RepoTools

Daniel Oberlin

Introduction

This document describes RepoTools, which is a set of command line programs that can be used to synchronize and monitor the integrity of large sets of file based data. Over time, it is possible for file data to become corrupted due to media, hardware, OS, or software failures. In some cases, data may become damaged silently. When this happens, simply copying or backing up files does not provide adequate protection. If the data integrity is not checked, corrupted files will be perpetuated over time and eventually the original data may become lost. RepoTools provides a means to detect file corruption early and to allow for the easy repair of problems when they arise.

RepoTools operates on repositories of data which are simply file system directory structures. A repository is a directory of files and subdirectories which are associated with a manifest file that contains information about each file including a hash of the file contents and the last-modified time attribute of the file. As files are added, changed, or removed from the repository the manifest file can be easily updated so that it stays in sync with the intended content of the repository. At any time, a repository can be checked against its manifest to determine if any of the files are missing or corrupted.

Typically, the manifest file is stored just underneath the top level directory of the repository. Because of this, the process of duplicating a repository is as simple as copying its directory structure to another location while taking care to preserve the last-modified time attributes of the files. Various tools such as Robocopy can assist with the task of copying a repository. Alternatively, RepoTools can be used to accomplish this quite easily and with some added benefits.

Once a repository has been copied, the integrity of the copy can be checked to make sure that no data was lost or corrupted during the copying process.

RepoTools provides facilities to keep a repository in sync with other repository copies that may exist. At any time, it is possible to diff two repositories and to report on any files that have changed content or any files that have been added to or removed from either repository. Various modes exist for synchronizing two repositories with each other. In this way, it is possible to maintain reliable backup copies or mirrors of a repository. If a repository is found to be damaged during an integrity check, then another copy may be used to efficiently repair or replace the original as needed.

Presently, there are two tools to support these operations:

1. RepoTool (rt) – this tool allows for the creation and maintenance of a single individual repository. This includes checking the status of a repository, validating its integrity, and updating the repository manifest when intentional changes have been made to the files in the repository.

2. RepoSync (rs) – this tool provides the ability to duplicate an existing repository, as well as functionality for the comparison and synchronization between a pair of repositories. This includes diffing two repositories and migrating updates from one to the other.

These tools were implemented using the Microsoft .NET framework. They are also usable on Mac OSX and other platforms which support the Mono framework. These tools were loosely inspired by the Git source control management system, but this software serves a different purpose and does not provide facilities for revision control.

RepoTool (rt)

Prior to using these tools, they should be installed by copying the executables and associated .DLL files to a convenient location and setting the system PATH (or OS equivalent) so that they can be used from the command line.

While learning the usage of these tools it will be helpful to prepare a testing directory of data that can be worked on without risk. A directory structure containing a set of small to medium sized files is ideal for this purpose. The directory structure may contain subdirectories and it may be located anywhere that access is permitted from within the file system. Note that locating a repository at the root directory of a file system may not be ideal because of special files or directories which might be restricted from normal user access.

Once RepoTools is installed, a repository can be created from the testing data by issuing the `rt create` command from the system shell or command prompt while inside the top level or root directory of the file set. By default, the manifest file is named `.repositoryManifest`, and it is placed at the root level of the repository when it is created. When a manifest file is first created, its file list is empty and not up to date with the contents of the repository. This can be confirmed by issuing the `rt status` command, again from the root level of the repository. The status command will report a number of new files which aren't being tracked in the manifest. To complete the creation of the repository, issue the `rt update` command which will bring the manifest up to date with the contents of the repository directory. Checking the status a second time after the update will show that there are no new files and that the manifest is up to date.

The `rt status` performs a quick check by comparing the last-modified time attributes and file sizes against their corresponding entries in the manifest. In order to check the actual data integrity of the files, issue the `rt validate` command which will compute the hash of each file in the repository and check it against the hash stored in the manifest. This operation is more computationally intensive and it will take longer to execute than the status operation.

Issuing `rt help`, or simply `rt` will show the syntax and all of the commands and options that are available with the RepoTool command.

In order to see metadata and other information about the repository, issue the `rt info` command. The GUID listed in the information identifies the repository. Copies or mirrors of this repository will share the same GUID.

The final section of information reported by the info command is a list of patterns which define file pathnames which will be ignored and not processed as part of the repository. The file and directory structure information contained within the manifest is represented internally with UNIX-style pathnames. This is true even while running under Windows, and it is intended to provide cross-platform compatibility between repositories that are synchronized between different operating systems. The list of ignore patterns contains UNIX-style regular expressions which are intended to match these UNIX-style pathnames as they are stored internally. By default, the following regular expression is included in the ignore list:

```
^\./\.repositoryManifest$
```

This rather cryptic looking string could use some explanation. The "^" and "$" characters anchor the pattern to the beginning and ending of the string respectively. The "\" character escapes special pattern matching characters, like the "." character. The result is that this pattern will match the path "`./.repositoryManifest`" exactly. The beginning "`./`" in the path indicates that the path is relative to the root of the repository, so this pattern will ignore the manifest file which is located in the root directory of the repository, which is appropriate. Other entries may be added to this list in order to ignore junk or temporary files which should not be tracked along with the actual repository contents. For example, Windows users might issue the following command:

```
rt edit -ignore /Thumbs\.db$
```

This edits the repository manifest and adds an ignore pattern for file pathnames that end with "`/Thumbs.db`". These are thumbnail image cache files that are sometimes added to directories by the Windows operating system.

In cases where many repositories are being created with large numbers of ignore patterns, it may be inconvenient to add the ignore patterns to each new repository manifest. In this case, it is possible to use a "manifest prototype" which will be used as a template for creating new manifests. Simply create an empty manifest, add the ignore paths to that manifest, and copy the manifest into the same directory that contains the repository tool executable binaries. Then rename the manifest from "`.repositoryManifest`" to "`.manifestPrototype`", and all newly created manifests will be automatically populated with the ignore patterns and other metadata (except GUID, name and description) listed in the manifest prototype.

The edit command can also be used in conjunction with the `-name` and `-description` options in order to provide name and description metadata for the contents of the repository. These are optional, but when used they should be added when the repository is fist created so that they are propagated consistently as the repository is copied.

Rename a file in the repository and issue the `rt status` command. The output should indicate that one file has been removed and one file has been added. In some cases, it may be useful to identify when files have simply been moved or renamed. Using the `–trackMoves` option can provide this information at the expense of taking the time to compute hash values for files that are not listed in the manifest. To see this in action, issue the `rt status –trackMoves` command. The output should reflect that one file has been moved. Assuming that the move was intentional, it makes sense to use the `rt update` command so that the repository is brought up to date with this change.

RepoSync (rs)

In most cases, a repository will not exist in isolation but rather in conjunction with one or more copies which are used for redundancy. Copying a repository is as easy as copying the repository directory structure to a new location while preserving the last-modified time attributes of the files. RepoSync can be used to do this with two simple steps. As an example, to make a duplicate copy of a repository, one could use the following two commands:

```
rs seed pathToOriginalRepository pathToNewRepositoryCopy
```

```
rs update pathToOriginalRepository pathToNewRepositoryCopy
```

For these commands, the actual operating system pathnames to the repository root directories would be given as the last two arguments. Be sure to use backslashes when running on Windows.

The first command creates a root directory for the new repository copy (if needed) and places an empty manifest into the directory. The new empty manifest shares the same GUID as the original repository, so the two repositories can by synchronized in various ways. The new manifest also shares other settings and metadata of the original manifest. It only lacks the file listings because the files from the original repository do not yet exist yet in the new repository copy.

The second command updates the new repository with all of the files from the original repository. The update command is one of several synchronization modes that can be used with RepoSync. See the help section for details about the various synchronization modes and how they operate.

This method of copying a repository has the added benefit that it will not copy any junk files which are specified as being ignored by the manifest. Other methods of copying might also duplicate these junk files and not result in a “clean” copy.

Now that the repository has been duplicated, it makes sense to check the integrity of the copy. From the command line, change directories to the new repository copy and issue the `rt validate` command to confirm that the repository was copied without error.

The RepoSync tool manages comparisons and updates between related repositories. Compare the status of the repository copy with that of the original by issuing the following command:

```
rs diff pathToOriginalRepository pathToNewRepositoryCopy
```

The result of the command should indicate that there are no differences – that is, the repositories are in sync with each other. The diff command produced its results by comparing the manifests of the two repositories and it assumed that the manifests were up to date with the contents of the repositories. In general, a repository manifest should be updated as soon as intentional changes are made to the repository – including file modifications, file additions and file removals.

In order to see `rs diff` in a useful context, make a few changes to each of the two repositories. In the repository copy, add a new file. In the original repository, delete a file. In the repository copy, make a change to one of the files – this could be editing a text file or replacing a binary file with new data.

From the command line, update the manifests of the original repository and the repository copy to reflect these changes by using the `rt update` command from the root directory of each of the two repositories. At this point, repeating the `rs diff` command should reveal the differences between the two repositories.

RepoSync provides three major functionalities: update, sync, and mirror. The syntax and options of the tool are described in detail by using the `rs help` command. Briefly:

- **update:** propagate any changes from the *source* repository to the *destination* repository, but don't change anything in the *source* repository. No extraneous files are removed from *dest*.

- **sync:** update *source* and *dest* with anything new from either side. Use the file system last modified dates to ensure that both sides have the most recent version of any file.

- **mirror:** make *dest* match *source*. Replacing or removing files in *dest* as necessary. No changes are made to *source*.

Using Encryption

In some cases, it may be desirable to maintain a repository copy in the care of others who provide remote network access to the data. This adds some protection against the possibility of data loss due to catastrophic events. For data that is personal in nature, RepoTools provides encryption that can be used to protect it and maintain privacy from others who may have access to it.

Although there are many tools that can be used for data encryption, the goal of RepoTools encryption is to maintain control of the encryption process by the owner on their computer while allowing the data to be validated (possibly by others) on the remote computer. This separation is important for two reasons. Most importantly, the encryption key should never be entered on or stored by the remote computer, or the data security could be compromised. And with regards to efficiency, the remote data should not

need to be decrypted in order to be validated – otherwise, the entire repository would need to be moved across the network to a secure computer prior to validation.

An encrypted repository is always created and maintained as a copy of another repository. To make an encrypted copy of a repository, use the following two commands:

```
rs seed pathToOriginalRepo pathToNewEncryptedRepo -cryptDest
```

```
rs update pathToOriginalRepo pathToNewEncryptedRepo -cryptDest
```

After each command, you will be prompted to enter the encryption key for the destination repository. The key is set when the repository is seeded, and the same key must be used for all subsequent access to the repository. The usual guidelines for password selection should be followed when choosing an encryption key.

Inspecting the contents of an encrypted repository reveals that the original subdirectory structure is collapsed and the original filenames have been replaced with uninterpretable hashed information. The file contents themselves are safely encrypted, and there is one additional file named "`.repositoryManifest.outer`".

The extra "outer" manifest file in the repository is actually an encrypted manifest that lists the original filenames, directory structure, manifest GUID, and other metadata that must be synchronized with other repository copies. None of this revealing metadata exists in the primary unencrypted manifest file.

All of the commands and options for RepoSync can also be used with encrypted repositories. It is only necessary to supply the `-cryptSource` or `-cryptDest` options as appropriate to indicate that either or both of the repositories are encrypted and requires a key. To recover an unencrypted repository from an encrypted repository, simply reverse the process of creation listed above and use the `-cryptSource` option instead to specify that the source repository is encrypted.

An encrypted repository is validated in exactly the same way as an unencrypted repository. No encryption key is necessary for validation because the encrypted repository is itself a valid repository which contains hash information from the encrypted file contents.

RepoSync uses the AES256 algorithm for its encryption. In order to protect against certain cryptographic attacks, a different *binary* key is used each time new file data is encrypted. Each binary key is generated by combining the user's secure *password key* with other data which is known as *salt*. The salt data does not need to be secure, but it should be essentially random, of sufficient length, and different for each file that is encrypted. For the encrypted repository data files, the salt data is derived from a hash of the file contents. This ensures that when the file is updated and re-encrypted, the salt data and resulting binary key will always be unique. For the encrypted manifest file, the salt data is derived from the GUID of the *unencrypted* manifest file. This GUID is changed each time the encrypted manifest is updated, again ensuring that the manifest will never be encrypted twice with the same binary key.