GitDB Documentation

Release 0.5.1

Sebastian Thiel

November 19, 2014

Contents

Contents:

CHAPTER 1

Overview

The *GitDB* project implements interfaces to allow read and write access to git repositories. In its core lies the *db* package, which contains all database types necessary to read a complete git repository. These are the `LooseObjectDB`, the `PackedDB` and the `ReferenceDB` which are combined into the `GitDB` to combine every aspect of the git database.

For this to work, GitDB implements pack reading, as well as loose object reading and writing. Data is always encapsulated in streams, which allows huge files to be handled as well as small ones, usually only chunks of the stream are kept in memory for processing, never the whole stream at once.

Interfaces are used to describe the API, making it easy to provide alternate implementations.

1.1 Installing GitDB

Its easiest to install gitdb using the *easy_install* program, which is part of the setuptools:

```
$ easy_install gitdb
```

As the command will install gitdb in your respective python distribution, you will most likely need root permissions to authorize the required changes.

If you have downloaded the source archive, the package can be installed by running the `setup.py` script:

```
$ python setup.py install
```

1.2 Getting Started

It is advised to have a look at the *Usage Guide* for a brief introduction on the different database implementations.

1.3 Source Repository

The latest source can be cloned using git from one of the following locations:

- git://gitorious.org/git-python/gitdb.git
- git://github.com/Byron/gitdb.git

1.3.1 License Information

GitDB is licensed under the New BSD License.

CHAPTER 2

Usage Guide

This text briefly introduces you to the basic design decisions and accompanying types.

2.1 Design

The *GitDB* project models a standard git object database and implements it in pure python. This means that data, being classified by one of four types, can can be stored in the database and will in future be referred to by the generated SHA1 key, which is a 20 byte string within python.

GitDB implements *RW* access to loose objects, as well as *RO* access to packed objects. Compound Databases allow to combine multiple object databases into one.

All data is read and written using streams, which effectively prevents more than a chunk of the data being kept in memory at once mostly [1].

2.2 Streams

In order to assure the object database can handle objects of any size, a stream interface is used for data retrieval as well as to fill data into the database.

2.2.1 Basic Stream Types

There are two fundamentally different types of streams, **IStream**s and **OStream**s. IStreams are mutable and are used to provide data streams to the database to create new objects.

OStreams are immutable and are used to read data from the database. The base of this type, **OInfo**, contains only type and size information of the queried object, but no stream, which is slightly faster to retrieve depending on the database.

OStreams are tuples, IStreams are lists. Both, OInfo and OStream, have the same member ordering which allows quick conversion from one type to another.

[1] When reading streams from packs, all deltas are currently applied and the result written into a memory map before the first byte is returned. Future versions of the delta-apply algorithm might improve on this.

2.3 Data Query and Data Addition

Databases support query and/or addition of objects using simple interfaces. They are called **ObjectDBR** for read-only access, and **ObjectDBW** for write access to create new objects.

Both have two sets of methods, one of which allows interacting with single objects, the other one allowing to handle a stream of objects simultaneously and asynchronously.

Acquiring information about an object from a database is easy if you have a SHA1 to refer to the object:

```
ldb = LooseObjectDB(fixture_path("../../.git/objects"))

for sha1 in ldb.sha_iter():
    oinfo = ldb.info(sha1)
    ostream = ldb.stream(sha1)
    assert oinfo[:3] == ostream[:3]

    assert len(ostream.read()) == ostream.size
# END for each sha in database
```

To store information, you prepare an *IStream* object with the required information. The provided stream will be read and converted into an object, and the respective 20 byte SHA1 identifier is stored in the IStream object:

```
data = "my data"
istream = IStream("blob", len(data), StringIO(data))

# the object does not yet have a sha
assert istream.binsha is None
ldb.store(istream)
# now the sha is set
assert len(istream.binsha) == 20
assert ldb.has_object(istream.binsha)
```

2.4 Asynchronous Operation

For each read or write method that allows a single-object to be handled, an *_async* version exists which reads items to be processed from a channel, and writes the operation's result into an output channel that is read by the caller or by other async methods, to support chaining.

Using asynchronous operations is easy, but chaining multiple operations together to form a complex one would require you to read the docs of the *async* package. At the current time, due to the *GIL*, the *GitDB* can only achieve true concurrency during zlib compression and decompression if big objects, if the respective c modules where compiled in *async*.

Asynchronous operations are scheduled by a *ThreadPool* which resides in the *gitdb.util* module:

```
from gitdb.util import pool

# set the pool to use two threads
pool.set_size(2)

# synchronize the mode of operation
pool.set_size(0)
```

Use async methods with readers, which supply items to be processed. The result is given through readers as well:

```
from async import IteratorReader

# Create a reader from an iterator
reader = IteratorReader(ldb.sha_iter())

# get reader for object streams
info_reader = ldb.stream_async(reader)

# read one
info = info_reader.read(1)[0]

# read all the rest until depletion
ostreams = info_reader.read()
```

2.5 Databases

A database implements different interfaces, one if which will always be the *ObjectDBR* interface to support reading of object information and streams.

The *Loose Object Database* as well as the *Packed Object Database* are *File Databases*, hence they operate on a directory which contains files they can read.

File databases implementing the *ObjectDBW* interface can also be forced to write their output into the specified stream, using the `set_ostream` method. This effectively allows you to redirect its output to anywhere you like.

Compound Databases are not implementing their own access type, but instead combine multiple database implementations into one. Examples for this database type are the *Reference Database*, which reads object locations from a file, and the *GitDB* which combines loose, packed and referenced objects into one database interface.

For more information about the individual database types, please see the *API Reference*, and the unittests for the respective types.

CHAPTER 3

API Reference

CHAPTER 4

Changelog

4.1 0.5.1

- Restored most basic python 2.4 compatibility, such that gitdb can be imported within python 2.4, pack access cannot work though. This at least allows Super-Projects to provide their own workarounds, or use everything but pack support.

4.2 0.5.0

Initial Release

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*