NAME

gitformat-pack - Git pack format

SYNOPSIS

$GIT_DIR/objects/pack/pack**-.{pack,idx}**
$GIT_DIR/objects/pack/pack-.rev
$GIT_DIR/objects/pack/pack-*.mtimes
$GIT_DIR/objects/pack/multi-pack-index

DESCRIPTION

The Git pack format is now Git stores most of its primary repository data. Over the lietime af a repository loose objects (if any) and smaller packs are consolidated into larger pack(s). See **git-gc**(1) and **git-pack-objects**(1).

The pack format is also used over-the-wire, see e.g. **gitprotocol-v2**(5), as well as being a part of other container formats in the case of **gitformat-bundle**(5).

CHECKSUMS AND OBJECT IDS

In a repository using the traditional SHA-1, pack checksums, index checksums, and object IDs (object names) mentioned below are all computed using SHA-1. Similarly, in SHA-256 repositories, these values are computed using SHA-256.

PACK-*.PACK FILES HAVE THE FOLLOWING FORMAT:

•

header appears at the beginning and consists of the following:

```
4-byte signature:
   The signature is: {’P’, ’A’, ’C’, ’K’}

4-byte version number (network byte order):
   Git currently accepts version number 2 or 3 but
   generates version 2 only.

4-byte number of objects contained in the pack (network byte order)

Observation: we cannot have more than 4G versions ;-) and
more than 4G objects in a pack.
```

•

header is followed by number of object entries, each of which looks like this:

```
(undeltified representation)
n-byte type and length (3-bit type, (n-1)*7+4-bit length)
compressed data

(deltified representation)
n-byte type and length (3-bit type, (n-1)*7+4-bit length)
base object name if OBJ_REF_DELTA or a negative relative
    offset from the delta object's position in the pack if this
    is an OBJ_OFS_DELTA object
compressed delta data

Observation: length of each object is encoded in a variable
length format and is not constrained to 32-bit or anything.
```

•
trailer records a pack checksum of all of the above.

Object types

Valid object types are:

•
(1)

•
(2)

•
(3)

•
(4)

•
(6)

•
(7)

Type 5 is reserved for future expansion. Type 0 is invalid.

Size encoding

This document uses the following "size encoding" of non-negative integers: From each byte, the seven least significant bits are used to form the resulting integer. As long as the most significant bit is 1, this process continues; the byte with MSB 0 provides the last seven bits. The seven-bit chunks are concatenated. Later values are more significant.

This size encoding should not be confused with the "offset encoding", which is also used in this document.

Deltified representation

Conceptually there are only four object types: commit, tree, tag and blob. However to save space, an object could be stored as a "delta" of another "base" object. These representations are assigned new types ofs-delta and ref-delta, which is only valid in a pack file.

Both ofs-delta and ref-delta store the "delta" to be applied to another object (called *base object*) to reconstruct the object. The difference between them is, ref-delta directly encodes base object name. If the base object is in the same pack, ofs-delta encodes the offset of the base object in the pack instead.

The base object could also be deltified if it’s in the same pack. Ref-delta can also refer to an object outside the pack (i.e. the so-called "thin pack"). When stored on disk however, the pack should be self contained to avoid cyclic dependency.

The delta data starts with the size of the base object and the size of the object to be reconstructed. These sizes are encoded using the size encoding from above. The remainder of the delta data is a sequence of instructions to reconstruct the object from the base object. If the base object is deltified, it must be converted to canonical form first. Each instruction appends more and more data to the target object until it’s complete. There are two supported instructions so far: one for copy a byte range from the source object and one for inserting new data embedded in the instruction itself.

Each instruction has variable length. Instruction type is determined by the seventh bit of the first octet. The following diagrams follow the convention in RFC 1951 (Deflate compressed data format).

Instruction to copy from base object

```
+----------+---------+---------+---------+---------+-------+-------+-------+
| 1xxxxxxx | offset1 | offset2 | offset3 | offset4 | size1 | size2 | size3 |
+----------+---------+---------+---------+---------+-------+-------+-------+
```

This is the instruction format to copy a byte range from the source object. It encodes the offset to copy from and the number of bytes to copy. Offset and size are in little-endian order.

All offset and size bytes are optional. This is to reduce the instruction size when encoding small offsets or sizes. The first seven bits in the first octet determines which of the next seven octets is present. If bit zero is set, offset1 is present. If bit one is set offset2 is present and so on.

Note that a more compact instruction does not change offset and size encoding. For example, if only offset2 is omitted like below, offset3 still contains bits 16-23. It does not become offset2 and contains bits 8-15 even if it’s right next to offset1.

```
+----------+---------+---------+
| 10000101 | offset1 | offset3 |
+----------+---------+---------+
```

In its most compact form, this instruction only takes up one byte (0x80) with both offset and size omitted, which will have default values zero. There is another exception: size zero is automatically converted to 0x10000.

Instruction to add new data

```
+----------+============+
| 0xxxxxxx |    data    |
+----------+============+
```

This is the instruction to construct target object without the base object. The following data is appended to the target object. The first seven bits of the first octet determines the size of data in bytes. The size must be non-zero.

Reserved instruction

```
+----------+============
| 00000000 |
+----------+============
```

This is the instruction reserved for future expansion.

ORIGINAL (VERSION 1) PACK-*.IDX FILES HAVE THE FOLLOWING FORMAT:

•

header consists of 256 4-byte network byte order integers. N-th entry of this table records the number of objects in the corresponding pack, the first byte of whose object name is less than or equal to N. This is called the *first-level fan-out* table.

ɸ
header is followed by sorted 24-byte entries, one entry per object in the pack. Each entry is:

    4-byte network byte order integer, recording where the
    object is stored in the packfile as the offset from the
    beginning.

    one object name of the appropriate size.

ɸ
file is concluded with a trailer:

    A copy of the pack checksum at the end of the corresponding
    packfile.

    Index checksum of all of the above.

Pack Idx file:

```
        --  +--------------------------------+
fanout      | fanout[0] = 2 (for example)    |-.
table       +--------------------------------+ |
            | fanout[1]                      | |
            +--------------------------------+ |
            | fanout[2]                      | |
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ |
            | fanout[255] = total objects    |---.
        --  +--------------------------------+ | |
main        | offset                         | | |
index       | object name 00XXXXXXXXXXXXXXXX | | |
table       +--------------------------------+ | |
            | offset                         | | |
            | object name 00XXXXXXXXXXXXXXXX | | |
            +--------------------------------+<+ |
          .-| offset                         |   |
          | | object name 01XXXXXXXXXXXXXXXX |   |
          | +--------------------------------+   |
```

```
                        | | offset                    |  |
                        | | object name 01XXXXXXXXXXXXXXXX |  |
                        | ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  |
                        | | offset                    |  |
                        | | object name FFXXXXXXXXXXXXXXXX |  |
                      --| +-------------------------------+<--+
              trailer   | | packfile checksum           |
                        | +-------------------------------+
                        | | idxfile checksum            |
                        | +-------------------------------+
                        .-------.
                                |
              Pack file entry: <+

   packed object header:
      1-byte size extension bit (MSB)
             type (next 3 bit)
             size0 (lower 4-bit)
      n-byte sizeN (as long as MSB is set, each 7-bit)
              size0..sizeN form 4+7+7+..+7 bit integer, size0
              is the least significant part, and sizeN is the
              most significant part.
   packed object data:
      If it is not DELTA, then deflated bytes (the size above
              is the size before compression).
      If it is REF_DELTA, then
        base object name (the size above is the
              size of the delta data that follows).
        delta data, deflated.
      If it is OFS_DELTA, then
        n-byte offset (see below) interpreted as a negative
              offset from the type-byte of the header of the
              ofs-delta entry (the size above is the size of
              the delta data that follows).
        delta data, deflated.

   offset encoding:
        n bytes with MSB set in all but the last one.
        The offset is then the number constructed by
        concatenating the lower 7 bit of each byte, and
```

for n >= 2 adding 2^7 + 2^14 + ... + 2^(7*(n-1))
to the result.

VERSION 2 PACK-*.IDX FILES SUPPORT PACKS LARGER THAN 4 GIB, AND

have some other reorganizations. They have the format:

- 4-byte magic number *\377tOc* which is an unreasonable fanout[0] value.

- 4-byte version number (= 2)

- 256-entry fan-out table just like v1.

- table of sorted object names. These are packed together without offset values to reduce the cache footprint of the binary search for a specific object name.

- table of 4-byte CRC32 values of the packed object data. This is new in v2 so compressed data can be copied directly from pack to pack during repacking without undetected data corruption.

- table of 4-byte offset values (in network byte order). These are usually 31-bit pack file offsets, but large offsets are encoded as an index into the next table with the msbit set.

- table of 8-byte offset entries (empty for pack files less than 2 GiB). Pack files are organized with heavily used objects toward the front, so most object references should not need to refer to this table.

- same trailer as a v1 pack file:

      A copy of the pack checksum at the end of
      corresponding packfile.

      Index checksum of all of the above.

PACK-*.REV FILES HAVE THE FORMAT:

- 4-byte magic number *0x52494458* (*RIDX*).

- 4-byte version identifier (= 1).

- 4-byte hash function identifier (= 1 for SHA-1, 2 for SHA-256).

- table of index positions (one per packed object, num_objects in total, each a 4-byte unsigned integer in network order), sorted by their corresponding offsets in the packfile.

- trailer, containing a:

  checksum of the corresponding packfile, and

  a checksum of all of the above.

All 4-byte numbers are in network order.

PACK-*.MTIMES FILES HAVE THE FORMAT:

All 4-byte numbers are in network byte order.

- 4-byte magic number *0x4d544d45* (*MTME*).

- 4-byte version identifier (= 1).

- 4-byte hash function identifier (= 1 for SHA-1, 2 for SHA-256).

- table of 4-byte unsigned integers. The ith value is the modification time (mtime) of the ith object in the corresponding pack by lexicographic (index) order. The mtimes count standard epoch seconds.

- trailer, containing a checksum of the corresponding packfile, and a checksum of all of the above (each

having length according to
the specified hash function).

MULTI-PACK-INDEX (MIDX) FILES HAVE THE FOLLOWING FORMAT:

The multi-pack-index files refer to multiple pack-files and loose objects.

In order to allow extensions that add extra data to the MIDX, we organize the body into "chunks" and provide a lookup table at the beginning of the body. The header includes certain length values, such as the number of packs, the number of base MIDX files, hash lengths and types.

All 4-byte numbers are in network order.

HEADER:

```
4-byte signature:
    The signature is: {’M’, ’I’, ’D’, ’X’}

1-byte version number:
    Git only writes or recognizes version 1.

1-byte Object Id Version
    We infer the length of object IDs (OIDs) from this value:
        1 => SHA-1
        2 => SHA-256
    If the hash type does not match the repository’s hash algorithm,
    the multi-pack-index file should be ignored with a warning
    presented to the user.

1-byte number of "chunks"

1-byte number of base multi-pack-index files:
    This value is currently always zero.

4-byte number of pack files
```

CHUNK LOOKUP:

```
(C + 1) * 12 bytes providing the chunk offsets:
    First 4 bytes describe chunk id. Value 0 is a terminating label.
    Other 8 bytes provide offset in current file for chunk to start.
```

```
            (Chunks are provided in file-order, so you can infer the length
            using the next chunk position if necessary.)

        The CHUNK LOOKUP matches the table of contents from
        the chunk-based file format, see linkgit:gitformat-chunk[5].

        The remaining data in the body is described one chunk at a time, and
        these chunks may be given in any order. Chunks are required unless
        otherwise specified.

    CHUNK DATA:

        Packfile Names (ID: {'P', 'N', 'A', 'M'})
            Stores the packfile names as concatenated, null-terminated strings.
            Packfiles must be listed in lexicographic order for fast lookups by
            name. This is the only chunk not guaranteed to be a multiple of four
            bytes in length, so should be the last chunk for alignment reasons.

        OID Fanout (ID: {'O', 'I', 'D', 'F'})
            The ith entry, F[i], stores the number of OIDs with first
            byte at most i. Thus F[255] stores the total
            number of objects.

        OID Lookup (ID: {'O', 'I', 'D', 'L'})
            The OIDs for all objects in the MIDX are stored in lexicographic
            order in this chunk.

        Object Offsets (ID: {'O', 'O', 'F', 'F'})
            Stores two 4-byte values for every object.
            1: The pack-int-id for the pack storing this object.
            2: The offset within the pack.
                If all offsets are less than 2^32, then the large offset chunk
                will not exist and offsets are stored as in IDX v1.
                If there is at least one offset value larger than 2^32-1, then
                the large offset chunk must exist, and offsets larger than
                2^31-1 must be stored in it instead. If the large offset chunk
                exists and the 31st bit is on, then removing that bit reveals
                the row in the large offsets containing the 8-byte offset of
                this object.
```

[Optional] Object Large Offsets (ID: {’L’, ’O’, ’F’, ’F’})
8-byte offsets into large packfiles.

[Optional] Bitmap pack order (ID: {’R’, ’I’, ’D’, ’X’})
A list of MIDX positions (one per object in the MIDX, num_objects in total, each a 4-byte unsigned integer in network byte order), sorted according to their relative bitmap/pseudo-pack positions.

TRAILER:

Index checksum of the above contents.

MULTI-PACK-INDEX REVERSE INDEXES

Similar to the pack-based reverse index, the multi-pack index can also be used to generate a reverse index.

Instead of mapping between offset, pack-, and index position, this reverse index maps between an object’s position within the MIDX, and that object’s position within a pseudo-pack that the MIDX describes (i.e., the ith entry of the multi-pack reverse index holds the MIDX position of ith object in pseudo-pack order).

To clarify the difference between these orderings, consider a multi-pack reachability bitmap (which does not yet exist, but is what we are building towards here). Each bit needs to correspond to an object in the MIDX, and so we need an efficient mapping from bit position to MIDX position.

One solution is to let bits occupy the same position in the oid-sorted index stored by the MIDX. But because oids are effectively random, their resulting reachability bitmaps would have no locality, and thus compress poorly. (This is the reason that single-pack bitmaps use the pack ordering, and not the .idx ordering, for the same purpose.)

So we’d like to define an ordering for the whole MIDX based around pack ordering, which has far better locality (and thus compresses more efficiently). We can think of a pseudo-pack created by the concatenation of all of the packs in the MIDX. E.g., if we had a MIDX with three packs (a, b, c), with 10, 15, and 20 objects respectively, we can imagine an ordering of the objects like:

```
|a,0|a,1|...|a,9|b,0|b,1|...|b,14|c,0|c,1|...|c,19|
```

where the ordering of the packs is defined by the MIDX’s pack list, and then the ordering of objects within each pack is the same as the order in the actual packfile.

Given the list of packs and their counts of objects, you can naïvely reconstruct that pseudo-pack ordering (e.g., the object at position 27 must be (c,1) because packs "a" and "b" consumed 25 of the slots). But there’s a catch. Objects may be duplicated between packs, in which case the MIDX only stores one pointer to the object (and thus we’d want only one slot in the bitmap).

Callers could handle duplicates themselves by reading objects in order of their bit-position, but that’s linear in the number of objects, and much too expensive for ordinary bitmap lookups. Building a reverse index solves this, since it is the logical inverse of the index, and that index has already removed duplicates. But, building a reverse index on the fly can be expensive. Since we already have an on-disk format for pack-based reverse indexes, let’s reuse it for the MIDX’s pseudo-pack, too.

Objects from the MIDX are ordered as follows to string together the pseudo-pack. Let **pack(o)** return the pack from which **o** was selected by the MIDX, and define an ordering of packs based on their numeric ID (as stored by the MIDX). Let **offset(o)** return the object offset of **o** within **pack(o)**. Then, compare **o1** and **o2** as follows:

•
one of **pack(o1)** and **pack(o2)** is preferred and the other is not, then the preferred one sorts first.

(This is a detail that allows the MIDX bitmap to determine which pack should be used by the pack-reuse mechanism, since it can ask the MIDX for the pack containing the object at bit position 0).

•
pack(o1) != pack(o2), then sort the two objects in descending order based on the pack ID.

•
pack(o1) = pack(o2), and the objects are sorted in pack-order (i.e., **o1** sorts ahead of **o2** exactly when **offset(o1) < offset(o2)**).

In short, a MIDX’s pseudo-pack is the de-duplicated concatenation of objects in packs stored by the MIDX, laid out in pack order, and the packs arranged in MIDX order (with the preferred pack coming first).

The MIDX’s reverse index is stored in the optional *RIDX* chunk within the MIDX itself.

CRUFT PACKS

The cruft packs feature offer an alternative to Git’s traditional mechanism of removing unreachable objects. This document provides an overview of Git’s pruning mechanism, and how a cruft pack can be used instead to accomplish the same.

Background

To remove unreachable objects from your repository, Git offers **git repack -Ad** (see **git-repack**(1)). Quoting from the documentation:

> [...] unreachable objects in a previous pack become loose, unpacked objects,
> instead of being left in the old pack. [...] loose unreachable objects will be
> pruned according to normal expiry rules with the next 'git gc' invocation.

Unreachable objects aren't removed immediately, since doing so could race with an incoming push which may reference an object which is about to be deleted. Instead, those unreachable objects are stored as loose objects and stay that way until they are older than the expiration window, at which point they are removed by **git-prune**(1).

Git must store these unreachable objects loose in order to keep track of their per-object mtimes. If these unreachable objects were written into one big pack, then either freshening that pack (because an object contained within it was re-written) or creating a new pack of unreachable objects would cause the pack's mtime to get updated, and the objects within it would never leave the expiration window. Instead, objects are stored loose in order to keep track of the individual object mtimes and avoid a situation where all cruft objects are freshened at once.

This can lead to undesirable situations when a repository contains many unreachable objects which have not yet left the grace period. Having large directories in the shards of **.git/objects** can lead to decreased performance in the repository. But given enough unreachable objects, this can lead to inode starvation and degrade the performance of the whole system. Since we can never pack those objects, these repositories often take up a large amount of disk space, since we can only zlib compress them, but not store them in delta chains.

Cruft packs

A cruft pack eliminates the need for storing unreachable objects in a loose state by including the per-object mtimes in a separate file alongside a single pack containing all loose objects.

A cruft pack is written by **git repack --cruft** when generating a new pack. **git-pack-objects**(1)'s **--cruft** option. Note that **git repack --cruft** is a classic all-into-one repack, meaning that everything in the resulting pack is reachable, and everything else is unreachable. Once written, the **--cruft** option instructs **git repack** to generate another pack containing only objects not packed in the previous step (which equates to packing all unreachable objects together). This progresses as follows:

1.

every object, marking any object which is (a) not contained in a kept-pack, and (b) whose mtime is within the grace period as a traversal tip.

2. a reachability traversal based on the tips gathered in the previous step, adding every object along the way to the pack.

3. the pack out, along with a **.mtimes** file that records the per-object timestamps.

This mode is invoked internally by **git-repack**(1) when instructed to write a cruft pack. Crucially, the set of in-core kept packs is exactly the set of packs which will not be deleted by the repack; in other words, they contain all of the repository's reachable objects.

When a repository already has a cruft pack, **git repack --cruft** typically only adds objects to it. An exception to this is when **git repack** is given the **--cruft-expiration** option, which allows the generated cruft pack to omit expired objects instead of waiting for **git-gc**(1) to expire those objects later on.

It is **git-gc**(1) that is typically responsible for removing expired unreachable objects.

Caution for mixed-version environments

Repositories that have cruft packs in them will continue to work with any older version of Git. Note, however, that previous versions of Git which do not understand the **.mtimes** file will use the cruft pack's mtime as the mtime for all of the objects in it. In other words, do not expect older (pre-cruft pack) versions of Git to interpret or even read the contents of the **.mtimes** file.

Note that having mixed versions of Git GC-ing the same repository can lead to unreachable objects never being completely pruned. This can happen under the following circumstances:

- older version of Git running GC explodes the contents of an existing cruft pack loose, using the cruft pack's mtime.

- newer version running GC collects those loose objects into a cruft pack, where the .mtime file reflects the loose object's actual mtimes, but the cruft pack mtime is "now".

Repeating this process will lead to unreachable objects not getting pruned as a result of repeatedly resetting the objects' mtimes to the present time.

If you are GC-ing repositories in a mixed version environment, consider omitting the **--cruft** option when using **git-repack**(1) and **git-gc**(1), and setting the **gc.cruftPacks** configuration to "false" until all writers understand cruft packs.

Alternatives

Notable alternatives to this design include:

- location of the per-object mtime data, and

- unreachable objects in multiple cruft packs.

On the location of mtime data, a new auxiliary file tied to the pack was chosen to avoid complicating the **.idx** format. If the **.idx** format were ever to gain support for optional chunks of data, it may make sense to consolidate the **.mtimes** format into the **.idx** itself.

Storing unreachable objects among multiple cruft packs (e.g., creating a new cruft pack during each repacking operation including only unreachable objects which aren’t already stored in an earlier cruft pack) is significantly more complicated to construct, and so aren’t pursued here. The obvious drawback to the current implementation is that the entire cruft pack must be re-written from scratch.

GIT

Part of the **git**(1) suite