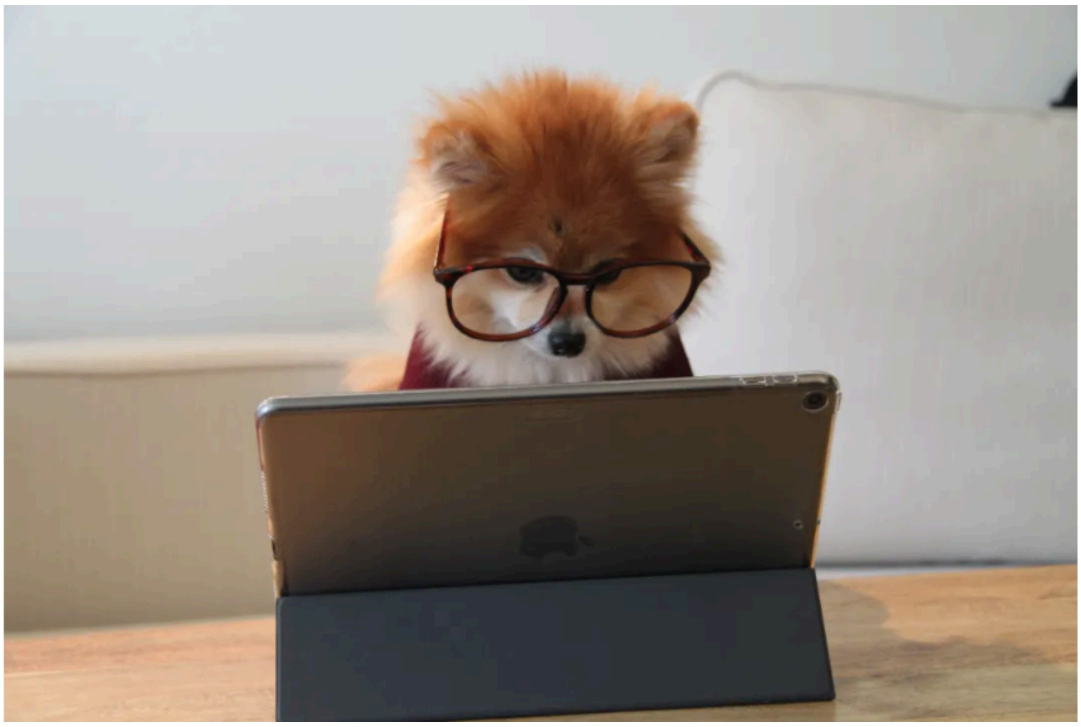
A Detailed Explanation of the Underlying Data Structures and Principles of Git

Alibaba Clouder March 2, 2021 14,884 0

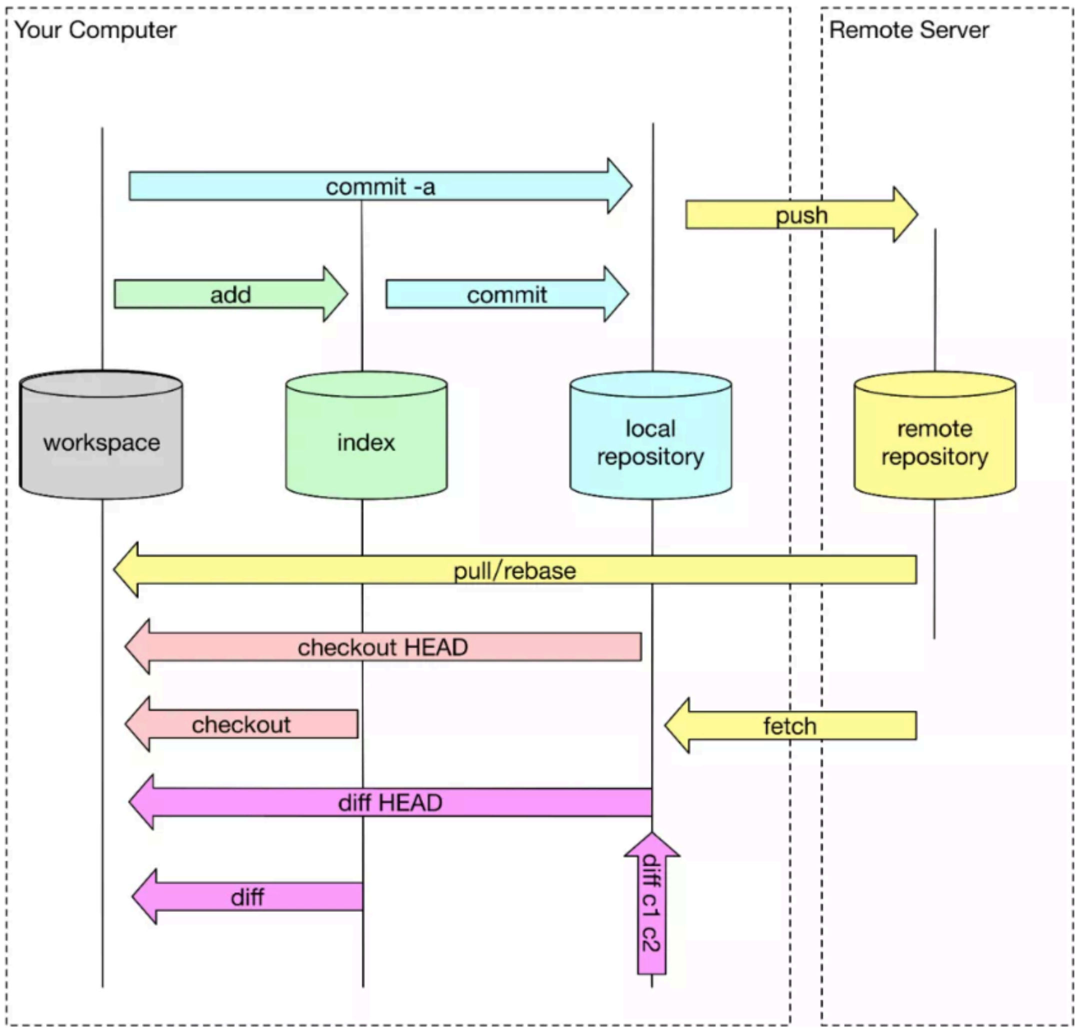
This article will systematically elaborate on the underlying knowledge of Git, and delve into the process of object query and the corresponding algorithms.

By Yushui



This article will systematically elaborate on the underlying knowledge of Git, including the changes in object lifecycle, underlying data structures, packet file structures, and packet file indexing. In addition, it will dive into the process of object query and the corresponding algorithms.

State Models



The preceding figure illustrates the different storage locations of Git objects in different lifecycle stages. You can change the lifecycle stages of Git objects using different Git commands.

Workspace

Our current workspace is the file structure that we can see in the local folder. When the workspace is initialized or cleaned, the contents of its files are consistent with those in the index. With file modifications, if the modified files in the workspace are not added to the index, the contents in the workspace become inconsistent with those in the index.

Index

In earlier versions, it is also called the cache, where the files are temporarily stored. With a commit operation, all the files that are temporarily stored in the index will be committed to the local repository together. Then, all the files in the local repository will be replaced by the files from the index. The index is a very important part of the architecture design of Git. However, it is difficult to understand.

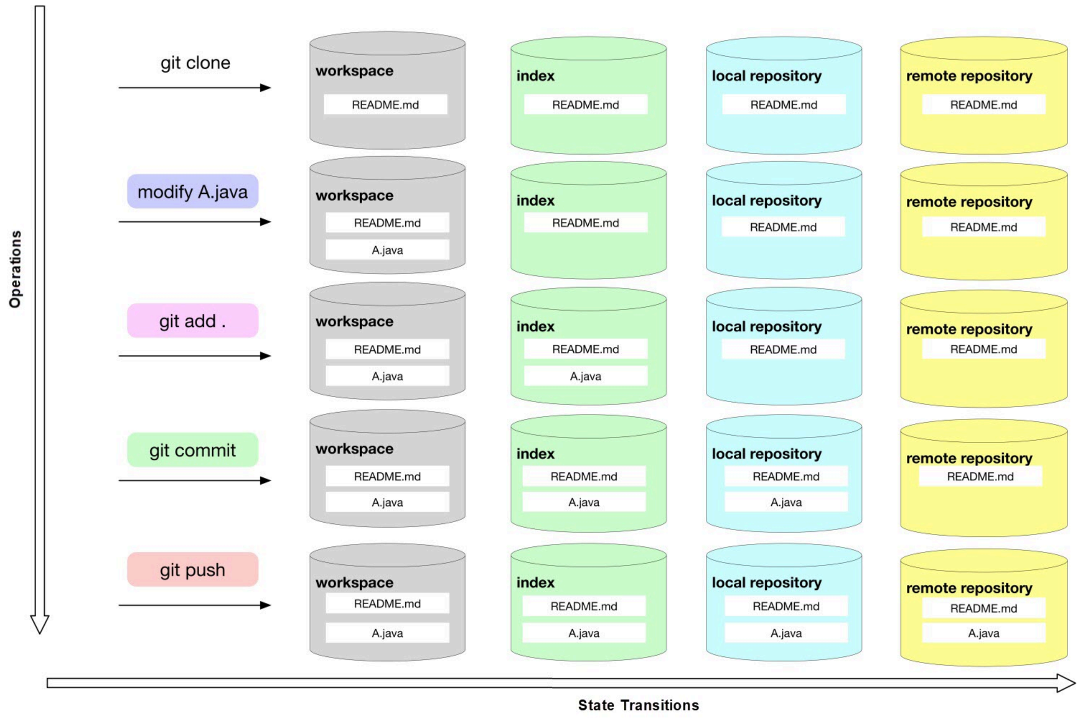
Local Repository

Git is a distributed version control system. Unlike other version control systems, it can work in a completely decentralized manner without communicating with the remote server. You can perform all offline operations locally, including log, history, commit, and diff. The core reason why you can perform these offline operations is that the local repository of Git is almost the same as the remote repository. Therefore, all offline operations can be completed locally, and the local repository will interact with the remote server when necessary.

Remote Repository

This is a centralized repository shared by everyone. If you want to update the local repository with contents from everyone else or upload your contents to share with others, the local repository will need to interact with the remote repository. The structure of the remote repository is generally the same as that of the local repository.

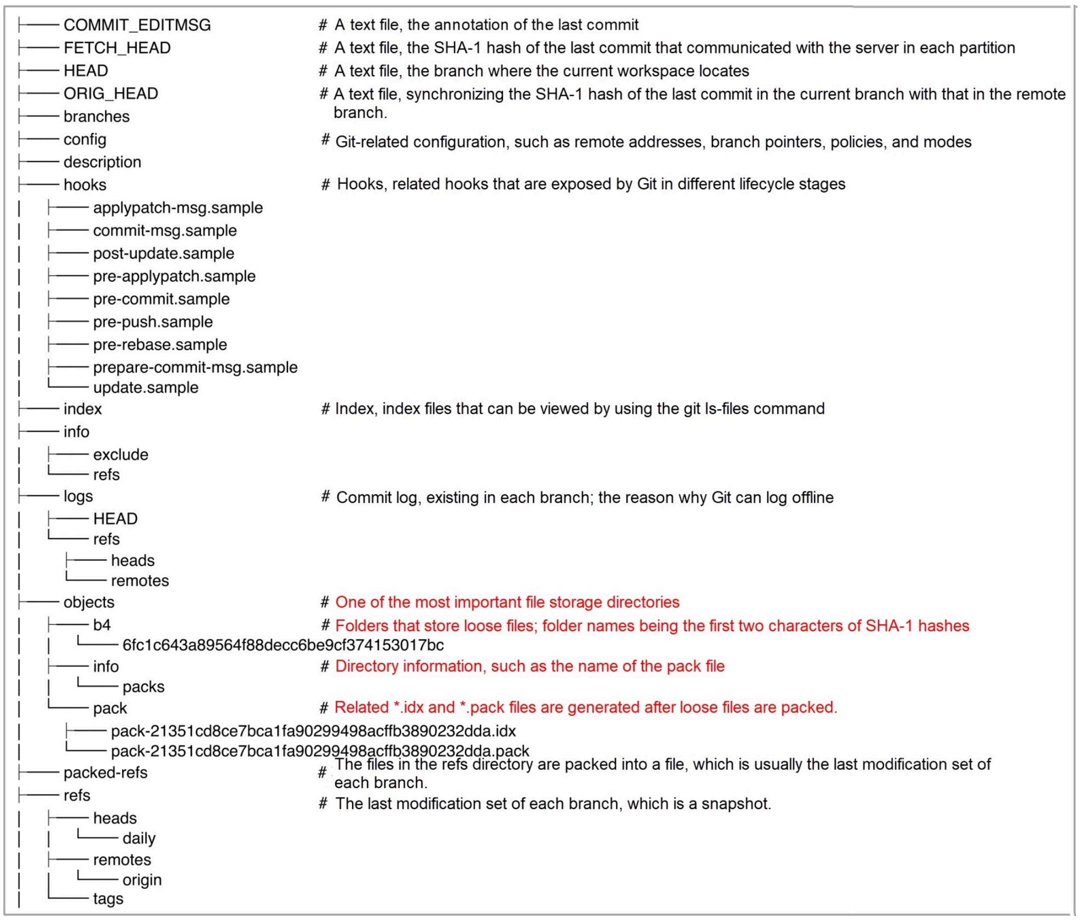
Different operations may put files in different Git lifecycle stages. The following image shows an example of file changes:



Object Models

Repository Structure

An important decentralized feature of Git is that it has a complete local repository, which is the .git file directory. With this repository, Git supports completely offline operations. All Git model objects are stored in this local repository. The tree structure of a Git repository and the corresponding descriptions are shown below:



```
├─── COMMIT_EDITMSG              # A text file, the annotation of the last commit
├─── FETCH_HEAD                  # A text file, the SHA-1 hash of the last commit that communicated with the server in each partition
├─── HEAD                        # A text file, the branch where the current workspace locates
├─── ORIG_HEAD                   # A text file, synchronizing the SHA-1 hash of the last commit in the current branch with that in the remote
├─── branches                      branch.
├─── config                      # Git-related configuration, such as remote addresses, branch pointers, policies, and modes
├─── description
├─── hooks                       # Hooks, related hooks that are exposed by Git in different lifecycle stages
│    ├─── applypatch-msg.sample
│    ├─── commit-msg.sample
│    ├─── post-update.sample
│    ├─── pre-applypatch.sample
│    ├─── pre-commit.sample
│    ├─── pre-push.sample
│    ├─── pre-rebase.sample
│    ├─── prepare-commit-msg.sample
│    └─── update.sample
├─── index                       # Index, index files that can be viewed by using the git ls-files command
├─── info
│    ├─── exclude
│    └─── refs
├─── logs                        # Commit log, existing in each branch; the reason why Git can log offline
│    ├─── HEAD
│    └─── refs
│         ├─── heads
│         └─── remotes
├─── objects                     # One of the most important file storage directories
│    ├─── b4                     # Folders that store loose files; folder names being the first two characters of SHA-1 hashes
│    │    └─── 6fc1c643a89564f88decc6be9cf374153017bc
│    ├─── info                   # Directory information, such as the name of the pack file
│    │    └─── packs
│    └─── pack                   # Related *.idx and *.pack files are generated after loose files are packed.
│         ├─── pack-21351cd8ce7bca1fa90299498acffb3890232dda.idx
│         └─── pack-21351cd8ce7bca1fa90299498acffb3890232dda.pack
├─── packed-refs                 # The files in the refs directory are packed into a file, which is usually the last modification set of
├─── refs                          each branch.
│    ├─── heads                  # The last modification set of each branch, which is a snapshot.
│    │    └─── daily
│    ├─── remotes
│    │    └─── origin
│    └─── tags
```

There are four major types of Git objects, blobs, trees, commits, and tags. The names of these objects are all SHA-1 hashes.

You can use the `git cat-file -t` command to view the type of each SHA-1. You can use the `git cat-file -p` command to view the contents and simple data structure of each object. The git cat-file command is an underlying core command of Git.

Blob Objects

They are used to store the contents of a single file. The file is typically a binary data file that does not contain any other file information, such as the file name and other metadata.

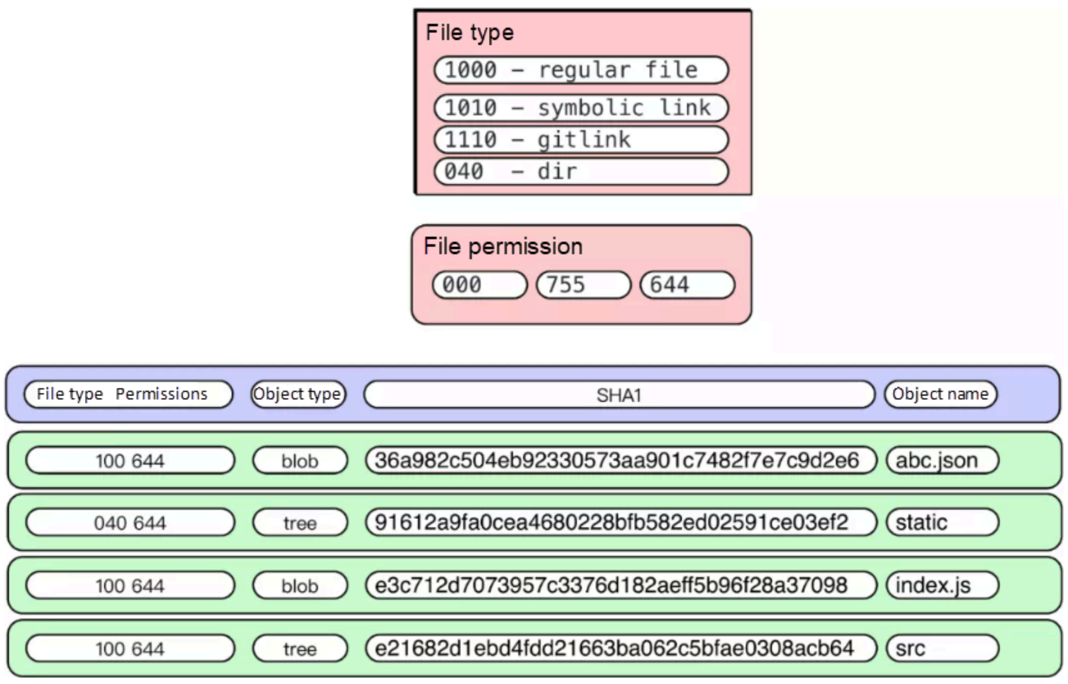
Tree Objects

They are the directory structures of the corresponding file systems. They contain subdirectories (trees), file lists (blobs), file types, and some permission models for data files.

The following figure shows the output:

```
→ git cat-file -t ed807a4d010a06ca83d448bc74c6cc79121c07c3
tree
→ git cat-file -p ed807a4d010a06ca83d448bc74c6cc79121c07c3
100644 blob 36a982c504eb92330573aa901c7482f7e7c9d2e6    .cise.yml
100644 blob c439a8da9e9cca4e7b29ee260aea008964a00e9a    .eslintignore
100644 blob 245b35b9162bec4ef798eb05b533e6c98633af5c    .eslintrc
100644 blob 10123778ec5206edcd6e8500cc78b77e79285f6d    .gitignore
100644 blob 1a48aa945106d7591b6342585b1c29998e486bf6    README.md
100644 blob 514f7cb2645f44dd9b66a87f869d42902174fe40    abc.json
040000 tree 8955f46834e3e35d74766639d740af922dcaccd3    cli_list10064
040000 tree e2b3ee59f6b030a45c0bf2770e6b0c1fa5f1d8c7    doc
100644 blob e3c712d7073957c3376d182aeff5b96f28a37098    index.js
040000 tree b4aadab8fc0228a14060321e3f89af50ba5817ca    lib040000 tre
100644 blob 95913ff73be1cc7dec869485e80072b6abdd7be4    package.json
040000 tree e21682d1ebd4fdd21663ba062c5bfae0308acb64    src
040000 tree 91612a9fa0cea4680228bfb582ed02591ce03ef2    static
040000 tree d0265f130d2c5cb023fe16c990ecd56d1a07b78c    task100644 bl
100644 blob fb8e6d3a39baf6e339e235de1a9ed7c3f1521d55    webpack.dll.c
040000 tree 5dd44553be0d7e528b8667ac3c027ddc0909ef36    webpack
```

The following gives a detailed description:



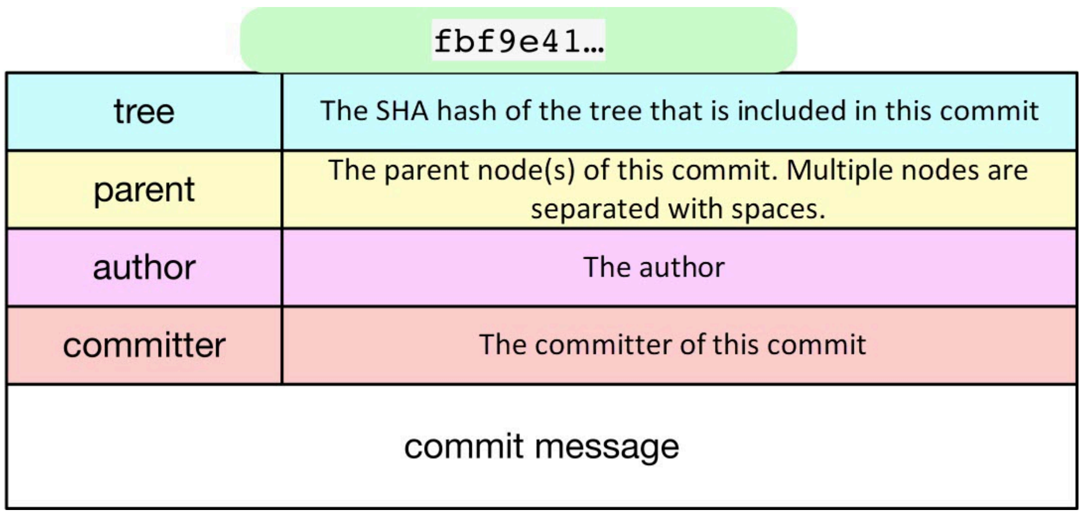
File type Permissions	Object type	SHA1	Object name
100 644	blob	36a982c504eb92330573aa901c7482f7e7c9d2e6	abc.json
040 644	tree	91612a9fa0cea4680228bfb582ed02591ce03ef2	static
100 644	blob	e3c712d7073957c3376d182aeff5b96f28a37098	index.js
100 644	tree	e21682d1ebd4fdd21663ba062c5bfae0308acb64	src

Commit Objects

They are the collection of all the files that are modified currently, and they are similar to the "transactions" that contain a series of operations. A commit object is a snapshot of a collection of modified files. With a commit operation, the modified files will be committed to the local repository. During a versioning process, you can use commit objects to retrieve the contents that are modified each time. These objects are the cornerstone of versioning.

```
→ git cat-file -t fbf9e415f77008b780b40805a9bb996b37a6ad2c
commit
→ git cat-file -p fbf9e415f77008b780b40805a9bb996b37a6ad2c
tree bd31831c26409eac7a79609592919e9dcd1a76f2
parent d62cf8ef977082319d8d8a0cf5150dfa1573c2b7
author xxx  1502331401 +0800
committer xxx  1502331401 +0800
Fix incremental bugs
```

The following shows a detailed description:



fbf9e41…

tree	The SHA hash of the tree that is included in this commit
parent	The parent node(s) of this commit. Multiple nodes are separated with spaces.
author	The author
committer	The committer of this commit
commit message	

Tag Objects

A tag is a "fixed branch." Once you have added a tag, the content represented by the tag will be permanently immutable because the tag is only associated with the last commit object in the version library at the time when the tag was added.

If you use a branch, the content will continuously change due to continuous commits because the last commit that the branch points to will continuously change. Therefore, the release of general applications or software versions typically uses tags.

Git supports two types of tags:

1. Lightweight

Creation Method:

```
git tag tagName
```

When you create a tag with this method, Git will directly point to a commit object instead of creating a real underlying tag object. At this point, the `git cat-file -t tagName` command will return a commit.

```
→ git cat-file -t v4
commit
→ git cat-file -p v4
tree ceab4f96440655b0ff1a783316c95450fa1fb436
parent 7f23c9ca70ce64fc58e8c7507c990c6c6a201d3d
author Yushui  1506224164 +0800
committer Yushui  1506224164 +0800
```

```
rawtest2
```

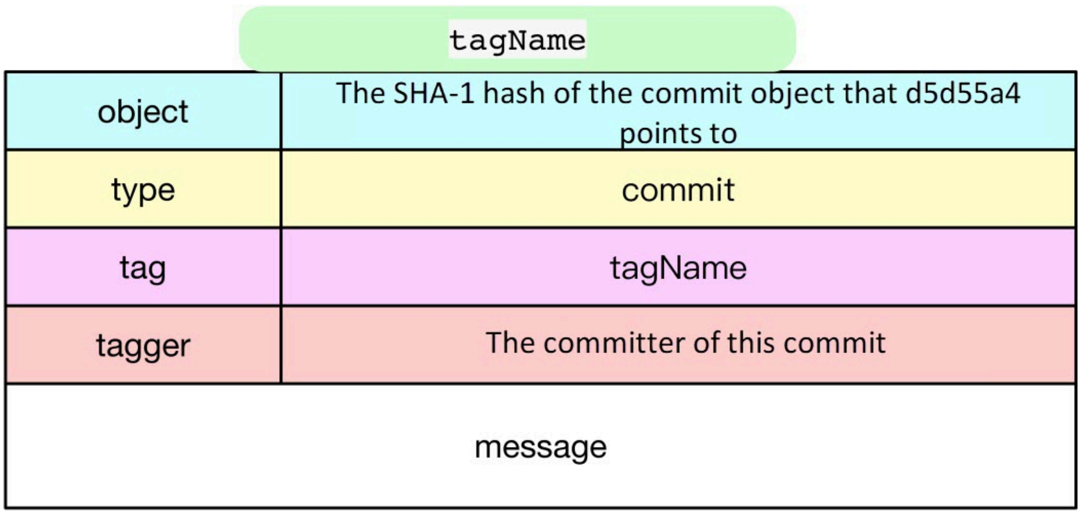
2. Annotated

Creation Method:

```
git tag -a tagName -m''
```

When you create a tag with this method, Git will create an underlying tag object that contains related commit information and additional information, such as the tagger. At this time, if you run the `git cat-file -t tagname` command, a tag will be returned.

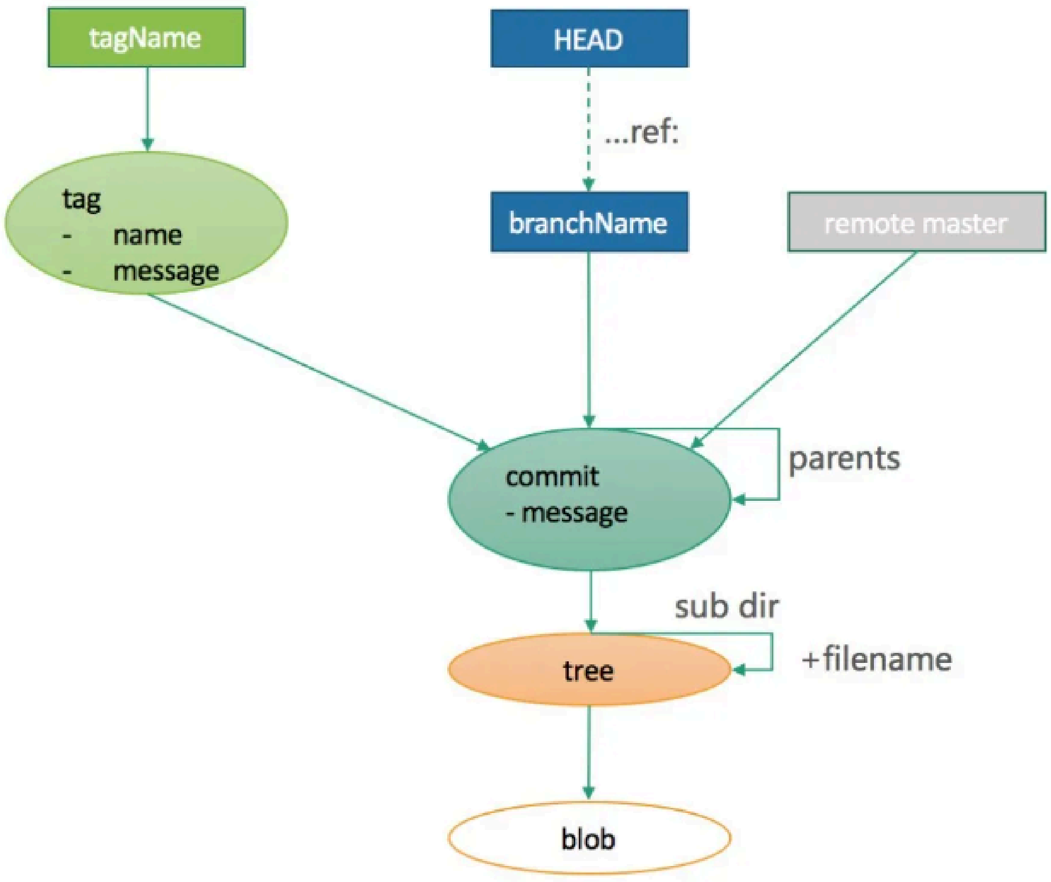
```
→ git cat-file -t v3
tag
→ git cat-file -p v3
object d5d55a49c337d36e16dd4b05bfca3816d8bf6de8   //Commit object SHA-
type commit
tag v3
tagger xxx  1506230900 +0800

Annotated tags tested by Yushui
```



tagName

object	The SHA-1 hash of the commit object that d5d55a4 points to
type	commit
tag	tagName
tagger	The committer of this commit
message	

The relationships between all object models are summarized below:



Storage Models

Concepts

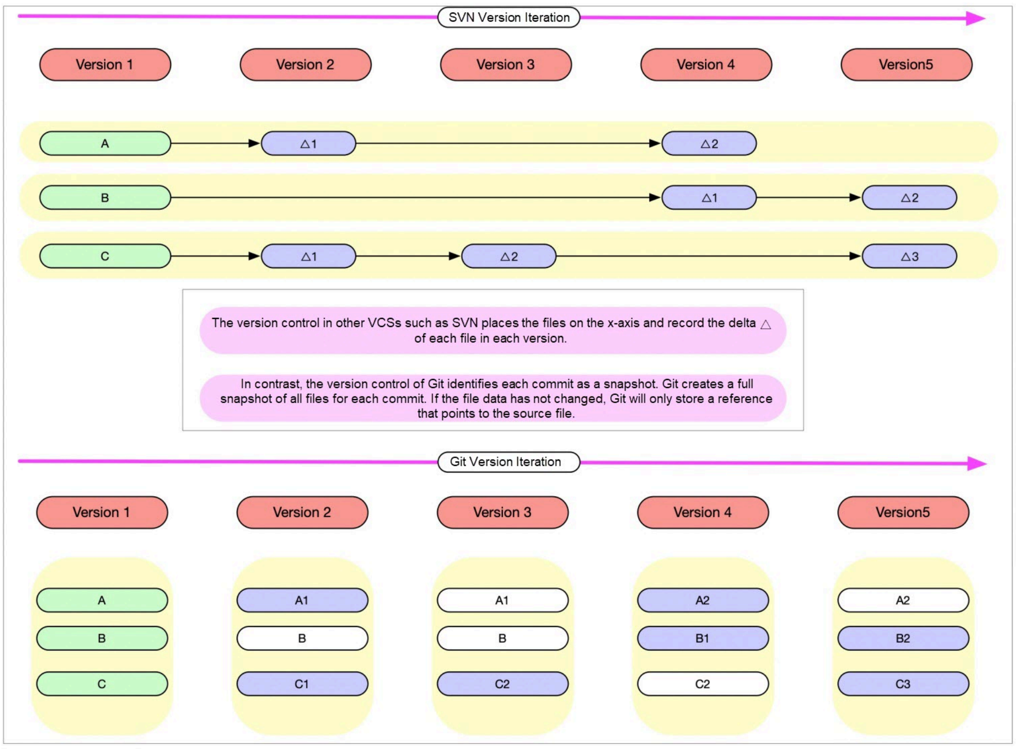
One major difference between Git and other version control systems (VCSs) is that the version control concepts in Git are completely different from the version implementation concepts in other VCSs. This is also the core reason why Git is a powerful version control tool.

The version control in other VCSs, such as Subversion (SVN), places the files on the x-axis and records the delta of each file in each version.

In contrast, the version control of Git identifies each commit as a snapshot. Git creates a full snapshot of all the files for each commit and stores the snapshot for reference.

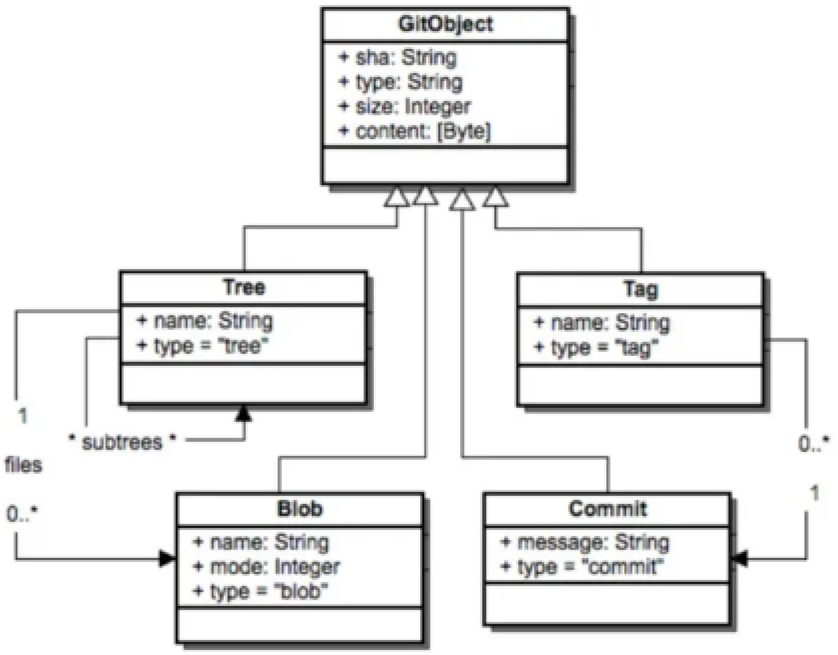
At the storage layer, if the data in a file has not changed, Git will only store a reference that points to the source file instead of directly storing the file multiple times, as you can see in the pack files.

This is shown in the following figure:


SVN Version Iteration
Version 1
Version 2
Version 3
Version 4
Version5
A
△1
△2
B
△1
△2
C
△1
△2
△3
The version control in other VCSs such as SVN places the files on the x-axis and record the delta △ of each file in each version.
In contrast, the version control of Git identifies each commit as a snapshot. Git creates a full snapshot of all files for each commit. If the file data has not changed, Git will only store a reference that points to the source file.
Git Version Iteration
Version 1
Version 2
Version 3
Version 4
Version5
A
B
C
A1
B
C1
A1
B
C2
A2
B1
C2
A2
B2
C3

Storage

Although the versions of Git are constantly updated due to the increasingly complex needs and functions, the main storage model remains almost unchanged. The following figure shows the storage model:



Retrieval Models

```
→ cd .git/objects/
→ ls
03    28    7f    ce    d0    d5    e6    f9    info pack
```

Git supports two types of objects:

The first type is loose objects, such as the 03, 28, 7f, ce, d0, d5, e6, and f9 folders in the preceding .git/objects directory. The name of each folder is the first two characters of the SHA-1 hash of the corresponding file. The maximum number of these folders is #OXFF 256.

The second type is packed objects, which are mostly pack files. These objects are mainly used for network file transfers to reduce bandwidth consumption.

To save the storage space, you can manually trigger the packing operation (using the `git gc` command) to pack loose objects into packed objects. You can also unpack pack files into loose objects (using the `git unpack-objects` command.)

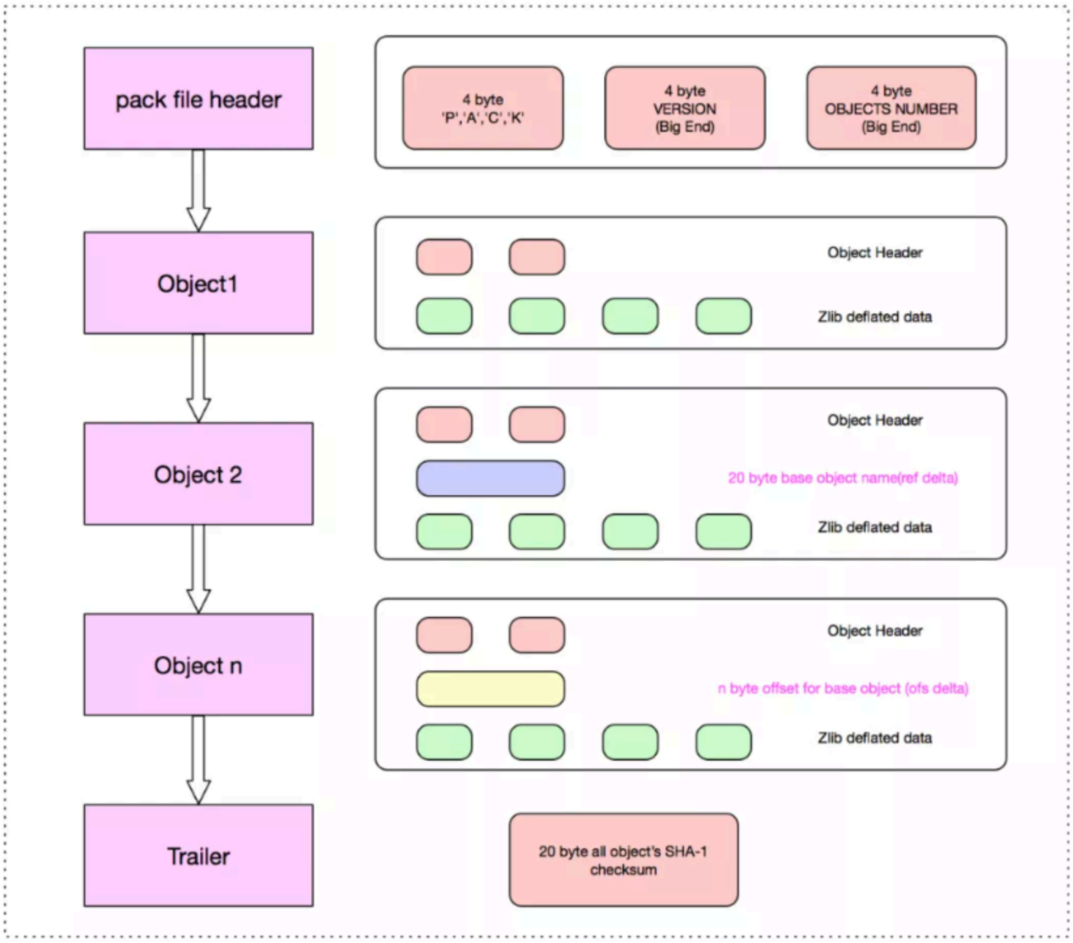
```
→ cd pack
→ ls
pack-efbf3149604d24e6ea427b025da0c59245b2c2ea.idx  pack-efbf3149604d24
```

To speed up the retrieval of pack files, Git generates corresponding idx files based on pack files.

Pack Files

Pack files are precisely and ingeniously designed to reduce file sizes, reduce file transfers, reduce network costs, and secure file transfers.

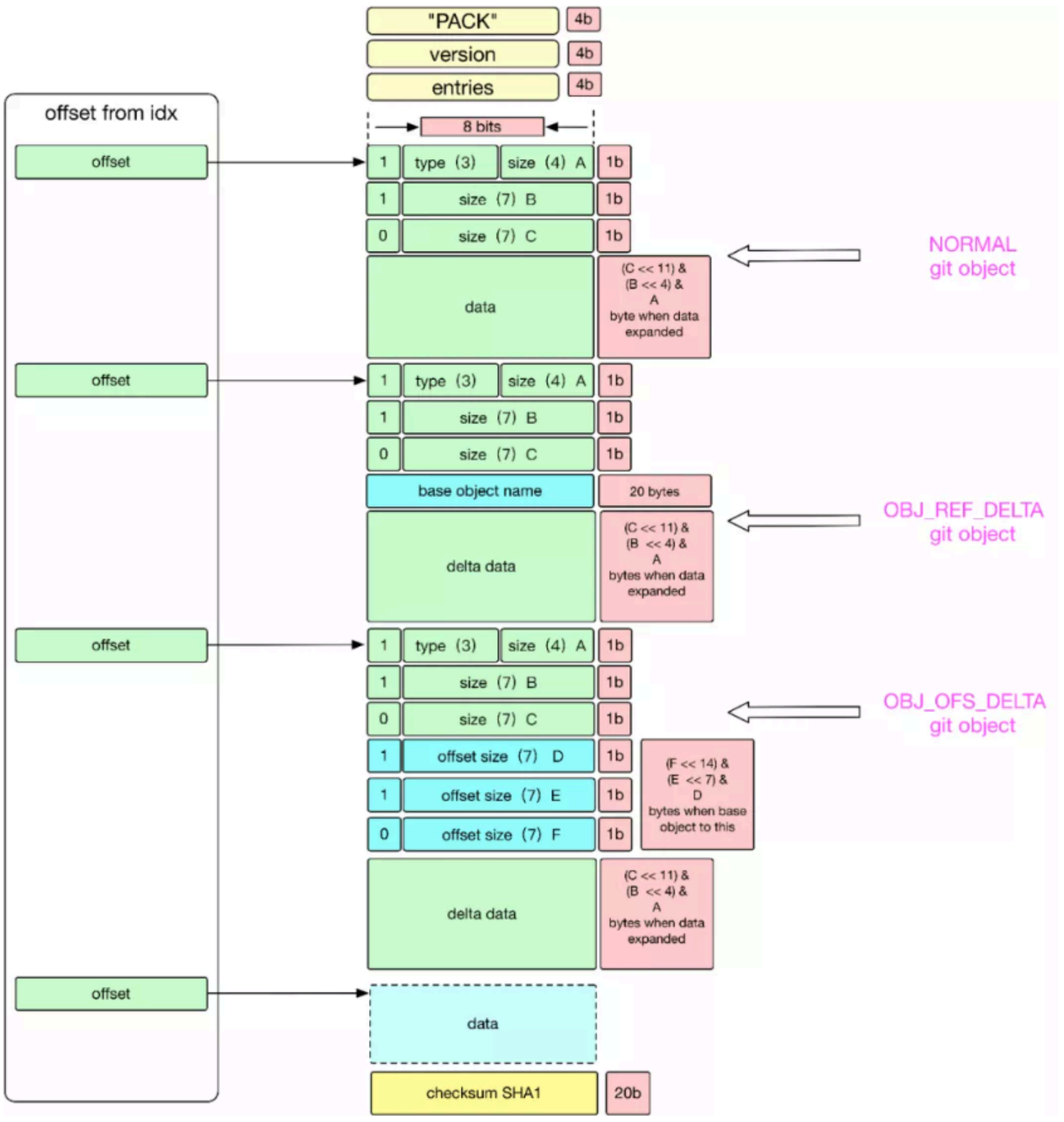
The following figure outlines the design of pack files:



A pack file consists of three main parts: header, body, and trailer.

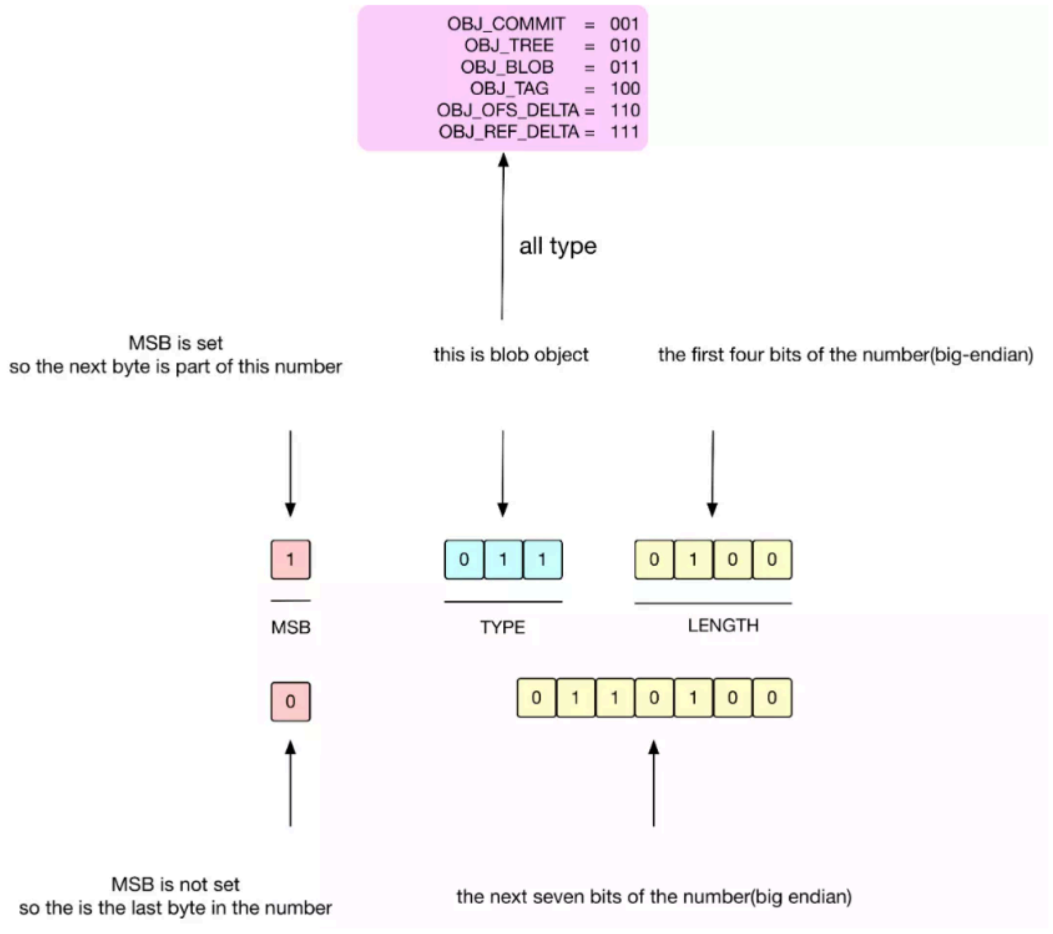
- The header consists of 4-byte "PACK," 4-byte "version," and 4-byte "number of objects."
- The body stores Git objects one by one. An idx file records the offset of each object in the pack file.
- The trailer is SHA-1 checksum of all the objects. It is designed for secure and reliable file transfers.

The following provides a specific pack file:



As shown in the preceding figure, you can locate an object using the corresponding idx file, and you will find that the object consists of two main parts: header and data.

1. Header



Among the first eight bits in the header, the first bit is the most significant bit (MSB.) The subsequent three bits indicate the current object type, which is one of the six main storage types. The latter four bits indicate a portion of the length of the object, which is not the full length of the object. The full length depends on the MSB and the subsequent bits. The complete algorithm is listed below:

If the first bit in the eight bits is 1, it indicates that the next byte is still part of the header and indicates the length of the object.

If the first bit in the eight bits is 0, it indicates that the data file starts from the next byte.

If the object type is `OBJ_OFS_DELTA` , it indicates that the current Git object only stores the deltas. For the basic part, the subsequent variable-length bytes indicate the offset of the base object from the current object. In addition, these bytes also use 1-bit MSB to indicate whether the next byte is part of the variable length. You can take the negative of the offset to figure out the number of bytes between the base object and the current object.

If the object type is `OBJ_REF_DELTA` , it indicates that the current Git object only stores the deltas. For the basic part, the SHA-1 hash of the base object is stored in 20 bytes.

2. Data

The data has been compressed with Zlib. It may either be full data or the delta data, depending on the storage type in the header. If the storage type is `OBJ_OFS_DELTA` or `OBJ_REF_DELTA`, the current object stores the delta data. In this case, if you want to obtain full data, you need to find the base object using recursion and then apply the delta data. Applying the delta data based on the base object is a sophisticated operation, and this article will not delve into it.

In the preceding section, we can find a detailed description of the file formats used for pack files. Next, let's take a closer look at the file formats in the local repository:

Without Deltas:

```
SHA-1 type size size-in-packfile offset-in-packfile
```

With Deltas:

```
SHA-1 type size size-in-packfile offset-in-packfile depth base-SHA-1

→ git verify-pack -v pack-efbf3149604d24e6ea427b025da0c59245b2c2ea.pac
cb5a93c4cf9c0ee5b7153a3a35a4fac7a7584804 commit 275 189 12
399334856af4ca4b49c0008a25b6a9f524e40350 commit 69 81 201 1 cb5a93c4cf
e0efbd5121c31964af1615cf24135a7c6c11cc1d commit 268 187 282
7bc9a5e0199bd4a6d4d223ce7e13239631df9635 commit 29 41 469 1 e0efbd5121
2e43c62f6ff99c88d20329487137f8dbabc8b3ec commit 220 157 510
b6f173085f49f109a00b2a3f08a7dc499cc47f1f commit 220 157 667
0466b3f1aadde74234f7dd3f4ef7f1505c50fb0c commit 220 157 824
76c5e45f8e295226b1bc5c8c7e2bc98d7eae6be1 commit 74 85 981 1 b6f173085f
2729f1fa896d384b49a2f5c53d483eacc0929ebb commit 172 127 1066
3cc58df83752123644fef39faab2393af643b1d2 blob   2 11 1193
62189d1a10cc2a544c4e5b9c4aba9493cf5782dc blob   8 15 1204
a9a5aecf429fd8a0d81fbd5fd37006bfa498d5c1 blob   4 13 1219
2b8982f7c281964658d2cd8b6c17b541533dd277 tree   104 105 1232
92c4aafa39ee387a1f8237f00c78c499aebaf0b2 tree   104 105 1337
223b7836fb19fdf64ba2d3cd6173c6a283141f78 blob   2 11 1442
1756ca64f21724f350fe2cc5cfb218883e314c3d tree   71 80 1453
e11ddfa79f01b01a8e1553bbffaa2d6c03ae9f6e tree   71 80 1533
f70f10e4db19068f79bc43844b49f3eece45c4e8 blob   2 11 1613
e982b6207b10a869164e2c8d19d25ffb059e6a16 tree   66 73 1624
f2e9f73f27124916344e0fd03bb449bc6feca59d tree   66 74 1697
d09da444f461d7cee3679666a1ded5ab79832ed0 tree   33 44 1771
non delta: 18 objects
```

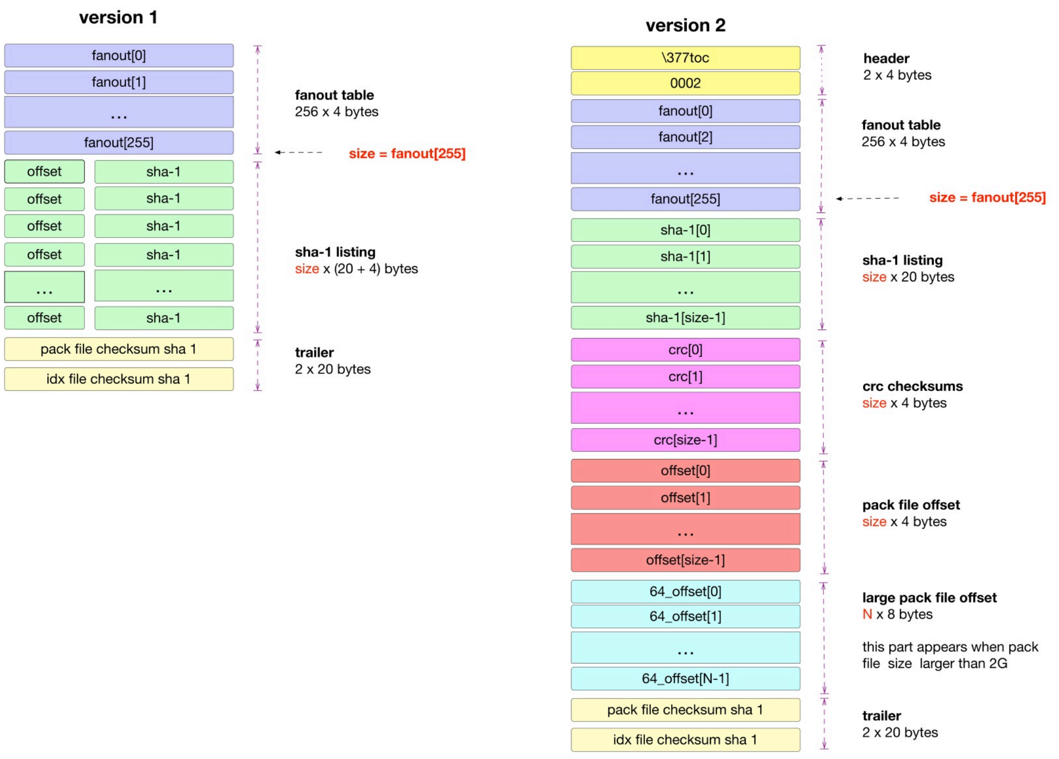
```
chain length = 1: 3 objects
pack-efbf3149604d24e6ea427b025da0c59245b2c2ea.pack: ok
```

For example, `399334856af4ca4b49c0008a25b6a9f524e40350(SHA-1)` indicates that the SHA-1 hash of the base object is `cb5a93c4cf9c0ee5b7153a3a35a4fac7a7584804`. The maximum depth of the base object is 1. If `cb5a93c4cf9c0ee5b7153a3a35a4fac7a7584804` has another referent, change the depth to 2.

The last four bytes in the header of a pack file indicate the number of objects in the pack file. The maximum number is 2 to the power of 32 (232.) Therefore, some large projects may include multiple pack files and multiple idx files.

What do we use the file sizes (the sizes of decompressed files) for? If we know the file sizes, we can figure out the volume of data streams for decompression and set the volume. A file size here does not indicate the offset of the next file. File offsets are in the index files, as shown in the following figure:

Index Files



Version 1 is relatively simple. Therefore, the following uses Version 2 as an example.

Version 2 uses a multi-layer mode: header, fanout, secure hash algorithm (SHA), cyclic redundancy check (CRC), offset, large file offset, and trailer.

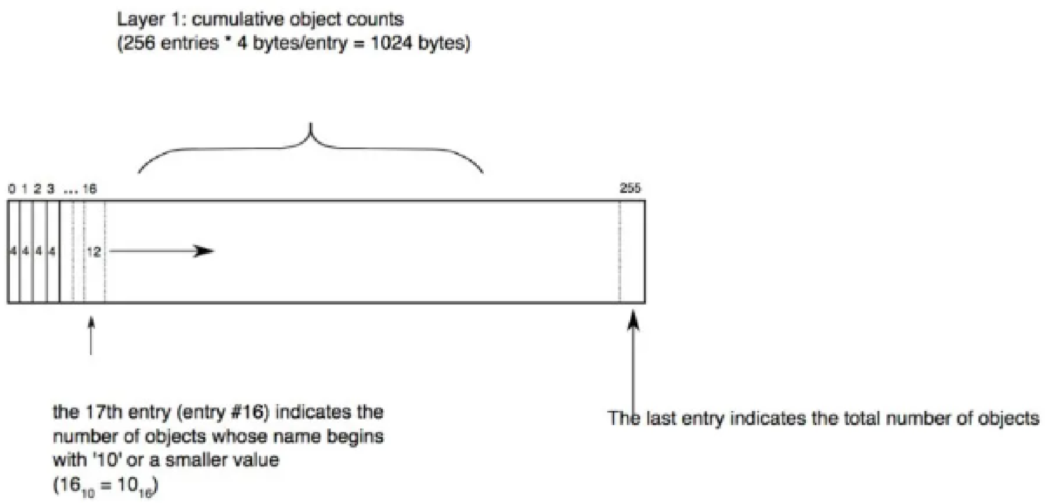
Header

The header in Version 2 contains eight bytes in total, while Version 1 does not have a header. In Version 2, the first four bytes are always 255, 116, 79, and 99, respectively, because they are also the first four bytes in Version 1. The subsequent four bytes indicate the version number, which is currently Version 2.

Fanout

The fanout layer, which is also called the fanout table, is one of the highlights of Git. The fanout arrays store the numbers of related objects, and the indexes are the corresponding hexadecimal numbers. The last fanout array stores the total number of objects in the entire pack file. The fanout table is the core of the entire Git retrieval, and it enables a quick query. With the fanout table, we can locate the start and end indexes of the arrays in the SHA layer. Once we have figured out the range of the SHA layer, we can perform a binary search within this layer instead of on all objects.

The total number of fanouts is 256, which is #0xFF in hexadecimal format. The fanout arrays use the first two characters of the SHA hashes as their indexes. These indexes correspond to the directory names of the loose files in the .git/objects directory and convert the hexadecimal directory names into decimal numbers. The values of these indexes are the numbers of files that have names starting with the two characters. In addition, these values are cumulative layer by layer, that is, the number of objects in an array is the sum of the numbers in the preceding arrays.


Layer 1: cumulative object counts
(256 entries * 4 bytes/entry = 1024 bytes)
0 1 2 3 ... 16
255
4 4 4 4
12
the 17th entry (entry #16) indicates the number of objects whose name begins with '10' or a smaller value
($16_{10} = 10_{16}$)
The last entry indicates the total number of objects

For example:

1. If the index is 0 and fanout[0] equals 10, it indicates that the total number of SHA-1 hashes starting with #0x00 is 10.
2. If the index is 1 and fanout[1] equals 15, it indicates that the total number of SHA-1 hashes starting with a number smaller than #0x01 is 15. We can conclude that fanout[1] equals 15 minus 10 because fanout[0] equals 10.

Why is the number of fanout[n] designed to include the number of fanout[n-1] in Git? The main purpose is to quickly identify the location where the retrieval in the SHA layer starts, so there is no need to accumulate the numbers of all the previous fanout[..n-1] arrays every time.

SHA

At this layer, the SHA-1 hashes of all objects are sorted by their names. This sorting method can facilitate binary search. Each SHA-1 hash takes 20 bytes.

CRC

Files are packed mainly to address network transmission problems. Therefore, it is necessary to perform CRC to prevent files from being damaged during network transmission. The CRC arrays correspond to the CRC checksum of each object.

Offset

This layer consists of four bytes, which indicate the offset of each SHA-1 file. However, if the file size is larger than 2 GB, the offset cannot be indicated by four bytes. In this case, the first bit in the four bytes is the MSB. If it is 1, the offset of the file is stored in the sixth layer, and the remaining 31 bits indicate the offset of the file at the large file offset layer. As shown in the figure, we can obtain the offset of the object in the pack file from the large file offset layer.

The first bit in the four bytes is the MSB. If it is 0, the remaining 31 bits indicate the offset of the stored object in the pack file. This situation does not involve the large file offset layer.

Large File Offset

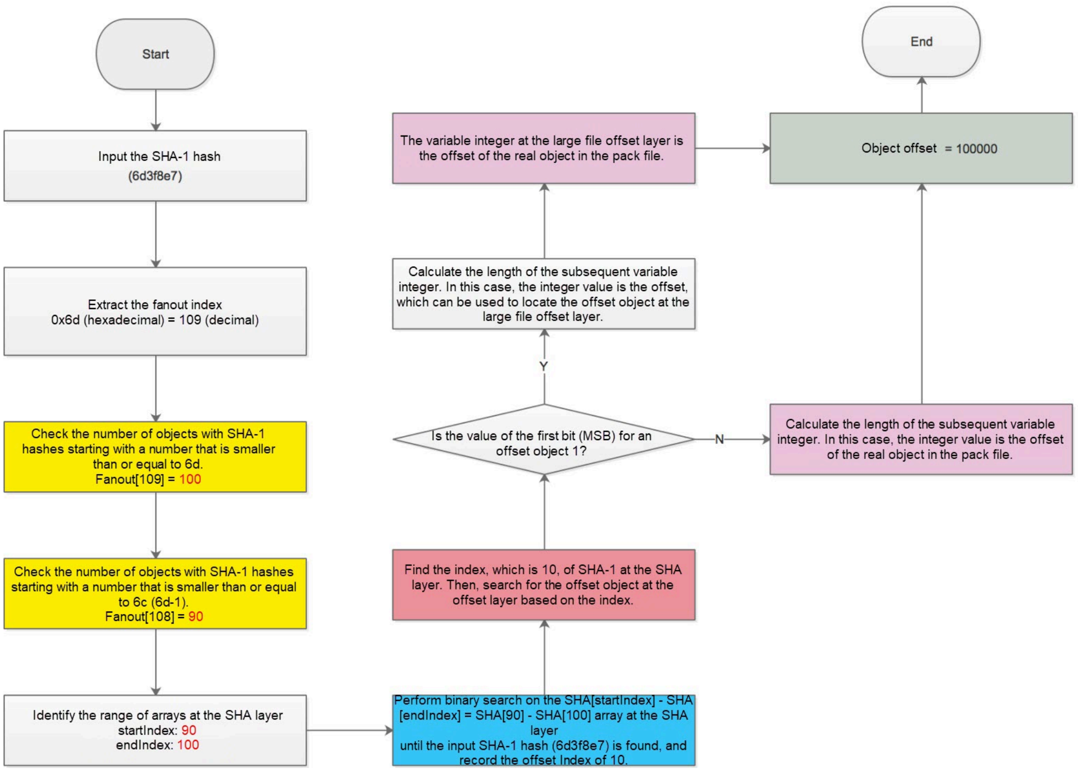
This layer is used to store the offsets of the files that are larger than 2 GB. If a file is larger than 2 GB, the last 31 bits of the offset layer specify the offset of the file at the large file offset layer. The large file offset layer uses eight bytes to indicate the location of an object in the pack file. Theoretically, the eight bytes can indicate a file size of 2 to the power of 64 (264.)

Trailer

This layer contains the pack file checksum and the associated idx checksum.

Indexing Process

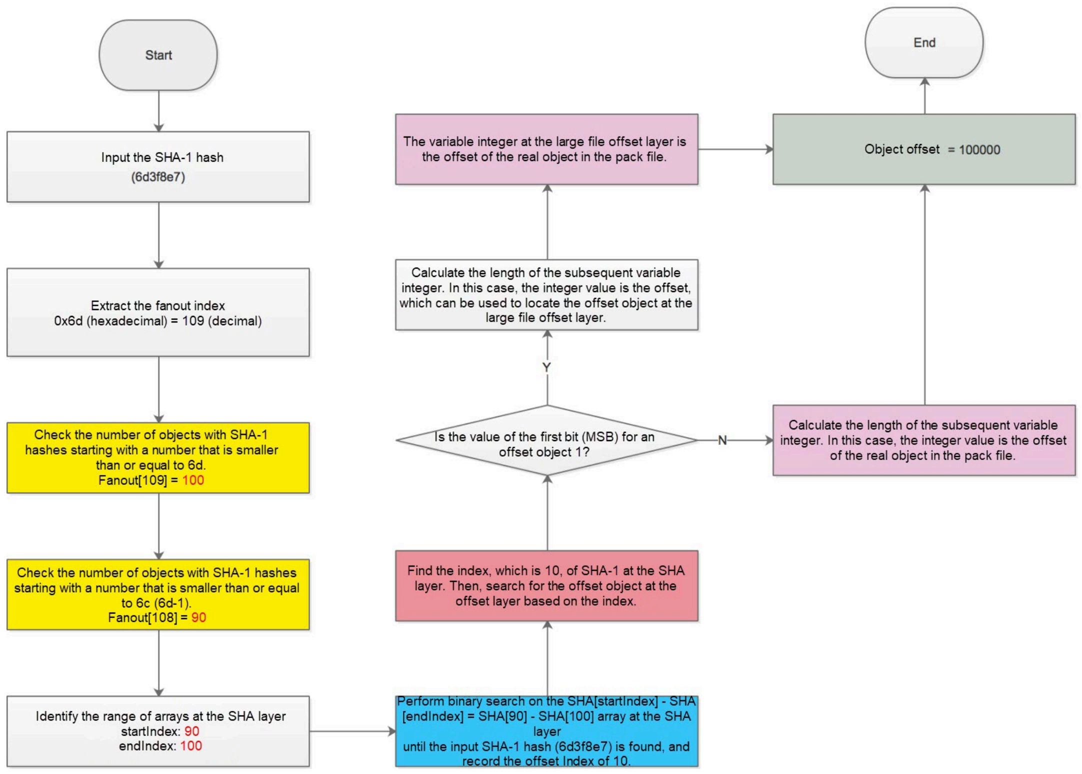
We can see the ingenious design of Git in the preceding multi-layer mode. The following figure shows the process of querying the offsets of the index files in Git.



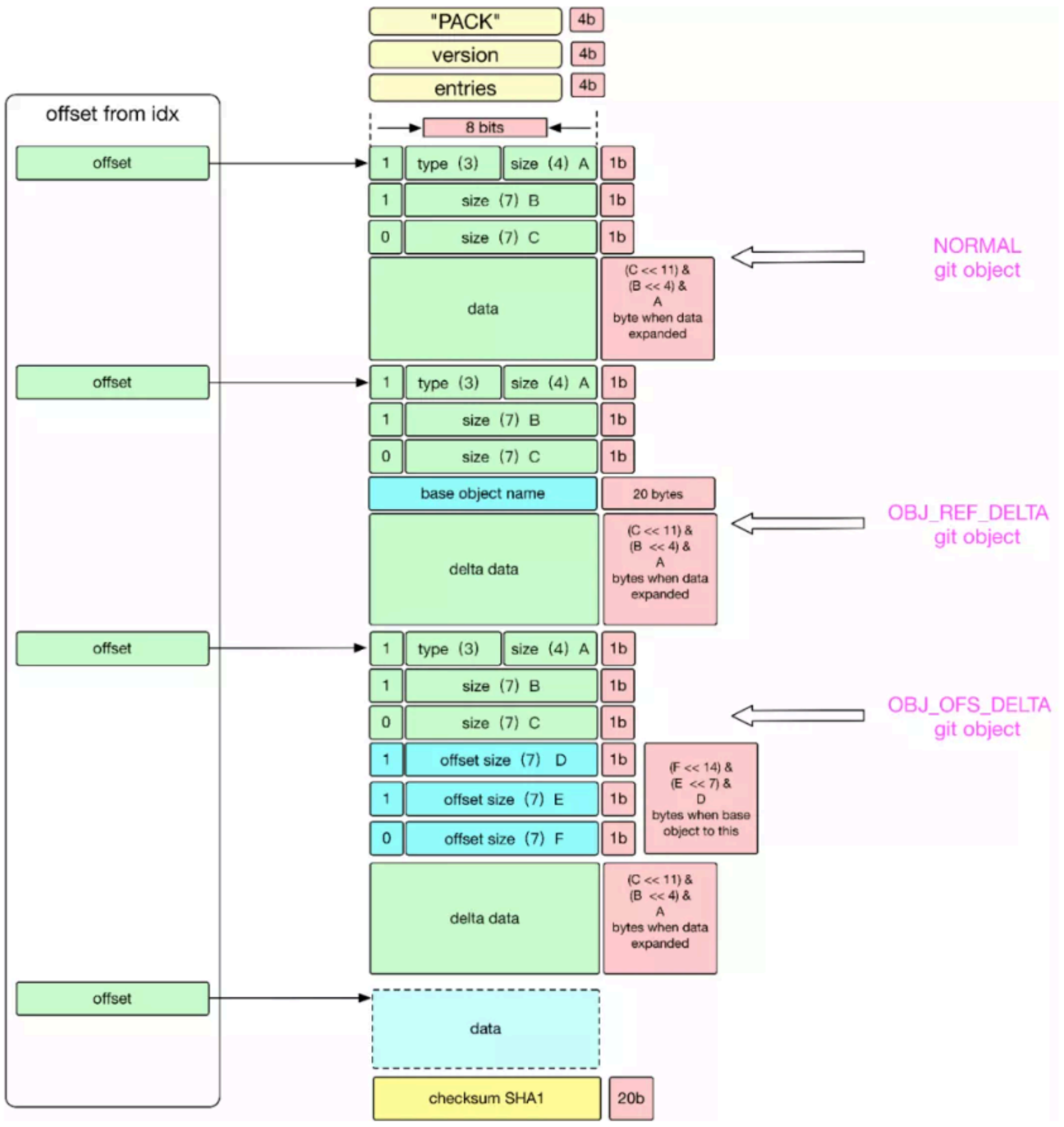
Start
Input the SHA-1 hash (6d3f8e7)
Extract the fanout index 0x6d (hexadecimal) = 109 (decimal)
Check the number of objects with SHA-1 hashes starting with a number that is smaller than or equal to 6d. Fanout[109] = 100
Check the number of objects with SHA-1 hashes starting with a number that is smaller than or equal to 6c (6d-1). Fanout[108] = 90
Identify the range of arrays at the SHA layer startIndex: 90 endIndex: 100
Perform binary search on the SHA[startIndex] - SHA [endIndex] = SHA[90] - SHA[100] array at the SHA layer until the input SHA-1 hash (6d3f8e7) is found, and record the offset Index of 10.
Find the index, which is 10, of SHA-1 at the SHA layer. Then, search for the offset object at the offset layer based on the index.
Is the value of the first bit (MSB) for an offset object 1?
Y
N
Calculate the length of the subsequent variable integer. In this case, the integer value is the offset, which can be used to locate the offset object at the large file offset layer.
The variable integer at the large file offset layer is the offset of the real object in the pack file.
Calculate the length of the subsequent variable integer. In this case, the integer value is the offset of the real object in the pack file.
Object offset = 100000
End

Query Algorithm

The following figure shows how to use the idx file to query the offset corresponding to the SHA-1 hash.


Start
Input the SHA-1 hash (6d3f8e7)
Extract the fanout index 0x6d (hexadecimal) = 109 (decimal)
Check the number of objects with SHA-1 hashes starting with a number that is smaller than or equal to 6d. Fanout[109] = 100
Check the number of objects with SHA-1 hashes starting with a number that is smaller than or equal to 6c (6d-1). Fanout[108] = 90
Identify the range of arrays at the SHA layer startIndex: 90 endIndex: 100
Perform binary search on the SHA[startIndex] - SHA[endIndex] = SHA[90] - SHA[100] array at the SHA layer until the input SHA-1 hash (6d3f8e7) is found, and record the offset Index of 10.
Find the index, which is 10, of SHA-1 at the SHA layer. Then, search for the offset object at the offset layer based on the index.
Is the value of the first bit (MSB) for an offset object 1?
Y
Calculate the length of the subsequent variable integer. In this case, the integer value is the offset, which can be used to locate the offset object at the large file offset layer.
The variable integer at the large file offset layer is the offset of the real object in the pack file.
N
Calculate the length of the subsequent variable integer. In this case, the integer value is the offset of the real object in the pack file.
Object offset = 100000
End

The following figure shows how to locate the objects in a pack file based on the offsets:



For a normal storage type, you can locate the files that are compressed with Zlib and directly decompress them.

For the delta data, recursion is performed to find the base object, and then the delta data is applied to the base object (see `git-apply-delta`).

Storage developer Indexing Data structure Git Repository

0 0 0

Share on

Read previous post:

DevOps Approach with Network-on-Chip and System-on-Chip

Read next post:

Deploying a FaaS Platform for Content Delivery

You may also like

Breakdown! A Detailed Comparison Between the Flutter Widget and CSS in Terms of Layout Principles

淘系技术 - December 9, 2020

An Interpretation of the Source Code of OceanBase (6): Detailed Explanation of Storage Engine

OceanBase - September 13, 2022

Implement Your Own Control Recognition Model with Pipcook

Alibaba F(x) Team - December 31, 2020

OBProxy: A Detailed Explanation of the Functional Modules and Features

OceanBase - August 26, 2022

Design Ideas for Improving the Transaction System of Ele.me, Alibaba's Food Delivery Service

Aliware - May 15, 2020

Application of Alink and Tensorflow on Flink in JD

Apache Flink Community China - April 13, 2022

Comments

Write your comment...

Post