that are used to send this request to behave in a certain way. The Pragma header may contain multiple values. For example, the following line of code inform all proxy servers that relay this request not to use a cached version of the object but to download the object from the specified location: Pragma: no-cache.

Allow header: This header lists the set of method supported by the resource identified by the requested URL. The purpose of this field is strictly to inform the recipient

of valid methods associated with the resource. The Allow header is not permitted in a request using the post method, and thus should be ignored if it is received as part of a post entity. For instance, Allow: get, head.

Content-Encoding header: This header is used to describe the type of encoding used on the entity. When present, its value indicates the decoding mechanism that must be applied to obtain the media type referenced by the Content-Type header. For example, Content-Encoding: x-gzip.

Content-Length header: This header indicates the size of the entity-body, in decimal number of octets, sent to the recipient or, in the case of the head method, the size of the entity-body that would have been sent had the request been a get. Applications should use this field to indicate the size of the entity-body to be transferred, regardless of the media type of the entity. A valid Content-Length field value is required on all HTTP/1.0 request messages containing an entity-body. Any Content- Length header greater than or equal to zero is a valid value. For example, Content-Length:32345.

Content-Type header: The Content-Type header indicates the media type of the entity-body sent to the recipient or, in the case of the head method, the media type that would have been sent had the request been a get. For example, Content-Type: text/html.

Expires header: The Expires header gives the date and time after which the entity should be considered invalid. This allows information providers to suggest the volatility of the resource or a date after which the information may no longer be accurate. Applications must not cache this entity beyond the date given. The presence of an Expires header does not imply that the original resource will change or cease to exist at, before, or after that time. However, information providers should include an Expires header with that date. For example, Expires: Thu, 29 Mar 2011 23:30:00 GMT.

Last-Modified header: The Last-Modified header indicates the date and time at which the sender believes the resource was last modified. The exact semantics of this field are defined in terms of how the recipient should interpret it. If the recipient has a copy of this resource that is older than the date given by the Last-Modified field, that copy should be considered stale For example, Last-Modified: Thu, 10 Aug 2000 12:12:12 GMT.

C. The entity body

The entity body is the content of the HTTP request itself. It is best to illustrate this with an example. An example of an HTTP header is given below.

Accept: application/vnd.ms-excel, application/msword,

Accept-Language:en-au
Connection:Keep-Alive
Host:localhost
Referrer:http://localhost/examples/jsp/num/demo.jsp
User-Agent:Mozilla/4.0 (compatible; Windows 98)
Content-Length:31
Content-Type:application/x-www-form-urlencoded
Accept-Encoding:gzip,deflate
LastName=borna,FirstName=keivan

A lot is revealed in the HTTP header above. The first line tells us that the browser that sends this request can accept a number of file formats, including Microsoft Excel and Microsoft Word. It is followed by the language used (in this case, Australian English), the type of connection (keep-alive), and the name of the host (localhost). It also tells the server that the request is sent from the demo.jsp which is located in http://localhost/examples/jsp/num/ directory. Then in the User-Agent entry, the Request tells that the user is using a Microsoft Internet Explorer version 4.01. The user operation system is also recorded to be Windows 98. Following the header is two pairs of carriage-return line-feed characters. The length of this separator is 4 bytes since each carriage-return line-feed character pair consists of the ASCII characters number 13 and 10. From the HTTP header above we can see that the body consists of the following code which has length 31:

LastName=borna,FirstName=keivan

This is clearly from a form with two input boxes: one called LastName with the value "borna" and the other named FirstName with the value "keivan".

IV. IMPLEMENTATIONS

We just finished dissecting an HTTP request and now we are ready to take this information to the coding stage. To program a complete file upload application, we need to know both the server side and the client side. First we present how to program the HTML on the client side, and then we will study the Java code for the server side.

A. The client side HTML

Prior to the RFC 1867 standard there were eight possible values for the type attribute of an input element: checkbox, hidden, image, password, radio, reset, submit, and text; see 5. While these form elements have proven useful in a wide variety of applications in which input from the user needs to be transferred to the server, none of these is useful for sending a file, either text or binary. Next, it was proposed that the type attribute of an input element has another possible value: file. In addition, it defines a new MIME media type, multipart/form-data, and specifies the behavior of HTML user agents when interpreting a form with enctype="multipart/form-data". More precisely, the client side looks as follows:

```
<form action="servlet/UploadHandler"
enctype="multipart/form-data"    method="post"\>
  File to Upload: <input name="filename" type="file"/>
  <input type="submit" value="Browse" />
```

</form>

When an input tag of type file is encountered, the browser might show a display of previously selected file names and a "Browse" button or selection method. Selecting the "Browse" button would cause the browser to enter into a file selection mode appropriate for the platform. Window-based browsers might pop up a file selection window, for example. In such a file selection dialog, the user would have the option of replacing a current selection, adding a new file selection, etc. For more details about this API see Sun's web site. What is of importance here is the HTTP request and how to process it. When working with the HTTP request, we will work with either *the javax.servlet.ServletRequest* interface or the *javax.servlet.http.HttpServletRequest* interface.

B. The server side code

Internet programming in Java always involves the use of servlets or JSP pages, depending on the architecture one choose to implement. This means that we will use one of the classes or interfaces in the *javax.servlet* package or the *javax.servlet.http* package. The most important interface is the servlet interface in *the javax.servlet* package that must be implemented by all servlets. The ServletRequest interface is a generic interface that is extended by *HttpServletRequest* to provide request information for HTTP servlets. We leave to the reader studying *HttpServletRequest* interface important methods. In order to describe our method server-side's part we use of two Java servlets named *UploadHandler* and *MultipartRequest*. The first servlet, UploadHandler, is responsible for the output. We mention that our implementation also holds for a wide range of uploading several files simultaneously. The second servlet, MultipartRequest, is responsible for holding the uploaded files, parameters and the parsed results. The Java class MultipartRequest consists of a private inner class named *Source*, a public inner class named *FileInfo*, the constructor, and some methods in order to describe the files encapsulated in the request. A general view of this class is shown in the following:

```
public class MultipartRequest {
    // To hold parsed results
    private MimeMultipart mimeparts;
```

```
    // To hold uploaded parameters
    private HashMap params = new HashMap();
    // To hold uploaded files
    private HashMap files = new HashMap();
    // Scratch buffer space
    private byte [] buf = new byte[8096];
    private class Source implements
javax.activation.DataSource{}
    // The constructor
    public MultipartRequest(HttpServletRequest req) throws
        MessagingException, IOException {}
    // getParameter, getParameterValues, getFileInfoNames,
//getFileInfo and getFileInfoValues methods in order to
//describe the files encapsulated in the request.
}
```

The private inner class *Source* which implements *javax.activation.DataSource* is a simple implementation of the DataSource interface. It provides the bridge between the HttpServletRequest and the Java Mail classes. The public inner class *FileInfo* is used to store information about uploaded files. Users of the MultipartRequest class should generally refer to this class as MultipartRequest.FileInfo (as for usual Java rules). The constructor accepts an *HttpServletRequest* (which it assumes to be from a post of a ENCTYPE="multipart/form-data" form) and parses all the information into a MimeMultipart object. It then iterates through that parsed object, extracting the parameters and files from it for the user. Finally we note that *BufferedInputStream* and *ByteArrayOutputStream* are used to handle the required processing for a file and to extract a parameter value from a header line.

REFERENCES

[1]   T. Anderson, "XForms tip: Use XForms to upload a file to Java", IBM series.

[2]   Samrat Bhattacharjee, William C. Cheng, Cheng-Fu Chou, Leana Golubchik, Samir Khuller, "Bistro: a framework for building scalable wide-area Upload applications", ACM SIGMETRICS Performance Evaluation Review, 28, Issue 2 (2000), 29-35.

[3]   B. Kurniawan, "Uploading files with Beans", ONJava.com.

[4]   E. Nebel, "Form-based file upload in HTML", Xerox Corporation, 1995.

[5]   Yan Yang, Leslie Cheung, Leana Golubchik, "Technology to support data gathering via the web: Data assignment in fault tolerant uploads for digital government applications: a genetic algorithms approach", Proceedings of the 2005 national conference on Digital government research dg.o2005, 2005.