Analizador Sintáctico

Ariana Bermúdez,Ximena Bolaños, Dylan Rodríguez

Instituto Tecnológico de Costa Rica

May 31, 2017

Análisis Sintáctico

Se hizo un analizador sintáctico con la ayuda de la herramienta de Bison, para el lenguaje C y que corre en C, este analizador trabaja en conjunto con Flex, para tomar los tokens que este le otorga y revisar con las gramáticas que les sean ingresadas.

Bison

Bison convierte de una gramática libre de contexto a un analizador sintáctico que emplea las tablas de Parsing LALR(1), siendo:

- L: Left algo
- A: ...
- L: ...
- R: rightmost
- (1): donde este uno significa que tiene como lookahead solo un símbolo.

Cabe destacar que Bison es compatible con Yacc. Sirve con C, C++ y Java.

Código

```
struct ref_lock {
char * ref_name ;
struct lock_file * lk
struct object_id old_oid ;
} ;
static int ref_resolves_to_object ( const char * refname ,
const struct object_id * oid ,
unsigned int flags )
{
if ( flags & REF_ISBROKEN )
return 0 ;
if ( ! has_sha1_file ( oid -> hash ) ) {
```

Código

```
error ( "%s does not point to a valid object!" , refname ) ;
return 0 ;
}
return 1 ;
struct packed_ref_cache {
struct ref_cache * cache ;
unsigned int referrers ;
struct lock_file * lock ;
struct stat_validity validity ;
struct files_ref_store {
struct ref_store base ;
unsigned int store_flags
char * gitdir ;
```

Código

```
char * gitcommondir ;
char * packed_refs_path ;
struct ref_cache * loose ;
struct packed_ref_cache * packed ;
} ;
static struct lock_file packlock ;
static void acquire_packed_ref_cache ( struct packed_ref_cache * packed_refs )
{
packed_refs -> referrers ++ ;
static int release_packed_ref_cache ( struct packed_ref_cache * packed_refs )
if ( ! -- packed_refs -> referrers ) {
free_ref_cache ( packed_refs -> cache ) ;
stat_validity_clear ( & packed_refs -> validity ) ;
```

Código

```
free ( packed_refs ) ;
return 1 ;
} else {
return 0 ;
}
}
static void clear_packed_ref_cache ( struct files_ref_store * refs )
{
if ( refs -> packed ) {
struct packed_ref_cache * packed_refs = refs -> packed ;
if ( packed_refs -> lock )
die ( "internal error: packed-ref cache cleared while locked" ) ;
refs -> packed = NULL ;
```

Código

```
release_packed_ref_cache ( packed_refs ) ;
}
}
static void clear_loose_ref_cache ( struct files_ref_store * refs )
{
if ( refs -> loose ) {
free_ref_cache ( refs -> loose ) ;
refs -> loose = NULL ;
}
}
static struct ref_store * files_ref_store_create ( const char * gitdir ,
unsigned int flags )
{
```

Código

```
struct files_ref_store * refs = xcalloc ( 1 , sizeof ( * refs ) ) ;
struct ref_store * ref_store = ( struct ref_store * ) refs ;
struct strbuf sb = STRBUF_INIT ;
base_ref_store_init ( ref_store , & refs_be_files ) ;
refs -> store_flags = flags ;
refs -> gitdir = xstrdup ( gitdir ) ;
get_common_dir_noenv ( & sb , gitdir ) ;
refs -> gitcommondir = strbuf_detach ( & sb , NULL ) ;
strbuf_addf ( & sb , "%s/packed-refs" , refs -> gitcommondir ) ;
refs -> packed_refs_path = strbuf_detach ( & sb , NULL ) ;
return ref_store ;
}
static void files_assert_main_repository ( struct files_ref_store * refs ,
```

Código

```
const char * caller )
{
if ( refs -> store_flags & REF_STORE_MAIN )
return ;
die ( "BUG: operation %s only allowed for main ref store" , caller ) ;
}
static struct files_ref_store * files_downcast ( struct ref_store * ref_store ,
unsigned int required_flags ,
const char * caller )
{
struct files_ref_store * refs ;
if ( ref_store -> be != & refs_be_files )
die ( "BUG: ref_store is type \"%s\" not \"files\" in %s" ,
```

Código

```
ref_store -> be -> name , caller ) ;
refs = ( struct files_ref_store * ) ref_store ;
if ( ( refs -> store_flags & required_flags ) != required_flags )
die ( "BUG: operation %s requires abilities 0x%x, but only have 0x%x" ,
caller , required_flags , refs -> store_flags ) ;
return refs ;
}
static const char PACKED_REFS_HEADER [ ] =
"# pack-refs with: peeled fully-peeled \n" ;
static const char * parse_ref_line ( struct strbuf * line , struct object_id * oid )
{
const char * ref ;
if ( parse_oid_hex ( line -> buf , oid , & ref ) < 0 )
```

Código

```
return NULL ;
if ( ! isspace ( * ref ++ ) )
return NULL ;
if ( isspace ( * ref ) )
return NULL ;
if ( line -> buf [ line -> len - 1 ] != '\n' )
return NULL ;
line -> buf [ -- line -> len ] = 0 ;
return ref ;
}
static void read_packed_refs ( FILE * f , struct ref_dir * dir )
{
struct ref_entry * last = NULL ;
```

Código

```
struct strbuf line = STRBUF_INIT ;
enum { PEELED_NONE , PEELED_TAGS , PEELED_FULLY } peeled = PEELED_NONE ;
while ( strbuf_getwholeline ( & line , f , '\n' ) != EOF ) {
struct object_id oid ;
const char * refname ;
const char * traits ;
if ( skip_prefix ( line . buf , "# pack-refs with:" , & traits ) ) {
if ( strstr ( traits , " fully-peeled " ) )
peeled = PEELED_FULLY ;
else if ( strstr ( traits , " peeled " ) )
peeled = PEELED_TAGS ;
continue ;
}
```

Código

```
refname = parse_ref_line ( & line , & oid ) ;
if ( refname ) {
int flag = REF_ISPACKED ;
if ( check_refname_format ( refname , REFNAME_ALLOW_ONELEVEL ) ) {
if ( ! refname_is_safe ( refname ) )
die ( "packed refname is dangerous: %s" , refname ) ;
oidclr ( & oid ) ;
flag |= REF_BAD_NAME | REF_ISBROKEN ;
}
last = create_ref_entry ( refname , & oid , flag , 0 ) ;
if ( peeled == PEELED_FULLY ||
( peeled == PEELED_TAGS && starts_with ( refname , "refs/tags/" ) ) )
last -> flag |= REF_KNOWS_PEELED ;
```

Código

```
add_ref_entry ( dir , last ) ;
continue ;
}
if ( last &&
line . buf [ 0 ] == '^' &&
line . len == 42 &&
line . buf [ 42 - 1 ] == '\n' &&
! get_oid_hex ( line . buf + 1 , & oid ) ) {
oidcpy ( & last -> u . value . peeled , & oid ) ;
last -> flag |= REF_KNOWS_PEELED ;
}
}
strbuf_release ( & line ) ;
```

Código

```
}
static const char * files_packed_refs_path ( struct files_ref_store * refs )
{
return refs -> packed_refs_path ;
}
static void files_reflog_path ( struct files_ref_store * refs ,
struct strbuf * sb ,
const char * refname )
{
if ( ! refname ) {
strbuf_addf ( sb , "%s/logs" , refs -> gitcommondir ) ;
return ;
}
```

Código

```
switch ( ref_type ( refname ) ) {
case REF_TYPE_PER_WORKTREE :
case REF_TYPE_PSEUDOREF :
strbuf_addf ( sb , "%s/logs/%s" , refs -> gitdir , refname ) ;
break ;
case REF_TYPE_NORMAL :
strbuf_addf ( sb , "%s/logs/%s" , refs -> gitcommondir , refname ) ;
break ;
default :
die ( "BUG: unknown ref type %d of ref %s" ,
ref_type ( refname ) , refname ) ;
}
}
```

Código

```
static void files_ref_path ( struct files_ref_store * refs ,
struct strbuf * sb ,
const char * refname )
{
switch ( ref_type ( refname ) ) {
case REF_TYPE_PER_WORKTREE :
case REF_TYPE_PSEUDOREF :
strbuf_addf ( sb , "%s/%s" , refs -> gitdir , refname ) ;
break ;
case REF_TYPE_NORMAL :
strbuf_addf ( sb , "%s/%s" , refs -> gitcommondir , refname ) ;
break ;
default :
```

Código

```
die ( "BUG: unknown ref type %d of ref %s" ,
ref_type ( refname ) , refname ) ;
}
}
static struct packed_ref_cache * get_packed_ref_cache ( struct files_ref_store * refs )
{
const char * packed_refs_file = files_packed_refs_path ( refs ) ;
if ( refs -> packed &&
! stat_validity_check ( & refs -> packed -> validity , packed_refs_file ) )
clear_packed_ref_cache ( refs ) ;
if ( ! refs -> packed ) {
FILE * f ;
refs -> packed = xcalloc ( 1 , sizeof ( * refs -> packed ) ) ;
```

Código

```
acquire_packed_ref_cache ( refs -> packed ) ;
refs -> packed -> cache = create_ref_cache ( & refs -> base , NULL ) ;
refs -> packed -> cache -> root -> flag &= ~ REF_INCOMPLETE ;
f = fopen ( packed_refs_file , "r" ) ;
if ( f ) {
stat_validity_update ( & refs -> packed -> validity , fileno ( f ) ) ;
read_packed_refs ( f , get_ref_dir ( refs -> packed -> cache -> root ) ) ;
fclose ( f ) ;
}
}
return refs -> packed ;
}
static struct ref_dir * get_packed_ref_dir ( struct packed_ref_cache * packed_ref_cache )
```

Código

```
{
return get_ref_dir ( packed_ref_cache -> cache -> root ) ;
}
static struct ref_dir * get_packed_refs ( struct files_ref_store * refs )
{
return get_packed_ref_dir ( get_packed_ref_cache ( refs ) ) ;
}
static void add_packed_ref ( struct files_ref_store * refs ,
const char * refname , const struct object_id * oid )
{
struct packed_ref_cache * packed_ref_cache = get_packed_ref_cache ( refs ) ;
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed refs not locked" ) ;
```

Código

```
add_ref_entry ( get_packed_ref_dir ( packed_ref_cache ) ,
create_ref_entry ( refname , oid , REF_ISPACKED , 1 ) ) ;
}
static void loose_fill_ref_dir ( struct ref_store * ref_store ,
struct ref_dir * dir , const char * dirname )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "fill_ref_dir" ) ;
DIR * d ;
struct dirent * de ;
int dirnamelen = strlen ( dirname ) ;
struct strbuf refname ;
struct strbuf path = STRBUF_INIT ;
```

Código

```
size_t path_baselen ;
files_ref_path ( refs , & path , dirname ) ;
path_baselen = path . len ;
d = opendir ( path . buf ) ;
if ( ! d ) {
strbuf_release ( & path ) ;
return ;
}
strbuf_init ( & refname , dirnamelen + 257 ) ;
strbuf_add ( & refname , dirname , dirnamelen ) ;
while ( ( de = readdir ( d ) ) != NULL ) {
struct object_id oid ;
struct stat st ;
```

Código

```
int flag ;
if ( de -> d_name [ 0 ] == '.' )
continue ;
if ( ends_with ( de -> d_name , ".lock" ) )
continue ;
strbuf_addstr ( & refname , de -> d_name ) ;
strbuf_addstr ( & path , de -> d_name ) ;
if ( stat ( path . buf , & st ) < 0 ) {
;
} else if ( S_ISDIR ( st . st_mode ) ) {
strbuf_addch ( & refname , '/' ) ;
add_entry_to_dir ( dir ,
create_dir_entry ( dir -> cache , refname . buf ,
```

Código

```
refname . len , 1 ) ) ;
} else {
if ( ! refs_resolve_ref_unsafe ( & refs -> base ,
refname . buf ,
RESOLVE_REF_READING ,
oid . hash , & flag ) ) {
oidclr ( & oid ) ;
flag |= REF_ISBROKEN ;
} else if ( is_null_oid ( & oid ) ) {
flag |= REF_ISBROKEN ;
}
if ( check_refname_format ( refname . buf ,
REFNAME_ALLOW_ONELEVEL ) ) {
```

Código

```
if ( ! refname_is_safe ( refname . buf ) )
die ( "loose refname is dangerous: %s" , refname . buf ) ;
oidclr ( & oid ) ;
flag |= REF_BAD_NAME | REF_ISBROKEN ;
}
add_entry_to_dir ( dir ,
create_ref_entry ( refname . buf , & oid , flag , 0 ) ) ;
}
strbuf_setlen ( & refname , dirnamelen ) ;
strbuf_setlen ( & path , path_baselen ) ;
}
strbuf_release ( & refname ) ;
strbuf_release ( & path ) ;
```

Código

```
closedir ( d ) ;
if ( ! strcmp ( dirname , "refs/" ) ) {
int pos = search_ref_dir ( dir , "refs/bisect/" , 12 ) ;
if ( pos < 0 ) {
struct ref_entry * child_entry = create_dir_entry (
dir -> cache , "refs/bisect/" , 12 , 1 ) ;
add_entry_to_dir ( dir , child_entry ) ;
}
}
}
static struct ref_cache * get_loose_ref_cache ( struct files_ref_store * refs )
{
if ( ! refs -> loose ) {
```

Código

```
refs -> loose = create_ref_cache ( & refs -> base , loose_fill_ref_dir ) ;
refs -> loose -> root -> flag &= ~ REF_INCOMPLETE ;
add_entry_to_dir ( get_ref_dir ( refs -> loose -> root ) ,
create_dir_entry ( refs -> loose , "refs/" , 5 , 1 ) ) ;
}
return refs -> loose ;
}
static struct ref_entry * get_packed_ref ( struct files_ref_store * refs ,
const char * refname )
{
return find_ref_entry ( get_packed_refs ( refs ) , refname ) ;
}
static int resolve_packed_ref ( struct files_ref_store * refs ,
```

Código

```
const char * refname ,
unsigned char * sha1 , unsigned int * flags )
{
struct ref_entry * entry ;
entry = get_packed_ref ( refs , refname ) ;
if ( entry ) {
hashcpy ( sha1 , entry -> u . value . oid . hash ) ;
* flags |= REF_ISPACKED ;
return 0 ;
}
return - 1 ;
}
static int files_read_raw_ref ( struct ref_store * ref_store ,
```

Código

```
const char * refname , unsigned char * sha1 ,
struct strbuf * referent , unsigned int * type )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "read_raw_ref" ) ;
struct strbuf sb_contents = STRBUF_INIT ;
struct strbuf sb_path = STRBUF_INIT ;
const char * path ;
const char * buf ;
struct stat st ;
int fd ;
int ret = - 1 ;
int save_errno ;
```

Código

```
int remaining_retries = 3 ;
* type = 0 ;
strbuf_reset ( & sb_path ) ;
files_ref_path ( refs , & sb_path , refname ) ;
path = sb_path . buf ;
stat_ref :
if ( remaining_retries -- <= 0 )
goto out ;
if ( lstat ( path , & st ) < 0 ) {
if ( errno != ENOENT )
goto out ;
if ( resolve_packed_ref ( refs , refname , sha1 , type ) ) {
errno = ENOENT ;
```

Código

```
goto out ;
}
ret = 0 ;
goto out ;
}
if ( S_ISLNK ( st . st_mode ) ) {
strbuf_reset ( & sb_contents ) ;
if ( strbuf_readlink ( & sb_contents , path , 0 ) < 0 ) {
if ( errno == ENOENT || errno == EINVAL )
goto stat_ref ;
else
goto out ;
}
```

Código

```
if ( starts_with ( sb_contents . buf , "refs/" ) &&
! check_refname_format ( sb_contents . buf , 0 ) ) {
strbuf_swap ( & sb_contents , referent ) ;
* type |= REF_ISSYMREF ;
ret = 0 ;
goto out ;
}
}
if ( S_ISDIR ( st . st_mode ) ) {
if ( resolve_packed_ref ( refs , refname , sha1 , type ) ) {
errno = EISDIR ;
goto out ;
}
```

Código

```
ret = 0 ;
goto out ;
}
fd = open ( path , O_RDONLY ) ;
if ( fd < 0 ) {
if ( errno == ENOENT && ! S_ISLNK ( st . st_mode ) )
goto stat_ref ;
else
goto out ;
}
strbuf_reset ( & sb_contents ) ;
if ( strbuf_read ( & sb_contents , fd , 256 ) < 0 ) {
int save_errno = errno ;
```

Código

```
close ( fd ) ;
errno = save_errno ;
goto out ;
}
close ( fd ) ;
strbuf_rtrim ( & sb_contents ) ;
buf = sb_contents . buf ;
if ( starts_with ( buf , "ref:" ) ) {
buf += 4 ;
while ( isspace ( * buf ) )
buf ++ ;
strbuf_reset ( referent ) ;
strbuf_addstr ( referent , buf ) ;
```

Código

```
* type |= REF_ISSYMREF ;
ret = 0 ;
goto out ;
}
if ( get_sha1_hex ( buf , sha1 ) ||
( buf [ 40 ] != '\0' && ! isspace ( buf [ 40 ] ) ) ) {
* type |= REF_ISBROKEN ;
errno = EINVAL ;
goto out ;
}
ret = 0 ;
out :
save_errno = errno ;
```

Código

```
strbuf_release ( & sb_path ) ;
strbuf_release ( & sb_contents ) ;
errno = save_errno ;
return ret ;
}
static void unlock_ref ( struct ref_lock * lock )
{
if ( lock -> lk )
rollback_lock_file ( lock -> lk ) ;
free ( lock -> ref_name ) ;
free ( lock ) ;
}
static int lock_raw_ref ( struct files_ref_store * refs ,
```

Código

```
const char * refname , int mustexist ,
const struct string_list * extras ,
const struct string_list * skip ,
struct ref_lock * * lock_p ,
struct strbuf * referent ,
unsigned int * type ,
struct strbuf * err )
{
struct ref_lock * lock ;
struct strbuf ref_file = STRBUF_INIT ;
int attempts_remaining = 3 ;
int ret = TRANSACTION_GENERIC_ERROR ;
assert ( err ) ;
```

Código

```
files_assert_main_repository ( refs , "lock_raw_ref" ) ;
* type = 0 ;
* lock_p = lock = xcalloc ( 1 , sizeof ( * lock ) ) ;
lock -> ref_name = xstrdup ( refname ) ;
files_ref_path ( refs , & ref_file , refname ) ;
retry :
switch ( safe_create_leading_directories ( ref_file . buf ) ) {
case SCLD_OK :
break ;
case SCLD_EXISTS :
if ( refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) ) {
if ( mustexist ) {
```

Código

```
strbuf_reset ( err ) ;
strbuf_addf ( err , "unable to resolve reference ’%s’" ,
refname ) ;
} else {
ret = TRANSACTION_NAME_CONFLICT ;
}
} else {
strbuf_addf ( err , "unable to create lock file %s.lock; "
"non-directory in the way" ,
ref_file . buf ) ;
}
goto error_return ;
case SCLD_VANISHED :
```

Código

```
if ( -- attempts_remaining > 0 )
goto retry ;
default :
strbuf_addf ( err , "unable to create directory for %s" ,
ref_file . buf ) ;
goto error_return ;
}
if ( ! lock -> lk )
lock -> lk = xcalloc ( 1 , sizeof ( struct lock_file ) ) ;
if ( hold_lock_file_for_update ( lock -> lk , ref_file . buf , LOCK_NO_DEREF ) < 0 ) {
if ( errno == ENOENT && -- attempts_remaining > 0 ) {
goto retry ;
} else {
```

Código

```
unable_to_lock_message ( ref_file . buf , errno , err ) ;
goto error_return ;
}
}
if ( files_read_raw_ref ( & refs -> base , refname ,
lock -> old_oid . hash , referent , type ) ) {
if ( errno == ENOENT ) {
if ( mustexist ) {
strbuf_addf ( err , "unable to resolve reference '%s'" ,
refname ) ;
goto error_return ;
} else {
}
```

Código

```
} else if ( errno == EISDIR ) {
if ( mustexist ) {
strbuf_addf ( err , "unable to resolve reference '%s'" ,
refname ) ;
goto error_return ;
} else if ( remove_dir_recursively ( & ref_file ,
REMOVE_DIR_EMPTY_ONLY ) ) {
if ( refs_verify_refname_available (
& refs -> base , refname ,
extras , skip , err ) ) {
ret = TRANSACTION_NAME_CONFLICT ;
goto error_return ;
} else {
```

Código

```
strbuf_addf ( err , "there is a non-empty directory ’%s’ "
"blocking reference ’%s’" ,
ref_file . buf , refname ) ;
goto error_return ;
}
}
} else if ( errno == EINVAL && ( * type & REF_ISBROKEN ) ) {
strbuf_addf ( err , "unable to resolve reference ’%s’: "
"reference broken" , refname ) ;
goto error_return ;
} else {
strbuf_addf ( err , "unable to resolve reference ’%s’: %s" ,
refname , strerror ( errno ) ) ;
```

Código

```
goto error_return ;
}
if ( refs_verify_refname_available (
& refs -> base , refname ,
extras , skip , err ) )
goto error_return ;
}
ret = 0 ;
goto out ;
error_return :
unlock_ref ( lock ) ;
* lock_p = NULL ;
out :
```

Código

```
strbuf_release ( & ref_file ) ;
return ret ;
}
static int files_peel_ref ( struct ref_store * ref_store ,
const char * refname , unsigned char * sha1 )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ | REF_STORE_ODB ,
"peel_ref" ) ;
int flag ;
unsigned char base [ 20 ] ;
if ( current_ref_iter && current_ref_iter -> refname == refname ) {
struct object_id peeled ;
```

Código

```
if ( ref_iterator_peel ( current_ref_iter , & peeled ) )
return - 1 ;
hashcpy ( sha1 , peeled . hash ) ;
return 0 ;
}
if ( refs_read_ref_full ( ref_store , refname ,
RESOLVE_REF_READING , base , & flag ) )
return - 1 ;
if ( flag & REF_ISPACKED ) {
struct ref_entry * r = get_packed_ref ( refs , refname ) ;
if ( r ) {
if ( peel_entry ( r , 0 ) )
return - 1 ;
```

Código

```
hashcpy ( sha1 , r -> u . value . peeled . hash ) ;
return 0 ;
}
}
return peel_object ( base , sha1 ) ;
}
struct files_ref_iterator {
struct ref_iterator base ;
struct packed_ref_cache * packed_ref_cache ;
struct ref_iterator * iter0 ;
unsigned int flags ;
} ;
static int files_ref_iterator_advance ( struct ref_iterator * ref_iterator )
```

Código

```
{
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
int ok ;
while ( ( ok = ref_iterator_advance ( iter -> iter0 ) ) == ITER_OK ) {
if ( iter -> flags & DO_FOR_EACH_PER_WORKTREE_ONLY &&
ref_type ( iter -> iter0 -> refname ) != REF_TYPE_PER_WORKTREE )
continue ;
if ( ! ( iter -> flags & DO_FOR_EACH_INCLUDE_BROKEN ) &&
! ref_resolves_to_object ( iter -> iter0 -> refname ,
iter -> iter0 -> oid ,
iter -> iter0 -> flags ) )
continue ;
```

Código

```
iter -> base . refname = iter -> iter0 -> refname ;
iter -> base . oid = iter -> iter0 -> oid ;
iter -> base . flags = iter -> iter0 -> flags ;
return ITER_OK ;
}
iter -> iter0 = NULL ;
if ( ref_iterator_abort ( ref_iterator ) != ITER_DONE )
ok = ITER_ERROR ;
return ok ;
}
static int files_ref_iterator_peel ( struct ref_iterator * ref_iterator ,
struct object_id * peeled )
{
```

Código

```
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
return ref_iterator_peel ( iter -> iter0 , peeled ) ;
}
static int files_ref_iterator_abort ( struct ref_iterator * ref_iterator )
{
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
int ok = ITER_DONE ;
if ( iter -> iter0 )
ok = ref_iterator_abort ( iter -> iter0 ) ;
release_packed_ref_cache ( iter -> packed_ref_cache ) ;
base_ref_iterator_free ( ref_iterator ) ;
```

Código

```
return ok ;
}
static struct ref_iterator_vtable files_ref_iterator_vtable = {
files_ref_iterator_advance ,
files_ref_iterator_peel ,
files_ref_iterator_abort
} ;
static struct ref_iterator * files_ref_iterator_begin (
struct ref_store * ref_store ,
const char * prefix , unsigned int flags )
{
struct files_ref_store * refs ;
struct ref_iterator * loose_iter , * packed_iter ;
```

Código

```
struct files_ref_iterator * iter ;
struct ref_iterator * ref_iterator ;
if ( ref_paranoia < 0 )
ref_paranoia = git_env_bool ( "GIT_REF_PARANOIA" , 0 ) ;
if ( ref_paranoia )
flags |= DO_FOR_EACH_INCLUDE_BROKEN ;
refs = files_downcast ( ref_store ,
REF_STORE_READ | ( ref_paranoia ? 0 : REF_STORE_ODB ) ,
"ref_iterator_begin" ) ;
iter = xcalloc ( 1 , sizeof ( * iter ) ) ;
ref_iterator = & iter -> base ;
base_ref_iterator_init ( ref_iterator , & files_ref_iterator_vtable ) ;
loose_iter = cache_ref_iterator_begin ( get_loose_ref_cache ( refs ) ,
```

Código

```
prefix , 1 ) ;
iter -> packed_ref_cache = get_packed_ref_cache ( refs ) ;
acquire_packed_ref_cache ( iter -> packed_ref_cache ) ;
packed_iter = cache_ref_iterator_begin ( iter -> packed_ref_cache -> cache ,
prefix , 0 ) ;
iter -> iter0 = overlay_ref_iterator_begin ( loose_iter , packed_iter ) ;
iter -> flags = flags ;
return ref_iterator ;
}
static int verify_lock ( struct ref_store * ref_store , struct ref_lock * lock ,
const unsigned char * old_sha1 , int mustexist ,
struct strbuf * err )
{
```

Código

```
assert ( err ) ;
if ( refs_read_ref_full ( ref_store , lock -> ref_name ,
mustexist ? RESOLVE_REF_READING : 0 ,
lock -> old_oid . hash , NULL ) ) {
if ( old_sha1 ) {
int save_errno = errno ;
strbuf_addf ( err , "can't verify ref '%s'" , lock -> ref_name ) ;
errno = save_errno ;
return - 1 ;
} else {
oidclr ( & lock -> old_oid ) ;
return 0 ;
}
```

Código

```
}
if ( old_sha1 && hashcmp ( lock -> old_oid . hash , old_sha1 ) ) {
strbuf_addf ( err , "ref ’%s’ is at %s but expected %s" ,
lock -> ref_name ,
oid_to_hex ( & lock -> old_oid ) ,
sha1_to_hex ( old_sha1 ) ) ;
errno = EBUSY ;
return - 1 ;
}
return 0 ;
}
static int remove_empty_directories ( struct strbuf * path )
{
```

Código

```
return remove_dir_recursively ( path , REMOVE_DIR_EMPTY_ONLY ) ;
}
static int create_reflock ( const char * path , void * cb )
{
struct lock_file * lk = cb ;
return hold_lock_file_for_update ( lk , path , LOCK_NO_DEREF ) < 0 ? - 1 : 0 ;
}
static struct ref_lock * lock_ref_sha1_basic ( struct files_ref_store * refs ,
const char * refname ,
const unsigned char * old_sha1 ,
const struct string_list * extras ,
const struct string_list * skip ,
unsigned int flags , int * type ,
```

Código

```
struct strbuf * err )
{
struct strbuf ref_file = STRBUF_INIT ;
struct ref_lock * lock ;
int last_errno = 0 ;
int mustexist = ( old_sha1 && ! is_null_sha1 ( old_sha1 ) ) ;
int resolve_flags = RESOLVE_REF_NO_RECURSE ;
int resolved ;
files_assert_main_repository ( refs , "lock_ref_sha1_basic" ) ;
assert ( err ) ;
lock = xcalloc ( 1 , sizeof ( struct ref_lock ) ) ;
if ( mustexist )
resolve_flags |= RESOLVE_REF_READING ;
```

Código

```
if ( flags & REF_DELETING )
resolve_flags |= RESOLVE_REF_ALLOW_BAD_NAME ;
files_ref_path ( refs , & ref_file , refname ) ;
resolved = ! ! refs_resolve_ref_unsafe ( & refs -> base ,
refname , resolve_flags ,
lock -> old_oid . hash , type ) ;
if ( ! resolved && errno == EISDIR ) {
if ( remove_empty_directories ( & ref_file ) ) {
last_errno = errno ;
if ( ! refs_verify_refname_available (
& refs -> base ,
refname , extras , skip , err ) )
strbuf_addf ( err , "there are still refs under '%s'" ,
```

Código

```
refname ) ;
goto error_return ;
}
resolved = ! ! refs_resolve_ref_unsafe ( & refs -> base ,
refname , resolve_flags ,
lock -> old_oid . hash , type ) ;
}
if ( ! resolved ) {
last_errno = errno ;
if ( last_errno != ENOTDIR ||
! refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) )
strbuf_addf ( err , "unable to resolve reference '%s': %s" ,
```

Código

```
refname , strerror ( last_errno ) ) ;
goto error_return ;
}
if ( is_null_oid ( & lock -> old_oid ) &&
refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) ) {
last_errno = ENOTDIR ;
goto error_return ;
}
lock -> lk = xcalloc ( 1 , sizeof ( struct lock_file ) ) ;
lock -> ref_name = xstrdup ( refname ) ;
if ( raceproof_create_file ( ref_file . buf , create_reflock , lock -> lk ) ) {
last_errno = errno ;
```

Código

```
unable_to_lock_message ( ref_file . buf , errno , err ) ;
goto error_return ;
}
if ( verify_lock ( & refs -> base , lock , old_sha1 , mustexist , err ) ) {
last_errno = errno ;
goto error_return ;
}
goto out ;
error_return :
unlock_ref ( lock ) ;
lock = NULL ;
out :
strbuf_release ( & ref_file ) ;
```

Código

```
errno = last_errno ;
return lock ;
}
static void write_packed_entry ( FILE * fh , const char * refname ,
const unsigned char * sha1 ,
const unsigned char * peeled )
{
fprintf_or_die ( fh , "%s %s\n" , sha1_to_hex ( sha1 ) , refname ) ;
if ( peeled )
fprintf_or_die ( fh , "^%s\n" , sha1_to_hex ( peeled ) ) ;
}
static int lock_packed_refs ( struct files_ref_store * refs , int flags )
{
```

Código

```
static int timeout_configured = 0 ;
static int timeout_value = 1000 ;
struct packed_ref_cache * packed_ref_cache ;
files_assert_main_repository ( refs , "lock_packed_refs" ) ;
if ( ! timeout_configured ) {
git_config_get_int ( "core.packedrefstimeout" , & timeout_value ) ;
timeout_configured = 1 ;
}
if ( hold_lock_file_for_update_timeout (
& packlock , files_packed_refs_path ( refs ) ,
flags , timeout_value ) < 0 )
return - 1 ;
packed_ref_cache = get_packed_ref_cache ( refs ) ;
```

Código

```
packed_ref_cache -> lock = & packlock ;
acquire_packed_ref_cache ( packed_ref_cache ) ;
return 0 ;
}
static int commit_packed_refs ( struct files_ref_store * refs )
{
struct packed_ref_cache * packed_ref_cache =
get_packed_ref_cache ( refs ) ;
int ok , error = 0 ;
int save_errno = 0 ;
FILE * out ;
struct ref_iterator * iter ;
files_assert_main_repository ( refs , "commit_packed_refs" ) ;
```

Código

```
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed-refs not locked" ) ;
out = fdopen_lock_file ( packed_ref_cache -> lock , "w" ) ;
if ( ! out )
die_errno ( "unable to fdopen packed-refs descriptor" ) ;
fprintf_or_die ( out , "%s" , PACKED_REFS_HEADER ) ;
iter = cache_ref_iterator_begin ( packed_ref_cache -> cache , NULL , 0 ) ;
while ( ( ok = ref_iterator_advance ( iter ) ) == ITER_OK ) {
struct object_id peeled ;
int peel_error = ref_iterator_peel ( iter , & peeled ) ;
write_packed_entry ( out , iter -> refname , iter -> oid -> hash ,
peel_error ? NULL : peeled . hash ) ;
}
```

Código

```
if ( ok != ITER_DONE )
die ( "error while iterating over references" ) ;
if ( commit_lock_file ( packed_ref_cache -> lock ) ) {
save_errno = errno ;
error = - 1 ;
}
packed_ref_cache -> lock = NULL ;
release_packed_ref_cache ( packed_ref_cache ) ;
errno = save_errno ;
return error ;
}
static void rollback_packed_refs ( struct files_ref_store * refs )
{
```

Código

```
struct packed_ref_cache * packed_ref_cache =
get_packed_ref_cache ( refs ) ;
files_assert_main_repository ( refs , "rollback_packed_refs" ) ;
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed-refs not locked" ) ;
rollback_lock_file ( packed_ref_cache -> lock ) ;
packed_ref_cache -> lock = NULL ;
release_packed_ref_cache ( packed_ref_cache ) ;
clear_packed_ref_cache ( refs ) ;
}
struct ref_to_prune {
struct ref_to_prune * next ;
unsigned char sha1 [ 20 ] ;
```

Código

```
char name [ FLEX_ARRAY ] ;
} ;
enum {
REMOVE_EMPTY_PARENTS_REF = 0x01 ,
REMOVE_EMPTY_PARENTS_REFLOG = 0x02
} ;
static void try_remove_empty_parents ( struct files_ref_store * refs ,
const char * refname ,
unsigned int flags )
{
struct strbuf buf = STRBUF_INIT ;
struct strbuf sb = STRBUF_INIT ;
char * p , * q ;
```

Código

```
int i ;
strbuf_addstr ( & buf , refname ) ;
p = buf . buf ;
for ( i = 0 ; i < 2 ; i ++ ) {
while ( * p && * p != '/' )
p ++ ;
while ( * p == '/' )
p ++ ;
}
q = buf . buf + buf . len ;
while ( flags & ( REMOVE_EMPTY_PARENTS_REF | REMOVE_EMPTY_PARENTS_REFLOG ) ) {
while ( q > p && * q != '/' )
q -- ;
```

Código

```
while ( q > p && * ( q - 1 ) == '/' )
q -- ;
if ( q == p )
break ;
strbuf_setlen ( & buf , q - buf . buf ) ;
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , buf . buf ) ;
if ( ( flags & REMOVE_EMPTY_PARENTS_REF ) && rmdir ( sb . buf ) )
flags &= ~ REMOVE_EMPTY_PARENTS_REF ;
strbuf_reset ( & sb ) ;
files_reflog_path ( refs , & sb , buf . buf ) ;
if ( ( flags & REMOVE_EMPTY_PARENTS_REFLOG ) && rmdir ( sb . buf ) )
flags &= ~ REMOVE_EMPTY_PARENTS_REFLOG ;
```

Código

```
}
strbuf_release ( & buf ) ;
strbuf_release ( & sb ) ;
}
static void prune_ref ( struct files_ref_store * refs , struct ref_to_prune * r )
{
struct ref_transaction * transaction ;
struct strbuf err = STRBUF_INIT ;
if ( check_refname_format ( r -> name , 0 ) )
return ;
transaction = ref_store_transaction_begin ( & refs -> base , & err ) ;
if ( ! transaction ||
ref_transaction_delete ( transaction , r -> name , r -> sha1 ,
```

Código

```
REF_ISPRUNING | REF_NODEREF , NULL , & err ) ||
ref_transaction_commit ( transaction , & err ) ) {
ref_transaction_free ( transaction ) ;
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
return ;
}
ref_transaction_free ( transaction ) ;
strbuf_release ( & err ) ;
}
static void prune_refs ( struct files_ref_store * refs , struct ref_to_prune * r )
{
while ( r ) {
```

Código

```
prune_ref ( refs , r ) ;
r = r -> next ;
}
}
static int files_pack_refs ( struct ref_store * ref_store , unsigned int flags )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE | REF_STORE_ODB ,
"pack_refs" ) ;
struct ref_iterator * iter ;
struct ref_dir * packed_refs ;
int ok ;
struct ref_to_prune * refs_to_prune = NULL ;
```

Código

```
lock_packed_refs ( refs , LOCK_DIE_ON_ERROR ) ;
packed_refs = get_packed_refs ( refs ) ;
iter = cache_ref_iterator_begin ( get_loose_ref_cache ( refs ) , NULL , 0 ) ;
while ( ( ok = ref_iterator_advance ( iter ) ) == ITER_OK ) {
struct ref_entry * packed_entry ;
int is_tag_ref = starts_with ( iter -> refname , "refs/tags/" ) ;
if ( ref_type ( iter -> refname ) != REF_TYPE_NORMAL )
continue ;
if ( ! ( flags & PACK_REFS_ALL ) && ! is_tag_ref )
continue ;
if ( iter -> flags & REF_ISSYMREF )
continue ;
if ( ! ref_resolves_to_object ( iter -> refname , iter -> oid , iter -> flags ) )
```

Código

```
continue ;
packed_entry = find_ref_entry ( packed_refs , iter -> refname ) ;
if ( packed_entry ) {
packed_entry -> flag = REF_ISPACKED ;
oidcpy ( & packed_entry -> u . value . oid , iter -> oid ) ;
} else {
packed_entry = create_ref_entry ( iter -> refname , iter -> oid ,
REF_ISPACKED , 0 ) ;
add_ref_entry ( packed_refs , packed_entry ) ;
}
oidclr ( & packed_entry -> u . value . peeled ) ;
if ( ( flags & PACK_REFS_PRUNE ) ) {
struct ref_to_prune * n ;
```

Código

```
FLEX_ALLOC_STR ( n , name , iter -> refname ) ;
hashcpy ( n -> sha1 , iter -> oid -> hash ) ;
n -> next = refs_to_prune ;
refs_to_prune = n ;
}
}
if ( ok != ITER_DONE )
die ( "error while iterating over references" ) ;
if ( commit_packed_refs ( refs ) )
die_errno ( "unable to overwrite old ref-pack file" ) ;
prune_refs ( refs , refs_to_prune ) ;
return 0 ;
}
```

Código

```
static int repack_without_refs ( struct files_ref_store * refs ,
struct string_list * refnames , struct strbuf * err )
{
struct ref_dir * packed ;
struct string_list_item * refname ;
int ret , needs_repacking = 0 , removed = 0 ;
files_assert_main_repository ( refs , "repack_without_refs" ) ;
assert ( err ) ;
for_each_string_list_item ( refname , refnames ) {
if ( get_packed_ref ( refs , refname -> string ) ) {
needs_repacking = 1 ;
break ;
}
```

Código

```
}
if ( ! needs_repacking )
return 0 ;
if ( lock_packed_refs ( refs , 0 ) ) {
unable_to_lock_message ( files_packed_refs_path ( refs ) , errno , err ) ;
return - 1 ;
}
packed = get_packed_refs ( refs ) ;
for_each_string_list_item ( refname , refnames )
if ( remove_entry_from_dir ( packed , refname -> string ) != - 1 )
removed = 1 ;
if ( ! removed ) {
rollback_packed_refs ( refs ) ;
```

Código

```
return 0 ;
}
ret = commit_packed_refs ( refs ) ;
if ( ret )
strbuf_addf ( err , "unable to overwrite old ref-pack file: %s" ,
strerror ( errno ) ) ;
return ret ;
}
static int files_delete_refs ( struct ref_store * ref_store ,
struct string_list * refnames , unsigned int flags )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "delete_refs" ) ;
```

Código

```
struct strbuf err = STRBUF_INIT ;
int i , result = 0 ;
if ( ! refnames -> nr )
return 0 ;
result = repack_without_refs ( refs , refnames , & err ) ;
if ( result ) {
if ( refnames -> nr == 1 )
error ( _ ( "could not delete reference %s: %s" ) ,
refnames -> items [ 0 ] . string , err . buf ) ;
else
error ( _ ( "could not delete references: %s" ) , err . buf ) ;
goto out ;
}
```

Código

```
for ( i = 0 ; i < refnames -> nr ; i ++ ) {
const char * refname = refnames -> items [ i ] . string ;
if ( refs_delete_ref ( & refs -> base , NULL , refname , NULL , flags ) )
result |= error ( _ ( "could not remove reference %s" ) , refname ) ;
}
out :
strbuf_release ( & err ) ;
return result ;
}
struct rename_cb {
const char * tmp_renamed_log ;
int true_errno ;
} ;
```

Código

```
static int rename_tmp_log_callback ( const char * path , void * cb_data )
{
struct rename_cb * cb = cb_data ;
if ( rename ( cb -> tmp_renamed_log , path ) ) {
cb -> true_errno = errno ;
if ( errno == ENOTDIR )
errno = EISDIR ;
return - 1 ;
} else {
return 0 ;
}
}
static int rename_tmp_log ( struct files_ref_store * refs , const char * newrefname )
```

Código

```
{
struct strbuf path = STRBUF_INIT ;
struct strbuf tmp = STRBUF_INIT ;
struct rename_cb cb ;
int ret ;
files_reflog_path ( refs , & path , newrefname ) ;
files_reflog_path ( refs , & tmp , "refs/.tmp-renamed-log" ) ;
cb . tmp_renamed_log = tmp . buf ;
ret = raceproof_create_file ( path . buf , rename_tmp_log_callback , & cb ) ;
if ( ret ) {
if ( errno == EISDIR )
error ( "directory not empty: %s" , path . buf ) ;
else
```

Código

```
error ( "unable to move logfile %s to %s: %s" ,
tmp . buf , path . buf ,
strerror ( cb . true_errno ) ) ;
}
strbuf_release ( & path ) ;
strbuf_release ( & tmp ) ;
return ret ;
}
static int write_ref_to_lockfile ( struct ref_lock * lock ,
const struct object_id * oid , struct strbuf * err ) ;
static int commit_ref_update ( struct files_ref_store * refs ,
struct ref_lock * lock ,
const struct object_id * oid , const char * logmsg ,
```

Código

```
struct strbuf * err ) ;
static int files_rename_ref ( struct ref_store * ref_store ,
const char * oldrefname , const char * newrefname ,
const char * logmsg )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "rename_ref" ) ;
struct object_id oid , orig_oid ;
int flag = 0 , logmoved = 0 ;
struct ref_lock * lock ;
struct stat loginfo ;
struct strbuf sb_oldref = STRBUF_INIT ;
struct strbuf sb_newref = STRBUF_INIT ;
```

Código

```
struct strbuf tmp_renamed_log = STRBUF_INIT ;
int log , ret ;
struct strbuf err = STRBUF_INIT ;
files_reflog_path ( refs , & sb_oldref , oldrefname ) ;
files_reflog_path ( refs , & sb_newref , newrefname ) ;
files_reflog_path ( refs , & tmp_renamed_log , "refs/.tmp-renamed-log" ) ;
log = ! lstat ( sb_oldref . buf , & loginfo ) ;
if ( log && S_ISLNK ( loginfo . st_mode ) ) {
ret = error ( "reflog for %s is a symlink" , oldrefname ) ;
goto out ;
}
if ( ! refs_resolve_ref_unsafe ( & refs -> base , oldrefname ,
RESOLVE_REF_READING | RESOLVE_REF_NO_RECURSE ,
```

Código

```
orig_oid . hash , & flag ) ) {
ret = error ( "refname %s not found" , oldrefname ) ;
goto out ;
}
if ( flag & REF_ISSYMREF ) {
ret = error ( "refname %s is a symbolic ref, renaming it is not supported" ,
oldrefname ) ;
goto out ;
}
if ( ! refs_rename_ref_available ( & refs -> base , oldrefname , newrefname ) ) {
ret = 1 ;
goto out ;
}
```

Código

```
if ( log && rename ( sb_oldref . buf , tmp_renamed_log . buf ) ) {
ret = error ( "unable to move logfile logs/%s to logs/" "refs/.tmp-renamed-log" ": %s" ,
oldrefname , strerror ( errno ) ) ;
goto out ;
}
if ( refs_delete_ref ( & refs -> base , logmsg , oldrefname ,
orig_oid . hash , REF_NODEREF ) ) {
error ( "unable to delete old %s" , oldrefname ) ;
goto rollback ;
}
if ( ! refs_read_ref_full ( & refs -> base , newrefname ,
RESOLVE_REF_READING | RESOLVE_REF_NO_RECURSE ,
oid . hash , NULL ) &&
```

Código

```
refs_delete_ref ( & refs -> base , NULL , newrefname ,
NULL , REF_NODEREF ) ) {
if ( errno == EISDIR ) {
struct strbuf path = STRBUF_INIT ;
int result ;
files_ref_path ( refs , & path , newrefname ) ;
result = remove_empty_directories ( & path ) ;
strbuf_release ( & path ) ;
if ( result ) {
error ( "Directory not empty: %s" , newrefname ) ;
goto rollback ;
}
} else {
```

Código

```
error ( "unable to delete existing %s" , newrefname ) ;
goto rollback ;
}
}
if ( log && rename_tmp_log ( refs , newrefname ) )
goto rollback ;
logmoved = log ;
lock = lock_ref_sha1_basic ( refs , newrefname , NULL , NULL , NULL ,
REF_NODEREF , NULL , & err ) ;
if ( ! lock ) {
error ( "unable to rename '%s' to '%s': %s" , oldrefname , newrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollback ;
```

Código

```
}
oidcpy ( & lock -> old_oid , & orig_oid ) ;
if ( write_ref_to_lockfile ( lock , & orig_oid , & err ) ||
commit_ref_update ( refs , lock , & orig_oid , logmsg , & err ) ) {
error ( "unable to write current sha1 into %s: %s" , newrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollback ;
}
ret = 0 ;
goto out ;
rollback :
lock = lock_ref_sha1_basic ( refs , oldrefname , NULL , NULL , NULL ,
REF_NODEREF , NULL , & err ) ;
```

Código

```
if ( ! lock ) {
error ( "unable to lock %s for rollback: %s" , oldrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollbacklog ;
}
flag = log_all_ref_updates ;
log_all_ref_updates = LOG_REFS_NONE ;
if ( write_ref_to_lockfile ( lock , & orig_oid , & err ) ||
commit_ref_update ( refs , lock , & orig_oid , NULL , & err ) ) {
error ( "unable to write current sha1 into %s: %s" , oldrefname , err . buf ) ;
strbuf_release ( & err ) ;
}
log_all_ref_updates = flag ;
```

Código

```
rollbacklog :
if ( logmoved && rename ( sb_newref . buf , sb_oldref . buf ) )
error ( "unable to restore logfile %s from %s: %s" ,
oldrefname , newrefname , strerror ( errno ) ) ;
if ( ! logmoved && log &&
rename ( tmp_renamed_log . buf , sb_oldref . buf ) )
error ( "unable to restore logfile %s from logs/" "refs/.tmp-renamed-log" ": %s" ,
oldrefname , strerror ( errno ) ) ;
ret = 1 ;
out :
strbuf_release ( & sb_newref ) ;
strbuf_release ( & sb_oldref ) ;
strbuf_release ( & tmp_renamed_log ) ;
```

Código

```
return ret ;
}
static int close_ref ( struct ref_lock * lock )
{
if ( close_lock_file ( lock -> lk ) )
return - 1 ;
return 0 ;
}
static int commit_ref ( struct ref_lock * lock )
{
char * path = get_locked_file_path ( lock -> lk ) ;
struct stat st ;
if ( ! lstat ( path , & st ) && S_ISDIR ( st . st_mode ) ) {
```

Código

```
size_t len = strlen ( path ) ;
struct strbuf sb_path = STRBUF_INIT ;
strbuf_attach ( & sb_path , path , len , len ) ;
remove_empty_directories ( & sb_path ) ;
strbuf_release ( & sb_path ) ;
} else {
free ( path ) ;
}
if ( commit_lock_file ( lock -> lk ) )
return - 1 ;
return 0 ;
}
static int open_or_create_logfile ( const char * path , void * cb )
```

Código

```
{
int * fd = cb ;
* fd = open ( path , O_APPEND | O_WRONLY | O_CREAT , 0666 ) ;
return ( * fd < 0 ) ? - 1 : 0 ;
}
static int log_ref_setup ( struct files_ref_store * refs ,
const char * refname , int force_create ,
int * logfd , struct strbuf * err )
{
struct strbuf logfile_sb = STRBUF_INIT ;
char * logfile ;
files_reflog_path ( refs , & logfile_sb , refname ) ;
logfile = strbuf_detach ( & logfile_sb , NULL ) ;
```

Código

```
if ( force_create || should_autocreate_reflog ( refname ) ) {
if ( raceproof_create_file ( logfile , open_or_create_logfile , logfd ) ) {
if ( errno == ENOENT )
strbuf_addf ( err , "unable to create directory for '%s': "
"%s" , logfile , strerror ( errno ) ) ;
else if ( errno == EISDIR )
strbuf_addf ( err , "there are still logs under '%s'" ,
logfile ) ;
else
strbuf_addf ( err , "unable to append to '%s': %s" ,
logfile , strerror ( errno ) ) ;
goto error ;
}
```

Código

```
} else {
* logfd = open ( logfile , O_APPEND | O_WRONLY , 0666 ) ;
if ( * logfd < 0 ) {
if ( errno == ENOENT || errno == EISDIR ) {
;
} else {
strbuf_addf ( err , "unable to append to '%s': %s" ,
logfile , strerror ( errno ) ) ;
goto error ;
}
}
}
if ( * logfd >= 0 )
```

Código

```
adjust_shared_perm ( logfile ) ;
free ( logfile ) ;
return 0 ;
error :
free ( logfile ) ;
return - 1 ;
}
static int files_create_reflog ( struct ref_store * ref_store ,
const char * refname , int force_create ,
struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "create_reflog" ) ;
```

Código

```
int fd ;
if ( log_ref_setup ( refs , refname , force_create , & fd , err ) )
return - 1 ;
if ( fd >= 0 )
close ( fd ) ;
return 0 ;
}
static int log_ref_write_fd ( int fd , const struct object_id * old_oid ,
const struct object_id * new_oid ,
const char * committer , const char * msg )
{
int msglen , written ;
unsigned maxlen , len ;
```

Código

```
char * logrec ;
msglen = msg ? strlen ( msg ) : 0 ;
maxlen = strlen ( committer ) + msglen + 100 ;
logrec = xmalloc ( maxlen ) ;
len = xsnprintf ( logrec , maxlen , "%s %s %s\n" ,
oid_to_hex ( old_oid ) ,
oid_to_hex ( new_oid ) ,
committer ) ;
if ( msglen )
len += copy_reflog_msg ( logrec + len - 1 , msg ) - 1 ;
written = len <= maxlen ? write_in_full ( fd , logrec , len ) : - 1 ;
free ( logrec ) ;
if ( written != len )
```

Código

```
return - 1 ;
return 0 ;
}
static int files_log_ref_write ( struct files_ref_store * refs ,
const char * refname , const struct object_id * old_oid ,
const struct object_id * new_oid , const char * msg ,
int flags , struct strbuf * err )
{
int logfd , result ;
if ( log_all_ref_updates == LOG_REFS_UNSET )
log_all_ref_updates = is_bare_repository ( ) ? LOG_REFS_NONE : LOG_REFS_NORMAL ;
result = log_ref_setup ( refs , refname ,
flags & REF_FORCE_CREATE_REFLOG ,
```

Código

```
& logfd , err ) ;
if ( result )
return result ;
if ( logfd < 0 )
return 0 ;
result = log_ref_write_fd ( logfd , old_oid , new_oid ,
git_committer_info ( 0 ) , msg ) ;
if ( result ) {
struct strbuf sb = STRBUF_INIT ;
int save_errno = errno ;
files_reflog_path ( refs , & sb , refname ) ;
strbuf_addf ( err , "unable to append to '%s': %s" ,
sb . buf , strerror ( save_errno ) ) ;
```

Código

```
strbuf_release ( & sb ) ;
close ( logfd ) ;
return - 1 ;
}
if ( close ( logfd ) ) {
struct strbuf sb = STRBUF_INIT ;
int save_errno = errno ;
files_reflog_path ( refs , & sb , refname ) ;
strbuf_addf ( err , "unable to append to '%s': %s" ,
sb . buf , strerror ( save_errno ) ) ;
strbuf_release ( & sb ) ;
return - 1 ;
}
```

Código

```
return 0 ;
}
static int write_ref_to_lockfile ( struct ref_lock * lock ,
const struct object_id * oid , struct strbuf * err )
{
static char term = '\n' ;
struct object * o ;
int fd ;
o = parse_object ( oid ) ;
if ( ! o ) {
strbuf_addf ( err ,
"trying to write ref '%s' with nonexistent object %s" ,
lock -> ref_name , oid_to_hex ( oid ) ) ;
```

Código

```
unlock_ref ( lock ) ;
return - 1 ;
}
if ( o -> type != OBJ_COMMIT && is_branch ( lock -> ref_name ) ) {
strbuf_addf ( err ,
"trying to write non-commit object %s to branch '%s'" ,
oid_to_hex ( oid ) , lock -> ref_name ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
fd = get_lock_file_fd ( lock -> lk ) ;
if ( write_in_full ( fd , oid_to_hex ( oid ) , GIT_SHA1_HEXSZ ) != GIT_SHA1_HEXSZ ||
write_in_full ( fd , & term , 1 ) != 1 ||
```

Código

```
close_ref ( lock ) < 0 ) {
strbuf_addf ( err ,
"couldn't write '%s'" , get_lock_file_path ( lock -> lk ) ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
return 0 ;
}
static int commit_ref_update ( struct files_ref_store * refs ,
struct ref_lock * lock ,
const struct object_id * oid , const char * logmsg ,
struct strbuf * err )
{
```

Código

```
files_assert_main_repository ( refs , "commit_ref_update" ) ;
clear_loose_ref_cache ( refs ) ;
if ( files_log_ref_write ( refs , lock -> ref_name ,
& lock -> old_oid , oid ,
logmsg , 0 , err ) ) {
char * old_msg = strbuf_detach ( err , NULL ) ;
strbuf_addf ( err , "cannot update the ref '%s': %s" ,
lock -> ref_name , old_msg ) ;
free ( old_msg ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
if ( strcmp ( lock -> ref_name , "HEAD" ) != 0 ) {
```

Código

```
struct object_id head_oid ;
int head_flag ;
const char * head_ref ;
head_ref = refs_resolve_ref_unsafe ( & refs -> base , "HEAD" ,
RESOLVE_REF_READING ,
head_oid . hash , & head_flag ) ;
if ( head_ref && ( head_flag & REF_ISSYMREF ) &&
! strcmp ( head_ref , lock -> ref_name ) ) {
struct strbuf log_err = STRBUF_INIT ;
if ( files_log_ref_write ( refs , "HEAD" ,
& lock -> old_oid , oid ,
logmsg , 0 , & log_err ) ) {
error ( "%s" , log_err . buf ) ;
```

Código

```
strbuf_release ( & log_err ) ;
}
}
}
if ( commit_ref ( lock ) ) {
strbuf_addf ( err , "couldn't set '%s'" , lock -> ref_name ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
unlock_ref ( lock ) ;
return 0 ;
}
static int create_ref_symlink ( struct ref_lock * lock , const char * target )
```

Código

```
{
int ret = - 1 ;
return ret ;
}
static void update_symref_reflog ( struct files_ref_store * refs ,
struct ref_lock * lock , const char * refname ,
const char * target , const char * logmsg )
{
struct strbuf err = STRBUF_INIT ;
struct object_id new_oid ;
if ( logmsg &&
! refs_read_ref_full ( & refs -> base , target ,
RESOLVE_REF_READING , new_oid . hash , NULL ) &&
```

Código

```
files_log_ref_write ( refs , refname , & lock -> old_oid ,
& new_oid , logmsg , 0 , & err ) ) {
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
}
}
static int create_symref_locked ( struct files_ref_store * refs ,
struct ref_lock * lock , const char * refname ,
const char * target , const char * logmsg )
{
if ( prefer_symlink_refs && ! create_ref_symlink ( lock , target ) ) {
update_symref_reflog ( refs , lock , refname , target , logmsg ) ;
return 0 ;
```

Código

```
}
if ( ! fdopen_lock_file ( lock -> lk , "w" ) )
return error ( "unable to fdopen %s: %s" ,
lock -> lk -> tempfile . filename . buf , strerror ( errno ) ) ;
update_symref_reflog ( refs , lock , refname , target , logmsg ) ;
fprintf ( lock -> lk -> tempfile . fp , "ref: %s\n" , target ) ;
if ( commit_ref ( lock ) < 0 )
return error ( "unable to write symref for %s: %s" , refname ,
strerror ( errno ) ) ;
return 0 ;
}
static int files_create_symref ( struct ref_store * ref_store ,
const char * refname , const char * target ,
```

Código

```
const char * logmsg )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "create_symref" ) ;
struct strbuf err = STRBUF_INIT ;
struct ref_lock * lock ;
int ret ;
lock = lock_ref_sha1_basic ( refs , refname , NULL ,
NULL , NULL , REF_NODEREF , NULL ,
& err ) ;
if ( ! lock ) {
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
```

Código

```
return - 1 ;
}
ret = create_symref_locked ( refs , lock , refname , target , logmsg ) ;
unlock_ref ( lock ) ;
return ret ;
}
static int files_reflog_exists ( struct ref_store * ref_store ,
const char * refname )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "reflog_exists" ) ;
struct strbuf sb = STRBUF_INIT ;
struct stat st ;
```

Código

```
int ret ;
files_reflog_path ( refs , & sb , refname ) ;
ret = ! lstat ( sb . buf , & st ) && S_ISREG ( st . st_mode ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int files_delete_reflog ( struct ref_store * ref_store ,
const char * refname )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "delete_reflog" ) ;
struct strbuf sb = STRBUF_INIT ;
int ret ;
```

Código

```
files_reflog_path ( refs , & sb , refname ) ;
ret = remove_path ( sb . buf ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int show_one_reflog_ent ( struct strbuf * sb , each_reflog_ent_fn fn , void * cb_data )
{
struct object_id ooid , noid ;
char * email_end , * message ;
timestamp_t timestamp ;
int tz ;
const char * p = sb -> buf ;
if ( ! sb -> len || sb -> buf [ sb -> len - 1 ] != '\n' ||
```

Código

```
parse_oid_hex ( p , & ooid , & p ) || * p ++ != ' ' ||
parse_oid_hex ( p , & noid , & p ) || * p ++ != ' ' ||
! ( email_end = strchr ( p , '>' ) ) ||
email_end [ 1 ] != ' ' ||
! ( timestamp = parse_timestamp ( email_end + 2 , & message , 10 ) ) ||
! message || message [ 0 ] != ' ' ||
( message [ 1 ] != '+' && message [ 1 ] != '-' ) ||
! isdigit ( message [ 2 ] ) || ! isdigit ( message [ 3 ] ) ||
! isdigit ( message [ 4 ] ) || ! isdigit ( message [ 5 ] ) )
return 0 ;
email_end [ 1 ] = '\0' ;
tz = strtol ( message + 1 , NULL , 10 ) ;
if ( message [ 6 ] != '\t' )
```

Código

```
message += 6 ;
else
message += 7 ;
return fn ( & ooid , & noid , p , timestamp , tz , message , cb_data ) ;
}
static char * find_beginning_of_line ( char * bob , char * scan )
{
while ( bob < scan && * ( -- scan ) != '\n' )
;
return scan ;
}
static int files_for_each_reflog_ent_reverse ( struct ref_store * ref_store ,
const char * refname ,
```

Código

```
each_reflog_ent_fn fn ,
void * cb_data )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"for_each_reflog_ent_reverse" ) ;
struct strbuf sb = STRBUF_INIT ;
FILE * logfp ;
long pos ;
int ret = 0 , at_tail = 1 ;
files_reflog_path ( refs , & sb , refname ) ;
logfp = fopen ( sb . buf , "r" ) ;
strbuf_release ( & sb ) ;
```

Código

```
if ( ! logfp )
return - 1 ;
if ( fseek ( logfp , 0 , SEEK_END ) < 0 )
ret = error ( "cannot seek back reflog for %s: %s" ,
refname , strerror ( errno ) ) ;
pos = ftell ( logfp ) ;
while ( ! ret && 0 < pos ) {
int cnt ;
size_t nread ;
char buf [ BUFSIZ ] ;
char * endp , * scanp ;
cnt = ( sizeof ( buf ) < pos ) ? sizeof ( buf ) : pos ;
if ( fseek ( logfp , pos - cnt , SEEK_SET ) ) {
```

Código

```
ret = error ( "cannot seek back reflog for %s: %s" ,
refname , strerror ( errno ) ) ;
break ;
}
nread = fread ( buf , cnt , 1 , logfp ) ;
if ( nread != 1 ) {
ret = error ( "cannot read %d bytes from reflog for %s: %s" ,
cnt , refname , strerror ( errno ) ) ;
break ;
}
pos -= cnt ;
scanp = endp = buf + cnt ;
if ( at_tail && scanp [ - 1 ] == '\n' )
```

Código

```
scanp -- ;
at_tail = 0 ;
while ( buf < scanp ) {
char * bp ;
bp = find_beginning_of_line ( buf , scanp ) ;
if ( * bp == '\n' ) {
strbuf_splice ( & sb , 0 , 0 , bp + 1 , endp - ( bp + 1 ) ) ;
scanp = bp ;
endp = bp + 1 ;
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
strbuf_reset ( & sb ) ;
if ( ret )
break ;
```

Código

```
} else if ( ! pos ) {
strbuf_splice ( & sb , 0 , 0 , buf , endp - buf ) ;
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
strbuf_reset ( & sb ) ;
break ;
}
if ( bp == buf ) {
strbuf_splice ( & sb , 0 , 0 , buf , endp - buf ) ;
break ;
}
}
}
if ( ! ret && sb . len )
```

Código

```
die ( "BUG: reverse reflog parser had leftover data" ) ;
fclose ( logfp ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int files_for_each_reflog_ent ( struct ref_store * ref_store ,
const char * refname ,
each_reflog_ent_fn fn , void * cb_data )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"for_each_reflog_ent" ) ;
FILE * logfp ;
```

Código

```
struct strbuf sb = STRBUF_INIT ;
int ret = 0 ;
files_reflog_path ( refs , & sb , refname ) ;
logfp = fopen ( sb . buf , "r" ) ;
strbuf_release ( & sb ) ;
if ( ! logfp )
return - 1 ;
while ( ! ret && ! strbuf_getwholeline ( & sb , logfp , '\n' ) )
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
fclose ( logfp ) ;
strbuf_release ( & sb ) ;
return ret ;
}
```

Código

```
struct files_reflog_iterator {
struct ref_iterator base ;
struct ref_store * ref_store ;
struct dir_iterator * dir_iterator ;
struct object_id oid ;
} ;
static int files_reflog_iterator_advance ( struct ref_iterator * ref_iterator )
{
struct files_reflog_iterator * iter =
( struct files_reflog_iterator * ) ref_iterator ;
struct dir_iterator * diter = iter -> dir_iterator ;
int ok ;
while ( ( ok = dir_iterator_advance ( diter ) ) == ITER_OK ) {
```

Código

```
int flags ;
if ( ! S_ISREG ( diter -> st . st_mode ) )
continue ;
if ( diter -> basename [ 0 ] == '.' )
continue ;
if ( ends_with ( diter -> basename , ".lock" ) )
continue ;
if ( refs_read_ref_full ( iter -> ref_store ,
diter -> relative_path , 0 ,
iter -> oid . hash , & flags ) ) {
error ( "bad ref for %s" , diter -> path . buf ) ;
continue ;
}
```

Código

```
iter -> base . refname = diter -> relative_path ;
iter -> base . oid = & iter -> oid ;
iter -> base . flags = flags ;
return ITER_OK ;
}
iter -> dir_iterator = NULL ;
if ( ref_iterator_abort ( ref_iterator ) == ITER_ERROR )
ok = ITER_ERROR ;
return ok ;
}
static int files_reflog_iterator_peel ( struct ref_iterator * ref_iterator ,
struct object_id * peeled )
{
```

Código

```
die ( "BUG: ref_iterator_peel() called for reflog_iterator" ) ;
}
static int files_reflog_iterator_abort ( struct ref_iterator * ref_iterator )
{
struct files_reflog_iterator * iter =
( struct files_reflog_iterator * ) ref_iterator ;
int ok = ITER_DONE ;
if ( iter -> dir_iterator )
ok = dir_iterator_abort ( iter -> dir_iterator ) ;
base_ref_iterator_free ( ref_iterator ) ;
return ok ;
}
static struct ref_iterator_vtable files_reflog_iterator_vtable = {
```

Código

```
files_reflog_iterator_advance ,
files_reflog_iterator_peel ,
files_reflog_iterator_abort
} ;
static struct ref_iterator * files_reflog_iterator_begin ( struct ref_store * ref_store )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"reflog_iterator_begin" ) ;
struct files_reflog_iterator * iter = xcalloc ( 1 , sizeof ( * iter ) ) ;
struct ref_iterator * ref_iterator = & iter -> base ;
struct strbuf sb = STRBUF_INIT ;
base_ref_iterator_init ( ref_iterator , & files_reflog_iterator_vtable ) ;
```

Código

```
files_reflog_path ( refs , & sb , NULL ) ;
iter -> dir_iterator = dir_iterator_begin ( sb . buf ) ;
iter -> ref_store = ref_store ;
strbuf_release ( & sb ) ;
return ref_iterator ;
}
static int ref_update_reject_duplicates ( struct string_list * refnames ,
struct strbuf * err )
{
int i , n = refnames -> nr ;
assert ( err ) ;
for ( i = 1 ; i < n ; i ++ )
if ( ! strcmp ( refnames -> items [ i - 1 ] . string , refnames -> items [ i ] . string ) ) {
```

Código

```
strbuf_addf ( err ,
"multiple updates for ref ’%s’ not allowed." ,
refnames -> items [ i ] . string ) ;
return 1 ;
}
return 0 ;
}
static int split_head_update ( struct ref_update * update ,
struct ref_transaction * transaction ,
const char * head_ref ,
struct string_list * affected_refnames ,
struct strbuf * err )
{
```

Código

```
struct string_list_item * item ;
struct ref_update * new_update ;
if ( ( update -> flags & REF_LOG_ONLY ) ||
( update -> flags & REF_ISPRUNING ) ||
( update -> flags & REF_UPDATE_VIA_HEAD ) )
return 0 ;
if ( strcmp ( update -> refname , head_ref ) )
return 0 ;
item = string_list_insert ( affected_refnames , "HEAD" ) ;
if ( item -> util ) {
strbuf_addf ( err ,
"multiple updates for 'HEAD' (including one "
"via its referent '%s') are not allowed" ,
```

Código

```
update -> refname ) ;
return TRANSACTION_NAME_CONFLICT ;
}
new_update = ref_transaction_add_update (
transaction , "HEAD" ,
update -> flags | REF_LOG_ONLY | REF_NODEREF ,
update -> new_oid . hash , update -> old_oid . hash ,
update -> msg ) ;
item -> util = new_update ;
return 0 ;
}
static int split_symref_update ( struct files_ref_store * refs ,
struct ref_update * update ,
```

Código

```
const char * referent ,
struct ref_transaction * transaction ,
struct string_list * affected_refnames ,
struct strbuf * err )
{
struct string_list_item * item ;
struct ref_update * new_update ;
unsigned int new_flags ;
item = string_list_insert ( affected_refnames , referent ) ;
if ( item -> util ) {
strbuf_addf ( err ,
"multiple updates for '%s' (including one "
"via symref '%s') are not allowed" ,
```

Código

```
referent , update -> refname ) ;
return TRANSACTION_NAME_CONFLICT ;
}
new_flags = update -> flags ;
if ( ! strcmp ( update -> refname , "HEAD" ) ) {
new_flags |= REF_UPDATE_VIA_HEAD ;
}
new_update = ref_transaction_add_update (
transaction , referent , new_flags ,
update -> new_oid . hash , update -> old_oid . hash ,
update -> msg ) ;
new_update -> parent_update = update ;
update -> flags |= REF_LOG_ONLY | REF_NODEREF ;
```

Código

```
update -> flags &= ~ REF_HAVE_OLD ;
item -> util = new_update ;
return 0 ;
}
static const char * original_update_refname ( struct ref_update * update )
{
while ( update -> parent_update )
update = update -> parent_update ;
return update -> refname ;
}
static int check_old_oid ( struct ref_update * update , struct object_id * oid ,
struct strbuf * err )
{
```

Código

```
if ( ! ( update -> flags & REF_HAVE_OLD ) ||
! oidcmp ( oid , & update -> old_oid ) )
return 0 ;
if ( is_null_oid ( & update -> old_oid ) )
strbuf_addf ( err , "cannot lock ref '%s': "
"reference already exists" ,
original_update_refname ( update ) ) ;
else if ( is_null_oid ( oid ) )
strbuf_addf ( err , "cannot lock ref '%s': "
"reference is missing but expected %s" ,
original_update_refname ( update ) ,
oid_to_hex ( & update -> old_oid ) ) ;
else
```

Código

```
strbuf_addf ( err , "cannot lock ref '%s': "
"is at %s but expected %s" ,
original_update_refname ( update ) ,
oid_to_hex ( oid ) ,
oid_to_hex ( & update -> old_oid ) ) ;
return - 1 ;
}
static int lock_ref_for_update ( struct files_ref_store * refs ,
struct ref_update * update ,
struct ref_transaction * transaction ,
const char * head_ref ,
struct string_list * affected_refnames ,
struct strbuf * err )
```

Código

```
{
struct strbuf referent = STRBUF_INIT ;
int mustexist = ( update -> flags & REF_HAVE_OLD ) &&
! is_null_oid ( & update -> old_oid ) ;
int ret ;
struct ref_lock * lock ;
files_assert_main_repository ( refs , "lock_ref_for_update" ) ;
if ( ( update -> flags & REF_HAVE_NEW ) && is_null_oid ( & update -> new_oid ) )
update -> flags |= REF_DELETING ;
if ( head_ref ) {
ret = split_head_update ( update , transaction , head_ref ,
affected_refnames , err ) ;
if ( ret )
```

Código

```
return ret ;
}
ret = lock_raw_ref ( refs , update -> refname , mustexist ,
affected_refnames , NULL ,
& lock , & referent ,
& update -> type , err ) ;
if ( ret ) {
char * reason ;
reason = strbuf_detach ( err , NULL ) ;
strbuf_addf ( err , "cannot lock ref '%s': %s" ,
original_update_refname ( update ) , reason ) ;
free ( reason ) ;
return ret ;
```

Código

```
}
update -> backend_data = lock ;
if ( update -> type & REF_ISSYMREF ) {
if ( update -> flags & REF_NODEREF ) {
if ( refs_read_ref_full ( & refs -> base ,
referent . buf , 0 ,
lock -> old_oid . hash , NULL ) ) {
if ( update -> flags & REF_HAVE_OLD ) {
strbuf_addf ( err , "cannot lock ref '%s': "
"error reading reference" ,
original_update_refname ( update ) ) ;
return - 1 ;
}
```

Código

```
} else if ( check_old_oid ( update , & lock -> old_oid , err ) ) {
return TRANSACTION_GENERIC_ERROR ;
}
} else {
ret = split_symref_update ( refs , update ,
referent . buf , transaction ,
affected_refnames , err ) ;
if ( ret )
return ret ;
}
} else {
struct ref_update * parent_update ;
if ( check_old_oid ( update , & lock -> old_oid , err ) )
```

Código

```
return TRANSACTION_GENERIC_ERROR ;
for ( parent_update = update -> parent_update ;
parent_update ;
parent_update = parent_update -> parent_update ) {
struct ref_lock * parent_lock = parent_update -> backend_data ;
oidcpy ( & parent_lock -> old_oid , & lock -> old_oid ) ;
}
}
if ( ( update -> flags & REF_HAVE_NEW ) &&
! ( update -> flags & REF_DELETING ) &&
! ( update -> flags & REF_LOG_ONLY ) ) {
if ( ! ( update -> type & REF_ISSYMREF ) &&
! oidcmp ( & lock -> old_oid , & update -> new_oid ) ) {
```

Código

```
} else if ( write_ref_to_lockfile ( lock , & update -> new_oid ,
err ) ) {
char * write_err = strbuf_detach ( err , NULL ) ;
update -> backend_data = NULL ;
strbuf_addf ( err ,
"cannot update ref '%s': %s" ,
update -> refname , write_err ) ;
free ( write_err ) ;
return TRANSACTION_GENERIC_ERROR ;
} else {
update -> flags |= REF_NEEDS_COMMIT ;
}
}
```

Código

```
if ( ! ( update -> flags & REF_NEEDS_COMMIT ) ) {
if ( close_ref ( lock ) ) {
strbuf_addf ( err , "couldn't close '%s.lock'" ,
update -> refname ) ;
return TRANSACTION_GENERIC_ERROR ;
}
}
return 0 ;
}
static int files_transaction_commit ( struct ref_store * ref_store ,
struct ref_transaction * transaction ,
struct strbuf * err )
{
```

Código

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE ,
"ref_transaction_commit" ) ;
int ret = 0 , i ;
struct string_list refs_to_delete = STRING_LIST_INIT_NODUP ;
struct string_list_item * ref_to_delete ;
struct string_list affected_refnames = STRING_LIST_INIT_NODUP ;
char * head_ref = NULL ;
int head_type ;
struct object_id head_oid ;
struct strbuf sb = STRBUF_INIT ;
assert ( err ) ;
if ( transaction -> state != REF_TRANSACTION_OPEN )
```

Código

```
die ( "BUG: commit called for transaction that is not open" ) ;
if ( ! transaction -> nr ) {
transaction -> state = REF_TRANSACTION_CLOSED ;
return 0 ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct string_list_item * item =
string_list_append ( & affected_refnames , update -> refname ) ;
item -> util = update ;
}
string_list_sort ( & affected_refnames ) ;
if ( ref_update_reject_duplicates ( & affected_refnames , err ) ) {
```

Código

```
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
head_ref = refs_resolve_refdup ( ref_store , "HEAD" ,
RESOLVE_REF_NO_RECURSE ,
head_oid . hash , & head_type ) ;
if ( head_ref && ! ( head_type & REF_ISSYMREF ) ) {
free ( head_ref ) ;
head_ref = NULL ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
ret = lock_ref_for_update ( refs , update , transaction ,
```

Código

```
head_ref , & affected_refnames , err ) ;
if ( ret )
goto cleanup ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
if ( update -> flags & REF_NEEDS_COMMIT ||
update -> flags & REF_LOG_ONLY ) {
if ( files_log_ref_write ( refs ,
lock -> ref_name ,
& lock -> old_oid ,
& update -> new_oid ,
```

Código

```
update -> msg , update -> flags ,
err ) ) {
char * old_msg = strbuf_detach ( err , NULL ) ;
strbuf_addf ( err , "cannot update the ref '%s': %s" ,
lock -> ref_name , old_msg ) ;
free ( old_msg ) ;
unlock_ref ( lock ) ;
update -> backend_data = NULL ;
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
}
if ( update -> flags & REF_NEEDS_COMMIT ) {
```

Código

```
clear_loose_ref_cache ( refs ) ;
if ( commit_ref ( lock ) ) {
strbuf_addf ( err , "couldn't set '%s'" , lock -> ref_name ) ;
unlock_ref ( lock ) ;
update -> backend_data = NULL ;
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
}
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
```

Código

```
if ( update -> flags & REF_DELETING &&
! ( update -> flags & REF_LOG_ONLY ) ) {
if ( ! ( update -> type & REF_ISPACKED ) ||
update -> type & REF_ISSYMREF ) {
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , lock -> ref_name ) ;
if ( unlink_or_msg ( sb . buf , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
update -> flags |= REF_DELETED_LOOSE ;
}
if ( ! ( update -> flags & REF_ISPRUNING ) )
```

Código

```
string_list_append ( & refs_to_delete ,
lock -> ref_name ) ;
}
}
if ( repack_without_refs ( refs , & refs_to_delete , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
for_each_string_list_item ( ref_to_delete , & refs_to_delete ) {
strbuf_reset ( & sb ) ;
files_reflog_path ( refs , & sb , ref_to_delete -> string ) ;
if ( ! unlink_or_warn ( sb . buf ) )
try_remove_empty_parents ( refs , ref_to_delete -> string ,
```

Código

```
REMOVE_EMPTY_PARENTS_REFLOG ) ;
}
clear_loose_ref_cache ( refs ) ;
cleanup :
strbuf_release ( & sb ) ;
transaction -> state = REF_TRANSACTION_CLOSED ;
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
if ( lock )
unlock_ref ( lock ) ;
if ( update -> flags & REF_DELETED_LOOSE ) {
try_remove_empty_parents ( refs , update -> refname ,
```

Código

```
REMOVE_EMPTY_PARENTS_REF ) ;
}
}
string_list_clear ( & refs_to_delete , 0 ) ;
free ( head_ref ) ;
string_list_clear ( & affected_refnames , 0 ) ;
return ret ;
}
static int ref_present ( const char * refname ,
const struct object_id * oid , int flags , void * cb_data )
{
struct string_list * affected_refnames = cb_data ;
return string_list_has_string ( affected_refnames , refname ) ;
```

Código

```
}
static int files_initial_transaction_commit ( struct ref_store * ref_store ,
struct ref_transaction * transaction ,
struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE ,
"initial_ref_transaction_commit" ) ;
int ret = 0 , i ;
struct string_list affected_refnames = STRING_LIST_INIT_NODUP ;
assert ( err ) ;
if ( transaction -> state != REF_TRANSACTION_OPEN )
die ( "BUG: commit called for transaction that is not open" ) ;
```

Código

```
for ( i = 0 ; i < transaction -> nr ; i ++ )
string_list_append ( & affected_refnames ,
transaction -> updates [ i ] -> refname ) ;
string_list_sort ( & affected_refnames ) ;
if ( ref_update_reject_duplicates ( & affected_refnames , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
if ( refs_for_each_rawref ( & refs -> base , ref_present ,
& affected_refnames ) )
die ( "BUG: initial ref transaction called with existing refs" ) ;
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
```

Código

```
if ( ( update -> flags & REF_HAVE_OLD ) &&
! is_null_oid ( & update -> old_oid ) )
die ( "BUG: initial ref transaction with old_sha1 set" ) ;
if ( refs_verify_refname_available ( & refs -> base , update -> refname ,
& affected_refnames , NULL ,
err ) ) {
ret = TRANSACTION_NAME_CONFLICT ;
goto cleanup ;
}
}
if ( lock_packed_refs ( refs , 0 ) ) {
strbuf_addf ( err , "unable to lock packed-refs file: %s" ,
strerror ( errno ) ) ;
```

Código

```
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
if ( ( update -> flags & REF_HAVE_NEW ) &&
! is_null_oid ( & update -> new_oid ) )
add_packed_ref ( refs , update -> refname ,
& update -> new_oid ) ;
}
if ( commit_packed_refs ( refs ) ) {
strbuf_addf ( err , "unable to commit packed-refs file: %s" ,
strerror ( errno ) ) ;
```

Código

```
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
cleanup :
transaction -> state = REF_TRANSACTION_CLOSED ;
string_list_clear ( & affected_refnames , 0 ) ;
return ret ;
}
struct expire_reflog_cb {
unsigned int flags ;
reflog_expiry_should_prune_fn * should_prune_fn ;
void * policy_cb ;
FILE * newlog ;
```

Código

```
struct object_id last_kept_oid ;
} ;
static int expire_reflog_ent ( struct object_id * ooid , struct object_id * noid ,
const char * email , timestamp_t timestamp , int tz ,
const char * message , void * cb_data )
{
struct expire_reflog_cb * cb = cb_data ;
struct expire_reflog_policy_cb * policy_cb = cb -> policy_cb ;
if ( cb -> flags & EXPIRE_REFLOGS_REWRITE )
ooid = & cb -> last_kept_oid ;
if ( ( * cb -> should_prune_fn ) ( ooid , noid , email , timestamp , tz ,
message , policy_cb ) ) {
if ( ! cb -> newlog )
```

Código

```
printf ( "would prune %s" , message ) ;
else if ( cb -> flags & EXPIRE_REFLOGS_VERBOSE )
printf ( "prune %s" , message ) ;
} else {
if ( cb -> newlog ) {
fprintf ( cb -> newlog , "%s %s %s %" PRItime " %+05d\t%s" ,
oid_to_hex ( ooid ) , oid_to_hex ( noid ) ,
email , timestamp , tz , message ) ;
oidcpy ( & cb -> last_kept_oid , noid ) ;
}
if ( cb -> flags & EXPIRE_REFLOGS_VERBOSE )
printf ( "keep %s" , message ) ;
}
```

Código

```
return 0 ;
}
static int files_reflog_expire ( struct ref_store * ref_store ,
const char * refname , const unsigned char * sha1 ,
unsigned int flags ,
reflog_expiry_prepare_fn prepare_fn ,
reflog_expiry_should_prune_fn should_prune_fn ,
reflog_expiry_cleanup_fn cleanup_fn ,
void * policy_cb_data )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "reflog_expire" ) ;
static struct lock_file reflog_lock ;
```

Código

```
struct expire_reflog_cb cb ;
struct ref_lock * lock ;
struct strbuf log_file_sb = STRBUF_INIT ;
char * log_file ;
int status = 0 ;
int type ;
struct strbuf err = STRBUF_INIT ;
struct object_id oid ;
memset ( & cb , 0 , sizeof ( cb ) ) ;
cb . flags = flags ;
cb . policy_cb = policy_cb_data ;
cb . should_prune_fn = should_prune_fn ;
lock = lock_ref_sha1_basic ( refs , refname , sha1 ,
```

Código

```
NULL , NULL , REF_NODEREF ,
& type , & err ) ;
if ( ! lock ) {
error ( "cannot lock ref '%s': %s" , refname , err . buf ) ;
strbuf_release ( & err ) ;
return - 1 ;
}
if ( ! refs_reflog_exists ( ref_store , refname ) ) {
unlock_ref ( lock ) ;
return 0 ;
}
files_reflog_path ( refs , & log_file_sb , refname ) ;
log_file = strbuf_detach ( & log_file_sb , NULL ) ;
```

Código

```
if ( ! ( flags & EXPIRE_REFLOGS_DRY_RUN ) ) {
if ( hold_lock_file_for_update ( & reflog_lock , log_file , 0 ) < 0 ) {
struct strbuf err = STRBUF_INIT ;
unable_to_lock_message ( log_file , errno , & err ) ;
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
goto failure ;
}
cb . newlog = fdopen_lock_file ( & reflog_lock , "w" ) ;
if ( ! cb . newlog ) {
error ( "cannot fdopen %s (%s)" ,
get_lock_file_path ( & reflog_lock ) , strerror ( errno ) ) ;
goto failure ;
```

Código

```
}
}
hashcpy ( oid . hash , sha1 ) ;
( * prepare_fn ) ( refname , & oid , cb . policy_cb ) ;
refs_for_each_reflog_ent ( ref_store , refname , expire_reflog_ent , & cb ) ;
( * cleanup_fn ) ( cb . policy_cb ) ;
if ( ! ( flags & EXPIRE_REFLOGS_DRY_RUN ) ) {
int update = ( flags & EXPIRE_REFLOGS_UPDATE_REF ) &&
! ( type & REF_ISSYMREF ) &&
! is_null_oid ( & cb . last_kept_oid ) ;
if ( close_lock_file ( & reflog_lock ) ) {
status |= error ( "couldn't write %s: %s" , log_file ,
strerror ( errno ) ) ;
```

Código

```
} else if ( update &&
( write_in_full ( get_lock_file_fd ( lock -> lk ) ,
oid_to_hex ( & cb . last_kept_oid ) , GIT_SHA1_HEXSZ ) != GIT_SHA1_HEXSZ ||
write_str_in_full ( get_lock_file_fd ( lock -> lk ) , "\n" ) != 1 ||
close_ref ( lock ) < 0 ) ) {
status |= error ( "couldn't write %s" ,
get_lock_file_path ( lock -> lk ) ) ;
rollback_lock_file ( & reflog_lock ) ;
} else if ( commit_lock_file ( & reflog_lock ) ) {
status |= error ( "unable to write reflog '%s' (%s)" ,
log_file , strerror ( errno ) ) ;
} else if ( update && commit_ref ( lock ) ) {
status |= error ( "couldn't set %s" , lock -> ref_name ) ;
```

Código

```
}
}
free ( log_file ) ;
unlock_ref ( lock )
return status ;
failure :
rollback_lock_file ( & reflog_lock ) ;
free ( log_file ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
static int files_init_db ( struct ref_store * ref_store , struct strbuf * err )
{
```

Código

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "init_db" ) ;
struct strbuf sb = STRBUF_INIT ;
files_ref_path ( refs , & sb , "refs/heads" )
safe_create_dir ( sb . buf , 1 ) ;
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , "refs/tags" ) ;
safe_create_dir ( sb . buf , 1 ) ;
strbuf_release ( & sb ) ;
return 0
}
struct ref_storage_be refs_be_files = {
NULL ,
```

Código

```
"files" ,
files_ref_store_create ,
files_init_db ,
files_transaction_commit ,
files_initial_transaction_commit ,
files_pack_refs ,
files_peel_ref ,
files_create_symref ,
files_delete_refs ,
files_rename_ref ,
files_ref_iterator_begin ,
files_read_raw_ref ,
files_reflog_iterator_begin ,
```

Código

```
files_for_each_reflog_ent ,
files_for_each_reflog_ent_reverse ,
files_reflog_exists ,
files_create_reflog ,
files_delete_reflog ,
```

Código con los Errores

```
struct ref_lock {
char * ref_name ;
struct lock_file * lk
/*Pruebas/error.c:4:6 syntax error, unexpected STRUCT*/
struct object_id old_oid ;
/*Pruebas/error.c:5:1 syntax error, unexpected RIGHT_BRACKET*/
} ;
static int ref_resolves_to_object ( const char * refname ,
const struct object_id * oid ,
unsigned int flags )
{
if ( flags & REF_ISBROKEN )
```

Código con los Errores

```
return 0 ;
if ( ! has_sha1_file ( oid -> hash ) ) {
error ( "%s does not point to a valid object!" , refname ) ;
return 0 ;
}
return 1 ;
struct packed_ref_cache {
struct ref_cache * cache ;
unsigned int referrers ;
struct lock_file * lock ;
struct stat_validity validity ;
struct files_ref_store {
struct ref_store base ;
```

Código con los Errores

```
unsigned int store_flags
/*Pruebas/error.c:25:4 syntax error, unexpected CHAR*/
char * gitdir ;
char * gitcommondir ;
char * packed_refs_path ;
struct ref_cache * loose ;
struct packed_ref_cache * packed ;
/*Pruebas/error.c:30:1 syntax error, unexpected RIGHT_BRACKET*/
} ;
static struct lock_file packlock ;
static void acquire_packed_ref_cache ( struct packed_ref_cache * packed_refs )
{
packed_refs -> referrers ++ ;
```

Código con los Errores

```
static int release_packed_ref_cache ( struct packed_ref_cache * packed_refs )
/*Pruebas/error.c:36:2 syntax error, unexpected IF*/
/*Pruebas/error.c:36:6 syntax error, unexpected EXCLAMATION*/
/*Pruebas/error.c:36:9 syntax error, unexpected DEC_OP*/
/*Pruebas/error.c:36:36 syntax error, unexpected RIGHT_PARENTHESIS*/
if ( ! -- packed_refs -> referrers ) {
free_ref_cache ( packed_refs -> cache ) ;
stat_validity_clear ( & packed_refs -> validity ) ;
free ( packed_refs ) ;
return 1 ;
/*Pruebas/error.c:41:6 syntax error, unexpected ELSE*/
} else {
return 0 ;
```

Código con los Errores

```
}
/*Pruebas/error.c:44:1 syntax error, unexpected RIGHT_BRACKET*/
}
static void clear_packed_ref_cache ( struct files_ref_store * refs )
{
if ( refs -> packed ) {
struct packed_ref_cache * packed_refs = refs -> packed ;
if ( packed_refs -> lock )
die ( "internal error: packed-ref cache cleared while locked" ) ;
refs -> packed = NULL ;
release_packed_ref_cache ( packed_refs ) ;
}
}
```

Código con los Errores

```
static void clear_loose_ref_cache ( struct files_ref_store * refs )
{
if ( refs -> loose ) {
free_ref_cache ( refs -> loose ) ;
refs -> loose = NULL ;
}
}
static struct ref_store * files_ref_store_create ( const char * gitdir ,
unsigned int flags )
{
struct files_ref_store * refs = xcalloc ( 1 , sizeof ( * refs ) ) ;
struct ref_store * ref_store = ( struct ref_store * ) refs ;
struct strbuf sb = STRBUF_INIT ;
```

Código con los Errores

```
base_ref_store_init ( ref_store , & refs_be_files ) ;
refs -> store_flags = flags ;
refs -> gitdir = xstrdup ( gitdir ) ;
get_common_dir_noenv ( & sb , gitdir ) ;
refs -> gitcommondir = strbuf_detach ( & sb , NULL ) ;
strbuf_addf ( & sb , "%s/packed-refs" , refs -> gitcommondir ) ;
refs -> packed_refs_path = strbuf_detach ( & sb , NULL ) ;
return ref_store ;
}
static void files_assert_main_repository ( struct files_ref_store * refs ,
const char * caller )
{
if ( refs -> store_flags & REF_STORE_MAIN )
```

Código con los Errores

```
return ;
die ( "BUG: operation %s only allowed for main ref store" , caller ) ;
}
static struct files_ref_store * files_downcast ( struct ref_store * ref_store ,
unsigned int required_flags ,
const char * caller )
{
struct files_ref_store * refs ;
if ( ref_store -> be != & refs_be_files )
die ( "BUG: ref_store is type \"%s\" not \"files\" in %s" ,
ref_store -> be -> name , caller ) ;
refs = ( struct files_ref_store * ) ref_store ;
if ( ( refs -> store_flags & required_flags ) != required_flags )
```

Código con los Errores

```
die ( "BUG: operation %s requires abilities 0x%x, but only have 0x%x" ,
caller , required_flags , refs -> store_flags ) ;
return refs ;
}
static const char PACKED_REFS_HEADER [ ] =
"# pack-refs with: peeled fully-peeled \n" ;
static const char * parse_ref_line ( struct strbuf * line , struct object_id * oid )
{
const char * ref ;
if ( parse_oid_hex ( line -> buf , oid , & ref ) < 0 )
return NULL ;
if ( ! isspace ( * ref ++ ) )
return NULL ;
```

Código con los Errores

```
if ( isspace ( * ref ) )
return NULL ;
if ( line -> buf [ line -> len - 1 ] != '\n' )
return NULL ;
line -> buf [ -- line -> len ] = 0 ;
return ref ;
}
static void read_packed_refs ( FILE * f , struct ref_dir * dir )
{
struct ref_entry * last = NULL ;
struct strbuf line = STRBUF_INIT ;
enum { PEELED_NONE , PEELED_TAGS , PEELED_FULLY } peeled = PEELED_NONE ;
while ( strbuf_getwholeline ( & line , f , '\n' ) != EOF ) {
```

Código con los Errores

```
struct object_id oid ;
const char * refname ;
const char * traits ;
if ( skip_prefix ( line . buf , "# pack-refs with:" , & traits ) ) {
if ( strstr ( traits , " fully-peeled " ) )
peeled = PEELED_FULLY ;
else if ( strstr ( traits , " peeled " ) )
peeled = PEELED_TAGS ;
continue ;
}
refname = parse_ref_line ( & line , & oid ) ;
if ( refname ) {
int flag = REF_ISPACKED ;
```

Código con los Errores

```
if ( check_refname_format ( refname , REFNAME_ALLOW_ONELEVEL ) ) {
if ( ! refname_is_safe ( refname ) )
die ( "packed refname is dangerous: %s" , refname ) ;
oidclr ( & oid ) ;
flag |= REF_BAD_NAME | REF_ISBROKEN ;
}
last = create_ref_entry ( refname , & oid , flag , 0 ) ;
if ( peeled == PEELED_FULLY ||
( peeled == PEELED_TAGS && starts_with ( refname , "refs/tags/" ) ) )
last -> flag |= REF_KNOWS_PEELED ;
add_ref_entry ( dir , last ) ;
continue ;
}
```

Código con los Errores

```
if ( last &&
line . buf [ 0 ] == '^' &&
line . len == 42 &&
line . buf [ 42 - 1 ] == '\n' &&
! get_oid_hex ( line . buf + 1 , & oid ) ) {
oidcpy ( & last -> u . value . peeled , & oid ) ;
last -> flag |= REF_KNOWS_PEELED ;
}
}
strbuf_release ( & line ) ;
}
static const char * files_packed_refs_path ( struct files_ref_store * refs )
{
```

Código con los Errores

```
return refs -> packed_refs_path ;
}
static void files_reflog_path ( struct files_ref_store * refs ,
struct strbuf * sb ,
const char * refname )
{
if ( ! refname ) {
strbuf_addf ( sb , "%s/logs" , refs -> gitcommondir ) ;
return ;
}
switch ( ref_type ( refname ) ) {
case REF_TYPE_PER_WORKTREE :
case REF_TYPE_PSEUDOREF :
```

Código con los Errores

```
strbuf_addf ( sb , "%s/logs/%s" , refs -> gitdir , refname ) ;
break ;
case REF_TYPE_NORMAL :
strbuf_addf ( sb , "%s/logs/%s" , refs -> gitcommondir , refname ) ;
break ;
default :
die ( "BUG: unknown ref type %d of ref %s" ,
ref_type ( refname ) , refname ) ;
}
}
static void files_ref_path ( struct files_ref_store * refs ,
struct strbuf * sb ,
const char * refname )
```

Código con los Errores

```
{
switch ( ref_type ( refname ) ) {
case REF_TYPE_PER_WORKTREE :
case REF_TYPE_PSEUDOREF :
strbuf_addf ( sb , "%s/%s" , refs -> gitdir , refname ) ;
break ;
case REF_TYPE_NORMAL :
strbuf_addf ( sb , "%s/%s" , refs -> gitcommondir , refname ) ;
break ;
default :
die ( "BUG: unknown ref type %d of ref %s" ,
ref_type ( refname ) , refname ) ;
}
```

Código con los Errores

```
}
static struct packed_ref_cache * get_packed_ref_cache ( struct files_ref_store * refs )
{
const char * packed_refs_file = files_packed_refs_path ( refs ) ;
if ( refs -> packed &&
! stat_validity_check ( & refs -> packed -> validity , packed_refs_file ) )
clear_packed_ref_cache ( refs ) ;
if ( ! refs -> packed ) {
FILE * f ;
refs -> packed = xcalloc ( 1 , sizeof ( * refs -> packed ) ) ;
acquire_packed_ref_cache ( refs -> packed ) ;
refs -> packed -> cache = create_ref_cache ( & refs -> base , NULL ) ;
refs -> packed -> cache -> root -> flag &= ~ REF_INCOMPLETE ;
```

Código con los Errores

```
f = fopen ( packed_refs_file , "r" ) ;
if ( f ) {
stat_validity_update ( & refs -> packed -> validity , fileno ( f ) ) ;
read_packed_refs ( f , get_ref_dir ( refs -> packed -> cache -> root ) ) ;
fclose ( f ) ;
}
}
return refs -> packed ;
}
static struct ref_dir * get_packed_ref_dir ( struct packed_ref_cache * packed_ref_cache )
{
return get_ref_dir ( packed_ref_cache -> cache -> root ) ;
}
```

Código con los Errores

```
static struct ref_dir * get_packed_refs ( struct files_ref_store * refs )
{
return get_packed_ref_dir ( get_packed_ref_cache ( refs ) ) ;
}
static void add_packed_ref ( struct files_ref_store * refs ,
const char * refname , const struct object_id * oid )
{
struct packed_ref_cache * packed_ref_cache = get_packed_ref_cache ( refs ) ;
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed refs not locked" ) ;
add_ref_entry ( get_packed_ref_dir ( packed_ref_cache ) ,
create_ref_entry ( refname , oid , REF_ISPACKED , 1 ) ) ;
}
```

Código con los Errores

```
static void loose_fill_ref_dir ( struct ref_store * ref_store ,
struct ref_dir * dir , const char * dirname )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "fill_ref_dir" ) ;
DIR * d ;
struct dirent * de ;
int dirnamelen = strlen ( dirname ) ;
struct strbuf refname ;
struct strbuf path = STRBUF_INIT ;
size_t path_baselen ;
files_ref_path ( refs , & path , dirname ) ;
path_baselen = path . len ;
```

Código con los Errores

```
d = opendir ( path . buf ) ;
if ( ! d ) {
strbuf_release ( & path ) ;
return ;
}
strbuf_init ( & refname , dirnamelen + 257 ) ;
strbuf_add ( & refname , dirname , dirnamelen ) ;
while ( ( de = readdir ( d ) ) != NULL ) {
struct object_id oid ;
struct stat st ;
int flag ;
if ( de -> d_name [ 0 ] == '.' )
continue ;
```

Código con los Errores

```
if ( ends_with ( de -> d_name , ".lock" ) )
continue ;
strbuf_addstr ( & refname , de -> d_name ) ;
strbuf_addstr ( & path , de -> d_name ) ;
if ( stat ( path . buf , & st ) < 0 ) {
;
} else if ( S_ISDIR ( st . st_mode ) ) {
strbuf_addch ( & refname , '/' ) ;
add_entry_to_dir ( dir ,
create_dir_entry ( dir -> cache , refname . buf ,
refname . len , 1 ) ) ;
} else {
if ( ! refs_resolve_ref_unsafe ( & refs -> base ,
```

Código con los Errores

```
refname . buf ,
RESOLVE_REF_READING ,
oid . hash , & flag ) ) {
oidclr ( & oid ) ;
flag |= REF_ISBROKEN ;
} else if ( is_null_oid ( & oid ) ) {
flag |= REF_ISBROKEN ;
}
if ( check_refname_format ( refname . buf ,
REFNAME_ALLOW_ONELEVEL ) ) {
if ( ! refname_is_safe ( refname . buf ) )
die ( "loose refname is dangerous: %s" , refname . buf ) ;
oidclr ( & oid ) ;
```

Código con los Errores

```
flag |= REF_BAD_NAME | REF_ISBROKEN ;
}
add_entry_to_dir ( dir ,
create_ref_entry ( refname . buf , & oid , flag , 0 ) ) ;
}
strbuf_setlen ( & refname , dirnamelen ) ;
strbuf_setlen ( & path , path_baselen ) ;
}
strbuf_release ( & refname ) ;
strbuf_release ( & path ) ;
closedir ( d ) ;
if ( ! strcmp ( dirname , "refs/" ) ) {
int pos = search_ref_dir ( dir , "refs/bisect/" , 12 ) ;
```

Código con los Errores

```
if ( pos < 0 ) {
struct ref_entry * child_entry = create_dir_entry (
dir -> cache , "refs/bisect/" , 12 , 1 ) ;
add_entry_to_dir ( dir , child_entry ) ;
}
}
}
static struct ref_cache * get_loose_ref_cache ( struct files_ref_store * refs )
{
if ( ! refs -> loose ) {
refs -> loose = create_ref_cache ( & refs -> base , loose_fill_ref_dir ) ;
refs -> loose -> root -> flag &= ~ REF_INCOMPLETE ;
add_entry_to_dir ( get_ref_dir ( refs -> loose -> root ) ,
```

Código con los Errores

```
create_dir_entry ( refs -> loose , "refs/" , 5 , 1 ) ) ;
}
return refs -> loose ;
}
static struct ref_entry * get_packed_ref ( struct files_ref_store * refs ,
const char * refname )
{
return find_ref_entry ( get_packed_refs ( refs ) , refname ) ;
}
static int resolve_packed_ref ( struct files_ref_store * refs ,
const char * refname ,
unsigned char * sha1 , unsigned int * flags )
{
```

Código con los Errores

```
struct ref_entry * entry ;
entry = get_packed_ref ( refs , refname ) ;
if ( entry ) {
hashcpy ( sha1 , entry -> u . value . oid . hash ) ;
* flags |= REF_ISPACKED ;
return 0 ;
}
return - 1 ;
}
static int files_read_raw_ref ( struct ref_store * ref_store ,
const char * refname , unsigned char * sha1 ,
struct strbuf * referent , unsigned int * type )
{
```

Código con los Errores

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "read_raw_ref" ) ;
struct strbuf sb_contents = STRBUF_INIT ;
struct strbuf sb_path = STRBUF_INIT ;
const char * path ;
const char * buf ;
struct stat st ;
int fd ;
int ret = - 1 ;
int save_errno ;
int remaining_retries = 3 ;
* type = 0 ;
strbuf_reset ( & sb_path ) ;
```

Código con los Errores

```
files_ref_path ( refs , & sb_path , refname ) ;
path = sb_path . buf ;
stat_ref :
if ( remaining_retries -- <= 0 )
goto out ;
if ( lstat ( path , & st ) < 0 ) {
if ( errno != ENOENT )
goto out ;
if ( resolve_packed_ref ( refs , refname , sha1 , type ) ) {
errno = ENOENT ;
goto out ;
}
ret = 0 ;
```

Código con los Errores

```
goto out ;
}
if ( S_ISLNK ( st . st_mode ) ) {
strbuf_reset ( & sb_contents ) ;
if ( strbuf_readlink ( & sb_contents , path , 0 ) < 0 ) {
if ( errno == ENOENT || errno == EINVAL )
goto stat_ref ;
else
goto out ;
}
if ( starts_with ( sb_contents . buf , "refs/" ) &&
! check_refname_format ( sb_contents . buf , 0 ) ) {
strbuf_swap ( & sb_contents , referent ) ;
```

Código con los Errores

```
* type |= REF_ISSYMREF ;
ret = 0 ;
goto out ;
}
}
if ( S_ISDIR ( st . st_mode ) ) {
if ( resolve_packed_ref ( refs , refname , sha1 , type ) ) {
errno = EISDIR ;
goto out ;
}
ret = 0 ;
goto out ;
}
```

Código con los Errores

```
fd = open ( path , O_RDONLY ) ;
if ( fd < 0 ) {
if ( errno == ENOENT && ! S_ISLNK ( st . st_mode ) )
goto stat_ref ;
else
goto out ;
}
strbuf_reset ( & sb_contents ) ;
if ( strbuf_read ( & sb_contents , fd , 256 ) < 0 ) {
int save_errno = errno ;
close ( fd ) ;
errno = save_errno ;
goto out ;
```

Código con los Errores

```
}
close ( fd ) ;
strbuf_rtrim ( & sb_contents ) ;
buf = sb_contents . buf ;
if ( starts_with ( buf , "ref:" ) ) {
buf += 4 ;
while ( isspace ( * buf ) )
buf ++ ;
strbuf_reset ( referent ) ;
strbuf_addstr ( referent , buf ) ;
* type |= REF_ISSYMREF ;
ret = 0 ;
goto out ;
```

Código con los Errores

```
}
if ( get_sha1_hex ( buf , sha1 ) ||
( buf [ 40 ] != '\0' && ! isspace ( buf [ 40 ] ) ) ) {
* type |= REF_ISBROKEN ;
errno = EINVAL ;
goto out ;
}
ret = 0 ;
out :
save_errno = errno ;
strbuf_release ( & sb_path ) ;
strbuf_release ( & sb_contents ) ;
errno = save_errno ;
```

Código con los Errores

```
return ret ;
}
static void unlock_ref ( struct ref_lock * lock )
{
if ( lock -> lk )
rollback_lock_file ( lock -> lk ) ;
free ( lock -> ref_name ) ;
free ( lock ) ;
}
static int lock_raw_ref ( struct files_ref_store * refs ,
const char * refname , int mustexist ,
const struct string_list * extras ,
const struct string_list * skip ,
```

Código con los Errores

```
struct ref_lock * * lock_p ,
struct strbuf * referent ,
unsigned int * type ,
struct strbuf * err )
{
struct ref_lock * lock ;
struct strbuf ref_file = STRBUF_INIT ;
int attempts_remaining = 3 ;
int ret = TRANSACTION_GENERIC_ERROR ;
assert ( err ) ;
files_assert_main_repository ( refs , "lock_raw_ref" ) ;
* type = 0 ;
* lock_p = lock = xcalloc ( 1 , sizeof ( * lock ) ) ;
```

Código con los Errores

```
lock -> ref_name = xstrdup ( refname ) ;
files_ref_path ( refs , & ref_file , refname ) ;
retry :
switch ( safe_create_leading_directories ( ref_file . buf ) ) {
case SCLD_OK :
break ;
case SCLD_EXISTS :
if ( refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) ) {
if ( mustexist ) {
strbuf_reset ( err ) ;
strbuf_addf ( err , "unable to resolve reference '%s'" ,
refname ) ;
```

Código con los Errores

```
} else {
ret = TRANSACTION_NAME_CONFLICT ;
}
} else {
strbuf_addf ( err , "unable to create lock file %s.lock; "
"non-directory in the way" ,
ref_file . buf ) ;
}
goto error_return ;
case SCLD_VANISHED :
if ( -- attempts_remaining > 0 )
goto retry ;
default :
```

Código con los Errores

```
strbuf_addf ( err , "unable to create directory for %s" ,
ref_file . buf ) ;
goto error_return ;
}
if ( ! lock -> lk )
lock -> lk = xcalloc ( 1 , sizeof ( struct lock_file ) ) ;
if ( hold_lock_file_for_update ( lock -> lk , ref_file . buf , LOCK_NO_DEREF ) < 0 ) {
if ( errno == ENOENT && -- attempts_remaining > 0 ) {
goto retry ;
} else {
unable_to_lock_message ( ref_file . buf , errno , err ) ;
goto error_return ;
}
```

Código con los Errores

```
}
if ( files_read_raw_ref ( & refs -> base , refname ,
lock -> old_oid . hash , referent , type ) ) {
if ( errno == ENOENT ) {
if ( mustexist ) {
strbuf_addf ( err , "unable to resolve reference ’%s’" ,
refname ) ;
goto error_return ;
} else {
}
} else if ( errno == EISDIR ) {
if ( mustexist ) {
strbuf_addf ( err , "unable to resolve reference ’%s’" ,
```

Código con los Errores

```
refname ) ;
goto error_return ;
} else if ( remove_dir_recursively ( & ref_file ,
REMOVE_DIR_EMPTY_ONLY ) ) {
if ( refs_verify_refname_available (
& refs -> base , refname ,
extras , skip , err ) ) {
ret = TRANSACTION_NAME_CONFLICT ;
goto error_return ;
} else {
strbuf_addf ( err , "there is a non-empty directory '%s' "
"blocking reference '%s'" ,
ref_file . buf , refname ) ;
```

Código con los Errores

```
goto error_return ;
}
}
} else if ( errno == EINVAL && ( * type & REF_ISBROKEN ) ) {
strbuf_addf ( err , "unable to resolve reference '%s': "
"reference broken" , refname ) ;
goto error_return ;
} else {
strbuf_addf ( err , "unable to resolve reference '%s': %s" ,
refname , strerror ( errno ) ) ;
goto error_return ;
}
if ( refs_verify_refname_available (
```

Código con los Errores

```
& refs -> base , refname ,
extras , skip , err ) )
goto error_return ;
}
ret = 0 ;
goto out ;
error_return :
unlock_ref ( lock ) ;
* lock_p = NULL ;
out :
strbuf_release ( & ref_file ) ;
return ret ;
}
```

Código con los Errores

```
static int files_peel_ref ( struct ref_store * ref_store ,
const char * refname , unsigned char * sha1 )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ | REF_STORE_ODB ,
"peel_ref" ) ;
int flag ;
unsigned char base [ 20 ] ;
if ( current_ref_iter && current_ref_iter -> refname == refname ) {
struct object_id peeled ;
if ( ref_iterator_peel ( current_ref_iter , & peeled ) )
return - 1 ;
hashcpy ( sha1 , peeled . hash ) ;
```

Código con los Errores

```
return 0 ;
}
if ( refs_read_ref_full ( ref_store , refname ,
RESOLVE_REF_READING , base , & flag ) )
return - 1 ;
if ( flag & REF_ISPACKED ) {
struct ref_entry * r = get_packed_ref ( refs , refname ) ;
if ( r ) {
if ( peel_entry ( r , 0 ) )
return - 1 ;
hashcpy ( sha1 , r -> u . value . peeled . hash ) ;
return 0 ;
}
```

Código con los Errores

```
}
return peel_object ( base , sha1 ) ;
}
struct files_ref_iterator {
struct ref_iterator base ;
struct packed_ref_cache * packed_ref_cache ;
struct ref_iterator * iter0 ;
unsigned int flags ;
} ;
static int files_ref_iterator_advance ( struct ref_iterator * ref_iterator )
{
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
```

Código con los Errores

```
int ok ;
while ( ( ok = ref_iterator_advance ( iter -> iter0 ) ) == ITER_OK ) {
if ( iter -> flags & DO_FOR_EACH_PER_WORKTREE_ONLY &&
ref_type ( iter -> iter0 -> refname ) != REF_TYPE_PER_WORKTREE )
continue ;
if ( ! ( iter -> flags & DO_FOR_EACH_INCLUDE_BROKEN ) &&
! ref_resolves_to_object ( iter -> iter0 -> refname ,
iter -> iter0 -> oid ,
iter -> iter0 -> flags ) )
continue ;
iter -> base . refname = iter -> iter0 -> refname ;
iter -> base . oid = iter -> iter0 -> oid ;
iter -> base . flags = iter -> iter0 -> flags ;
```

Código con los Errores

```
return ITER_OK ;
}
iter -> iter0 = NULL ;
if ( ref_iterator_abort ( ref_iterator ) != ITER_DONE )
ok = ITER_ERROR ;
return ok ;
}
static int files_ref_iterator_peel ( struct ref_iterator * ref_iterator ,
struct object_id * peeled )
{
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
return ref_iterator_peel ( iter -> iter0 , peeled ) ;
```

Código con los Errores

```
}
static int files_ref_iterator_abort ( struct ref_iterator * ref_iterator )
{
struct files_ref_iterator * iter =
( struct files_ref_iterator * ) ref_iterator ;
int ok = ITER_DONE ;
if ( iter -> iter0 )
ok = ref_iterator_abort ( iter -> iter0 ) ;
release_packed_ref_cache ( iter -> packed_ref_cache ) ;
base_ref_iterator_free ( ref_iterator ) ;
return ok ;
}
static struct ref_iterator_vtable files_ref_iterator_vtable = {
```

Código con los Errores

```
files_ref_iterator_advance ,
files_ref_iterator_peel ,
files_ref_iterator_abort
} ;
static struct ref_iterator * files_ref_iterator_begin (
struct ref_store * ref_store ,
const char * prefix , unsigned int flags )
{
struct files_ref_store * refs ;
struct ref_iterator * loose_iter , * packed_iter ;
struct files_ref_iterator * iter ;
struct ref_iterator * ref_iterator ;
if ( ref_paranoia < 0 )
```

Código con los Errores

```
ref_paranoia = git_env_bool ( "GIT_REF_PARANOIA" , 0 ) ;
if ( ref_paranoia )
flags |= DO_FOR_EACH_INCLUDE_BROKEN ;
refs = files_downcast ( ref_store ,
REF_STORE_READ | ( ref_paranoia ? 0 : REF_STORE_ODB ) ,
"ref_iterator_begin" ) ;
iter = xcalloc ( 1 , sizeof ( * iter ) ) ;
ref_iterator = & iter -> base ;
base_ref_iterator_init ( ref_iterator , & files_ref_iterator_vtable ) ;
loose_iter = cache_ref_iterator_begin ( get_loose_ref_cache ( refs ) ,
prefix , 1 ) ;
iter -> packed_ref_cache = get_packed_ref_cache ( refs ) ;
acquire_packed_ref_cache ( iter -> packed_ref_cache ) ;
```

Código con los Errores

```
packed_iter = cache_ref_iterator_begin ( iter -> packed_ref_cache -> cache ,
prefix , 0 ) ;
iter -> iter0 = overlay_ref_iterator_begin ( loose_iter , packed_iter ) ;
iter -> flags = flags ;
return ref_iterator ;
}
static int verify_lock ( struct ref_store * ref_store , struct ref_lock * lock ,
const unsigned char * old_sha1 , int mustexist ,
struct strbuf * err )
{
assert ( err ) ;
if ( refs_read_ref_full ( ref_store , lock -> ref_name ,
mustexist ? RESOLVE_REF_READING : 0 ,
```

Código con los Errores

```
lock -> old_oid . hash , NULL ) ) {
if ( old_sha1 ) {
int save_errno = errno ;
strbuf_addf ( err , "can't verify ref '%s'" , lock -> ref_name ) ;
errno = save_errno ;
return - 1 ;
} else {
oidclr ( & lock -> old_oid ) ;
return 0 ;
}
}
if ( old_sha1 && hashcmp ( lock -> old_oid . hash , old_sha1 ) ) {
strbuf_addf ( err , "ref '%s' is at %s but expected %s" ,
```

Código con los Errores

```
lock -> ref_name ,
oid_to_hex ( & lock -> old_oid ) ,
sha1_to_hex ( old_sha1 ) ) ;
errno = EBUSY ;
return - 1 ;
}
return 0 ;
}
static int remove_empty_directories ( struct strbuf * path )
{
return remove_dir_recursively ( path , REMOVE_DIR_EMPTY_ONLY ) ;
}
static int create_reflock ( const char * path , void * cb )
```

Código con los Errores

```
{
struct lock_file * lk = cb ;
return hold_lock_file_for_update ( lk , path , LOCK_NO_DEREF ) < 0 ? - 1 : 0 ;
}
static struct ref_lock * lock_ref_sha1_basic ( struct files_ref_store * refs ,
const char * refname ,
const unsigned char * old_sha1 ,
const struct string_list * extras ,
const struct string_list * skip ,
unsigned int flags , int * type ,
struct strbuf * err )
{
struct strbuf ref_file = STRBUF_INIT ;
```

Código con los Errores

```
struct ref_lock * lock ;
int last_errno = 0 ;
int mustexist = ( old_sha1 && ! is_null_sha1 ( old_sha1 ) ) ;
int resolve_flags = RESOLVE_REF_NO_RECURSE ;
int resolved ;
files_assert_main_repository ( refs , "lock_ref_sha1_basic" ) ;
assert ( err ) ;
lock = xcalloc ( 1 , sizeof ( struct ref_lock ) ) ;
if ( mustexist )
resolve_flags |= RESOLVE_REF_READING ;
if ( flags & REF_DELETING )
resolve_flags |= RESOLVE_REF_ALLOW_BAD_NAME ;
files_ref_path ( refs , & ref_file , refname ) ;
```

Código con los Errores

```
resolved = ! ! refs_resolve_ref_unsafe ( & refs -> base ,
refname , resolve_flags ,
lock -> old_oid . hash , type ) ;
if ( ! resolved && errno == EISDIR ) {
if ( remove_empty_directories ( & ref_file ) ) {
last_errno = errno ;
if ( ! refs_verify_refname_available (
& refs -> base ,
refname , extras , skip , err ) )
strbuf_addf ( err , "there are still refs under '%s'" ,
refname ) ;
goto error_return ;
}
```

Código con los Errores

```
resolved = ! ! refs_resolve_ref_unsafe ( & refs -> base ,
refname , resolve_flags ,
lock -> old_oid . hash , type ) ;
}
if ( ! resolved ) {
last_errno = errno ;
if ( last_errno != ENOTDIR ||
! refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) )
strbuf_addf ( err , "unable to resolve reference '%s': %s" ,
refname , strerror ( last_errno ) ) ;
goto error_return ;
}
```

Código con los Errores

```
if ( is_null_oid ( & lock -> old_oid ) &&
refs_verify_refname_available ( & refs -> base , refname ,
extras , skip , err ) ) {
last_errno = ENOTDIR ;
goto error_return ;
}
lock -> lk = xcalloc ( 1 , sizeof ( struct lock_file ) ) ;
lock -> ref_name = xstrdup ( refname ) ;
if ( raceproof_create_file ( ref_file . buf , create_reflock , lock -> lk ) ) {
last_errno = errno ;
unable_to_lock_message ( ref_file . buf , errno , err ) ;
goto error_return ;
}
```

Código con los Errores

```
if ( verify_lock ( & refs -> base , lock , old_sha1 , mustexist , err ) ) {
last_errno = errno ;
goto error_return ;
}
goto out ;
error_return :
unlock_ref ( lock ) ;
lock = NULL ;
out :
strbuf_release ( & ref_file ) ;
errno = last_errno ;
return lock ;
}
```

Código con los Errores

```
static void write_packed_entry ( FILE * fh , const char * refname ,
const unsigned char * sha1 ,
const unsigned char * peeled )
{
fprintf_or_die ( fh , "%s %s\n" , sha1_to_hex ( sha1 ) , refname ) ;
if ( peeled )
fprintf_or_die ( fh , "^%s\n" , sha1_to_hex ( peeled ) ) ;
}
static int lock_packed_refs ( struct files_ref_store * refs , int flags )
{
static int timeout_configured = 0 ;
static int timeout_value = 1000 ;
struct packed_ref_cache * packed_ref_cache ;
```

Código con los Errores

```
files_assert_main_repository ( refs , "lock_packed_refs" ) ;
if ( ! timeout_configured ) {
git_config_get_int ( "core.packedrefstimeout" , & timeout_value ) ;
timeout_configured = 1 ;
}
if ( hold_lock_file_for_update_timeout (
& packlock , files_packed_refs_path ( refs ) ,
flags , timeout_value ) < 0 )
return - 1 ;
packed_ref_cache = get_packed_ref_cache ( refs ) ;
packed_ref_cache -> lock = & packlock ;
acquire_packed_ref_cache ( packed_ref_cache ) ;
return 0 ;
```

Código con los Errores

```
}
static int commit_packed_refs ( struct files_ref_store * refs )
{
struct packed_ref_cache * packed_ref_cache =
get_packed_ref_cache ( refs ) ;
int ok , error = 0 ;
int save_errno = 0 ;
FILE * out ;
struct ref_iterator * iter ;
files_assert_main_repository ( refs , "commit_packed_refs" ) ;
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed-refs not locked" ) ;
out = fdopen_lock_file ( packed_ref_cache -> lock , "w" ) ;
```

Código con los Errores

```
if ( ! out )
die_errno ( "unable to fdopen packed-refs descriptor" ) ;
fprintf_or_die ( out , "%s" , PACKED_REFS_HEADER ) ;
iter = cache_ref_iterator_begin ( packed_ref_cache -> cache , NULL , 0 ) ;
while ( ( ok = ref_iterator_advance ( iter ) ) == ITER_OK ) {
struct object_id peeled ;
int peel_error = ref_iterator_peel ( iter , & peeled ) ;
write_packed_entry ( out , iter -> refname , iter -> oid -> hash ,
peel_error ? NULL : peeled . hash ) ;
}
if ( ok != ITER_DONE )
die ( "error while iterating over references" ) ;
if ( commit_lock_file ( packed_ref_cache -> lock ) ) {
```

Código con los Errores

```
save_errno = errno ;
error = - 1 ;
}
packed_ref_cache -> lock = NULL ;
release_packed_ref_cache ( packed_ref_cache ) ;
errno = save_errno ;
return error ;
}
static void rollback_packed_refs ( struct files_ref_store * refs )
{
struct packed_ref_cache * packed_ref_cache =
get_packed_ref_cache ( refs ) ;
files_assert_main_repository ( refs , "rollback_packed_refs" ) ;
```

Código con los Errores

```
if ( ! packed_ref_cache -> lock )
die ( "internal error: packed-refs not locked" ) ;
rollback_lock_file ( packed_ref_cache -> lock ) ;
packed_ref_cache -> lock = NULL ;
release_packed_ref_cache ( packed_ref_cache ) ;
clear_packed_ref_cache ( refs ) ;
}
struct ref_to_prune {
struct ref_to_prune * next ;
unsigned char sha1 [ 20 ] ;
char name [ FLEX_ARRAY ] ;
} ;
enum {
```

Código con los Errores

```
REMOVE_EMPTY_PARENTS_REF = 0x01 ,
REMOVE_EMPTY_PARENTS_REFLOG = 0x02
} ;
static void try_remove_empty_parents ( struct files_ref_store * refs ,
const char * refname ,
unsigned int flags )
{
struct strbuf buf = STRBUF_INIT ;
struct strbuf sb = STRBUF_INIT ;
char * p , * q ;
int i ;
strbuf_addstr ( & buf , refname ) ;
p = buf . buf ;
```

Código con los Errores

```
for ( i = 0 ; i < 2 ; i ++ ) {
while ( * p && * p != '/' )
p ++ ;
while ( * p == '/' )
p ++ ;
}
q = buf . buf + buf . len ;
while ( flags & ( REMOVE_EMPTY_PARENTS_REF | REMOVE_EMPTY_PARENTS_REFLOG ) ) {
while ( q > p && * q != '/' )
q -- ;
while ( q > p && * ( q - 1 ) == '/' )
q -- ;
if ( q == p )
```

Código con los Errores

```
break ;
strbuf_setlen ( & buf , q - buf . buf ) ;
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , buf . buf ) ;
if ( ( flags & REMOVE_EMPTY_PARENTS_REF ) && rmdir ( sb . buf ) )
flags &= ~ REMOVE_EMPTY_PARENTS_REF ;
strbuf_reset ( & sb ) ;
files_reflog_path ( refs , & sb , buf . buf ) ;
if ( ( flags & REMOVE_EMPTY_PARENTS_REFLOG ) && rmdir ( sb . buf ) )
flags &= ~ REMOVE_EMPTY_PARENTS_REFLOG ;
}
strbuf_release ( & buf ) ;
strbuf_release ( & sb ) ;
```

Código con los Errores

```
}
static void prune_ref ( struct files_ref_store * refs , struct ref_to_prune * r )
{
struct ref_transaction * transaction ;
struct strbuf err = STRBUF_INIT ;
if ( check_refname_format ( r -> name , 0 ) )
return ;
transaction = ref_store_transaction_begin ( & refs -> base , & err ) ;
if ( ! transaction ||
ref_transaction_delete ( transaction , r -> name , r -> sha1 ,
REF_ISPRUNING | REF_NODEREF , NULL , & err ) ||
ref_transaction_commit ( transaction , & err ) ) {
ref_transaction_free ( transaction ) ;
```

Código con los Errores

```
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
return ;
}
ref_transaction_free ( transaction ) ;
strbuf_release ( & err ) ;
}
static void prune_refs ( struct files_ref_store * refs , struct ref_to_prune * r )
{
while ( r ) {
prune_ref ( refs , r ) ;
r = r -> next ;
}
```

Código con los Errores

```
}
static int files_pack_refs ( struct ref_store * ref_store , unsigned int flags )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE | REF_STORE_ODB ,
"pack_refs" ) ;
struct ref_iterator * iter ;
struct ref_dir * packed_refs ;
int ok ;
struct ref_to_prune * refs_to_prune = NULL ;
lock_packed_refs ( refs , LOCK_DIE_ON_ERROR ) ;
packed_refs = get_packed_refs ( refs ) ;
iter = cache_ref_iterator_begin ( get_loose_ref_cache ( refs ) , NULL , 0 ) ;
```

Código con los Errores

```
while ( ( ok = ref_iterator_advance ( iter ) ) == ITER_OK ) {
struct ref_entry * packed_entry ;
int is_tag_ref = starts_with ( iter -> refname , "refs/tags/" ) ;
if ( ref_type ( iter -> refname ) != REF_TYPE_NORMAL )
continue ;
if ( ! ( flags & PACK_REFS_ALL ) && ! is_tag_ref )
continue ;
if ( iter -> flags & REF_ISSYMREF )
continue ;
if ( ! ref_resolves_to_object ( iter -> refname , iter -> oid , iter -> flags ) )
continue ;
packed_entry = find_ref_entry ( packed_refs , iter -> refname ) ;
if ( packed_entry ) {
```

Código con los Errores

```
packed_entry -> flag = REF_ISPACKED ;
oidcpy ( & packed_entry -> u . value . oid , iter -> oid ) ;
} else {
packed_entry = create_ref_entry ( iter -> refname , iter -> oid ,
REF_ISPACKED , 0 ) ;
add_ref_entry ( packed_refs , packed_entry ) ;
}
oidclr ( & packed_entry -> u . value . peeled ) ;
if ( ( flags & PACK_REFS_PRUNE ) ) {
struct ref_to_prune * n ;
FLEX_ALLOC_STR ( n , name , iter -> refname ) ;
hashcpy ( n -> sha1 , iter -> oid -> hash ) ;
n -> next = refs_to_prune ;
```

Código con los Errores

```
refs_to_prune = n ;
}
}
if ( ok != ITER_DONE )
die ( "error while iterating over references" ) ;
if ( commit_packed_refs ( refs ) )
die_errno ( "unable to overwrite old ref-pack file" ) ;
prune_refs ( refs , refs_to_prune ) ;
return 0 ;
}
static int repack_without_refs ( struct files_ref_store * refs ,
struct string_list * refnames , struct strbuf * err )
{
```

Código con los Errores

```
struct ref_dir * packed ;
struct string_list_item * refname ;
int ret , needs_repacking = 0 , removed = 0 ;
files_assert_main_repository ( refs , "repack_without_refs" ) ;
assert ( err ) ;
/*Pruebas/error.c:970:50 syntax error, unexpected LEFT_BRACKET*/
for_each_string_list_item ( refname , refnames ) {
/*Pruebas/error.c:971:2 syntax error, unexpected IF*/
if ( get_packed_ref ( refs , refname -> string ) ) {
needs_repacking = 1 ;
break ;
}
/*Pruebas/error.c:975:1 syntax error, unexpected RIGHT_BRACKET*/
```

Código con los Errores

```
}
/*Pruebas/error.c:976:2 syntax error, unexpected IF*/
/*Pruebas/error.c:976:6 syntax error, unexpected EXCLAMATION*/
/*Pruebas/error.c:976:24 syntax error, unexpected RIGHT_PARENTHESIS*/
if ( ! needs_repacking )
/*Pruebas/error.c:977:6 syntax error, unexpected RETURN*/
/*Pruebas/error.c:977:8 syntax error, unexpected INTEGER*/
return 0 ;
/*Pruebas/error.c:978:2 syntax error, unexpected IF*/
if ( lock_packed_refs ( refs , 0 ) ) {
unable_to_lock_message ( files_packed_refs_path ( refs ) , errno , err ) ;
return - 1 ;
}
```

Código con los Errores

```
packed = get_packed_refs ( refs ) ;
for_each_string_list_item ( refname , refnames )
/*Pruebas/error.c:984:2 syntax error, unexpected IF*/
/*Pruebas/error.c:984:60 syntax error, unexpected NE_OP*/
/*Pruebas/error.c:984:62 syntax error, unexpected MINUS*/
/*Pruebas/error.c:984:64 syntax error, unexpected INTEGER*/
/*Pruebas/error.c:984:66 syntax error, unexpected RIGHT_PARENTHESIS*/
if ( remove_entry_from_dir ( packed , refname -> string ) != - 1 )
removed = 1 ;
/*Pruebas/error.c:986:2 syntax error, unexpected IF*/
/*Pruebas/error.c:986:6 syntax error, unexpected EXCLAMATION*/
/*Pruebas/error.c:986:16 syntax error, unexpected RIGHT_PARENTHESIS*/
if ( ! removed ) {
```

Código con los Errores

```
rollback_packed_refs ( refs ) ;
return 0 ;
}
ret = commit_packed_refs ( refs ) ;
/*Pruebas/error.c:991:2 syntax error, unexpected IF*/
if ( ret )
strbuf_addf ( err , "unable to overwrite old ref-pack file: %s" ,
strerror ( errno ) ) ;
/*Pruebas/error.c:994:6 syntax error, unexpected RETURN*/
return ret ;
/*Pruebas/error.c:995:1 syntax error, unexpected RIGHT_BRACKET*/
}
static int files_delete_refs ( struct ref_store * ref_store ,
```

Código con los Errores

```
struct string_list * refnames , unsigned int flags )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "delete_refs" ) ;
struct strbuf err = STRBUF_INIT ;
int i , result = 0 ;
if ( ! refnames -> nr )
return 0 ;
result = repack_without_refs ( refs , refnames , & err ) ;
if ( result ) {
if ( refnames -> nr == 1 )
error ( _ ( "could not delete reference %s: %s" ) ,
refnames -> items [ 0 ] . string , err . buf ) ;
```

Código con los Errores

```
else
error ( _ ( "could not delete references: %s" ) , err . buf ) ;
goto out ;
}
for ( i = 0 ; i < refnames -> nr ; i ++ ) {
const char * refname = refnames -> items [ i ] . string ;
if ( refs_delete_ref ( & refs -> base , NULL , refname , NULL , flags ) )
result |= error ( _ ( "could not remove reference %s" ) , refname ) ;
}
out :
strbuf_release ( & err ) ;
return result ;
}
```

Código con los Errores

```
struct rename_cb {
const char * tmp_renamed_log ;
int true_errno ;
} ;
static int rename_tmp_log_callback ( const char * path , void * cb_data )
{
struct rename_cb * cb = cb_data ;
if ( rename ( cb -> tmp_renamed_log , path ) ) {
cb -> true_errno = errno ;
if ( errno == ENOTDIR )
errno = EISDIR ;
return - 1 ;
} else {
```

Código con los Errores

```
return 0 ;
}
}
static int rename_tmp_log ( struct files_ref_store * refs , const char * newrefname )
{
struct strbuf path = STRBUF_INIT ;
struct strbuf tmp = STRBUF_INIT ;
struct rename_cb cb ;
int ret ;
files_reflog_path ( refs , & path , newrefname ) ;
files_reflog_path ( refs , & tmp , "refs/.tmp-renamed-log" ) ;
cb . tmp_renamed_log = tmp . buf ;
ret = raceproof_create_file ( path . buf , rename_tmp_log_callback , & cb ) ;
```

Código con los Errores

```
if ( ret ) {
if ( errno == EISDIR )
error ( "directory not empty: %s" , path . buf ) ;
else
error ( "unable to move logfile %s to %s: %s" ,
tmp . buf , path . buf ,
strerror ( cb . true_errno ) ) ;
}
strbuf_release ( & path ) ;
strbuf_release ( & tmp ) ;
return ret ;
}
static int write_ref_to_lockfile ( struct ref_lock * lock ,
```

Código con los Errores

```
const struct object_id * oid , struct strbuf * err ) ;
static int commit_ref_update ( struct files_ref_store * refs ,
struct ref_lock * lock ,
const struct object_id * oid , const char * logmsg ,
struct strbuf * err ) ;
static int files_rename_ref ( struct ref_store * ref_store ,
const char * oldrefname , const char * newrefname ,
const char * logmsg )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "rename_ref" ) ;
struct object_id oid , orig_oid ;
int flag = 0 , logmoved = 0 ;
```

Código con los Errores

```
struct ref_lock * lock ;
struct stat loginfo ;
struct strbuf sb_oldref = STRBUF_INIT ;
struct strbuf sb_newref = STRBUF_INIT ;
struct strbuf tmp_renamed_log = STRBUF_INIT ;
int log , ret ;
struct strbuf err = STRBUF_INIT ;
files_reflog_path ( refs , & sb_oldref , oldrefname ) ;
files_reflog_path ( refs , & sb_newref , newrefname ) ;
files_reflog_path ( refs , & tmp_renamed_log , "refs/.tmp-renamed-log" ) ;
log = ! lstat ( sb_oldref . buf , & loginfo ) ;
if ( log && S_ISLNK ( loginfo . st_mode ) ) {
ret = error ( "reflog for %s is a symlink" , oldrefname ) ;
```

Código con los Errores

```
goto out ;
}
if ( ! refs_resolve_ref_unsafe ( & refs -> base , oldrefname ,
RESOLVE_REF_READING | RESOLVE_REF_NO_RECURSE ,
orig_oid . hash , & flag ) ) {
ret = error ( "refname %s not found" , oldrefname ) ;
goto out ;
}
if ( flag & REF_ISSYMREF ) {
ret = error ( "refname %s is a symbolic ref, renaming it is not supported" ,
oldrefname ) ;
goto out ;
}
```

Código con los Errores

```
if ( ! refs_rename_ref_available ( & refs -> base , oldrefname , newrefname ) ) {
ret = 1 ;
goto out ;
}
if ( log && rename ( sb_oldref . buf , tmp_renamed_log . buf ) ) {
ret = error ( "unable to move logfile logs/%s to logs/" "refs/.tmp-renamed-log" ": %s" ,
oldrefname , strerror ( errno ) ) ;
goto out ;
}
if ( refs_delete_ref ( & refs -> base , logmsg , oldrefname ,
orig_oid . hash , REF_NODEREF ) ) {
error ( "unable to delete old %s" , oldrefname ) ;
goto rollback ;
```

Código con los Errores

```
}
if ( ! refs_read_ref_full ( & refs -> base , newrefname ,
RESOLVE_REF_READING | RESOLVE_REF_NO_RECURSE ,
oid . hash , NULL ) &&
refs_delete_ref ( & refs -> base , NULL , newrefname ,
NULL , REF_NODEREF ) ) {
if ( errno == EISDIR ) {
struct strbuf path = STRBUF_INIT ;
int result ;
files_ref_path ( refs , & path , newrefname ) ;
result = remove_empty_directories ( & path ) ;
strbuf_release ( & path ) ;
if ( result ) {
```

Código con los Errores

```
error ( "Directory not empty: %s" , newrefname ) ;
goto rollback ;
}
} else {
error ( "unable to delete existing %s" , newrefname ) ;
goto rollback ;
}
}
if ( log && rename_tmp_log ( refs , newrefname ) )
goto rollback ;
logmoved = log ;
lock = lock_ref_sha1_basic ( refs , newrefname , NULL , NULL , NULL ,
REF_NODEREF , NULL , & err ) ;
```

Código con los Errores

```
if ( ! lock ) {
error ( "unable to rename '%s' to '%s': %s" , oldrefname , newrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollback ;
}
oidcpy ( & lock -> old_oid , & orig_oid ) ;
if ( write_ref_to_lockfile ( lock , & orig_oid , & err ) ||
commit_ref_update ( refs , lock , & orig_oid , logmsg , & err ) ) {
error ( "unable to write current sha1 into %s: %s" , newrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollback ;
}
ret = 0 ;
```

Código con los Errores

```
goto out ;
rollback :
lock = lock_ref_sha1_basic ( refs , oldrefname , NULL , NULL , NULL ,
REF_NODEREF , NULL , & err ) ;
if ( ! lock ) {
error ( "unable to lock %s for rollback: %s" , oldrefname , err . buf ) ;
strbuf_release ( & err ) ;
goto rollbacklog ;
}
flag = log_all_ref_updates ;
log_all_ref_updates = LOG_REFS_NONE ;
if ( write_ref_to_lockfile ( lock , & orig_oid , & err ) ||
commit_ref_update ( refs , lock , & orig_oid , NULL , & err ) ) {
```

Código con los Errores

```
error ( "unable to write current sha1 into %s: %s" , oldrefname , err . buf ) ;
strbuf_release ( & err ) ;
}
log_all_ref_updates = flag ;
rollbacklog :
if ( logmoved && rename ( sb_newref . buf , sb_oldref . buf ) )
error ( "unable to restore logfile %s from %s: %s" ,
oldrefname , newrefname , strerror ( errno ) ) ;
if ( ! logmoved && log &&
rename ( tmp_renamed_log . buf , sb_oldref . buf ) )
error ( "unable to restore logfile %s from logs/" "refs/.tmp-renamed-log" ": %s" ,
oldrefname , strerror ( errno ) ) ;
ret = 1 ;
```

Código con los Errores

```
out :
strbuf_release ( & sb_newref ) ;
strbuf_release ( & sb_oldref ) ;
strbuf_release ( & tmp_renamed_log ) ;
return ret ;
}
static int close_ref ( struct ref_lock * lock )
{
if ( close_lock_file ( lock -> lk ) )
return - 1 ;
return 0 ;
}
static int commit_ref ( struct ref_lock * lock )
```

Código con los Errores

```
{
char * path = get_locked_file_path ( lock -> lk ) ;
struct stat st ;
if ( ! lstat ( path , & st ) && S_ISDIR ( st . st_mode ) ) {
size_t len = strlen ( path ) ;
struct strbuf sb_path = STRBUF_INIT ;
strbuf_attach ( & sb_path , path , len , len ) ;
remove_empty_directories ( & sb_path ) ;
strbuf_release ( & sb_path ) ;
} else {
free ( path ) ;
}
if ( commit_lock_file ( lock -> lk ) )
```

Código con los Errores

```
return - 1 ;
return 0 ;
}
static int open_or_create_logfile ( const char * path , void * cb )
{
int * fd = cb ;
* fd = open ( path , O_APPEND | O_WRONLY | O_CREAT , 0666 ) ;
return ( * fd < 0 ) ? - 1 : 0 ;
}
static int log_ref_setup ( struct files_ref_store * refs ,
const char * refname , int force_create ,
int * logfd , struct strbuf * err )
{
```

Código con los Errores

```
struct strbuf logfile_sb = STRBUF_INIT ;
char * logfile ;
files_reflog_path ( refs , & logfile_sb , refname ) ;
logfile = strbuf_detach ( & logfile_sb , NULL ) ;
if ( force_create || should_autocreate_reflog ( refname ) ) {
if ( raceproof_create_file ( logfile , open_or_create_logfile , logfd ) ) {
if ( errno == ENOENT )
strbuf_addf ( err , "unable to create directory for '%s': "
"%s" , logfile , strerror ( errno ) ) ;
else if ( errno == EISDIR )
strbuf_addf ( err , "there are still logs under '%s'" ,
logfile ) ;
else
```

Código con los Errores

```
strbuf_addf ( err , "unable to append to '%s': %s" ,
logfile , strerror ( errno ) ) ;
goto error ;
}
} else {
* logfd = open ( logfile , O_APPEND | O_WRONLY , 0666 ) ;
if ( * logfd < 0 ) {
if ( errno == ENOENT || errno == EISDIR ) {
;
} else {
strbuf_addf ( err , "unable to append to '%s': %s" ,
logfile , strerror ( errno ) ) ;
goto error ;
```

Código con los Errores

```
}
}
}
if ( * logfd >= 0 )
adjust_shared_perm ( logfile ) ;
free ( logfile ) ;
return 0 ;
error :
free ( logfile ) ;
return - 1 ;
}
static int files_create_reflog ( struct ref_store * ref_store ,
const char * refname , int force_create ,
```

Código con los Errores

```
struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "create_reflog" ) ;
int fd ;
if ( log_ref_setup ( refs , refname , force_create , & fd , err ) )
return - 1 ;
if ( fd >= 0 )
close ( fd ) ;
return 0 ;
}
static int log_ref_write_fd ( int fd , const struct object_id * old_oid ,
const struct object_id * new_oid ,
```

Código con los Errores

```
const char * committer , const char * msg )
{
int msglen , written ;
unsigned maxlen , len ;
char * logrec ;
msglen = msg ? strlen ( msg ) : 0 ;
maxlen = strlen ( committer ) + msglen + 100 ;
logrec = xmalloc ( maxlen ) ;
len = xsnprintf ( logrec , maxlen , "%s %s %s\n" ,
oid_to_hex ( old_oid ) ,
oid_to_hex ( new_oid ) ,
committer ) ;
if ( msglen )
```

Código con los Errores

```
len += copy_reflog_msg ( logrec + len - 1 , msg ) - 1 ;
written = len <= maxlen ? write_in_full ( fd , logrec , len ) : - 1 ;
free ( logrec ) ;
if ( written != len )
return - 1 ;
return 0 ;
}
static int files_log_ref_write ( struct files_ref_store * refs ,
const char * refname , const struct object_id * old_oid ,
const struct object_id * new_oid , const char * msg ,
int flags , struct strbuf * err )
{
int logfd , result ;
```

Código con los Errores

```
if ( log_all_ref_updates == LOG_REFS_UNSET )
log_all_ref_updates = is_bare_repository ( ) ? LOG_REFS_NONE : LOG_REFS_NORMAL ;
result = log_ref_setup ( refs , refname ,
flags & REF_FORCE_CREATE_REFLOG ,
& logfd , err ) ;
if ( result )
return result ;
if ( logfd < 0 )
return 0 ;
result = log_ref_write_fd ( logfd , old_oid , new_oid ,
git_committer_info ( 0 ) , msg ) ;
if ( result ) {
struct strbuf sb = STRBUF_INIT ;
```

Código con los Errores

```
int save_errno = errno ;
files_reflog_path ( refs , & sb , refname ) ;
strbuf_addf ( err , "unable to append to ’%s’: %s" ,
sb . buf , strerror ( save_errno ) ) ;
strbuf_release ( & sb ) ;
close ( logfd ) ;
return - 1 ;
}
if ( close ( logfd ) ) {
struct strbuf sb = STRBUF_INIT ;
int save_errno = errno ;
files_reflog_path ( refs , & sb , refname ) ;
strbuf_addf ( err , "unable to append to ’%s’: %s" ,
```

Código con los Errores

```
sb . buf , strerror ( save_errno ) ) ;
strbuf_release ( & sb ) ;
return - 1 ;
}
return 0 ;
}
static int write_ref_to_lockfile ( struct ref_lock * lock ,
const struct object_id * oid , struct strbuf * err )
{
static char term = '\n' ;
struct object * o ;
int fd ;
o = parse_object ( oid ) ;
```

Código con los Errores

```
if ( ! o ) {
strbuf_addf ( err ,
"trying to write ref '%s' with nonexistent object %s" ,
lock -> ref_name , oid_to_hex ( oid ) ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
if ( o -> type != OBJ_COMMIT && is_branch ( lock -> ref_name ) ) {
strbuf_addf ( err ,
"trying to write non-commit object %s to branch '%s'" ,
oid_to_hex ( oid ) , lock -> ref_name ) ;
unlock_ref ( lock ) ;
return - 1 ;
```

Código con los Errores

```
}
fd = get_lock_file_fd ( lock -> lk ) ;
if ( write_in_full ( fd , oid_to_hex ( oid ) , GIT_SHA1_HEXSZ ) != GIT_SHA1_HEXSZ ||
write_in_full ( fd , & term , 1 ) != 1 ||
close_ref ( lock ) < 0 ) {
strbuf_addf ( err ,
"couldn't write '%s'" , get_lock_file_path ( lock -> lk ) ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
return 0 ;
}
static int commit_ref_update ( struct files_ref_store * refs ,
```

Código con los Errores

```
struct ref_lock * lock ,
const struct object_id * oid , const char * logmsg ,
struct strbuf * err )
{
files_assert_main_repository ( refs , "commit_ref_update" ) ;
clear_loose_ref_cache ( refs ) ;
if ( files_log_ref_write ( refs , lock -> ref_name ,
& lock -> old_oid , oid ,
logmsg , 0 , err ) ) {
char * old_msg = strbuf_detach ( err , NULL ) ;
strbuf_addf ( err , "cannot update the ref '%s': %s" ,
lock -> ref_name , old_msg ) ;
free ( old_msg ) ;
```

Código con los Errores

```
unlock_ref ( lock ) ;
return - 1 ;
}
if ( strcmp ( lock -> ref_name , "HEAD" ) != 0 ) {
struct object_id head_oid ;
int head_flag ;
const char * head_ref ;
head_ref = refs_resolve_ref_unsafe ( & refs -> base , "HEAD" ,
RESOLVE_REF_READING ,
head_oid . hash , & head_flag ) ;
if ( head_ref && ( head_flag & REF_ISSYMREF ) &&
! strcmp ( head_ref , lock -> ref_name ) ) {
struct strbuf log_err = STRBUF_INIT ;
```

Código con los Errores

```
if ( files_log_ref_write ( refs , "HEAD" ,
& lock -> old_oid , oid ,
logmsg , 0 , & log_err ) ) {
error ( "%s" , log_err . buf ) ;
strbuf_release ( & log_err ) ;
}
}
}
if ( commit_ref ( lock ) ) {
strbuf_addf ( err , "couldn't set '%s'" , lock -> ref_name ) ;
unlock_ref ( lock ) ;
return - 1 ;
}
```

Código con los Errores

```
unlock_ref ( lock ) ;
return 0 ;
}
static int create_ref_symlink ( struct ref_lock * lock , const char * target )
{
int ret = - 1 ;
return ret ;
}
static void update_symref_reflog ( struct files_ref_store * refs ,
struct ref_lock * lock , const char * refname ,
const char * target , const char * logmsg )
{
struct strbuf err = STRBUF_INIT ;
```

Código con los Errores

```
struct object_id new_oid ;
if ( logmsg &&
! refs_read_ref_full ( & refs -> base , target ,
RESOLVE_REF_READING , new_oid . hash , NULL ) &&
files_log_ref_write ( refs , refname , & lock -> old_oid ,
& new_oid , logmsg , 0 , & err ) ) {
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
}
}
static int create_symref_locked ( struct files_ref_store * refs ,
struct ref_lock * lock , const char * refname ,
const char * target , const char * logmsg )
```

Código con los Errores

```
{
if ( prefer_symlink_refs && ! create_ref_symlink ( lock , target ) ) {
update_symref_reflog ( refs , lock , refname , target , logmsg ) ;
return 0 ;
}
if ( ! fdopen_lock_file ( lock -> lk , "w" ) )
return error ( "unable to fdopen %s: %s" ,
lock -> lk -> tempfile . filename . buf , strerror ( errno ) ) ;
update_symref_reflog ( refs , lock , refname , target , logmsg ) ;
fprintf ( lock -> lk -> tempfile . fp , "ref: %s\n" , target ) ;
if ( commit_ref ( lock ) < 0 )
return error ( "unable to write symref for %s: %s" , refname ,
strerror ( errno ) ) ;
```

Código con los Errores

```
return 0 ;
}
static int files_create_symref ( struct ref_store * ref_store ,
const char * refname , const char * target ,
const char * logmsg )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "create_symref" ) ;
struct strbuf err = STRBUF_INIT ;
struct ref_lock * lock ;
int ret ;
lock = lock_ref_sha1_basic ( refs , refname , NULL ,
NULL , NULL , REF_NODEREF , NULL ,
```

Código con los Errores

```
& err ) ;
if ( ! lock ) {
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
return - 1 ;
}
ret = create_symref_locked ( refs , lock , refname , target , logmsg ) ;
unlock_ref ( lock ) ;
return ret ;
}
static int files_reflog_exists ( struct ref_store * ref_store ,
const char * refname )
{
```

Código con los Errores

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ , "reflog_exists" ) ;
struct strbuf sb = STRBUF_INIT ;
struct stat st ;
int ret ;
files_reflog_path ( refs , & sb , refname ) ;
ret = ! lstat ( sb . buf , & st ) && S_ISREG ( st . st_mode ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int files_delete_reflog ( struct ref_store * ref_store ,
const char * refname )
{
```

Código con los Errores

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "delete_reflog" ) ;
struct strbuf sb = STRBUF_INIT ;
int ret ;
files_reflog_path ( refs , & sb , refname ) ;
ret = remove_path ( sb . buf ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int show_one_reflog_ent ( struct strbuf * sb , each_reflog_ent_fn fn , void * cb_data )
{
struct object_id ooid , noid ;
char * email_end , * message ;
```

Código con los Errores

```
timestamp_t timestamp ;
int tz ;
const char * p = sb -> buf ;
if ( ! sb -> len || sb -> buf [ sb -> len - 1 ] != '\n' ||
parse_oid_hex ( p , & ooid , & p ) || * p ++ != ' ' ||
parse_oid_hex ( p , & noid , & p ) || * p ++ != ' ' ||
! ( email_end = strchr ( p , '>' ) ) ||
email_end [ 1 ] != ' ' ||
! ( timestamp = parse_timestamp ( email_end + 2 , & message , 10 ) ) ||
! message || message [ 0 ] != ' ' ||
( message [ 1 ] != '+' && message [ 1 ] != '-' ) ||
! isdigit ( message [ 2 ] ) || ! isdigit ( message [ 3 ] ) ||
! isdigit ( message [ 4 ] ) || ! isdigit ( message [ 5 ] ) )
```

Código con los Errores

```
return 0 ;
email_end [ 1 ] = '\0' ;
tz = strtol ( message + 1 , NULL , 10 ) ;
if ( message [ 6 ] != '\t' )
message += 6 ;
else
message += 7 ;
return fn ( & ooid , & noid , p , timestamp , tz , message , cb_data ) ;
}
static char * find_beginning_of_line ( char * bob , char * scan )
{
while ( bob < scan && * ( -- scan ) != '\n' )
;
```

Código con los Errores

```
return scan ;
}
static int files_for_each_reflog_ent_reverse ( struct ref_store * ref_store ,
const char * refname ,
each_reflog_ent_fn fn ,
void * cb_data )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"for_each_reflog_ent_reverse" ) ;
struct strbuf sb = STRBUF_INIT ;
FILE * logfp ;
long pos ;
```

Código con los Errores

```
int ret = 0 , at_tail = 1 ;
files_reflog_path ( refs , & sb , refname ) ;
logfp = fopen ( sb . buf , "r" ) ;
strbuf_release ( & sb ) ;
if ( ! logfp )
return - 1 ;
if ( fseek ( logfp , 0 , SEEK_END ) < 0 )
ret = error ( "cannot seek back reflog for %s: %s" ,
refname , strerror ( errno ) ) ;
pos = ftell ( logfp ) ;
while ( ! ret && 0 < pos ) {
int cnt ;
size_t nread ;
```

Código con los Errores

```
char buf [ BUFSIZ ] ;
char * endp , * scanp ;
cnt = ( sizeof ( buf ) < pos ) ? sizeof ( buf ) : pos ;
if ( fseek ( logfp , pos - cnt , SEEK_SET ) ) {
ret = error ( "cannot seek back reflog for %s: %s" ,
refname , strerror ( errno ) ) ;
break ;
}
nread = fread ( buf , cnt , 1 , logfp ) ;
if ( nread != 1 ) {
ret = error ( "cannot read %d bytes from reflog for %s: %s" ,
cnt , refname , strerror ( errno ) ) ;
break ;
```

Código con los Errores

```
}
pos -= cnt ;
scanp = endp = buf + cnt ;
if ( at_tail && scanp [ - 1 ] == '\n' )
scanp -- ;
at_tail = 0 ;
while ( buf < scanp ) {
char * bp ;
bp = find_beginning_of_line ( buf , scanp ) ;
if ( * bp == '\n' ) {
strbuf_splice ( & sb , 0 , 0 , bp + 1 , endp - ( bp + 1 ) ) ;
scanp = bp ;
endp = bp + 1 ;
```

Código con los Errores

```
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
strbuf_reset ( & sb ) ;
if ( ret )
break ;
} else if ( ! pos ) {
strbuf_splice ( & sb , 0 , 0 , buf , endp - buf ) ;
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
strbuf_reset ( & sb ) ;
break ;
}
if ( bp == buf ) {
strbuf_splice ( & sb , 0 , 0 , buf , endp - buf ) ;
break ;
```

Código con los Errores

```
}
}
}
if ( ! ret && sb . len )
die ( "BUG: reverse reflog parser had leftover data" ) ;
fclose ( logfp ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int files_for_each_reflog_ent ( struct ref_store * ref_store ,
const char * refname ,
each_reflog_ent_fn fn , void * cb_data )
{
```

Código con los Errores

```
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"for_each_reflog_ent" ) ;
FILE * logfp ;
struct strbuf sb = STRBUF_INIT ;
int ret = 0 ;
files_reflog_path ( refs , & sb , refname ) ;
logfp = fopen ( sb . buf , "r" ) ;
strbuf_release ( & sb ) ;
if ( ! logfp )
return - 1 ;
while ( ! ret && ! strbuf_getwholeline ( & sb , logfp , '\n' ) )
ret = show_one_reflog_ent ( & sb , fn , cb_data ) ;
```

Código con los Errores

```
fclose ( logfp ) ;
strbuf_release ( & sb ) ;
return ret ;
}
struct files_reflog_iterator {
struct ref_iterator base ;
struct ref_store * ref_store ;
struct dir_iterator * dir_iterator ;
struct object_id oid ;
} ;
static int files_reflog_iterator_advance ( struct ref_iterator * ref_iterator )
{
struct files_reflog_iterator * iter =
```

Código con los Errores

```
( struct files_reflog_iterator * ) ref_iterator ;
struct dir_iterator * diter = iter -> dir_iterator ;
int ok ;
while ( ( ok = dir_iterator_advance ( diter ) ) == ITER_OK ) {
int flags ;
if ( ! S_ISREG ( diter -> st . st_mode ) )
continue ;
if ( diter -> basename [ 0 ] == '.' )
continue ;
if ( ends_with ( diter -> basename , ".lock" ) )
continue ;
if ( refs_read_ref_full ( iter -> ref_store ,
diter -> relative_path , 0 ,
```

Código con los Errores

```
iter -> oid . hash , & flags ) ) {
error ( "bad ref for %s" , diter -> path . buf ) ;
continue ;
}
iter -> base . refname = diter -> relative_path ;
iter -> base . oid = & iter -> oid ;
iter -> base . flags = flags ;
return ITER_OK ;
}
iter -> dir_iterator = NULL ;
if ( ref_iterator_abort ( ref_iterator ) == ITER_ERROR )
ok = ITER_ERROR ;
return ok ;
```

Código con los Errores

```
}
static int files_reflog_iterator_peel ( struct ref_iterator * ref_iterator ,
struct object_id * peeled )
{
die ( "BUG: ref_iterator_peel() called for reflog_iterator" ) ;
}
static int files_reflog_iterator_abort ( struct ref_iterator * ref_iterator )
{
struct files_reflog_iterator * iter =
( struct files_reflog_iterator * ) ref_iterator ;
int ok = ITER_DONE ;
if ( iter -> dir_iterator )
ok = dir_iterator_abort ( iter -> dir_iterator ) ;
```

Código con los Errores

```
base_ref_iterator_free ( ref_iterator ) ;
return ok ;
}
static struct ref_iterator_vtable files_reflog_iterator_vtable = {
files_reflog_iterator_advance ,
files_reflog_iterator_peel ,
files_reflog_iterator_abort
} ;
static struct ref_iterator * files_reflog_iterator_begin ( struct ref_store * ref_store )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_READ ,
"reflog_iterator_begin" ) ;
```

Código con los Errores

```
struct files_reflog_iterator * iter = xcalloc ( 1 , sizeof ( * iter ) ) ;
struct ref_iterator * ref_iterator = & iter -> base ;
struct strbuf sb = STRBUF_INIT ;
base_ref_iterator_init ( ref_iterator , & files_reflog_iterator_vtable ) ;
files_reflog_path ( refs , & sb , NULL ) ;
iter -> dir_iterator = dir_iterator_begin ( sb . buf ) ;
iter -> ref_store = ref_store ;
strbuf_release ( & sb ) ;
return ref_iterator ;
}
static int ref_update_reject_duplicates ( struct string_list * refnames ,
struct strbuf * err )
{
```

Código con los Errores

```
int i , n = refnames -> nr ;
assert ( err ) ;
for ( i = 1 ; i < n ; i ++ )
if ( ! strcmp ( refnames -> items [ i - 1 ] . string , refnames -> items [ i ] . string ) ) {
strbuf_addf ( err ,
"multiple updates for ref '%s' not allowed." ,
refnames -> items [ i ] . string ) ;
return 1 ;
}
return 0 ;
}
static int split_head_update ( struct ref_update * update ,
struct ref_transaction * transaction ,
```

Código con los Errores

```
const char * head_ref ,
struct string_list * affected_refnames ,
struct strbuf * err )
{
struct string_list_item * item ;
struct ref_update * new_update ;
if ( ( update -> flags & REF_LOG_ONLY ) ||
( update -> flags & REF_ISPRUNING ) ||
( update -> flags & REF_UPDATE_VIA_HEAD ) )
return 0 ;
if ( strcmp ( update -> refname , head_ref ) )
return 0 ;
item = string_list_insert ( affected_refnames , "HEAD" ) ;
```

Código con los Errores

```
if ( item -> util ) {
strbuf_addf ( err ,
"multiple updates for 'HEAD' (including one "
"via its referent '%s') are not allowed" ,
update -> refname ) ;
return TRANSACTION_NAME_CONFLICT ;
}
new_update = ref_transaction_add_update (
transaction , "HEAD" ,
update -> flags | REF_LOG_ONLY | REF_NODEREF ,
update -> new_oid . hash , update -> old_oid . hash ,
update -> msg ) ;
item -> util = new_update ;
```

Código con los Errores

```
return 0 ;
}
static int split_symref_update ( struct files_ref_store * refs ,
struct ref_update * update ,
const char * referent ,
struct ref_transaction * transaction ,
struct string_list * affected_refnames ,
struct strbuf * err )
{
struct string_list_item * item ;
struct ref_update * new_update ;
unsigned int new_flags ;
item = string_list_insert ( affected_refnames , referent ) ;
```

Código con los Errores

```
if ( item -> util ) {
strbuf_addf ( err ,
"multiple updates for '%s' (including one "
"via symref '%s') are not allowed" ,
referent , update -> refname ) ;
return TRANSACTION_NAME_CONFLICT ;
}
new_flags = update -> flags ;
if ( ! strcmp ( update -> refname , "HEAD" ) ) {
new_flags |= REF_UPDATE_VIA_HEAD ;
}
new_update = ref_transaction_add_update (
transaction , referent , new_flags ,
```

Código con los Errores

```
update -> new_oid . hash , update -> old_oid . hash ,
update -> msg ) ;
new_update -> parent_update = update ;
update -> flags |= REF_LOG_ONLY | REF_NODEREF ;
update -> flags &= ~ REF_HAVE_OLD ;
item -> util = new_update ;
return 0 ;
}
static const char * original_update_refname ( struct ref_update * update )
{
while ( update -> parent_update )
update = update -> parent_update ;
return update -> refname ;
```

Código con los Errores

```
}
static int check_old_oid ( struct ref_update * update , struct object_id * oid ,
struct strbuf * err )
{
if ( ! ( update -> flags & REF_HAVE_OLD ) ||
! oidcmp ( oid , & update -> old_oid ) )
return 0 ;
if ( is_null_oid ( & update -> old_oid ) )
strbuf_addf ( err , "cannot lock ref '%s': "
"reference already exists" ,
original_update_refname ( update ) ) ;
else if ( is_null_oid ( oid ) )
strbuf_addf ( err , "cannot lock ref '%s': "
```

Código con los Errores

```
"reference is missing but expected %s" ,
original_update_refname ( update ) ,
oid_to_hex ( & update -> old_oid ) ) ;
else
strbuf_addf ( err , "cannot lock ref '%s': "
"is at %s but expected %s" ,
original_update_refname ( update ) ,
oid_to_hex ( oid ) ,
oid_to_hex ( & update -> old_oid ) ) ;
return - 1 ;
}
static int lock_ref_for_update ( struct files_ref_store * refs ,
struct ref_update * update ,
```

Código con los Errores

```
struct ref_transaction * transaction ,
const char * head_ref ,
struct string_list * affected_refnames ,
struct strbuf * err )
{
struct strbuf referent = STRBUF_INIT ;
int mustexist = ( update -> flags & REF_HAVE_OLD ) &&
! is_null_oid ( & update -> old_oid ) ;
int ret ;
struct ref_lock * lock ;
files_assert_main_repository ( refs , "lock_ref_for_update" ) ;
if ( ( update -> flags & REF_HAVE_NEW ) && is_null_oid ( & update -> new_oid ) )
update -> flags |= REF_DELETING ;
```

Código con los Errores

```
if ( head_ref ) {
ret = split_head_update ( update , transaction , head_ref ,
affected_refnames , err ) ;
if ( ret )
return ret ;
}
ret = lock_raw_ref ( refs , update -> refname , mustexist ,
affected_refnames , NULL ,
& lock , & referent ,
& update -> type , err ) ;
if ( ret ) {
char * reason ;
reason = strbuf_detach ( err , NULL ) ;
```

Código con los Errores

```
strbuf_addf ( err , "cannot lock ref '%s': %s" ,
original_update_refname ( update ) , reason ) ;
free ( reason ) ;
return ret ;
}
update -> backend_data = lock ;
if ( update -> type & REF_ISSYMREF ) {
if ( update -> flags & REF_NODEREF ) {
if ( refs_read_ref_full ( & refs -> base ,
referent . buf , 0 ,
lock -> old_oid . hash , NULL ) ) {
if ( update -> flags & REF_HAVE_OLD ) {
strbuf_addf ( err , "cannot lock ref '%s': "
```

Código con los Errores

```
"error reading reference" ,
original_update_refname ( update ) ) ;
return - 1 ;
}
} else if ( check_old_oid ( update , & lock -> old_oid , err ) ) {
return TRANSACTION_GENERIC_ERROR ;
}
} else {
ret = split_symref_update ( refs , update ,
referent . buf , transaction ,
affected_refnames , err ) ;
if ( ret )
return ret ;
```

Código con los Errores

```
}
} else {
struct ref_update * parent_update ;
if ( check_old_oid ( update , & lock -> old_oid , err ) )
return TRANSACTION_GENERIC_ERROR ;
for ( parent_update = update -> parent_update ;
parent_update ;
parent_update = parent_update -> parent_update ) {
struct ref_lock * parent_lock = parent_update -> backend_data ;
oidcpy ( & parent_lock -> old_oid , & lock -> old_oid ) ;
}
}
if ( ( update -> flags & REF_HAVE_NEW ) &&
```

Código con los Errores

```
! ( update -> flags & REF_DELETING ) &&
! ( update -> flags & REF_LOG_ONLY ) ) {
if ( ! ( update -> type & REF_ISSYMREF ) &&
! oidcmp ( & lock -> old_oid , & update -> new_oid ) ) {
} else if ( write_ref_to_lockfile ( lock , & update -> new_oid ,
err ) ) {
char * write_err = strbuf_detach ( err , NULL ) ;
update -> backend_data = NULL ;
strbuf_addf ( err ,
"cannot update ref '%s': %s" ,
update -> refname , write_err ) ;
free ( write_err ) ;
return TRANSACTION_GENERIC_ERROR ;
```

Código con los Errores

```
} else {
update -> flags |= REF_NEEDS_COMMIT ;
}
}
if ( ! ( update -> flags & REF_NEEDS_COMMIT ) ) {
if ( close_ref ( lock ) ) {
strbuf_addf ( err , "couldn't close '%s.lock'" ,
update -> refname ) ;
return TRANSACTION_GENERIC_ERROR ;
}
}
return 0 ;
}
```

Código con los Errores

```
static int files_transaction_commit ( struct ref_store * ref_store ,
struct ref_transaction * transaction ,
struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE ,
"ref_transaction_commit" ) ;
int ret = 0 , i ;
struct string_list refs_to_delete = STRING_LIST_INIT_NODUP ;
struct string_list_item * ref_to_delete ;
struct string_list affected_refnames = STRING_LIST_INIT_NODUP ;
char * head_ref = NULL ;
int head_type ;
```

Código con los Errores

```
struct object_id head_oid ;
struct strbuf sb = STRBUF_INIT ;
assert ( err ) ;
if ( transaction -> state != REF_TRANSACTION_OPEN )
die ( "BUG: commit called for transaction that is not open" ) ;
if ( ! transaction -> nr ) {
transaction -> state = REF_TRANSACTION_CLOSED ;
return 0 ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct string_list_item * item =
string_list_append ( & affected_refnames , update -> refname ) ;
```

Código con los Errores

```
item -> util = update ;
}
string_list_sort ( & affected_refnames ) ;
if ( ref_update_reject_duplicates ( & affected_refnames , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
head_ref = refs_resolve_refdup ( ref_store , "HEAD" ,
RESOLVE_REF_NO_RECURSE ,
head_oid . hash , & head_type ) ;
if ( head_ref && ! ( head_type & REF_ISSYMREF ) ) {
free ( head_ref ) ;
head_ref = NULL ;
```

Código con los Errores

```
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
ret = lock_ref_for_update ( refs , update , transaction ,
head_ref , & affected_refnames , err ) ;
if ( ret )
goto cleanup ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
if ( update -> flags & REF_NEEDS_COMMIT ||
update -> flags & REF_LOG_ONLY ) {
```

Código con los Errores

```
if ( files_log_ref_write ( refs ,
lock -> ref_name ,
& lock -> old_oid ,
& update -> new_oid ,
update -> msg , update -> flags ,
err ) ) {
char * old_msg = strbuf_detach ( err , NULL ) ;
strbuf_addf ( err , "cannot update the ref '%s': %s" ,
lock -> ref_name , old_msg ) ;
free ( old_msg ) ;
unlock_ref ( lock ) ;
update -> backend_data = NULL ;
ret = TRANSACTION_GENERIC_ERROR ;
```

Código con los Errores

```
goto cleanup ;
}
}
if ( update -> flags & REF_NEEDS_COMMIT ) {
clear_loose_ref_cache ( refs ) ;
if ( commit_ref ( lock ) ) {
strbuf_addf ( err , "couldn't set '%s'" , lock -> ref_name ) ;
unlock_ref ( lock ) ;
update -> backend_data = NULL ;
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
}
```

Código con los Errores

```
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
if ( update -> flags & REF_DELETING &&
! ( update -> flags & REF_LOG_ONLY ) ) {
if ( ! ( update -> type & REF_ISPACKED ) ||
update -> type & REF_ISSYMREF ) {
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , lock -> ref_name ) ;
if ( unlink_or_msg ( sb . buf , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
```

Código con los Errores

```
}
update -> flags |= REF_DELETED_LOOSE ;
}
if ( ! ( update -> flags & REF_ISPRUNING ) )
string_list_append ( & refs_to_delete ,
lock -> ref_name ) ;
}
}
if ( repack_without_refs ( refs , & refs_to_delete , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
/*Pruebas/error.c:1984:64 syntax error, unexpected LEFT_BRACKET*/
```

Código con los Errores

```
for_each_string_list_item ( ref_to_delete , & refs_to_delete ) {
strbuf_reset ( & sb ) ;
/*Pruebas/error.c:1986:50 syntax error, unexpected PTR_OP*/
/*Pruebas/error.c:1986:59 syntax error, unexpected RIGHT_PARENTHESIS*/
files_reflog_path ( refs , & sb , ref_to_delete -> string ) ;
/*Pruebas/error.c:1987:2 syntax error, unexpected IF*/
/*Pruebas/error.c:1987:6 syntax error, unexpected EXCLAMATION*/
/*Pruebas/error.c:1987:36 syntax error, unexpected RIGHT_PARENTHESIS*/
if ( ! unlink_or_warn ( sb . buf ) )
try_remove_empty_parents ( refs , ref_to_delete -> string ,
REMOVE_EMPTY_PARENTS_REFLOG ) ;
/*Pruebas/error.c:1990:1 syntax error, unexpected RIGHT_BRACKET*/
}
```

Código con los Errores

```
clear_loose_ref_cache ( refs ) ;
cleanup :
strbuf_release ( & sb ) ;
transaction -> state = REF_TRANSACTION_CLOSED ;
/*Pruebas/error.c:1995:3 syntax error, unexpected FOR*/
/*Pruebas/error.c:1995:13 syntax error, unexpected SEMICOLON*/
/*Pruebas/error.c:1995:17 syntax error, unexpected LESS*/
/*Pruebas/error.c:1995:44 syntax error, unexpected RIGHT_PARENTHESIS*/
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
struct ref_lock * lock = update -> backend_data ;
if ( lock )
unlock_ref ( lock ) ;
```

Código con los Errores

```
if ( update -> flags & REF_DELETED_LOOSE ) {
try_remove_empty_parents ( refs , update -> refname ,
REMOVE_EMPTY_PARENTS_REF ) ;
}
}
string_list_clear ( & refs_to_delete , 0 ) ;
free ( head_ref ) ;
string_list_clear ( & affected_refnames , 0 ) ;
/*Pruebas/error.c:2008:6 syntax error, unexpected RETURN*/
return ret ;
/*Pruebas/error.c:2009:1 syntax error, unexpected RIGHT_BRACKET*/
}
static int ref_present ( const char * refname ,
```

Código con los Errores

```
const struct object_id * oid , int flags , void * cb_data )
{
struct string_list * affected_refnames = cb_data ;
return string_list_has_string ( affected_refnames , refname ) ;
}
static int files_initial_transaction_commit ( struct ref_store * ref_store ,
struct ref_transaction * transaction ,
struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE ,
"initial_ref_transaction_commit" ) ;
int ret = 0 , i ;
```

Código con los Errores

```
struct string_list affected_refnames = STRING_LIST_INIT_NODUP ;
assert ( err ) ;
if ( transaction -> state != REF_TRANSACTION_OPEN )
die ( "BUG: commit called for transaction that is not open" ) ;
for ( i = 0 ; i < transaction -> nr ; i ++ )
string_list_append ( & affected_refnames ,
transaction -> updates [ i ] -> refname ) ;
string_list_sort ( & affected_refnames ) ;
if ( ref_update_reject_duplicates ( & affected_refnames , err ) ) {
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
if ( refs_for_each_rawref ( & refs -> base , ref_present ,
```

Código con los Errores

```
& affected_refnames ) )
die ( "BUG: initial ref transaction called with existing refs" ) ;
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
if ( ( update -> flags & REF_HAVE_OLD ) &&
! is_null_oid ( & update -> old_oid ) )
die ( "BUG: initial ref transaction with old_sha1 set" ) ;
if ( refs_verify_refname_available ( & refs -> base , update -> refname ,
& affected_refnames , NULL ,
err ) ) {
ret = TRANSACTION_NAME_CONFLICT ;
goto cleanup ;
}
```

Código con los Errores

```
}
if ( lock_packed_refs ( refs , 0 ) ) {
strbuf_addf ( err , "unable to lock packed-refs file: %s" ,
strerror ( errno ) ) ;
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
for ( i = 0 ; i < transaction -> nr ; i ++ ) {
struct ref_update * update = transaction -> updates [ i ] ;
if ( ( update -> flags & REF_HAVE_NEW ) &&
! is_null_oid ( & update -> new_oid ) )
add_packed_ref ( refs , update -> refname ,
& update -> new_oid ) ;
```

Código con los Errores

```
}
if ( commit_packed_refs ( refs ) ) {
strbuf_addf ( err , "unable to commit packed-refs file: %s" ,
strerror ( errno ) ) ;
ret = TRANSACTION_GENERIC_ERROR ;
goto cleanup ;
}
cleanup :
transaction -> state = REF_TRANSACTION_CLOSED ;
string_list_clear ( & affected_refnames , 0 ) ;
return ret ;
}
struct expire_reflog_cb {
```

Código con los Errores

```
unsigned int flags ;
reflog_expiry_should_prune_fn * should_prune_fn ;
void * policy_cb ;
FILE * newlog ;
struct object_id last_kept_oid ;
} ;
static int expire_reflog_ent ( struct object_id * ooid , struct object_id * noid ,
const char * email , timestamp_t timestamp , int tz ,
const char * message , void * cb_data )
{
struct expire_reflog_cb * cb = cb_data ;
struct expire_reflog_policy_cb * policy_cb = cb -> policy_cb ;
if ( cb -> flags & EXPIRE_REFLOGS_REWRITE )
```

Código con los Errores

```
ooid = & cb -> last_kept_oid ;
if ( ( * cb -> should_prune_fn ) ( ooid , noid , email , timestamp , tz ,
message , policy_cb ) ) {
if ( ! cb -> newlog )
printf ( "would prune %s" , message ) ;
else if ( cb -> flags & EXPIRE_REFLOGS_VERBOSE )
printf ( "prune %s" , message ) ;
} else {
if ( cb -> newlog ) {
/*Pruebas/error.c:2098:45 syntax error, unexpected IDENTIFIER*/
/*Pruebas/error.c:2098:58 syntax error, unexpected LITERAL*/
/*Pruebas/error.c:2098:60 syntax error, unexpected COMMA*/
fprintf ( cb -> newlog , "%s %s %s %" PRItime " %+05d\t%s" ,
```

Código con los Errores

```
oid_to_hex ( ooid ) , oid_to_hex ( noid ) ,
/*Pruebas/error.c:2100:34 syntax error, unexpected RIGHT_PARENTHESIS*/
email , timestamp , tz , message ) ;
oidcpy ( & cb -> last_kept_oid , noid ) ;
/*Pruebas/error.c:2102:1 syntax error, unexpected RIGHT_BRACKET*/
}
/*Pruebas/error.c:2103:2 syntax error, unexpected IF*/
if ( cb -> flags & EXPIRE_REFLOGS_VERBOSE )
printf ( "keep %s" , message ) ;
/*Pruebas/error.c:2105:1 syntax error, unexpected RIGHT_BRACKET*/
}
/*Pruebas/error.c:2106:6 syntax error, unexpected RETURN*/
/*Pruebas/error.c:2106:8 syntax error, unexpected INTEGER*/
```

Código con los Errores

```
return 0 ;
/*Pruebas/error.c:2107:1 syntax error, unexpected RIGHT_BRACKET*/
}
static int files_reflog_expire ( struct ref_store * ref_store ,
const char * refname , const unsigned char * sha1 ,
unsigned int flags ,
reflog_expiry_prepare_fn prepare_fn ,
reflog_expiry_should_prune_fn should_prune_fn ,
reflog_expiry_cleanup_fn cleanup_fn ,
void * policy_cb_data )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "reflog_expire" ) ;
```

Código con los Errores

```
static struct lock_file reflog_lock ;
struct expire_reflog_cb cb ;
struct ref_lock * lock ;
struct strbuf log_file_sb = STRBUF_INIT ;
char * log_file ;
int status = 0 ;
int type ;
struct strbuf err = STRBUF_INIT ;
struct object_id oid ;
memset ( & cb , 0 , sizeof ( cb ) ) ;
cb . flags = flags ;
cb . policy_cb = policy_cb_data ;
cb . should_prune_fn = should_prune_fn ;
```

Código con los Errores

```
lock = lock_ref_sha1_basic ( refs , refname , sha1 ,
NULL , NULL , REF_NODEREF ,
& type , & err ) ;
if ( ! lock ) {
error ( "cannot lock ref '%s': %s" , refname , err . buf ) ;
strbuf_release ( & err ) ;
return - 1 ;
}
if ( ! refs_reflog_exists ( ref_store , refname ) ) {
unlock_ref ( lock ) ;
return 0 ;
}
files_reflog_path ( refs , & log_file_sb , refname ) ;
```

Código con los Errores

```
log_file = strbuf_detach ( & log_file_sb , NULL ) ;
if ( ! ( flags & EXPIRE_REFLOGS_DRY_RUN ) ) {
if ( hold_lock_file_for_update ( & reflog_lock , log_file , 0 ) < 0 ) {
struct strbuf err = STRBUF_INIT ;
unable_to_lock_message ( log_file , errno , & err ) ;
error ( "%s" , err . buf ) ;
strbuf_release ( & err ) ;
goto failure ;
}
cb . newlog = fdopen_lock_file ( & reflog_lock , "w" ) ;
if ( ! cb . newlog ) {
error ( "cannot fdopen %s (%s)" ,
get_lock_file_path ( & reflog_lock ) , strerror ( errno ) ) ;
```

Código con los Errores

```
goto failure ;
}
}
hashcpy ( oid . hash , sha1 ) ;
( * prepare_fn ) ( refname , & oid , cb . policy_cb ) ;
refs_for_each_reflog_ent ( ref_store , refname , expire_reflog_ent , & cb ) ;
( * cleanup_fn ) ( cb . policy_cb ) ;
if ( ! ( flags & EXPIRE_REFLOGS_DRY_RUN ) ) {
int update = ( flags & EXPIRE_REFLOGS_UPDATE_REF ) &&
! ( type & REF_ISSYMREF ) &&
! is_null_oid ( & cb . last_kept_oid ) ;
if ( close_lock_file ( & reflog_lock ) ) {
status |= error ( "couldn't write %s: %s" , log_file ,
```

Código con los Errores

```
strerror ( errno ) ) ;
} else if ( update &&
( write_in_full ( get_lock_file_fd ( lock -> lk ) ,
oid_to_hex ( & cb . last_kept_oid ) , GIT_SHA1_HEXSZ ) != GIT_SHA1_HEXSZ ||
write_str_in_full ( get_lock_file_fd ( lock -> lk ) , "\n" ) != 1 ||
close_ref ( lock ) < 0 ) ) {
status |= error ( "couldn't write %s" ,
get_lock_file_path ( lock -> lk ) ) ;
rollback_lock_file ( & reflog_lock ) ;
} else if ( commit_lock_file ( & reflog_lock ) ) {
status |= error ( "unable to write reflog '%s' (%s)" ,
log_file , strerror ( errno ) ) ;
} else if ( update && commit_ref ( lock ) ) {
```

Código con los Errores

```
status |= error ( "couldn't set %s" , lock -> ref_name ) ;
}
}
free ( log_file ) ;
unlock_ref ( lock )
/*Pruebas/error.c:2188:6 syntax error, unexpected RETURN*/
return status ;
failure :
rollback_lock_file ( & reflog_lock ) ;
free ( log_file ) ;
unlock_ref ( lock ) ;
/*Pruebas/error.c:2193:6 syntax error, unexpected RETURN*/
/*Pruebas/error.c:2193:8 syntax error, unexpected MINUS*/
```

Código con los Errores

```
/*Pruebas/error.c:2193:10 syntax error, unexpected INTEGER*/
return - 1 ;
/*Pruebas/error.c:2194:1 syntax error, unexpected RIGHT_BRACKET*/
}
static int files_init_db ( struct ref_store * ref_store , struct strbuf * err )
{
struct files_ref_store * refs =
files_downcast ( ref_store , REF_STORE_WRITE , "init_db" ) ;
struct strbuf sb = STRBUF_INIT ;
files_ref_path ( refs , & sb , "refs/heads" )
/*Pruebas/error.c:2201:15 syntax error, unexpected IDENTIFIER*/
/*Pruebas/error.c:2201:28 syntax error, unexpected COMMA*/
/*Pruebas/error.c:2201:30 syntax error, unexpected INTEGER*/
```

Código con los Errores

```
/*Pruebas/error.c:2201:32 syntax error, unexpected RIGHT_PARENTHESIS*/
safe_create_dir ( sb . buf , 1 ) ;
strbuf_reset ( & sb ) ;
files_ref_path ( refs , & sb , "refs/tags" ) ;
safe_create_dir ( sb . buf , 1 ) ;
strbuf_release ( & sb ) ;
/*Pruebas/error.c:2206:6 syntax error, unexpected RETURN*/
/*Pruebas/error.c:2206:8 syntax error, unexpected INTEGER*/
return 0
/*Pruebas/error.c:2207:1 syntax error, unexpected RIGHT_BRACKET*/
}
struct ref_storage_be refs_be_files = {
NULL ,
```

Código con los Errores

```
"files" ,
files_ref_store_create ,
files_init_db ,
files_transaction_commit ,
files_initial_transaction_commit ,
files_pack_refs ,
files_peel_ref ,
files_create_symref ,
files_delete_refs ,
files_rename_ref ,
files_ref_iterator_begin ,
files_read_raw_ref ,
files_reflog_iterator_begin ,
```

Código con los Errores

```
files_for_each_reflog_ent ,
files_for_each_reflog_ent_reverse ,
files_reflog_exists ,
files_create_reflog ,
files_delete_reflog ,
/*Pruebas/error.c:2228:19 syntax error, unexpected £end*/
```