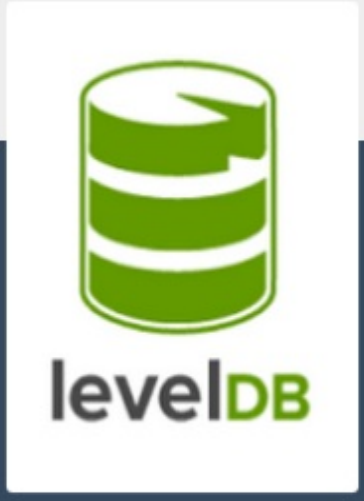
LevelDB中文文档



LevelDB提供一个持久的键值存储库。keys 和 values 都可以是任意的字节数组。key是通过用户提供的比较器函数在键值存储器内进行排序的。

目　录

致谢

当前文档 《LevelDB中文文档》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-02-23。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：kevins.pro https://kevins.pro/leveldb_chinese_doc.html

文档地址：http://www.bookstack.cn/books/leveldb_chinese_doc

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

LevelDB中文文档

> 欢迎转载，转载请注明出处！原文地址 Follow me on GitHub ^_^

> 作者: Jeff Dean, Sanjay Ghemawat原文: https://rawgit.com/google/leveldb/master/doc/index.html译者: KevinsBobo

LevelDB提供一个持久的键值存储库。keys 和 values 都可以是任意的字节数组。key是通过用户提供的比较器函数在键值存储器内进行排序的。

创建并打开一个数据库 - Opening A Database

一个LevelDB数据库有一个名字并且对应一个文件系统的文件夹。所有的数据库内容都存储在这个文件夹下。在下面的例子中展示了怎样创建并打开一个数据库：

```
#include <cassert>
#include "leveldb/db.h"

leveldb::DB* db;
leveldb::Options options;
options.create_if_missing = true;
leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
assert(status.ok());
...
```

如果想要在存在已创建数据库的情况下引发错误，则在 `leveldb::DB::Open` 前加上

```
options.error_if_exists = true;
```

状态 - Status

你可能已经注意到了上面的 `leveldb::Status` 这个类型。在 `leveldb` 中可能遇到错误的函数大多都返回这个类型的值。你可以检查返回的结果是不是正确执行，并且可以打印相关联的错误信息：

```
1.  leveldb::Status s = ...;
2.  if (!s.ok()) cerr << s.ToString() << endl;
```

关闭一个数据库 Closing A Database

当你完成数据库相关操作后，只用删除这个数据库对象就可以了。例:

```
1.  ... open the db as described above ...
2.  ... do something with db ...
3.  delete db;
```

读写操作 - Reads And Writes

数据库提供 `Put, Delete, Get` 这些方法来修改和查询数据库。比如，以下操作时将存储在 `key1` 的值移动到 `key2` 中去：

```
1.  std::string value;
2.  leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);
3.  if (s.ok()) s = db->Put(leveldb::WriteOptions(), key2, value);
4.  if (s.ok()) s = db->Delete(leveldb::WriteOptions(), key1);
```

原子更新 - Atomic Updates

注意：上面的操作如果进程在 `Put key2` 和 `Delete key1` 两个操作之间结束，那么这两个键将存储相同的值。因此，尽可能使用 `WriteBatch` 类来避免这类问题：

```
#include "leveldb/write_batch.h"
...
std::string value;
leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);
if (s.ok()) {
  leveldb::WriteBatch batch;
  batch.Delete(key1);
  batch.Put(key2, value);
  s = db->Write(leveldb::WriteOptions(), &batch);
}
```

`WriteBatch` 对象保存对数据库进行的一系列操作，然后在这一批次中按照顺序应用这些操作。注意：这里先进行 `Delete` 操作，然后再进行 `Put` 操作，是因为在 `key1` 和 `key2` 相同的情况下不会错误的将该值丢弃。

`WriteBatch` 类除了原子性的优势外，也可以用于通过将大量个体变动放置在同一批次中而加速批量更新。

同步写入 - Synchronous Writes

在默认情况下， `levedb` 中的每一次写操作都是异步的：它会在把写入操作从进程中推送到操作系统后返回，而从操作系统内存到底层持久化存储的传输是异步的。对于特定的写操作，是可以打开同步 `sync` 标志使写操作一直到数据被传输到底层存储器后再返回。（在基于Posix标准的操作系统系统中，这一步是通过在写操作返回之前调用 `fsync(…)` 或 `fdatasync(…)` 或 `nsync(…, MS_SYNC)` 实现的。）

```
1. leveldb::WriteOptions write_options;
2. write_options.sync = true;
3. db->Put(write_options, ...);
```

异步写操作一般比同步写操作快很多很多。但异步写入的缺点是，在机器宕机时有可能导致最后几步的更新丢失。请注意，如果是在写入过程中的宕机（而非重新启动），即使 `sync` 设置为 `false` ，更新操作也会认为已经将更新从内存中推送到了操作系统。

通常可以安全地使用异步写入。比如，当加载大量数据到数据库中时，可以通过在宕机后重新启动批量加载来处理丢失的更新。有一个可用的混合方案，将多次写入的第N次写入设置为同步的，并在宕机重启后的情况下，批量加载由前一次运行的最后一次同步写入之后重新开始。（同步写入时可以更新描述宕机后批量加载重新开始的标记。）

`WriteBatch` 提供一个代替异步写操作的选择：将多个更新操作放置在同一个 `WriteBatch` 对象中然后使用同步写入一起应用（ `write_options.sync` 设置为 `ture` ）；这时同步写入的额外成本开销将在这批次中所有的写入操作中摊销。

并发 - Concurrency

一个数据库在同一时间内只能由一个进程打开， `leveldb` 通过从操作系统中获取锁的方式防止误用。在单个进程中，同一个 `leveldb::DB` 对象可以安全的由多个并发线程共享使用。即，不同的线程可以同时写入或获取迭代器，或在没有任何外部同步的情况在同一数据库上调用 `Get` （ `leveldb` 的实现过程中将自动自行所需的同步）。但是其他对象（如 `Iterator` 和 `WriteBatch` ）可能需要外部同步。如果两个线程共享这样的对象，它们必须使用自己的协议锁来保护自己的访问。更多详细的细节在公共的头文件中描述。

迭代 - Iteration

下面的例子演示了如何从数据库中成对的打印键值：

```
leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());
for (it->SeekToFirst(); it->Valid(); it->Next()) {
  cout << it->key().ToString() << ": "  << it->value().ToString() << endl;
}
assert(it->status().ok());  // Check for any errors found during the scan
delete it;
```

下面的修改后的例子演示了如何仅获取 `[start, limit)` 范围内的键；

```
for (it->Seek(start);
     it->Valid() && it->key().ToString() < limit;
     it->Next()) {
  ...
}
```

还可以通过相反的顺序进行处理。（警告：反向迭代比正向迭代慢一些）

```
for (it->SeekToLast(); it->Valid(); it->Prev()) {
  ...
}
```

快照 - Snapshots

快照在键值存储的整体状态上提供了一致的只读视图。 `ReadOptions::snapshot` 可能是 `non-NULL` ，表示读取操作在特定的 `DB` 版本状态上进行的。如果 `ReadOptions::snapshot` 是 `NULL` ，则读取操作将在当前状态上进行隐式的快照操作。

快照是通过 `DB::GetSnapshot` 方法进行创建的:

```
leveldb::ReadOptions options;
options.snapshot = db->GetSnapshot();
... apply some updates to db ...
leveldb::Iterator* iter = db->NewIterator(options);
... read using iter to view the state when the snapshot was created ...
delete iter;
db->ReleaseSnapshot(options.snapshot);
```

注意: 当一个快照长期不用时，应该通过 `DB::ReleaseSnapshot` 接口释放它。这样既可以让底层实现丢弃那些为支持该快照的读取操作而进行维护的一些状态数据。

Slice - LevelDB使用的数据结构

前面遇到的 `it->key()` 和 `it->value()` 调用的返回值就是 `leveldb::Slice` 类型的实例。 `Slice` 是一个简单的结构，它包含了一个 `length` 和一个指向外部字节数组的指针。返回 `Slice` 类型要比返回 `std::string` 类型的开销小得多，因为这样我们就不需要对那些比较大的键值进行拷贝了。此外， `leveldb` 方法不返回以 `nul` 结尾的C风格字符串，因为 `leveldb` 的键和值允许包含 `\0` 字符。

C++字符串和C风格字符串能够很容易的转换为 `Slice` 类型：

```
1.  leveldb::Slice s1 = "hello";
2.
3.  std::string str("world");
4.  leveldb::Slice s2 = str;
```

一个 `Slice` 类型也很容易的就能转换回C++字符串：

```
1.  std::string str = s1.ToString();
2.  assert(str == std::string("hello"));
```

在使用 `Slice` 类型时要格外小心，因为它依赖调用者来保证 `Slice` 指向的外部字符数组有效。比如下面这个例子就是有问题的：

```
1.  leveldb::Slice slice;
2.  if (...) {
3.    std::string str = ...;
4.    slice = str;
5.  }
6.  Use(slice);
```

因为 `if` 语句块是有作用域的，所以当 `if` 语句执行完后 `str` 将会被析构，此时 `slice` 指向的空间就不存在了。

比较器 - Comparators

前面的例子使用了按照字典序的默认排序函数对 `key` 进行排序。然而，你也可以在打开一个数据库时为其提供一个自定义的比较器。例如，假设数据库的每个 `key` 由两个数字著称，我们应该先按照第一个数字排序，如果相等再按照第二个数字进行排序。首先，定义一个满足如下规则的 `leveldb::Conmparator` 的子类：

```
class TwoPartComparator : public leveldb::Comparator {
 public:
  // Three-way comparison function:
  //   if a < b: negative result
  //   if a > b: positive result
  //   else: zero result
  int Compare(const leveldb::Slice& a, const leveldb::Slice& b) const {
    int a1, a2, b1, b2;
    ParseKey(a, &a1, &a2);
    ParseKey(b, &b1, &b2);
    if (a1 < b1) return -1;
    if (a1 > b1) return +1;
    if (a2 < b2) return -1;
    if (a2 > b2) return +1;
    return 0;
  }

  // Ignore the following methods for now:
  const char* Name() const { return "TwoPartComparator"; }
  void FindShortestSeparator(std::string*, const leveldb::Slice&) const { }
  void FindShortSuccessor(std::string*) const { }
};
```

现在使用这个自定义的比较器创建数据库：

```
TwoPartComparator cmp;
leveldb::DB* db;
leveldb::Options options;
options.create_if_missing = true;
options.comparator = &cmp;
leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
...
```

向后兼容性 - Backwards compatibility

比较器的 `Name` 方法的返回值将会在数据库创建时与之绑定，并且在以后每次打开数据库的时候进行检查。如果 `name` 发生变化，那么 `leveldb::DB::Open` 将会调用失败。因此，只有在新的 `key` 格式及比较函数和现在的数据库不兼容时才需要修改 `name` ，同时所有现有的数据库数据将会被丢弃。

当然，你也可以通过预先的计划来逐步改变你的 `key` 格式。例如，你可以在每个 `key` 的末尾存储一个版本号（一个字节的应该可以满足大多数用途）。当希望使用一种新的 `key` 格式时（比如，给 `TwoPartComparator` 增加一个可选的第三块内容），(a) 保持 `comparator` 的 `name` 不变，(b) 给新的 `key` 增加版本号，(c) 改变比较器函数，使得它可以通过 `key` 里的版本号来决定如何解释它们。

性能 - Performance

可以通过修改定义在 `include/leveldb/options.h` 中的默认值值来对性能进行调整和优化。

Block size - 块大小

`leveldb` 将相邻的 `key` 组合在一块儿放进同一个 `block` 中，这样的一个 `block` 是与持久化存储设备进行传输的基本单元。默认的 `block` 大小约为 `4096` 个未压缩字节。那些经常需要扫描整个数据库内容的应用可能希望增加这个值的大小。对于小的 `value` 值进行大量的单点读取的应用，想要改进性能的话可以尝试将这个值减小。在这个值小于1千字节或大于几兆字节并没有太多的好处。还要注意，压缩对于那些比较大的 `block` 更有效一些。

Compression - 压缩

每个 `block` 在被写入持久化存储器之前都会被单独压缩。由于默认的压缩方法速度非常快，并且对不可压缩的数据禁用压缩，因此压缩默认的状态是打开的。只有在极少数的情况下，应用才会完全关闭压缩，但只有在通过 `benchmarks` 能看到性能提升时才应该这样做：

```
1. leveldb::Options options;
2. options.compression = leveldb::kNoCompression;
3. ... leveldb::DB::Open(options, name, ...) ....
```

Cache - 缓存

数据库的内容存储在文件系统的一组文件中，并且每个文件存储的都是一系列压缩过的 `blocks` 。如果 `options.cache` 的值时 `non-NULL` 的，那么那些用过的未被压缩的 `block` 的内容将被存储在缓存中。

```
1. #include "leveldb/cache.h"
2.
3. leveldb::Options options;
4. options.cache = leveldb::NewLRUCache(100 * 1048576);  // 100MB cache
5. leveldb::DB* db;
6. leveldb::DB::Open(options, name, &db);
7. ... use the db ...
8. delete db
9. delete options.cache;
```

注意，缓存里存放的时未压缩的数据，因此它的大小是应用层面的数据大小，而不是压缩后的。（压缩过的 `block` 是交给操作系统缓冲区去缓存的，或是由客户端提供的自定义的 `Env` 实现来完成

的）。

当执行大批量读取操作时，应用程序可能会希望禁用高速缓存，从而不会由于大批量的数据读取操作而消耗大量的缓存。一个针对迭代器的 `option` 参数可以实现这个目的：

```
leveldb::ReadOptions options;
options.fill_cache = false;
leveldb::Iterator* it = db->NewIterator(options);
for (it->SeekToFirst(); it->Valid(); it->Next()) {
  ...
}
```

Key Layout - 键的布局方式

需要注意硬盘的传输和缓存的单位时一个 `block` 。相邻的 `key` （跟据数据库的排序顺序）通常放置在同一个块中。因此，应用可以通过将相邻的 `key` 放置在一起进行访问、将不经常使用的 `key` 放置在单独的 `key值` 空间内的方法来提高性能。

例如，假设我们在 `leveldb` 上实现一个简单的文件系统。我们通常需要存储的数据应该是：

```
filename -> permission-bits, length, list of file_block_ids
file_block_id -> data
```

我们可以为 `filename` 添加个前缀（比如'/'），为 `file_block_id` 添加个前缀（比如'0'），这样在扫描文件元数据时就不需要去获取和缓存大量的文件内容。

Filters - 过滤器

由于 `leveldb` 的数据是按组存放在硬盘上的，所以一个 `Get()` 调用可能需要在硬盘上执行多次读取操作。可选的 `FilterPolicy` 机制可以显著的减少磁盘的读取操作量。

```
leveldb::Options options;
options.filter_policy = NewBloomFilterPolicy(10);
leveldb::DB* db;
leveldb::DB::Open(options, "/tmp/testdb", &db);
... use the database ...
delete db;
delete options.filter_policy;
```

上面的代码将基于Bloom Filter的过滤策略与数据库相关联。基于 `Bloom Filter` 的过滤依赖于在内存中每个 `key` 中保存一定量数据位（在上面的例子中 `NewBloomFilterPolicy` 的参数是10，说明每个 `key` 中增加的数据位是10位）。此过滤器将调用 `Get()` 时所需的不必要的磁盘读取操作数

量减少了大约100倍。增加每个 `key` 的数据位能过更多的减少硬盘的读取操作数，但是要以更多的内存开销为代价。因此我们建议那些数据不适合在内存中存放的应用和需要做很多随机读取操作的应用设置过滤器策略。

如果你使用的是自定义的比较器，那么应该确保你使用的过滤器策略和自定义比较器相兼容。例如，在一个比较器中比较 `key` 时忽略其尾部的空格，那么 `NewBloomFilterPolicy` 就不能和这样的比较器兼容。相应的，应用也可以提供一个忽略 `key` 尾部空格的自定义过滤器策略。例如：

```
1.  class CustomFilterPolicy : public leveldb::FilterPolicy {
2.   private:
3.    FilterPolicy* builtin_policy_;
4.   public:
5.    CustomFilterPolicy() : builtin_policy_(NewBloomFilterPolicy(10)) { }
6.    ~CustomFilterPolicy() { delete builtin_policy_; }
7.
8.    const char* Name() const { return "IgnoreTrailingSpacesFilter"; }
9.
10.   void CreateFilter(const Slice* keys, int n, std::string* dst) const {
11.     // Use builtin bloom filter code after removing trailing spaces
12.     std::vector<Slice> trimmed(n);
13.     for (int i = 0; i < n; i++) {
14.       trimmed[i] = RemoveTrailingSpaces(keys[i]);
15.     }
16.     return builtin_policy_->CreateFilter(&trimmed[i], n, dst);
17.   }
18.
19.   bool KeyMayMatch(const Slice& key, const Slice& filter) const {
20.     // Use builtin bloom filter code after removing trailing spaces
21.     return builtin_policy_->KeyMayMatch(RemoveTrailingSpaces(key), filter);
22.   }
23. };
```

更高级的应用可以使用一系列其他的用于概括一组 `key` 的机制的过滤策略而不使用 `bloom filter` 过滤策略。更多的细节参见 `leveldb/filter_policy.h` 。

校验和 - Checksums

`leveldb` 会为所有存储在文件系统中的数据生成 `checksums` 。提供了两个参数来控制进行什么程度的 `checksums` 验证：

- `ReadOptions::verify_checksums` ，这个参数设置为 `ture` 代表了对从文件系统读取的所有数据进行强制的 `checksum` 验证。但默认情况下不会进行这样的强制验证。
- `Options::paranoid_checks` ，这个参数可以在打开数据库之前设置为 `ture` ，这会在打开数据库时只要检测到内部损坏的话引发错误。错误产生的时机取决与数据库出现问题的部分，可能在打开数据库时引发或者在后面执行到某些数据库操作时引发。在默认情况下是不执行检查的，即使数据库的持久化存储器某些部分出现问题也可以继续使用数据库。如果数据库坏了（也许是因为设置了 `paranoid_checks` 导致数据库无法打开），`leveldb::RepairDB` 函数可以用来恢复尽可能多的数据。

近似大小 - Approximate Size

`GetApproximateSizes` 方法可以用于获取一个或多个 `key range` 占用的文件系统空间的近似大小：

```
1. leveldb::Range ranges[2];
2. ranges[0] = leveldb::Range("a", "c");
3. ranges[1] = leveldb::Range("x", "z");
4. uint64_t sizes[2];
5. leveldb::Status s = db->GetApproximateSizes(ranges, 2, sizes);
```

上面的调用中会将 `size[0]` 设置为 `[a, c)` 范围内 `key` 占用的近似大小，而将 `size[1]` 设置为 `[x, z)` 范围内 `key` 占用的近似大小。

环境 - Environment

由 `leveldb` 执行的所有文件操作（和其他操作系统的调用操作）都通过 `leveldb::Env` 对象统一管理。高级的客户端可以自己提供 `Env` 来实现更好的控制。例如，应用可以在文件IO中引入人为的延迟来限制 `leveldb` 对系统中其他活动的影响：

```
1.  class SlowEnv : public leveldb::Env {
2.    .. implementation of the Env interface ...
3.  };
4.
5.  SlowEnv env;
6.  leveldb::Options options;
7.  options.env = &env;
8.  Status s = leveldb::DB::Open(options, ...);
```

移植 - Porting

`leveldb` 通过提供 `leveldb/prot/port.h` 中的 `types/methods/functions` 的平台描述来实现将其移植到新的平台上。具体细节可以参考 `leveldb/prot/port_example.h`

其他信息 - Other Information

关于 `leveldb` 的更多实现信息可以参考下面的文档

- Implementation notes
- LevelDB实现(译)
- Format of an immutable Table file
- Format of a log file