filename: rclone-commands-and-usage-multif-20250620.txt
https://docs.oracle.com/en/learn/migr-ocistorage-rclone-p2/index.html#overview-of-rclone-and-basic-terms

Move Data into OCI Cloud Storage Services using Rclone

Introduction

This is tutorial 2 of a four tutorial series that shows you various ways to migrate data into Oracle Cloud Infrastructure (OCI) cloud storage services. The series is set up so you can review Tutorial 1: Use Migration Tools to Move Data into OCI Cloud Storage Services to get a broad understanding of the various tools and then proceed to the related tutorial(s) or documents relevant to your migration needs. This tutorial will focus on using Rclone to migrate data into OCI cloud storage services.

OCI provides customers with high-performance computing and low-cost cloud storage options. Through on-demand local, object, file, block, and archive storage, Oracle addresses key storage workload requirements and use cases.

OCI cloud storage services offers fast, secure, and durable cloud storage options for all your enterprise needs. Starting with the high performance options such as OCI File Storage with Lustre and OCI Block Volumes service; fully managed exabyte scale filesystems from OCI File Storage service with high performance mount targets; to highly durable and scalable OCI Object Storage. Our solutions can meet your demands, ranging from performance intensive applications such as AI/ML workloads, to exabyte-scale data lakes.

Rclone is an open source, command-line utility to migrate data to the cloud, or between cloud storage vendors. Rclone can be used to do one-time migration as well as periodical synchronization between source and destination storage. Rclone can migrate data to and from object storage, file storage, mounted drives, and between 70 supported storage types. OCI Object Storage is natively supported as a Rclone backend provider. Rclone processes can be scaled up and scaled out to increase the transfer performance using parameter options.

Determine the amount of data that needs to be migrated, and the downtime available to cut-over to the new OCI storage platform. Batch migrations are a good choice to break down the migration into manageable increments. Batch migrations will enable you to schedule downtime for specific applications across different windows. Some customers have the flexibility to do a one-time migration over a scheduled maintenance window over 2-4 days. OCI FastConnect can be used to create a dedicated, private connection between OCI and your environment, with port speeds from 1G to 400G to speed up the data transfer process. OCI FastConnect can be integrated with partner solutions such as Megaport and ConsoleConnect to create a private connection to your data center or cloud-to-cloud interconnection to move data more directly from another cloud vendor into OCI cloud storage service. For more information, see FastConnect integration with Megaport Cloud Router.

Audience

DevOps engineers, developers, OCI cloud storage administrators and users, IT managers, OCI power users, and application administrators.

Objective

Learn how to use Rclone to copy and synchronize data into OCI cloud storage services.

* Use Rclone for migrating filesystem data (local, NAS, cloud hosted) into OCI Object Storage.
* Migrate data from another cloud object or blob storage into OCI Object Storage.
* Use Rclone on Oracle Cloud Infrastructure Kubernetes Engine (OKE) to migrate data from OCI File Storage to OCI Object Storage.

Prerequisites

* An OCI account.
* Virtual machine (VM) instance on OCI to deploy the migration tools or a system where you can deploy and use migration tools.
* Oracle Cloud Infrastructure Command Line Interface (OCI CLI) installed with a working config file in your home directory in a subdirectory called .oci. For more information, see Setting up the Configuration File.
* Access to an OCI Object Storage bucket.
* User permissions in OCI to use OCI Object Storage, have access to manage buckets and objects or manage object-family for at least 1 bucket or compartment. For more information, see Common Policies and Policy Reference.
* User permission to create, export, and mount OCI File Storage, or access to an OCI File Storage mount target that is already mounted on a VM, or another NFS mount or local file system to use for copying data to and from. For more information, see Manage File Storage Policy.
* Familiarity with using a terminal/shell interface on Mac OS, Linux, Berkeley Software Distribution (BSD), or Windows PowerShell, command prompt, or bash.
* Familiarity with installing software on a Linux system and have some experience or understanding of Kubernetes.
* Basic knowledge of Oracle Cloud Infrastructure Identity and Access Management (OCI IAM) and working with dynamic groups with a Ubuntu host in a dynamic group. For more information, see Managing Dynamic Groups.
* Review Migration Essentials for Moving Data into OCI Cloud Storage to install Rclone and other migration tools.
* To know the migration tools we can use, see Tutorial 1: Use Migration Tools to Move Data into OCI Cloud Storage Services.

Overview of Rclone and Basic Terms

Rclone is a helpful migration tool because of the many protocols and cloud providers it supports and ease of configuration. It is a good general purpose migration tool for any type of data set. Rclone works particularly well for data sets that can be split up into batches to scale-out across nodes for faster data transfer.

Rclone can be used to migrate:

* File system data (OCI File Storage, OCI Block Storage, OCI File Storage with Lustre, on-prem file system, and on-prem NFS) to other file system storage types and to/from object storage (including OCI Object Storage).
* Object storage from supported cloud providers to and from OCI Object Storage.

Rclone Commands and Flags:

* Understand Rclone Performance
  Rclone is a good general-purpose tool for syncing or copying files between filesystem data, other cloud providers, and OCI cloud storage services. The performance will depend on how much you can scale-up and scale-out. We recommend running various test on your migration systems with a sample migration set to determine when you hit the thresholds of your network bandwidth.
  For example, your source bucket has 10 folders/prefixes, each having about 1TB. You could split the migration across 2 VMs of large CPU/RAM capacity and trigger multiple Rclone copy processes in parallel from the two VMs. Depending on each folder's topology and the compute capacity, the Rclone parameters can be adjusted to improve transfer speed.
  You could start with running the following commands on 2 VM's and then adjust the transfer count and checker count until you saturate the NIC on each VM.

```
$> rclone copy --progress --transfers 10 --checkers 8 --no-check-dest aws_s3_virgina:/source_bucket_name/folder
1 \
iad_oss_native:/destination_bucket_name/folder1

$> rclone copy --progress --transfers 50 --checkers 8 --no-check-dest aws_s3_virgina:/source_bucket_name/folder
2 \
iad_oss_native:/destination_bucket_name/folder2
```

  + Systems or VM instances with more CPU, memory, and network bandwidth can run more file transfers and checkers in parallel. Scaling up to systems with more resources will allow faster performance.
  + If your data can be split up into various batches based on structure, you can also run Rclone on multiple systems or VM instances to scale-out.

  We recommend scaling up and out to improve Rclone performance. Our testing included 2 VM's to run Rclone transfers in parallel to scale out. If you have a larger data set, you may want to use up to 4 machines or even use Bare Metal (BM) instances.

* Rclone Copy and Sync Commands
  + Rclone copy command copies source files or objects to the destination. It will skip files that are identical on source and destination, testing by size and modification time or md5sum. The copy command does not delete files from the destination.
  + Rclone sync command synchronizes the source with the destination, also skipping identical files. The destination will be modified to match the source, which means files not matching the source will be deleted.

Note: Be careful using sync, and use it only when you want the destination to look exactly like the source. Use the copy command when you just want to copy new files to the destination.

* Use Right Rclone Command Line Flags
  There are several Rclone command line flags that can be used with Rclone that affect how fast data migration occurs. It is important to understand how some of these flags work to get the best data transfer throughput.

  + --no-traverse: This only works with the copy command, do not travel through the destination file system. This flag saves time because it will not conduct lists on the destination to determine which files to copy over. It will check files one at a time to determine if it needs to be copied. One-by-one may seem slow but can be faster when you have a very small number of files/objects to copy over to a destination with many files already present.

  + --no-check-dest: This only works with the copy command and will not check or list the destination files to determine what needs to be copied or moved which minimizes API calls. Files are always transferred. Use this command when you know you want everything on the source copied regardless of what is on the destination or if you know the destination is empty.

Note: Use no-traverse or no-check-dest commands, many users put both on the command line which is not necessary.

  - If your target is empty or you want all files copied from the source to the destination no matter what, use no-check-dest.
  - When you have a few very large files that need to be migrated, use no-traverse which will check each file to see if it is current on the destination before copying it to the source; this could save on list API calls and the amount of data copied to the target.

  + --ignore-checksum: This will really speed up the transfer, however Rclone will not check for data corruption during transfer.

  + --oos-disable-checksum: Do not store MD5 checksum with object metadata. Rclone calculates the MD5 checksum of the data before uploading and adds it to the object metadata, which is great for data integrity, however it causes delays before large files begin the upload process.

  + --transfers <int>: Number of file transfers to run in parallel (default 4). Scale this number up based on the size of the system where you are running rclone, you can do test runs and increase the integer until you reach max transfer speed for your host. We really recommend testing and raising this number until you get acceptable performance, we have seen customers raise this number between 64-3000 to get desired performance.

  + --checkers <int>: Number of checks to run in parallel (default 8). Amount of file checkers to run in parallel, be cautious as it can drain server health and cause problems on the destination. If you have a system with very large memory, bump this number up by increments

of 2. The maximum number we tested this setting with good results in the test environment is 64, typically 8-10 is sufficient. Checkers can be anywhere from 25-50% of the transfer number; when the transfer number is higher this number tends to be closer to 25%.

Note: When scaling out with multiple hosts running Rclone transfers and checkers you may hit a 429 "TooManyRequests" error, should this happen start by lowering the amount of checkers in increments of 2 until you reach 10. If lowering the checkers is not enough, you will also need to lower the number of transfers.

+ --progress: This will show progress during transfer.

+ --fast-list: Use recursive list if available; uses more memory but fewer transactions/API calls. This is a good option to use when you have a moderate number of files in nested directories. Do not use with no-traverse or no-check-dest since they are contrary flags. Can be used with the copy or sync command.

+ --oos-no-check-bucket: Use this when you know the bucket exists, it reduces the number of transactions Rclone conducts, it sets Rclone to assume the bucket exists and to start moving data into it.

+ --oos-upload-cutoff: Files larger than this size will be uploaded in chunks, the default is 200MiB.

+ --oos-chunk-size: When uploading files larger than the upload cutoff setting or files with unknown size they will be uploaded as multipart uploads using this chunk size. Rclone will automatically increase chunk size when uploading a large file of known size to stay below the 10,000 chunks limit. The default is 5MiB.

+ --oos-upload-concurrency <int>: This is used for multipart uploads and is the number of chunks uploaded concurrently. If you are uploading small numbers of large files over high-speed links and these uploads do not fully utilize your bandwidth, then increasing this may help to speed up the transfers. The default is 8, if this is not utilizing bandwidth increase slowly to improve bandwidth usage.

Note: Multi-part uploads will use extra memory when using the parameters: --transfers <int>, --oos-upload-concurrency <int> and --oos-chunk-size. Single part uploads do not use extra memory. When setting these parameters consider your network latency, the more latency, the more likely single part uploads will be faster.

* Rclone Configuration File Example for OCI Object Storage

```
[oci]
type = oracleobjectstorage
namespace = xxxxxxxxxxx
compartment = ocid1.compartment.oc1..xxxxxxxxx
region = us-ashburn-1
provider = user_principal_auth
config_file = /.oci/config
config_profile = Default
```

* Basic Rclone Command Format

```
rclone <flags> <command> <source> <dest>
```

+ Example of running Rclone copy from a local filesystem source or OCI File Storage source to OCI Object Storage destination.

```
$> rclone copy /src/path oci:bucket-name
```

+ Example of running Rclone copy from OCI Object Storage source to a local filesystem or OCI File Storage destination.

```
$> rclone copy oci:bucket-name /src/path
```

+ Example of running Rclone copy from an S3 source to a OCI Object Storage destination.

```
$> rclone copy s3:s3-bucket-name oci:bucket-name
```

Note: When migrating from AWS and using server-side encryption with KMS, make sure rclone is configured with server_side_encryption = aws:kms to avoid checksum errors. For more information, see Rclone S3 KMS and Rclone S3 configuration

Note: Format of the sync command will be basically the same, simply replace copy with sync.

Rclone Usage Examples

* Example 1: Use Rclone to migrate a small number of small files copied into a destination already containing data with a high file or object count.

```
$> rclone --progress  --transfers 16  --oos-no-check-bucket --checkers 8  --no-traverse copy <source> <dest>
```

* Example 2: Rclone with fewer large files with multi-part uploads.

```
$> rclone --progress  --oos-no-check-bucket --fast-list --no-traverse --transfers 8 --oos-chunk-size 10M \
--oos-upload-concurrency 10 --checkers 10  copy <source> <dest>
```

Note: These are starting points for the options --transfers , --oos-chunk-size, --oos-upload-concurrency, and --checkers, you will need to adjust them based on your file/object size, memory and resources available on the systems you are using for migrating data. Adjust them up until you get sufficient bandwidth usage to migrate your data optimally. If your system is very small, you may need to adjust these numbers down to conserve resources.

* Example: 3 Use Rclone for scale-out run on 3 BM machines with 100 Gbps NIC, data set mixed size with multi-part uploads with petabytes of data, bucket not empty, OCI File Storage service to OCI Object Storage service.

```
$> rclone --progress --stats-one-line --max-stats-groups 10 --fast-list --oos-no-check-bucket \
--oos-upload-cutoff 10M --transfers 64 --checkers 32 --oos-chunk-size 512Mi --oos-upload-concurrency 12 \
--oos-disable-checksum --oos-attempt-resume-upload --oos-leave-parts-on-error \
--no-check-dest /src/path oci:bucket
```

Additional flags used:

+ --stats-one-line: Make the stats fit on one line.

+ --max-stats-group: Maximum number of stats groups to keep in memory, on max oldest is discarded (default 1000).

+ --oos-attempt-resume-upload: Attempt to resume previously started multi-part upload for the object.

+ --oos-leave-parts-on-error: Avoid calling abort upload on a failure, leaving all successfully uploaded parts for manual recovery.

Migrate a Large Number Files using Rclone

Rclone syncs on a directory-by-directory basis. If you are migrating tens of millions of files/objects, it is important to make sure the directories/prefixes are divided up into around 10,000 files/objects or lower per directory. This is to prevent Rclone from using too much memory and then crashing. Many customers with a high count (100's of millions or more) of small files often run into this issue. If all your files are in a single directory, divide them up first.

1. Run the following command to get a list of files in the source.

```
$> rclone lsf --files-only -R src:bucket | sort > src
```

2. Break up the file into chunks of 1,000-10,000 lines, using split. The following split command will divide up the files into chunks of 1,000 and then put them in files named src_## such as src_00.

```
$> split -l 1000 --numeric-suffixes src src_
```

3. Distribute the files to multiple VM instances to scale out the data transfer. Each Rclone command should look like:

```
rclone --progress --oos-no-check-bucket --no-traverse --transfers 500 copy remote1:source-bucket \
remote2:dest-bucket --files-from src_00
```

Alternatively, a simple for loop can be used to iterate through the file lists generated from the split command. During testing with ~270,000 files in a single bucket, we saw copy times improve 40x, your mileage may vary.

Note: Splitting up the files by directory structure or using the split utility is an important way to optimize transfers.

Use Rclone, OKE and fpart together for Moving Data from File Systems to OCI Object Storage

Multiple Kubernetes pods can be used to scale out data transfer between file systems and object storage. Parallelization speeds up data transfers to storage systems that are relatively high latency and are high throughput. The approach combining Rclone, OKE and fpart partitions directory structures into multiple chunks and runs the data transfer in parallel on containers either on the same compute node or across multiple nodes. Running across multiple nodes aggregates the network throughput and compute power of each node.

* Filesystem partitioner (Fpart) is a tool that can be used to partition the directory structure. It can call tools such rsync, tar, and Rclone with a file system partition to run in parallel, and independent of each other. We will use fpart with Rclone.
* fpsync is a wrapper script that uses fpart to runs the transfer tools (rsync, Rclone) in parallel. The fpsync command is run from an fpsync operator host. The fpsync tool also has options to use separate worker nodes. The modified fpsync supports Rclone and also Kubernetes pods.
* kubectl manages Kubernetes jobs.

Follow the steps:

1. Identify a host that will be your fpsync operator host that has access to the migration source data and Rclone is installed.

2. Run the following command to install kubectl.

```
$> curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubec
tl"
$> chmod 755 kubectl
$> cp -a kubectl /usr/bin
```

3. Create an OCI IAM policy for the fpsync operator host to manage the OKE cluster. The following policy can be used for this purpose. A more granular permission can be configured to achieve the bare minimum requirement to control the pods.

```
Allow dynamic-group fpsync-host to manage cluster-family in compartment storage
```

4. Setup the kubeconfig file to have access to the OKE cluster. For more information, see Setting Up Local Access to Clusters.

5. Install and patch fpart and fpsync. The fpsync patch is required to run Rclone or rsync in parallel to scale out the data transfer. The fpsync that comes with the fpart package does not support Rclone or Kubernetes pods, a patch is needed to support these tools. Run the following command to install on Ubuntu.

```
$> apt-get install fpart
$> git clone https://github.com/aboovv1976/fpsync-k8s-rclone.git
$> cd fpsync-k8s-rclone/
```

```
$> cp -p /usr/bin/fpsync /usr/bin/k-fpsync
$> patch /usr/bin/k-fpsync fpsync.patch
```

6. Build the container image.
   The docker image build specification available in rclone-rsync-image can be used to build the container image. Once the image is built, it should be uploaded to a registry that can be accessed from the OKE cluster.

```
$> rclone-rsync-image
$> docker build -t rclone-rsync .
$> docker login
$> docker tag rclone-rsync:latest <registry url/rclone-rsync:latest>
$> docker push <registry url/rclone-rsync:latest>
```

   A copy of the image is maintained in fra.ocir.io/fsssolutions/rclone-rsync:latest. The sample directory contains some examples output files.

7. Run k-fpsync. The patched fpsync (k-fpsync) can partition the source file system and scale out the transfer using multiple Kubernetes pods. The Kubernetes pod anti-affinity rule is configured to prefer nodes that do not have any running transfer worker pods. This helps to utilize the bandwidth on the nodes effectively to optimize performance. For more information, see Assigning Pods to Nodes.
   Mount the source file system on the fpart operator host and create a shared directory that will be accessed by all the pods. This is the directory where all the log files and partition files are kept.
   The following command transfers data from the filesystem /data/src to the OCI Object Storage bucket rclone-2. It will start 2 pods at a time to transfer the file system partition created by fpart.

```
$> mkdir /data/fpsync
$> PART_SIZE=512 && ./k-fpsync -v -k fra.ocir.io/fsssolutions/rclone-rsync:latest,lustre-pvc  -m rclone \
-d /data/fpsync  -f $PART_SIZE -n 2 -o "--oos-no-check-bucket --oos-upload-cutoff 10Mi \
--multi-thread-cutoff 10Mi --no-check-dest --multi-thread-streams 64 --transfers $PART_SIZE  \
--oos-upload-concurrency 8 --oos-disable-checksum  --oos-leave-parts-on-error" /data/src/ rclone:rclone-2
```

Note: The logs for the run are kept in the run-ID directory, in the following example they are in /data/fpsync/{Run-Id}/log directory. The sample outputs are provided in the sample directory.

(Optional) Test Environments

Recommendations are made based on testing and customer interactions.

Note: Runs from the bulk copy script, os sync and s5cmd results are included to give more information on performance. Learn about using the bulk copy script from here: Use Oracle Cloud Infrastructure Object Storage Python Utilities for Bulk Operations. For more information about using os sync and the s5cmd, see Tutorial 3: Move Data into OCI Cloud Storage Services using OCI Object Storage Sync and S5cmd.

Test Environment 1:

1 VM instance VM.Standard.E4.Flex, 1 OCPU, 1Gbps network bandwidth, 16GB of memory. To simulate on-premises to OCI migration copied data from PHX NFS to IAD.

Data Sets

* Data Set 1:

Total Size	File Count	File Size Range
3TB	3	1TB

Method	To-From	Time	Command	Flags
os sync	NFS/File PHX to Object IAD	123m17.102s	NA	--parallel-operations-count 100
s5cmd	NFS/File PHX to Object IAD	239m20.625s	copy	run commands.txt, default run --numworkers 256
rclone	NFS/File PHX to Object IAD	178m27.101s	copy	--transfers=100 --oos-no-check-bucket --fast-list --checkers 64 --retries 2 --no-check-dest

Note: Our tests showed os sync running the fastest for this data set.

* Data set 2:

Total Size	File Count	File Size Range
9.787GB	20,000	1MB

Method	To-From	Time	Command	Flags
s5cmd	NFS/File PHX to Object IAD	1m12.746s	copy	default run --numworkers 256
os sync	NFS/File PHX to Object IAD	2m48.742s	NA	--parallel-operations-count 1000

rclone	NFS/File PHX to Object IAD	1m52.886s	copy	--transfers=500 --oos-no-check-bucket --no-check-dest

Note: Our tests showed s5cmd performing the best for this data set.

Test Environment 2:
VM Instances: 2 VM instances were used for each test, we used a VM.Standard.E4.Flex with 24 OCPU, 24Gbps network bandwidth, 384GB of memory. Oracle Linux 8 was used for Linux testing.

Data sets used in testing: 14 main directories with the following file count and sizes.

Data Set Directory	Size	File count	Size of Each File
Directory 1	107.658 GiB	110,242	1 MiB
Directory 2	1.687 GiB	110,569	15 MiB
Directory 3	222 GiB	111	2 GiB
Directory 4	1.265 TiB	1,295	1 GiB
Directory 5	26.359 GiB	1,687	16 MiB
Directory 6	105.281 MiB	26,952	4 KiB
Directory 7	29.697 MiB	30,410	1 KiB
Directory 8	83.124 GiB	340,488	256 KiB
Directory 9	21.662 GiB	354,909	64 KiB
Directory 10	142.629 GiB	36,514	4 MiB
Directory 11	452.328 MiB	57,898	8 MiB
Directory 12	144 GiB	72	2GiB
Directory 13	208.500 GiB	834	256 MiB
Directory 14	54.688 GiB	875	64 MiB

Note:
* The 14 directories were split between the 2 VM instances.
* Each VM ran 7 commands/processes, 1 for each directory unless otherwise noted.

Method	To-From	Time	Command	Flags/ Notes
s5cmd	NFS/File PHX to Object IAD	54m41.814s	copy	--numworkers 74
os sync	NFS/File PHX to Object IAD	65m43.200s	NA	--parallel-operations-count 50
rclone	NFS/File PHX to Object IAD	111m59.704s	copy	--oos-no-check-bucket --no-check-dest --ignore-checksum --oos-disable-checksum --transfers 50
rclone	Object PHX to Object IAD	28m55.663s	copy	--oos-no-check-bucket --no-check-dest --ignore-checksum --oos-disable-checksum --transfers 400, same command run across 2 VM's for a concurrency of 800 transfers
python bulk copy script	Object PHX to Object IAD	25m43.715s	Default	1 VM, 50 workers, 100,000 files queued at a time

The s5cmd and os sync commands do well over filesystem/NFS to object storage. The bulk copy script only does bucket-to-bucket transfers and was not tested for NFS migration.

Only rclone and the python bulk copy script are capable of doing bucket-to-bucket transfers across regions so the other tools were not tested for it. The python bulk copy script perfoms better on the cross region bucket-to-bucket data, however is only compatible with OCI Object Storage while rclone supports many backends and cloud providers.

Small test runs were conducted using rclone to transfer data from Microsoft Azure Blob Storage, Amazon Simple Storage Service (Amazon S3), and Google Cloud Platform Cloud Storage to OCI Object Storage to verify the tools works for these types of transfers. For more information, see Move data to object storage in the cloud by using Rclone.

Test Environment 3:
VM Instances: 1-2 VM instances were used for each test, we used a VM.Standard.E4.Flex with 24 OCPU, 24Gbps network bandwidth, 384GB of memory. Oracle Linux 8 was used for Linux testing. All tests were bucket-to-bucket.

Total Size	File Count	File Size Range
7.74 TiB	1,000,000	30 MiB

Method	To-From	Time	Command	Flags/ Notes
Method	To-From	Time	Command	Flags Notes-1 Notes-2

rclone	Object-to-Object IAD -> IAD	18h39m11.4s	copy	--oos-no-check-bucket --fast-list --no-traverse --transfers 500 --oos-chunk-size 10Mi 1 VM, very slow due to the high file count and listing calls to source
rclone	Object-to-Object IAD -> IAD	55m8.431s	copy	--oos-no-check-bucket --no-traverse --transfers 500 --oos-chunk-size 10Mi --files-from <file> 2 VM's, 500 transfers per VM, Vobject/file list fed 1,000 files at a time, prevents listing on source and destination and improves performance
python bulk copy script	Object-to-Object IAD -> IAD	28m21.013s	NA	Default 1 VM, 50 workers, 100,000 files queued at a time
python bulk copy script	Object-to-Object IAD -> IAD	NA	NA	Default 2 VMs, 50 workers per VM, 100,000 2 VMs, 50 workers per VM, 100,000 files queued at a time. Received 429 errors, script hung and could not complete
s5cmd	Object-to-Object IAD -> IAD	14m10.864s	copy	Defaults (256 workers) 1 VM NA
s5cmd	Object-to-Object IAD -> IAD	7m50.013s	copy	Defaults 2 VM's, 256 workers each VM Ran in abuot half the time as 1 VM
s5cmd	Object-to-Object IAD -> IAD	3m23.382s	copy	--numworkers 1000 1 VM, 1000 workers Across multiple tests we found this was the optimal run for this data set with the s5cmd
rclone	Object-to-Object IAD -> PHX	184m36.536s	copy	--oos-no-check-bucket --no-traverse --transfers 500 --oos-chunk-size 10Mi --files-from <file> 2 VM's, 500 transfers per VM, object/file list fed 1,000 files at a time
python bulk copy script	Object-to-Object IAD -> PHX	35m31.633s	NA	Default 1VM, 50 workers, 100,000 files queued at a time

The s5cmd command ran consistently best for the large file count and small files. The s5cmd is limited because it can only do bucket-to-bucket copies within the same tenancy and same region.

Notice high improvements to rclone once files are fed to the command and from scaling out to another VM. Rclone may run slower than other tools, it is the most versatile in the various platforms it supports and types of migrations it can perform.

The OCI Object Storage Bulk Copy Python API can only use the OCI Native CopyObject API and can only get up to a concurrency of 50 workers before being throttled.

Tests for IAD to PHX were only done on what worked best in IAD to IAD and problematic tests were not re-run. The s5cmd was not run for IAD to PHX because it can only do bucket-to-buckets copies within the same region.

Next Steps

Proceed to the related tutorial(s) relevant to your migration needs. To move data into OCI cloud storage services:

* Using OCI Object Storage Sync and S5cmd, see Tutorial 3: Move Data into OCI Cloud Storage Services using OCI Object Storage Sync and S5cmd.
* Using Fpsync and Rsync for file system data migrations, see Tutorial 4: Move Data into OCI Cloud Storage Services using Fpsync and Rsync for File System Data Migrations.

---

https://www.chpc.utah.edu/documentation/software/rclone.php

Rclone

Rclone is a command-line program that supports file transfers and syncing of files between local storage and Google Drive as well as a number of other storage services, including Box, Dropbox and Swift/S3-based services. Rclone offers options to optimize a transfer and reach higher transfer speeds than other common transfer tools such as scp and rsync. It is written in Go and is free software using a MIT license.

Installation of Rclone

If you wish to use rclone to transfer files to or from CHPC file systems, you can use the CHPC installation of rclone. There is a module to set the proper environment to use the tool. To use you first need to

```
$> module load rclone
```

If you wish to transfer files to/from file systems where neither the source or destination is on CHPC storage, for example when you want to use rclone to move files from your local desktop to Google Drive, you will first need to download and install rclone on the source or destination device.

Configuration of Rclone

The next step is to configure rclone for the transfer partner. Below we give four examples - one for Google Drive, a second for the CHPC archive storage solution, pando, a third for the University OneDrive storage, and a fourth for the University Box storage. When on CHPC resources, you must do the configuration while in your home directory. You can do the configuration step for multiple storage locations, such as with space on pando at CHPC, for use with the university's Google Drive or OneDrive storage, and/or a personal Google Drive/OneDrive space.

Please read updates on the use of the University Google Drive at end of this page!

There is a University of Utah central IT knowledge base article detailing university level storage options such as GSuite for Education, Microsoft O365, and Box. This article includes information on the suitability of the different storage options for different types of data. Please follow these university guidelines to determine a suitable location for your data.

To configure:
Note: If you are doing this to set up rclone to transfer data to/from CHPC resources, you must do this from a fastX session (preferred, easiest if using the web client of fastx with a full desktop) or an ssh session with X-forwarding enabled on CHPC resources, such as with an interactive cluster node. In one of the configuration steps a web browser will be opened on the server on which you have the ssh session, and without X-forwarding this will not work; in addition, the backup URL given at this point is only valid from the server on which you are doing the configuration.

1. Load the rclone module into your environment and then start the interactive rclone configuration.

```
$> module load rclone
```

```
rclone config
```

This command will create a .rclone.conf file in your current directory which contains the setup information. You will be asked a few questions:

2. Choose 'New remote' (note that you can have multiple remotes. If you have existing remotes they will be listed in the text that appears after you issue the 'rlone config' command. Also note that if you have configured a connection, but it no longer works (as the setups do time out) you can choose to 'Edit existing remote'.

```
[u1234567@notchpeak1 ~]$ rclone config
Current remotes:
Name      Type
====      ====
pandoDr   drive
e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> n
```

3. Enter a name for the Google Drive Rclone handle. This will be typed out whenever you want to access the Drive, so make it short.

```
name> gcloud
```

4. Select Google Drive ("drive"). At the time of creating this document Google Drive was choice 16. You should check this before entering your choice.

```
<long list of types>
16 / Google Drive
   \ "drive"
<several more types>
Storage> 16
```

5. Leave the Client ID and Client Secret fields blank--just press enter

```
<information>
Enter a string value. Press Enter for the default ("").
```

```
client_id>
<information>
client_secret>
<more information>
```

6. Option Scope - choose 1 (full access) for Google Drive

```
 <information on Scope>
scope> 1
```

7. ID of root folder - just press enter

8. Service account credentials - just press enter

9. Advanced configuration - choose no

```
<information>
root_folder_id>
<information>
service_account_file>
Edit advanced config? (y/n)
y) Yes
n) No
y/n> n
```

10. Choose 'Yes' for auto-config. (If you are recreating a remote, it will ask to refresh. Choose 'Yes')

```
Remote config
Already have a token - refresh?
y) Yes
n) No
y/n> y
Use auto config?
 * Say Y if not sure
 * Say N if you are working on a remote or headless machine
y) Yes
n) No
y/n> y
```

11. A web page will appear (you may first be given a choice of which browser you wish to use) that will make you select the google account to associate the rclone configuration. You will need to allow rclone access. Once completed, the page will tell you that you were successful, and that you need to return to your fastX or ssh session to complete the configuration. Note: at this point there will also be a URL given in case the web page does not open. This URL is only valid from the server on which you are doing the configuration. You cannot use this URL from your local machine.

12.You should select no to the team drive question.

```
Configure this as a team drive?
y) Yes
n) No
y/n> n
```

13. Choose Yes to confirm the configuration, and then quit the config

```
--------------
[gcloud]
type = drive
token = <don't share your token ID>
------------
y) Yes this is OK
e) Edit this remote
d) Delete this remote
y/e/d> y
Current remotes:
Name       Type
=====     ======
gcloud     drive
pandoDr    drive
e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> q
```

You are now enabled to access your Google Drive via rclone.

If you find you are having issues with connecting to your gcloud or box, or need to refresh a connection or token that has expired, look at this troubleshooting page for help.

Example 2: Configuration for transferring files to/from CHPC pando archive storage
Rclone uses the Ceph Gateway box, pando-rgw01.chpc.utah.edu to interact with the archive storage. Please note that groups must purchase space on the Ceph archive storage in order to make use of this space. For additional information about the archive storage please see its description on the CHPC storage services page. For the Elm protected environment archive storage, the gateway machine is elm-rgw01.int.chpc.utah.edu.

As before, you will have to first load the rclone module and then you need to create a file in your $HOME/.config/rclone directory called .rclone.conf with the following five lines, using the name of your choice, and the keys you were given when the space was provisioned. Note that if you already have a .rclone.conf file, you can append these to that file, with a blank line after the previous configurations.

```
[name]
type = s3
access_key_id = your secret key
secret_access_key = your access key
endpoint = https://pando-rgw01.chpc.utah.edu/ # for general use Pando storage
endpoint = https://elm-rgw01.int.chpc.utah.edu/ # for Protected Environment Elm storage
```

The secret and access key are provided to the PI when the Pando space is provisioned. They are in a file located in the top level of the PI's home directory under name XXXX_pando_key, XXXX standing for the group's name.

Test if configuration works by rclone lsd {name}: You will see a list of all the 'buckets' (Ceph term for folders) associated with the gateway (make sure to include the trailing colon). As at this point you do not yet have any buckets, this should not give any results, but it should also not return any errors . To create a bucket run: rclone mkdir {$name}:{bucket}

NOTE: the bucket name used must be unique to all of pando. If you use a name that has already been taken, you will get the following error: "ERROR : Attempt 1/3 failed with 1 errors and: Forbidden: Forbidden status code: 403, request id: tx000000000000000e0f683-005afc84dc-19b5a49-default, host id:" If this happens, redo with a different bucket name.

Example 3: Configuration for transferring files to/from the University of Utah Box
To configure rclone usage with the University Box storage, follow the steps for Google Drive in Example 1. There are only three changes:

1. Instead of choosing Google Drive in the third step, you will find the number for Box (it was 7 when this documentation was created).
2. In step 6, instead of asking for the 'Scope" it will ask about "box_sub_type". Choose option 1.
3. In step 11, when the web browser opens, you will choose the SSO option, and sign in with your utah.edu email address.

Example 3: Configuration for transferring files to/from the University of Utah OneDrive
To configure rclone usage with the University OneDrive storage, follow the steps for Google Drive in Example 1. There are only four changes:

1. Instead of choosing Google Drive in the third step, you will find the number for OneDrive (it was 27 when this documentation was created).
2. Steps 7 and 8 are not asked for OneDrive configuration to RClone.
3. In step 6, instead of asking for the 'Scope" it will ask about the "national cloud region for OneDrive". Choose option 1.
4. After verifying your credentials to OneDrive, it will ask if the drive is okay. Respond with yes (y).

Rclone Usage

In this section some common rclone usage cases are presented. In the following the name mydrive is being used. You would need to use the name you choose when doing your configuration. Note the trailing colon. This indicates to rclone that "mydrive" is a remote storage system, rather than a file or directory called "mydrive" in your current working directory. At any point, you may verify that these changes were successful by viewing your Drive from within a web browser. There is a short training video that covers the information presented below. Note that in Ceph, folders are referred to as buckets

- List all files in your Drive: rclone ls mydrive:
- List top-level buckets in your Drive: rclone lsd mydrive:
- Create bucket on PANDO:rclone mkdir {$name}:{bucket}
- Copy a file between two sources: rclone copy SOURCE DESTINATION
  - Example: To copy a file called "rclone-test.txt" from your local machine home directory to your Drive, or a subdirectory within it:
    - rclone copy ~/rclone-test.txt mydrive:
    - rclone copy ~/rclone-test.txt mydrive:my-rc-folder
  - You can also transfer files directly between your Drive and another remote storage system, such as an object storage service for which you have configured rclone:
    - rclone copy mydrive:rclone-test.txt myobjectstorage:some-bucket
- Synchronizing directories is done with the sync option. rclone sync SOURCE/ DESTINATION/ [--drive-use-trash]
  - Rclone can synchronize an entire Drive folder with the destination directory. This is a full synchronization, so files at the destination prior to the sync will be overwritten or deleted. Double check the destination and its contents, and be mindful if the directory is already being synchronized by other services.
  - Example: To make a folder called "backup" on Google Drive, then sync a directory from the local machine to the new folder.
    - rclone mkdir mydrive:backup
    - rclone sync ~/local-folder mydrive:backup

Rclone Options

While the full list of options can be found in the official MANUAL file in the Rclone github repo (or 'man rclone' if rclone is installed), some important options are:

- --config=FILE (default FILE=.rclone.conf)

Specifies the rclone configuration file to use. Only necessary if the desired config file is not the default (which may be ~/.rclone.conf or ~/.config/rclone/rclone.conf, depending on the version used).

- --transfers=N (default N=4)

Number of file transfers to be run in parallel. Increasing this may increase the overall speed of a large transfer, as long as the network and remote storage system can handle it (bandwidth and memory).

* --drive-chunk-size=SIZE (default SIZE=8192)

The chunk size for a transfer in kilobytes; must be a power of 2 and at least 256. Each chunk is buffered in memory prior to the transfer, so increasing this increases how much memory is used.

* --drive-use-trash

Sends files to Google Drive's trash instead of deleting (prior to a directory sync for instance). Note that this is not a default option, because the Trash is not accessible through Rclone and must be managed through a web browser.

* --drive-formats (docx, pdf, txt, etc.)

Sets the format used when exporting files. For example, the option '--drive-formats pdf' will automatically convert the chosen file(s) to PDF format.

Additional Important Considerations

* Google limits transfers to about 2 files per second. This may cause uploads of many small files to be much slower than the upload rate. However, it will not stop the transfer and will continue to retry files that were blocked by Google's rate limit. Considering compressing small files into a single larger file if this becomes a problem.
* The campus firewall may impede larger transfers. The University has a Science DMZ network with Data Transfer Nodes (DTNs), which can be used to safely and conveniently facilitate larger transfers without the firewall's limitations. Additional information on data transfer services can be found on our data transfer services page.
* Transfer ratemay vary heavily. A number of factors, including the current state of Google's resources as well as University resources, determine the rate of transfer. Results may vary over minutes, hours, or days. If there is a consistent problem, check if machines associated with the transfer are running into network/disk/memory bottlenecks.

Automating Transfers Using Rclone

The file transfer process can be scripted, and the script can be used in conjunction with a cron job to automate a transfer. With these scripts a user can set up a periodic backup their data.

Examples for doing this to transfer to Google Drive are given in /uufs/chpc.utah.edu/sys/installdir/rclone/etc . The example scripts provided can be adapted for transferring data to other destinations such as pando. elm, or UBox. Should you need assistance in setting up an automated transfer, please send a request to helpdesk@chpc.utah.edu.

Exploring the Effects of Options on Performance

In order to explore the effects of Rclone options on data transfer performance, we completed multiple transfers of the contents of a directory on a data transfer node (DTN), to a folder on Google drive. This directory contained 16 files, each 1.7GB each.

The data transfer node has a 40gbs connection on the University of Utah Science DMZ. To explore performance, we completed runs with the number of parallel transfers set to 4 and to 16 and with the chunk size set to 8MB, 16MB, and 32MB. The command was run ten times for each combination of options. A base transfer using a single transfer and a chunk size of 8MB is included as a reference point. The below chart displays the achieved transfer rates of the different scenarios.

As the chart shows, increasing both the number of parallel transfers and the chunk size improves the transfer rate over using the default sizes.

A few notes about GoogleDrive storage:

* Any University faculty, student or staff can activate and access their official U of U Google Drive by visiting the above link and logging in.
* The University's Google Drive offered via the GSuite for Education is now suitable for storing sensitive and restricted data. For additional security information, consult the Security Section of the above link.
* Limits are 25 GB for students and 150 GB for faculty and staff. The University's Google agreement also allows UIT to provide users, departments, colleges, and other units with additional Google Workspace storage for $200/terabyte (TB) per year. The University's intended use of Google Workspace is to store university-related collaboration files, not large-scale research data, long-term storage, or system backups.

---

https://rclone.org/filtering/

Rclone: Filtering, includes and excludes

Filter flags determine which files rclone sync, move, ls, lsl, md5sum, sha1sum, size, delete, check and similar commands apply to.

They are specified in terms of path/file name patterns; path/file lists; file age and size, or presence of a file in a directory. Bucket based remotes without the concept of directory apply filters to object key, age and size in an analogous way.

Rclone purge does not obey filters.

To test filters without risk of damage to data, apply them to rclone ls, or with the --dry-run and -vv flags.

Rclone filter patterns can only be used in filter command line options, not in the specification of a remote.

E.g. rclone copy "remote:dir*.jpg" /path/to/dir does not have a filter effect. rclone copy remote:dir /path/to/dir --include "*.jpg" does.

Important Avoid mixing any two of --include..., --exclude... or --filter... flags in an rclone command. The results might not be what you expect. Instead use a --filter... flag.

Patterns for matching path/file names
Pattern syntax
Here is a formal definition of the pattern syntax, examples are below.

Rclone matching rules follow a glob style:
* matches any sequence of non-separator (/) characters
** matches any sequence of characters including / separators
? matches any single non-separator (/) character
[[!] { character-range }]
character class (must be non-empty)
{ pattern-list }
pattern alternatives
{{ regexp }}
regular expression to match
c matches character c (c != *, **, ?, \, [, {, })
\c matches reserved character c (c = *, **, ?, \, [, {, }) or character class

character-range:
c matches character c (c != \, -,])
\c matches reserved character c (c = \, -,])
lo - hi matches character c for lo <= c <= hi

pattern-list:
pattern { , pattern }
comma-separated (without spaces) patterns

character classes (see Go regular expression reference) include:
Named character classes (e.g. [\d], [^\d], [\D], [^\D])
Perl character classes (e.g. \s, \S, \w, \W)
ASCII character classes (e.g. [[:alnum:]], [[:alpha:]], [[:punct:]], [[:xdigit:]])

regexp for advanced users to insert a regular expression - see below for more info:
Any re2 regular expression not containing `}}`

If the filter pattern starts with a / then it only matches at the top level of the directory tree, relative to the root of the remote (not necessarily the root of the drive). If it does not start with / then it is matched starting at the end of the path/file name but it only matches a complete path element - it must match from a / separator or the beginning of the path/file.
file.jpg - matches "file.jpg"
- matches "directory/file.jpg"
- doesn't match "afile.jpg"
- doesn't match "directory/afile.jpg"
/file.jpg - matches "file.jpg" in the root directory of the remote
- doesn't match "afile.jpg"
- doesn't match "directory/file.jpg"

The top level of the remote might not be the top level of the drive.

E.g. for a Microsoft Windows local directory structure
```
F:
+-- bkp
+-- data
|   +-- excl
|   |   +-- 123.jpg
|   |   +-- 456.jpg
|   +-- incl
|   |   +-- document.pdf
```

To copy the contents of folder data into folder bkp excluding the contents of subfolder exclthe following command treats F:\data and F:\bkp as top level for filtering.

```
rclone copy F:\data\ F:\bkp\ --exclude=/excl/**
```

Important Use / in path/file name patterns and not \ even if running on Microsoft Windows.

Simple patterns are case sensitive unless the --ignore-case flag is used.

Without --ignore-case (default)
potato - matches "potato"
- doesn't match "POTATO"

With --ignore-case
potato - matches "potato"
- matches "POTATO"

Using regular expressions in filter patterns
The syntax of filter patterns is glob style matching (like bash uses) to make things easy for users. However this does not provide absolute control over the matching, so for advanced users rclone also provides a regular expression syntax.

The regular expressions used are as defined in the Go regular expression reference. Regular expressions should be enclosed in {{ }}. They will match only the last path segment if the glob doesn't start with / or the whole path name if it does. Note that rclone does not attempt to parse the supplied regular expression, meaning that using any regular expression filter will prevent rclone from using directory filter rules, as it will instead check every path against the supplied regular expression(s).

Here is how the {{regexp}} is transformed into an full regular expression to match the entire path:

```
{{regexp}}  becomes (^|/)(regexp)$
/{{regexp}} becomes ^(regexp)$
```

Regexp syntax can be mixed with glob syntax, for example

```
*.{{jpe?g}} to match file.jpg, file.jpeg but not file.png
```

You can also use regexp flags - to set case insensitive, for example

```
*.{{(?i)jpg}} to match file.jpg, file.JPG but not file.png
```

Be careful with wildcards in regular expressions - you don't want them to match path separators normally. To match any file name starting with start and ending with end write

```
{{start[^/]*end\.jpg}}
```

Not

```
{{start.*end\.jpg}}
```

Which will match a directory called start with a file called end.jpg in it as the .* will match / characters.

Note that you can use -vv --dump filters to show the filter patterns in regexp format - rclone implements the glob patterns by transforming them into regular expressions.

Filter pattern examples

Description	Pattern	Matches	Does not match
Wildcard	*.jpg	/file.jpg	/file.png
		/dir/file.jpg	/dir/file.png
Rooted	/*.jpg	/file.jpg	/file.png
		/file2.jpg	/dir/file.jpg
Alternates	*.{jpg,png}	/file.jpg	/file.gif
		/dir/file.png	/dir/file.gif
Path Wildcard	dir/**	/dir/anyfile	file.png
		/subdir/dir/subsubdir/anyfile	/subdir/file.png
Any Char	*.t?t	/file.txt	/file.qxt
		/dir/file.tzt	/dir/file.png
Range	*.[a-z]	/file.a	/file.0
		/dir/file.b	/dir/file.1
Escape	*.\?\?\?	/file.???	/file.abc
		/dir/file.???	/dir/file.def
Class	*.\d\d\d	/file.012	/file.abc
		/dir/file.345	/dir/file.def
Regexp	*.{{jpe?g}}	/file.jpeg	/file.png
		/dir/file.jpg	/dir/file.jpeeg
Rooted Regexp	/{{.*\.jpe?g}}	/file.jpeg	/file.png
		/file.jpg	/dir/file.jpg

How filter rules are applied to files

Rclone path/file name filters are made up of one or more of the following flags:

- --include
- --include-from
- --exclude
- --exclude-from
- --filter
- --filter-from

There can be more than one instance of individual flags.

Rclone internally uses a combined list of all the include and exclude rules. The order in which rules are processed can influence the result of the filter.

All flags of the same type are processed together in the order above, regardless of what order the different types of flags are included on the command line.

Multiple instances of the same flag are processed from left to right according to their position in the command line.

To mix up the order of processing includes and excludes use --filter... flags.

Within --include-from, --exclude-from and --filter-from flags rules are processed from top to bottom of the referenced file.

If there is an --include or --include-from flag specified, rclone implies a - ** rule which it adds to the bottom of the internal rule list. Specifying a + rule with a --filter... flag does not imply that rule.

Each path/file name passed through rclone is matched against the combined filter list. At first match to a rule the path/file name is included or excluded and no further filter rules are processed for that path/file.

If rclone does not find a match, after testing against all rules (including the implied rule if appropriate), the path/file name is included.

Any path/file included at that stage is processed by the rclone command.

--files-from and --files-from-raw flags over-ride and cannot be combined with other filter options.

To see the internal combined rule list, in regular expression form, for a command add the --dump filters flag. Running an rclone command with --dump filters and -vv flags lists the internal filter elements and shows how they are applied to each source path/file. There is not currently a means provided to pass regular expression filter options into rclone directly though character class filter rules contain character classes. Go regular expression reference

How filter rules are applied to directories

Rclone commands are applied to path/file names not directories. The entire contents of a directory can be matched to a filter by the pattern directory/* or recursively by directory/**.

Directory filter rules are defined with a closing / separator.

E.g. /directory/subdirectory/ is an rclone directory filter rule.

Rclone commands can use directory filter rules to determine whether they recurse into subdirectories. This potentially optimises access to a remote by avoiding listing unnecessary directories. Whether optimisation is desirable depends on the specific filter rules and source remote content.

If any regular expression filters are in use, then no directory recursion optimisation is possible, as rclone must check every path against the supplied regular expression(s).

Directory recursion optimisation occurs if either:

* A source remote does not support the rclone ListR primitive. local, sftp, Microsoft OneDrive and WebDAV do not support ListR. Google Drive and most bucket type storage do. Full list
* On other remotes (those that support ListR), if the rclone command is not naturally recursive, and provided it is not run with the --fast-list flag. ls, lsf -R and size are naturally recursive but sync, copy and move are not.
* Whenever the --disable ListR flag is applied to an rclone command.

Rclone commands imply directory filter rules from path/file filter rules. To view the directory filter rules rclone has implied for a command specify the --dump filters flag.

E.g. for an include rule

```
/a/*.jpg
```

Rclone implies the directory include rule

```
/a/
```

Directory filter rules specified in an rclone command can limit the scope of an rclone command but path/file filters still have to be specified.

E.g. rclone ls remote: --include /directory/ will not match any files. Because it is an --include option the --exclude ** rule is implied, and the /directory/ pattern serves only to optimise access to the remote by ignoring everything outside of that directory.

E.g. rclone ls remote: --filter-from filter-list.txt with a file filter-list.txt:

```
- /dir1/
- /dir2/
+ *.pdf
- **
```

All files in directories dir1 or dir2 or their subdirectories are completely excluded from the listing. Only files of suffix pdf in the root of remote: or its subdirectories are listed. The - ** rule prevents listing of any path/files not previously matched by the rules above.

Option exclude-if-present creates a directory exclude rule based on the presence of a file in a directory and takes precedence over other rclone directory filter rules.

When using pattern list syntax, if a pattern item contains either / or **, then rclone will not able to imply a directory filter rule from this pattern list.

E.g. for an include rule

```
{dir1/**,dir2/**}
```

Rclone will match files below directories dir1 or dir2 only, but will not be able to use this filter to exclude a directory dir3 from being traversed.

Directory recursion optimisation may affect performance, but normally not the result. One exception to this is sync operations with option --create-empty-src-dirs, where any traversed empty directories will be created. With the pattern list example {dir1/**,dir2/**} above, this would create an empty directory dir3 on destination (when it exists on source). Changing the filter to {dir1,dir2}/**, or splitting it into two include rules --include dir1/** --include dir2/**, will match the same files while also filtering directories, with the result that an empty directory dir3 will no longer be created.

```
--exclude - Exclude files matching pattern
   Excludes path/file names from an rclone command based on a single exclude rule.

   This flag can be repeated. See above for the order filter flags are processed in.

   --exclude should not be used with --include, --include-from, --filter or --filter-from flags.

   --exclude has no effect when combined with --files-from or --files-from-raw flags.

   E.g. rclone ls remote: --exclude *.bak excludes all .bak files from listing.

   E.g. rclone size remote: --exclude "/dir/**" returns the total size of all files on remote: excluding
   those in root directory dir and sub directories.

   E.g. on Microsoft Windows rclone ls remote: --exclude "*\[{JP,KR,HK}\]*" lists the files in remote:
   without [JP] or [KR] or [HK] in their name. Quotes prevent the shell from interpreting the \
   characters.\ characters escape the [ and ] so an rclone filter treats them literally rather than as a
   character-range. The { and } define an rclone pattern list. For other operating systems single quotes
   are required ie rclone ls remote: --exclude '*\[{JP,KR,HK}\]*'

--exclude-from - Read exclude patterns from file
   Excludes path/file names from an rclone command based on rules in a named file. The file contains a
   list of remarks and pattern rules.

   For an example exclude-file.txt:
# a sample exclude rule file
*.bak
file2.jpg

   rclone ls remote: --exclude-from exclude-file.txt lists the files on remote: except those named
   file2.jpg or with a suffix .bak. That is equivalent to rclone ls remote: --exclude file2.jpg
   --exclude "*.bak".

   This flag can be repeated. See above for the order filter flags are processed in.

   The --exclude-from flag is useful where multiple exclude filter rules are applied to an rclone
   command.

   --exclude-from should not be used with --include, --include-from, --filter or --filter-from flags.

   --exclude-from has no effect when combined with --files-from or --files-from-raw flags.

   --exclude-from followed by - reads filter rules from standard input.

--include - Include files matching pattern
   Adds a single include rule based on path/file names to an rclone command.

   This flag can be repeated. See above for the order filter flags are processed in.

   --include has no effect when combined with --files-from or --files-from-raw flags.

   --include implies --exclude ** at the end of an rclone internal filter list. Therefore if you mix
   --include and --include-from flags with --exclude, --exclude-from, --filter or --filter-from, you
   must use include rules for all the files you want in the include statement. For more flexibility use
   the --filter-from flag.

   E.g. rclone ls remote: --include "*.{png,jpg}" lists the files on remote: with suffix .png and .jpg.
   All other files are excluded.

   E.g. multiple rclone copy commands can be combined with --include and a pattern-list.
$> rclone copy /vol1/A remote:A
$> rclone copy /vol1/B remote:B

   is equivalent to:
$> rclone copy /vol1 remote: --include "{A,B}/**"

   E.g. rclone ls remote:/wheat --include "??[^[:punct:]]*" lists the files remote: directory wheat (and
   subdirectories) whose third character is not punctuation. This example uses an ASCII character
   class.

--include-from - Read include patterns from file
   Adds path/file names to an rclone command based on rules in a named file. The file contains a list of
   remarks and pattern rules.

   For an example include-file.txt:
# a sample include rule file
*.jpg
file2.avi

   rclone ls remote: --include-from include-file.txt lists the files on remote: with name file2.avi or
   suffix .jpg. That is equivalent to rclone ls remote: --include file2.avi --include "*.jpg".

   This flag can be repeated. See above for the order filter flags are processed in.

   The --include-from flag is useful where multiple include filter rules are applied to an rclone
   command.
```

```
   --include-from implies --exclude ** at the end of an rclone internal filter list. Therefore if you
   mix --include and --include-from flags with --exclude, --exclude-from, --filter or --filter-from, you
   must use include rules for all the files you want in the include statement. For more flexibility use
   the --filter-from flag.

   --include-from has no effect when combined with --files-from or --files-from-raw flags.

   --include-from followed by - reads filter rules from standard input.

--filter - Add a file-filtering rule
   Specifies path/file names to an rclone command, based on a single include or exclude rule, in + or -
   format.

   This flag can be repeated. See above for the order filter flags are processed in.

   --filter + differs from --include. In the case of --include rclone implies an --exclude * rule which
   it adds to the bottom of the internal rule list. --filter...+ does not imply that rule.

   --filter has no effect when combined with --files-from or --files-from-raw flags.

   --filter should not be used with --include, --include-from, --exclude or --exclude-from flags.

   E.g. rclone ls remote: --filter "- *.bak" excludes all .bak files from a list of remote:.

--filter-from - Read filtering patterns from a file
   Adds path/file names to an rclone command based on rules in a named file. The file contains a list of
   remarks and pattern rules. Include rules start with + and exclude rules with - . ! clears existing
   rules. Rules are processed in the order they are defined.

   This flag can be repeated. See above for the order filter flags are processed in.

   Arrange the order of filter rules with the most restrictive first and work down.

   Lines starting with # or ; are ignored, and can be used to write comments. Inline comments are not
   supported. Use -vv --dump filters to see how they appear in the final regexp.

   E.g. for filter-file.txt:
# a sample filter rule file
- secret*.jpg
+ *.jpg
+ *.png
+ file2.avi
- /dir/tmp/** # WARNING! This text will be treated as part of the path.
- /dir/Trash/**
+ /dir/**
# exclude everything else
- *

   rclone ls remote: --filter-from filter-file.txt lists the path/files on remote: including all jpg and
   png files, excluding any matching secret*.jpg and including file2.avi. It also includes everything in
   the directory dir at the root of remote, except remote:dir/Trash which it excludes. Everything else
   is excluded.

   E.g. for an alternative filter-file.txt:
- secret*.jpg
+ *.jpg
+ *.png
+ file2.avi
- *

   Files file1.jpg, file3.png and file2.avi are listed whilst secret17.jpg and files without the suffix
   .jpg or .png are excluded.

   E.g. for an alternative filter-file.txt:
+ *.jpg
+ *.gif
!
+ 42.doc
- *

   Only file 42.doc is listed. Prior rules are cleared by the !.

--files-from - Read list of source-file names
   Adds path/files to an rclone command from a list in a named file. Rclone processes the path/file
   names in the order of the list, and no others.

   Other filter flags (--include, --include-from, --exclude, --exclude-from, --filter and --filter-from)
   are ignored when --files-from is used.

   --files-from expects a list of files as its input. Leading or trailing whitespace is stripped from
   the input lines. Lines starting with # or ; are ignored.

   --files-from followed by - reads the list of files from standard input.

   Rclone commands with a --files-from flag traverse the remote, treating the names in --files-from as a
```

set of filters.

If the --no-traverse and --files-from flags are used together an rclone command does not traverse the remote. Instead it addresses each path/file named in the file individually. For each path/file name, that requires typically 1 API call. This can be efficient for a short --files-from list and a remote containing many files.

Rclone commands do not error if any names in the --files-from file are missing from the source remote.

The --files-from flag can be repeated in a single rclone command to read path/file names from more than one file. The files are read from left to right along the command line.

Paths within the --files-from file are interpreted as starting with the root specified in the rclone command. Leading / separators are ignored. See --files-from-raw if you need the input to be processed in a raw manner.

E.g. for a file files-from.txt:

```
# comment
file1.jpg
subdir/file2.jpg
```

rclone copy --files-from files-from.txt /home/me/pics remote:pics copies the following, if they exist, and only those files.

```
/home/me/pics/file1.jpg        -->remote:pics/file1.jpg
/home/me/pics/subdir/file2.jpg -->remote:pics/subdir/file2.jpg
```

E.g. to copy the following files referenced by their absolute paths:

```
/home/user1/42
/home/user1/dir/ford
/home/user2/prefect
```

First find a common subdirectory - in this case /home and put the remaining files in files-from.txt with or without leading /, e.g.

```
user1/42
user1/dir/ford
user2/prefect
```

Then copy these to a remote:

```
$> rclone copy --files-from files-from.txt /home remote:backup
```

The three files are transferred as follows:

```
/home/user1/42       -->remote:backup/user1/important
/home/user1/dir/ford -->remote:backup/user1/dir/file
/home/user2/prefect  -->remote:backup/user2/stuff
```

Alternatively if / is chosen as root files-from.txt will be:

```
/home/user1/42
/home/user1/dir/ford
/home/user2/prefect
```

The copy command will be:

```
$> rclone copy --files-from files-from.txt / remote:backup
```

Then there will be an extra home directory on the remote:

```
/home/user1/42       -->remote:backup/home/user1/42
/home/user1/dir/ford -->remote:backup/home/user1/dir/ford
/home/user2/prefect  -->remote:backup/home/user2/prefect
```

--files-from-raw - Read list of source-file names without any processing

This flag is the same as --files-from except that input is read in a raw manner. Lines with leading / trailing whitespace, and lines starting with ; or # are read without any processing. rclone lsf has a compatible format that can be used to export file lists from remotes for input to --files-from-raw.

--ignore-case - make searches case insensitive

By default, rclone filter patterns are case sensitive. The --ignore-case flag makes all of the filters patterns on the command line case insensitive.

E.g. --include "zaphod.txt" does not match a file Zaphod.txt. With --ignore-case a match is made.

Quoting shell metacharacters

Rclone commands with filter patterns containing shell metacharacters may not as work as expected in your shell and may require quoting.

E.g. linux, OSX (* metacharacter)

- --include *.jpg
- --include '*.jpg'
- --include='*.jpg'

Microsoft Windows expansion is done by the command, not shell, so --include *.jpg does not require quoting.

If the rclone error Command .... needs .... arguments maximum: you provided .... non flag arguments: is encountered, the cause is commonly spaces within the name of a remote or flag value. The fix then

```
   is to quote values containing spaces.

Other filters
--min-size - Don't transfer any file smaller than this
   Controls the minimum size file within the scope of an rclone command. Default units are KiB but
   abbreviations B, K, M, G, T or P are valid.

   E.g. rclone ls remote: --min-size 50k lists files on remote: of 50 KiB size or larger.

   See the size option docs for more info.

--max-size - Don't transfer any file larger than this
   Controls the maximum size file within the scope of an rclone command. Default units are KiB but
   abbreviations B, K, M, G, T or P are valid.

   E.g. rclone ls remote: --max-size 1G lists files on remote: of 1 GiB size or smaller.

   See the size option docs for more info.

--max-age - Don't transfer any file older than this
   Controls the maximum age of files within the scope of an rclone command.

   --max-age applies only to files and not to directories.

   E.g. rclone ls remote: --max-age 2d lists files on remote: of 2 days old or less.

   See the time option docs for valid formats.

--min-age - Don't transfer any file younger than this
   Controls the minimum age of files within the scope of an rclone command. (see --max-age for valid
   formats)

   --min-age applies only to files and not to directories.

   E.g. rclone ls remote: --min-age 2d lists files on remote: of 2 days old or more.

   See the time option docs for valid formats.

--hash-filter - Deterministically select a subset of files
   The --hash-filter flag enables selecting a deterministic subset of files, useful for:
    1. Running large sync operations across multiple machines.
    2. Checking a subset of files for bitrot.
    3. Any other operations where a sample of files is required.

Syntax
   The flag takes two parameters expressed as a fraction:
--hash-filter K/N

     * N: The total number of partitions (must be a positive integer).
     * K: The specific partition to select (an integer from 0 to N).

   For example:
     * --hash-filter 1/3: Selects the first third of the files.
     * --hash-filter 2/3 and --hash-filter 3/3: Select the second and third partitions, respectively.

   Each partition is non-overlapping, ensuring all files are covered without duplication.

Random Partition Selection
   Use @ as K to randomly select a partition:
--hash-filter @/M

   For example, --hash-filter @/3 will randomly select a number between 0 and 2. This will stay constant
   across retries.

How It Works
     * Rclone takes each file's full path, normalizes it to lowercase, and applies Unicode
       normalization.
     * It then hashes the normalized path into a 64 bit number.
     * The hash result is reduced modulo N to assign the file to a partition.
     * If the calculated partition does not match K the file is excluded.
     * Other filters may apply if the file is not excluded.

   Important: Rclone will traverse all directories to apply the filter.

Usage Notes
     * Safe to use with rclone sync; source and destination selections will match.
     * Do not use with --delete-excluded, as this could delete unselected files.
     * Ignored if --files-from is used.

Examples

Dividing files into 4 partitions
   Assuming the current directory contains file1.jpg through file9.jpg:
$> rclone lsf --hash-filter 0/4 .
file1.jpg
file5.jpg
```

```
$> rclone lsf --hash-filter 1/4 .
file3.jpg
file6.jpg
file9.jpg

$> rclone lsf --hash-filter 2/4 .
file2.jpg
file4.jpg

$> rclone lsf --hash-filter 3/4 .
file7.jpg
file8.jpg

$> rclone lsf --hash-filter 4/4 . # the same as --hash-filter 0/4
file1.jpg
file5.jpg

Syncing the first quarter of files
$> rclone sync --hash-filter 1/4 source:path destination:path

Checking a random 1% of files for integrity
$> rclone check --download --hash-filter @/100 source:path destination:path

Other flags
--delete-excluded - Delete files on dest excluded from sync
   Important this flag is dangerous to your data - use with --dry-run and -v first.

   In conjunction with rclone sync, --delete-excluded deletes any files on the destination which are
   excluded from the command.

   E.g. the scope of rclone sync --interactive A: B: can be restricted:
$> rclone --min-size 50k --delete-excluded sync A: B:

   All files on B: which are less than 50 KiB are deleted because they are excluded from the rclone sync
   command.

--dump filters - dump the filters to the output
   Dumps the defined filters to standard output in regular expression format.

   Useful for debugging.

Exclude directory based on a file
   The --exclude-if-present flag controls whether a directory is within the scope of an rclone command
   based on the presence of a named file within it. The flag can be repeated to check for multiple file
   names, presence of any of them will exclude the directory.

   This flag has a priority over other filter flags.

   E.g. for the following directory structure:
dir1/file1
dir1/dir2/file2
dir1/dir2/dir3/file3
dir1/dir2/dir3/.ignore

   The command rclone ls --exclude-if-present .ignore dir1 does not list dir3, file3 or .ignore.

Metadata filters
   The metadata filters work in a very similar way to the normal file name filters, except they match
   metadata on the object.

   The metadata should be specified as key=value patterns. This may be wildcarded using the normal
   filter patterns or regular expressions.

   For example if you wished to list only local files with a mode of 100664 you could do that with:
$> rclone lsf -M --files-only --metadata-include "mode=100664" .

   Or if you wished to show files with an atime, mtime or btime at a given date:
$> rclone lsf -M --files-only --metadata-include "[abm]time=2022-12-16*" .

   Like file filtering, metadata filtering only applies to files not to directories.

   The filters can be applied using these flags.
     * --metadata-include - Include metadatas matching pattern
     * --metadata-include-from - Read metadata include patterns from file (use - to read from stdin)
     * --metadata-exclude - Exclude metadatas matching pattern
     * --metadata-exclude-from - Read metadata exclude patterns from file (use - to read from stdin)
     * --metadata-filter - Add a metadata filtering rule
     * --metadata-filter-from - Read metadata filtering patterns from a file (use - to read from stdin)

   Each flag can be repeated. See the section on how filter rules are applied for more details -
   these flags work in an identical way to the file name filtering flags, but instead of file name
   patterns have metadata patterns.

Common pitfalls
   The most frequent filter support issues on the rclone forum are:
```

    * Not using paths relative to the root of the remote
    * Not using / to match from the root of a remote
    * Not using ** to match the contents of a directory

---
https://rclone.org/commands/

Rclone Commands

   This is an index of all commands in rclone. Run rclone command --help to see the help for that
   command.

Command	Description
rclone	Show help for rclone commands, flags and backends.
rclone about	Get quota information from the remote.
rclone authorize	Remote authorization.
rclone backend	Run a backend-specific command.
rclone bisync	Perform bidirectional synchronization between two paths.
rclone cat	Concatenates any files and sends them to stdout.
rclone check	Checks the files in the source and destination match.
rclone checksum	Checks the files in the destination against a SUM file.
rclone cleanup	Clean up the remote if possible.
rclone completion	Output completion script for a given shell.
rclone completion bash	Output bash completion script for rclone.
rclone completion fish	Output fish completion script for rclone.
rclone completion powershell	Output powershell completion script for rclone.
rclone completion zsh	Output zsh completion script for rclone.
rclone config	Enter an interactive configuration session.
rclone config create	Create a new remote with name, type and options.
rclone config delete	Delete an existing remote.
rclone config disconnect	Disconnects user from remote
rclone config dump	Dump the config file as JSON.
rclone config edit	Enter an interactive configuration session.
rclone config encryption	set, remove and check the encryption for the config file
rclone config encryption check	Check that the config file is encrypted
rclone config encryption remove	Remove the config file encryption password
rclone config encryption set	Set or change the config file encryption password
rclone config file	Show path of configuration file in use.
rclone config password	Update password in an existing remote.
rclone config paths	Show paths used for configuration, cache, temp etc.
rclone config providers	List in JSON format all the providers and options.
rclone config reconnect	Re-authenticates user with remote.
rclone config redacted	Print redacted (decrypted) config file, or the redacted config for a single remote.
rclone config show	Print (decrypted) config file, or the config for a single remote.
rclone config touch	Ensure configuration file exists.
rclone config update	Update options in an existing remote.
rclone config userinfo	Prints info about logged in user of remote.
rclone convmv	Convert file and directory names in place.
rclone copy	Copy files from source to dest, skipping identical files.
rclone copyto	Copy files from source to dest, skipping identical files.
rclone copyurl	Copy the contents of the URL supplied content to dest:path.
rclone cryptcheck	Cryptcheck checks the integrity of an encrypted remote.
rclone cryptdecode	Cryptdecode returns unencrypted file names.
rclone dedupe	Interactively find duplicate filenames and delete/rename them.
rclone delete	Remove the files in path.
rclone deletefile	Remove a single file from remote.
rclone gendocs	Output markdown docs for rclone to the directory supplied.
rclone gitannex	Speaks with git-annex over stdin/stdout.
rclone hashsum	Produces a hashsum file for all the objects in the path.
rclone link	Generate public link to file/folder.
rclone listremotes	List all the remotes in the config file and defined in environment variables.
rclone ls	List the objects in the path with size and path.
rclone lsd	List all directories/containers/buckets in the path.
rclone lsf	List directories and objects in remote:path formatted for parsing.
rclone lsjson	List directories and objects in the path in JSON format.
rclone lsl	List the objects in path with modification time, size and path.
rclone md5sum	Produces an md5sum file for all the objects in the path.
rclone mkdir	Make the path if it doesn't already exist.
rclone mount	Mount the remote as file system on a mountpoint.
rclone move	Move files from source to dest.
rclone moveto	Move file or directory from source to dest.
rclone ncdu	Explore a remote with a text based user interface.
rclone nfsmount	Mount the remote as file system on a mountpoint.
rclone obscure	Obscure password for use in the rclone config file.
rclone purge	Remove the path and all of its contents.
rclone rc	Run a command against a running rclone.
rclone rcat	Copies standard input to file on remote.
rclone rcd	Run rclone listening to remote control commands only.
rclone rmdir	Remove the empty directory at path.

```
rclone rmdirs                Remove empty directories under the path.
rclone selfupdate            Update the rclone binary.
rclone serve                 Serve a remote over a protocol.
rclone serve dlna            Serve remote:path over DLNA
rclone serve docker          Serve any remote on docker's volume plugin API.
rclone serve ftp             Serve remote:path over FTP.
rclone serve http            Serve the remote over HTTP.
rclone serve nfs             Serve the remote as an NFS mount
rclone serve restic          Serve the remote for restic's REST API.
rclone serve s3              Serve remote:path over s3.
rclone serve sftp            Serve the remote over SFTP.
rclone serve webdav          Serve remote:path over WebDAV.
rclone settier               Changes storage class/tier of objects in remote.
rclone sha1sum               Produces an sha1sum file for all the objects in the path.
rclone size                  Prints the total size and number of objects in remote:path.
rclone sync                  Make source and dest identical, modifying destination only.
rclone test                  Run a test command
rclone test changenotify     Log any change notify requests for the remote passed in.
rclone test histogram        Makes a histogram of file name characters.
rclone test info             Discovers file name or other limitations for paths.
rclone test makefile         Make files with random contents of the size given
rclone test makefiles        Make a random file hierarchy in a directory
rclone test memory           Load all the objects at remote:path into memory and report memory stats.
rclone touch                 Create new file or change file modification time.
rclone tree                  List the contents of the remote in a tree like fashion.
rclone version               Show the version number.
```

---
https://rclone.org/docs/

Rclone - documentation

Usage
   Rclone is a command line program to manage files on cloud storage. After download and
   install, continue here to learn how to use it: Initial configuration, what the basic
   syntax looks like, describes the various subcommands, the various options, and more.

Configure
   First, you'll need to configure rclone. As the object storage systems have quite complicated
   authentication these are kept in a config file. (See the --config entry for how to find the
   config file and choose its location.)

   The easiest way to make the config is to run rclone with the config option:
$> rclone config

Basic syntax
   Rclone syncs a directory tree from one storage system to another.

   Its syntax is like this
$> rclone subcommand [options] <parameters> <parameters...>

   A subcommand is a the rclone operation required, (e.g. sync, copy, ls).

   An option is a single letter flag (e.g. -v) or a group of single letter flags (e.g. -Pv) or a long
   flag (e.g. --progress). No options are required. Options can come after the subcommand or in between
   parameters too or on the end, but only global options can be used before the subcommand. Anything
   after a -- option will not be interpreted as an option so if you need to add a parameter which starts
   with a - then put a -- on its own first, eg
$> rclone lsf -- -directory-starting-with-dash

   A parameter is usually a file path or rclone remote, eg /path/to/file or remote:path/to/file but
   it can be other things - the subcommand help will tell you what.

   Source and destination paths are specified by the name you gave the storage system in the config file
   then the sub path, e.g. "drive:myfolder" to look at "myfolder" in Google drive.

   You can define as many storage paths as you like in the config file.

   Please use the --interactive/-i flag while learning rclone to avoid accidental data loss.

Subcommands
   rclone uses a system of subcommands. For example
$> rclone ls remote:path # lists a remote
$> rclone copy /local/path remote:path # copies /local/path to the remote
$> rclone sync --interactive /local/path remote:path # syncs /local/path to the remote

   The main rclone commands with most used first
     * rclone config - Enter an interactive configuration session.
     * rclone copy - Copy files from source to dest, skipping already copied.
     * rclone sync - Make source and dest identical, modifying destination only.
     * rclone bisync - Bidirectional synchronization between two paths.
     * rclone move - Move files from source to dest.
     * rclone delete - Remove the contents of path.
     * rclone purge - Remove the path and all of its contents.

```
     * rclone mkdir - Make the path if it doesn't already exist.
     * rclone rmdir - Remove the path.
     * rclone rmdirs - Remove any empty directories under the path.
     * rclone check - Check if the files in the source and destination match.
     * rclone ls - List all the objects in the path with size and path.
     * rclone lsd - List all directories/containers/buckets in the path.
     * rclone lsl - List all the objects in the path with size, modification time and path.
     * rclone md5sum - Produce an md5sum file for all the objects in the path.
     * rclone sha1sum - Produce a sha1sum file for all the objects in the path.
     * rclone size - Return the total size and number of objects in remote:path.
     * rclone version - Show the version number.
     * rclone cleanup - Clean up the remote if possible.
     * rclone dedupe - Interactively find duplicate files and delete/rename them.
     * rclone authorize - Remote authorization.
     * rclone cat - Concatenate any files and send them to stdout.
     * rclone copyto - Copy files from source to dest, skipping already copied.
     * rclone completion - Output shell completion scripts for rclone.
     * rclone gendocs - Output markdown docs for rclone to the directory supplied.
     * rclone listremotes - List all the remotes in the config file.
     * rclone mount - Mount the remote as a mountpoint.
     * rclone moveto - Move file or directory from source to dest.
     * rclone obscure - Obscure password for use in the rclone.conf
     * rclone cryptcheck - Check the integrity of an encrypted remote.
     * rclone about - Get quota information from the remote.

Copying single files
   rclone normally syncs or copies directories. However, if the source remote points to a file, rclone
   will just copy that file. The destination remote must point to a directory - rclone will give the
   error Failed to create file system for "remote:file": is a file not a directory if it isn't.

   For example, suppose you have a remote with a file in called test.jpg, then you could copy just that
   file like this
$> rclone copy remote:test.jpg /tmp/download

   The file test.jpg will be placed inside /tmp/download.

   This is equivalent to specifying
$> rclone copy --files-from /tmp/files remote: /tmp/download

   Where /tmp/files contains the single line
test.jpg

   It is recommended to use copy when copying individual files, not sync. They have pretty much the same
   effect but copy will use a lot less memory.

Syntax of remote paths
   The syntax of the paths passed to the rclone command are as follows.

/path/to/dir

   This refers to the local file system.

   On Windows \ may be used instead of / in local paths only, non local paths must use /. See local
   filesystem documentation for more about Windows-specific paths.

   These paths needn't start with a leading / - if they don't then they will be relative to the current
   directory.

remote:path/to/dir

   This refers to a directory path/to/dir on remote: as defined in the config file (configured with
   rclone config).

remote:/path/to/dir

   On most backends this is refers to the same directory as remote:path/to/dir and that format should be
   preferred. On a very small number of remotes (FTP, SFTP, Dropbox for business) this will refer to a
   different directory. On these, paths without a leading / will refer to your "home" directory and
   paths with a leading / will refer to the root.

:backend:path/to/dir

   This is an advanced form for creating remotes on the fly. backend should be the name or prefix of a
   backend (the type in the config file) and all the configuration for the backend should be provided on
   the command line (or in environment variables).

   Here are some examples:
$> rclone lsd --http-url https://pub.rclone.org :http:

   To list all the directories in the root of https://pub.rclone.org/.
$> rclone lsf --http-url https://example.com :http:path/to/dir

   To list files and directories in https://example.com/path/to/dir/
$> rclone copy --http-url https://example.com :http:path/to/dir /tmp/dir

   To copy files and directories in https://example.com/path/to/dir to /tmp/dir.
```

```
$> rclone copy --sftp-host example.com :sftp:path/to/dir /tmp/dir

   To copy files and directories from example.com in the relative directory path/to/dir to /tmp/dir
   using sftp.

Connection strings
   The above examples can also be written using a connection string syntax, so instead of providing the
   arguments as command line parameters --http-url https://pub.rclone.org they are provided as part of
   the remote specification as a kind of connection string.
$> rclone lsd ":http,url='https://pub.rclone.org':"
$> rclone lsf ":http,url='https://example.com':path/to/dir"
$> rclone copy ":http,url='https://example.com':path/to/dir" /tmp/dir
$> rclone copy :sftp,host=example.com:path/to/dir /tmp/dir

   These can apply to modify existing remotes as well as create new remotes with the on the fly syntax.
   This example is equivalent to adding the --drive-shared-with-me parameter to the remote gdrive:.
$> rclone lsf "gdrive,shared_with_me:path/to/dir"

   The major advantage to using the connection string style syntax is that it only applies to the
   remote, not to all the remotes of that type of the command line. A common confusion is this attempt
   to copy a file shared on google drive to the normal drive which does not work because the
   --drive-shared-with-me flag applies to both the source and the destination.
$> rclone copy --drive-shared-with-me gdrive:shared-file.txt gdrive:

   However using the connection string syntax, this does work.
$> rclone copy "gdrive,shared_with_me:shared-file.txt" gdrive:

   Note that the connection string only affects the options of the immediate backend. If for example
   gdriveCrypt is a crypt based on gdrive, then the following command will not work as intended, because
   shared_with_me is ignored by the crypt backend:
$> rclone copy "gdriveCrypt,shared_with_me:shared-file.txt" gdriveCrypt:

   The connection strings have the following syntax
remote,parameter=value,parameter2=value2:path/to/dir
:backend,parameter=value,parameter2=value2:path/to/dir

   If the parameter has a : or , then it must be placed in quotes " or ', so
remote,parameter="colon:value",parameter2="comma,value":path/to/dir
:backend,parameter='colon:value',parameter2='comma,value':path/to/dir

   If a quoted value needs to include that quote, then it should be doubled, so
remote,parameter="with""quote",parameter2='with''quote':path/to/dir

   This will make parameter be with"quote and parameter2 be with'quote.

   If you leave off the =parameter then rclone will substitute =true which works very well with flags.
   For example, to use s3 configured in the environment you could use:
$> rclone lsd :s3,env_auth:

   Which is equivalent to
$> rclone lsd :s3,env_auth=true:

   Note that on the command line you might need to surround these connection strings with " or ' to stop
   the shell interpreting any special characters within them.

   If you are a shell master then you'll know which strings are OK and which aren't, but if you aren't
   sure then enclose them in " and use ' as the inside quote. This syntax works on all OSes.
$> rclone copy ":http,url='https://example.com':path/to/dir" /tmp/dir

   On Linux/macOS some characters are still interpreted inside " strings in the shell (notably \ and $
   and ") so if your strings contain those you can swap the roles of " and ' thus. (This syntax does not
   work on Windows.)
$> rclone copy ':http,url="https://example.com":path/to/dir' /tmp/dir

Connection strings, config and logging
   If you supply extra configuration to a backend by command line flag, environment variable or
   connection string then rclone will add a suffix based on the hash of the config to the name of the
   remote, eg
$> rclone -vv lsf --s3-chunk-size 20M s3:

   Has the log message
DEBUG : s3: detected overridden config - adding "{Srj1p}" suffix to name

   This is so rclone can tell the modified remote apart from the unmodified remote when caching the
   backends.

   This should only be noticeable in the logs.

   This means that on the fly backends such as
$> rclone -vv lsf :s3,env_auth:

   Will get their own names
DEBUG : :s3: detected overridden config - adding "{YTu53}" suffix to name

Valid remote names
   Remote names are case sensitive, and must adhere to the following rules:
```

```
     * May contain number, letter, _, -, ., +, @ and space.
     * May not start with - or space.
     * May not end with space.

   Starting with rclone version 1.61, any Unicode numbers and letters are allowed, while in older
   versions it was limited to plain ASCII (0-9, A-Z, a-z). If you use the same rclone configuration from
   different shells, which may be configured with different character encoding, you must be cautious to
   use characters that are possible to write in all of them. This is mostly a problem on Windows, where
   the console traditionally uses a non-Unicode character set - defined by the so-called "code page".

   Do not use single character names on Windows as it creates ambiguity with Windows drives' names,
   e.g.: remote called C is indistinguishable from C drive. Rclone will always assume that single letter
   name refers to a drive.

Quoting and the shell
   When you are typing commands to your computer you are using something called the command line shell.
   This interprets various characters in an OS specific way.

   Here are some gotchas which may help users unfamiliar with the shell rules

Linux / OSX
   If your names have spaces or shell metacharacters (e.g. *, ?, $, ', ", etc.) then you must quote
   them. Use single quotes ' by default.
$> rclone copy 'Important files?' remote:backup

   If you want to send a ' you will need to use ", e.g.
$> rclone copy "O'Reilly Reviews" remote:backup

   The rules for quoting metacharacters are complicated and if you want the full details you'll have to
   consult the manual page for your shell.

Windows
   If your names have spaces in you need to put them in ", e.g.
$> rclone copy "E:\folder name\folder name\folder name" remote:backup

   If you are using the root directory on its own then don't quote it (see #464 for why), e.g.
$> rclone copy E:\ remote:backup

Copying files or directories with : in the names
   rclone uses : to mark a remote name. This is, however, a valid filename component in non-Windows
   OSes. The remote name parser will only search for a : up to the first / so if you need to act on a
   file or directory like this then use the full path starting with a /, or use ./ as a current
   directory prefix.

   So to sync a directory called sync:me to a remote called remote: use
$> rclone sync --interactive ./sync:me remote:path

   or
$> rclone sync --interactive /full/path/to/sync:me remote:path

Server Side Copy
   Most remotes (but not all - see the overview) support server-side copy.

   This means if you want to copy one folder to another then rclone won't download all the files and
   re-upload them; it will instruct the server to copy them in place.

   Eg
$> rclone copy s3:oldbucket s3:newbucket

   Will copy the contents of oldbucket to newbucket without downloading and re-uploading.

   Remotes which don't support server-side copy will download and re-upload in this case.

   Server side copies are used with sync and copy and will be identified in the log when using the -v
   flag. The move command may also use them if remote doesn't support server-side move directly. This is
   done by issuing a server-side copy then a delete which is much quicker than a download and re-upload.

   Server side copies will only be attempted if the remote names are the same.

   This can be used when scripting to make aged backups efficiently, e.g.
$> rclone sync --interactive remote:current-backup remote:previous-backup
$> rclone sync --interactive /path/to/files remote:current-backup

Metadata support
   Metadata is data about a file (or directory) which isn't the contents of the file (or directory).
   Normally rclone only preserves the modification time and the content (MIME) type where possible.

   Rclone supports preserving all the available metadata on files and directories when using the
   --metadata or -M flag.

   Exactly what metadata is supported and what that support means depends on the backend. Backends that
   support metadata have a metadata section in their docs and are listed in the features table (Eg
   local, s3)

   Some backends don't support metadata, some only support metadata on files and some support metadata
   on both files and directories.
```

Rclone only supports a one-time sync of metadata. This means that metadata will be synced from the source object to the destination object only when the source object has changed and needs to be re-uploaded. If the metadata subsequently changes on the source object without changing the object itself then it won't be synced to the destination object. This is in line with the way rclone syncs Content-Type without the --metadata flag.

Using --metadata when syncing from local to local will preserve file attributes such as file mode, owner, extended attributes (not Windows).

Note that arbitrary metadata may be added to objects using the --metadata-set key=value flag when the object is first uploaded. This flag can be repeated as many times as necessary.

The --metadata-mapper flag can be used to pass the name of a program in which can transform metadata when it is being copied from source to destination.

Rclone supports --metadata-set and --metadata-mapper when doing sever side Move and server side Copy, but not when doing server side DirMove (renaming a directory) as this would involve recursing into the directory. Note that you can disable DirMove with --disable DirMove and rclone will revert back to using Move for each individual object where --metadata-set and --metadata-mapper are supported.

Types of metadata

Metadata is divided into two type. System metadata and User metadata.

Metadata which the backend uses itself is called system metadata. For example on the local backend the system metadata uid will store the user ID of the file when used on a unix based platform.

Arbitrary metadata is called user metadata and this can be set however is desired.

When objects are copied from backend to backend, they will attempt to interpret system metadata if it is supplied. Metadata may change from being user metadata to system metadata as objects are copied between different backends. For example copying an object from s3 sets the content-type metadata. In a backend which understands this (like azureblob) this will become the Content-Type of the object. In a backend which doesn't understand this (like the local backend) this will become user metadata. However should the local object be copied back to s3, the Content-Type will be set correctly.

Metadata framework

Rclone implements a metadata framework which can read metadata from an object and write it to the object when (and only when) it is being uploaded.

This metadata is stored as a dictionary with string keys and string values.

There are some limits on the names of the keys (these may be clarified further in the future).

* must be lower case
* may be a-z 0-9 containing . - or _
* length is backend dependent

Each backend can provide system metadata that it understands. Some backends can also store arbitrary user metadata.

Where possible the key names are standardized, so, for example, it is possible to copy object metadata from s3 to azureblob for example and metadata will be translated appropriately.

Some backends have limits on the size of the metadata and rclone will give errors on upload if they are exceeded.

Metadata preservation

The goal of the implementation is to

1. Preserve metadata if at all possible
2. Interpret metadata if at all possible

The consequences of 1 is that you can copy an S3 object to a local disk then back to S3 losslessly. Likewise you can copy a local file with file attributes and xattrs from local disk to s3 and back again losslessly.

The consequence of 2 is that you can copy an S3 object with metadata to Azureblob (say) and have the metadata appear on the Azureblob object also.

Standard system metadata

Here is a table of standard system metadata which, if appropriate, a backend may implement.

key	description	example
mode	File type and mode: octal, unix style	0100664
uid	User ID of owner: decimal number	500
gid	Group ID of owner: decimal number	500
rdev	Device ID (if special file) => hexadecimal	0
atime	Time of last access: RFC 3339	2006-01-02T15:04:05.999999999Z07:00
mtime	Time of last modification: RFC 3339	2006-01-02T15:04:05.999999999Z07:00
btime	Time of file creation (birth): RFC 3339	2006-01-02T15:04:05.999999999Z07:00
utime	Time of file upload: RFC 3339	2006-01-02T15:04:05.999999999Z07:00
cache-control	Cache-Control header	no-cache
content-disposition	Content-Disposition header	inline
content-encoding	Content-Encoding header	gzip
content-language	Content-Language header	en-US
content-type	Content-Type header	text/plain

The metadata keys mtime and content-type will take precedence if supplied in the metadata over reading the Content-Type or modification time of the source object.

Hashes are not included in system metadata as there is a well defined way of reading those already.

Options
Rclone has a number of options to control its behaviour.

Options that take parameters can have the values passed in two ways, --option=value or --option value. However boolean (true/false) options behave slightly differently to the other options in that --boolean sets the option to true and the absence of the flag sets it to false. It is also possible to specify --boolean=false or --boolean=true. Note that --boolean false is not valid - this is parsed as --boolean and the false is parsed as an extra command line argument for rclone.

Options documented to take a stringArray parameter accept multiple values. To pass more than one value, repeat the option; for example: --include value1 --include value2.

Time or duration options
TIME or DURATION options can be specified as a duration string or a time string.

A duration string is a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as "300ms", "-1.5h" or "2h45m". Default units are seconds or the following abbreviations are valid:
* ms - Milliseconds
* s - Seconds
* m - Minutes
* h - Hours
* d - Days
* w - Weeks
* M - Months
* y - Years

These can also be specified as an absolute time in the following formats:
* RFC3339 - e.g. 2006-01-02T15:04:05Z or 2006-01-02T15:04:05+07:00
* ISO8601 Date and time, local timezone - 2006-01-02T15:04:05
* ISO8601 Date and time, local timezone - 2006-01-02 15:04:05
* ISO8601 Date - 2006-01-02 (YYYY-MM-DD)

Size options
Options which use SIZE use KiB (multiples of 1024 bytes) by default. However, a suffix of B for Byte, K for KiB, M for MiB, G for GiB, T for TiB and P for PiB may be used. These are the binary units, e.g. 1, 2**10, 2**20, 2**30 respectively.

--backup-dir=DIR
When using sync, copy or move any files which would have been overwritten or deleted are moved in their original hierarchy into this directory.

If --suffix is set, then the moved files will have the suffix added to them. If there is a file with the same path (after the suffix has been added) in DIR, then it will be overwritten.

The remote in use must support server-side move or copy and you must use the same remote as the destination of the sync. The backup directory must not overlap the destination directory without it being excluded by a filter rule.

For example
```
$> rclone sync --interactive /path/to/local remote:current --backup-dir remote:old
```

will sync /path/to/local to remote:current, but for any files which would have been updated or deleted will be stored in remote:old.

If running rclone from a script you might want to use today's date as the directory name passed to --backup-dir to store the old files, or you might want to pass --suffix with today's date.

See --compare-dest and --copy-dest.

--bind string
Local address to bind to for outgoing connections. This can be an IPv4 address (1.2.3.4), an IPv6 address (1234::789A) or host name. If the host name doesn't resolve or resolves to more than one IP address it will give an error.

You can use --bind 0.0.0.0 to force rclone to use IPv4 addresses and --bind ::0 to force rclone to use IPv6 addresses.

--bwlimit=BANDWIDTH_SPEC
This option controls the bandwidth limit. For example
```
--bwlimit 10M
```

would mean limit the upload and download bandwidth to 10 MiB/s. NB this is bytes per second not bits per second. To use a single limit, specify the desired bandwidth in KiB/s, or use a suffix B|K|M|G|T|P. The default is 0 which means to not limit bandwidth.

The upload and download bandwidth can be specified separately, as --bwlimit UP:DOWN, so
```
--bwlimit 10M:100k
```

would mean limit the upload bandwidth to 10 MiB/s and the download bandwidth to 100 KiB/s. Either limit can be "off" meaning no limit, so to just limit the upload bandwidth you would use
--bwlimit 10M:off

this would limit the upload bandwidth to 10 MiB/s but the download bandwidth would be unlimited.

When specified as above the bandwidth limits last for the duration of run of the rclone binary.

It is also possible to specify a "timetable" of limits, which will cause certain limits to be applied at certain times. To specify a timetable, format your entries as WEEKDAY-HH:MM,BANDWIDTH WEEKDAY-HH:MM,BANDWIDTH... where: WEEKDAY is optional element.
* BANDWIDTH can be a single number, e.g.100k or a pair of numbers for upload:download, e.g.10M:1M.
* WEEKDAY can be written as the whole word or only using the first 3 characters. It is optional.
* HH:MM is an hour from 00:00 to 23:59.

Entries can be separated by spaces or semicolons.

Note: Semicolons can be used as separators instead of spaces to avoid parsing issues in environments like Docker.

An example of a typical timetable to avoid link saturation during daytime working hours could be:

Using spaces as separators: --bwlimit "08:00,512k 12:00,10M 13:00,512k 18:00,30M 23:00,off"

Using semicolons as separators: --bwlimit "08:00,512k;12:00,10M;13:00,512k;18:00,30M;23:00,off"

In these examples, the transfer bandwidth will be set to 512 KiB/s at 8am every day. At noon, it will rise to 10 MiB/s, and drop back to 512 KiB/sec at 1pm. At 6pm, the bandwidth limit will be set to 30 MiB/s, and at 11pm it will be completely disabled (full speed). Anything between 11pm and 8am will remain unlimited.

An example of timetable with WEEKDAY could be:
Using spaces as separators: --bwlimit "Mon-00:00,512 Fri-23:59,10M Sat-10:00,1M Sun-20:00,off"

Using semicolons as separators: --bwlimit "Mon-00:00,512;Fri-23:59,10M;Sat-10:00,1M;Sun-20:00,off"

It means that, the transfer bandwidth will be set to 512 KiB/s on Monday. It will rise to 10 MiB/s before the end of Friday. At 10:00 on Saturday it will be set to 1 MiB/s. From 20:00 on Sunday it will be unlimited.

Timeslots without WEEKDAY are extended to the whole week. So this example:

--bwlimit "Mon-00:00,512 12:00,1M Sun-20:00,off"

Is equivalent to this:

--bwlimit "Mon-00:00,512Mon-12:00,1M Tue-12:00,1M Wed-12:00,1M Thu-12:00,1M Fri-12:00,1M Sat-12:00,1M Sun-12:00,1M Sun-20:00,off"

Bandwidth limit apply to the data transfer for all backends. For most backends the directory listing bandwidth is also included (exceptions being the non HTTP backends, ftp, sftp and storj).

Note that the units are Byte/s, not bit/s. Typically connections are measured in bit/s - to convert divide by 8. For example, let's say you have a 10 Mbit/s connection and you wish rclone to use half of it - 5 Mbit/s. This is 5/8 = 0.625 MiB/s so you would use a --bwlimit 0.625M parameter for rclone.

On Unix systems (Linux, macOS, ...) the bandwidth limiter can be toggled by sending a SIGUSR2 signal to rclone. This allows to remove the limitations of a long running rclone transfer and to restore it back to the value specified with --bwlimit quickly when needed. Assuming there is only one rclone instance running, you can toggle the limiter like this:
kill -SIGUSR2 $(pidof rclone)

If you configure rclone with a remote control then you can use change the bwlimit dynamically:
$> rclone rc core/bwlimit rate=1M

--bwlimit-file=BANDWIDTH_SPEC
This option controls per file bandwidth limit. For the options see the --bwlimit flag.

For example use this to allow no transfers to be faster than 1 MiB/s
--bwlimit-file 1M

This can be used in conjunction with --bwlimit.

Note that if a schedule is provided the file will use the schedule in effect at the start of the transfer.

--buffer-size=SIZE
Use this sized buffer to speed up file transfers. Each --transfer will use this much memory for buffering.

When using mount or cmount each open file descriptor will use this much memory for buffering. See the mount documentation for more details.

Set to 0 to disable the buffering for the minimum memory usage.

Note that the memory allocation of the buffers is influenced by the --use-mmap flag.

--cache-dir=DIR
   Specify the directory rclone will use for caching, to override the default.

   Default value is depending on operating system:
     * Windows %LocalAppData%\rclone, if LocalAppData is defined.
     * macOS $HOME/Library/Caches/rclone if HOME is defined.
     * Unix $XDG_CACHE_HOME/rclone if XDG_CACHE_HOME is defined, else $HOME/.cache/rclone if HOME is
       defined.
     * Fallback (on all OS) to $TMPDIR/rclone, where TMPDIR is the value from --temp-dir.

   You can use the config paths command to see the current value.

   Cache directory is heavily used by the VFS File Caching mount feature, but also by serve,
   GUI and other parts of rclone.

--check-first
   If this flag is set then in a sync, copy or move, rclone will do all the checks to see whether files
   need to be transferred before doing any of the transfers. Normally rclone would start running
   transfers as soon as possible.

   This flag can be useful on IO limited systems where transfers interfere with checking.

   It can also be useful to ensure perfect ordering when using --order-by.

   If both --check-first and --order-by are set when doing rclone move then rclone will use the transfer
   thread to delete source files which don't need transferring. This will enable perfect ordering of the
   transfers and deletes but will cause the transfer stats to have more items in than expected.

   Using this flag can use more memory as it effectively sets --max-backlog to infinite. This means that
   all the info on the objects to transfer is held in memory before the transfers start.

--checkers=N
   Originally controlling just the number of file checkers to run in parallel, e.g. by rclone copy. Now
   a fairly universal parallelism control used by rclone in several places.

   Note: checkers do the equality checking of files during a sync. For some storage systems (e.g. S3,
   Swift, Dropbox) this can take a significant amount of time so they are run in parallel.

   The default is to run 8 checkers in parallel. However, in case of slow-reacting backends you may need
   to lower (rather than increase) this default by setting --checkers to 4 or less threads. This is
   especially advised if you are experiencing backend server crashes during file checking phase (e.g. on
   subsequent or top-up backups where little or no file copying is done and checking takes up most of
   the time). Increase this setting only with utmost care, while monitoring your server health and file
   checking throughput.

-c, --checksum
   Normally rclone will look at modification time and size of files to see if they are equal. If you set
   this flag then rclone will check the file hash and size to determine if files are equal.

   This is useful when the remote doesn't support setting modified time and a more accurate sync is
   desired than just checking the file size.

   This is very useful when transferring between remotes which store the same hash type on the object,
   e.g. Drive and Swift. For details of which remotes support which hash type see the table in the
   overview section.

   Eg rclone --checksum sync s3:/bucket swift:/bucket would run much quicker than without the --checksum
   flag.

   When using this flag, rclone won't update mtimes of remote files if they are incorrect as it would
   normally.

--color WHEN
   Specify when colors (and other ANSI codes) should be added to the output.

   AUTO (default) only allows ANSI codes when the output is a terminal

   NEVER never allow ANSI codes

   ALWAYS always add ANSI codes, regardless of the output format (terminal or file)

--compare-dest=DIR
   When using sync, copy or move DIR is checked in addition to the destination for files. If a file
   identical to the source is found that file is NOT copied from source. This is useful to copy just
   files that have changed since the last backup.

   You must use the same remote as the destination of the sync. The compare directory must not overlap
   the destination directory.

   See --copy-dest and --backup-dir.

--config=CONFIG_FILE
   Specify the location of the rclone configuration file, to override the default. E.g. rclone config
   --config="rclone.conf".

The exact default is a bit complex to describe, due to changes introduced through different versions of rclone while preserving backwards compatibility, but in most cases it is as simple as:
  * %APPDATA%/rclone/rclone.conf on Windows
  * 🏠/.config/rclone/rclone.conf on other

The complete logic is as follows: Rclone will look for an existing configuration file in any of the following locations, in priority order:
 1. rclone.conf (in program directory, where rclone executable is)
 2. %APPDATA%/rclone/rclone.conf (only on Windows)
 3. $XDG_CONFIG_HOME/rclone/rclone.conf (on all systems, including Windows)
 4. 🏠/.config/rclone/rclone.conf (see below for explanation of 🏠 symbol)
 5. 🏠/.rclone.conf

If no existing configuration file is found, then a new one will be created in the following location:
  * On Windows: Location 2 listed above, except in the unlikely event that APPDATA is not defined, then location 4 is used instead.
  * On Unix: Location 3 if XDG_CONFIG_HOME is defined, else location 4.
  * Fallback to location 5 (on all OS), when the rclone directory cannot be created, but if also a home directory was not found then path .rclone.conf relative to current working directory will be used as a final resort.

The 🏠 symbol in paths above represent the home directory of the current user on any OS, and the value is defined as following:
  * On Windows: %HOME% if defined, else %USERPROFILE%, or else %HOMEDRIVE%\%HOMEPATH%.
  * On Unix: $HOME if defined, else by looking up current user in OS-specific user database (e.g. passwd file), or else use the result from shell command cd && pwd.

If you run rclone config file you will see where the default location is for you. Running rclone config touch will ensure a configuration file exists, creating an empty one in the default location if there is none.

The fact that an existing file rclone.conf in the same directory as the rclone executable is always preferred, means that it is easy to run in "portable" mode by downloading rclone executable to a writable directory and then create an empty file rclone.conf in the same directory.

If the location is set to empty string "" or path to a file with name notfound, or the os null device represented by value NUL on Windows and /dev/null on Unix systems, then rclone will keep the configuration file in memory only.

You may see a log message "Config file not found - using defaults" if there is no configuration file. This can be supressed, e.g. if you are using rclone entirely with on the fly remotes, by using memory-only configuration file or by creating an empty configuration file, as described above.

The file format is basic INI: Sections of text, led by a [section] header and followed by key=value entries on separate lines. In rclone each remote is represented by its own section, where the section name defines the name of the remote. Options are specified as the key=value entries, where the key is the option name without the --backend- prefix, in lowercase and with _ instead of -. E.g. option --mega-hard-delete corresponds to key hard_delete. Only backend options can be specified. A special, and required, key type identifies the storage system, where the value is the internal lowercase name as returned by command rclone help backends. Comments are indicated by ; or # at the beginning of a line.

Example:
```
[megaremote]
type = mega
user = you@example.com
pass = PDPcQVVjVtzFY-GTdDFozqBhTdsPg3qH
```

Note that passwords are in obscured form. Also, many storage systems uses token-based authentication instead of passwords, and this requires additional steps. It is easier, and safer, to use the interactive command rclone config instead of manually editing the configuration file.

The configuration file will typically contain login information, and should therefore have restricted permissions so that only the current user can read it. Rclone tries to ensure this when it writes the file. You may also choose to encrypt the file.

When token-based authentication are used, the configuration file must be writable, because rclone needs to update the tokens inside it.

To reduce risk of corrupting an existing configuration file, rclone will not write directly to it when saving changes. Instead it will first write to a new, temporary, file. If a configuration file already existed, it will (on Unix systems) try to mirror its permissions to the new file. Then it will rename the existing file to a temporary name as backup. Next, rclone will rename the new file to the correct name, before finally cleaning up by deleting the backup file.

If the configuration file path used by rclone is a symbolic link, then this will be evaluated and rclone will write to the resolved path, instead of overwriting the symbolic link. Temporary files used in the process (described above) will be written to the same parent directory as that of the resolved configuration file, but if this directory is also a symbolic link it will not be resolved and the temporary files will be written to the location of the directory symbolic link.

--contimeout=TIME
Set the connection timeout. This should be in go time format which looks like 5s for 5 seconds, 10m for 10 minutes, or 3h30m.

The connection timeout is the amount of time rclone will wait for a connection to go through to a

remote object storage system. It is 1m by default.

--copy-dest=DIR
When using sync, copy or move DIR is checked in addition to the destination for files. If a file identical to the source is found that file is server-side copied from DIR to the destination. This is useful for incremental backup.

The remote in use must support server-side copy and you must use the same remote as the destination of the sync. The compare directory must not overlap the destination directory.

See --compare-dest and --backup-dir.

--dedupe-mode MODE
Mode to run dedupe command in. One of interactive, skip, first, newest, oldest, rename. The default is interactive.
See the dedupe command for more information as to what these options mean.

--default-time TIME
If a file or directory does have a modification time rclone can read then rclone will display this fixed time instead.

The default is 2000-01-01 00:00:00 UTC. This can be configured in any of the ways shown in the time or duration options.

For example --default-time 2020-06-01 to set the default time to the 1st of June 2020 or --default-time 0s to set the default time to the time rclone started up.

--disable FEATURE,FEATURE,...
This disables a comma separated list of optional features. For example to disable server-side move and server-side copy use:
--disable move,copy

The features can be put in any case.

To see a list of which features can be disabled use:
--disable help

The features a remote has can be seen in JSON format with:
$> rclone backend features remote:

See the overview features and optional features to get an idea of which feature does what.

Note that some features can be set to true if they are true/false feature flag features by prefixing them with !. For example the CaseInsensitive feature can be forced to false with --disable CaseInsensitive and forced to true with --disable '!CaseInsensitive'. In general it isn't a good idea doing this but it may be useful in extremis.

(Note that ! is a shell command which you will need to escape with single quotes or a backslash on unix like platforms.)

This flag can be useful for debugging and in exceptional circumstances (e.g. Google Drive limiting the total volume of Server Side Copies to 100 GiB/day).

--disable-http2
This stops rclone from trying to use HTTP/2 if available. This can sometimes speed up transfers due to a problem in the Go standard library.

--dscp VALUE
Specify a DSCP value or name to use in connections. This could help QoS system to identify traffic class. BE, EF, DF, LE, CSx and AFxx are allowed.

See the description of differentiated services to get an idea of this field. Setting this to 1 (LE) to identify the flow to SCAVENGER class can avoid occupying too much bandwidth in a network with DiffServ support (RFC 8622).

For example, if you configured QoS on router to handle LE properly. Running:
$> rclone copy --dscp LE from:/from to:/to

would make the priority lower than usual internet flows.

This option has no effect on Windows (see golang/go#42728).

-n, --dry-run
Do a trial run with no permanent changes. Use this to see what rclone would do without actually doing it. Useful when setting up the sync command which deletes files in the destination.

--expect-continue-timeout=TIME
the request headers if the request has an "Expect: 100-continue" header. Not all backends support using this.

Zero means no timeout and causes the body to be sent immediately, without waiting for the server to approve. This time does not include the time to send the request header.

The default is 1s. Set to 0 to disable.

--error-on-no-transfer

By default, rclone will exit with return code 0 if there were no errors.

This option allows rclone to return exit code 9 if no files were transferred between the source and destination. This allows using rclone in scripts, and triggering follow-on actions if data was copied, or skipping if not.

NB: Enabling this option turns a usually non-fatal error into a potentially fatal one - please check and adjust your scripts accordingly!

--fix-case
Normally, a sync to a case insensitive dest (such as macOS / Windows) will not result in a matching filename if the source and dest filenames have casing differences but are otherwise identical. For example, syncing hello.txt to HELLO.txt will normally result in the dest filename remaining HELLO.txt. If --fix-case is set, then HELLO.txt will be renamed to hello.txt to match the source.

NB:
* directory names with incorrect casing will also be fixed
* --fix-case will be ignored if --immutable is set
* using --local-case-sensitive instead is not advisable; it will cause HELLO.txt to get deleted!
* the old dest filename must not be excluded by filters. Be especially careful with --files-from, which does not respect --ignore-case!
* on remotes that do not support server-side move, --fix-case will require downloading the file and re-uploading it. To avoid this, do not use --fix-case.

--fs-cache-expire-duration=TIME
When using rclone via the API rclone caches created remotes for 5 minutes by default in the "fs cache". This means that if you do repeated actions on the same remote then rclone won't have to build it again from scratch, which makes it more efficient.

This flag sets the time that the remotes are cached for. If you set it to 0 (or negative) then rclone won't cache the remotes at all.

Note that if you use some flags, eg --backup-dir and if this is set to 0 rclone may build two remotes (one for the source or destination and one for the --backup-dir where it may have only built one before.

--fs-cache-expire-interval=TIME
This controls how often rclone checks for cached remotes to expire. See the --fs-cache-expire-duration documentation above for more info. The default is 60s, set to 0 to disable expiry.

--header
Add an HTTP header for all transactions. The flag can be repeated to add multiple headers.

If you want to add headers only for uploads use --header-upload and if you want to add headers only for downloads use --header-download.

This flag is supported for all HTTP based backends even those not supported by --header-upload and --header-download so may be used as a workaround for those with care.
$> rclone ls remote:test --header "X-Rclone: Foo" --header "X-LetMeIn: Yes"

--header-download
Add an HTTP header for all download transactions. The flag can be repeated to add multiple headers.
$> rclone sync --interactive s3:test/src □/dst --header-download "X-Amz-Meta-Test: Foo" --header-download "X-Am
z-
Meta-Test2: Bar"

See the GitHub issue here for currently supported backends.

--header-upload
Add an HTTP header for all upload transactions. The flag can be repeated to add multiple headers.
$> rclone sync --interactive □/src s3:test/dst --header-upload "Content-Disposition: attachment; filename='cool
.h
tml'" --header-upload "X-Amz-Meta-Test: FooBar"

See the GitHub issue here for currently supported backends.

--human-readable
Rclone commands output values for sizes (e.g. number of bytes) and counts (e.g. number of files) either as raw numbers, or in human-readable format.

In human-readable format the values are scaled to larger units, indicated with a suffix shown after the value, and rounded to three decimals. Rclone consistently uses binary units (powers of 2) for sizes and decimal units (powers of 10) for counts. The unit prefix for size is according to IEC standard notation, e.g. Ki for kibi. Used with byte unit, 1 KiB means 1024 Byte. In list type of output, only the unit prefix appended to the value (e.g. 9.762Ki), while in more textual output the full unit is shown (e.g. 9.762 KiB). For counts the SI standard notation is used, e.g. prefix k for kilo. Used with file counts, 1k means 1000 files.

The various list commands output raw numbers by default. Option --human-readable will make them output values in human-readable format instead (with the short unit prefix).

The about command outputs human-readable by default, with a command-specific option --full to output the raw numbers instead.

Command size outputs both human-readable and raw numbers in the same output.

The tree command also considers --human-readable, but it will not use the exact same notation as the other commands: It rounds to one decimal, and uses single letter suffix, e.g. K instead of Ki. The reason for this is that it relies on an external library.

The interactive command ncdu shows human-readable by default, and responds to key u for toggling human-readable format.

--ignore-case-sync
Using this option will cause rclone to ignore the case of the files when synchronizing so files will not be copied/synced when the existing filenames are the same, even if the casing is different.

--ignore-checksum
Normally rclone will check that the checksums of transferred files match, and give an error "corrupted on transfer" if they don't.

You can use this option to skip that check. You should only use it if you have had the "corrupted on transfer" error message and you are sure you might want to transfer potentially corrupted data.

--ignore-existing
Using this option will make rclone unconditionally skip all files that exist on the destination, no matter the content of these files.

While this isn't a generally recommended option, it can be useful in cases where your files change due to encryption. However, it cannot correct partial transfers in case a transfer was interrupted.

When performing a move/moveto command, this flag will leave skipped files in the source location unchanged when a file with the same name exists on the destination.

--ignore-size
Normally rclone will look at modification time and size of files to see if they are equal. If you set this flag then rclone will check only the modification time. If --checksum is set then it only checks the checksum.

It will also cause rclone to skip verifying the sizes are the same after transfer.

This can be useful for transferring files to and from OneDrive which occasionally misreports the size of image files (see #399 for more info).

-I, --ignore-times
Using this option will cause rclone to unconditionally upload all files regardless of the state of files on the destination.

Normally rclone would skip any files that have the same modification time and are the same size (or have the same checksum if using --checksum).

--immutable
Treat source and destination files as immutable and disallow modification.

With this option set, files will be created and deleted as requested, but existing files will never be updated. If an existing file does not match between the source and destination, rclone will give the error Source and destination exist but do not match: immutable file modified.

Note that only commands which transfer files (e.g. sync, copy, move) are affected by this behavior, and only modification is disallowed. Files may still be deleted explicitly (e.g. delete, purge) or implicitly (e.g. sync, move). Use copy --immutable if it is desired to avoid deletion as well as modification.

This can be useful as an additional layer of protection for immutable or append-only data sets (notably backup archives), where modification implies corruption and should not be propagated.

--inplace
The --inplace flag changes the behaviour of rclone when uploading files to some backends (backends with the PartialUploads feature flag set) such as:
  * local
  * ftp
  * sftp
  * pcloud

Without --inplace (the default) rclone will first upload to a temporary file with an extension like this, where XXXXXX represents a hash of the source file's fingerprint and .partial is --partial-suffix value (.partial by default).
original-file-name.XXXXXX.partial

(rclone will make sure the final name is no longer than 100 characters by truncating the original-file-name part if necessary).

When the upload is complete, rclone will rename the .partial file to the correct name, overwriting any existing file at that point. If the upload fails then the .partial file will be deleted.

This prevents other users of the backend from seeing partially uploaded files in their new names and prevents overwriting the old file until the new one is completely uploaded.

If the --inplace flag is supplied, rclone will upload directly to the final name without creating a .partial file.

This means that an incomplete file will be visible in the directory listings while the upload is in progress and any existing files will be overwritten as soon as the upload starts. If the transfer fails then the file will be deleted. This can cause data loss of the existing file if the transfer fails.

Note that on the local file system if you don't use --inplace hard links (Unix only) will be broken. And if you do use --inplace you won't be able to update in use executables.

Note also that versions of rclone prior to v1.63.0 behave as if the --inplace flag is always supplied.

-i, --interactive
This flag can be used to tell rclone that you wish a manual confirmation before destructive operations.

It is recommended that you use this flag while learning rclone especially with rclone sync.

For example
```
$> rclone delete --interactive /tmp/dir
rclone: delete "important-file.txt"?
y) Yes, this is OK (default)
n) No, skip this
s) Skip all delete operations with no more questions
!) Do all delete operations with no more questions
q) Exit rclone now.
y/n/s/!/q> n
```

The options mean
- y: Yes, this operation should go ahead. You can also press Return for this to happen. You'll be asked every time unless you choose s or !.
- n: No, do not do this operation. You'll be asked every time unless you choose s or !.
- s: Skip all the following operations of this type with no more questions. This takes effect until rclone exits. If there are any different kind of operations you'll be prompted for them.
- !: Do all the following operations with no more questions. Useful if you've decided that you don't mind rclone doing that kind of operation. This takes effect until rclone exits . If there are any different kind of operations you'll be prompted for them.
- q: Quit rclone now, just in case!

--leave-root
During rmdirs it will not remove root directory, even if it's empty.

--links / -l
Normally rclone will ignore symlinks or junction points (which behave like symlinks under Windows).

If you supply this flag then rclone will copy symbolic links from any supported backend backend, and store them as text files, with a .rclonelink suffix in the destination.

The text file will contain the target of the symbolic link.

The --links / -l flag enables this feature for all supported backends and the VFS. There are individual flags for just enabling it for the VFS --vfs-links and the local backend --local-links if required.

--list-cutoff N
When syncing rclone needs to sort directory entries before comparing them. Below this threshold (1,000,000) by default, rclone will store the directory entries in memory. 1,000,000 entries will take approx 1GB of RAM to store. Above this threshold rclone will store directory entries on disk and sort them without using a lot of memory.

Doing this is slightly less efficient then sorting them in memory and will only work well for the bucket based backends (eg s3, b2, azureblob, swift) but these are the only backends likely to have millions of entries in a directory.

--log-file=FILE
Log all of rclone's output to FILE. This is not active by default. This can be useful for tracking down problems with syncs in combination with the -v flag. See the Logging section for more info.

If FILE exists then rclone will append to it.

Note that if you are using the logrotate program to manage rclone's logs, then you should use the copytruncate option as rclone doesn't have a signal to rotate logs.

--log-format LIST
Comma separated list of log format options. The accepted options are:
- date - Add a date in the format YYYY/MM/YY to the log.
- time - Add a time to the log in format HH:MM:SS.
- microseconds - Add microseconds to the time in format HH:MM:SS.SSSSSS.
- UTC - Make the logs in UTC not localtime.
- longfile - Adds the source file and line number of the log statement.
- shortfile - Adds the source file and line number of the log statement.
- pid - Add the process ID to the log - useful with rclone mount --daemon.
- nolevel - Don't add the level to the log.
- json - Equivalent to adding --use-json-log

They are added to the log line in the order above.

The default log format is "date,time".

--log-level LEVEL
This sets the log level for rclone. The default log level is NOTICE.

DEBUG is equivalent to -vv. It outputs lots of debug info - useful for bug reports and really finding out what rclone is doing.

INFO is equivalent to -v. It outputs information about each transfer and prints stats once a minute by default.

NOTICE is the default log level if no logging flags are supplied. It outputs very little when things are working normally. It outputs warnings and significant events.

ERROR is equivalent to -q. It only outputs error messages.

--windows-event-log LEVEL
If this is configured (the default is OFF) then logs of this level and above will be logged to the Windows event log in addition to the normal logs. These will be logged in JSON format as described below regardless of what format the main logs are configured for.

The Windows event log only has 3 levels of severity Info, Warning and Error. If enabled we map rclone levels like this.
* Error <-- ERROR (and above)
* Warning <-- WARNING (note that this level is defined but not currently used).
* Info <-- NOTICE, INFO and DEBUG.

Rclone will declare its log source as "rclone" if it is has enough permissions to create the registry key needed. If not then logs will appear as "Application". You can run rclone version --windows-event-log DEBUG once as administrator to create the registry key in advance.

Note that the --windows-event-log level must be greater (more severe) than or equal to the --log-level. For example to log DEBUG to a log file but ERRORs to the event log you would use
--log-file rclone.log --log-level DEBUG --windows-event-log ERROR

This option is only supported Windows platforms.

--use-json-log
This switches the log format to JSON for rclone. The fields of JSON log are level, msg, source, time. The JSON logs will be printed on a single line, but are shown expanded here for clarity.

```
{
  "time": "2025-05-13T17:30:51.036237518+01:00",
  "level": "debug",
  "msg": "4 go routines active\n",
  "source": "cmd/cmd.go:298"
}
```

Completed data transfer logs will have extra size information. Logs which are about a particular object will have object and objectType fields also.

```
{
  "time": "2025-05-13T17:38:05.540846352+01:00",
  "level": "info",
  "msg": "Copied (new) to: file2.txt",
  "size": 6,
  "object": "file.txt",
  "objectType": "*local.Object",
  "source": "operations/copy.go:368"
}
```

Stats logs will contain a stats field which is the same as returned from the rc call core/stats.

```
{
  "time": "2025-05-13T17:38:05.540912847+01:00",
  "level": "info",
  "msg": "...text version of the stats...",
  "stats": {
    "bytes": 6,
    "checks": 0,
    "deletedDirs": 0,
    "deletes": 0,
    "elapsedTime": 0.000904825,
    ...truncated for clarity...
    "totalBytes": 6,
    "totalChecks": 0,
    "totalTransfers": 1,
    "transferTime": 0.000882794,
    "transfers": 1
  },
  "source": "accounting/stats.go:569"
}
```

--low-level-retries NUMBER
This controls the number of low level retries rclone does.

A low level retry is used to retry a failing operation - typically one HTTP request. This might be uploading a chunk of a big file for example. You will see low level retries in the log with the -v flag.

This shouldn't need to be changed from the default in normal operations. However, if you get a lot of low level retries you may wish to reduce the value so rclone moves on to a high level retry (see the --retries flag) quicker.

Disable low level retries with --low-level-retries 1.

--max-backlog=N
This is the maximum allowable backlog of files in a sync/copy/move queued for being checked or transferred.

This can be set arbitrarily large. It will only use memory when the queue is in use. Note that it will use in the order of N KiB of memory when the backlog is in use.

Setting this large allows rclone to calculate how many files are pending more accurately, give a more accurate estimated finish time and make --order-by work more accurately.

Setting this small will make rclone more synchronous to the listings of the remote which may be desirable.

Setting this to a negative number will make the backlog as large as possible.

--max-buffer-memory=SIZE
If set, don't allocate more than SIZE amount of memory as buffers. If not set or set to 0 or off this will not limit the amount of memory in use.

This includes memory used by buffers created by the --buffer flag and buffers used by multi-thread transfers.

Most multi-thread transfers do not take additional memory, but some do depending on the backend (eg the s3 backend for uploads). This means there is a tension between total setting --transfers as high as possible and memory use.

Setting --max-buffer-memory allows the buffer memory to be controlled so that it doesn't overwhelm the machine and allows --transfers to be set large.

--max-connections=N
This sets the maximum number of concurrent calls to the backend API. It may not map 1:1 to TCP or HTTP connections depending on the backend in use and the use of HTTP1 vs HTTP2.

When downloading files, backends only limit the initial opening of the stream. The bulk data download is not counted as a connection. This means that the --max--connections flag won't limit the total number of downloads.

Note that it is possible to cause deadlocks with this setting so it should be used with care.

If you are doing a sync or copy then make sure --max-connections is one more than the sum of --transfers and --checkers.

If you use --check-first then --max-connections just needs to be one more than the maximum of --checkers and --transfers.

So for --max-connections 3 you'd use --checkers 2 --transfers 2 --check-first or --checkers 1 --transfers 1.

Setting this flag can be useful for backends which do multipart uploads to limit the number of simultaneous parts being transferred.

--max-delete=N
This tells rclone not to delete more than N files. If that limit is exceeded then a fatal error will be generated and rclone will stop the operation in progress.

--max-delete-size=SIZE
Rclone will stop deleting files when the total size of deletions has reached the size specified. It defaults to off.

If that limit is exceeded then a fatal error will be generated and rclone will stop the operation in progress.

--max-depth=N
This modifies the recursion depth for all the commands except purge.

So if you do rclone --max-depth 1 ls remote:path you will see only the files in the top level directory. Using --max-depth 2 means you will see all the files in first two directory levels and so on.

For historical reasons the lsd command defaults to using a --max-depth of 1 - you can override this with the command line flag.

You can use this command to disable recursion (with --max-depth 1).

Note that if you use this with sync and --delete-excluded the files not recursed through are considered excluded and will be deleted on the destination. Test first with --dry-run if you are not sure what will happen.

--max-duration=TIME

Rclone will stop transferring when it has run for the duration specified. Defaults to off.

When the limit is reached all transfers will stop immediately. Use --cutoff-mode to modify this behaviour.

Rclone will exit with exit code 10 if the duration limit is reached.

--max-transfer=SIZE
Rclone will stop transferring when it has reached the size specified. Defaults to off.

When the limit is reached all transfers will stop immediately. Use --cutoff-mode to modify this behaviour.

Rclone will exit with exit code 8 if the transfer limit is reached.

--cutoff-mode=hard|soft|cautious
This modifies the behavior of --max-transfer and --max-duration Defaults to --cutoff-mode=hard.

Specifying --cutoff-mode=hard will stop transferring immediately when Rclone reaches the limit.

Specifying --cutoff-mode=soft will stop starting new transfers when Rclone reaches the limit.

Specifying --cutoff-mode=cautious will try to prevent Rclone from reaching the limit. Only applicable for --max-transfer

-M, --metadata

Setting this flag enables rclone to copy the metadata from the source to the destination. For local backends this is ownership, permissions, xattr etc. See the metadata section for more info.

--metadata-mapper SpaceSepList
If you supply the parameter --metadata-mapper /path/to/program then rclone will use that program to map metadata from source object to destination object.

The argument to this flag should be a command with an optional space separated list of arguments. If one of the arguments has a space in then enclose it in ", if you want a literal " in an argument then enclose the argument in " and double the ". See CSV encoding for more info.

```
--metadata-mapper "python bin/test_metadata_mapper.py"
--metadata-mapper 'python bin/test_metadata_mapper.py "argument with a space"'
--metadata-mapper 'python bin/test_metadata_mapper.py "argument with ""two"" quotes"'
```

This uses a simple JSON based protocol with input on STDIN and output on STDOUT. This will be called for every file and directory copied and may be called concurrently.

The program's job is to take a metadata blob on the input and turn it into a metadata blob on the output suitable for the destination backend.

Input to the program (via STDIN) might look like this. This provides some context for the Metadata which may be important.

* SrcFs is the config string for the remote that the object is currently on.
* SrcFsType is the name of the source backend.
* DstFs is the config string for the remote that the object is being copied to
* DstFsType is the name of the destination backend.
* Remote is the path of the object relative to the root.
* Size, MimeType, ModTime are attributes of the object.
* IsDir is true if this is a directory (not yet implemented).
* ID is the source ID of the object if known.
* Metadata is the backend specific metadata as described in the backend docs.

```
{
    "SrcFs": "gdrive:",
    "SrcFsType": "drive",
    "DstFs": "newdrive:user",
    "DstFsType": "onedrive",
    "Remote": "test.txt",
    "Size": 6,
    "MimeType": "text/plain; charset=utf-8",
    "ModTime": "2022-10-11T17:53:10.286745272+01:00",
    "IsDir": false,
    "ID": "xyz",
    "Metadata": {
        "btime": "2022-10-11T16:53:11Z",
        "content-type": "text/plain; charset=utf-8",
        "mtime": "2022-10-11T17:53:10.286745272+01:00",
        "owner": "user1@domain1.com",
        "permissions": "...",
        "description": "my nice file",
        "starred": "false"
    }
}
```

The program should then modify the input as desired and send it to STDOUT. The returned Metadata field will be used in its entirety for the destination object. Any other fields will be ignored. Note in this example we translate user names and permissions and add something to the description:

```
{
    "Metadata": {
```

```
        "btime": "2022-10-11T16:53:11Z",
        "content-type": "text/plain; charset=utf-8",
        "mtime": "2022-10-11T17:53:10.286745272+01:00",
        "owner": "user1@domain2.com",
        "permissions": "...",
        "description": "my nice file [migrated from domain1]",
        "starred": "false"
    }
}
```

Metadata can be removed here too.

An example python program might look something like this to implement the above transformations.

```
import sys, json

i = json.load(sys.stdin)
metadata = i["Metadata"]
# Add tag to description
if "description" in metadata:
    metadata["description"] += " [migrated from domain1]"
else:
    metadata["description"] = "[migrated from domain1]"
# Modify owner
if "owner" in metadata:
    metadata["owner"] = metadata["owner"].replace("domain1.com", "domain2.com")
o = { "Metadata": metadata }
json.dump(o, sys.stdout, indent="\t")
```

You can find this example (slightly expanded) in the rclone source code at bin/test_metadata_mapper.py.

If you want to see the input to the metadata mapper and the output returned from it in the log you can use -vv --dump mapper.

See the metadata section for more info.

--metadata-set key=value
Add metadata key = value when uploading. This can be repeated as many times as required. See the metadata section for more info.

--modify-window=TIME
When checking whether a file has been modified, this is the maximum allowed time difference that a file can have and still be considered equivalent.

The default is 1ns unless this is overridden by a remote. For example OS X only stores modification times to the nearest second so if you are reading and writing to an OS X filing system this will be 1s by default.

This command line flag allows you to override that computed default.

--multi-thread-write-buffer-size=SIZE
When transferring with multiple threads, rclone will buffer SIZE bytes in memory before writing to disk for each thread.

This can improve performance if the underlying filesystem does not deal well with a lot of small writes in different positions of the file, so if you see transfers being limited by disk write speed, you might want to experiment with different values. Specially for magnetic drives and remote file systems a higher value can be useful.

Nevertheless, the default of 128k should be fine for almost all use cases, so before changing it ensure that network is not really your bottleneck.

As a final hint, size is not the only factor: block size (or similar concept) can have an impact. In one case, we observed that exact multiples of 16k performed much better than other values.

--multi-thread-chunk-size=SizeSuffix
Normally the chunk size for multi thread transfers is set by the backend. However some backends such as local and smb (which implement OpenWriterAt but not OpenChunkWriter) don't have a natural chunk size.

In this case the value of this option is used (default 64Mi).

--multi-thread-cutoff=SIZE
When transferring files above SIZE to capable backends, rclone will use multiple threads to transfer the file (default 256M).

Capable backends are marked in the overview as MultithreadUpload. (They need to implement either the OpenWriterAt or OpenChunkWriter internal interfaces). These include include, local, s3, azureblob, b2, oracleobjectstorage and smb at the time of writing.

On the local disk, rclone preallocates the file (using fallocate(FALLOC_FL_KEEP_SIZE) on unix or NTSetInformationFile on Windows both of which takes no time) then each thread writes directly into the file at the correct place. This means that rclone won't create fragmented or sparse files and there won't be any assembly time at the end of the transfer.

The number of threads used to transfer is controlled by --multi-thread-streams.

Use -vv if you wish to see info about the threads.

This will work with the sync/copy/move commands and friends copyto/moveto. Multi thread transfers will be used with rclone mount and rclone serve if --vfs-cache-mode is set to writes or above.

Most multi-thread transfers do not take additional memory, but some do (for example uploading to s3). In the worst case memory usage can be at maximum --transfers * --multi-thread-chunk-size * --multi-thread-streams or specifically for the s3 backend --transfers * --s3-chunk-size * --s3-concurrency. However you can use the the --max-buffer-memory flag to control the maximum memory used here.

NB that this only works with supported backends as the destination but will work with any backend as the source.

NB that multi-thread copies are disabled for local to local copies as they are faster without unless --multi-thread-streams is set explicitly.

NB on Windows using multi-thread transfers to the local disk will cause the resulting files to be sparse. Use --local-no-sparse to disable sparse files (which may cause long delays at the start of transfers) or disable multi-thread transfers with --multi-thread-streams 0

--multi-thread-streams=N
When using multi thread transfers (see above --multi-thread-cutoff) this sets the number of streams to use. Set to 0 to disable multi thread transfers (Default 4).

If the backend has a --backend-upload-concurrency setting (eg --s3-upload-concurrency) then this setting will be used as the number of transfers instead if it is larger than the value of --multi-thread-streams or --multi-thread-streams isn't set.

--name-transform COMMAND[=XXXX]
--name-transform introduces path name transformations for rclone copy, rclone sync, and rclone move. These transformations enable modifications to source and destination file names by applying prefixes, suffixes, and other alterations during transfer operations. For detailed docs and examples, see convmv.

--no-check-dest
The --no-check-dest can be used with move or copy and it causes rclone not to check the destination at all when copying files.

This means that:
* the destination is not listed minimising the API calls
* files are always transferred
* this can cause duplicates on remotes which allow it (e.g. Google Drive)
* --retries 1 is recommended otherwise you'll transfer everything again on a retry

This flag is useful to minimise the transactions if you know that none of the files are on the destination.

This is a specialized flag which should be ignored by most users!

--no-gzip-encoding
Don't set Accept-Encoding: gzip. This means that rclone won't ask the server for compressed files automatically. Useful if you've set the server to return files with Content-Encoding: gzip but you uploaded compressed files.

There is no need to set this in normal operation, and doing so will decrease the network transfer efficiency of rclone.

--no-traverse
The --no-traverse flag controls whether the destination file system is traversed when using the copy or move commands. --no-traverse is not compatible with sync and will be ignored if you supply it with sync.

If you are only copying a small number of files (or are filtering most of the files) and/or have a large number of files on the destination then --no-traverse will stop rclone listing the destination and save time.

However, if you are copying a large number of files, especially if you are doing a copy where lots of the files under consideration haven't changed and won't need copying then you shouldn't use --no-traverse.

See rclone copy for an example of how to use it.

--no-unicode-normalization
Don't normalize unicode characters in filenames during the sync routine.

Sometimes, an operating system will store filenames containing unicode parts in their decomposed form (particularly macOS). Some cloud storage systems will then recompose the unicode, resulting in duplicate files if the data is ever copied back to a local filesystem.

Using this flag will disable that functionality, treating each unicode character as unique. For example, by default [e/] and [e'] will be normalized into the same character. With --no-unicode-normalization they will be treated as unique characters.

--no-update-modtime

When using this flag, rclone won't update modification times of remote files if they are incorrect as it would normally.

This can be used if the remote is being synced with another tool also (e.g. the Google Drive client).

--no-update-dir-modtime
When using this flag, rclone won't update modification times of remote directories if they are incorrect as it would normally.

--order-by string
The --order-by flag controls the order in which files in the backlog are processed in rclone sync, rclone copy and rclone move.

The order by string is constructed like this. The first part describes what aspect is being measured:
* size - order by the size of the files
* name - order by the full path of the files
* modtime - order by the modification date of the files

This can have a modifier appended with a comma:
* ascending or asc - order so that the smallest (or oldest) is processed first
* descending or desc - order so that the largest (or newest) is processed first
* mixed - order so that the smallest is processed first for some threads and the largest for others

If the modifier is mixed then it can have an optional percentage (which defaults to 50), e.g. size,mixed,25 which means that 25% of the threads should be taking the smallest items and 75% the largest. The threads which take the smallest first will always take the smallest first and likewise the largest first threads. The mixed mode can be useful to minimise the transfer time when you are transferring a mixture of large and small files - the large files are guaranteed upload threads and bandwidth and the small files will be processed continuously.

If no modifier is supplied then the order is ascending.

For example
* --order-by size,desc - send the largest files first
* --order-by modtime,ascending - send the oldest files first
* --order-by name - send the files with alphabetically by path first

If the --order-by flag is not supplied or it is supplied with an empty string then the default ordering will be used which is as scanned. With --checkers 1 this is mostly alphabetical, however with the default --checkers 8 it is somewhat random.

Limitations

The --order-by flag does not do a separate pass over the data. This means that it may transfer some files out of the order specified if
* there are no files in the backlog or the source has not been fully scanned yet
* there are more than --max-backlog files in the backlog

Rclone will do its best to transfer the best file it has so in practice this should not cause a problem. Think of --order-by as being more of a best efforts flag rather than a perfect ordering.

If you want perfect ordering then you will need to specify --check-first which will find all the files which need transferring first before transferring any.

--partial-suffix
When --inplace is not used, it causes rclone to use the --partial-suffix as suffix for temporary files.

Suffix length limit is 16 characters.

The default is .partial.

--password-command SpaceSepList
This flag supplies a program which should supply the config password when run. This is an alternative to rclone prompting for the password or setting the RCLONE_CONFIG_PASS variable. It is also used when setting the config password for the first time.

The argument to this should be a command with a space separated list of arguments. If one of the arguments has a space in then enclose it in ", if you want a literal " in an argument then enclose the argument in " and double the ". See CSV encoding for more info.

Eg
--password-command "echo hello"
--password-command 'echo "hello with space"'
--password-command 'echo "hello with ""quotes"" and space"'

Note that when changing the configuration password the environment variable RCLONE_PASSWORD_CHANGE=1 will be set. This can be used to distinguish initial decryption of the config file from the new password.

See the Configuration Encryption for more info.

See a Windows PowerShell example on the Wiki.

-P, --progress

This flag makes rclone update the stats in a static block in the terminal providing a realtime overview of the transfer.

Any log messages will scroll above the static block. Log messages will push the static block down to the bottom of the terminal where it will stay.

Normally this is updated every 500mS but this period can be overridden with the --stats flag.

This can be used with the --stats-one-line flag for a simpler display.

To change the display length of filenames (for different terminal widths), see the --stats-file-name-length option. The default output is formatted for 80 character wide terminals.

Note: On Windows until this bug is fixed all non-ASCII characters will be replaced with . when --progress is in use.

--progress-terminal-title
This flag, when used with -P/--progress, will print the string ETA: %s to the terminal title.

-q, --quiet

This flag will limit rclone's output to error messages only.

--refresh-times
The --refresh-times flag can be used to update modification times of existing files when they are out of sync on backends which don't support hashes.

This is useful if you uploaded files with the incorrect timestamps and you now wish to correct them.

This flag is only useful for destinations which don't support hashes (e.g. crypt).

This can be used any of the sync commands sync, copy or move.

To use this flag you will need to be doing a modification time sync (so not using --size-only or --checksum). The flag will have no effect when using --size-only or --checksum.

If this flag is used when rclone comes to upload a file it will check to see if there is an existing file on the destination. If this file matches the source with size (and checksum if available) but has a differing timestamp then instead of re-uploading it, rclone will update the timestamp on the destination file. If the checksum does not match rclone will upload the new file. If the checksum is absent (e.g. on a crypt backend) then rclone will update the timestamp.

Note that some remotes can't set the modification time without re-uploading the file so this flag is less useful on them.

Normally if you are doing a modification time sync rclone will update modification times without --refresh-times provided that the remote supports checksums and the checksums match on the file. However if the checksums are absent then rclone will upload the file rather than setting the timestamp as this is the safe behaviour.

--retries int
Retry the entire sync if it fails this many times it fails (default 3).

Some remotes can be unreliable and a few retries help pick up the files which didn't get transferred because of errors.

Disable retries with --retries 1.

--retries-sleep=TIME
This sets the interval between each retry specified by --retries

The default is 0. Use 0 to disable.

--server-side-across-configs
Allow server-side operations (e.g. copy or move) to work across different configurations.

This can be useful if you wish to do a server-side copy or move between two remotes which use the same backend but are configured differently.

Note that this isn't enabled by default because it isn't easy for rclone to tell if it will work between any two configurations.

--size-only
Normally rclone will look at modification time and size of files to see if they are equal. If you set this flag then rclone will check only the size.

This can be useful transferring files from Dropbox which have been modified by the desktop sync client which doesn't set checksums of modification times in the same way as rclone.

--stats=TIME
Commands which transfer data (sync, copy, copyto, move, moveto) will print data transfer stats at regular intervals to show their progress.

This sets the interval.

The default is 1m. Use 0 to disable.

If you set the stats interval then all commands can show stats. This can be useful when running other commands, check or mount for example.

Stats are logged at INFO level by default which means they won't show at default log level NOTICE. Use --stats-log-level NOTICE or -v to make them show. See the Logging section for more info on log levels.

Note that on macOS you can send a SIGINFO (which is normally ctrl-T in the terminal) to make the stats print immediately.

--stats-file-name-length integer
By default, the --stats output will truncate file names and paths longer than 40 characters. This is equivalent to providing --stats-file-name-length 40. Use --stats-file-name-length 0 to disable any truncation of file names printed by stats.

--stats-log-level string
Log level to show --stats output at. This can be DEBUG, INFO, NOTICE, or ERROR. The default is INFO. This means at the default level of logging which is NOTICE the stats won't show - if you want them to then use --stats-log-level NOTICE. See the Logging section for more info on log levels.

--stats-one-line
When this is specified, rclone condenses the stats into a single line showing the most important stats only.

--stats-one-line-date
When this is specified, rclone enables the single-line stats and prepends the display with a date string. The default is 2006/01/02 15:04:05 -

--stats-one-line-date-format
When this is specified, rclone enables the single-line stats and prepends the display with a user-supplied date string. The date string MUST be enclosed in quotes. Follow golang specs for date formatting syntax.

--stats-unit=bits|bytes
By default, data transfer rates will be printed in bytes per second.

This option allows the data rate to be printed in bits per second.

Data transfer volume will still be reported in bytes.

The rate is reported as a binary unit, not SI unit. So 1 Mbit/s equals 1,048,576 bit/s and not 1,000,000 bit/s.

The default is bytes.

--suffix=SUFFIX
When using sync, copy or move any files which would have been overwritten or deleted will have the suffix added to them. If there is a file with the same path (after the suffix has been added), then it will be overwritten.

The remote in use must support server-side move or copy and you must use the same remote as the destination of the sync.

This is for use with files to add the suffix in the current directory or with --backup-dir. See --backup-dir for more info.

For example
$> rclone copy --interactive /path/to/local/file remote:current --suffix .bak

will copy /path/to/local to remote:current, but for any files which would have been updated or deleted have .bak added.

If using rclone sync with --suffix and without --backup-dir then it is recommended to put a filter rule in excluding the suffix otherwise the sync will delete the backup files.
$> rclone sync --interactive /path/to/local/file remote:current --suffix .bak --exclude "*.bak"

--suffix-keep-extension
When using --suffix, setting this causes rclone put the SUFFIX before the extension of the files that it backs up rather than after.

So let's say we had --suffix -2019-01-01, without the flag file.txt would be backed up to file.txt-2019-01-01 and with the flag it would be backed up to file-2019-01-01.txt. This can be helpful to make sure the suffixed files can still be opened.

If a file has two (or more) extensions and the second (or subsequent) extension is recognised as a valid mime type, then the suffix will go before that extension. So file.tar.gz would be backed up to file-2019-01-01.tar.gz whereas file.badextension.gz would be backed up to file.badextension-2019-01-01.gz.

--syslog
On capable OSes (not Windows or Plan9) send all log output to syslog.

This can be useful for running rclone in a script or rclone mount.

--temp-dir=DIR

Specify the directory rclone will use for temporary files, to override the default. Make sure the directory exists and have accessible permissions.

By default the operating system's temp directory will be used:
  * On Unix systems, $TMPDIR if non-empty, else /tmp.
  * On Windows, the first non-empty value from %TMP%, %TEMP%, %USERPROFILE%, or the Windows directory.

When overriding the default with this option, the specified path will be set as value of environment variable TMPDIR on Unix systems and TMP and TEMP on Windows.

You can use the config paths command to see the current value.

--tpslimit float
Limit transactions per second to this number. Default is 0 which is used to mean unlimited transactions per second.

A transaction is roughly defined as an API call; its exact meaning will depend on the backend. For HTTP based backends it is an HTTP PUT/GET/POST/etc and its response. For FTP/SFTP it is a round trip transaction over TCP.

For example, to limit rclone to 10 transactions per second use --tpslimit 10, or to 1 transaction every 2 seconds use --tpslimit 0.5.

Use this when the number of transactions per second from rclone is causing a problem with the cloud storage provider (e.g. getting you banned or rate limited).

This can be very useful for rclone mount to control the behaviour of applications using it.

This limit applies to all HTTP based backends and to the FTP and SFTP backends. It does not apply to the local backend or the Storj backend.

See also --tpslimit-burst.

--tpslimit-burst int
Max burst of transactions for --tpslimit (default 1).

Normally --tpslimit will do exactly the number of transaction per second specified. However if you supply --tps-burst then rclone can save up some transactions from when it was idle giving a burst of up to the parameter supplied.

For example if you provide --tpslimit-burst 10 then if rclone has been idle for more than 10*--tpslimit then it can do 10 transactions very quickly before they are limited again.

This may be used to increase performance of --tpslimit without changing the long term average number of transactions per second.

--track-renames
By default, rclone doesn't keep track of renamed files, so if you rename a file locally then sync it to a remote, rclone will delete the old file on the remote and upload a new copy.

An rclone sync with --track-renames runs like a normal sync, but keeps track of objects which exist in the destination but not in the source (which would normally be deleted), and which objects exist in the source but not the destination (which would normally be transferred). These objects are then candidates for renaming.

After the sync, rclone matches up the source only and destination only objects using the --track-renames-strategy specified and either renames the destination object or transfers the source and deletes the destination object. --track-renames is stateless like all of rclone's syncs.

To use this flag the destination must support server-side copy or server-side move, and to use a hash based --track-renames-strategy (the default) the source and the destination must have a compatible hash.

If the destination does not support server-side copy or move, rclone will fall back to the default behaviour and log an error level message to the console.

Encrypted destinations are not currently supported by --track-renames if --track-renames-strategy includes hash.

Note that --track-renames is incompatible with --no-traverse and that it uses extra memory to keep track of all the rename candidates.

Note also that --track-renames is incompatible with --delete-before and will select --delete-after instead of --delete-during.

--track-renames-strategy (hash,modtime,leaf,size)
This option changes the file matching criteria for --track-renames.

The matching is controlled by a comma separated selection of these tokens:
  * modtime - the modification time of the file - not supported on all backends
  * hash - the hash of the file contents - not supported on all backends
  * leaf - the name of the file not including its directory name
  * size - the size of the file (this is always enabled)

The default option is hash.

Using --track-renames-strategy modtime,leaf would match files based on modification time, the leaf of the file name and the size only.

Using --track-renames-strategy modtime or leaf can enable --track-renames support for encrypted destinations.

Note that the hash strategy is not supported with encrypted destinations.

--delete-(before,during,after)
This option allows you to specify when files on your destination are deleted when you sync folders.

Specifying the value --delete-before will delete all files present on the destination, but not on the source before starting the transfer of any new or updated files. This uses two passes through the file systems, one for the deletions and one for the copies.

Specifying --delete-during will delete files while checking and uploading files. This is the fastest option and uses the least memory.

Specifying --delete-after (the default value) will delay deletion of files until all new/updated files have been successfully transferred. The files to be deleted are collected in the copy pass then deleted after the copy pass has completed successfully. The files to be deleted are held in memory so this mode may use more memory. This is the safest mode as it will only delete files if there have been no errors subsequent to that. If there have been errors before the deletions start then you will get the message not deleting files as there were IO errors.

--fast-list
When doing anything which involves a directory listing (e.g. sync, copy, ls - in fact nearly every command), rclone has different strategies to choose from.

The basic strategy is to list one directory and processes it before using more directory lists to process any subdirectories. This is a mandatory backend feature, called List, which means it is supported by all backends. This strategy uses small amount of memory, and because it can be parallelised it is fast for operations involving processing of the list results.

Some backends provide the support for an alternative strategy, where all files beneath a directory can be listed in one (or a small number) of transactions. Rclone supports this alternative strategy through an optional backend feature called ListR. You can see in the storage system overview documentation's optional features section which backends it is enabled for (these tend to be the bucket-based ones, e.g. S3, B2, GCS, Swift). This strategy requires fewer transactions for highly recursive operations, which is important on backends where this is charged or heavily rate limited. It may be faster (due to fewer transactions) or slower (because it can't be parallelized) depending on different parameters, and may require more memory if rclone has to keep the whole listing in memory.

Which listing strategy rclone picks for a given operation is complicated, but in general it tries to choose the best possible. It will prefer ListR in situations where it doesn't need to store the listed files in memory, e.g. for unlimited recursive ls command variants. In other situations it will prefer List, e.g. for sync and copy, where it needs to keep the listed files in memory, and is performing operations on them where parallelization may be a huge advantage.

Rclone is not able to take all relevant parameters into account for deciding the best strategy, and therefore allows you to influence the choice in two ways: You can stop rclone from using ListR by disabling the feature, using the --disable option (--disable ListR), or you can allow rclone to use ListR where it would normally choose not to do so due to higher memory usage, using the --fast-list option. Rclone should always produce identical results either way. Using --disable ListR or --fast-list on a remote which doesn't support ListR does nothing, rclone will just ignore it.

A rule of thumb is that if you pay for transactions and can fit your entire sync listing into memory, then --fast-list is recommended. If you have a very big sync to do, then don't use --fast-list, otherwise you will run out of memory. Run some tests and compare before you decide, and if in doubt then just leave the default, let rclone decide, i.e. not use --fast-list.

--timeout=TIME
This sets the IO idle timeout. If a transfer has started but then becomes idle for this long it is considered broken and disconnected.

The default is 5m. Set to 0 to disable.

--transfers=N
The number of file transfers to run in parallel. It can sometimes be useful to set this to a smaller number if the remote is giving a lot of timeouts or bigger if you have lots of bandwidth and a fast remote.

The default is to run 4 file transfers in parallel.

Look at --multi-thread-streams if you would like to control single file transfers.

-u, --update

This forces rclone to skip any files which exist on the destination and have a modified time that is newer than the source file.

This can be useful in avoiding needless transfers when transferring to a remote which doesn't support modification times directly (or when using --use-server-modtime to avoid extra API calls) as it is more accurate than a --size-only check and faster than using --checksum. On such remotes (or when

using --use-server-modtime) the time checked will be the uploaded time.

If an existing destination file has a modification time older than the source file's, it will be updated if the sizes are different. If the sizes are the same, it will be updated if the checksum is different or not available.

If an existing destination file has a modification time equal (within the computed modify window) to the source file's, it will be updated if the sizes are different. The checksum will not be checked in this case unless the --checksum flag is provided.

In all other cases the file will not be updated.

Consider using the --modify-window flag to compensate for time skews between the source and the backend, for backends that do not support mod times, and instead use uploaded times. However, if the backend does not support checksums, note that syncing or copying within the time skew window may still result in additional transfers for safety.

--use-mmap
If this flag is set then rclone will use anonymous memory allocated by mmap on Unix based platforms and VirtualAlloc on Windows for its transfer buffers (size controlled by --buffer-size). Memory allocated like this does not go on the Go heap and can be returned to the OS immediately when it is finished with.

If this flag is not set then rclone will allocate and free the buffers using the Go memory allocator which may use more memory as memory pages are returned less aggressively to the OS.

It is possible this does not work well on all platforms so it is disabled by default; in the future it may be enabled by default.

--use-server-modtime
Some object-store backends (e.g, Swift, S3) do not preserve file modification times (modtime). On these backends, rclone stores the original modtime as additional metadata on the object. By default it will make an API call to retrieve the metadata when the modtime is needed by an operation.

Use this flag to disable the extra API call and rely instead on the server's modified time. In cases such as a local to remote sync using --update, knowing the local file is newer than the time it was last uploaded to the remote is sufficient. In those cases, this flag can speed up the process and reduce the number of API calls necessary.

Using this flag on a sync operation without also using --update would cause all files modified at any time other than the last upload time to be uploaded again, which is probably not what you want.

-v, -vv, --verbose

With -v rclone will tell you about each file that is transferred and a small number of significant events.

With -vv rclone will become very verbose telling you about every file it considers and transfers. Please send bug reports with a log with this setting.

When setting verbosity as an environment variable, use RCLONE_VERBOSE=1 or RCLONE_VERBOSE=2 for -v and -vv respectively.

-V, --version

Prints the version number

SSL/TLS options

The outgoing SSL/TLS connections rclone makes can be controlled with these options. For example this can be very useful with the HTTP or WebDAV backends. Rclone HTTP servers have their own set of configuration for SSL/TLS which you can find in their documentation.

--ca-cert stringArray
This loads the PEM encoded certificate authority certificates and uses it to verify the certificates of the servers rclone connects to.

If you have generated certificates signed with a local CA then you will need this flag to connect to servers using those certificates.

--client-cert string
This loads the PEM encoded client side certificate.

This is used for mutual TLS authentication.

The --client-key flag is required too when using this.

--client-key string
This loads the PEM encoded client side private key used for mutual TLS authentication. Used in conjunction with --client-cert.

--no-check-certificate=true/false
--no-check-certificate controls whether a client verifies the server's certificate chain and host name. If --no-check-certificate is true, TLS accepts any certificate presented by the server and any host name in that certificate. In this mode, TLS is susceptible to man-in-the-middle attacks.

   This option defaults to false.

   This should be used only for testing.

Configuration Encryption

   Your configuration file contains information for logging in to your cloud services. This means that you should keep your rclone.conf file in a secure location.

   If you are in an environment where that isn't possible, you can add a password to your configuration. This means that you will have to supply the password every time you start rclone.

   To add a password to your rclone configuration, execute rclone config.

```
>rclone config
Current remotes:

e) Edit existing remote
n) New remote
d) Delete remote
s) Set configuration password
q) Quit config
e/n/d/s/q>
```

   Go into s, Set configuration password:

```
e/n/d/s/q> s
Your configuration is not encrypted.
If you add a password, you will protect your login information to cloud services.
a) Add Password
q) Quit to main menu
a/q> a
Enter NEW configuration password:
password:
Confirm NEW password:
password:
Password set
Your configuration is encrypted.
c) Change Password
u) Unencrypt configuration
q) Quit to main menu
c/u/q>
```

   Your configuration is now encrypted, and every time you start rclone you will have to supply the password. See below for details. In the same menu, you can change the password or completely remove encryption from your configuration.

   There is no way to recover the configuration if you lose your password.

   You can also use
   * rclone config encryption set to set the config encryption directly
   * rclone config encryption remove to remove it
   * rclone config encryption check to check that it is encrypted properly.

   rclone uses nacl secretbox which in turn uses XSalsa20 and Poly1305 to encrypt and authenticate your configuration with secret-key cryptography. The password is SHA-256 hashed, which produces the key for secretbox. The hashed password is not stored.

   While this provides very good security, we do not recommend storing your encrypted rclone configuration in public if it contains sensitive information, maybe except if you use a very strong password.

   If it is safe in your environment, you can set the RCLONE_CONFIG_PASS environment variable to contain your password, in which case it will be used for decrypting the configuration.

   You can set this for a session from a script. For unix like systems save this to a file called set-rclone-password:

```
#!/bin/echo Source this file don't run it

read -s RCLONE_CONFIG_PASS
export RCLONE_CONFIG_PASS
```

   Then source the file when you want to use it. From the shell you would do source set-rclone-password. It will then ask you for the password and set it in the environment variable.

   An alternate means of supplying the password is to provide a script which will retrieve the password and print on standard output. This script should have a fully specified path name and not rely on any environment variables. The script is supplied either via --password-command="..." command line argument or via the RCLONE_PASSWORD_COMMAND environment variable.

   One useful example of this is using the passwordstore application to retrieve the password:

```
export RCLONE_PASSWORD_COMMAND="pass rclone/config"
```

   If the passwordstore password manager holds the password for the rclone configuration, using the script method means the password is primarily protected by the passwordstore system, and is never embedded in the clear in scripts, nor available for examination using the standard commands available. It is quite possible with long running rclone sessions for copies of passwords to be innocently captured in log files or terminal scroll buffers, etc. Using the script method of

supplying the password enhances the security of the config password considerably.

If you are running rclone inside a script, unless you are using the --password-command method, you might want to disable password prompts. To do that, pass the parameter --ask-password=false to rclone. This will make rclone fail instead of asking for a password if RCLONE_CONFIG_PASS doesn't contain a valid password, and --password-command has not been supplied.

Whenever running commands that may be affected by options in a configuration file, rclone will look for an existing file according to the rules described above, and load any it finds. If an encrypted file is found, this includes decrypting it, with the possible consequence of a password prompt. When executing a command line that you know are not actually using anything from such a configuration file, you can avoid it being loaded by overriding the location, e.g. with one of the documented special values for memory-only configuration. Since only backend options can be stored in configuration files, this is normally unnecessary for commands that do not operate on backends, e.g. completion. However, it will be relevant for commands that do operate on backends in general, but are used without referencing a stored remote, e.g. listing local filesystem paths, or connection strings: rclone --config="" ls .

Configuration Encryption Cheatsheet

You can quickly apply a configuration encryption without plain-text at rest or transfer. Detailed instructions for popular OSes:

Mac

* Generate and store a password

security add-generic-password -a rclone -s config -w $(openssl rand -base64 40)
* Add the retrieval instruction to your .zprofile / .profile

export RCLONE_PASSWORD_COMMAND="/usr/bin/security find-generic-password -a rclone -s config -w"

Linux

* Prerequisite

Linux doesn't come with a default password manager. Let's install the "pass" utility using a package manager, e.g. apt install pass, yum install pass, etc.; then initialize a password store:

pass init rclone
* Generate and store a password

echo $(openssl rand -base64 40) | pass insert -m rclone/config
* Add the retrieval instruction

export RCLONE_PASSWORD_COMMAND="/usr/bin/pass rclone/config"

Windows

* Generate and store a password

New-Object -TypeName PSCredential -ArgumentList "rclone", (ConvertTo-SecureString -String ([System.Web.Security.Membership]::GeneratePassword(40, 10)) -AsPlainText -Force) | Export-Clixml -Path "rclone-credential.xml"
* Add the password retrieval instruction

[Environment]::SetEnvironmentVariable("RCLONE_PASSWORD_COMMAND", "[System.Runtime.InteropServices.Marshal]::PtrToStringAuto([System.Runtime.InteropServices.Marshal]::SecureStringToBSTR((Import-Clixml -Path "rclone-credential.xml").Password))")

Encrypt the config file (all systems)

* Execute rclone config -> s
* Add/update the password from previous steps

Developer options

These options are useful when developing or debugging rclone. There are also some more remote specific options which aren't documented here which are used for testing. These start with remote name e.g. --drive-test-option - see the docs for the remote in question.

--dump flag,flag,flag
The --dump flag takes a comma separated list of flags to dump info about.

Note that some headers including Accept-Encoding as shown may not be correct in the request and the response may not show Content-Encoding if the go standard libraries auto gzip encoding was in effect. In this case the body of the request will be gunzipped before showing it.

The available flags are:

--dump headers
Dump HTTP headers with Authorization: lines removed. May still contain sensitive info. Can be very verbose. Useful for debugging only.

Use --dump auth if you do want the Authorization: headers.

--dump bodies
   Dump HTTP headers and bodies - may contain sensitive info. Can be very verbose. Useful for debugging only.

   Note that the bodies are buffered in memory so don't use this for enormous files.

--dump requests
   Like --dump bodies but dumps the request bodies and the response headers. Useful for debugging download problems.

--dump auth
   Dump HTTP headers - will contain sensitive info such as Authorization: headers - use --dump headers to dump without Authorization: headers. Can be very verbose. Useful for debugging only.

--dump filters
   Dump the filters to the output. Useful to see exactly what include and exclude options are filtering on.

--dump goroutines
   This dumps a list of the running go-routines at the end of the command to standard output.

--dump openfiles
   This dumps a list of the open files at the end of the command. It uses the lsof command to do that so you'll need that installed to use it.

--dump mapper
   This shows the JSON blobs being sent to the program supplied with --metadata-mapper and received from it. It can be useful for debugging the metadata mapper interface.

--memprofile=FILE
   Write memory profile to file. This can be analysed with go tool pprof.

Filtering
   For the filtering options
     * --delete-excluded
     * --filter
     * --filter-from
     * --exclude
     * --exclude-from
     * --exclude-if-present
     * --include
     * --include-from
     * --files-from
     * --files-from-raw
     * --min-size
     * --max-size
     * --min-age
     * --max-age
     * --hash-filter
     * --dump filters
     * --metadata-include
     * --metadata-include-from
     * --metadata-exclude
     * --metadata-exclude-from
     * --metadata-filter
     * --metadata-filter-from

   See the filtering section.

Remote control
   For the remote control options and for instructions on how to remote control rclone
     * --rc
     * and anything starting with --rc-

   See the remote control section.

Logging
   rclone has 4 levels of logging, ERROR, NOTICE, INFO and DEBUG.

   By default, rclone logs to standard error. This means you can redirect standard error and still see the normal output of rclone commands (e.g. rclone ls).

   By default, rclone will produce Error and Notice level messages.

   If you use the -q flag, rclone will only produce Error messages.

   If you use the -v flag, rclone will produce Error, Notice and Info messages.

   If you use the -vv flag, rclone will produce Error, Notice, Info and Debug messages.

   You can also control the log levels with the --log-level flag.

   If you use the --log-file=FILE option, rclone will redirect Error, Info and Debug messages along with standard error to FILE.

   If you use the --syslog flag then rclone will log to syslog and the --syslog-facility control which

facility it uses.

Rclone prefixes all log messages with their level in capitals, e.g. INFO which makes it easy to grep the log file for different kinds of information.

Metrics

Rclone can publish metrics in the OpenMetrics/Prometheus format.

To enable the metrics endpoint, use the --metrics-addr flag. Metrics can also be published on the --rc-addr port if the --rc flag and --rc-enable-metrics flags are supplied or if using rclone rcd --rc-enable-metrics

Rclone provides extensive configuration options for the metrics HTTP endpoint. These settings are grouped under the Metrics section and have a prefix --metrics-*.

When metrics are enabled with --rc-enable-metrics, they will be published on the same port as the rc API. In this case, the --metrics-* flags will be ignored, and the HTTP endpoint configuration will be managed by the --rc-* parameters.

Exit Code

If any errors occur during the command execution, rclone will exit with a non-zero exit code. This allows scripts to detect when rclone operations have failed.

During the startup phase, rclone will exit immediately if an error is detected in the configuration. There will always be a log message immediately before exiting.

When rclone is running it will accumulate errors as it goes along, and only exit with a non-zero exit code if (after retries) there were still failed transfers. For every error counted there will be a high priority log message (visible with -q) showing the message and which file caused the problem. A high priority message is also shown when starting a retry so the user can see that any previous error messages may not be valid after the retry. If rclone has done a retry it will log a high priority message if the retry was successful.

List of exit codes

- 0 - Success
- 1 - Error not otherwise categorised
- 2 - Syntax or usage error
- 3 - Directory not found
- 4 - File not found
- 5 - Temporary error (one that more retries might fix) (Retry errors)
- 6 - Less serious errors (like 461 errors from dropbox) (NoRetry errors)
- 7 - Fatal error (one that more retries won't fix, like account suspended) (Fatal errors)
- 8 - Transfer exceeded - limit set by --max-transfer reached
- 9 - Operation successful, but no files transferred (Requires --error-on-no-transfer)
- 10 - Duration exceeded - limit set by --max-duration reached

Environment Variables

Rclone can be configured entirely using environment variables. These can be used to set defaults for options or config file entries.

Options

Every option in rclone can have its default set by environment variable.

To find the name of the environment variable, first, take the long option name, strip the leading --, change - to _, make upper case and prepend RCLONE_.

For example, to always set --stats 5s, set the environment variable RCLONE_STATS=5s. If you set stats on the command line this will override the environment variable setting.

Or to always use the trash in drive --drive-use-trash, set RCLONE_DRIVE_USE_TRASH=true.

Verbosity is slightly different, the environment variable equivalent of --verbose or -v is RCLONE_VERBOSE=1, or for -vv, RCLONE_VERBOSE=2.

The same parser is used for the options and the environment variables so they take exactly the same form.

The options set by environment variables can be seen with the -vv flag, e.g. rclone version -vv.

Options that can appear multiple times (type stringArray) are treated slightly differently as environment variables can only be defined once. In order to allow a simple mechanism for adding one or many items, the input is treated as a CSV encoded string. For example

Environment Variable	Equivalent options
RCLONE_EXCLUDE="*.jpg"	--exclude "*.jpg"
RCLONE_EXCLUDE="*.jpg,*.png"	--exclude "*.jpg" --exclude "*.png"
RCLONE_EXCLUDE='"*.jpg","*.png"'	--exclude "*.jpg" --exclude "*.png"
RCLONE_EXCLUDE='"/directory with comma , in it /**"'	`--exclude "/directory with comma , in it /**"

If stringArray options are defined as environment variables and options on the command line then all the values will be used.

Config file

You can set defaults for values in the config file on an individual remote basis. The names of the config items are documented in the page for each backend.

To find the name of the environment variable, you need to set, take RCLONE_CONFIG_ + name of remote + _ + name of config file option and make it all uppercase. Note one implication here is the remote's name must be convertible into a valid environment variable name, so it can only contain letters, digits, or the _ (underscore) character.

For example, to configure an S3 remote named mys3: without a config file (using unix ways of setting environment variables):

```
$> export RCLONE_CONFIG_MYS3_TYPE=s3
$> export RCLONE_CONFIG_MYS3_ACCESS_KEY_ID=XXX
$> export RCLONE_CONFIG_MYS3_SECRET_ACCESS_KEY=XXX
$> rclone lsd mys3:
          -1 2016-09-21 12:54:21        -1 my-bucket
$> rclone listremotes | grep mys3
mys3:
```

Note that if you want to create a remote using environment variables you must create the ..._TYPE variable as above.

Note that the name of a remote created using environment variable is case insensitive, in contrast to regular remotes stored in config file as documented above. You must write the name in uppercase in the environment variable, but as seen from example above it will be listed and can be accessed in lowercase, while you can also refer to the same remote in uppercase:

```
$> rclone lsd mys3:
          -1 2016-09-21 12:54:21        -1 my-bucket
$> rclone lsd MYS3:
          -1 2016-09-21 12:54:21        -1 my-bucket
```

Note that you can only set the options of the immediate backend, so RCLONE_CONFIG_MYS3CRYPT_ACCESS_KEY_ID has no effect, if myS3Crypt is a crypt remote based on an S3 remote. However RCLONE_S3_ACCESS_KEY_ID will set the access key of all remotes using S3, including myS3Crypt.

Note also that now rclone has connection strings, it is probably easier to use those instead which makes the above example

```
$> rclone lsd :s3,access_key_id=XXX,secret_access_key=XXX:
```

Precedence

The various different methods of backend configuration are read in this order and the first one with a value is used.

- Parameters in connection strings, e.g. myRemote,skip_links:
- Flag values as supplied on the command line, e.g. --skip-links
- Remote specific environment vars, e.g. RCLONE_CONFIG_MYREMOTE_SKIP_LINKS (see above).
- Backend-specific environment vars, e.g. RCLONE_LOCAL_SKIP_LINKS.
- Backend generic environment vars, e.g. RCLONE_SKIP_LINKS.
- Config file, e.g. skip_links = true.
- Default values, e.g. false - these can't be changed.

So if both --skip-links is supplied on the command line and an environment variable RCLONE_LOCAL_SKIP_LINKS is set, the command line flag will take preference.

The backend configurations set by environment variables can be seen with the -vv flag, e.g. rclone about myRemote: -vv.

For non backend configuration the order is as follows:

- Flag values as supplied on the command line, e.g. --stats 5s.
- Environment vars, e.g. RCLONE_STATS=5s.
- Default values, e.g. 1m - these can't be changed.

Other environment variables

- RCLONE_CONFIG_PASS set to contain your config file password (see Configuration Encryption section)
- HTTP_PROXY, HTTPS_PROXY and NO_PROXY (or the lowercase versions thereof).
  - HTTPS_PROXY takes precedence over HTTP_PROXY for https requests.
  - The environment values may be either a complete URL or a "host[:port]" for, in which case the "http" scheme is assumed.
- USER and LOGNAME values are used as fallbacks for current username. The primary method for looking up username is OS-specific: Windows API on Windows, real user ID in /etc/passwd on Unix systems. In the documentation the current username is simply referred to as $USER.
- RCLONE_CONFIG_DIR - rclone sets this variable for use in config files and sub processes to point to the directory holding the config file.

The options set by environment variables can be seen with the -vv and --log-level=DEBUG flags, e.g. rclone version -vv.

---

https://www.andyibanez.com/posts/rclone-basics-encryption/

rclone: From Basics to Encryption
Aug 20, 2019

The original version of this article was titled "rclone and Encryption Tutorial" and was posted in the old version of my website. The original article was written in February 2017 for a much older

version of rclone. This revised article covers rclone 1.48, and it has been rewritten from scratch to improve its quality. The examples used in the original article have been kept.

rclone is a command line tool, similar to rsync, with the difference that it can sync, move, copy, and in general do other file operations on cloud storage services, such as Dropbox and Google Drive. You can use rclone to create backups of your servers or personal computers or to simply store your files in the cloud, optionally adding encryption.

rclone supports many different services, from customer services like Dropbox, Google Drive, and One Drive, to bucket-based storage services such as Wasabi and Backblaze 2, and even more traditional protocols such as FTP and Webdav. It even supports the local filesystem, so you can use rclone fully locally as well. For a list of all supported services and protocols, see this page.

Getting Started

Installation

The installation process for rclone is straightforward. There's little point in me explaining how to install it, because the installation instructions page has instructions for all major OSes.

If you are on a Mac, I suggest you install it with Homebrew. If you have Homebrew installed, just type this command in your terminal followed by the Enter key:

```
$> brew install rclone
```

If you are on Windows instead, you can install it with Chocolatey. Simply open a Console window and type the following, followed by the Enter key:

```
$> choco install rclone
```

Once you have it installed, we'll explore some basic terminology to get you started with rclone.

Basic Terminology

The most important concept of rclone is the remote. A remote is simply a cloud service you have configured in the tool. For example, if you configure your Dropbox account, it becomes a remote. You can have more than one remote for each cloud service type, so if you have two Dropbox accounts, you can configure a remote for each. You can then interact with your remotes with the command line. If you added your Dropbox and Google Accounts, you can share files between them (note that in that case the files will be downloaded to your computer before being sent to the destination remote), and you can also send files from your local filesystem to a remote, and vice-versa.

With rclone, you execute commands. Things like copy, move, sync are all rclone commands. There's many more, and this article will not even cover half of them. It's good to know which ones exist though, so you can start adapting rclone to your use cases. Run the following command:

```
$> rclone
```

And the output will give you a list of commands rclone can do. It looks like this (this list has been truncated):

```
  copy            Copy files from source to dest, skipping already copied
  copyto          Copy files from source to dest, skipping already copied
  copyurl         Copy url content to dest.
  cryptcheck      Cryptcheck checks the integrity of a crypted remote.
  cryptdecode     Cryptdecode returns unencrypted file names.
  dbhashsum       Produces a Dropbox hash file for all the objects in the path.
  dedupe          Interactively find duplicate files and delete/rename them.
  delete          Remove the contents of path.
```

rclone commands typically have at least two parameters: a source and a destination, where each is a remote. For commands that take two remote parameters, the source goes before the destination.

Make sure you understand the four concepts above before moving.

rclone Basics

Rclone Configuration

You are about to configure your first remote. Open a command line window (or a Console window) and type the following:

```
$> rclone config
```

The config command allows you to add, delete, and modify existing remotes. All your remotes are stored in a config file. If you run the rclone config file command, you will see where your config file is stored. In my system, it prints /Users/andyibanez/.config/rclone/rclone.conf. Generally, you can recreate the contents of this file in another system or in a clean setup, but once we start dealing with encryption things can be a little bit messy in terms of management of your config file, so I recommend you keep a backup of it at all times.

One additional thing you can with rclone config is to encrypt your config file itself (not to be confused with encrypted remotes, which we will explore later). Encrypting your config file is useful if you are you going to be using rclone in a computer you don't control, unless you trust the host enough that they aren't going to be looking at your rclone data.

When running rclone, you can also pass it an alternative config file with the --config file. When this flag is missing, it will use the config stored in the path of rclone config file. We won't explore this idea further, but if you want to organize your remotes into different files or have any other use for separate config files, know that the option to have them is there.

Your first remote

Once you run rclone config, you will see an output similar to this:

```
Name                 Type
====                 ====
GoogleDrive          google drive

e) Edit existing remote
n) New remote
d) Delete remote
s) Set configuration password
q) Quit config
e/n/d/s/q>
```

Since you do not have any remotes yet, you will not see the first four lines of the output. The first four lines show you your existing remotes and what service they are configured with. In the example above, you can see I have a remote of type google drive. In this example, we will configure a Google Drive remote, but feel free to use other types to experiment.

Press n followed by the Enter key to create a new remote. The first prompt will be a name. This is just a name you assign to your remote so you can identify it later when doing a command such as rclone listremotes to view a list of all remotes in your config file. The name can be anything, just make sure it doesn't have any spaces. For example, if you want to add two Dropbox accounts, one for personal use and another one for business, you could create two remotes called DropboxPersonal or DropboxBusiness. I'll name mine aibanezDrive.

The next prompt will ask you for the Storage type.

```
 1 / A stackable unification remote, which can appear to merge the contents of several remotes
   \ "union"
 2 / Alias for an existing remote
   \ "alias"
 3 / Amazon Drive
   \ "amazon cloud drive"
 4 / Amazon S3 Compliant Storage Provider (AWS, Alibaba, Ceph, Digital Ocean, Dreamhost, IBM COS, Minio, etc)
   \ "s3"
 5 / Backblaze B2
   \ "b2"
 6 / Box
   \ "box"
 7 / Cache a remote
   \ "cache"
 8 / Dropbox
   \ "dropbox"
 9 / Encrypt/Decrypt a remote
   \ "crypt"
10 / FTP Connection
   \ "ftp"
11 / Google Cloud Storage (this is not Google Drive)
   \ "google cloud storage"
12 / Google Drive
   \ "drive"
13 / Hubic
   \ "hubic"
14 / JottaCloud
   \ "jottacloud"
15 / Koofr
   \ "koofr"
16 / Local Disk
   \ "local"
17 / Mega
   \ "mega"
18 / Microsoft Azure Blob Storage
   \ "azureblob"
19 / Microsoft OneDrive
   \ "onedrive"
20 / OpenDrive
   \ "opendrive"
21 / Openstack Swift (Rackspace Cloud Files, Memset Memstore, OVH)
   \ "swift"
22 / Pcloud
   \ "pcloud"
23 / QingCloud Object Storage
   \ "qingstor"
24 / SSH/SFTP Connection
   \ "sftp"
25 / Webdav
   \ "webdav"
26 / Yandex Disk
   \ "yandex"
27 / http Connection
   \ "http"
```

Since we are configuring Google Drive, write in 12 and press Enter.

Important Note!

At the time this article was written, the current rclone version was 1.48. You can check your rclone version by running:

```
$> rclone version
```

```
Andys-iMac-2:andyibanez andyibanez $> rclone version
rclone v1.48.0
- os/arch: darwin/amd64
- go version: go1.12.6
```

There's generally no changes with how rclone commands work in each release, but it's very common for each update to add new cloud storage providers. In this current version, Google Drive is option 12, but if you follow this article in a newer version, it's possible that the number assigned to Google Drive has changed. In fact, the initial version of this article which I wrote two years ago back in 2017, had Google Drive on number 7! So be wary of that if you are reading this article and following it step by step.

Many providers will prompt you for a client_id and a client_secret. You don't need these most of the time. In general, if you are having any problems with rclone using specific remotes, the community will recommend you to use these parameters and how they should be used. If you are familiar with how web APIs work for services like Google Drive, then you know how to use these parameters and in that case I recommend you set them - but in case you don't the defaults should work fine, at least for the purposes of this article.

For now, press Enter when it prompts you for each to leave them blank. But if you are inclined to fill these values (which is recommended for long term usage anyway), you can go here to find instructions on how to get your client ID and client secret.

The output will look like this:

```
Google Application Client Id
Setting your own is recommended.
See https://rclone.org/drive/#making-your-own-client-id for how to create your own.
If you leave this blank, it will use an internal key which is low performance.
Enter a string value. Press Enter for the default ("").
client_id>
Google Application Client Secret
Setting your own is recommended.
Enter a string value. Press Enter for the default ("").
client_secret>
```

You will then be asked for the scope. This configuration is specific to Google Drive. This setting allows you choose what "space" it will have in your Google Drive folder. The first option will grant rclone access to every single folder and file in your drive, with the exception of App Data. App Data is a special folder. Generally mobile applications that interact with Google Drive can store their data here and have it inaccessible to anyone outside. Think of WhatsApp for Google Drive - it's backups are stored in the App Data folder and it's not visible even to you. You can choose this folder to give rclone a special container to store files in. This will prevent your main Google Drive view from being polluted with encrypted files that you can't view without rclone itself, which is handy if you use the web client. I personally use this option for my backups, but know that if you use this option, it's really recommended you backup your config file.

All other scopes but the first one are outside the scope of this article, so write 1 followed by the Enter key. This will grant rclone standard access to the root of your Google Drive account and it will be able to see all your files and folders.

```
Scope that rclone should use when requesting access from drive.
Enter a string value. Press Enter for the default ("").
Choose a number from below, or type in your own value
 1 / Full access all files, excluding Application Data Folder.
   \ "drive"
 2 / Read-only access to file metadata and file contents.
   \ "drive.readonly"
   / Access to files created by rclone only.
 3 | These are visible in the drive website.
   | File authorization is revoked when the user deauthorizes the app.
   \ "drive.file"
   / Allows read and write access to the Application Data folder.
 4 | This is not visible in the drive website.
   \ "drive.appfolder"
   / Allows read-only access to file metadata but
 5 | does not allow any access to read or download file content.
   \ "drive.metadata.readonly"
scope> 1
```

Next you will be asked to put in a root_folder_id. This is an interesting option, but not one we will be using in this article. It allows you restrict rclone to certain folders, even the Computers folders the Backup and Sync program uses to store files in. For now leave it blank and press Enter.

```
ID of the root folder
Leave blank normally.
Fill in to access "Computers" folders. (see docs).
Enter a string value. Press Enter for the default ("").
root_folder_id>
```

You will now be asked for the path to a JSON file (service_account_file). This allows you to change the authentication method with Google Drive. We definitely don't want this now, so leave it blank and press Enter.

```
Service Account Credentials JSON file path
Leave blank normally.
Needed only if you want use SA instead of interactive login.
Enter a string value. Press Enter for the default ("").
service_account_file>
```

   After that you will be asked if you want to use advanced configurations. We aren't, so press n
   followed by Enter.
```
Edit advanced config? (y/n)
y) Yes
n) No
y/n> n
```

   Finally, you will be asked if you want to use auto config. If you configure rclone in your Mac, or
   Windows, or in another system with a GUI, you can say "yes" to this. If you were configuring rclone
   in a headless system (like SSHing into a Linux system with no GUI), you need to say "no" to this.

   Since you are working on a Mac, Windows, or Linux with a GUI, press y followed by Enter.
```
Remote config
Use auto config?
 * Say Y if not sure
 * Say N if you are working on a remote or headless machine
y) Yes
n) No
```

   This will launch your default browser with a Google Drive authorization page.
   rclone Authorization Page

   Once you are done authorizing your account, you will see another page with this in it:
   rclone Success

   If you are asked if you want to configure it as a team drive, press n followed by Enter:
```
Configure this as a team drive?
y) Yes
n) No
y/n> n
```

   rclone will print something like this in the console at the end, showing you information about your
   newly created remote. It may look differently depending on how and what parameters you have
   configured.
```
[aibanezDrive]
client_id =
client_secret =
token = {...}
```

   Press y followed by Enter. Your new remote has been created, and you are finally read to use it.

Interacting With Your First Remote
   Before we do anything, run:
```
$> rclone listremotes
```

   To view a list of all your remotes. Your newly created remote should look like this:
```
aibanezDrive:
```

   As you configure more remotes, they will show up in that list.

   We will create a list of files and directories and upload them to our Google Drive account using
   rclone. I need to remind you that we are currently playing with a non encrypted remote, so don't
   choose any files you wouldn't store unencrypted in the cloud. We will create the encrypted remote
   later.

   Go to your desktop and create a few files there. I created a directory called ~/Desktop/Cards and I
   created this directory structure with files:
```
Cards
 - Sakura
   - The Mirror
   - The Fly
   - The Sword
   - The Arrow
 - Shaoran
   - The Time
   - The Storm
   - The Return
   - The Freeze
 - None.txt
```

   Where Sakura and Shaoran are directories, and everything starting with The is a file. If you want to
   use this same directory structure, you can run this small bash script to create the files in the same
   location as mine:
```
mkdir ~/Desktop/Cards && cd ~/Desktop/Cards
mkdir Sakura Shaoran
touch Sakura/The\ Mirror
touch Sakura/The\ Fly
touch Sakura/The\ Sword
touch Sakura/The\ Return
touch Shaoran/The\ Time
touch Shaoran/The\ Storm
touch Shaoran/The\ Return
touch Shaoran/The\ Freeze
touch None.txt
```

Copying the Local Directory to Your Newly Created Google Drive Remote

The command to copy files and directories is very straight forward.

```
$> rclone copy LOCAL_DIRECTORY_OR_FILE REMOTE_DIRECTORY
```

This will copy LOCAL_DIRECTORY_OR_FILE to REMOTE_DIRECTORY. Fun little feature here is that rclone will skip everything that already exists in REMOTE_DIRECTORY by default, so you can use this command to do a very primitive version of incremental backups. For backups, I recommend you use the sync command instead, as it will mirror the LOCAL_DIRECTORY_OR_FILE exactly as it is on REMOTE_DIRECTORY. When you add files they will be added to the destination; when you change files they will be changed in the destination as well; when you delete files, they will also be deleted from the destination. Covering sync is outside the scope of this article, but keep in mind if you are interested in doing backups.

You can copy entire directories or single files only. rclone is very flexible in that aspect.

When you want to copy something to a remote, you write your remote name, followed by a :, and finally the path where you want to copy to.

Going back to the example we are building, rewrite the last command like this:

```
$> rclone copy ~/Desktop/Cards aibanezDrive:/Cards
```

Depending on your internet connection, it could take a while, but they are all small files so hopefully that's not the case.

After running the command, rclone will give you a nice little report of how everything went.

```
Transferred:      0 Bytes (0 Bytes/s)
Errors:                 0
Checks:                 0
Transferred:            9
Elapsed time:       28.9s
```

Always check these stats after running rclone to ensure your destination is not missing any data. When rclone detects the same files on the destination as in the origin already, it will run a Check instead of uploading it directly. This will ensure that, if the files are the same, they are skipped. If the files are different, they will be overridden in the destination.

Login to your Google Drive account using your web browser. You will see the Cards directory at the root of your account.

rclone Cards
And if you go inside that directory, you will find your other directories and files.
rclone More Dirs
rclone Files

Back on your terminal or console, you can type rclone lsd aibanezDrive:/Cards. This will list all the contents inside the Cards directory. If you do rclone ls aibanezDrive:/Cards instead, it will list all the contents instead. Be careful with this, because if you have a lot of data, ls can take a very long time to complete, as it will try to list every single file inside the specified path. You should prefer lsd over ls unless you understand how they are used.

Copying Files From Your Remotes to Your Local Filesystem

In the last section we copied local files to our remote. This time we will do it the other way around. Doing this is as easy as switching the source and destination around.

Go ahead and upload any file to your Google Drive account. I uploaded a file called "IMG_1434.JPG" and I uploaded it the root of my account. If I wanted to copy this file from my Drive account to my Local computer, inside the ~/Desktop/Cards directory using rclone, I'd use this command:

```
$> rclone copy aibanezDrive:/IMG_1434.JPG ~/Desktop/Cards
```

And don't forget that you can use this to download single files or entire directories if you please.

Additional Commands

It's beyond the scope of this article to talk about every single command and flag, but I thought I should mention a few of the most used ones.

rclone Supports a whole lot of commands as listed in their Documentation. Some useful ones are sync (it keeps a remote directory in sync with a local one), move, delete (remove the contents of a path), purge (remove a path and all its contents), ls (List all the objects in the specified path), lsd (list all the directories under the path), mkdir, and rmdir.

Some interesting flags you can use with your commands include but are not limited to --bwlimit, to limit the amount of bandwidth rclone will use, and --transfers to limit the amount of files that get transferred in batch.

Important Note!
It's important to note that some commands and flags may not work or may behave differently based on what kind of remote you are using. The Storage Systems Overview page has a helpful table and a few notes for how commands may behave. There's also a page for every supported remote that lists their quirks, features, and possible different behaviors for common commands.

Setting up Encryption with rclone

When we discussed what a remote means earlier, I said that a remote is a cloud account added to rclone. This is not strictly true, as there two notable exceptions to this.

There's two remote types that don't act as a cloud account. Instead, they sit "on top" of a normal

remote to do their job. Said remotes are crypt and cache, and I just call them "special" remotes. We will only cover crypt in this article.

The crypt remote takes another remote name as part of its configuration. The crypt remote itself never touches the cloud directly. Instead, when you are uploading something, it encrypts the data, hands it over to the underlying remote, and this underlying remote will do the uploading on its behalf. Likewise, when you download something, crypt just requests the encrypted data to the underlying remote, which in turns gives the encrypted blobs to crypt, who will decrypt them. Everything crypt does is to encrypt and decrypt data, and the underlying remote does the actual downloading and uploading.

In this section, I will show you how the aibanezDrive remote we created earlier acts as the underlying remote for a new crypt remote we are about to create. Note that once you configure a crypt remote on top of a normal remote, you can still use both remotes normally, so be careful you choose the right remote when uploading, otherwise you might upload something sensitive unencrypted by default.

Creating a Crypt Remote

Before we create the new remote, delete the /Cards folder from your Google Drive (if you created it), as we will be using this same directory to show how crypt works.

We need to run rclone config again. Press n followed by Enter to create a new remote.

One thing I don't like about rclone is that, when listing your remotes, you can't see what underlying remote a crypt remote is using, so when prompted for the name, I recommend you choose one that can help you identify both the crypt and the underlying remote. I will be naming mine aibanezDrive_Crypt.

After the name, you will be asked for the Storage type. Choose 9 which is the crypt type.

You will now be prompted for the underlying remote. Remember, this is the remote that will do the actual uploading and downloading, so write the name of the remote where you want to store your encrypted data. In this case it's aibanezDrive.

```
Remote to encrypt/decrypt.
Normally should contain a ':' and a path, eg "myremote:path/to/dir",
"myremote:bucket" or maybe "myremote:" (not recommended).
Enter a string value. Press Enter for the default ("").
remote> aibanezDrive:/Cards
```

A small discussion is in order here. First, the remote> we are using contains both the remote and a path (aibanezDrive:/Cards). A crypt remote can store it's data in just a folder somewhere in the underlying remote, or it can use the entirety of it. You should configure a special folder to store the encrypted data in, because rclone will have problems dealing with the contents of a remote that has both encrypted and unencrypted data. In the example above, the underlying remote is aibanezDrive:, and the root folder for crypt is /Cards inside of it. This means that the crypt remote will not be able to write or read outside the /Cards directory. When you use Google Drive through the drive client, you will see the /Cards directory, but everything inside of it will be gibberish.

Next you will be asked if you want to encrypt the file names. Rclone can upload the filenames with their original names while encrypting their content. The choice is up to you, but I prefer to encrypt the file names. So write 2 followed by Enter.

```
How to encrypt the filenames.
Enter a string value. Press Enter for the default ("standard").
Choose a number from below, or type in your own value
 1 / Don't encrypt the file names.  Adds a ".bin" extension only.
   \ "off"
 2 / Encrypt the filenames see the docs for the details.
   \ "standard"
 3 / Very simple filename obfuscation.
   \ "obfuscate"
```

Important Note!
Some remote types can't deal with very long file names or paths. Encrypting the files will make the file names longer as rclone stores metadata within the encrypted filename.

As a personal anecdote, when I used the Amazon Drive remote, there was a limitation to file length and I couldn't nest my files very deeply because of it. I haven't had any similar issues with Google Drive yet, but your mileage may very.

In general, if your file names are below 156 characters, you should be fine on all providers. Please refer to the crypt documentation to see if you can find any limitations with the file paths lengths with the remote types you intend on using.

You will now be asked if you want to encrypt directory names. Just like file names, you can leave them unencrypted. I prefer to encrypt them, but keep in mind that directory names are also longer when encrypted.

Press 1 followed by Enter.

```
Option to either encrypt directory names or leave them intact.
Enter a boolean value (true or false). Press Enter for the default ("true").
Choose a number from below, or type in your own value
 1 / Encrypt directory names.
   \ "true"
 2 / Don't encrypt directory names, leave them intact.
   \ "false"
directory_name_encryption> 1
```

Next you will be asked if you want to provide your own encryption password or have one generated by rclone. This is where you need to be careful. The password will be stored in a secure manner inside your config file. It's very hard to retrieve the password from the config file later. Once the crypt remote has a password, you never have to worry about writing it down again as rclone will handle all that for you. For this reason, if you choose yes, you have to make sure you will never lose your password. If you choose generate, you have to be responsible and backup your rclone config file. You should always backup your config file regardless of what you choose, but if you chose y and you remember your password, you can recreate the remote and access your data in case you lose the original config. If you choose g and you lose your config file, you will never have access to the encrypted data ever again. So be careful here, and remember, backup, backup, backup!

Choose y to write your own password for now.

```
Password or pass phrase for encryption.
y) Yes type in my own password
g) Generate random password
n) No leave this optional password blank
y/g/n> y
```

You will be prompted twice for your password.

To complicate things a bit, you are given the option to generate a second password. The options are the same as above, as are the implications of each. I always choose y because I generate and keep this data stored in my password manager.

```
Password or pass phrase for salt. Optional but recommended.
Should be different to the previous password.
y) Yes type in my own password
g) Generate random password
n) No leave this optional password blank
y/g/n> y
```

Next you will be prompted if you want to use advanced config. Press n followed by Enter.

Finally rclone will print your newly created crypt remote info.

```
[aibanezDrive_Crypt]
type = crypt
remote = aibanezDrive:/Cards
filename_encryption = standard
directory_name_encryption = true
password = *** ENCRYPTED ***
password2 = *** ENCRYPTED ***
```

We are ready to use our new shiny remote.

Using a Crypt Remote

The nice thing about rclone is that once you have used a remote type, you have used them all. The syntax for the common commands such as copy does not change at all, whether you are using a Dropbox remote or a crypt remote. Run the following to upload the /Desktop/Cards directory once again, but this time with your crypt remote:

```
$> rclone copy /Desktop/Cards aibanezDrive_Crypt:/
```

Now, open Google Drive in the web client and open the /Cards directory. You will see the right amount of folders and files, but their filenames are complete gibberish and even their contents are encrypted.

rclone Crypt Results

Now the only way to access these files is with rclone. rclone knows how to do the right thing. List the remote's contents:

```
$> rclone ls aibanezDrive_Crypt:/

Andys-iMac:Cards andyibanez$ rclone ls aibanezDrive_Crypt:/
   358768 IMG_1434.JPG
        0 None.txt
        0 Shaoran/The Freeze
        0 Shaoran/The Return
        0 Shaoran/The Storm
        0 Shaoran/The Time
        0 Sakura/The Sword
        0 Sakura/The Mirror
        0 Sakura/The Return
        0 Sakura/The Fly
```

And rclone will show you the decrypted filenames for you.

If you use copy to copy from the crypt remote to your computer, rclone will decrypt the files and their names. rclone's encryption method is really transparent, and you never have to concern yourself with the details of how it's done.

```
$> rclone copy aibanezDrive_Crypt:/Sakura /Desktop/restored
```

Conclusion

rclone is a great tool to interact with cloud storage services from the command line. It's encryption support is transparent and very easy to use. As long as you keep a backup of your rclone.conf file, you will never lose access to your encrypted files.

Where to Go From Here

Once you are familiar with rclone, I recommend you explore the Official Site of the project so you can learn more advanced tasks you can do. Take some time to explore the documentation to learn

how to use other commands, as well as how to use certain flags with certain commands. A particular command I use a lot is sync with the --log-file and --backup-dir flags. This allows me to check that all files were synced properly to the remote, and to move the ones that were different or deleted someplace else instead of outright removing them (for incremental backups). But there's much more you can do, and the only way to learn all the power of rclone is to browse the documentation and play around with it. In the future, I may share my backup strategies with rclone to give you ideas how you can use this amazing tool.

---
https://docs.rc.uab.edu/data_management/transfer/rclone/

RClone
RClone is a powerful command line tool for transferring and synchronizing files over the internet between various machines, servers and cloud storage services. It is highly recommended for small to moderate amounts of data. For very large amounts of data consider using Globus for increased robustness against failure. Where Globus is not available, rclone is still suitable.

RClone requires a modest amount of setup time on local machines, but once setup can be used fairly easily. RClone uses the concepts of "remotes", which is an abstract term for any storage service or device that is not physically part of the local machine. Many remotes are offered, including SFTP and various UAB Cloud Storage Solutions. SFTP may be used to access Cheaha, cloud.rc and other laptop and desktop computers.

To use RClone effectively, you'll need to setup remotes before using the various commands. Most file manipulation commands on Linux can be found in the RClone commands, but may have slightly different names, e.g. cp is rclone copy.

RClone is very powerful and, as such, has a wide variety of configuration options and flags to fine tune behavior. We will only cover the basics needed to install the software, setup remotes relevant to work at UAB, and some basic usage commands.

Quick Tutorial for Cheaha
To use RClone for simple data transfer to and from Cheaha, follow these steps:
1. Load the module using module load rclone.
2. Set up a remote storage provider
3. Copy files using the rclone cp command.
    + To transfer from Cheaha, use rclone cp :path/on/cheaha remote:path/on/dest
    + To transfer to Cheaha, use rclone cp remote:/path/on/remote :path/on/cheaha.

   Be sure to replace remote with the name of the remote you set up in step 2.

Installing
Installing on Cheaha
On Cheaha, RClone is already installed as a Module. Use module load rclone to load it.

Installing on Linux and cloud.rc
See Installing Software for general good practice on installing software, then use the following command.

```
$> curl https://rclone.org/install.sh | sudo bash
```

Open a new terminal and enter rclone to verify installation.

Installing on Windows
It is highly recommended to install rclone in Windows Subsystem for Linux (WSL).

To instead install natively on Windows, you will need to use the following instructions.
1. Download the appropriate version from the downloads page.
2. Extract rclone.exe into a memorable folder on your system. Do not put it into Program Files.
3. In the Start Menu type env and look for the application "Edit the system Environment Variables" to open the System Properties dialog.
4. Click the "Environment Variables..." button.
5. Under "User variables for $USER" find the variable "Path".
6. Click "Path" to select it.
7. Click the "Edit..." button to open a new dialog.
8. Click the "New" button.
9. Type in the folder path to rclone.exe as C:/path/to/rclone_folder, modified appropriately.
10. Click the "OK" button to confirm and close the dialog box.
11. Click the "OK" button to confirm and close the Environment Variables dialog box.
12. Click the "OK" button to confirm and close the System Properties dialog box.
13. Verify the installation by typing "cmd" in the Start Menu and opening the Command Prompt application.
14. Type rclone and you should see the rclone help text.

MacOS
Follow the online instructions for installing with brew.

Setting Up Remotes
RClone is capable of interfacing with many remote cloud services, as well as using sftp for connecting two personal computers or servers. We will only cover those cloud services relevant to UAB use. We will not cover how to connect to any other cloud services using RClone. More detailed information is available at the RClone documentation

Important
Cloud access tokens are always supplied with an expiration date for security reasons. You will need to repeat the setup process periodically to regain access via RClone.

Note
RClone has an unusual user interface, using alternating red and green blocks to differentiate list items. The colors do not convey any particular meaning beyond differentiation.

Setting Up an SFTP Remote

RClone connects two personal computers or servers using SFTP which is built on SSH, so a lot of these instructions mirror what would be done with an SSH configuration.

1. Generate a Key Pair for use with the remote machine.
2. At the terminal enter rclone config.
3. Follow the prompts to choose sftp.
4. Enter the following values as they come up, using defaults for other values.
   + name> Name of the remote for future reference and configuration
   + host> Remote IP address or cheaha.rc.uab.edu for Cheaha
   + user> The user you will log into on the remote machine
   + key_file> The absolute path to the private key file on the local machine, something like ~/.ssh/private_key_ed25519
   + key_file_pass> The passphrase used to secure the private key file (optional, but highly recommended)
5. Verify by using rclone lsd <name>.

The official docuemntation for rclone sftp is here.

Setting Up UAB Cloud Remotes

The setup process for UAB cloud remotes is generally the same, except for the specifics of authentication. The instruction template is outlined below and will point you to the authentication section specific to each remote when it becomes relevant.

As you step through the process, you will ultimately open two terminal windows and a browser window, and will need to copy text between the terminal windows. The first terminal window will be used to setup the RClone cloud remote. The second terminal will be used to authenticate to that cloud service and gain a token that will be passed back to the first terminal. Authentication will happen by logging into the service in a browser window. This setup method is necessary for any machine where a browser is not readily available, such as a cloud.rc virtual machine. To facilitate setup on these machines, the second terminal will be opened on a machine with RClone and a browser. An example of what this setup might look like is given below.

!overview of windows used for authenticating cloud remotes via rclone

Important
If you are using RClone in Windows Subsystem for Linux (WSL), you won't be able to open a browser using WSL. Instead, you will need to Install RClone on Windows and use the Windows Command Prompt terminal to use rclone authorize.

1. Open a terminal on the device you wish to authorize to access the chosen cloud service provider using RClone. This terminal will be referred to as terminal-1.
2. At terminal-1 enter rclone config.
3. Follow the prompts to choose one of the following. The selection here will be used later and will be referred to as <remote>.
4. UAB Box: select Box. <remote> will be replaced by box.
5. UAB SharePoint Site: select Microsoft OneDrive. <remote> will be replaced by onedrive.
6. UAB OneDrive: select Microsoft OneDrive. <remote> will be replaced by onedrive.
7. Enter a short, memorable name for future reference when prompted with name>. Keep this <name> in mind as it will be how you access the remote when Using Commands.
8. Press enter to leave all additional prompts blank until "Use auto config?". Type "n", for no, and press enter.
9. The prompt should now read config_token>.
10. On a machine with a browser, such as your personal computer, open a new terminal and enter rclone authorize "<remote>". Replacing <remote> with the value from step (3). This terminal will be referred to as terminal-2.
11. When the browser window opens, use it to authenticate to your selected service.
    + Authenticate to UAB Box.
    + Authenticate to Microsoft OneDrive.
    + Other services not officially supported by UAB IT are possible, but are not documented here. You will need to provide your own credentials for these services.
12. Terminal-2 will print a secret token, which will appear like in the following image. You will need to copy the portion highlighted in the image, between the lines with ---> and <---.
    !rclone authentication token sample
13. Copy and paste the token from the terminal-2 to terminal-1.
14. Follow the remaining prompts.
15. Verify success by using rclone lsd <name>: in terminal-1.

Authenticating to Cloud Remotes

Authenticating to UAB Box

1. Click "Use Single Sign On (SSO)".
   !box authentication dialog with single sign on highlighted
2. Type in your UAB email address (not your @uabmc.edu email!).
3. Click "Authorize".
   !box single sign on authentication dialog
4. You will be redirected to the UAB SSO page.
5. Authenticate with your BlazerID credentials.
6. You will be asked to grant permission to the RClone software. Click "Grant access to Box" if you want the software to work with Box. If you do not grant permission, you will not be able to use RClone with Box.
   !box grant permission request dialog
7. You will be redirected to a "Success!" page. Return to Terminal (5) to find the authentication token.
   !success page

    8. Return to Setting up UAB Cloud Remotes.

   Warning
   Tokens are set to expire after some time of disuse to decrease risk of a data breach. If your token expires, you can Reconnect to an Existing Remote rather than recreate the remote configuration completely from scratch.

Authenticating to Microsoft OneDrive
    1. Type in your UAB email address (not your @uabmc.edu email!).
    2. Click "Next".
       !onedrive authentication dialog
    3. If prompted, click "Work or school account".
       !onedrive account selection dialog
    4. You will be asked to grant permission to the RClone software. Click "Accept" if you want the software to work with OneDrive. If you do not grant permission, you will not be able to use RClone with OneDrive.
       !onedrive grant permission request dialog
    5. You will be redirected to a "Success!" page. Return to Terminal (5) to find the authentication token.
       !success page
    6. Next you will return to the general instructions. Before you do, note that you'll be asked to choose which type of OneDrive service to access. The prompt will look like the image below. For UAB, the two relevant selections will be (1) to access your personal OneDrive space and (3) for a SharePoint Site, e.g. for a lab or department.
       !rclone selection prompt among list of onedrive services
    7. With your selection in mind, return to Setting up UAB Cloud Remotes.

Setting Up an S3 LTS Remote
   The full S3 configuration process can be done from a single command line terminal. Open a terminal and enter rclone config to begin the configuration process.

   Note
   The locations where you will need to input either a command or select an option are preceded with a $ for easier navigation.

```
$> rclone config

2022/02/22 13:02:15 NOTICE: Config file "/home/mdefende/.config/rclone/rclone.conf" not found - using defaults
No remotes found - make a new one
n) New remote
s) Set configuration password
q) Quit config

# select 'n' to create a new remote
$> n/s/q> n

# name the new remote
$> name> uablts
```

   At this point, you've created a new remote configuration called uablts. This will be the remote name used in further commands. You can name the remote whatever you would like, but will need to replace uablts in the instructions with whichever name you chose, if you chose a different name.

```
...
4 / Amazon Drive
  \ (amazon cloud drive)
5 / Amazon S3 Compliant Storage Providers including AWS, Alibaba, Ceph, Digital Ocean, Dreamhost, IBM COS, Lyv
e Cloud, Minio, RackCorp, SeaweedFS, and Tencent COS
  \ (s3)
6 / Backblaze B2
  \ (b2)
...

$> Storage> 5

...
2 / Alibaba Cloud Object Storage System (OSS) formerly Aliyun
  \ (Alibaba)
3 / Ceph Object Storage
  \ (Ceph)
4 / Digital Ocean Spaces
  \ (DigitalOcean)
...

$> provider> 3

Option env_auth.
Get AWS credentials from runtime (environment variables or EC2/ECS meta data if no env vars).
Only applies if access_key_id and secret_access_key is blank.
Choose a number from below, or type in your own boolean value (true or false).
Press Enter for the default (false).
 1 / Enter AWS credentials in the next step.
   \ (false)
 2 / Get AWS credentials from the environment (env vars or IAM).
   \ (true)

$> env_auth> 1 (or leave blank)
```

```
Option access_key_id.
AWS Access Key ID.
Leave blank for anonymous access or runtime credentials.
Enter a value. Press Enter to leave empty.

$> access_key_id> (Enter your access key given to you by research computing)

Option secret_access_key.
AWS Secret Access Key (password).
Leave blank for anonymous access or runtime credentials.
Enter a value. Press Enter to leave empty.

$> secret_access_key> (Enter your secret access key given to you by research computing here)

Option region.
Region to connect to.
Leave blank if you are using an S3 clone and you don't have a region.
Choose a number from below, or type in your own value.
Press Enter to leave empty.
   / Use this if unsure.
 1 | Will use v4 signatures and an empty region.
   \ ()
   / Use this only if v4 signatures don't work.
 2 | E.g. pre Jewel/v10 CEPH.
   \ (other-v2-signature)

$> region> (Leave empty)

Option endpoint.
Endpoint for S3 API.
Required when using an S3 clone.
Enter a value. Press Enter to leave empty.

$> endpoint> s3.lts.rc.uab.edu

   From here, press Enter to accept default options until it gives you a summary of your connection
[uablts]
type = s3
provider = Ceph
access_key_id = ****************** # these will be filled in on your screen
secret_access_key = ********************************
endpoint = s3.lts.rc.uab.edu
--------------------
y) Yes this is OK (default)
e) Edit this remote
d) Delete this remote
y/e/d>

   Make sure everything looks correct here, then press Enter. At this point, it will bring you back to
   the main configuration menu. You can choose the Quit Config option and exit back to a basic terminal.

Reconnecting to an Existing Remote
   When your tokens expire, rather than recreate the remote from scratch, simply use the following
   command with your existing remote <name>.
rclone config reconnect <name>:

     * If you already have a token, you will be asked if you want to refresh it. Choose yes if so, then
       continue.
     * You will be prompted with Use auto config?. If you are on a machine with no access to a browser,
       respond n, as in the original setup.
     * Follow the steps in the appropriate section under Authenticating to Cloud Remotes, as in the
       original setup.

Usage
   RClone is a powerful tool with many commands available. We will only cover a small subset of the
   available commands, as most are beyond typical usage, so please see the RClone documentation for
   more information.

   All commands have access to the global flags. An important global flag is --dry-run to show what
   will happen without actually executing the command, which can be helpful to prevent costly mistakes.
   Other Helpful Global Flags are also available.

   The various remotes each have their own individual page with their own specific flags, and are linked
   in the relevant Setting up Remotes section above.

   Important
   Remote paths are always prefixed by the name of the remote like cheaha:/path/to/files. The colon
   character : is required for all remote paths. Local paths have no prefix like /path/to/local/files.
   RClone can thus be used between any two machines that are configured where rclone is being used,
   including from the local machine to itself. In the following instructions, replace <remote:> by the
   appropriate remote name from configuration. To access local files, leave <remote:> off entirely.

   Important
   Remember to use quotes " around paths with spaces like "path\to\a folder with spaces"

Usage Concept
```

All rclone commands follow the same general patterns outlined below. source is the name of the source remote and destination is the name of the destination remote. Remotes must be set up before using commands. To work with local files use the format :path/to/data instead of remote:path/to/data. The colon is necessary to access local files.

* Single-source commands like ls:

```
$> rclone ls <flags...> source:path/to/data
```

* Transfer commands like cp:

```
$> rclone cp <flags...> source:path/to/data destination:path/to/data
```

Source always comes before destination. To change the direction of file transfer, swap the order of source and destination.

Creating a Directory

To create a directory use rclone mkdir <remote:><path>.

Example: rclone mkdir box:manuscript.

Listing Files and Directories

To list files on a machine use rclone ls <remote:><path>.

Example rclone ls box:.

To list directories on a machine use rclone lsd <remote:><path>.

Example: rclone lsd box: should show manuscript.

Copying Files

To copy files without changing their name, or to recursively copy directory content, use rclone copy <source:><path> <destination:><path>. Note that the directory contents are copied, so when copying a directory, be sure that directory exists on the remote and that you are copying into it.

Example: rclone copy "C:\users\Name\My Documents" box:manuscript

To copy a single file and change its name, use rclone copyto <source:><path/oldname> <destination:><path/newname>.

Example rclone copyto "C:\users\Name\My Documents\manuscript.docx" box:manuscript\newest.docx

Syncing Between Two Devices

To make a destination directory's contents identical to a source directory, use rclone sync <source:><path> <destination:><path>

Example: rclone sync cheaha:"C:\users\Name\My Documents" box:manuscript.

Danger

rclone sync is a destructive operation and cannot be undone! If files exist on the destination that do not exist on the source, then they will be deleted permanently from the destination. To avoid accidental destruction of files use the --immutable flag.

Other Helpful Global Flags

Flag used with any RClone command are called global flags. Below are some useful global flags.

* -C or --checksum: Skip syncing files based on checksum instead of last modified time.
* --dry-run: Show what will happen if the command were executed. No changes are made.
* --immutable: Do not allow any files to be modified. Helpful to avoid unintended deletions and overwrites.
* --max-depth <integer>: Only recurse to <integer> depth within directory tree. Using rclone ls --max-depth 1 means only show top-level files in the current directory.
* -P or --progress: Show progress of command as it runs.
* --quiet: Print as little as possible. Useful in scripts.
* -u or --update: Skips files that are newer on the remote.

---

https://rc.byu.edu/wiki/?id=Rclone

Rclone

Last changed on Wed Jan 24 13:54:27 2024

Rclone allows one to move files and directories to and from cloud storage via the command line. In combination with box.byu.edu, where BYU students and faculty get unlimited free storage, it can make storing and backing up archival data much easier. Rclone+Box will help users who routinely run up against storage space constraints and who wish to back up data that can only fit in compute. Those who wish to collaborate without making others get ORC accounts can upload to Box with Rclone, then share their data with collaborators (even if those collaborators don't have Box accounts).

This tutorial will show how to configure Rclone with Box, a few of the most useful commands, and a couple of worked examples. It is by no means comprehensive, so those wanting to learn more should reference the documentation, which is excellent.

Note that while the storage on Box is unlimited, expansive storage comes at a cost: Box is slow (especially with small files), so it can take a while to move big chunks of data; if you have many small files, we recommend backing up with Kopia, which aggregates small files and will significantly speed up transfer. Additionally, files stored on Box cannot exceed 32 GB in size, although rclone easily works around this with its chunker overlay.

   To use Rclone on the supercomputer, you'll need to load the rclone environment module with module load rclone.

Configuration
   Keep in mind that Rclone need only be configured once--as soon as you've finished the steps below, you should never need to do so again as long as you use it at least monthly.

rclone authorize box
   To use Rclone with Box, one needs to get an authorization token, which acts somewhat like a password and allows Rclone to access your files. You'll need a visual environment with a web browser to get this token; we recommend using viz. Open a terminal and run module load rclone && rclone authorize box. Firefox should open, where you can log in with single sign on and get the token. If you're on the supercomputer, you should now be able to access box with rclone--you can confirm with rclone lsd box:.

   After you've run rclone authorize box, you can move on to configuration; the easiest way is to use our default rclone.conf, but you can also run rclone config manually if you want more customization.

The Easy Way
   Rclone stores its data in a configuration file located at ~/.config/rclone/rclone.conf. Since most people will use roughly the same configuration to access files on Box, we provide a default rclone.conf here:

```
[boxRaw]
type = box
token = PASTE_TOKEN_HERE

[box]
type = chunker
remote = boxRaw:
chunk_size = 30G
hash_type = sha1
```

   This describes 2 "remotes": your Box storage (the [boxRaw] section), and a chunker remote that overlays your Box storage (the [box] section). This chunker remote allows files greater than 32GB to be stored on Box by transparently splitting such files (see the Chunker section below).

   Now that you've authorized Box, ~/.config/rclone/rclone.conf should exist; replace it with the file above, keeping the token you find there originally.

   In this config file, the chunker remote uses the base Box directory; if you want to sequester it in a particular directory, just put the directory name after the colon, e.g. remote = boxRaw:ORCChunkerRemote.

$> rclone config
   This is an alternative to using our sample rclone.conf; it will allow you to choose different names and options than those we provide as default.

   To access Rclone, log in to the supercomputer and load the rclone module:

```
$> module load rclone
```

   Once that's done, run rclone config. This will give you a few options:

```
No remotes found - make a new one
n) New remote
s) Set configuration password
q) Quit config
n/s/q> n
```

   Enter n to make a new remote. Give it a name (e.g. boxRaw), then choose which storage service you'd like to configure (you can type box for box.byu.edu, drive for Google Drive, etc.).

   It'll ask for Box App Client Id and Box App Client Secret; most users should simply hit enter to leave these blank. You'll then be asked if you want to "Edit advanced config" (most users should enter n):

```
Edit advanced config? (y/n)
y) Yes
n) No
y/n> n
```

   Next, you will be asked whether to use auto config:

```
Remote config
Use auto config?
 * Say Y if not sure
 * Say N if you are working on a remote or headless machine
y) Yes
n) No
y/n> n
```

   Since you are working on a remote machine, enter n. You will then be presented with a message prompting you to run rclone authorize "box" on your local machine:

```
For this to work, you will need rclone available on a machine that has a web browser available.
Execute the following on your machine:
    rclone authorize "box"
Then paste the result below:
result>
```

   Run rclone authorize "box" in a command prompt on your local machine (see rclone authorize box above)
   and paste the authorization information (of the form
   {"access_token":"XXXXX","token_type":"bearer","refresh_token":"XXXXX","expiry":"2019-01-01T00:00:00-0
   6:00"}) after result> on the remote terminal:
result> {"access_token":"XXXXX","token_type":"bearer","refresh_token":"XXXXX","expiry":"2019-01-01T00:00:00-06:
00"}

[boxRaw]
type = box
client_id =
client_secret =
token = {"access_token":"XXXXX","token_type":"bearer","refresh_token":"XXXXX","expiry":"2019-01-01T00:00:00-06:
00"}
----
y) Yes this is OK
e) Edit this remote
d) Delete this remote
y/e/d> y

   After entering y, you're finished configuring Rclone to work with Box. You'll almost surely want to
   set up a chunker overlay (see Chunker below) and, if you're working with sensitive data, you may
   want to set up an encrypted overlay on top of that (see Crypt below).

Special Remotes
Chunker
   The chunker overlay automatically splits files larger than a certain threshold on upload, which
   allows us to use Box to store files larger than 32GB. When downloading, the split files are
   automatically rejoined. We strongly recommend using chunker--it has no real downsides and preempts
   the frustration of cryptic upload errors when one tries to move a big file up to Box.

   Big files are split into pieces transparently--if you download them outside of the chunker overlay
   and thus don't get the benefit of automatic concatenation, you can just use cat to put them together.
   Split files are named filename.rclone-chunk.000, filename.rclone-chunk.001, etc., so a simple cat
   filename.rclone-chunk.* > filename is all that's needed to combine them. Note that you don't need to
   do this if you access the files via your chunker remote.

   You can copy the [box] section of the example config file in The Easy Way above into your
   /.config/rclone/rclone.conf to create a chunker remote.

Crypt
   The crypt overlay encrypts files in a given remote directory such that they cannot (assuming a
   good password) be read even if someone obtains the files; a downside of this is that you can't simply
   download the files from box.byu.edu and use them directly. As such, we don't recommend the crypt
   overlay for most users. If you do decide to use it, make sure to check whether such a setup satisfies
   any regulations your data is governed by.

   To set up a crypt remote, create a remote directory for it (e.g. box:crypt-remote), run rclone
   config, choose a name (e.g. boxCrypt), crypt storage type, then follow the prompts to finish
   configuring. File name obfuscation has slight usability disadvantages, but "standard" obfuscation is
   the most secure. You'll choose a password and a salt (please don't neglect the salt), and your
   crypt remote is configured.

Usage
   This tutorial will only cover the basics due to the clarity and breadth of Rclone's exceptional
   documentation, which should be your first resource when learning its usage. Typing rclone --help to
   see a good synopsis of each command. For help on a specific command, you can also use rclone
   <command> --help (e.g. rclone copy --help).

Listing files
   Rclone gives a few methods for listing files; none of them are quite like Unix's ls, but rclone lsf
   --max-depth 1 remote:path/to/dir comes close. A few more examples:
Recursively list all files at "box" remote
$> rclone ls box:

Show directories in "mydir" at "box"
$> rclone lsd box:mydir

Recursively list files in "mydir/dir1" at "box" with more detail
$> rclone lsl box:mydir/dir1

Moving and Copying
   rclone copy and rclone move behave a essentially like Unix's cp and mv, respectively; you can copy
   and move to or from the remote. Example usage:
Copy remote file, mydata.txt, from "mydir" at "box"
$> rclone copy box:mydir/mydata.txt $HOME/data/

Move a tarball from compute to "mydir/compute-backup" at "box"
$> rclone move /compute/my-tarball.tar.gz box:mydir/compute-backup

Creating Directories
   rclone mkdir behaves like Unix's mkdir; to create a new directory on a remote, you would use
   something like:
$> rclone mkdir box:mydir/myNewDirectory

Examples
Move Archival Data to Box

Say you have a directory, ~/compute/dataset, with data that needs to be kept, but you don't expect to do any work on it with the supercomputer and you're running out of space. You can either move it directly, or consolidate and compress it then move it. Moving it directly is easier and you'll be able to look at the data directly at box.byu.edu, but compressing then moving could be faster.

Generally, if you have a few big files you won't be slowed down too much by copying directly, but if you have many small files it will take a long time. Under ideal conditions, you can copy 4 files per second (across all processes--Box limits transfers by user). If you have a million files, that means it will take at least a few days to transfer them, no matter how small they each are.

To move without compressing, simply use:

```
$> rclone move ~/compute/dataset box:mydir/dataset
```

There are two main ways to consolidate and compress then move data. This one is slower and more reliable:

```
$> tar -czf dataset.tar.gz ~/compute/dataset
$> rclone move dataset.tar.gz box:mydir/dataset.tar.gz
```

This one is faster and doesn't use significant disk space, but the work will be lost of the command is interrupted:

```
$> tar -czf - ~/compute/dataset | rclone rcat box:mydir/dataset.tar.gz
```

Back up compute with Box

Before backing up directly with Rclone, consider using a full-featured backup tool like Kopia. For most use cases, it will be faster, use less space, and be more secure. The main advantage of using Rclone directly is that your directory structure will be mirrored on Box.

Perhaps you have a large set of data in ~/compute/dataset, which is too big to fit in your home directory, that you would like to back up weekly. Say you set up the following directory structure to store the backups:

```
box:
'-- backup
    '-- dataset
        '-- old
        '-- primary
```

...by running:

```
$> rclone mkdir box:backup
$> rclone mkdir box:backup/dataset
$> rclone mkdir box:backup/dataset/old
# primary will be created by the copy
```

The current backup will live at box:backup/dataset/primary, while older snapshots, organized by date, will go in box:backup/dataset/old/. To get started, let's copy over dataset to the current backup directory at box:backup:

```
$> rclone copy ~/compute/dataset box:backup/dataset/primary
```

Keep in mind that Box is slow, so this may take some time. If you want to exit your ssh session while the copy is going, you may want to use screen or tmux to make the transfer.

Once the copy is done, you'll need to back up every week (or however frequently you would like to). This could go something like:

```
$> module load rclone
$> PRIMARY="box:backup/dataset/primary"
$> OLD="backup/dataset/old/dataset-$(date +%F_%H-%M)"
$> rclone sync "$HOME/compute/dataset" "$PRIMARY" --backup-dir "$OLD"
```

If you want to do this regularly, you can put it in a script and run it at your convenience; you can use cron to run it automatically at regular intervals. To make the script (we'll call it do_rclone_backup.sh) execute weekly, use crontab -e to edit your crontab and enter something along the lines of 0 X * * Y bash /path/to/do_rclone_backup.sh (replacing X with an hour, 0-24, and Y with a day of the week, 0-6). Your backup script will now run once a week with no intervention from you. This tutorial goes into more depth in case you want to back up more or less frequently or would like to learn more about cron generally.

---

https://cloud.garr.it/support/kb/objstore/rclone_quick_tutorial/

Rclone quick tutorial: interaction with object storage

Installing and configuring

Linux:

If not yet installed on your machine, you can install rclone with the following command:

```
$> curl https://rclone.org/install.sh | sudo bash
```

Windows:

In order to use Rclone on Windows systems, you need a bash. If you don't have one yet, you can download git bash from:
https://gitforwindows.org/

Install it and then download rclone from:
https://rclone.org/downloads/

Unzip the archive and open the bash in the rclone directory. From here, you can use rclone command as follow:

```
$> ./rclone

Download OpenStack credentials
   Create and download an application credential from openstack dashboard as app-credentials.sh.

   In order to use Rclone, you can either load the configuration from environment either write them
   directly in the configuration file. Choose the best approach for your needs.

   In both cases, you should edit the Rclone configuration file:
$> nano .rclone.conf

Case 1: Take variables from environment
   Add the following text to rclone.conf:
[garr-cloud]
type = swift
env_auth = true

   Then execute the content of the file:
$> source app-credentials.sh

Case 2: Write variables in the configuration file
   Take note of these three variables in app-credentials.sh:
OS_REGION_NAME
OS_APPLICATION_CREDENTIAL_ID
OS_APPLICATION_CREDENTIAL_SECRET

   Add the following text to rclone.conf:
[garr-cloud]
type = swift
auth = https://keystone.cloud.garr.it:5000/v3/
auth_version = 3
region = <insert here the content of OS_REGION_NAME>
application_credential_id = <insert here the content of OS_APPLICATION_CREDENTIAL_ID>
application_credential_secret = <insert here the content of OS_APPLICATION_CREDENTIAL_SECRET>

   Mind that env_auth = true takes variables from environment, so you shouldn't insert it in this case.

Case 3: Use EC2 credentials
   First, you need to install the Openstack cli as described here in the cli tutorial.

   Then execute the content of the file:
$> source app-credentials.sh

   And create the ec2 credentials:
$> openstack ec2 credentials create

   Take note of these two variables:
+------------+-----------------------+
| Field      | Value                 |
+------------+-----------------------+
| access     | <access_key>          |
| secret     | <secret_key>          |
+------------+-----------------------+

   Add the following text to rclone.conf:
[garr-cloud]
type = s3
provider = AWS
access_key_id =  <insert here the content of access_key>
secret_access_key =  <insert here the content of secret_key>
endpoint = https://swift.cloud.garr.it

   Note
   You can use EC2 credentials to access object storage with other tools. Check S3 interface to
   object storage.

Check configuration
   Note
   On windows systems you have to run rclone inside the executable folder and prepend ./ (i.e. ./rclone)

   Now you can verify your configuration with:
$> rclone lsd garr-cloud:
$> echo $?

   The command above will most probably return no output, but the exit code should be 'zero'.

Check total used space
   Unfortunately, at time of writing rclone capability of interfacing with the Ceph storage system is
   rather limited, and the command does not provide information on quota limits. However, it does
   provide information on the total used space:
$> rclone about garr-cloud:
Used:    190.364M
Objects: 19

Copy files and directories to cloud
   Make a container on your remote object store:
```

```
$> rclone mkdir garr-cloud:test-cont

   Copy a local file to the container:
$> rclone copy sample_file.txt garr-cloud:test-cont
$> rclone ls garr-cloud:test-cont
    1103 sample_file.txt

   Now, suppose you have these files on your local filesystem:
$> ls -lR /tmp/test_dir/
    /tmp/test_dir/:
    total 8
    -rw-rw-r-- 1 ubuntu ubuntu 1103 Nov 13 15:31 file1.txt
    drwxrwxr-x 2 ubuntu ubuntu 4096 Nov 13 15:32 subdir1

    /tmp/test_dir/subdir1:
    total 4
    -rw-rw-r-- 1 ubuntu ubuntu 459 Nov 13 15:32 file2.txt

   Execute the following command to synchronize it with the remote:
$> rclone sync /tmp/test_dir/ garr-cloud:test-cont/sublevel
$> rclone ls garr-cloud:test-cont
    1103 sample_file.txt
    1103 sublevel/file1.txt
    459 sublevel/subdir1/file2.txt

   Mind behaviour of sync! It makes destination identical to source

Copy files from remote to local
   The following command copies files from remote to a local directory, create it if not exists:
$> rclone -P copy garr-cloud:test-cont/sublevel checkDir/
    Transferred:        1.525k / 1.525 kBytes, 100%, 4.222 kBytes/s, ETA 0s
    Errors:                 0
    Checks:                 0 / 0, -
    Transferred:            2 / 2, 100%
    Elapsed time:       300ms

$> ls -lR checkDir/
    checkDir/:
    total 12
    -rw-rw-r-- 1 ubuntu ubuntu 1103 Nov 13 15:31 file1.txt
    -rw-rw-r-- 1 ubuntu ubuntu 1103 Nov 13 15:28 sample_file.txt
    drwxrwxr-x 2 ubuntu ubuntu 4096 Nov 13 15:53 subdir1

    checkDir/subdir1:
    total 4
    -rw-rw-r-- 1 ubuntu ubuntu 459 Nov 13 15:32 file2.txt

Mounting object storage on local filesystem
   Linux:
   First, you need to create a directory on which you will mount your filesystem:
$> mkdir ~/mnt-rclone

   Then you can simply mount your object storage with:
$> rclone -vv --vfs-cache-mode writes mount garr-cloud: ~/mnt-rclone

   Windows:
   First you have to download Winsfp:
   http://www.secfs.net/winfsp/rel/

   WinFsp is an open source Windows File System Proxy which provides a FUSE emulation layer.

   Then you can simply mount your object storage with (no need to create the directory in advance):
$> ./rclone -vv --vfs-cache-mode writes mount garr-cloud: C:/mnt-rclone

   vfs-cache-mode flag enable file caching, you can use either writes or full option. For further
   explanation you can see official documentation at the link:
   https://rclone.org/commands/rclone_mount/#file-caching

   Now that your object storage is mounted, you can list, create and delete files in it.

Unmount object storage
   To unmount, simply press CTRL-C and the mount will be interrupted.

Create encrypted directory
   You can encrypt a directory in your remote container and decrypt it easily through rclone commands.
   First, we create a new remote in the rclone configuration file. It will be a subdirectory of your
   working remote (i.e. garr-cloud remote and crypt-dir directory). So, everything inside
   garr-cloud:crypt-dir will be encrypted and anything outside won't.:
$> nano .rclone.conf

   Copy and paste the following text at the end of the file:
[garr-cloud-crypt]
type = crypt
remote = garr-cloud:crypt_dir
filename_encryption = standard
directory_name_encryption = true
```

```
   Then set the passwords that will be saved obscured inside the config file:
$> rclone config password garr-cloud-crypt password <type_a_password>
$> rclone config password garr-cloud-crypt password2 <type_another_password>

   Now we need to create crypt_dir directory inside garr-cloud:
$> rclone mkdir garr-dev:crypt_dir

   or, if you have garr-cloud still mounted on /mnt-rclone:
$> mkdir /mnt-rclone/crypt_dir

   Every file you will create inside garr-cloud-crypt container will be encrypted. You can try it
   following these steps. First, we create a file to copy:
$> echo "Hello world" > test.txt

   Then, we copy it to the remote, in a new directory named test_dir:
$> rclone copy test.txt garr-cloud-crypt:test_dir

   Now we can list the new created files through the mounted filesystem:
$> cd /mnt-rclone/crypt_dir/
$> ls -R

   You will get an output similar to this:
.:
0ruoo4gjnnuk01p4gok56li8ts

./0ruoo4gjnnuk01p4gok56li8ts:
b4tc2rcdasquuns71k9fa2uiss

   and if you cat the content of the file, you will see that it has been encrypted:
$> cat 0ruoo4gjnnuk01p4gok56li8ts/b4tc2rcdasquuns71k9fa2uiss

   To access and decrypt the file in a complete transparent way, you can copy it from the remote
   through:
$> rclone ls garr-cloud-crypt:

    1048576 test_dir/test.txt

$> rclone copy garr-cloud-crypt: new_dir
$> ls new_dir

    new_dir/:
    test_dir

    new_dir/test_dir:
    test.txt

$> cat new_dir/test_dir/test.txt
    Hello world


---
https://rclone.org/googlephotos/

clone - Google Photos

   The rclone backend for Google Photos is a specialized backend for transferring photos and videos
   to and from Google Photos.

   NB The Google Photos API which rclone uses has quite a few limitations, so please read the
   limitations section carefully to make sure it is suitable for your use.

   NB From March 31, 2025 rclone can only download photos it uploaded. This limitation is due to policy
   changes at Google. You may need to run rclone config reconnect remote: to make rclone work again
   after upgrading to rclone v1.70.

Configuration
   The initial setup for google cloud storage involves getting a token from Google Photos which you need
   to do in your browser. rclone config walks you through it.

   Here is an example of how to make a remote called remote. First run:
$> rclone config

   This will guide you through an interactive setup process:
No remotes found, make a new one?
n) New remote
s) Set configuration password
q) Quit config
n/s/q> n
name> remote
Type of storage to configure.
Enter a string value. Press Enter for the default ("").
Choose a number from below, or type in your own value
[snip]
XX / Google Photos
   \ "google photos"
```

```
[snip]
Storage> google photos
** See help for google photos backend at: https://rclone.org/googlephotos/ **

Google Application Client Id
Leave blank normally.
Enter a string value. Press Enter for the default ("").
client_id>
Google Application Client Secret
Leave blank normally.
Enter a string value. Press Enter for the default ("").
client_secret>
Set to make the Google Photos backend read only.

If you choose read only then rclone will only request read only access
to your photos, otherwise rclone will request full access.
Enter a boolean value (true or false). Press Enter for the default ("false").
read_only>
Edit advanced config? (y/n)
y) Yes
n) No
y/n> n
Remote config
Use web browser to automatically authenticate rclone with remote?
 * Say Y if the machine running rclone has a web browser you can use
 * Say N if running rclone on a (remote) machine without web browser access
If not sure try Y. If Y failed, try N.
y) Yes
n) No
y/n> y
If your browser doesn't open automatically go to the following link: http://127.0.0.1:53682/auth
Log in and authorize rclone for access
Waiting for code...
Got code

*** IMPORTANT: All media items uploaded to Google Photos with rclone
*** are stored in full resolution at original quality.  These uploads
*** will count towards storage in your Google Account.

Configuration complete.
Options:
- type: google photos
- token: {"access_token":"XXX","token_type":"Bearer","refresh_token":"XXX","expiry":"2019-06-28T17:38:04.64493
0156+01:00"}
Keep this "remote" remote?
y) Yes this is OK
e) Edit this remote
d) Delete this remote
y/e/d> y

   See the remote setup docs for how to set it up on a machine with no Internet browser available.

   Note that rclone runs a webserver on your local machine to collect the token as returned from Google
   if using web browser to automatically authenticate. This only runs from the moment it opens your
   browser to the moment you get back the verification code. This is on http://127.0.0.1:53682/ and this
   may require you to unblock it temporarily if you are running a host firewall, or use manual mode.

   This remote is called remote and can now be used like this

   See all the albums in your photos
$> rclone lsd remote:album

   Make a new album
$> rclone mkdir remote:album/newAlbum

   List the contents of an album
$> rclone ls remote:album/newAlbum

   Sync /home/local/images to the Google Photos, removing any excess files in the album.
$> rclone sync --interactive /home/local/image remote:album/newAlbum

Layout
   As Google Photos is not a general purpose cloud storage system, the backend is laid out to help you
   navigate it.

   The directories under media show different ways of categorizing the media. Each file will appear
   multiple times. So if you want to make a backup of your google photos you might choose to backup
   remote:media/by-month. (NB remote:media/by-day is rather slow at the moment so avoid for syncing.)

   Note that all your photos and videos will appear somewhere under media, but they may not appear under
   album unless you've put them into albums.
/
- upload
    - file1.jpg
    - file2.jpg
    - ...
```

```
- media
    - all
        - file1.jpg
        - file2.jpg
        - ...
    - by-year
        - 2000
            - file1.jpg
            - ...
        - 2001
            - file2.jpg
            - ...
        - ...
    - by-month
        - 2000
            - 2000-01
                - file1.jpg
                - ...
            - 2000-02
                - file2.jpg
                - ...
        - ...
    - by-day
        - 2000
            - 2000-01-01
                - file1.jpg
                - ...
            - 2000-01-02
                - file2.jpg
                - ...
        - ...
- album
    - album name
    - album name/sub
- shared-album
    - album name
    - album name/sub
- feature
    - favorites
        - file1.jpg
        - file2.jpg
```

There are two writable parts of the tree, the upload directory and sub directories of the album directory.

The upload directory is for uploading files you don't want to put into albums. This will be empty to start with and will contain the files you've uploaded for one rclone session only, becoming empty again when you restart rclone. The use case for this would be if you have a load of files you just want to once off dump into Google Photos. For repeated syncing, uploading to album will work better.

Directories within the album directory are also writeable and you may create new directories (albums) under album. If you copy files with a directory hierarchy in there then rclone will create albums with the / character in them. For example if you do

```
$> rclone copy /path/to/images remote:album/images
```

and the images directory contains

```
images
    - file1.jpg
    dir
        file2.jpg
    dir2
        dir3
            file3.jpg
```

Then rclone will create the following albums with the following files in

- images
    - file1.jpg
- images/dir
    - file2.jpg
- images/dir2/dir3
    - file3.jpg

This means that you can use the album path pretty much like a normal filesystem and it is a good target for repeated syncing.

The shared-album directory shows albums shared with you or by you. This is similar to the Sharing tab in the Google Photos web interface.

Standard options

Here are the Standard options specific to google photos (Google Photos).

--gphotos-client-id

OAuth Client Id.

Leave blank normally.

```
   Properties:
     * Config: client_id
     * Env Var: RCLONE_GPHOTOS_CLIENT_ID
     * Type: string
     * Required: false

--gphotos-client-secret
   OAuth Client Secret.

   Leave blank normally.

   Properties:
     * Config: client_secret
     * Env Var: RCLONE_GPHOTOS_CLIENT_SECRET
     * Type: string
     * Required: false

--gphotos-read-only
   Set to make the Google Photos backend read only.

   If you choose read only then rclone will only request read only access to your photos, otherwise
   rclone will request full access.

   Properties:
     * Config: read_only
     * Env Var: RCLONE_GPHOTOS_READ_ONLY
     * Type: bool
     * Default: false

Advanced options
   Here are the Advanced options specific to google photos (Google Photos).

--gphotos-token
   OAuth Access Token as a JSON blob.

   Properties:
     * Config: token
     * Env Var: RCLONE_GPHOTOS_TOKEN
     * Type: string
     * Required: false

--gphotos-auth-url
   Auth server URL.

   Leave blank to use the provider defaults.

   Properties:
     * Config: auth_url
     * Env Var: RCLONE_GPHOTOS_AUTH_URL
     * Type: string
     * Required: false

--gphotos-token-url
   Token server url.

   Leave blank to use the provider defaults.

   Properties:
     * Config: token_url
     * Env Var: RCLONE_GPHOTOS_TOKEN_URL
     * Type: string
     * Required: false

--gphotos-client-credentials
   Use client credentials OAuth flow.

   This will use the OAUTH2 client Credentials Flow as described in RFC 6749.

   Note that this option is NOT supported by all backends.

   Properties:
     * Config: client_credentials
     * Env Var: RCLONE_GPHOTOS_CLIENT_CREDENTIALS
     * Type: bool
     * Default: false

--gphotos-read-size
   Set to read the size of media items.

   Normally rclone does not read the size of media items since this takes another transaction. This
   isn't necessary for syncing. However rclone mount needs to know the size of files in advance of
   reading them, so setting this flag when using rclone mount is recommended if you want to read the
   media.

   Properties:
     * Config: read_size
```

```
     * Env Var: RCLONE_GPHOTOS_READ_SIZE
     * Type: bool
     * Default: false

--gphotos-start-year
   Year limits the photos to be downloaded to those which are uploaded after the given year.

   Properties:
     * Config: start_year
     * Env Var: RCLONE_GPHOTOS_START_YEAR
     * Type: int
     * Default: 2000

--gphotos-include-archived
   Also view and download archived media.

   By default, rclone does not request archived media. Thus, when syncing, archived media is not visible
   in directory listings or transferred.

   Note that media in albums is always visible and synced, no matter their archive status.

   With this flag, archived media are always visible in directory listings and transferred.

   Without this flag, archived media will not be visible in directory listings and won't be transferred.

   Properties:
     * Config: include_archived
     * Env Var: RCLONE_GPHOTOS_INCLUDE_ARCHIVED
     * Type: bool
     * Default: false

--gphotos-proxy
   Use the gphotosdl proxy for downloading the full resolution images

   The Google API will deliver images and video which aren't full resolution, and/or have EXIF data
   missing.

   However if you use the gphotosdl proxy then you can download original, unchanged images.

   This runs a headless browser in the background.

   Download the software from gphotosdl

   First run with
$> gphotosdl -login

   Then once you have logged into google photos close the browser window and run
$> gphotosdl

   Then supply the parameter --gphotos-proxy "http://localhost:8282" to make rclone use the proxy.

   Properties:
     * Config: proxy
     * Env Var: RCLONE_GPHOTOS_PROXY
     * Type: string
     * Required: false

--gphotos-encoding
   The encoding for the backend.

   See the encoding section in the overview for more info.

   Properties:
     * Config: encoding
     * Env Var: RCLONE_GPHOTOS_ENCODING
     * Type: Encoding
     * Default: Slash,CrLf,InvalidUtf8,Dot

--gphotos-batch-mode
   Upload file batching sync|async|off.

   This sets the batch mode used by rclone.

   This has 3 possible values
     * off - no batching
     * sync - batch uploads and check completion (default)
     * async - batch upload and don't check completion

   Rclone will close any outstanding batches when it exits which may make a delay on quit.

   Properties:
     * Config: batch_mode
     * Env Var: RCLONE_GPHOTOS_BATCH_MODE
     * Type: string
     * Default: "sync"
```

--gphotos-batch-size
   Max number of files in upload batch.

   This sets the batch size of files to upload. It has to be less than 50.

   By default this is 0 which means rclone will calculate the batch size depending on the setting of
   batch_mode.
     * batch_mode: async - default batch_size is 50
     * batch_mode: sync - default batch_size is the same as --transfers
     * batch_mode: off - not in use

   Rclone will close any outstanding batches when it exits which may make a delay on quit.

   Setting this is a great idea if you are uploading lots of small files as it will make them a lot
   quicker. You can use --transfers 32 to maximise throughput.

   Properties:
     * Config: batch_size
     * Env Var: RCLONE_GPHOTOS_BATCH_SIZE
     * Type: int
     * Default: 0

--gphotos-batch-timeout
   Max time to allow an idle upload batch before uploading.

   If an upload batch is idle for more than this long then it will be uploaded.

   The default for this is 0 which means rclone will choose a sensible default based on the batch_mode
   in use.
     * batch_mode: async - default batch_timeout is 10s
     * batch_mode: sync - default batch_timeout is 1s
     * batch_mode: off - not in use

   Properties:
     * Config: batch_timeout
     * Env Var: RCLONE_GPHOTOS_BATCH_TIMEOUT
     * Type: Duration
     * Default: 0s

--gphotos-batch-commit-timeout
   Max time to wait for a batch to finish committing. (no longer used)

   Properties:
     * Config: batch_commit_timeout
     * Env Var: RCLONE_GPHOTOS_BATCH_COMMIT_TIMEOUT
     * Type: Duration
     * Default: 10m0s

--gphotos-description
   Description of the remote.

   Properties:
     * Config: description
     * Env Var: RCLONE_GPHOTOS_DESCRIPTION
     * Type: string
     * Required: false

Limitations
   Only images and videos can be uploaded. If you attempt to upload non videos or images or formats that
   Google Photos doesn't understand, rclone will upload the file, then Google Photos will give an error
   when it is put turned into a media item.

   NB From March 31, 2025 rclone can only download photos it uploaded. This limitation is due to policy
   changes at Google. You may need to run rclone config reconnect remote: to make rclone work again
   after upgrading to rclone v1.70.

   Note that all media items uploaded to Google Photos through the API are stored in full resolution at
   "original quality" and will count towards your storage quota in your Google Account. The API does not
   offer a way to upload in "high quality" mode..

   rclone about is not supported by the Google Photos backend. Backends without this capability cannot
   determine free space for an rclone mount or use policy mfs (most free space) as a member of an rclone
   union remote.

   See List of backends that do not support rclone about See rclone about

Downloading Images
   When Images are downloaded this strips EXIF location (according to the docs and my tests). This is a
   limitation of the Google Photos API and is covered by bug #112096115.

   The current google API does not allow photos to be downloaded at original resolution. This is very
   important if you are, for example, relying on "Google Photos" as a backup of your photos. You will
   not be able to use rclone to redownload original images. You could use 'google takeout' to recover
   the original photos as a last resort

   NB you can use the --gphotos-proxy flag to use a headless browser to download images in full

resolution.

Downloading Videos
When videos are downloaded they are downloaded in a really compressed version of the video compared to downloading it via the Google Photos web interface. This is covered by bug #113672044.

NB you can use the --gphotos-proxy flag to use a headless browser to download images in full resolution.

Duplicates
If a file name is duplicated in a directory then rclone will add the file ID into its name. So two files called file.jpg would then appear as file {123456}.jpg and file {ABCDEF}.jpg (the actual IDs are a lot longer alas!).

If you upload the same image (with the same binary data) twice then Google Photos will deduplicate it. However it will retain the filename from the first upload which may confuse rclone. For example if you uploaded an image to upload then uploaded the same image to album/my_album the filename of the image in album/my_album will be what it was uploaded with initially, not what you uploaded it with to album. In practise this shouldn't cause too many problems.

Modification times
The date shown of media in Google Photos is the creation date as determined by the EXIF information, or the upload date if that is not known.

This is not changeable by rclone and is not the modification date of the media on local disk. This means that rclone cannot use the dates from Google Photos for syncing purposes.

Size
The Google Photos API does not return the size of media. This means that when syncing to Google Photos, rclone can only do a file existence check.

It is possible to read the size of the media, but this needs an extra HTTP HEAD request per media item so is very slow and uses up a lot of transactions. This can be enabled with the --gphotos-read-size option or the read_size = true config parameter.

If you want to use the backend with rclone mount you may need to enable this flag (depending on your OS and application using the photos) otherwise you may not be able to read media off the mount. You'll need to experiment to see if it works for you without the flag.

Albums
Rclone can only upload files to albums it created. This is a limitation of the Google Photos API.

Rclone can remove files it uploaded from albums it created only.

Deleting files
Rclone can remove files from albums it created, but note that the Google Photos API does not allow media to be deleted permanently so this media will still remain. See bug #109759781.

Rclone cannot delete files anywhere except under album.

Deleting albums
The Google Photos API does not support deleting albums - see bug #135714733.
