Very first version of Git in Git

Linus Torvalds

e83c51633

Contents

1 README

GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

 - random three-letter combination that is pronounceable, and not
   actually used by any common UNIX command. The fact that it is a
   mispronounciation of "get" may or may not be relevant.
 - stupid. contemptible and despicable. simple. Take your pick from the
   dictionary of slang.
 - "global information tracker": you're in a good mood, and it actually
   works for you. Angels sing, and a light suddenly fills the room.
 - "goddamn idiotic truckload of sh*t": when it breaks

This is a stupid (but extremely fast) directory content manager. It
doesn't do a whole lot, but what it _does_ do is track directory
contents efficiently.

There are two object abstractions: the "object database", and the
"current directory cache".

The Object Database (SHA1_FILE_DIRECTORY)

The object database is literally just a content-addressable collection
of objects. All objects are named by their content, which is
approximated by the SHA1 hash of the object itself. Objects may refer
to other objects (by referencing their SHA1 hash), and so you can build
up a hierarchy of objects.

There are several kinds of objects in the content-addressable collection
database. They are all in deflated with zlib, and start off with a tag
of their type, and size information about the data. The SHA1 hash is
always the hash of the _compressed_ object, not the original one.

In particular, the consistency of an object can always be tested
independently of the contents or the type of the object: all objects can
be validated by verifying that (a) their hashes match the content of the
file and (b) the object successfully inflates to a stream of bytes that
forms a sequence of <ascii tag without space> + <space> + <ascii decimal
size> + <byte\0> + <binary object data>.

BLOB: A "blob" object is nothing but a binary blob of data, and doesn't
refer to anything else. There is no signature or any other verification
of the data, so while the object is consistent (it _is_ indexed by its
sha1 hash, so the data itself is certainly correct), it has absolutely
no other attributes. No name associations, no permissions. It is
purely a blob of data (ie normally "file contents").

TREE: The next hierarchical object type is the "tree" object. A tree
object is a list of permission/name/blob data, sorted by name. In other
words the tree object is uniquely determined by the set contents, and so
two separate but identical trees will always share the exact same
object.

Again, a "tree" object is just a pure data abstraction: it has no
history, no signatures, no verification of validity, except that the
contents are again protected by the hash itself. So you can trust the

contents of a tree, the same way you can trust the contents of a blob,
but you don't know where those contents _came_ from.

Side note on trees: since a "tree" object is a sorted list of
"filename+content", you can create a diff between two trees without
actually having to unpack two trees. Just ignore all common parts, and
your diff will look right. In other words, you can effectively (and
efficiently) tell the difference between any two random trees by O(n)
where "n" is the size of the difference, rather than the size of the
tree.

Side note 2 on trees: since the name of a "blob" depends entirely and
exclusively on its contents (ie there are no names or permissions
involved), you can see trivial renames or permission changes by noticing
that the blob stayed the same. However, renames with data changes need
a smarter "diff" implementation.

CHANGESET: The "changeset" object is an object that introduces the
notion of history into the picture. In contrast to the other objects,
it doesn't just describe the physical state of a tree, it describes how
we got there, and why.

A "changeset" is defined by the tree-object that it results in, the
parent changesets (zero, one or more) that led up to that point, and a
comment on what happened. Again, a changeset is not trusted per se:
the contents are well-defined and "safe" due to the cryptographically
strong signatures at all levels, but there is no reason to believe that
the tree is "good" or that the merge information makes sense. The
parents do not have to actually have any relationship with the result,
for example.

Note on changesets: unlike real SCM's, changesets do not contain rename
information or file mode chane information. All of that is implicit in
the trees involved (the result tree, and the result trees of the
parents), and describing that makes no sense in this idiotic file
manager.

TRUST: The notion of "trust" is really outside the scope of "git", but
it's worth noting a few things. First off, since everything is hashed
with SHA1, you _can_ trust that an object is intact and has not been
messed with by external sources. So the name of an object uniquely
identifies a known state - just not a state that you may want to trust.

Furthermore, since the SHA1 signature of a changeset refers to the
SHA1 signatures of the tree it is associated with and the signatures
of the parent, a single named changeset specifies uniquely a whole
set of history, with full contents. You can't later fake any step of
the way once you have the name of a changeset.

So to introduce some real trust in the system, the only thing you need
to do is to digitally sign just _one_ special note, which includes the
name of a top-level changeset. Your digital signature shows others that
you trust that changeset, and the immutability of the history of
changesets tells others that they can trust the whole history.

In other words, you can easily validate a whole archive by just sending
out a single email that tells the people the name (SHA1 hash) of the top
changeset, and digitally sign that email using something like GPG/PGP.

In particular, you can also have a separate archive of "trust points" or
tags, which document your (and other peoples) trust. You may, of
course, archive these "certificates of trust" using "git" itself, but
it's not something "git" does for you.

Another way of saying the same thing: "git" itself only handles content
integrity, the trust has to come from outside.

Current Directory Cache (".dircache/index")

The "current directory cache" is a simple binary file, which contains an
efficient representation of a virtual directory content at some random
time. It does so by a simple array that associates a set of names,
dates, permissions and content (aka "blob") objects together. The cache
is always kept ordered by name, and names are unique at any point in
time, but the cache has no long-term meaning, and can be partially
updated at any time.

In particular, the "current directory cache" certainly does not need to
be consistent with the current directory contents, but it has two very
important attributes:

 (a) it can re-generate the full state it caches (not just the directory
     structure: through the "blob" object it can regenerate the data too)

     As a special case, there is a clear and unambiguous one-way mapping
     from a current directory cache to a "tree object", which can be
     efficiently created from just the current directory cache without
     actually looking at any other data. So a directory cache at any
     one time uniquely specifies one and only one "tree" object (but
     has additional data to make it easy to match up that tree object
     with what has happened in the directory)

and

 (b) it has efficient methods for finding inconsistencies between that
     cached state ("tree object waiting to be instantiated") and the
     current state.

Those are the two ONLY things that the directory cache does. It's a
cache, and the normal operation is to re-generate it completely from a
known tree object, or update/compare it with a live tree that is being
developed. If you blow the directory cache away entirely, you haven't
lost any information as long as you have the name of the tree that it
described.

(But directory caches can also have real information in them: in
particular, they can have the representation of an intermediate tree
that has not yet been instantiated. So they do have meaning and usage
outside of caching - in one sense you can think of the current directory
cache as being the "work in progress" towards a tree commit).

2 Makefile

```
CFLAGS=-g
CC=gcc

PROG=update-cache show-diff init-db write-tree read-tree commit-tree cat-file

all: $(PROG)

install: $(PROG)
    install $(PROG) $(HOME)/bin/

LIBS= -lssl

init-db: init-db.o

update-cache: update-cache.o read-cache.o
    $(CC) $(CFLAGS) -o update-cache update-cache.o read-cache.o $(LIBS)

show-diff: show-diff.o read-cache.o
    $(CC) $(CFLAGS) -o show-diff show-diff.o read-cache.o $(LIBS)

write-tree: write-tree.o read-cache.o
    $(CC) $(CFLAGS) -o write-tree write-tree.o read-cache.o $(LIBS)

read-tree: read-tree.o read-cache.o
    $(CC) $(CFLAGS) -o read-tree read-tree.o read-cache.o $(LIBS)

commit-tree: commit-tree.o read-cache.o
    $(CC) $(CFLAGS) -o commit-tree commit-tree.o read-cache.o $(LIBS)

cat-file: cat-file.o read-cache.o
    $(CC) $(CFLAGS) -o cat-file cat-file.o read-cache.o $(LIBS)

read-cache.o: cache.h
show-diff.o: cache.h

clean:
    rm -f *.o $(PROG) temp_git_file_*

backup: clean
    cd .. ; tar czvf dircache.tar.gz dir-cache
```

3 cache.h

```
#ifndef CACHE_H
#define CACHE_H

#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <sys/mman.h>

#include <openssl/sha.h>
#include <zlib.h>

/*
 * Basic data structures for the directory cache
 *
 * NOTE NOTE NOTE! This is all in the native CPU byte format. It's
 * not even trying to be portable. It's trying to be efficient. It's
 * just a cache, after all.
 */

#define CACHE_SIGNATURE 0x44495243    /* "DIRC" */
struct cache_header {
    unsigned int signature;
    unsigned int version;
    unsigned int entries;
    unsigned char sha1[20];
};

/*
 * The "cache_time" is just the low 32 bits of the
 * time. It doesn't matter if it overflows - we only
 * check it for equality in the 32 bits we save.
 */
struct cache_time {
    unsigned int sec;
    unsigned int nsec;
};

/*
 * dev/ino/uid/gid/size are also just tracked to the low 32 bits
 * Again - this is just a (very strong in practice) heuristic that
 * the inode hasn't changed.
 */
struct cache_entry {
    struct cache_time ctime;
    struct cache_time mtime;
    unsigned int st_dev;
    unsigned int st_ino;
    unsigned int st_mode;
    unsigned int st_uid;
    unsigned int st_gid;
    unsigned int st_size;
```

```
56      unsigned char sha1[20];
57      unsigned short namelen;
58      unsigned char name[0];
59  };
60
61  const char *sha1_file_directory;
62  struct cache_entry **active_cache;
63  unsigned int active_nr, active_alloc;
64
65  #define DB_ENVIRONMENT "SHA1_FILE_DIRECTORY"
66  #define DEFAULT_DB_ENVIRONMENT ".dircache/objects"
67
68  #define cache_entry_size(len) ((offsetof(struct cache_entry,name) + (len) + 8) & ~7)
69  #define ce_size(ce) cache_entry_size((ce)->namelen)
70
71  #define alloc_nr(x) (((x)+16)*3/2)
72
73  /* Initialize the cache information */
74  extern int read_cache(void);
75
76  /* Return a statically allocated filename matching the sha1 signature */
77  extern char *sha1_file_name(unsigned char *sha1);
78
79  /* Write a memory buffer out to the sha file */
80  extern int write_sha1_buffer(unsigned char *sha1, void *buf, unsigned int size);
81
82  /* Read and unpack a sha1 file into memory, write memory to a sha1 file */
83  extern void * read_sha1_file(unsigned char *sha1, char *type, unsigned long *size);
84  extern int write_sha1_file(char *buf, unsigned len);
85
86  /* Convert to/from hex/sha1 representation */
87  extern int get_sha1_hex(char *hex, unsigned char *sha1);
88  extern char *sha1_to_hex(unsigned char *sha1);    /* static buffer! */
89
90  /* General helper functions */
91  extern void usage(const char *err);
92
93  #endif /* CACHE_H */
```

4 read-cache.c

```c
#include "cache.h"

const char *sha1_file_directory = NULL;
struct cache_entry **active_cache = NULL;
unsigned int active_nr = 0, active_alloc = 0;

void usage(const char *err)
{
    fprintf(stderr, "read-tree: %s\n", err);
    exit(1);
}

static unsigned hexval(char c)
{
    if (c >= '0' && c <= '9')
        return c - '0';
    if (c >= 'a' && c <= 'f')
        return c - 'a' + 10;
    if (c >= 'A' && c <= 'F')
        return c - 'A' + 10;
    return ~0;
}

int get_sha1_hex(char *hex, unsigned char *sha1)
{
    int i;
    for (i = 0; i < 20; i++) {
        unsigned int val = (hexval(hex[0]) << 4) | hexval(hex[1]);
        if (val & ~0xff)
            return -1;
        *sha1++ = val;
        hex += 2;
    }
    return 0;
}

char * sha1_to_hex(unsigned char *sha1)
{
    static char buffer[50];
    static const char hex[] = "0123456789abcdef";
    char *buf = buffer;
    int i;

    for (i = 0; i < 20; i++) {
        unsigned int val = *sha1++;
        *buf++ = hex[val >> 4];
        *buf++ = hex[val & 0xf];
    }
    return buffer;
}

/*
 * NOTE! This returns a statically allocated buffer, so you have to be
 * careful about using it. Do a "strdup()" if you need to save the
 * filename.
```

```
 */
char *sha1_file_name(unsigned char *sha1)
{
    int i;
    static char *name, *base;

    if (!base) {
        char *sha1_file_directory = getenv(DB_ENVIRONMENT) ? : DEFAULT_DB_ENVIRONMENT;
        int len = strlen(sha1_file_directory);
        base = malloc(len + 60);
        memcpy(base, sha1_file_directory, len);
        memset(base+len, 0, 60);
        base[len] = '/';
        base[len+3] = '/';
        name = base + len + 1;
    }
    for (i = 0; i < 20; i++) {
        static char hex[] = "0123456789abcdef";
        unsigned int val = sha1[i];
        char *pos = name + i*2 + (i > 0);
        *pos++ = hex[val >> 4];
        *pos = hex[val & 0xf];
    }
    return base;
}

void * read_sha1_file(unsigned char *sha1, char *type, unsigned long *size)
{
    z_stream stream;
    char buffer[8192];
    struct stat st;
    int i, fd, ret, bytes;
    void *map, *buf;
    char *filename = sha1_file_name(sha1);

    fd = open(filename, O_RDONLY);
    if (fd < 0) {
        perror(filename);
        return NULL;
    }
    if (fstat(fd, &st) < 0) {
        close(fd);
        return NULL;
    }
    map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd);
    if (-1 == (int)(long)map)
        return NULL;

    /* Get the data stream */
    memset(&stream, 0, sizeof(stream));
    stream.next_in = map;
    stream.avail_in = st.st_size;
    stream.next_out = buffer;
    stream.avail_out = sizeof(buffer);

    inflateInit(&stream);
    ret = inflate(&stream, 0);
```

```
114     if (sscanf(buffer, "%10s %lu", type, size) != 2)
115         return NULL;
116     bytes = strlen(buffer) + 1;
117     buf = malloc(*size);
118     if (!buf)
119         return NULL;
120 
121     memcpy(buf, buffer + bytes, stream.total_out - bytes);
122     bytes = stream.total_out - bytes;
123     if (bytes < *size && ret == Z_OK) {
124         stream.next_out = buf + bytes;
125         stream.avail_out = *size - bytes;
126         while (inflate(&stream, Z_FINISH) == Z_OK)
127             /* nothing */;
128     }
129     inflateEnd(&stream);
130     return buf;
131 }
132 
133 int write_sha1_file(char *buf, unsigned len)
134 {
135     int size;
136     char *compressed;
137     z_stream stream;
138     unsigned char sha1[20];
139     SHA_CTX c;
140 
141     /* Set it up */
142     memset(&stream, 0, sizeof(stream));
143     deflateInit(&stream, Z_BEST_COMPRESSION);
144     size = deflateBound(&stream, len);
145     compressed = malloc(size);
146 
147     /* Compress it */
148     stream.next_in = buf;
149     stream.avail_in = len;
150     stream.next_out = compressed;
151     stream.avail_out = size;
152     while (deflate(&stream, Z_FINISH) == Z_OK)
153         /* nothing */;
154     deflateEnd(&stream);
155     size = stream.total_out;
156 
157     /* Sha1.. */
158     SHA1_Init(&c);
159     SHA1_Update(&c, compressed, size);
160     SHA1_Final(sha1, &c);
161 
162     if (write_sha1_buffer(sha1, compressed, size) < 0)
163         return -1;
164     printf("%s\n", sha1_to_hex(sha1));
165     return 0;
166 }
167 
168 int write_sha1_buffer(unsigned char *sha1, void *buf, unsigned int size)
169 {
170     char *filename = sha1_file_name(sha1);
171     int i, fd;
```

```
172
173     fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
174     if (fd < 0)
175         return (errno == EEXIST) ? 0 : -1;
176     write(fd, buf, size);
177     close(fd);
178     return 0;
179 }
180
181 static int error(const char * string)
182 {
183     fprintf(stderr, "error: %s\n", string);
184     return -1;
185 }
186
187 static int verify_hdr(struct cache_header *hdr, unsigned long size)
188 {
189     SHA_CTX c;
190     unsigned char sha1[20];
191
192     if (hdr->signature != CACHE_SIGNATURE)
193         return error("bad signature");
194     if (hdr->version != 1)
195         return error("bad version");
196     SHA1_Init(&c);
197     SHA1_Update(&c, hdr, offsetof(struct cache_header, sha1));
198     SHA1_Update(&c, hdr+1, size - sizeof(*hdr));
199     SHA1_Final(sha1, &c);
200     if (memcmp(sha1, hdr->sha1, 20))
201         return error("bad header sha1");
202     return 0;
203 }
204
205 int read_cache(void)
206 {
207     int fd, i;
208     struct stat st;
209     unsigned long size, offset;
210     void *map;
211     struct cache_header *hdr;
212
213     errno = EBUSY;
214     if (active_cache)
215         return error("more than one cachefile");
216     errno = ENOENT;
217     sha1_file_directory = getenv(DB_ENVIRONMENT);
218     if (!sha1_file_directory)
219         sha1_file_directory = DEFAULT_DB_ENVIRONMENT;
220     if (access(sha1_file_directory, X_OK) < 0)
221         return error("no access to SHA1 file directory");
222     fd = open(".dircache/index", O_RDONLY);
223     if (fd < 0)
224         return (errno == ENOENT) ? 0 : error("open failed");
225
226     map = (void *)-1;
227     if (!fstat(fd, &st)) {
228         map = NULL;
229         size = st.st_size;
```

```
230         errno = EINVAL;
231         if (size > sizeof(struct cache_header))
232             map = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
233     }
234     close(fd);
235     if (-1 == (int)(long)map)
236         return error("mmap failed");
237 
238     hdr = map;
239     if (verify_hdr(hdr, size) < 0)
240         goto unmap;
241 
242     active_nr = hdr->entries;
243     active_alloc = alloc_nr(active_nr);
244     active_cache = calloc(active_alloc, sizeof(struct cache_entry *));
245 
246     offset = sizeof(*hdr);
247     for (i = 0; i < hdr->entries; i++) {
248         struct cache_entry *ce = map + offset;
249         offset = offset + ce_size(ce);
250         active_cache[i] = ce;
251     }
252     return active_nr;
253 
254 unmap:
255     munmap(map, size);
256     errno = EINVAL;
257     return error("verify header failed");
258 }
```

5 update-cache.c

```c
#include "cache.h"

static int cache_name_compare(const char *name1, int len1, const char *name2, int len2)
{
    int len = len1 < len2 ? len1 : len2;
    int cmp;

    cmp = memcmp(name1, name2, len);
    if (cmp)
        return cmp;
    if (len1 < len2)
        return -1;
    if (len1 > len2)
        return 1;
    return 0;
}

static int cache_name_pos(const char *name, int namelen)
{
    int first, last;

    first = 0;
    last = active_nr;
    while (last > first) {
        int next = (last + first) >> 1;
        struct cache_entry *ce = active_cache[next];
        int cmp = cache_name_compare(name, namelen, ce->name, ce->namelen);
        if (!cmp)
            return -next-1;
        if (cmp < 0) {
            last = next;
            continue;
        }
        first = next+1;
    }
    return first;
}

static int remove_file_from_cache(char *path)
{
    int pos = cache_name_pos(path, strlen(path));
    if (pos < 0) {
        pos = -pos-1;
        active_nr--;
        if (pos < active_nr)
            memmove(active_cache + pos, active_cache + pos + 1, (active_nr - pos - 1) * sizeof(struc
    }
}

static int add_cache_entry(struct cache_entry *ce)
{
    int pos;

    pos = cache_name_pos(ce->name, ce->namelen);

```

```
    /* existing match? Just replace it */
    if (pos < 0) {
        active_cache[-pos-1] = ce;
        return 0;
    }

    /* Make sure the array is big enough .. */
    if (active_nr == active_alloc) {
        active_alloc = alloc_nr(active_alloc);
        active_cache = realloc(active_cache, active_alloc * sizeof(struct cache_entry *));
    }

    /* Add it in.. */
    active_nr++;
    if (active_nr > pos)
        memmove(active_cache + pos + 1, active_cache + pos, (active_nr - pos - 1) * sizeof(ce));
    active_cache[pos] = ce;
    return 0;
}

static int index_fd(const char *path, int namelen, struct cache_entry *ce, int fd, struct stat *st)
{
    z_stream stream;
    int max_out_bytes = namelen + st->st_size + 200;
    void *out = malloc(max_out_bytes);
    void *metadata = malloc(namelen + 200);
    void *in = mmap(NULL, st->st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    SHA_CTX c;

    close(fd);
    if (!out || (int)(long)in == -1)
        return -1;

    memset(&stream, 0, sizeof(stream));
    deflateInit(&stream, Z_BEST_COMPRESSION);

    /*
     * ASCII size + nul byte
     */
    stream.next_in = metadata;
    stream.avail_in = 1+sprintf(metadata, "blob %lu", (unsigned long) st->st_size);
    stream.next_out = out;
    stream.avail_out = max_out_bytes;
    while (deflate(&stream, 0) == Z_OK)
        /* nothing */;

    /*
     * File content
     */
    stream.next_in = in;
    stream.avail_in = st->st_size;
    while (deflate(&stream, Z_FINISH) == Z_OK)
        /*nothing */;

    deflateEnd(&stream);

    SHA1_Init(&c);
    SHA1_Update(&c, out, stream.total_out);
```

```
114     SHA1_Final(ce->sha1, &c);
115 
116     return write_sha1_buffer(ce->sha1, out, stream.total_out);
117 }
118 
119 static int add_file_to_cache(char *path)
120 {
121     int size, namelen;
122     struct cache_entry *ce;
123     struct stat st;
124     int fd;
125 
126     fd = open(path, O_RDONLY);
127     if (fd < 0) {
128         if (errno == ENOENT)
129             return remove_file_from_cache(path);
130         return -1;
131     }
132     if (fstat(fd, &st) < 0) {
133         close(fd);
134         return -1;
135     }
136     namelen = strlen(path);
137     size = cache_entry_size(namelen);
138     ce = malloc(size);
139     memset(ce, 0, size);
140     memcpy(ce->name, path, namelen);
141     ce->ctime.sec = st.st_ctime;
142     ce->ctime.nsec = st.st_ctim.tv_nsec;
143     ce->mtime.sec = st.st_mtime;
144     ce->mtime.nsec = st.st_mtim.tv_nsec;
145     ce->st_dev = st.st_dev;
146     ce->st_ino = st.st_ino;
147     ce->st_mode = st.st_mode;
148     ce->st_uid = st.st_uid;
149     ce->st_gid = st.st_gid;
150     ce->st_size = st.st_size;
151     ce->namelen = namelen;
152 
153     if (index_fd(path, namelen, ce, fd, &st) < 0)
154         return -1;
155 
156     return add_cache_entry(ce);
157 }
158 
159 static int write_cache(int newfd, struct cache_entry **cache, int entries)
160 {
161     SHA_CTX c;
162     struct cache_header hdr;
163     int i;
164 
165     hdr.signature = CACHE_SIGNATURE;
166     hdr.version = 1;
167     hdr.entries = entries;
168 
169     SHA1_Init(&c);
170     SHA1_Update(&c, &hdr, offsetof(struct cache_header, sha1));
171     for (i = 0; i < entries; i++) {
```

```
172         struct cache_entry *ce = cache[i];
173         int size = ce_size(ce);
174         SHA1_Update(&c, ce, size);
175     }
176     SHA1_Final(hdr.sha1, &c);
177 
178     if (write(newfd, &hdr, sizeof(hdr)) != sizeof(hdr))
179         return -1;
180 
181     for (i = 0; i < entries; i++) {
182         struct cache_entry *ce = cache[i];
183         int size = ce_size(ce);
184         if (write(newfd, ce, size) != size)
185             return -1;
186     }
187     return 0;
188 }
189 
190 /*
191  * We fundamentally don't like some paths: we don't want
192  * dot or dot-dot anywhere, and in fact, we don't even want
193  * any other dot-files (.dircache or anything else). They
194  * are hidden, for chist sake.
195  *
196  * Also, we don't want double slashes or slashes at the
197  * end that can make pathnames ambiguous.
198  */
199 static int verify_path(char *path)
200 {
201     char c;
202 
203     goto inside;
204     for (;;) {
205         if (!c)
206             return 1;
207         if (c == '/') {
208 inside:
209             c = *path++;
210             if (c != '/' && c != '.' && c != '\0')
211                 continue;
212             return 0;
213         }
214         c = *path++;
215     }
216 }
217 
218 int main(int argc, char **argv)
219 {
220     int i, newfd, entries;
221 
222     entries = read_cache();
223     if (entries < 0) {
224         perror("cache corrupted");
225         return -1;
226     }
227 
228     newfd = open(".dircache/index.lock", O_RDWR | O_CREAT | O_EXCL, 0600);
229     if (newfd < 0) {
```

```
230         perror("unable to create new cachefile");
231         return -1;
232     }
233     for (i = 1 ; i < argc; i++) {
234         char *path = argv[i];
235         if (!verify_path(path)) {
236             fprintf(stderr, "Ignoring path %s\n", argv[i]);
237             continue;
238         }
239         if (add_file_to_cache(path)) {
240             fprintf(stderr, "Unable to add %s to database\n", path);
241             goto out;
242         }
243     }
244     if (!write_cache(newfd, active_cache, active_nr) && !rename(".dircache/index.lock", ".dircache/i
245         return 0;
246 out:
247     unlink(".dircache/index.lock");
248 }
```

6 init-db.c

```c
#include "cache.h"

int main(int argc, char **argv)
{
    char *sha1_dir = getenv(DB_ENVIRONMENT), *path;
    int len, i, fd;

    if (mkdir(".dircache", 0700) < 0) {
        perror("unable to create .dircache");
        exit(1);
    }

    /*
     * If you want to, you can share the DB area with any number of branches.
     * That has advantages: you can save space by sharing all the SHA1 objects.
     * On the other hand, it might just make lookup slower and messier. You
     * be the judge.
     */
    sha1_dir = getenv(DB_ENVIRONMENT);
    if (sha1_dir) {
        struct stat st;
        if (!stat(sha1_dir, &st) < 0 && S_ISDIR(st.st_mode))
            return;
        fprintf(stderr, "DB_ENVIRONMENT set to bad directory %s: ", sha1_dir);
    }

    /*
     * The default case is to have a DB per managed directory.
     */
    sha1_dir = DEFAULT_DB_ENVIRONMENT;
    fprintf(stderr, "defaulting to private storage area\n");
    len = strlen(sha1_dir);
    if (mkdir(sha1_dir, 0700) < 0) {
        if (errno != EEXIST) {
            perror(sha1_dir);
            exit(1);
        }
    }
    path = malloc(len + 40);
    memcpy(path, sha1_dir, len);
    for (i = 0; i < 256; i++) {
        sprintf(path+len, "/%02x", i);
        if (mkdir(path, 0700) < 0) {
            if (errno != EEXIST) {
                perror(path);
                exit(1);
            }
        }
    }
    return 0;
}
```

7 commit-tree.c

```c
#include "cache.h"

#include <pwd.h>
#include <time.h>

#define BLOCKING (1ul << 14)
#define ORIG_OFFSET (40)

/*
 * Leave space at the beginning to insert the tag
 * once we know how big things are.
 *
 * FIXME! Share the code with "write-tree.c"
 */
static void init_buffer(char **bufp, unsigned int *sizep)
{
    char *buf = malloc(BLOCKING);
    memset(buf, 0, ORIG_OFFSET);
    *sizep = ORIG_OFFSET;
    *bufp = buf;
}

static void add_buffer(char **bufp, unsigned int *sizep, const char *fmt, ...)
{
    char one_line[2048];
    va_list args;
    int len;
    unsigned long alloc, size, newsize;
    char *buf;

    va_start(args, fmt);
    len = vsnprintf(one_line, sizeof(one_line), fmt, args);
    va_end(args);
    size = *sizep;
    newsize = size + len;
    alloc = (size + 32767) & ~32767;
    buf = *bufp;
    if (newsize > alloc) {
        alloc = (newsize + 32767) & ~32767;
        buf = realloc(buf, alloc);
        *bufp = buf;
    }
    *sizep = newsize;
    memcpy(buf + size, one_line, len);
}

static int prepend_integer(char *buffer, unsigned val, int i)
{
    buffer[--i] = '\0';
    do {
        buffer[--i] = '0' + (val % 10);
        val /= 10;
    } while (val);
    return i;
}
```

```
56
57 static void finish_buffer(char *tag, char **bufp, unsigned int *sizep)
58 {
59     int taglen;
60     int offset;
61     char *buf = *bufp;
62     unsigned int size = *sizep;
63
64     offset = prepend_integer(buf, size - ORIG_OFFSET, ORIG_OFFSET);
65     taglen = strlen(tag);
66     offset -= taglen;
67     buf += offset;
68     size -= offset;
69     memcpy(buf, tag, taglen);
70
71     *bufp = buf;
72     *sizep = size;
73 }
74
75 static void remove_special(char *p)
76 {
77     char c;
78     char *dst = p;
79
80     for (;;) {
81         c = *p;
82         p++;
83         switch(c) {
84         case '\n': case '<': case '>':
85             continue;
86         }
87         *dst++ = c;
88         if (!c)
89             break;
90     }
91 }
92
93 /*
94  * Having more than two parents may be strange, but hey, there's
95  * no conceptual reason why the file format couldn't accept multi-way
96  * merges. It might be the "union" of several packages, for example.
97  *
98  * I don't really expect that to happen, but this is here to make
99  * it clear that _conceptually_ it's ok..
100  */
101 #define MAXPARENT (16)
102
103 int main(int argc, char **argv)
104 {
105     int i, len;
106     int parents = 0;
107     unsigned char tree_sha1[20];
108     unsigned char parent_sha1[MAXPARENT][20];
109     char *gecos, *realgecos;
110     char *email, realemail[1000];
111     char *date, *realdate;
112     char comment[1000];
113     struct passwd *pw;
```

```
114     time_t now;
115     char *buffer;
116     unsigned int size;
117 
118     if (argc < 2 || get_sha1_hex(argv[1], tree_sha1) < 0)
119         usage("commit-tree <sha1> [-p <sha1>]* < changelog");
120 
121     for (i = 2; i < argc; i += 2) {
122         char *a, *b;
123         a = argv[i]; b = argv[i+1];
124         if (!b || strcmp(a, "-p") || get_sha1_hex(b, parent_sha1[parents]))
125             usage("commit-tree <sha1> [-p <sha1>]* < changelog");
126         parents++;
127     }
128     if (!parents)
129         fprintf(stderr, "Committing initial tree %s\n", argv[1]);
130     pw = getpwuid(getuid());
131     if (!pw)
132         usage("You don't exist. Go away!");
133     realgecos = pw->pw_gecos;
134     len = strlen(pw->pw_name);
135     memcpy(realemail, pw->pw_name, len);
136     realemail[len] = '@';
137     gethostname(realemail+len+1, sizeof(realemail)-len-1);
138     time(&now);
139     realdate = ctime(&now);
140 
141     gecos = getenv("COMMITTER_NAME") ? : realgecos;
142     email = getenv("COMMITTER_EMAIL") ? : realemail;
143     date = getenv("COMMITTER_DATE") ? : realdate;
144 
145     remove_special(gecos); remove_special(realgecos);
146     remove_special(email); remove_special(realemail);
147     remove_special(date); remove_special(realdate);
148 
149     init_buffer(&buffer, &size);
150     add_buffer(&buffer, &size, "tree %s\n", sha1_to_hex(tree_sha1));
151 
152     /*
153      * NOTE! This ordering means that the same exact tree merged with a
154      * different order of parents will be a _different_ changeset even
155      * if everything else stays the same.
156      */
157     for (i = 0; i < parents; i++)
158         add_buffer(&buffer, &size, "parent %s\n", sha1_to_hex(parent_sha1[i]));
159 
160     /* Person/date information */
161     add_buffer(&buffer, &size, "author %s <%s> %s\n", gecos, email, date);
162     add_buffer(&buffer, &size, "committer %s <%s> %s\n\n", realgecos, realemail, realdate);
163 
164     /* And add the comment */
165     while (fgets(comment, sizeof(comment), stdin) != NULL)
166         add_buffer(&buffer, &size, "%s", comment);
167 
168     finish_buffer("commit ", &buffer, &size);
169 
170     write_sha1_file(buffer, size);
171     return 0;
```

```
172 }
```

8 read-tree.c

```c
#include "cache.h"

static int unpack(unsigned char *sha1)
{
    void *buffer;
    unsigned long size;
    char type[20];

    buffer = read_sha1_file(sha1, type, &size);
    if (!buffer)
        usage("unable to read sha1 file");
    if (strcmp(type, "tree"))
        usage("expected a 'tree' node");
    while (size) {
        int len = strlen(buffer)+1;
        unsigned char *sha1 = buffer + len;
        char *path = strchr(buffer, ' ')+1;
        unsigned int mode;
        if (size < len + 20 || sscanf(buffer, "%o", &mode) != 1)
            usage("corrupt 'tree' file");
        buffer = sha1 + 20;
        size -= len + 20;
        printf("%o %s (%s)\n", mode, path, sha1_to_hex(sha1));
    }
    return 0;
}

int main(int argc, char **argv)
{
    int fd;
    unsigned char sha1[20];

    if (argc != 2)
        usage("read-tree <key>");
    if (get_sha1_hex(argv[1], sha1) < 0)
        usage("read-tree <key>");
    sha1_file_directory = getenv(DB_ENVIRONMENT);
    if (!sha1_file_directory)
        sha1_file_directory = DEFAULT_DB_ENVIRONMENT;
    if (unpack(sha1) < 0)
        usage("unpack failed");
    return 0;
}
```

9 write-tree.c

```c
#include "cache.h"

static int check_valid_sha1(unsigned char *sha1)
{
    char *filename = sha1_file_name(sha1);
    int ret;

    /* If we were anal, we'd check that the sha1 of the contents actually matches */
    ret = access(filename, R_OK);
    if (ret)
        perror(filename);
    return ret;
}

static int prepend_integer(char *buffer, unsigned val, int i)
{
    buffer[--i] = '\0';
    do {
        buffer[--i] = '0' + (val % 10);
        val /= 10;
    } while (val);
    return i;
}

#define ORIG_OFFSET (40)    /* Enough space to add the header of "tree <size>\0" */

int main(int argc, char **argv)
{
    unsigned long size, offset, val;
    int i, entries = read_cache();
    char *buffer;

    if (entries <= 0) {
        fprintf(stderr, "No file-cache to create a tree of\n");
        exit(1);
    }

    /* Guess at an initial size */
    size = entries * 40 + 400;
    buffer = malloc(size);
    offset = ORIG_OFFSET;

    for (i = 0; i < entries; i++) {
        struct cache_entry *ce = active_cache[i];
        if (check_valid_sha1(ce->sha1) < 0)
            exit(1);
        if (offset + ce->namelen + 60 > size) {
            size = alloc_nr(offset + ce->namelen + 60);
            buffer = realloc(buffer, size);
        }
        offset += sprintf(buffer + offset, "%o %s", ce->st_mode, ce->name);
        buffer[offset++] = 0;
        memcpy(buffer + offset, ce->sha1, 20);
        offset += 20;
    }
```

```
56
57      i = prepend_integer(buffer, offset - ORIG_OFFSET, ORIG_OFFSET);
58      i -= 5;
59      memcpy(buffer+i, "tree ", 5);
60
61      buffer += i;
62      offset -= i;
63
64      write_sha1_file(buffer, offset);
65      return 0;
66  }
```

10 cat-file.c

```c
#include "cache.h"

int main(int argc, char **argv)
{
    unsigned char sha1[20];
    char type[20];
    void *buf;
    unsigned long size;
    char template[] = "temp_git_file_XXXXXX";
    int fd;

    if (argc != 2 || get_sha1_hex(argv[1], sha1))
        usage("cat-file: cat-file <sha1>");
    buf = read_sha1_file(sha1, type, &size);
    if (!buf)
        exit(1);
    fd = mkstemp(template);
    if (fd < 0)
        usage("unable to create tempfile");
    if (write(fd, buf, size) != size)
        strcpy(type, "bad");
    printf("%s: %s\n", template, type);
}
```

11 show-diff.c

```c
#include "cache.h"

#define MTIME_CHANGED	0x0001
#define CTIME_CHANGED	0x0002
#define OWNER_CHANGED	0x0004
#define MODE_CHANGED	0x0008
#define INODE_CHANGED	0x0010
#define DATA_CHANGED	0x0020

static int match_stat(struct cache_entry *ce, struct stat *st)
{
	unsigned int changed = 0;

	if (ce->mtime.sec  != (unsigned int)st->st_mtim.tv_sec ||
	    ce->mtime.nsec != (unsigned int)st->st_mtim.tv_nsec)
	    changed |= MTIME_CHANGED;
	if (ce->ctime.sec  != (unsigned int)st->st_ctim.tv_sec ||
	    ce->ctime.nsec != (unsigned int)st->st_ctim.tv_nsec)
	    changed |= CTIME_CHANGED;
	if (ce->st_uid != (unsigned int)st->st_uid ||
	    ce->st_gid != (unsigned int)st->st_gid)
	    changed |= OWNER_CHANGED;
	if (ce->st_mode != (unsigned int)st->st_mode)
	    changed |= MODE_CHANGED;
	if (ce->st_dev != (unsigned int)st->st_dev ||
	    ce->st_ino != (unsigned int)st->st_ino)
	    changed |= INODE_CHANGED;
	if (ce->st_size != (unsigned int)st->st_size)
	    changed |= DATA_CHANGED;
	return changed;
}

static void show_differences(struct cache_entry *ce, struct stat *cur,
	void *old_contents, unsigned long long old_size)
{
	static char cmd[1000];
	FILE *f;

	snprintf(cmd, sizeof(cmd), "diff -u - %s", ce->name);
	f = popen(cmd, "w");
	fwrite(old_contents, old_size, 1, f);
	pclose(f);
}

int main(int argc, char **argv)
{
	int entries = read_cache();
	int i;

	if (entries < 0) {
	    perror("read_cache");
	    exit(1);
	}
	for (i = 0; i < entries; i++) {
	    struct stat st;
```

```
56          struct cache_entry *ce = active_cache[i];
57          int n, changed;
58          unsigned int mode;
59          unsigned long size;
60          char type[20];
61          void *new;
62
63          if (stat(ce->name, &st) < 0) {
64              printf("%s: %s\n", ce->name, strerror(errno));
65              continue;
66          }
67          changed = match_stat(ce, &st);
68          if (!changed) {
69              printf("%s: ok\n", ce->name);
70              continue;
71          }
72          printf("%.*s:  ", ce->namelen, ce->name);
73          for (n = 0; n < 20; n++)
74              printf("%02x", ce->sha1[n]);
75          printf("\n");
76          new = read_sha1_file(ce->sha1, type, &size);
77          show_differences(ce, &st, new, size);
78          free(new);
79      }
80      return 0;
81  }
```