git-fast-import(1) Manual Page

NAME

git-fast-import - Backend for fast Git data importers

SYNOPSIS

```
frontend | git fast-import [<options>]
```

DESCRIPTION

This program is usually not what the end user wants to run directly. Most end users want to use one of the existing frontend programs, which parses a specific type of foreign source and feeds the contents stored there to *git fast-import*.

fast-import reads a mixed command/data stream from standard input and writes one or more packfiles directly into the current repository. When EOF is received on standard input, fast import writes out updated branch and tag refs, fully updating the current repository with the newly imported data.

The fast-import backend itself can import into an empty repository (one that has already been initialized by *git init*) or incrementally update an existing populated repository. Whether or not incremental imports are supported from a particular foreign source depends on the frontend program in use.

OPTIONS

--force

Force updating modified existing branches, even if doing so would cause commits to be lost (as the new commit does not contain the old commit).

--quiet

Disable the output shown by --stats, making fast-import usually be silent when it is successful. However, if the import stream has directives intended to show user output (e.g. `progress` directives), the corresponding messages will still be shown.

--stats

Display some basic statistics about the objects fast-import has created, the packfiles they were stored into, and the memory used by fast-import during this run. Showing this output is currently the default, but can be disabled with --quiet.

--allow-unsafe-features

Many command-line options can be provided as part of the fast-import stream itself by using the `feature` or `option` commands. However, some of these options are unsafe (e.g., allowing fast-import to access the filesystem outside of the repository). These options are disabled by default, but can be allowed by providing this option on the command line. This currently impacts only the `export-marks`, `import-marks`, and `import-marks-if-exists` feature commands.

```
Only enable this option if you trust the program generating the
fast-import stream! This option is enabled automatically for
remote-helpers that use the `import` capability, as they are
already trusted to run their own code.
```

Options for Frontends

--cat-blob-fd=<fd>

Write responses to `get-mark`, `cat-blob`, and `ls` queries to the file descriptor <fd> instead of `stdout`. Allows `progress` output intended for the end-user to be separated from other output.

--date-format=<fmt>

Specify the type of dates the frontend will supply to fast-import within `author`, `committer` and `tagger` commands. See “Date Formats” below for details about which formats are supported, and their syntax.

--done

Terminate with error if there is no `done` command at the end of the stream. This option might be useful for detecting errors that cause the frontend to terminate before it has started to write a stream.

Locations of Marks Files

--export-marks=<file>

Dumps the internal marks table to <file> when complete. Marks are written one per line as `:markid SHA-1`. Frontends can use this file to validate imports after they have been completed, or to save the marks table across incremental runs. As <file> is only opened and truncated at checkpoint (or completion) the same path can also be safely given to --import-marks.

--import-marks=<file>

Before processing any input, load the marks specified in <file>. The input file must exist, must be readable, and must use the same format as produced by --export-marks. Multiple options may be supplied to import more than one set of marks. If a mark is defined to different values, the last file wins.

--import-marks-if-exists=<file>

Like --import-marks but instead of erroring out, silently skips the file if it does not exist.

--[no-]relative-marks

After specifying --relative-marks the paths specified with --import-marks= and --export-marks= are relative to an internal directory in the current repository. In git-fast-import this means that the paths are relative to the .git/info/fast-import directory. However, other importers may use a different location.

Relative and non-relative marks may be combined by interweaving --(no-)-relative-marks with the --(import|export)-marks= options.

Submodule Rewriting

--rewrite-submodules-from=<name>:<file>

--rewrite-submodules-to=<name>:<file>

Rewrite the object IDs for the submodule specified by <name> from the values used in the from <file> to those used in the to <file>. The from marks should have been created by `git fast-export`, and the to marks should have been created by `git fast-import` when importing that same submodule.

<name> may be any arbitrary string not containing a colon character, but the same value must be used with both options when specifying corresponding marks. Multiple submodules may be specified with different values for <name>. It is an error not to use these options in corresponding pairs.

These options are primarily useful when converting a repository from one hash algorithm to another; without them, fast-import will fail if it encounters a submodule because it has no way of writing the object ID into the new hash algorithm.

Performance and Compression Tuning

--active-branches=<n>

Maximum number of branches to maintain active at once. See “Memory Utilization” below for details. Default is 5.

--big-file-threshold=<n>

Maximum size of a blob that fast-import will attempt to create a delta for, expressed in bytes. The default is 512m (512 MiB). Some importers may wish to lower this on systems with constrained memory.

--depth=<n>

Maximum delta depth, for blob and tree deltification. Default is 50.

--export-pack-edges=<file>

After creating a packfile, print a line of data to <file> listing the filename of the packfile and the last commit on each branch that was written to that packfile. This information may be useful after importing projects whose total object set exceeds the 4 GiB packfile limit, as these commits can be used as edge points during calls to *git pack-objects*.

--max-pack-size=<n>

Maximum size of each output packfile. The default is unlimited.

fastimport.unpackLimit

See git-config(1)

PERFORMANCE

The design of fast-import allows it to import large projects in a minimum amount of memory usage and processing time. Assuming the frontend is able to keep up with fast-import and feed it a constant stream of data, import times for projects holding 10+ years of history and containing 100,000+ individual commits are generally completed in just 1-2 hours on quite modest (~$2,000 USD) hardware.

Most bottlenecks appear to be in foreign source data access (the source just cannot extract revisions fast enough) or disk IO (fast-import writes as fast as the disk will take the data). Imports will run faster if the source data is stored on a different drive than the destination Git repository (due to less IO contention).

DEVELOPMENT COST

A typical frontend for fast-import tends to weigh in at approximately 200 lines of Perl/Python/Ruby code. Most developers have been able to create working importers in just a couple of hours, even though it is their first exposure to fast-import, and sometimes even to Git. This is an ideal situation, given that most conversion tools are throw-away (use once, and never look back).

PARALLEL OPERATION

Like *git push* or *git fetch*, imports handled by fast-import are safe to run alongside parallel `git repack -a -d` or `git gc` invocations, or any other Git operation (including *git prune*, as loose objects are never used by fast-import).

fast-import does not lock the branch or tag refs it is actively importing. After the import, during its ref update phase, fast-import tests each existing branch ref to verify the update will be a fast-forward update (the commit stored in the ref is contained in the new history of the commit to be written). If the update is not a fast-forward update, fast-import will skip updating that ref and instead prints a warning message. fast-import will always attempt to update all branch refs, and does not stop on the first failure.

Branch updates can be forced with --force, but it’s recommended that this only be used on an otherwise quiet repository. Using --force is not necessary for an initial import into an empty repository.

TECHNICAL DISCUSSION

fast-import tracks a set of branches in memory. Any branch can be created or modified at any point during the import process by sending a `commit` command on the input stream. This design allows a frontend program to process an unlimited number of branches simultaneously, generating commits in the order they are available from the source data. It also simplifies the frontend programs considerably.

fast-import does not use or alter the current working directory, or any file within it. (It does however update the current Git repository, as referenced by `GIT_DIR`.) Therefore an import frontend may use the working directory for its own purposes, such as extracting file revisions from the foreign source. This ignorance of the working directory also allows fast-import to run very quickly, as it does not need to perform any costly file update operations when switching between branches.

INPUT FORMAT

With the exception of raw file data (which Git does not interpret) the fast-import input format is text (ASCII) based. This text based format simplifies development and debugging of frontend programs, especially when a higher level language such as Perl, Python or Ruby is being used.

fast-import is very strict about its input. Where we say SP below we mean**exactly** one space. Likewise LF means one (and only one) linefeed and HT one (and only one) horizontal tab. Supplying additional whitespace characters will cause unexpected results, such as branch names or file names with leading or trailing spaces in their name, or early termination of fast-import when it encounters unexpected input.

Stream Comments

To aid in debugging frontends fast-import ignores any line that begins with `#` (ASCII pound/hash) up to and including the line ending `LF`. A comment line may contain any sequence of bytes that does not contain an LF and therefore may be used to include any detailed debugging information that might be specific to the frontend and useful when inspecting a fast-import data stream.

Date Formats

The following date formats are supported. A frontend should select the format it will use for this import by passing the format name in the --date-format=<fmt> command-line option.

`raw`

This is the Git native format and is `<time> SP <offutc>`. It is also fast-import's default format, if --date-format was not specified.

The time of the event is specified by `<time>` as the number of seconds since the UNIX epoch (midnight, Jan 1, 1970, UTC) and is written as an ASCII decimal integer.

The local offset is specified by `<offutc>` as a positive or negative offset from UTC. For example EST (which is 5 hours behind UTC) would be expressed in `<tz>` by "-0500" while UTC is "+0000". The local offset does not affect `<time>`; it is used only as an advisement to help formatting routines display the timestamp.

If the local offset is not available in the source material, use "+0000", or the most common local offset. For example many organizations have a CVS repository which has only ever been accessed by users who are located in the same location and time zone. In this case a reasonable offset from UTC could be assumed.

Unlike the `rfc2822` format, this format is very strict. Any variation in formatting will cause fast-import to reject the value, and some sanity checks on the numeric values may also be performed.

`raw-permissive`

This is the same as `raw` except that no sanity checks on the numeric epoch and local offset are performed. This can be useful when trying to filter or import an existing history with e.g. bogus timezone values.

`rfc2822`

This is the standard email format as described by RFC 2822.

An example value is “Tue Feb 6 11:22:18 2007 -0500”. The Git parser is accurate, but a little on the lenient side. It is the same parser used by *git am* when applying patches received from email.

Some malformed strings may be accepted as valid dates. In some of these cases Git will still be able to obtain the correct date from the malformed string. There are also some types of malformed strings which Git will parse wrong, and yet consider valid. Seriously malformed strings will be rejected.

Unlike the `raw` format above, the time zone/UTC offset information contained in an RFC 2822 date string is used to adjust the date value to UTC prior to storage. Therefore it is important that this information be as accurate as possible.

If the source material uses RFC 2822 style dates, the frontend should let fast-import handle the parsing and conversion (rather than attempting to do it itself) as the Git parser has been well tested in the wild.

Frontends should prefer the `raw` format if the source material already uses UNIX-epoch format, can be coaxed to give dates in that format, or its format is easily convertible to it, as there is no ambiguity in parsing.

`now`

Always use the current time and time zone. The literal `now` must always be supplied for `<when>`.

This is a toy format. The current time and time zone of this system is always copied into the identity string at the time it is being created by fast-import. There is no way to specify a different time or time zone.

This particular format is supplied as it’s short to implement and may be useful to a process that wants to create a new commit right now, without needing to use a working directory or *git update-index*.

If separate `author` and `committer` commands are used in a `commit` the timestamps may not match, as the system clock will be polled twice (once for each command). The only way to ensure that both author and committer identity information has the same timestamp is to omit `author` (thus copying from `committer`) or to use a date format other than `now`.

Commands

fast-import accepts several commands to update the current repository and control the current import process. More detailed discussion (with examples) of each command follows later.

`commit`

Creates a new branch or updates an existing branch by creating a new commit and updating the branch to point at the newly created commit.

`tag`

Creates an annotated tag object from an existing commit or branch. Lightweight tags are not supported by this command, as they are not recommended for recording meaningful points in time.

`reset`

Reset an existing branch (or a new branch) to a specific revision. This command must be used to change a branch to a specific revision without making a commit on it.

`blob`

Convert raw file data into a blob, for future use in a `commit` command. This command is optional and is not needed to perform an import.

`alias`

Record that a mark refers to a given object without first creating any new object. Using --import-marks and referring to missing marks will cause fast-import to fail, so aliases can provide a way to set otherwise pruned commits to a valid value (e.g. the nearest non-pruned ancestor).

`checkpoint`

Forces fast-import to close the current packfile, generate its unique SHA-1 checksum and index, and start a new packfile. This command is optional and is not needed to perform an import.

`progress`

Causes fast-import to echo the entire line to its own standard output. This command is optional and is not needed to perform an import.

`done`

Marks the end of the stream. This command is optional unless the `done` feature was requested using the `--done` command-line option or `feature done` command.

`get-mark`

Causes fast-import to print the SHA-1 corresponding to a mark to the file descriptor set with `--cat-blob-fd`, or `stdout` if unspecified.

`cat-blob`

Causes fast-import to print a blob in *cat-file --batch* format to the file descriptor set with `--cat-blob-fd` or `stdout` if unspecified.

`ls`

Causes fast-import to print a line describing a directory entry in *ls-tree*format to the file descriptor set with `--cat-blob-fd` or `stdout` if unspecified.

`feature`

Enable the specified feature. This requires that fast-import supports the specified feature, and aborts if it does not.

`option`

Specify any of the options listed under OPTIONS that do not change stream semantic to suit the frontend's needs. This command is optional and is not needed to perform an import.

`commit`

Create or update a branch with a new commit, recording one logical change to the project.

```
	'commit' SP <ref> LF
	mark?
	original-oid?
	('author' (SP <name>)? SP LT <email> GT SP <when> LF)?
	'committer' (SP <name>)? SP LT <email> GT SP <when> LF
	('encoding' SP <encoding>)?
	data
	('from' SP <commit-ish> LF)?
	('merge' SP <commit-ish> LF)*
	(filemodify | filedelete | filecopy | filerename | filedeleteall |
notemodify)*
	LF?
```

where `<ref>` is the name of the branch to make the commit on. Typically branch names are prefixed with `refs/heads/` in Git, so importing the CVS branch symbol `RELENG-1_0` would use `refs/heads/RELENG-1_0` for the value of `<ref>`. The value of `<ref>` must be a valid refname in Git. As `LF` is not valid in a Git refname, no quoting or escaping syntax is supported here.

A `mark` command may optionally appear, requesting fast-import to save a reference to the newly created commit for future use by the frontend (see below for format). It is very common for frontends to mark every commit they create, thereby allowing future branch creation from any imported commit.

The `data` command following `committer` must supply the commit message (see below for `data` command syntax). To import an empty commit message use a 0 length data. Commit messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Zero or more `filemodify`, `filedelete`, `filecopy`, `filerename`, `filedeleteall` and `notemodify` commands may be included to update the contents of the branch prior to creating the commit. These commands may be supplied in any order. However it is recommended that a `filedeleteall` command precede all `filemodify`, `filecopy`, `filerename` and `notemodify` commands in the same commit, as `filedeleteall` wipes the branch clean (see below).

The `LF` after the command is optional (it used to be required). Note that for reasons of backward compatibility, if the commit ends with a `data` command (i.e. it has no `from`, `merge`, `filemodify`, `filedelete`, `filecopy`, `filerename`, `filedeleteall` or `notemodify` commands) then two `LF` commands may appear at the end of the command instead of just one.

`author`

An `author` command may optionally appear, if the author information might differ from the committer information. If `author` is omitted then fast-import will automatically use the committer's information for the author portion of the commit. See below for a description of the fields in `author`, as they are identical to `committer`.

`committer`

The `committer` command indicates who made this commit, and when they made it.

Here `<name>` is the person's display name (for example "Com M Itter") and `<email>` is the person's email address ("cm@example.com"). `LT` and `GT` are the literal less-than (\x3c) and greater-than (\x3e) symbols. These are required to delimit the email address from the other fields in the line. Note that `<name>` and `<email>` are free-form and may contain any sequence of bytes, except `LT`, `GT` and `LF`. `<name>` is typically UTF-8 encoded.

The time of the change is specified by `<when>` using the date format that was selected by the --date-format=<fmt> command-line option. See "Date Formats" above for the set of supported formats, and their syntax.

`encoding`

The optional `encoding` command indicates the encoding of the commit message. Most commits are UTF-8 and the encoding is omitted, but this allows importing commit messages into git without first reencoding them.

`from`

The `from` command is used to specify the commit to initialize this branch from. This revision will be the first ancestor of the new commit. The state of the tree built at this commit will begin with the state at the `from` commit, and be altered by the content modifications in this commit.

Omitting the `from` command in the first commit of a new branch will cause fast-import to create that commit with no ancestor. This tends to be desired only for the initial commit of a project. If the frontend creates all files from scratch when making a new branch, a `merge` command may be used instead of `from` to start the commit with an empty tree. Omitting the `from` command on existing branches is usually desired, as the current commit on that branch is automatically assumed to be the first ancestor of the new commit.

As `LF` is not valid in a Git refname or SHA-1 expression, no quoting or escaping syntax is supported within `<commit-ish>`.

Here `<commit-ish>` is any of the following:

- The name of an existing branch already in fast-import's internal branch table. If fast-import doesn't know the name, it's treated as a SHA-1 expression.
- A mark reference, `:<idnum>`, where `<idnum>` is the mark number.

  The reason fast-import uses `:` to denote a mark reference is this character is not legal in a Git branch name. The leading `:` makes it easy to distinguish between the mark 42 (`:42`) and the branch 42 (`42` or `refs/heads/42`), or an abbreviated SHA-1 which happened to consist only of base-10 digits.

  Marks must be declared (via `mark`) before they can be used.
- A complete 40 byte or abbreviated commit SHA-1 in hex.
- Any valid Git SHA-1 expression that resolves to a commit. See "SPECIFYING REVISIONS" in gitrevisions(7) for details.
- The special null SHA-1 (40 zeros) specifies that the branch is to be removed.

The special case of restarting an incremental import from the current branch value should be written as:

```
	from refs/heads/branch^0
```

The `^0` suffix is necessary as fast-import does not permit a branch to start from itself, and the branch is created in memory before the `from` command is even read from the input. Adding `^0` will force fast-import to resolve the commit through Git's revision parsing library, rather than its internal branch table, thereby loading in the existing value of the branch.

`merge`

Includes one additional ancestor commit. The additional ancestry link does not change the way the tree state is built at this commit. If the `from` command is omitted when creating a new branch, the first `merge` commit will be the first ancestor of the current commit, and the branch will start out with no files. An unlimited number of `merge` commands per commit are permitted by fast-import, thereby establishing an n-way merge.

Here `<commit-ish>` is any of the commit specification expressions also accepted by `from` (see above).

`filemodify`

Included in a `commit` command to add a new file or change the content of an existing file. This command has two different means of specifying the content of the file.

External data format

The data content for the file was already supplied by a prior `blob` command. The frontend just needs to connect it.

```
	'M' SP <mode> SP <dataref> SP <path> LF
```

Here usually `<dataref>` must be either a mark reference (`:<idnum>`) set by a prior `blob` command, or a full 40-byte SHA-1 of an existing Git blob object. If `<mode>` is `040000`` then `<dataref>` must be the full 40-byte SHA-1 of an existing Git tree object or a mark reference set with `--import-marks`.

Inline data format

The data content for the file has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
	'M' SP <mode> SP 'inline' SP <path> LF
	data
```

See below for a detailed description of the `data` command.

In both formats `<mode>` is the type of file entry, specified in octal. Git only supports the following modes:

- `100644` or `644`: A normal (not-executable) file. The majority of files in most projects use this mode. If in doubt, this is what you want.
- `100755` or `755`: A normal, but executable, file.
- `120000`: A symlink, the content of the file will be the link target.
- `160000`: A gitlink, SHA-1 of the object refers to a commit in another repository. Git links can only be specified by SHA or through a commit mark. They are used to implement submodules.
- `040000`: A subdirectory. Subdirectories can only be specified by SHA or through a tree mark set with `--import-marks`.

In both formats `<path>` is the complete path of the file to be added (if not already existing) or modified (if already existing).

A `<path>` string must use UNIX-style directory separators (forward slash `/`), may contain any byte other than `LF`, and must not start with double quote (`"`).

A path can use C-style string quoting; this is accepted in all cases and mandatory if the filename starts with double quote or contains `LF`. In C-style quoting, the complete name should be surrounded with double quotes, and any `LF`, backslash, or double quote characters must be escaped by preceding them with a backslash (e.g., `"path/with\n, \\ and \" in it"`).

The value of `<path>` must be in canonical form. That is it must not:

- contain an empty directory component (e.g. `foo//bar` is invalid),
- end with a directory separator (e.g. `foo/` is invalid),
- start with a directory separator (e.g. `/foo` is invalid),

- contain the special component `.` or `..` (e.g. `foo/./bar` and `foo/../bar` are invalid).

The root of the tree can be represented by an empty string as `<path>`.

It is recommended that `<path>` always be encoded using UTF-8.

`filedelete`

Included in a `commit` command to remove a file or recursively delete an entire directory from the branch. If the file or directory removal makes its parent directory empty, the parent directory will be automatically removed too. This cascades up the tree until the first non-empty directory or the root is reached.

```
	'D' SP <path> LF
```

here `<path>` is the complete path of the file or subdirectory to be removed from the branch. See `filemodify` above for a detailed description of `<path>`.

`filecopy`

Recursively copies an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be completely replaced by the content copied from the source.

```
	'C' SP <path> SP <path> LF
```

here the first `<path>` is the source location and the second `<path>` is the destination. See `filemodify` above for a detailed description of what `<path>` may look like. To use a source path that contains SP the path must be quoted.

A `filecopy` command takes effect immediately. Once the source location has been copied to the destination any future commands applied to the source location will not impact the destination of the copy.

`filerename`

Renames an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be replaced by the source directory.

```
	'R' SP <path> SP <path> LF
```

here the first `<path>` is the source location and the second `<path>` is the destination. See `filemodify` above for a detailed description of what `<path>` may look like. To use a source path that contains SP the path must be quoted.

A `filerename` command takes effect immediately. Once the source location has been renamed to the destination any future commands applied to the source location will create new files there and not impact the destination of the rename.

Note that a `filerename` is the same as a `filecopy` followed by a `filedelete` of the source location. There is a slight performance advantage to using `filerename`, but the advantage is so small that it is never worth trying to convert a delete/add pair in source material into a rename for fast-import. This `filerename` command is provided just to simplify frontends that already have rename information and don’t want bother with decomposing it into a `filecopy` followed by a `filedelete`.

`filedeleteall`

Included in a `commit` command to remove all files (and also all directories) from the branch. This command resets the internal branch structure to have no files in it, allowing the frontend to subsequently add all interesting files from scratch.

```
	'deleteall' LF
```

This command is extremely useful if the frontend does not know (or does not care to know) what files are currently on the branch, and therefore cannot generate the proper `filedelete` commands to update the content.

Issuing a `filedeleteall` followed by the needed `filemodify` commands to set the correct content will produce the same results as sending only the needed `filemodify` and `filedelete` commands. The `filedeleteall` approach may however require fast-import to use slightly more memory per active branch (less than 1 MiB for even most large projects); so frontends that can easily obtain only the affected paths for a commit are encouraged to do so.

`notemodify`

Included in a `commit` `<notes_ref>` command to add a new note annotating a `<commit-ish>` or change this annotation contents. Internally it is similar to filemodify 100644 on `<commit-ish>` path (maybe split into subdirectories). It's not advised to use any other commands to write to the `<notes_ref>` tree except `filedeleteall` to delete all existing notes in this tree. This command has two different means of specifying the content of the note.

External data format

The data content for the note was already supplied by a prior `blob` command. The frontend just needs to connect it to the commit that is to be annotated.

```
	'N' SP <dataref> SP <commit-ish> LF
```

Here `<dataref>` can be either a mark reference (`:<idnum>`) set by a prior `blob` command, or a full 40-byte SHA-1 of an existing Git blob object.

Inline data format

The data content for the note has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
	'N' SP 'inline' SP <commit-ish> LF
	data
```

See below for a detailed description of the `data` command.

In both formats `<commit-ish>` is any of the commit specification expressions also accepted by `from` (see above).

`mark`

Arranges for fast-import to save a reference to the current object, allowing the frontend to recall this object at a future point in time, without knowing its SHA-1. Here the current object is the object creation command the `mark` command appears within. This can be `commit`, `tag`, and `blob`, but `commit` is the most common usage.

```
	'mark' SP ':' <idnum> LF
```

where `<idnum>` is the number assigned by the frontend to this mark. The value of `<idnum>` is expressed as an ASCII decimal integer. The value 0 is reserved and cannot be used as a mark. Only values greater than or equal to 1 may be used as marks.

New marks are created automatically. Existing marks can be moved to another object simply by reusing the same `<idnum>` in another `mark` command.

`original-oid`

Provides the name of the object in the original source control system. fast-import will simply ignore this directive, but filter processes which operate on and modify the stream before feeding to fast-import may have uses for this information

```
	'original-oid' SP <object-identifier> LF
```

where `<object-identifier>` is any string not containing LF.

`tag`

Creates an annotated tag referring to a specific commit. To create lightweight (non-annotated) tags see the `reset` command below.

```
	'tag' SP <name> LF
	mark?
	'from' SP <commit-ish> LF
	original-oid?
	'tagger' (SP <name>)? SP LT <email> GT SP <when> LF
	data
```

where `<name>` is the name of the tag to create.

Tag names are automatically prefixed with `refs/tags/` when stored in Git, so importing the CVS branch symbol `RELENG-1_0-FINAL` would use just `RELENG-1_0-FINAL` for `<name>`, and fast-import will write the corresponding ref as `refs/tags/RELENG-1_0-FINAL`.

The value of `<name>` must be a valid refname in Git and therefore may contain forward slashes. As `LF` is not valid in a Git refname, no quoting or escaping syntax is supported here.

The `from` command is the same as in the `commit` command; see above for details.

The `tagger` command uses the same format as `committer` within `commit`; again see above for details.

The `data` command following `tagger` must supply the annotated tag message (see below for `data` command syntax). To import an empty tag message use a 0 length data. Tag messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Signing annotated tags during import from within fast-import is not supported. Trying to include your own PGP/GPG signature is not recommended, as the frontend does not (easily) have access to the complete set of bytes which normally goes into such a signature. If signing is required, create lightweight tags from within fast-import with `reset`, then create the annotated versions of those tags offline with the standard *git tag* process.

`reset`

Creates (or recreates) the named branch, optionally starting from a specific revision. The reset command allows a frontend to issue a new `from`command for an existing branch, or to create a new branch from an existing commit without creating a new commit.

```
        'reset' SP <ref> LF
        ('from' SP <commit-ish> LF)?
        LF?
```

For a detailed description of `<ref>` and `<commit-ish>` see above under `commit` and `from`.

The `LF` after the command is optional (it used to be required).

The `reset` command can also be used to create lightweight (non-annotated) tags. For example:

```
reset refs/tags/938
from :938
```

would create the lightweight tag `refs/tags/938` referring to whatever commit mark `:938` references.

`blob`

Requests writing one file revision to the packfile. The revision is not connected to any commit; this connection must be formed in a subsequent `commit` command by referencing the blob through an assigned mark.

```
        'blob' LF
        mark?
        original-oid?
        data
```

The mark command is optional here as some frontends have chosen to generate the Git SHA-1 for the blob on their own, and feed that directly to `commit`. This is typically more work than it’s worth however, as marks are inexpensive to store and easy to use.

`data`

Supplies raw data (for use as blob/file content, commit messages, or annotated tag messages) to fast-import. Data can be supplied using an exact byte count or delimited with a terminating line. Real frontends intended for production-quality conversions should always use the exact byte count format, as it is more robust and performs better. The delimited format is intended primarily for testing fast-import.

Comment lines appearing within the `<raw>` part of `data` commands are always taken to be part of the body of the data and are therefore never ignored by fast-import. This makes it safe to import any file/message content whose lines might start with `#`.

Exact byte count format

The frontend must specify the number of bytes of data.

```
        'data' SP <count> LF
        <raw> LF?
```

where `<count>` is the exact number of bytes appearing within `<raw>`. The value of `<count>` is expressed as an ASCII decimal integer. The `LF` on either side of `<raw>` is not included in `<count>` and will not be included in the imported data.

The `LF` after `<raw>` is optional (it used to be required) but recommended. Always including it makes debugging a fast-import stream easier as the next command always starts in column 0 of the next line, even if `<raw>` did not end with an `LF`.

Delimited format

A delimiter string is used to mark the end of the data. fast-import will compute the length by searching for the delimiter. This format is primarily useful for testing and is not recommended for real data.

```
        'data' SP '<<' <delim> LF
        <raw> LF
        <delim> LF
        LF?
```

where `<delim>` is the chosen delimiter string. The string `<delim>` must not appear on a line by itself within `<raw>`, as otherwise fast-import will think the data ends earlier than it really does. The `LF` immediately trailing `<raw>` is part of `<raw>`. This is one of the limitations of the delimited format, it is impossible to supply a data chunk which does not have an LF as its last byte.

The `LF` after `<delim> LF` is optional (it used to be required).

`alias`

Record that a mark refers to a given object without first creating any new object.

```
	'alias' LF
	mark
	'to' SP <commit-ish> LF
	LF?
```

For a detailed description of `<commit-ish>` see above under `from`.

`checkpoint`

Forces fast-import to close the current packfile, start a new one, and to save out all current branch refs, tags and marks.

```
	'checkpoint' LF
	LF?
```

Note that fast-import automatically switches packfiles when the current packfile reaches --max-pack-size, or 4 GiB, whichever limit is smaller. During an automatic packfile switch fast-import does not update the branch refs, tags or marks.

As a `checkpoint` can require a significant amount of CPU time and disk IO (to compute the overall pack SHA-1 checksum, generate the corresponding index file, and update the refs) it can easily take several minutes for a single `checkpoint` command to complete.

Frontends may choose to issue checkpoints during extremely large and long running imports, or when they need to allow another Git process access to a branch. However given that a 30 GiB Subversion repository can be loaded into Git through fast-import in about 3 hours, explicit checkpointing may not be necessary.

The `LF` after the command is optional (it used to be required).

`progress`

Causes fast-import to print the entire `progress` line unmodified to its standard output channel (file descriptor 1) when the command is processed from the input stream. The command otherwise has no impact on the current import, or on any of fast-import's internal state.

```
	'progress' SP <any> LF
	LF?
```

The `<any>` part of the command may contain any sequence of bytes that does not contain `LF`. The `LF` after the command is optional. Callers may wish to process the output through a tool such as sed to remove the leading part of the line, for example:

```
frontend | git fast-import | sed 's/^progress //'
```

Placing a `progress` command immediately after a `checkpoint` will inform the reader when the `checkpoint` has been completed and it can safely access the refs that fast-import updated.

`get-mark`

Causes fast-import to print the SHA-1 corresponding to a mark to stdout or to the file descriptor previously arranged with the `--cat-blob-fd` argument. The command otherwise has no impact on the current import; its purpose is to retrieve SHA-1s that later commits might want to refer to in their commit messages.

```
	'get-mark' SP ':' <idnum> LF
```

See "Responses To Commands" below for details about how to read this output safely.

`cat-blob`

Causes fast-import to print a blob to a file descriptor previously arranged with the `--cat-blob-fd` argument. The command otherwise has no impact on the current import; its main purpose is to retrieve blobs that may be in fast-import's memory but not accessible from the target repository.

```
        'cat-blob' SP <dataref> LF
```

The `<dataref>` can be either a mark reference (`:<idnum>`) set previously or a full 40-byte SHA-1 of a Git blob, preexisting or ready to be written.

Output uses the same format as `git cat-file --batch`:

```
<sha1> SP 'blob' SP <size> LF
<contents> LF
```

This command can be used where a `filemodify` directive can appear, allowing it to be used in the middle of a commit. For a `filemodify` using an inline directive, it can also appear right before the `data` directive.

See "Responses To Commands" below for details about how to read this output safely.

`ls`

Prints information about the object at a path to a file descriptor previously arranged with the `--cat-blob-fd` argument. This allows printing a blob from the active commit (with `cat-blob`) or copying a blob or tree from a previous commit for use in the current one (with `filemodify`).

The `ls` command can also be used where a `filemodify` directive can appear, allowing it to be used in the middle of a commit.

Reading from the active commit

This form can only be used in the middle of a `commit`. The path names a directory entry within fast-import's active commit. The path must be quoted in this case.

```
	'ls' SP <path> LF
```

Reading from a named tree

The `<dataref>` can be a mark reference (`:<idnum>`) or the full 40-byte SHA-1 of a Git tag, commit, or tree object, preexisting or waiting to be written. The path is relative to the top level of the tree named by `<dataref>`.

```
	'ls' SP <dataref> SP <path> LF
```

See `filemodify` above for a detailed description of `<path>`.

Output uses the same format as `git ls-tree <tree> -- <path>`:

```
<mode> SP ('blob' | 'tree' | 'commit') SP <dataref> HT <path> LF
```

The <dataref> represents the blob, tree, or commit object at <path> and can be used in later *get-mark*, *cat-blob*, *filemodify*, or *ls* commands.

If there is no file or subtree at that path, *git fast-import* will instead report

```
missing SP <path> LF
```

See "Responses To Commands" below for details about how to read this output safely.

`feature`

Require that fast-import supports the specified feature, or abort if it does not.

```
        'feature' SP <feature> ('=' <argument>)? LF
```

The <feature> part of the command may be any one of the following:

date-format

export-marks

relative-marks

no-relative-marks

force

Act as though the corresponding command-line option with a leading `--` was passed on the command line (see OPTIONS, above).

import-marks

import-marks-if-exists

Like --import-marks except in two respects: first, only one "feature import-marks" or "feature import-marks-if-exists" command is allowed per stream; second, an --import-marks= or --import-marks-if-exists command-line option overrides any of these "feature" commands in the stream; third, "feature import-marks-if-exists" like a corresponding command-line option silently skips a nonexistent file.

get-mark

cat-blob

ls

Require that the backend support the *get-mark*, *cat-blob*, or *ls* command respectively. Versions of fast-import not supporting the specified command will exit with a message indicating so. This lets the import error out early with a clear message, rather than wasting time on the early part of an import before the unsupported command is detected.

notes

Require that the backend support the *notemodify* (N) subcommand to the *commit* command. Versions of fast-import not supporting notes will exit with a message indicating so.

done

Error out if the stream ends without a *done* command. Without this feature, errors causing the frontend to end abruptly at a convenient point in the stream can go undetected. This may occur, for example, if an import front end dies in mid-operation without emitting SIGTERM or SIGKILL at its subordinate git fast-import instance.

`option`

Processes the specified option so that git fast-import behaves in a way that suits the frontend's needs. Note that options specified by the frontend are overridden by any options the user may specify to git fast-import itself.

```
    'option' SP <option> LF
```

The `<option>` part of the command may contain any of the options listed in the OPTIONS section that do not change import semantics, without the leading `--` and is treated in the same way.

Option commands must be the first commands on the input (not counting feature commands), to give an option command after any non-option command is an error.

The following command-line options change import semantics and may therefore not be passed as option:

- date-format
- import-marks
- export-marks
- cat-blob-fd
- force

`done`

If the `done` feature is not in use, treated as if EOF was read. This can be used to tell fast-import to finish early.

If the `--done` command-line option or `feature done` command is in use, the `done` command is mandatory and marks the end of the stream.

RESPONSES TO COMMANDS

New objects written by fast-import are not available immediately. Most fast-import commands have no visible effect until the next checkpoint (or completion). The frontend can send commands to fill fast-import's input pipe without worrying about how quickly they will take effect, which improves performance by simplifying scheduling.

For some frontends, though, it is useful to be able to read back data from the current repository as it is being updated (for example when the source material describes objects in terms of patches to be applied to previously imported objects). This can be accomplished by connecting the frontend and fast-import via bidirectional pipes:

```
mkfifo fast-import-output
frontend <fast-import-output |
git fast-import >fast-import-output
```

A frontend set up this way can use `progress`, `get-mark`, `ls`, and `cat-blob` commands to read information from the import in progress.

To avoid deadlock, such frontends must completely consume any pending output from `progress`, `ls`, `get-mark`, and `cat-blob` before performing writes to fast-import that might block.

CRASH REPORTS

If fast-import is supplied invalid input it will terminate with a non-zero exit status and create a crash report in the top level of the Git repository it was importing into. Crash reports contain a snapshot of the internal fast-import state as well as the most recent commands that lead up to the crash.

All recent commands (including stream comments, file changes and progress commands) are shown in the command history within the crash report, but raw file data and commit messages are excluded from the crash report. This exclusion saves space within the report file and reduces the amount of buffering that fast-import must perform during execution.

After writing a crash report fast-import will close the current packfile and export the marks table. This allows the frontend developer to inspect the repository state and resume the import from the point where it crashed. The modified branches and tags are not updated during a crash, as the import did not complete successfully. Branch and tag information can be found in the crash report and must be applied manually if the update is needed.

An example crash:

```
$ cat >in <<END_OF_INPUT
# my very first test commit
commit refs/heads/master
committer Shawn O. Pearce <spearce> 19283 -0400
# who is that guy anyway?
data <<EOF
this is my commit
EOF
M 644 inline .gitignore
data <<EOF
.gitignore
EOF
M 777 inline bob
END_OF_INPUT
```

```
$ git fast-import <in
fatal: Corrupt mode: M 777 inline bob
fast-import: dumping crash report to .git/fast_import_crash_8434
```

```
$ cat .git/fast_import_crash_8434
fast-import crash report:
    fast-import process: 8434
    parent process     : 1391
    at Sat Sep 1 00:58:12 2007
```

```
fatal: Corrupt mode: M 777 inline bob
```

```
Most Recent Commands Before Crash
---------------------------------
  # my very first test commit
  commit refs/heads/master
  committer Shawn O. Pearce <spearce> 19283 -0400
  # who is that guy anyway?
  data <<EOF
  M 644 inline .gitignore
  data <<EOF
* M 777 inline bob
```

```
Active Branch LRU
-----------------
    active_branches = 1 cur, 5 max
```

```
pos  clock name
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 1)      0 refs/heads/master
```

```
Inactive Branches
-----------------
refs/heads/master:
  status      : active loaded dirty
  tip commit  : 0000000000000000000000000000000000000000
  old tree    : 0000000000000000000000000000000000000000
  cur tree    : 0000000000000000000000000000000000000000
  commit clock: 0
  last pack   :
```

```
-------------------
END OF CRASH REPORT
```

TIPS AND TRICKS

The following tips and tricks have been collected from various users of fast-import, and are offered here as suggestions.

Use One Mark Per Commit

When doing a repository conversion, use a unique mark per commit (`mark :<n>`) and supply the --export-marks option on the command line. fast-import will dump a file which lists every mark and the Git object SHA-1 that corresponds to it. If the frontend can tie the marks back to the source repository, it is easy to verify the accuracy and completeness of the import by comparing each Git commit to the corresponding source revision.

Coming from a system such as Perforce or Subversion this should be quite simple, as the fast-import mark can also be the Perforce changeset number or the Subversion revision number.

Freely Skip Around Branches

Don’t bother trying to optimize the frontend to stick to one branch at a time during an import. Although doing so might be slightly faster for fast-import, it tends to increase the complexity of the frontend code considerably.

The branch LRU builtin to fast-import tends to behave very well, and the cost of activating an inactive branch is so low that bouncing around between branches has virtually no impact on import performance.

Handling Renames

When importing a renamed file or directory, simply delete the old name(s) and modify the new name(s) during the corresponding commit. Git performs rename detection after-the-fact, rather than explicitly during a commit.

Use Tag Fixup Branches

Some other SCM systems let the user create a tag from multiple files which are not from the same commit/changeset. Or to create tags which are a subset of the files available in the repository.

Importing these tags as-is in Git is impossible without making at least one commit which “fixes up” the files to match the content of the tag. Use fast-import’s `reset` command to reset a dummy branch outside of your normal branch space to the base commit for the tag, then commit one or more file fixup commits, and finally tag the dummy branch.

For example since all normal branches are stored under `refs/heads/` name the tag fixup branch `TAG_FIXUP`. This way it is impossible for the fixup branch used by the importer to have namespace conflicts with real branches imported from the source (the name `TAG_FIXUP` is not `refs/heads/TAG_FIXUP`).

When committing fixups, consider using `merge` to connect the commit(s) which are supplying file revisions to the fixup branch. Doing so will allow tools such as *git blame* to track through the real commit history and properly annotate the source files.

After fast-import terminates the frontend will need to do `rm .git/TAG_FIXUP` to remove the dummy branch.

Import Now, Repack Later

As soon as fast-import completes the Git repository is completely valid and ready for use. Typically this takes only a very short time, even for considerably large projects (100,000+ commits).

However repacking the repository is necessary to improve data locality and access performance. It can also take hours on extremely large projects (especially if -f and a large --window parameter is used). Since repacking is safe to run alongside readers and writers, run the repack in the background and let it finish when it finishes. There is no reason to wait to explore your new Git project!

If you choose to wait for the repack, don’t try to run benchmarks or performance tests until repacking is completed. fast-import outputs suboptimal packfiles that are simply never seen in real use situations.

Repacking Historical Data

If you are repacking very old imported data (e.g. older than the last year), consider expending some extra CPU time and supplying --window=50 (or higher) when you run *git repack*. This will take longer, but will also produce a smaller packfile. You only need to expend the effort once, and everyone using your project will benefit from the smaller repository.

Include Some Progress Messages

Every once in a while have your frontend emit a `progress` message to fast-import. The contents of the messages are entirely free-form, so one suggestion would be to output the current month and year each time the current commit date moves into the next month. Your users will feel better knowing how much of the data stream has been processed.

PACKFILE OPTIMIZATION

When packing a blob fast-import always attempts to deltify against the last blob written. Unless specifically arranged for by the frontend, this will probably not be a prior version of the same file, so the generated delta will not be the smallest possible. The resulting packfile will be compressed, but will not be optimal.

Frontends which have efficient access to all revisions of a single file (for example reading an RCS/CVS ,v file) can choose to supply all revisions of that file as a sequence of consecutive `blob` commands. This allows fast-import to deltify the different file revisions against each other, saving space in the final packfile. Marks can be used to later identify individual file revisions during a sequence of `commit` commands.

The packfile(s) created by fast-import do not encourage good disk access patterns. This is caused by fast-import writing the data in the order it is received on standard input, while Git typically organizes data within packfiles to make the most recent (current tip) data appear before historical data. Git also clusters commits together, speeding up revision traversal through better cache locality.

For this reason it is strongly recommended that users repack the repository with `git repack -a -d` after fast-import completes, allowing Git to reorganize the packfiles for faster data access. If blob deltas are suboptimal (see above) then also adding the `-f` option to force recomputation of all deltas can significantly reduce the final packfile size (30-50% smaller can be quite typical).

Instead of running `git repack` you can also run `git gc --aggressive`, which will also optimize other things after an import (e.g. pack loose refs). As noted in the "AGGRESSIVE" section in git-gc(1) the `--aggressive` option will find new deltas with the `-f` option to git-repack(1). For the reasons elaborated on above using `--aggressive` after a fast-import is one of the few cases where it's known to be worthwhile.

MEMORY UTILIZATION

There are a number of factors which affect how much memory fast-import requires to perform an import. Like critical sections of core Git, fast-import uses its own memory allocators to amortize any overheads associated with malloc. In practice fast-import tends to amortize any malloc overheads to 0, due to its use of large block allocations.

per object

fast-import maintains an in-memory structure for every object written in this execution. On a 32 bit system the structure is 32 bytes, on a 64 bit system the structure is 40 bytes (due to the larger pointer sizes). Objects in the table are not deallocated until fast-import terminates. Importing 2 million objects on a 32 bit system will require approximately 64 MiB of memory.

The object table is actually a hashtable keyed on the object name (the unique SHA-1). This storage configuration allows fast-import to reuse an existing or already written object and avoid writing duplicates to the output packfile. Duplicate blobs are surprisingly common in an import, typically due to branch merges in the source.

per mark

Marks are stored in a sparse array, using 1 pointer (4 bytes or 8 bytes, depending on pointer size) per mark. Although the array is sparse, frontends are still strongly encouraged to use marks between 1 and n, where n is the total number of marks required for this import.

per branch

Branches are classified as active and inactive. The memory usage of the two classes is significantly different.

Inactive branches are stored in a structure which uses 96 or 120 bytes (32 bit or 64 bit systems, respectively), plus the length of the branch name (typically under 200 bytes), per branch. fast-import will easily handle as many as 10,000 inactive branches in under 2 MiB of memory.

Active branches have the same overhead as inactive branches, but also contain copies of every tree that has been recently modified on that branch. If subtree `include` has not been modified since the branch became active, its contents will not be loaded into memory, but if subtree `src` has been modified by a commit since the branch became active, then its contents will be loaded in memory.

As active branches store metadata about the files contained on that branch, their in-memory storage size can grow to a considerable size (see below).

fast-import automatically moves active branches to inactive status based on a simple least-recently-used algorithm. The LRU chain is updated on each `commit` command. The maximum number of active branches can be increased or decreased on the command line with --active-branches=.

per active tree

Trees (aka directories) use just 12 bytes of memory on top of the memory required for their entries (see “per active file” below). The cost of a tree is virtually 0, as its overhead amortizes out over the individual file entries.

per active file entry

Files (and pointers to subtrees) within active trees require 52 or 64 bytes (32/64 bit platforms) per entry. To conserve space, file and tree names are pooled in a common string table, allowing the filename “Makefile” to use just 16 bytes (after including the string header overhead) no matter how many times it occurs within the project.

The active branch LRU, when coupled with the filename string pool and lazy loading of subtrees, allows fast-import to efficiently import projects with 2,000+ branches and 45,114+ files in a very limited memory footprint (less than 2.7 MiB per active branch).

SIGNALS

Sending **SIGUSR1** to the *git fast-import* process ends the current packfile early, simulating a `checkpoint` command. The impatient operator can use this facility to peek at the objects and refs from an import in progress, at the cost of some added running time and worse compression.