### 张金源/76066001

我这报告基于google开发的软件LevelDB**,** is an [open source](https://en.wikipedia.org/wiki/Open_source) [on-disk key-value](https://en.wikipedia.org/wiki/NoSQL" \l "Classification_based_on_data_model) store written by [Google](https://en.wikipedia.org/wiki/Google) fellows [Jeffrey Dean](https://en.wikipedia.org/wiki/Jeff_Dean_(computer_scientist)) and Sanjay Ghemawat. Inspired by [Bigtable](https://en.wikipedia.org/wiki/Bigtable), LevelDB is hosted on [GitHub](https://en.wikipedia.org/wiki/GitHub) under the [New BSD License](https://en.wikipedia.org/wiki/Modified_BSD_License) and has been ported to a variety of [Unix](https://en.wikipedia.org/wiki/Unix)-based systems, [Mac OS X](https://en.wikipedia.org/wiki/Mac_OS_X), [Windows](https://en.wikipedia.org/wiki/Microsoft_Windows), and [Android](https://en.wikipedia.org/wiki/Android_(operating_system)). 它的文件夹里包括cmake, db(源代码), doc, helpers, include, issues, port, table, util. 我报告要分析他的db文件夹的一些代码。

# autocompact\_test.cc

// Copyright (c) 2013 The LevelDB Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file. See the AUTHORS file for names of contributors.

#include "leveldb/db.h"

#include "db/db\_impl.h"

#include "leveldb/cache.h"

#include "util/testharness.h"

#include "util/testutil.h"

namespace leveldb {

class AutoCompactTest {

public:

std::string dbname\_;

Cache\* tiny\_cache\_;

Options options\_;

DB\* db\_;

AutoCompactTest() {

dbname\_ = test::TmpDir() + "/autocompact\_test";

tiny\_cache\_ = NewLRUCache(100);

options\_.block\_cache = tiny\_cache\_;

DestroyDB(dbname\_, options\_);

options\_.create\_if\_missing = true;

options\_.compression = kNoCompression;

ASSERT\_OK(DB::Open(options\_, dbname\_, &db\_));

}

~AutoCompactTest() {

delete db\_;

DestroyDB(dbname\_, Options());

delete tiny\_cache\_;

}

std::string Key(int i) {

char buf[100];

snprintf(buf, sizeof(buf), "key%06d", i);

return std::string(buf);

}

uint64\_t Size(const Slice& start, const Slice& limit) {

Range r(start, limit);

uint64\_t size;

db\_->GetApproximateSizes(&r, 1, &size);

return size;

}

void DoReads(int n);

};

static const int kValueSize = 200 \* 1024;

static const int kTotalSize = 100 \* 1024 \* 1024;

static const int kCount = kTotalSize / kValueSize;

// Read through the first n keys repeatedly and check that they get

// compacted (verified by checking the size of the key space).

void AutoCompactTest::DoReads(int n) {

std::string value(kValueSize, 'x');

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

// Fill database

for (int i = 0; i < kCount; i++) {

ASSERT\_OK(db\_->Put(WriteOptions(), Key(i), value));

}

ASSERT\_OK(dbi->TEST\_CompactMemTable());

// Delete everything

for (int i = 0; i < kCount; i++) {

ASSERT\_OK(db\_->Delete(WriteOptions(), Key(i)));

}

ASSERT\_OK(dbi->TEST\_CompactMemTable());

// Get initial measurement of the space we will be reading.

const int64\_t initial\_size = Size(Key(0), Key(n));

const int64\_t initial\_other\_size = Size(Key(n), Key(kCount));

// Read until size drops significantly.

std::string limit\_key = Key(n);

for (int read = 0; true; read++) {

ASSERT\_LT(read, 100) << "Taking too long to compact";

Iterator\* iter = db\_->NewIterator(ReadOptions());

for (iter->SeekToFirst();

iter->Valid() && iter->key().ToString() < limit\_key;

iter->Next()) {

// Drop data

}

delete iter;

// Wait a little bit to allow any triggered compactions to complete.

Env::Default()->SleepForMicroseconds(1000000);

uint64\_t size = Size(Key(0), Key(n));

fprintf(stderr, "iter %3d => %7.3f MB [other %7.3f MB]\n",

read+1, size/1048576.0, Size(Key(n), Key(kCount))/1048576.0);

if (size <= initial\_size/10) {

break;

}

}

// Verify that the size of the key space not touched by the reads

// is pretty much unchanged.

const int64\_t final\_other\_size = Size(Key(n), Key(kCount));

ASSERT\_LE(final\_other\_size, initial\_other\_size + 1048576);

ASSERT\_GE(final\_other\_size, initial\_other\_size/5 - 1048576);

}

TEST(AutoCompactTest, ReadAll) {

DoReads(kCount);

}

TEST(AutoCompactTest, ReadHalf) {

DoReads(kCount/2);

}

} // namespace leveldb

int main(int argc, char\*\* argv) {

return leveldb::test::RunAllTests();

}

上面它提供了注释已经很明显解释它的功能。。

# builder.cc

// Copyright (c) 2011 The LevelDB Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file. See the AUTHORS file for names of contributors.

#include "db/builder.h"

#include "db/filename.h"

#include "db/dbformat.h"

#include "db/table\_cache.h"

#include "db/version\_edit.h"

#include "leveldb/db.h"

#include "leveldb/env.h"

#include "leveldb/iterator.h"

namespace leveldb {

Status BuildTable(const std::string& dbname,

Env\* env,

const Options& options,

TableCache\* table\_cache,

Iterator\* iter,

FileMetaData\* meta) {

Status s;

meta->file\_size = 0;

iter->SeekToFirst();

std::string fname = TableFileName(dbname, meta->number);

if (iter->Valid()) {

WritableFile\* file;

s = env->NewWritableFile(fname, &file);

if (!s.ok()) {

return s;

}

TableBuilder\* builder = new TableBuilder(options, file);

meta->smallest.DecodeFrom(iter->key());

for (; iter->Valid(); iter->Next()) {

Slice key = iter->key();

meta->largest.DecodeFrom(key);

builder->Add(key, iter->value());

}

// Finish and check for builder errors

s = builder->Finish();

if (s.ok()) {

meta->file\_size = builder->FileSize();

assert(meta->file\_size > 0);

}

delete builder;

// Finish and check for file errors

if (s.ok()) {

s = file->Sync();

}

if (s.ok()) {

s = file->Close();

}

delete file;

file = nullptr;

if (s.ok()) {

// Verify that the table is usable

Iterator\* it = table\_cache->NewIterator(ReadOptions(),

meta->number,

meta->file\_size);

s = it->status();

delete it;

}

}

// Check for input iterator errors

if (!iter->status().ok()) {

s = iter->status();

}

if (s.ok() && meta->file\_size > 0) {

// Keep it

} else {

env->DeleteFile(fname);

}

return s;

}

} // namespace leveldb

这builder的代码他功能但是创建。

# builder.h

// Copyright (c) 2011 The LevelDB Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file. See the AUTHORS file for names of contributors.

#ifndef STORAGE\_LEVELDB\_DB\_BUILDER\_H\_

#define STORAGE\_LEVELDB\_DB\_BUILDER\_H\_

#include "leveldb/status.h"

namespace leveldb {

struct Options;

struct FileMetaData;

class Env;

class Iterator;

class TableCache;

class VersionEdit;

// Build a Table file from the contents of \*iter. The generated file

// will be named according to meta->number. On success, the rest of

// \*meta will be filled with metadata about the generated table.

// If no data is present in \*iter, meta->file\_size will be set to

// zero, and no Table file will be produced.

Status BuildTable(const std::string& dbname,

Env\* env,

const Options& options,

TableCache\* table\_cache,

Iterator\* iter,

FileMetaData\* meta);

} // namespace leveldb

#endif // STORAGE\_LEVELDB\_DB\_BUILDER\_H\_

这是builder的头文件，在注释已经分析了它的功能。

# c\_test.c

/\* Copyright (c) 2011 The LevelDB Authors. All rights reserved.

Use of this source code is governed by a BSD-style license that can be

found in the LICENSE file. See the AUTHORS file for names of contributors. \*/

#include "leveldb/c.h"

#include <stddef.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

const char\* phase = "";

static void StartPhase(const char\* name) {

fprintf(stderr, "=== Test %s\n", name);

phase = name;

}

#define CheckNoError(err) \

if ((err) != NULL) { \

fprintf(stderr, "%s:%d: %s: %s\n", \_\_FILE\_\_, \_\_LINE\_\_, phase, (err)); \

abort(); \

}

#define CheckCondition(cond) \

if (!(cond)) { \

fprintf(stderr, "%s:%d: %s: %s\n", \_\_FILE\_\_, \_\_LINE\_\_, phase, #cond); \

abort(); \

}

static void CheckEqual(const char\* expected, const char\* v, size\_t n) {

if (expected == NULL && v == NULL) {

// ok

} else if (expected != NULL && v != NULL && n == strlen(expected) &&

memcmp(expected, v, n) == 0) {

// ok

return;

} else {

fprintf(stderr, "%s: expected '%s', got '%s'\n",

phase,

(expected ? expected : "(null)"),

(v ? v : "(null"));

abort();

}

}

static void Free(char\*\* ptr) {

if (\*ptr) {

free(\*ptr);

\*ptr = NULL;

}

}

static void CheckGet(

leveldb\_t\* db,

const leveldb\_readoptions\_t\* options,

const char\* key,

const char\* expected) {

char\* err = NULL;

size\_t val\_len;

char\* val;

val = leveldb\_get(db, options, key, strlen(key), &val\_len, &err);

CheckNoError(err);

CheckEqual(expected, val, val\_len);

Free(&val);

}

static void CheckIter(leveldb\_iterator\_t\* iter,

const char\* key, const char\* val) {

size\_t len;

const char\* str;

str = leveldb\_iter\_key(iter, &len);

CheckEqual(key, str, len);

str = leveldb\_iter\_value(iter, &len);

CheckEqual(val, str, len);

}

// Callback from leveldb\_writebatch\_iterate()

static void CheckPut(void\* ptr,

const char\* k, size\_t klen,

const char\* v, size\_t vlen) {

int\* state = (int\*) ptr;

CheckCondition(\*state < 2);

switch (\*state) {

case 0:

CheckEqual("bar", k, klen);

CheckEqual("b", v, vlen);

break;

case 1:

CheckEqual("box", k, klen);

CheckEqual("c", v, vlen);

break;

}

(\*state)++;

}

// Callback from leveldb\_writebatch\_iterate()

static void CheckDel(void\* ptr, const char\* k, size\_t klen) {

int\* state = (int\*) ptr;

CheckCondition(\*state == 2);

CheckEqual("bar", k, klen);

(\*state)++;

}

static void CmpDestroy(void\* arg) { }

static int CmpCompare(void\* arg, const char\* a, size\_t alen,

const char\* b, size\_t blen) {

int n = (alen < blen) ? alen : blen;

int r = memcmp(a, b, n);

if (r == 0) {

if (alen < blen) r = -1;

else if (alen > blen) r = +1;

}

return r;

}

static const char\* CmpName(void\* arg) {

return "foo";

}

// Custom filter policy

static unsigned char fake\_filter\_result = 1;

static void FilterDestroy(void\* arg) { }

static const char\* FilterName(void\* arg) {

return "TestFilter";

}

static char\* FilterCreate(

void\* arg,

const char\* const\* key\_array, const size\_t\* key\_length\_array,

int num\_keys,

size\_t\* filter\_length) {

\*filter\_length = 4;

char\* result = malloc(4);

memcpy(result, "fake", 4);

return result;

}

unsigned char FilterKeyMatch(

void\* arg,

const char\* key, size\_t length,

const char\* filter, size\_t filter\_length) {

CheckCondition(filter\_length == 4);

CheckCondition(memcmp(filter, "fake", 4) == 0);

return fake\_filter\_result;

}

int main(int argc, char\*\* argv) {

leveldb\_t\* db;

leveldb\_comparator\_t\* cmp;

leveldb\_cache\_t\* cache;

leveldb\_env\_t\* env;

leveldb\_options\_t\* options;

leveldb\_readoptions\_t\* roptions;

leveldb\_writeoptions\_t\* woptions;

char\* dbname;

char\* err = NULL;

int run = -1;

CheckCondition(leveldb\_major\_version() >= 1);

CheckCondition(leveldb\_minor\_version() >= 1);

StartPhase("create\_objects");

cmp = leveldb\_comparator\_create(NULL, CmpDestroy, CmpCompare, CmpName);

env = leveldb\_create\_default\_env();

cache = leveldb\_cache\_create\_lru(100000);

dbname = leveldb\_env\_get\_test\_directory(env);

CheckCondition(dbname != NULL);

options = leveldb\_options\_create();

leveldb\_options\_set\_comparator(options, cmp);

leveldb\_options\_set\_error\_if\_exists(options, 1);

leveldb\_options\_set\_cache(options, cache);

leveldb\_options\_set\_env(options, env);

leveldb\_options\_set\_info\_log(options, NULL);

leveldb\_options\_set\_write\_buffer\_size(options, 100000);

leveldb\_options\_set\_paranoid\_checks(options, 1);

leveldb\_options\_set\_max\_open\_files(options, 10);

leveldb\_options\_set\_block\_size(options, 1024);

leveldb\_options\_set\_block\_restart\_interval(options, 8);

leveldb\_options\_set\_max\_file\_size(options, 3 << 20);

leveldb\_options\_set\_compression(options, leveldb\_no\_compression);

roptions = leveldb\_readoptions\_create();

leveldb\_readoptions\_set\_verify\_checksums(roptions, 1);

leveldb\_readoptions\_set\_fill\_cache(roptions, 0);

woptions = leveldb\_writeoptions\_create();

leveldb\_writeoptions\_set\_sync(woptions, 1);

StartPhase("destroy");

leveldb\_destroy\_db(options, dbname, &err);

Free(&err);

StartPhase("open\_error");

db = leveldb\_open(options, dbname, &err);

CheckCondition(err != NULL);

Free(&err);

StartPhase("leveldb\_free");

db = leveldb\_open(options, dbname, &err);

CheckCondition(err != NULL);

leveldb\_free(err);

err = NULL;

StartPhase("open");

leveldb\_options\_set\_create\_if\_missing(options, 1);

db = leveldb\_open(options, dbname, &err);

CheckNoError(err);

CheckGet(db, roptions, "foo", NULL);

StartPhase("put");

leveldb\_put(db, woptions, "foo", 3, "hello", 5, &err);

CheckNoError(err);

CheckGet(db, roptions, "foo", "hello");

StartPhase("compactall");

leveldb\_compact\_range(db, NULL, 0, NULL, 0);

CheckGet(db, roptions, "foo", "hello");

StartPhase("compactrange");

leveldb\_compact\_range(db, "a", 1, "z", 1);

CheckGet(db, roptions, "foo", "hello");

StartPhase("writebatch");

{

leveldb\_writebatch\_t\* wb = leveldb\_writebatch\_create();

leveldb\_writebatch\_put(wb, "foo", 3, "a", 1);

leveldb\_writebatch\_clear(wb);

leveldb\_writebatch\_put(wb, "bar", 3, "b", 1);

leveldb\_writebatch\_put(wb, "box", 3, "c", 1);

leveldb\_writebatch\_delete(wb, "bar", 3);

leveldb\_write(db, woptions, wb, &err);

CheckNoError(err);

CheckGet(db, roptions, "foo", "hello");

CheckGet(db, roptions, "bar", NULL);

CheckGet(db, roptions, "box", "c");

int pos = 0;

leveldb\_writebatch\_iterate(wb, &pos, CheckPut, CheckDel);

CheckCondition(pos == 3);

leveldb\_writebatch\_destroy(wb);

}

StartPhase("iter");

{

leveldb\_iterator\_t\* iter = leveldb\_create\_iterator(db, roptions);

CheckCondition(!leveldb\_iter\_valid(iter));

leveldb\_iter\_seek\_to\_first(iter);

CheckCondition(leveldb\_iter\_valid(iter));

CheckIter(iter, "box", "c");

leveldb\_iter\_next(iter);

CheckIter(iter, "foo", "hello");

leveldb\_iter\_prev(iter);

CheckIter(iter, "box", "c");

leveldb\_iter\_prev(iter);

CheckCondition(!leveldb\_iter\_valid(iter));

leveldb\_iter\_seek\_to\_last(iter);

CheckIter(iter, "foo", "hello");

leveldb\_iter\_seek(iter, "b", 1);

CheckIter(iter, "box", "c");

leveldb\_iter\_get\_error(iter, &err);

CheckNoError(err);

leveldb\_iter\_destroy(iter);

}

StartPhase("approximate\_sizes");

{

int i;

int n = 20000;

char keybuf[100];

char valbuf[100];

uint64\_t sizes[2];

const char\* start[2] = { "a", "k00000000000000010000" };

size\_t start\_len[2] = { 1, 21 };

const char\* limit[2] = { "k00000000000000010000", "z" };

size\_t limit\_len[2] = { 21, 1 };

leveldb\_writeoptions\_set\_sync(woptions, 0);

for (i = 0; i < n; i++) {

snprintf(keybuf, sizeof(keybuf), "k%020d", i);

snprintf(valbuf, sizeof(valbuf), "v%020d", i);

leveldb\_put(db, woptions, keybuf, strlen(keybuf), valbuf, strlen(valbuf),

&err);

CheckNoError(err);

}

leveldb\_approximate\_sizes(db, 2, start, start\_len, limit, limit\_len, sizes);

CheckCondition(sizes[0] > 0);

CheckCondition(sizes[1] > 0);

}

StartPhase("property");

{

char\* prop = leveldb\_property\_value(db, "nosuchprop");

CheckCondition(prop == NULL);

prop = leveldb\_property\_value(db, "leveldb.stats");

CheckCondition(prop != NULL);

Free(&prop);

}

StartPhase("snapshot");

{

const leveldb\_snapshot\_t\* snap;

snap = leveldb\_create\_snapshot(db);

leveldb\_delete(db, woptions, "foo", 3, &err);

CheckNoError(err);

leveldb\_readoptions\_set\_snapshot(roptions, snap);

CheckGet(db, roptions, "foo", "hello");

leveldb\_readoptions\_set\_snapshot(roptions, NULL);

CheckGet(db, roptions, "foo", NULL);

leveldb\_release\_snapshot(db, snap);

}

StartPhase("repair");

{

leveldb\_close(db);

leveldb\_options\_set\_create\_if\_missing(options, 0);

leveldb\_options\_set\_error\_if\_exists(options, 0);

leveldb\_repair\_db(options, dbname, &err);

CheckNoError(err);

db = leveldb\_open(options, dbname, &err);

CheckNoError(err);

CheckGet(db, roptions, "foo", NULL);

CheckGet(db, roptions, "bar", NULL);

CheckGet(db, roptions, "box", "c");

leveldb\_options\_set\_create\_if\_missing(options, 1);

leveldb\_options\_set\_error\_if\_exists(options, 1);

}

StartPhase("filter");

for (run = 0; run < 2; run++) {

// First run uses custom filter, second run uses bloom filter

CheckNoError(err);

leveldb\_filterpolicy\_t\* policy;

if (run == 0) {

policy = leveldb\_filterpolicy\_create(

NULL, FilterDestroy, FilterCreate, FilterKeyMatch, FilterName);

} else {

policy = leveldb\_filterpolicy\_create\_bloom(10);

}

// Create new database

leveldb\_close(db);

leveldb\_destroy\_db(options, dbname, &err);

leveldb\_options\_set\_filter\_policy(options, policy);

db = leveldb\_open(options, dbname, &err);

CheckNoError(err);

leveldb\_put(db, woptions, "foo", 3, "foovalue", 8, &err);

CheckNoError(err);

leveldb\_put(db, woptions, "bar", 3, "barvalue", 8, &err);

CheckNoError(err);

leveldb\_compact\_range(db, NULL, 0, NULL, 0);

fake\_filter\_result = 1;

CheckGet(db, roptions, "foo", "foovalue");

CheckGet(db, roptions, "bar", "barvalue");

if (phase == 0) {

// Must not find value when custom filter returns false

fake\_filter\_result = 0;

CheckGet(db, roptions, "foo", NULL);

CheckGet(db, roptions, "bar", NULL);

fake\_filter\_result = 1;

CheckGet(db, roptions, "foo", "foovalue");

CheckGet(db, roptions, "bar", "barvalue");

}

leveldb\_options\_set\_filter\_policy(options, NULL);

leveldb\_filterpolicy\_destroy(policy);

}

StartPhase("cleanup");

leveldb\_close(db);

leveldb\_options\_destroy(options);

leveldb\_readoptions\_destroy(roptions);

leveldb\_writeoptions\_destroy(woptions);

leveldb\_free(dbname);

leveldb\_cache\_destroy(cache);

leveldb\_comparator\_destroy(cmp);

leveldb\_env\_destroy(env);

fprintf(stderr, "PASS\n");

return 0;

}

这段代码的功能是测试的公能。

# corruption\_test.cc

// Copyright (c) 2011 The LevelDB Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file. See the AUTHORS file for names of contributors.

#include "leveldb/db.h"

#include <errno.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include "leveldb/cache.h"

#include "leveldb/env.h"

#include "leveldb/table.h"

#include "leveldb/write\_batch.h"

#include "db/db\_impl.h"

#include "db/filename.h"

#include "db/log\_format.h"

#include "db/version\_set.h"

#include "util/logging.h"

#include "util/testharness.h"

#include "util/testutil.h"

namespace leveldb {

static const int kValueSize = 1000;

class CorruptionTest {

public:

test::ErrorEnv env\_;

std::string dbname\_;

Cache\* tiny\_cache\_;

Options options\_;

DB\* db\_;

CorruptionTest() {

tiny\_cache\_ = NewLRUCache(100);

options\_.env = &env\_;

options\_.block\_cache = tiny\_cache\_;

dbname\_ = test::TmpDir() + "/corruption\_test";

DestroyDB(dbname\_, options\_);

db\_ = nullptr;

options\_.create\_if\_missing = true;

Reopen();

options\_.create\_if\_missing = false;

}

~CorruptionTest() {

delete db\_;

DestroyDB(dbname\_, Options());

delete tiny\_cache\_;

}

Status TryReopen() {

delete db\_;

db\_ = nullptr;

return DB::Open(options\_, dbname\_, &db\_);

}

void Reopen() {

ASSERT\_OK(TryReopen());

}

void RepairDB() {

delete db\_;

db\_ = nullptr;

ASSERT\_OK(::leveldb::RepairDB(dbname\_, options\_));

}

void Build(int n) {

std::string key\_space, value\_space;

WriteBatch batch;

for (int i = 0; i < n; i++) {

//if ((i % 100) == 0) fprintf(stderr, "@ %d of %d\n", i, n);

Slice key = Key(i, &key\_space);

batch.Clear();

batch.Put(key, Value(i, &value\_space));

WriteOptions options;

// Corrupt() doesn't work without this sync on windows; stat reports 0 for

// the file size.

if (i == n - 1) {

options.sync = true;

}

ASSERT\_OK(db\_->Write(options, &batch));

}

}

void Check(int min\_expected, int max\_expected) {

int next\_expected = 0;

int missed = 0;

int bad\_keys = 0;

int bad\_values = 0;

int correct = 0;

std::string value\_space;

Iterator\* iter = db\_->NewIterator(ReadOptions());

for (iter->SeekToFirst(); iter->Valid(); iter->Next()) {

uint64\_t key;

Slice in(iter->key());

if (in == "" || in == "~") {

// Ignore boundary keys.

continue;

}

if (!ConsumeDecimalNumber(&in, &key) ||

!in.empty() ||

key < next\_expected) {

bad\_keys++;

continue;

}

missed += (key - next\_expected);

next\_expected = key + 1;

if (iter->value() != Value(key, &value\_space)) {

bad\_values++;

} else {

correct++;

}

}

delete iter;

fprintf(stderr,

"expected=%d..%d; got=%d; bad\_keys=%d; bad\_values=%d; missed=%d\n",

min\_expected, max\_expected, correct, bad\_keys, bad\_values, missed);

ASSERT\_LE(min\_expected, correct);

ASSERT\_GE(max\_expected, correct);

}

void Corrupt(FileType filetype, int offset, int bytes\_to\_corrupt) {

// Pick file to corrupt

std::vector<std::string> filenames;

ASSERT\_OK(env\_.GetChildren(dbname\_, &filenames));

uint64\_t number;

FileType type;

std::string fname;

int picked\_number = -1;

for (size\_t i = 0; i < filenames.size(); i++) {

if (ParseFileName(filenames[i], &number, &type) &&

type == filetype &&

int(number) > picked\_number) { // Pick latest file

fname = dbname\_ + "/" + filenames[i];

picked\_number = number;

}

}

ASSERT\_TRUE(!fname.empty()) << filetype;

struct stat sbuf;

if (stat(fname.c\_str(), &sbuf) != 0) {

const char\* msg = strerror(errno);

ASSERT\_TRUE(false) << fname << ": " << msg;

}

if (offset < 0) {

// Relative to end of file; make it absolute

if (-offset > sbuf.st\_size) {

offset = 0;

} else {

offset = sbuf.st\_size + offset;

}

}

if (offset > sbuf.st\_size) {

offset = sbuf.st\_size;

}

if (offset + bytes\_to\_corrupt > sbuf.st\_size) {

bytes\_to\_corrupt = sbuf.st\_size - offset;

}

// Do it

std::string contents;

Status s = ReadFileToString(Env::Default(), fname, &contents);

ASSERT\_TRUE(s.ok()) << s.ToString();

for (int i = 0; i < bytes\_to\_corrupt; i++) {

contents[i + offset] ^= 0x80;

}

s = WriteStringToFile(Env::Default(), contents, fname);

ASSERT\_TRUE(s.ok()) << s.ToString();

}

int Property(const std::string& name) {

std::string property;

int result;

if (db\_->GetProperty(name, &property) &&

sscanf(property.c\_str(), "%d", &result) == 1) {

return result;

} else {

return -1;

}

}

// Return the ith key

Slice Key(int i, std::string\* storage) {

char buf[100];

snprintf(buf, sizeof(buf), "%016d", i);

storage->assign(buf, strlen(buf));

return Slice(\*storage);

}

// Return the value to associate with the specified key

Slice Value(int k, std::string\* storage) {

Random r(k);

return test::RandomString(&r, kValueSize, storage);

}

};

TEST(CorruptionTest, Recovery) {

Build(100);

Check(100, 100);

Corrupt(kLogFile, 19, 1); // WriteBatch tag for first record

Corrupt(kLogFile, log::kBlockSize + 1000, 1); // Somewhere in second block

Reopen();

// The 64 records in the first two log blocks are completely lost.

Check(36, 36);

}

TEST(CorruptionTest, RecoverWriteError) {

env\_.writable\_file\_error\_ = true;

Status s = TryReopen();

ASSERT\_TRUE(!s.ok());

}

TEST(CorruptionTest, NewFileErrorDuringWrite) {

// Do enough writing to force minor compaction

env\_.writable\_file\_error\_ = true;

const int num = 3 + (Options().write\_buffer\_size / kValueSize);

std::string value\_storage;

Status s;

for (int i = 0; s.ok() && i < num; i++) {

WriteBatch batch;

batch.Put("a", Value(100, &value\_storage));

s = db\_->Write(WriteOptions(), &batch);

}

ASSERT\_TRUE(!s.ok());

ASSERT\_GE(env\_.num\_writable\_file\_errors\_, 1);

env\_.writable\_file\_error\_ = false;

Reopen();

}

TEST(CorruptionTest, TableFile) {

Build(100);

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

dbi->TEST\_CompactRange(0, nullptr, nullptr);

dbi->TEST\_CompactRange(1, nullptr, nullptr);

Corrupt(kTableFile, 100, 1);

Check(90, 99);

}

TEST(CorruptionTest, TableFileRepair) {

options\_.block\_size = 2 \* kValueSize; // Limit scope of corruption

options\_.paranoid\_checks = true;

Reopen();

Build(100);

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

dbi->TEST\_CompactRange(0, nullptr, nullptr);

dbi->TEST\_CompactRange(1, nullptr, nullptr);

Corrupt(kTableFile, 100, 1);

RepairDB();

Reopen();

Check(95, 99);

}

TEST(CorruptionTest, TableFileIndexData) {

Build(10000); // Enough to build multiple Tables

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

Corrupt(kTableFile, -2000, 500);

Reopen();

Check(5000, 9999);

}

TEST(CorruptionTest, MissingDescriptor) {

Build(1000);

RepairDB();

Reopen();

Check(1000, 1000);

}

TEST(CorruptionTest, SequenceNumberRecovery) {

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v1"));

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v2"));

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v3"));

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v4"));

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v5"));

RepairDB();

Reopen();

std::string v;

ASSERT\_OK(db\_->Get(ReadOptions(), "foo", &v));

ASSERT\_EQ("v5", v);

// Write something. If sequence number was not recovered properly,

// it will be hidden by an earlier write.

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "v6"));

ASSERT\_OK(db\_->Get(ReadOptions(), "foo", &v));

ASSERT\_EQ("v6", v);

Reopen();

ASSERT\_OK(db\_->Get(ReadOptions(), "foo", &v));

ASSERT\_EQ("v6", v);

}

TEST(CorruptionTest, CorruptedDescriptor) {

ASSERT\_OK(db\_->Put(WriteOptions(), "foo", "hello"));

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

dbi->TEST\_CompactRange(0, nullptr, nullptr);

Corrupt(kDescriptorFile, 0, 1000);

Status s = TryReopen();

ASSERT\_TRUE(!s.ok());

RepairDB();

Reopen();

std::string v;

ASSERT\_OK(db\_->Get(ReadOptions(), "foo", &v));

ASSERT\_EQ("hello", v);

}

TEST(CorruptionTest, CompactionInputError) {

Build(10);

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

const int last = config::kMaxMemCompactLevel;

ASSERT\_EQ(1, Property("leveldb.num-files-at-level" + NumberToString(last)));

Corrupt(kTableFile, 100, 1);

Check(5, 9);

// Force compactions by writing lots of values

Build(10000);

Check(10000, 10000);

}

TEST(CorruptionTest, CompactionInputErrorParanoid) {

options\_.paranoid\_checks = true;

options\_.write\_buffer\_size = 512 << 10;

Reopen();

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

// Make multiple inputs so we need to compact.

for (int i = 0; i < 2; i++) {

Build(10);

dbi->TEST\_CompactMemTable();

Corrupt(kTableFile, 100, 1);

env\_.SleepForMicroseconds(100000);

}

dbi->CompactRange(nullptr, nullptr);

// Write must fail because of corrupted table

std::string tmp1, tmp2;

Status s = db\_->Put(WriteOptions(), Key(5, &tmp1), Value(5, &tmp2));

ASSERT\_TRUE(!s.ok()) << "write did not fail in corrupted paranoid db";

}

TEST(CorruptionTest, UnrelatedKeys) {

Build(10);

DBImpl\* dbi = reinterpret\_cast<DBImpl\*>(db\_);

dbi->TEST\_CompactMemTable();

Corrupt(kTableFile, 100, 1);

std::string tmp1, tmp2;

ASSERT\_OK(db\_->Put(WriteOptions(), Key(1000, &tmp1), Value(1000, &tmp2)));

std::string v;

ASSERT\_OK(db\_->Get(ReadOptions(), Key(1000, &tmp1), &v));

ASSERT\_EQ(Value(1000, &tmp2).ToString(), v);

dbi->TEST\_CompactMemTable();

ASSERT\_OK(db\_->Get(ReadOptions(), Key(1000, &tmp1), &v));

ASSERT\_EQ(Value(1000, &tmp2).ToString(), v);

}

} // namespace leveldb

int main(int argc, char\*\* argv) {

return leveldb::test::RunAllTests();

}

corruption\_test的功能是检测Data Base是否有损坏（corruption）。

在我报告当中我只能给老师提供一些代码的主要功能分析，因为这软件有点大，功能也很多，上面我分析了一些主要的功能。它的注释已经解释了函数的功能。其实这软件是谷歌用C++开发的软件，代码量也大概5000多。对我来说这软件挺好的。