[leveldb简介 1](#_Toc465413045)

[leveldb局限性 1](#_Toc465413046)

[打开数据库 1](#_Toc465413047)

[状态 2](#_Toc465413048)

[关闭数据库 2](#_Toc465413049)

[读写操作 2](#_Toc465413050)

[原子更新（Atomic Updates） 2](#_Toc465413051)

[同步写（Synchronous Writes） 2](#_Toc465413052)

[并发（Concurrency） 3](#_Toc465413053)

[迭代（Iteration） 3](#_Toc465413054)

[快照（Snapshots） 3](#_Toc465413055)

[片（Slice） 4](#_Toc465413056)

[比较器（Comparators） 4](#_Toc465413057)

## leveldb简介

leveldb是google实现的单机版k-v存储[系统](http://www.2cto.com/os/)，可按照字符串键值顺序映射进行存储，具有以下几个特点

1. key和value都是任意的字节数组，支持内存和持久化存储

2. 数据都是按照key排序

3. 用户可以重写排序函数

4. 包含基本的数据操作接口，Put(key,value)，Get(key)，Delete(key)

5. 多操作可以当成一次原子操作

6. 用户可以通过生成snapshot，使得读取操作不受写操作影响，读取过程中看到最终数据一致性

7. 支持迭代器对数据的操作

## leveldb局限性

1. leveldb非关系型[数据库](http://www.2cto.com/database/)，不支持SQL查询也不支持索引

2. 同一时间只支持单进程(支持多线程)访问db

3. 不支持客户端-服务器模型，用户需要自己封装

## 打开数据库

       leveldb[数据库](http://lib.csdn.net/base/14)名称与文件系统目录相同，数据库中所有的数据都是存储在该目录下。接下来的例子就展示如何打开一个数据库，必要时创建它：

1. #include <assert>
2. #include "leveldb/db.h"
4. leveldb::DB\* db;
5. leveldb::Options options;
6. options.create\_if\_missing = **true**;
7. leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
8. assert(status.ok());

       如果你认为在数据库存在的情况下创建数据库是一个错误，则可以在打开数据库前在代码中添加一行代码如下：

1. options.error\_if\_exists = **true**;

## 状态

       你可能已经注意到：以上代码中有leveldb::Status类型，这种类型的值被作为大多数leveldb函数的返回值，以用来标记一个错误。你可以用它来判断函数是否执行成功，也可以打印相关的错误信息。

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

## 关闭数据库

       当对数据库操作完毕后，释放database对象，如下所示：

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

## 读写操作

       leveldb提供了Put，Delete和Get函数来修改/查询数据库，例如，以下代码将key1的value复制到key2上。

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，如果一个进程在插入了key2之后并且在删除key1之前终止，相同的值可能会存储在多个键。这样的问题可以通过使用WriteBatch类来进行一系列的原子更新。

1. #include "leveldb/write\_batch.h"
2. ...
3. std::string value;
4. leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);
5. **if** (s.ok()) {
6. leveldb::WriteBatch batch;
7. batch.Delete(key1);
8. batch.Put(key2, value);
9. s = db->Write(leveldb::WriteOptions(), &batch);
10. }

       这个WriteBatch类包含了对数据库进行的一系列操作，并且这些操作是有序的。注意我们称之为在Put之前Delete，这样是为了如果key1和key2是相同的，我们最终不会错误的删除其对应的值。除了它的原子性的好处外，writebatch也可以用来加快处理大量的updates，在同一批中添加独立的update。

## 同步写（Synchronous Writes）

       默认情况下，每次写LevelDB是异步的：它在写操作从当前进程传递到操作系统后就返回了，数据从系统内存到底层持久存储是异步发生的。同步标志可以被打开，此时是为了确保写操作的数据被写入到持久存储中后才返回。（POSIX系统，这是通过调用fsync实施（…）或fdatasync（…）或msync（…，ms\_sync）写操作返回之前。）

1. leveldb::WriteOptions write\_options;
2. write\_options.sync = **true**;
3. db->Put(write\_options, ...);

       异步写入速度通常是同步写入的一千多倍，异步写入的缺点是，机器崩溃可能导致最后几次更新丢失。请注意，一个崩溃的写作过程（即，不是一个重新启动）不会造成任何损失，即使同步写入失败了，在数据真正写入持久存储前该更新是先从进程内存传递到操作系统的。

       异步写入可以安全地使用。例如，当往数据库中写入大量数据时，你可以处理丢失的更新通过在崩溃后重启大容量的写入操作。一种混合方案可能是每N个写是同步的，并且在发生崩溃事件时，批量写入重新启动运是在最后的同步写之前完成的。（同步写可以更新一个标记，说明在崩溃的地方重新启动。）

       writebatch提供了一种替代异步写。多个更新可被放在同一个writebatch中一起使用一个同步写（即write\_options.sync设置为true）。该同步写的额外成本将被分摊到所有写操作中。

## 并发（Concurrency）

       一个数据库只能一次被一个进程打开，Leveldb的实现是从操作系统获取一个锁来防止误操作。在单一进程中，这个leveldb::DB对象可以被通过并发的线程安全地共享。即在没有任何外部同步情况下，不同的线程可以写入、读取迭代器或获得相同的数据库（LevelDB实现会自动同步所需的）。然而其他对象（如迭代器和writebatch）可能需要外部同步，如果2个线程共享这样的对象，他们必须使用他们自己的锁定协议来同步访问，更多的细节可在公共头文件。

## 迭代（Iteration）

       接下来的例子展示如何打印数据库中的所有键值信息：

1. leveldb::Iterator\* it = db->NewIterator(leveldb::ReadOptions());
2. **for** (it->SeekToFirst(); it->Valid(); it->Next()) {
3. cout << it->key().ToString() << ": "  << it->value().ToString() << endl;
4. }
5. assert(it->status().ok());  // Check for any errors found during the scan
6. **delete** it;

       接下来的版本展示如何处理键在范围[start, limit)的数据：

1. **for** (it->Seek(start);
2. it->Valid() && it->key().ToString() < limit;
3. it->Next()) {
4. ...
5. }

       当然，你也可以逆序输出键值信息（注意：逆向迭代可能比正向迭代稍慢）：

1. **for** (it->SeekToLast(); it->Valid(); it->Prev()) {
2. ...
3. }

## 快照（Snapshots）

       快照提供了对整个键值状态一致的只读版本，ReadOptions::snapshot是!NULL的话，表示在特定版本的数据库状态上进行读操作；ReadOptions::snapshot是NULL的话，表示在隐式版本的数据库状态上进行读操作。Snapshots是通过DB::GetSnapshot()方法来创建的。

1. leveldb::ReadOptions options;
2. options.snapshot = db->GetSnapshot();
3. ... apply some updates to db ...
4. leveldb::Iterator\* iter = db->NewIterator(options);
5. ... read **using** iter to view the state when the snapshot was created ...
6. **delete** iter;
7. db->ReleaseSnapshot(options.snapshot);

       注意，当快照不再需要，应该使用DB::releasesnapshot接口来释放它，这可以摆脱持续的读取快照操作。

## 片（Slice）

       it->key()和it->value()的返回值被称为leveldb::Slice类型，Slice是一个简单的shu结构，它包含一个长度和一个外部字节数组的指针。返回一个Slice比返回一个标准的std::string类型更有效率，因为我们不需要复制潜在的大的键和值。此外，leveldb方法不返回'\0'结尾的C-type字符串，这样的话，leveldb的键值就可以包含'\0'数据了。

       C++的strings和以'\0'结尾的C-type字符串也已很容易转化为Slice类型：

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

       一个Slice类型很容易转换回C++ string类型：

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

       使用Slice类型是需注意，要保证其内部的字节数组指针指向内容是可用的，以下代码作为错误示例：

1. leveldb::Slice slice;
2. **if** (...) {
3. std::string str = ...;
4. slice = str;
5. }
6. // 此时slice所指内存已经不可用，有可能发生未知错误
7. Use(slice);

## 比较器（Comparators）

       前面的例子对键使用的是默认排序函数，即字典序排序。当打开数据库时，你可以提供自定义比较器。例如，假设每个数据库的键包含2个数字，我们首先按照第一个数字进行排序，相同时在按照第二个数字排序。首先，定义一个leveldb::Comparator的子类，如下所示：

1. **class** TwoPartComparator : **public** leveldb::Comparator {
2. **public**:
3. // Three-way comparison function:
4. //   if a < b: negative result
5. //   if a > b: positive result
6. //   else: zero result
7. **int** Compare(**const** leveldb::Slice& a, **const** leveldb::Slice& b) **const** {
8. **int** a1, a2, b1, b2;
9. ParseKey(a, &a1, &a2);
10. ParseKey(b, &b1, &b2);
11. **if** (a1 < b1) **return** -1;
12. **if** (a1 > b1) **return** +1;
13. **if** (a2 < b2) **return** -1;
14. **if** (a2 > b2) **return** +1;
15. **return** 0;
16. }
18. // Ignore the following methods for now:
19. **const** **char**\* Name() **const** { **return** "TwoPartComparator"; }
20. **void** FindShortestSeparator(std::string\*, **const** leveldb::Slice&) **const** { }
21. **void** FindShortSuccessor(std::string\*) **const** { }
22. };

       现在创建一个使用自定义比较器的数据库：

1. TwoPartComparator cmp;
2. leveldb::DB\* db;
3. leveldb::Options options;
4. options.create\_if\_missing = **true**;
5. options.comparator = &cmp;
6. leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
7. ...