TAR File Management With PHP Archive_Tar

By The Disenchanted Developer

2003−07−17

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/PHP/TAR_File/

Rolling Your Own

The PHP user community never ceases to amaze me.

No fewer than six times in the past few months have I needed a custom thingammyjig for one of my projects and, after having made the decision to design and code it myself, checked Google on the off−chance that someone had already written it and found exactly what I needed in a PHP user group or code repository. Not only is this incredibly cool − like all lazy programmers, I hate reinventing the wheel − but it's also a testament to the immense popularity of the language, and to the creativity it inspires among developers on a daily basis.

This tutorial deals with one such tool, the Archive_Tar class. This is not a tool you might be familiar with, it's not even a tool that you'll use on a regular basis. However, in case you ever need a way to build a TAR archive through PHP code, it's an invaluable addition to your PHP armory. Keep reading, and let me introduce you.

Back To Basics

The Archive_Tar class is brought to you by PEAR, the PHP Extension and Application Repository (http://pear.php.net). In case you didn't know, PEAR is an online repository of free PHP software, including classes and modules for everything from data archiving to XML parsing. When you install PHP, a whole bunch of PEAR modules get installed as well; the Archive_Tar class is one of them.

In case your PHP distribution didn't include Archive_Tar, or if you chose not to install the PEAR component, you can get yourself a copy from the official PEAR Web site, at http://pear.php.net − simply unzip the distribution archive into your PEAR directory and you're ready to roll!

Let's begin with something simple − packing a bunch of files into a single TAR archive with Archive_Tar object methods (this tutorial uses Archive_Tar 1.11).

```
<?php

// include class
require("Tar.php");

// create Archive_Tar() object
// specify filename for output file
$tar = new Archive_Tar("data.tar");

// set up file list
$files = array("package.dtd", "package.xml", "filter.xsl", "../../includes/php/adodb.php");

// build archive
```

```
$tar−>create($files) or die("Could not create archive!");

?>
```

This is a pretty simple script, but I'll hit the high points anyway:

1. The first step, as always, is to include the required class file in your script.

```
<?php

// include class
require("Tar.php");

?>
```

You can either provide an absolute path to this file, or do what most lazy programmers do − include the path to your PEAR installation in PHP's "include_path" variable, so that you can access any of the PEAR classes without needing to type in long, convoluted file paths.

2. The next step is to instantiate an object of the Archive_Tar class.

```
<?php

// create Archive_Tar() object
// specify filename for output file
$tar = new Archive_Tar("data.tar");

?>
```

Note that the object constructor requires, as argument, the name of the TAR file to be created. You can include a path here as well if you would like the file to be created in a location other than the current directory.

3. Third, you need to tell the newly−created object which files to include in the archive. This information is provided to the object's create() method as an array of file names.

```
<?php

// set up file list
$files = array("package.dtd", "package.xml", "filter.xsl", "../../includes/php/adodb.php");

// build archive
$tar−>create($files) or die("Could not create archive!");

?>
```

Obviously, you should replace the file list in the example above with information that reflects your own system. If Archive_Tar cannot locate any of the files specified in the file list, it will simply skip over it to the next one.

Note that the create() method will replace a previously−existing file with the same name. If you don't want this to happen, you should wrap your call to create() in a file_exists() test.

Once you run this script through your browser, you'll see that a file named "data.tar" appears in the same directory as the script.

```
$ ls −l data.tar
−rwxrw−rw− 1 nobody nobody 72704 Jul 8 14:31 data.tar
```

This is the TAR file created by the lines of code above.

If you examine this file with the "tar" command, you'll see that it contains all the files specified in your PHP script.

```
$ tar −tvf data.tar
−rw−rw−rw− 0/0 2876 2002−04−09 21:04:08 package.dtd
−rw−rw−rw− 0/0 2891 2002−11−18 01:22:20 package.xml
−rw−rw−rw− 0/0 707 2003−02−19 19:18:14 filter.xsl
−rw−rw−rw− 0/0 62790 2002−06−12 17:17:20 includes/php/adodb.php
```

Zip Zap Zoom

It's important to note that the file created in the previous example is an uncompressed archive − the files contained within it are not compressed to save space. If you'd like to create a compressed archive, Archive_Tar supports that option too, allowing you to apply GZIP or BZIP compression to the TAR archive.

Compressing the archive is a piece of cake − simply add the value "gz" or "bz2" as a second argument to the Archive_Tar object constructor when instantiating it, as illustrated below:

```
<?php

// include class
require("Tar.php");

// create Archive_Tar() object
// specify filename for output file and compression method
$tar = new Archive_Tar("data.tar.gz", "gz");
```

```
// set up file list
$files = array("package.dtd", "package.xml", "filter.xsl", "../../includes/php/adodb.php");

// build archive
$tar−>create($files) or die("Could not create archive!");

?>
```

A quick check reveals that the archive file created in this example is much smaller than the one created in the previous example,

```
$ ls −l data.tar.gz
−rwxrw−rw− 1 nobody nobody 19951 Jul 8 2003 data.tar.gz
```

and a call to the "file" utility verifies that it is, indeed, a GZIP−compressed archive.

```
$ file data.tar.gz
data.tar.gz: gzip compressed data, deflated, last modified: Thu Jan 1 05:30:00 1970
```

Note that in order for this to work, your PHP build must include support for the zlib and bzip2 compression libraries. Unix users can enable this support by recompiling PHP with the "−−with−zlib" and "−−with−bz2" arguments to the PHP "configure" script, respectively. Windows users get a better deal: pre−compiled extensions for these libraries are included by default in the Windows PHP distribution, and they can be enabled simply by uncommenting the extensions in the "php.ini" configuration file.

Further instructions on how to perform these procedures are available in the PHP manual and documentation.

Adding It All Up

You can also add files to a previously−created archive with the add() method, which accepts a list of files and merely adds them to the specified archive file. If the file does not already exist, add() will create it for you.

The following example illustrates, by adding some files to the "data.tar" file created in the previous example:

```
<?php

// include class
require("Tar.php");

if (file_exists("data.tar"))
{
// create Archive_Tar() object
// specify filename for output file
$tar = new Archive_Tar("data.tar");

// add files to existing archive
$tar−>add(array("employees.xml", "departments.xml")) or die ("Could not add files!"); } else {
```

```
die("File does not exist!");
}

?>
```

It's important to note that the add() method will add files to the archive even if they already exist. This behaviour can lead to problems in certain situations − for example, if you ran the script above four times, the resulting archive would hold four copies of each of the files in the file list.

The ability to selectively add files to an archive comes in handy when you need to iterate over a file collection and only archive those which meet certain criteria − for example, all those files older than a specific date. Consider the following example, which illustrates how you might do this:

```
<?php

// include class
require("Tar.php");

// get UNIX timestamp for 1 Jun 2003
$ts = mktime(0, 0, 0, 6, 1, 2003);

// set directory
$dir = "scripts/";

// create tar file
$tar = new Archive_Tar("data.tar");

// iterate over files in directory
$handle = opendir("scripts/");
while (false !== ($file = readdir($handle)))
{
if ($file != "." && $file != ".." && filemtime($dir . "/" . $file) < $ts)
{
// if file was modified before $ts
// add to archive
$tar−>add($dir . "/" . $file) or die ("Could not add file!");
}
}
closedir($handle);

?>
```

In this case, the filemtime() function has been used to obtain the last modification time of each file in the named directory. Files with a timestamp older than the specified date get add()−ed to the archive as they are encountered.

Building An Index

You can obtain a list of all the files in a TAR archive with the Archive_Tar object's listContents() method, which returns an array of arrays containing detailed information on the contents of the archive. Each element of the returned array represents a file in the source archive, each element is itself structured as an associative array of key−value pairs containing details on the file name and path (key "filename"), size ("size"), owner and group ID ("uid" and "gid"), permissions ("fileperms"), last modification time ("mtime") and whether the entry is a file or directory ("type").

Consider the following example, which demonstrates by using the
listContents() method to obtain a list of all the files within a TAR file, together with their size and last modification times:

```
<?php

// include class
require("Tar.php");

// create tar file
$tar = new Archive_Tar("data.tar");

// read and print tar file contents
if (($arr = $tar−>listContent()) != 0)
{
foreach ($arr as $a)
{
echo $a['filename'] . ", " . $a['size'] . " bytes, last modified on " . date("Y−m−d", $a['mtime']) . "\r\n";
}
}

?>
```

Here's an example of the output:

```
scripts/pear.bat, 1570 bytes, last modified on 2002−04−09 scripts/pear.in, 6145 bytes, last modified on
2002−04−09 scripts/pearize.in, 4549 bytes, last modified on 2002−01−31 scripts/pearwin.php, 7507 bytes,
last modified on 2002−02−20 scripts/php−config.in, 511 bytes, last modified on 2001−08−13
scripts/phpextdist, 620 bytes, last modified on 2001−01−10 scripts/phpize.in, 727 bytes, last modified on
2002−04−17 scripts/phptar.in, 5322 bytes, last modified on 2001−09−28 scripts/test2.php, 487 bytes, last
modified on 2002−09−28
```

In And Out

The Archive_Tar object also allows you to extract the contents of a compressed (or uncompressed) TAR file with its extract() method, which accepts the target directory as argument. Consider the following example, which demonstrates by extracting the contents of the named archive to the temporary directory:

```
<?php

// include class
require("Tar.php");

// use tar file
$tar = new Archive_Tar("uudeview−0.5.18.tar.gz");

// extract contents of tar file into named directory
$tar−>extract("/tmp/xfiles") or die ("Could not extract files!"); ?>
```

A quick look in "/tmp/xfiles" after running the script above will show you that the files have been successfully extracted.

```
$ ls −lR /tmp/xfiles/
/tmp/xfiles/:
total 4
drwxr−xr−x 8 nobody nobody 4096 Jul 10 12:22 uudeview−0.5.18

/tmp/xfiles/uudeview−0.5.18:
total 264
−rwxrw−rw− 1 nobody nobody 2416 Jun 7 1996 acconfig.h
−rwxrw−rw− 1 nobody nobody 7980 Jun 7 1996 aclocal.m4
−rwxrw−rw− 1 nobody nobody 3544 Jun 4 2001 config.h.in
−rwxrw−rw− 1 nobody nobody 94247 Apr 2 2002 configure
−rwxrw−rw− 1 nobody nobody 17256 Apr 2 2002 configure.in
−rwxrw−rw− 1 nobody nobody 18007 Jun 6 2001 COPYING

... snip ...
```

Note that if the target directory does not exist, Archive_Tar will attempt to create it for you.

You can also selectively extract certain files from the source archive with the Archive_Tar object's extractList() method, which additionally accepts a list of the file names to be pulled out of the archive. Here's an example:

```
<?php

// include class
require("Tar.php");
```

```
// use tar file
$tar = new Archive_Tar("uudeview−0.5.18.tar.gz");

// extract selected files from tar file $tar−>extractList(array("uudeview−0.5.18/configure",
"uudeview−0.5.18/install−sh"), "/tmp/xfiles") or die ("Could not extract files!");

?>
```

The example above would extract only the files "configure" and "install−sh" from the TAR archive into the target directory.

```
$ ls −lR /tmp/xfiles/
/tmp/xfiles/:
total 4
drwxr−xr−x 2 nobody nobody 4096 Jul 10 12:24 uudeview−0.5.18

/tmp/xfiles/uudeview−0.5.18:
total 108
−rwxrw−rw− 1 nobody nobody 94247 Apr 2 2002 configure
−rwxrw−rw− 1 nobody nobody 4772 Jun 7 1996 install−sh
```

Note that you will have to specify the full path (within the archive) to the files to be extracted in order to successfully use the extractList() function. If the path to any of the files specified in the file list is incorrect, extractList() will simply skip over that file.

Absolute Power

For users who need more fine−grained control over the archive creation and extraction process, Archive_Tar offers the createModify(), addModify() and
extractModify() methods. These methods work in the same manner as the create(), add() and extract() methods discussed previously, except that they also allow you to determine the directory structure of the TAR package and its output.

What does this mean? Consider the following simple directory structure:

```
tempstuff/
tempstuff/menu.php
tempstuff/menu.xml
tempstuff/config.inc
```

Now, if you were to create an archive of these files using the create() method, as in the script below,

```
<?php

// include class
require("Tar.php");
```

```
// create Archive_Tar() object
// specify filename for output file and compression method
$tar = new Archive_Tar("menu.tar.gz", "gz");

// set up file list
$files = array("tempstuff/menu.xml", "tempstuff/config.inc", "tempstuff/menu.php");

// build archive
$tar−>create($files) or die("Could not create archive!");

?>
```

the resulting TAR file would look (and be extracted) like this:

tempstuff/
tempstuff/menu.php
tempstuff/menu.xml
tempstuff/config.inc

What the functions above let you do is control the directory structure inside the TAR archive, and thereby determine the output when the archive is exploded. So, if you wanted (for example) to replace the outer directory "tempstuff/" in the archive above with something more descriptive, like "menutools−0.51/scripts/", you could use the createModify() method instead, as in the script below:

```
<?php

// include class
require("Tar.php");

// create Archive_Tar() object
// specify filename for output file and compression method
$tar = new Archive_Tar("menu.tar.gz", "gz");

// set up file list
$files = array("tempstuff/menu.xml", "tempstuff/config.inc", "tempstuff/menu.php");

// build archive
// set up new directory hierarchy within archive $tar−>createModify($files, "menutools−0.51/scripts/",
"tempstuff") or die("Could not create archive!");

?>
```

The only difference between this script and previous ones: this one uses the createModify() method instead of the create() method, and passes that method two additional arguments. The first of these arguments is the directory path to be added to each file ("menutools−0.51/scripts/" in this example); the second is the directory to be amputated when packing the files into the archive ("tempstuff/" in this example).

Now, if you attempt to explode the resulting TAR archive (either using the "tar" command or the Archive_Tar extract() method), the addition of the custom path above will result in the files within the archive being placed in a "menutools−0.51/scripts/" directory in your extraction area.

menutools−0.51/scripts/
menutools−0.51/scripts/menu.php
menutools−0.51/scripts/menu.xml menutools−0.51/scripts/config.inc

In a similar manner, when using the extractList() function, there may arise a situation when you don't want the directory structure maintained within the TAR file to be preserved during the extraction of specific files. In such a case, you can tell Archive_Tar to ignore the relative path and directory tree hierarchy associated with each file, by adding a third, optional, parameter to extractList().

In the following example, even though the files within the archive are organized in a particular hierarchical structure, the third argument to
extractList() tells Archive_Tar to ignore a section of the hierarchy while extracting files.

```
<?php

// include class
require("Tar.php");

// use tar file
$tar = new Archive_Tar("uudeview−0.5.18.tar.gz");

// extract selected files from tar file $tar−>extractList(array("uudeview−0.5.18/configure",
"uudeview−0.5.18/install−sh"), "/tmp/xfiles", "uudeview−0.5.18") or die ("Could not extract files!");

?>
```

The addModify() method lets you do something similar, but with an existing archive.

Look in the class documentation for more examples of how this feature can be used.

X−Ray Vision

So that's the theory − now for a couple of examples that illustrate it in practice. This first example application accepts a TAR file for upload and prints its contents using the Archive_Tar object's listContents() method:

```
<html>

<head>
```

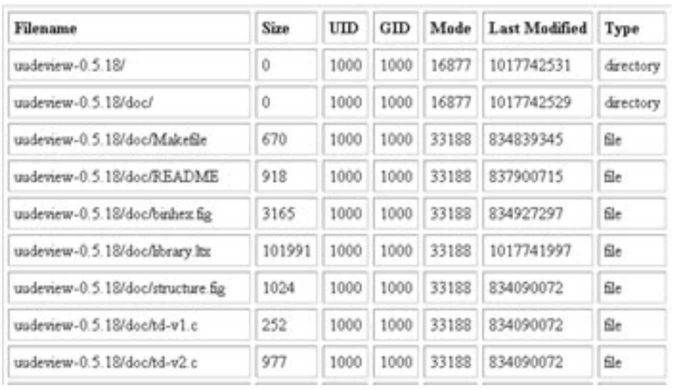
```
        </head>

<body bgcolor="white">
<?
if (!$_POST['submit'])
{
?>
<form action="<?=$_SERVER['PHP_SELF']; ?>" method="POST" enctype="multipart/form−data">
Select a file:
<input type="file" name="file">
<p>
<input type="Submit" name="submit" value="Send File">
</form>
<?
}
else
{
// check to make sure this is a tar file
if ($_FILES['file']['type'] != "application/x−gzip−compressed")
{
die("Unsupported file type!");
}
?>
<table border="1" cellspacing="5" cellpadding="5">
<tr>
<td><b>Filename</b></td>
<td><b>Size</b></td>
<td><b>UID</b></td>
<td><b>GID</b></td>
<td><b>Mode</b></td>
<td><b>Last Modified</b></td>
<td><b>Type</b></td>
</tr>
<?
// include class
require("Tar.php");

// set up object
$tar = new Archive_Tar($_FILES['file']['tmp_name']);

// read and print tar file contents
if (($arr = $tar−>listContent()) != 0)
{
foreach ($arr as $a)
{
echo "<tr>";
echo "<td>" . $a['filename'] . "</td>";
echo "<td>" . $a['size'] . "</td>";
echo "<td>" . $a['uid'] . "</td>";
echo "<td>" . $a['gid'] . "</td>";
echo "<td>" . $a['mode'] . "</td>";
echo "<td>" . $a['mtime'] . "</td>";
if ($a['typeflag'] == 5) { $type = "directory"; } else { $type = "file"; }
echo "<td>" . $type . "</td>";
echo "</tr>";
}
}
?>
</table>
<?
}
?>

</body>
</html>
```

Most of this should be familiar to you if you've ever dealt with HTTP file uploads in PHP. The script above is divided into two main parts, separated from each other by an "if" loop.

The first part of the script checks if the form has been submitted and, if not, displays a file selection box which the user can use to select a TAR file for upload. Note that since this POST transaction involves a file transfer, the encoding type of the form field must be set to "multipart/form−data".

Once a file has been selected and the form submitted, the second half of the script comes into play. This code first examines the $_FILES array to check if the uploaded file is of the correct type and − if it is − instantiates an object of the Archive_Tar class to read it. The Archive_Tar object's listContents() method is then used to obtain a list of the files within the archive, and display this information in a neatly−formatted HTML table. The columns in the table correspond to the keys of the array returned by listContents() − file name, size, ownership and permissions, and a flag indicating whether the item is a file or directory.

Here's an example of what the output looks like:



Filename	Size	UID	GID	Mode	Last Modified	Type
uudeview-0.5.18/	0	1000	1000	16877	1017742531	directory
uudeview-0.5.18/doc/	0	1000	1000	16877	1017742529	directory
uudeview-0.5.18/doc/Makefile	670	1000	1000	33188	834839345	file
uudeview-0.5.18/doc/README	918	1000	1000	33188	837900715	file
uudeview-0.5.18/doc/binhex.fig	3165	1000	1000	33188	834927297	file
uudeview-0.5.18/doc/library.ltx	101991	1000	1000	33188	1017741997	file
uudeview-0.5.18/doc/structure.fig	1024	1000	1000	33188	834090072	file
uudeview-0.5.18/doc/td-v1.c	252	1000	1000	33188	834090072	file
uudeview-0.5.18/doc/td-v2.c	977	1000	1000	33188	834090072	file

Note that the script above is illustrative only − allowing users to upload files to your Web application is an inherently dangerous process and one which opens up multiple security holes. The example is included here to demonstrate a possible application of the Archive_Tar class; if you plan to use it in a live environment, you should beef up the security checks within the code to ensure that nothing malicious can be uploaded by a user.

Backing Up...

Another very common application of Archive_Tar would be to package a set of files on a host into an archive (say, for backup purposes) and make it available for download through a Web browser. Possible users of such an application might be Web hosting services which want to allow customers to download backups of their data via their Web browsers, or intranet applications which allow administrators to download daily, weekly or monthly backups of the data and/or log files created during the intervening period.

Consider the following script, which asks the user to input a file path on the server, and then creates an compressed archive of the contents of that directory (and its subdirectories) for the user to download

```
<html>

<head>

</head>
```

```
<body bgcolor="white">
<?
// if form not submitted, display input field
if (!$_POST['submit'])
{
?>
<form action="<?=$_SERVER['PHP_SELF']; ?>" method="POST">
Select a directory to backup:
<input type="text" name="path">
<p>
<input type="Submit" name="submit" value="Start Backup">
</form>
<?
}
else
{
// check to make sure a valid file path has been provided
if (!file_exists($_POST['path']))
{
die("Invalid file path!");
}

// include class
require("Tar.php");

// set filename
$filename = "backup−" . date("Ymd", mktime()) . ".tgz";

// instantiate object
$tar = new Archive_Tar($filename, "gz");

// build archive
$tar−>create($_POST['path']) or die("Could not create archive!");

// provide download link
echo "Click <a href=" . $filename . ">here</a> to download backup"; } ?>
```

```
</body>
</html>
```

Again, this is fairly simple − the file path provided by the user is first checked for validity by the latter half of the script, and an Archive_Tar object is created to hold the compressed contents of the specified directory. The create() method is then called to actually perform the archive creation and compression, and the user is then provided with a link to download the compressed archive. Primitive, yes, but it works.

As before, the standard warning applies − if you plan to use this kind of tool in a live environment, you must put in lots of security checks to avoid malicious usage. At the very least, you should restrict access to the directories that can be backed up; failure to do so would allow a hacker to download important files (such as the system password file) and thereby gain access to sensitive user or system information.

... And Packing Up

And that's about it for this tutorial. Over the last few pages, I took you on a whirlwind tour of the Archive_Tar class, one of the more useful items in PEAR's collection of filesystem tools. I showed you the basics of creating a TAR file, adding files to it, viewing its contents, and extracting it to a specific directory.

I also showed you a couple of more advanced tricks, including compressing the archive with either GZIP or BZIP2 compression and using built−in class methods to alter the directory structure of the files within the archive. Finally, I wrapped things up with two examples of how the Archive_Tar class could be used in real Web applications, by building a simple TAR file viewer and a file backup utility.

In case you'd like to read more about the topics discussed in this article, you should consider bookmarking the following sites:

The official Archive_Tar Web page, at http://pear.php.net/package−info.php?pacid=24

The PEAR Web site, at http://pear.php.net/

The GNU TAR pages, at http://www.gnu.org/software/tar/tar.html

The GZIP pages, at http://www.gzip.org/

The BZIP2 pages, at http://www.digistar.com/bzip2/

Until next time...happy archiving!

Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!