DBDB: Dog Bed DataBase

作者简介

Taavi Burns 是Countermeasure 乐团中最新的男低音(有时是男高音)。Taavi致力于打破常规。比如说他参加过的工作有，IBM(做C和Perl)，FreshBooks(做所有的东西)，Points.com(做Python)，现在在PagerDuty(做Scala)。 除此之外 - 当他没有在玩他的Brompton牌折叠式自行车时，你可能会发现他和他的儿子一起玩 我的世界(Minecraft)，或者和他的妻子一起参加跑酷(或攀岩或其他冒险)。 Taavi的爱好很广泛。

简介

DBDB (狗床数据库) 是一个用Python实现的简单的键值对存储数据库(key/value database)。它把键与值相关联，并将该关联存储在磁盘上。

DBDB 的特点是在电脑崩溃或程序出错的时候也能保证数据的安全。它也避免了把所有数据同时保存在内存中，所以你可以储存比内存容量更多的数据。

往事

记得有一次我在找一个BASIC程序里的bug的时候，电脑屏幕上突然出现了几个闪烁的光点，然后程序就提前终止了。当我再次检查代码的时候，代码的最后几行消失了。

我问了我麻麻的一个懂编程的朋友，然后我意识到程序崩溃的原因是程序太大了，占用到了电脑的显存。屏幕上闪烁的光点是苹果BASIC电脑的一个特性，这表示内存不够用了。

从那一刻开始，我变得非常注意内存的使用。我学习了指针和malloc，我学习了数据是怎么存放进内存的。我会非常非常小心的使用内存。

几年后，在学习一个面向过程的编程语言 Erlang 的时候，我懂得了进程间的通信不需要把数据再拷贝一份以供另一个进程使用，因为所有数据都是不可变的(immutable)。然后我就喜欢上了 Clojure 的那些不可变的数据结构。

当我在2013年看到CouchDB的时候，我认识到了管理复杂数据的结构和机制。

我认识到了可以设计一个以不可变的(immutable)数据为基础的系统。

我写了一些相关的文章，

因为我觉着描述 CouchDB 的核心数据储存原理(根据我的理解)会很有有意思。

当我写到一个二叉树自平衡的算法的时候，我才发现这个算法描述起来好复杂啊，边缘情况(edge case) 太多了，比如说解释二叉树的一处改变的时候，树的其他部分为什么也会改变。然后我就不知道怎么写了。

吸取了那次的经验教训，我仔细观察了使用递归算法来更新不可变二叉树，感觉这个算法更简单明了，和蔼可亲。

所以我又双叒叕一次感受到了不可变的结构的好处。

然后，就有了DBDB。

为啥这个项目很有趣？

很多项目都会使用到数据库。但是你没有必要去自己写一个，就算只是存储json文件在磁盘上，也会有各种各样的边缘情况(edge case)需要去考虑，比如说：

- 存储空间不足时会发生什么？
- 电脑没电了，数据怎么保存？
- 如果数据大小超过可用内存怎么办？ (台式机上的大多数应用程序基本不会出现这种情况，但移动设备或服务器端Web应用程序可能会发生)

如果自己写一个数据库，你就知道数据库是怎么处理这些情况的了。

我们在这里讨论的技术和概念应适用于应对各种情况(包括发生故障时)。

关于不足。。。

DBDB 的不足之处

数据库为保证事务(transaction)是正确可靠的，必须具备的四个特性(ACID属性)：原子性(atomicity)，一致性(consistency)，独立性(isolation)和持久性(durability)。

DBDB 的数据更新方法具有原子性和持久性。这两个属性在后面的章节会做详细介绍。因为我们没有对储存的数据做限制，所以DBDB 不能保持数据的一致性。独立性在DBDB里也没有实现。

应用程序的代码当然可以保证自己的一致性，但是实现独立性需要一个事务管理器(transaction manager)。 我们不会在这里试图增加事务管理器的部分，但是你可以通过 CircleDB chapter 来进一步了解事务管理器。

还有一些系统维护的问题需要考虑。 在DBDB中，陈旧数据(Stale data)不会被收回处理，因此更新数据(甚至是使用相同的键)将会导致最终消耗掉所有的磁盘空间(你会很快发现为什么会这样)。PostgreSQL 把处理陈旧数据的过程称之为“吸尘(vacuuming)”，这使得旧的行空间可以重新使用，CouchDB 把处理陈旧数据的过程称之为“压缩”，通过把正常数据(live data)的部分重写入新文件，并替代旧的文件。

DBDB 可以添加“压缩旧数据”的功能，这就留给读者们作为小练习了【尾注1】。

DBDB 的架构

DBDB 把“将数据放在磁盘某处”(数据是怎么分布在文件中的;物理层)与数据的逻辑结构(本例中为二叉树;逻辑层) 从键/值存储的内容中分离出来(比如：键`a`与值`foo`的关联;公共API)。

许多数据库为了提高效能，通常把逻辑层和物理层分开实现。 比如说，DB2的SMS(文件系统中的文件)和DMS(原始块设备)或MySQL的 替代引擎.

DBDB 的设计

本文使用了大量篇幅介绍一个程序是怎么从无到有的写出来的。但是，这并不是大多数人参与、开发代码的方式。我们通常先是阅读别人写的代码，然后通过修改或者拓展这些代码来达到自己的需求。

所以，我们假设DBDB 是一个完整的项目，然后去了解它的流程和逻辑。让我们先从DBDB的包含的文件开始了解吧。

DBDB 包含的文件

下列文件的排列顺序是从前到后，比如说，第一个文件就是距离终端用户“最近的”。

- `tool.py` 是一个在终端中执行的命令行工具。
- `interface.py` 定义了一个 `DBDB` 类，它使用二叉树(`BinaryTree`) 来实现了Python中的字典(`dict`)。
- `logical.py` 定义了逻辑层。是使用键/值存储的接口(interface)。
    - `LogicalBase` 提供了使用get, set, commit 的接口，用了一个子类来完成具体的实现。它还用于管理存储的锁定，和内部节点的解引用(dereferencing)。
    - `ValueRef` 是一个Python的对象，是存在数据库中的二进制大型对象 BLOB(basic large object). 它间接使我们能够避免将整个数据存储一次性加载到内存中。
- `binary_tree.py` 定义了逻辑接口下的二叉树算法。
    - `BinaryTree` 提供二叉树的具体实现，包括get, insert, 和delete。BinaryTree 是一个 不变的 (immutable) 的树，所以数据的更新会产生一个新的树。
    - `BinaryNode` 实现了二叉树的节点的类。
    - `BinaryNodeRef` 是一个特殊的`ValueRef`实现，用来实现 `BinaryNode` 的序列化(serialise)和反序列化(deserialise)。
- `physical.py` 定义了物理层，`Storage` 类提供了持久的，(大部分是)只可添加的(append-only)记录存储。

每个文件都只包含一个类，换句话说，每个类只能由一个改变的原因(“each class should have only one reason to change”)。

读取数据

一个简单的例子：从数据库里读取一个数据。一起来看看怎么从 `example.db` 数据库里获取键为`foo`的值吧。

```
$ python -m dbdb.tool example.db get foo
```

这行代码运行了 `dbdb.tool` 中的 `main()` 函数。

```
# dbdb/tool.py
def main(argv):
    if not (4 <= len(argv) <= 5):
        usage()
        return BAD_ARGS
    dbname, verb, key, value = (argv[1:] + [None])[:4]
    if verb not in {'get', 'set', 'delete'}:
        usage()
        return BAD_VERB
    db = dbdb.connect(dbname)          # CONNECT
    try:
        if verb == 'get':
```

```
            sys.stdout.write(db[key])  # GET VALUE
        elif verb == 'set':
            db[key] = value
            db.commit()
        else:
            del db[key]
            db.commit()
    except KeyError:
        print("Key not found", file=sys.stderr)
        return BAD_KEY
    return OK
```

函数 `connect()` 会打开一个数据库文件(或者是创建一个新的，但是永远不会覆盖其它的文件)，然后返回一个名为`DBDB`的实例。

```
# dbdb/__init__.py
def connect(dbname):
    try:
        f = open(dbname, 'r+b')
    except IOError:
        fd = os.open(dbname, os.O_RDWR | os.O_CREAT)
        f = os.fdopen(fd, 'r+b')
    return DBDB(f)
```

```
# dbdb/interface.py
class DBDB(object):

    def __init__(self, f):
        self._storage = Storage(f)
        self._tree = BinaryTree(self._storage)
```

从上面的代码中，我们可以看到`DBDB`的实例中包含了一个对`Storage`实例的引用，它还把这个引用分享给了`self._tree`。为什么要这样呢？`self._tree`不可以有自己单独的对存储的访问吗？

关于哪个对象应该“拥有”一个资源的问题，在设计中通常是一个重要的问题，因为它影响到了程序的安全性。我们稍后会解释这个问题。

当我们获得`DBDB`的实例后，获取一个键的值就会通过`dict`的查找功能完成(Python的解释器会调用`DBDB.__getitem__()`)。

```
# dbdb/interface.py
class DBDB(object):
# ...
    def __getitem__(self, key):
        self._assert_not_closed()
        return self._tree.get(key)
```

```
    def _assert_not_closed(self):
        if self._storage.closed:
            raise ValueError('Database closed.')
```

`__getitem__()` 通过调用 `_assert_not_closed` 确保数据库仍处于打开状态。啊哈！这里我们看到了一个DBDB需要直接访问 `Storage`实例的原因：因此它可以强制执行前提条件。(你同意这个设计吗？你能想出一个不同的方式吗？)

然后DBDB通过调用`_tree.get()`函数(由`LogicalBase`提供)来查找`key`所对应的值：

```
# dbdb/logical.py
class LogicalBase(object):
# ...
    def get(self, key):
        if not self._storage.locked:
            self._refresh_tree_ref()
        return self._get(self._follow(self._tree_ref), key)
```

`get()` 先检查了储存是否被锁。目前，我们并不明白为什么在这里可能会有一个锁，但是我们可以猜到它是用来管理数据写入权限的。如果存储没有锁定会发生什么呢？

```
# dbdb/logical.py
class LogicalBase(object):
# ...
def _refresh_tree_ref(self):
        self._tree_ref = self.node_ref_class(
            address=self._storage.get_root_address())
```

`_refresh_tree_ref` 将磁盘上数据树的“视图”更新了，这使我们能够读取最新的数据。

如果我们读取数据的时候，数据被锁了呢？这说明其他的进程或许正在更新这部分数据；我们读取的数据不太可能是最新的。 这通常被称为“脏读”(dirty read)。这种模式允许许多读者访问数据，而不用担心阻塞，相对的缺点就是数据可能不是最新的。

现在，一起来看看是怎么具体拿取数据的。

```
# dbdb/binary_tree.py
class BinaryTree(LogicalBase):
# ...
    def _get(self, node, key):
        while node is not None:
            if key < node.key:
                node = self._follow(node.left_ref)
            elif node.key < key:
                node = self._follow(node.right_ref)
            else:
```

```
                return self._follow(node.value_ref)
        raise KeyError
```

这里用到了用到了二叉搜索。上文中介绍过了`Node` 和 `NodeRef` 是`BinaryTree`中的对象。他们是不可变的，所以他们的值不会改变。`Node`类包括键值和左右子项，这些不会改变。当更换根节点时，整个`BinaryTree`的内容才会明显变化。 这意味着在执行搜索时，我们不需要担心我们的树的内容被改变。

当找到了相应的值后，`main()`函数会把这个值写入到`stdout`，输出的值中，并且不会包含换行符。

插入和更新

现在，我们在`example.db`数据库中，把`foo`键的值设为`bar`：

```
$ python -m dbdb.tool example.db set foo bar
```

这段代码会运行`dbdb.tool`文件中的`main()`函数。这里，我们只介绍几个重要的地方。

```
# dbdb/tool.py
def main(argv):
    ...
    db = dbdb.connect(dbname)          # CONNECT
    try:
        ...
        elif verb == 'set':
            db[key] = value            # SET VALUE
            db.commit()                # COMMIT
        ...
    except KeyError:
        ...
```

给键赋值，我们使用了`db[key] = value`的方法来调用`DBDB.__setitem__()`

```
# dbdb/interface.py
class DBDB(object):
# ...
    def __setitem__(self, key, value):
        self._assert_not_closed()
        return self._tree.set(key, value)
```

`__setitem__` 确保了数据库的链接是打开的，然后调用`_tree.set()`来把键`key` 和值`value`存入`_tree`

`_tree.set()` 由 `LogicalBase` 提供:

```
# dbdb/logical.py
class LogicalBase(object):
```

```
# ...
    def set(self, key, value):
        if self._storage.lock():
            self._refresh_tree_ref()
        self._tree_ref = self._insert(
            self._follow(self._tree_ref), key, self.value_ref_class(value))
```

`set()` 先检查了数据有没有被锁定。

```
# dbdb/storage.py
class Storage(object):
    ...
    def lock(self):
        if not self.locked:
            portalocker.lock(self._f, portalocker.LOCK_EX)
            self.locked = True
            return True
        else:
            return False
```

这里有两个重要的点需要注意：

- 我们使用了的第三方库提供的锁，名叫portalocker。
- 如果数据库被锁定了，`lock()`函数会返回`False`。否则，会返回`True`

回到`_tree.set()`，现在我们明白了为什么需要先检查数据的锁(`lock()`)了：它会调用`_refresh_tree_ref`函数来获取最新的根节点。然后它会用一个新的树(已经插入 / 更新过数据)来替代原有的树。

插入和更新一个树，不会改变任何一个节点。因为`_insert()`会返回一个新的树。新树与老树会共享数据不变的部分以节省内存和执行时间。我们使用了递归来实现：

```
# dbdb/binary_tree.py
class BinaryTree(LogicalBase):
# ...
    def _insert(self, node, key, value_ref):
        if node is None:
            new_node = BinaryNode(
                self.node_ref_class(), key, value_ref,
self.node_ref_class(), 1)
        elif key < node.key:
            new_node = BinaryNode.from_node(
                node,
                left_ref=self._insert(
                    self._follow(node.left_ref), key, value_ref))
        elif node.key < key:
            new_node = BinaryNode.from_node(
                node,
                right_ref=self._insert(
```

```
                    self._follow(node.right_ref), key, value_ref))
        else:
            new_node = BinaryNode.from_node(node, value_ref=value_ref)
        return self.node_ref_class(referent=new_node)
```

请注意我们总是返回一个新的节点(含在一个`NodeRef`中)。我们建一个新的节点，它会与旧的节点共享未改变的部分。而不是更新旧的节点上的数据。这使我们的二叉树变得 不可变(immutable)。

你可能意识到有有个奇怪的地方：我们还没对磁盘上的数据做任何处理呢。我们目前所做的只是通过移动树的节点来操纵我们对磁盘数据的视图。

为了真正的把新的数据保存在硬盘上，我们需要调用`commit()`函数。我们在前面的讲`set`操作的章节见过这个函数。

`commit`会把所有的脏状态(dirty state)写入内存中的，然后保存下磁盘地址作为树的新根节点。

从commit的接口开始看：

```
# dbdb/interface.py
class DBDB(object):
# ...
    def commit(self):
        self._assert_not_closed()
        self._tree.commit()
```

`_tree.commit()`是在`LogicalBase`里面实现的：

```
# dbdb/logical.py
class LogicalBase(object)
# ...
    def commit(self):
        self._tree_ref.store(self._storage)
        self._storage.commit_root_address(self._tree_ref.address)
```

`NodeRef`的序列化(serialise)是通过让它们的子节点使用`prepare_to_store()`完成序列化 而完成的。

```
# dbdb/logical.py
class ValueRef(object):
# ...
    def store(self, storage):
        if self._referent is not None and not self._address:
            self.prepare_to_store(storage)
            self._address =
storage.write(self.referent_to_string(self._referent))
```

这里的LogicalBase里面的self._tree_ref其实使用了BinaryNodeRef(ValueRef的子类)。所以prepare_to_store()的具体实现方式为：

```
# dbdb/binary_tree.py
class BinaryNodeRef(ValueRef):
    def prepare_to_store(self, storage):
        if self._referent:
            self._referent.store_refs(storage)
```

[](The BinaryNode in question, _referent, asks its refs to store themselves:)

```
# dbdb/binary_tree.py
class BinaryNode(object):
# ...
    def store_refs(self, storage):
        self.value_ref.store(storage)
        self.left_ref.store(storage)
        self.right_ref.store(storage)
```

这个递归会在任何NodeRef有未写入的数据更新(比如说缺少_address)的时候一直循环下去。

现在让我们来回忆一下ValueRef里的store方法。store()里的最后一步是序列化这个节点，然后保存它的存储地址：

```
# dbdb/logical.py
class ValueRef(object):
# ...
    def store(self, storage):
        if self._referent is not None and not self._address:
            self.prepare_to_store(storage)
            self._address =
storage.write(self.referent_to_string(self._referent))
```

这里，NodeRef的 _referent保证会有引用的地址，所以我们通过创建这个节点的字节串(bytestring)来序列化它：

```
# dbdb/binary_tree.py
class BinaryNodeRef(ValueRef):
# ...
    @staticmethod
    def referent_to_string(referent):
        return pickle.dumps({
            'left': referent.left_ref.address,
            'key': referent.key,
            'value': referent.value_ref.address,
            'right': referent.right_ref.address,
```

```
            'length': referent.length,
        })
```

在`store()`中更新地址在实际上是改变`ValueRef`。 因为它对用户可见的值没有任何影响，所以我们可以认为它是不可变的。

根节点`_tree_ref`在`store()`之后(在`LogicalBase.commit()`中)，所有的数据就已经保存在磁盘上了。现在我们可以调用根地址的提交(commit)了：

```
# dbdb/physical.py
class Storage(object):
# ...
    def commit_root_address(self, root_address):
        self.lock()
        self._f.flush()
        self._seek_superblock()
        self._write_integer(root_address)
        self._f.flush()
        self.unlock()
```

我们确保句柄(file handle)已被刷新(所以系统就知道了我们想要所有数据都被保存起来，比如：SSD)以及返回了根节点的地址。我们知道最后一次写入是具有原子性(atomic)的，因为我们将磁盘地址存储在扇区边界上(sector boundary)。这是文件中的最靠前的，所以无论扇区大小如何，这都是正确的，单扇区磁盘写入能由磁盘硬件保证原子性。

因为根节点地址要么是旧值要么是新值(没有中间值)，所以其他进程可以从数据库中读取而不用锁定。 外部进程可能会获取到旧的或新的数据。所以，提交(commit)是原子性的。

因为我们在赋予根节点地址之前，会把新的数据写入磁盘并调用`fsync` syscall [尾注2]，所以未提交的数据是无法访问的。 相反，一旦根节点地址被更新，我们知道它引用的所有数据也在磁盘上。以这种方式，提交(commit)也具有持久性(durability)。

就是这样！

NodeRefs 是怎么节省空间的

为了避免把这个树的数据同时保存在内存中，当从磁盘读取逻辑节点时，其左和右子节点的磁盘地址(还有值)将被加载到内存中。所以访问子节点需要调用一个额外的函数`NodeRef.get()`来获取真正的数据。

`NodeRef` 只需包含一个地址：

```
+---------+
| NodeRef |
| ------- |
| addr=3  |
| get()   |
+---------+
```

当点用 `get()`后，`NodeRef` 会返回具体的节点，并包括其两个子节点的`NodeRef`类。

```
+---------+     +---------+     +---------+
| NodeRef |     | Node    |     | NodeRef |
| ------- |     | ------- | +-> | ------- |
| addr=3  |     | key=A   | |   | addr=1  |
| get() ------> | value=B | |   +---------+
+---------+     | left  ----+
                | right ----+   +---------+
                +---------+ |   | NodeRef |
                            +-> | ------- |
                                | addr=2  |
                                +---------+
```

当树的更改未提交时，它们保存在内存中，包括从根(root)向下到更改的叶(leaves)。 当更改还没保存到磁盘时，所以被更改的节点包含具体的键和值，但是没有磁盘地址。 处理写入的进程可以看到这些未提交的更改，并且可以在发出提交之前再次对其进行更改，这是因为`NodeRef.get()`有值的话，会返回一个未提交的值; 在通过API访问时，提交和未提交的数据之间没有区别。其他读者可以看到最新的数据，因为新的更改在根节点的地址被写入磁盘前，是看不到的。并发的更新操作会被磁盘上的文件锁阻止。文件会在第一次更新时上锁，并在提交后解锁。

留给读者的小练习

DBDB 允许多进程同时访问同一个数据库。为做到这一点，我们付出的是，读取有时获得的是陈旧的数据。如果我们需要总是读取最新的数据该怎么办？ 一个常见的用例是读取值，然后根据该值进行更新。 你如何在“DBDB”上实现这个方法呢？你需要做什么权衡来做到这个功能呢？

更新数据存储的算法可以通过改变`interface.py`文件中的这个词`BinaryTree`来使用别的算法。 比如说可以用 B-trees, B+ trees 或其他的结构来提高数据库的效能。一个平衡的二叉树需要做O(log2(n))次节点的读取，来查找值。而B+树只需要更少的次数，比如O(log32(n))次，因为每个节点有32个子节点(而不是2个)。这会很大的提高练习的难度。比如40亿条数据中查找一条信息，这需要大约 log2(2 ^ {32})= 32 至 log32(2 ^ {32})=6.4 次查找。每个查找都是随机访问，因为开销非常大，所以这是难以做到的。SSD或许可以用延迟解决，但I/O的节省仍然存在。

默认情况下，值以字节的形式(为了能直接传入到`Storage`)存储在`ValueRef`里。二叉树的节点是`ValueRef`的子类。通过json 或者 msgpack 格式保存更丰富的数据则需要编写自己的文件并将其设置为“value_ref_class”。`BinaryNodeRef` 就是一个使用 pickle 来序列化数据的例子。

压缩数据库是另一个有趣的练习。 压缩可以随着树的移动通过中间遍历(infix-of-median traversal)完成。如果树节点全部在一起可能是最好的，因为它们是遍历以查找任何数据的。将尽可能多的中间节点打包进到磁盘扇区中可以提高读取性能，至少是在压缩之后，可以提高读取效率。 这里有一些细节需要注意(例如，内存使用)，如果你打算完成这个练习。 请记住：在修改前后，总是注意记录性能的改变！你经常会对结果感到惊讶。

模式与原则

有些没有实现的测试接口。作为开发DBDB的一部分，我写了一些描述我想要使用DBDB的测试。第一个测试针对数据库的内存版本进行，然后我将DBDB的存储方式扩展到了磁盘，甚至后来添加了NodeRefs的概念。而且

大多数测试并不需要改变，这让我对开发DBDB更加有信心。

遵守单一责任原则(Single Responsibility Principle)。类最多只能有一个改变的原因。这不是严格按照DBDB的情况，但也多种拓展途径与只需要局部的变化。Refactoring as I added features was a pleasure!

总结

DBDB 是一个实现了简单功能的数据库，但是它也可以变的很复杂。我为了管理它的复杂性，做的最重要的事情就是用不可变数据结构实现一个表面上可变的对象(implement an ostensibly mutable object with an immutable data structure.)。 我鼓励你以后遇到某些棘手问题(非常多的边际情况)的时候，考虑使用这种方法。

---

1. 额外的功能：您能保证压实的树结构是平衡的吗？ 这有助于保持性能。
2. 在文件描述符(file descriptor)上调用`fsync`请求会让操作系统和硬盘驱动器(或SSD)立即写入所有缓冲的数据。操作系统和驱动器通常为保证性能，不会立即写入所有内容。