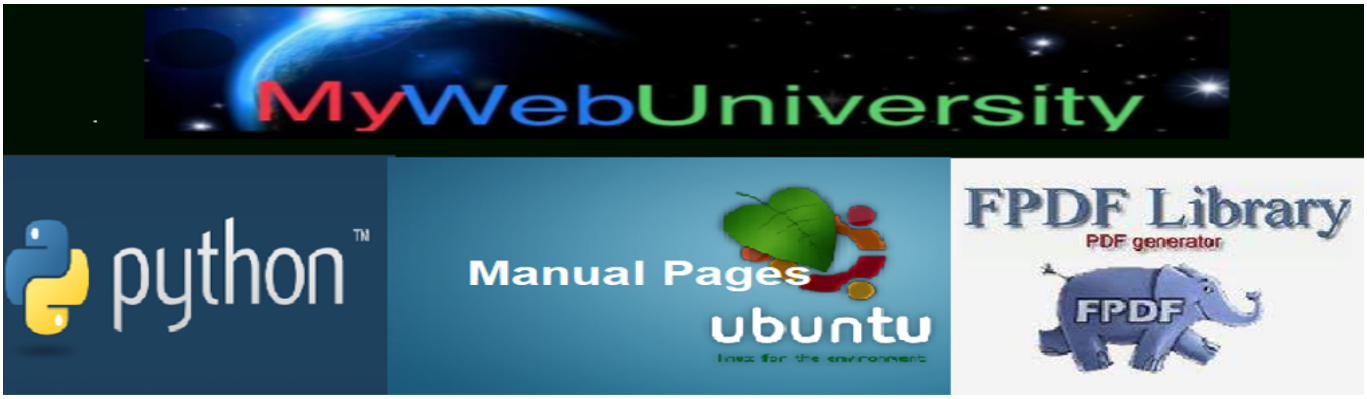

Rocky Enterprise Linux 9.2 Manual Pages on command 'git-bundle.1'

$ man git-bundle.1

GIT-BUNDLE(1) Git Manual GIT-BUNDLE(1)

NAME

git-bundle - Move objects and refs by archive

SYNOPSIS

git bundle create [-q | --quiet | --progress | --all-progress] [--all-progress-implied]
[--version=<version>] <file> <git-rev-list-args>

git bundle verify [-q | --quiet] <file>

git bundle list-heads <file> [<refname>...]

git bundle unbundle [--progress] <file> [<refname>...]

DESCRIPTION

Create, unpack, and manipulate "bundle" files. Bundles are used for the "offline" transfer of Git objects without an active "server" sitting on the other side of the network connection.

They can be used to create both incremental and full backups of a repository, and to relay the state of the references in one repository to another.

Git commands that fetch or otherwise "read" via protocols such as ssh:// and https:// can also operate on bundle files. It is possible git-clone(1) a new repository from a bundle, to use git-fetch(1) to fetch from one, and to list the references contained within it with git-ls-remote(1). There?s no corresponding "write" support, i.e.a git push into a bundle is not supported.

See the "EXAMPLES" section below for examples of how to use bundles.

BUNDLE FORMAT

Bundles are .pack files (see git-pack-objects(1)) with a header indicating what references

are contained within the bundle.

Like the the packed archive format itself bundles can either be self-contained, or be created using exclusions. See the "OBJECT PREREQUISITES" section below.

Bundles created using revision exclusions are "thin packs" created using the --thin option to git-pack-objects(1), and unbundled using the --fix-thin option to git-index-pack(1).

There is no option to create a "thick pack" when using revision exclusions, and users should not be concerned about the difference. By using "thin packs", bundles created using exclusions are smaller in size. That they?re "thin" under the hood is merely noted here as a curiosity, and as a reference to other documentation.

See the bundle-format documentation[1] for more details and the discussion of "thin pack" in the pack format documentation[2] for further details.

OPTIONS

create [options] <file> <git-rev-list-args>

Used to create a bundle named file. This requires the <git-rev-list-args> arguments to define the bundle contents. options contains the options specific to the git bundle create subcommand.

verify <file>

Used to check that a bundle file is valid and will apply cleanly to the current repository. This includes checks on the bundle format itself as well as checking that the prerequisite commits exist and are fully linked in the current repository. git bundle prints a list of missing commits, if any, and exits with a non-zero status.

list-heads <file>

Lists the references defined in the bundle. If followed by a list of references, only references matching those given are printed out.

unbundle <file>

Passes the objects in the bundle to git index-pack for storage in the repository, then prints the names of all defined references. If a list of references is given, only references matching those in the list are printed. This command is really plumbing, intended to be called only by git fetch.

<git-rev-list-args>

A list of arguments, acceptable to git rev-parse and git rev-list (and containing a named ref, see SPECIFYING REFERENCES below), that specifies the specific objects and references to transport. For example, master~10..master causes the current master

reference to be packaged along with all objects added since its 10th ancestor commit. There is no explicit limit to the number of references and objects that may be packaged.

[<refname>...]

A list of references used to limit the references reported as available. This is principally of use to git fetch, which expects to receive only those references asked for and not necessarily everything in the pack (in this case, git bundle acts like git fetch-pack).

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--all-progress

When --stdout is specified then progress report is displayed during the object count and compression phases but inhibited during the write-out phase. The reason is that in some cases the output stream is directly linked to another command which may wish to display progress status of its own as it processes incoming pack data. This flag is like --progress except that it forces progress report for the write-out phase as well even if --stdout is used.

--all-progress-implied

This is used to imply --all-progress whenever progress display is activated. Unlike --all-progress this flag doesn?t actually force any progress display by itself.

--version=<version>

Specify the bundle version. Version 2 is the older format and can only be used with SHA-1 repositories; the newer version 3 contains capabilities that permit extensions. The default is the oldest supported format, based on the hash algorithm in use.

-q, --quiet

This flag makes the command not to report its progress on the standard error stream.

SPECIFYING REFERENCES

Revisions must be accompanied by reference names to be packaged in a bundle.

More than one reference may be packaged, and more than one set of prerequisite objects can be specified. The objects packaged are those not contained in the union of the prerequisites.

The git bundle create command resolves the reference names for you using the same rules as git rev-parse --abbrev-ref=loose. Each prerequisite can be specified explicitly (e.g. ^master~10), or implicitly (e.g. master~10..master, --since=10.days.ago master).

All of these simple cases are OK (assuming we have a "master" and "next" branch):

```
$ git bundle create master.bundle master
$ echo master | git bundle create master.bundle --stdin
$ git bundle create master-and-next.bundle master next
$ (echo master; echo next) | git bundle create master-and-next.bundle --stdin
```

And so are these (and the same but omitted --stdin examples):

```
$ git bundle create recent-master.bundle master~10..master
$ git bundle create recent-updates.bundle master~10..master next~5..next
```

A revision name or a range whose right-hand-side cannot be resolved to a reference is not accepted:

```
$ git bundle create HEAD.bundle $(git rev-parse HEAD)
fatal: Refusing to create empty bundle.
$ git bundle create master-yesterday.bundle master~10..master~5
fatal: Refusing to create empty bundle.
```

OBJECT PREREQUISITES

When creating bundles it is possible to create a self-contained bundle that can be unbundled in a repository with no common history, as well as providing negative revisions to exclude objects needed in the earlier parts of the history.

Feeding a revision such as new to git bundle create will create a bundle file that contains all the objects reachable from the revision new. That bundle can be unbundled in any repository to obtain a full history that leads to the revision new:

```
$ git bundle create full.bundle new
```

A revision range such as old..new will produce a bundle file that will require the revision old (and any objects reachable from it) to exist for the bundle to be "unbundle"-able:

```
$ git bundle create full.bundle old..new
```

A self-contained bundle without any prerequisites can be extracted into anywhere, even into an empty repository, or be cloned from (i.e., new, but not old..new).

It is okay to err on the side of caution, causing the bundle file to contain objects already in the destination, as these are ignored when unpacking at the destination.

If you want to match git clone --mirror, which would include your refs such as refs/remotes/*, use --all. If you want to provide the same set of refs that a clone directly from the source repository would get, use --branches --tags for the <git-rev-list-args>.

The git bundle verify command can be used to check whether your recipient repository has the required prerequisite commits for a bundle.

EXAMPLES

Assume you want to transfer the history from a repository R1 on machine A to another repository R2 on machine B. For whatever reason, direct connection between A and B is not allowed, but we can move data from A to B via some mechanism (CD, email, etc.). We want to update R2 with development made on the branch master in R1.

To bootstrap the process, you can first create a bundle that does not have any prerequisites. You can use a tag to remember up to what commit you last processed, in order to make it easy to later update the other repository with an incremental bundle:

```
machineA$ cd R1
machineA$ git bundle create file.bundle master
machineA$ git tag -f lastR2bundle master
```

Then you transfer file.bundle to the target machine B. Because this bundle does not require any existing object to be extracted, you can create a new repository on machine B by cloning from it:

```
machineB$ git clone -b master /home/me/tmp/file.bundle R2
```

This will define a remote called "origin" in the resulting repository that lets you fetch and pull from the bundle. The $GIT_DIR/config file in R2 will have an entry like this:

```
[remote "origin"]
    url = /home/me/tmp/file.bundle
    fetch = refs/heads/*:refs/remotes/origin/*
```

To update the resulting mine.git repository, you can fetch or pull after replacing the bundle stored at /home/me/tmp/file.bundle with incremental updates.

After working some more in the original repository, you can create an incremental bundle to update the other repository:

```
machineA$ cd R1
machineA$ git bundle create file.bundle lastR2bundle..master
machineA$ git tag -f lastR2bundle master
```

You then transfer the bundle to the other machine to replace /home/me/tmp/file.bundle, and pull from it.

```
machineB$ cd R2
machineB$ git pull
```

If you know up to what commit the intended recipient repository should have the necessary objects, you can use that knowledge to specify the prerequisites, giving a cut-off point to limit the revisions and objects that go in the resulting bundle. The previous example used the lastR2bundle tag for this purpose, but you can use any other options that you would give to the git-log(1) command. Here are more examples:

You can use a tag that is present in both:

```
$ git bundle create mybundle v1.0.0..master
```

You can use a prerequisite based on time:

```
$ git bundle create mybundle --since=10.days master
```

You can use the number of commits:

```
$ git bundle create mybundle -10 master
```

You can run git-bundle verify to see if you can extract from a bundle that was created with a prerequisite:

```
$ git bundle verify mybundle
```

This will list what commits you must have in order to extract from the bundle and will error out if you do not have them.

A bundle from a recipient repository?s point of view is just like a regular repository which it fetches or pulls from. You can, for example, map references when fetching:

```
$ git fetch mybundle master:localRef
```

You can also see what references it offers:

```
$ git ls-remote mybundle
```

GIT

Part of the git(1) suite

NOTES

1. the bundle-format documentation

   file:///usr/share/doc/git/html/technical/bundle-format.html

2. the pack format documentation

   file:///usr/share/doc/git/html/technical/pack-format.html