Apache Ignite binary client Python API Documentation

Release 0.1.0

Apache Software Foundation (ASF)

Jun 27, 2018

CONTENTS:

CHAPTER

ONE

BASIC INFORMATION

1.1 What it is?

This is an Apache Ignite lightweight (binary protocol) client library, written in Python, abbreviated as *pyignite*.

Apache Ignite is a memory-centric distributed database, caching, and processing platform for transactional, analytical, and streaming workloads delivering in-memory speeds at petabyte scale.

Ignite binary client protocol provides user applications the ability to communicate with an existing Ignite cluster without starting a full-fledged Ignite node. An application can connect to the cluster through a raw TCP socket.

1.2 Prerequisites

- *Python 3.4* or above (3.6 is tested),
- Access to *Apache Ignite 2.5* node, local or remote. Higher versions of Ignite may or may not work.

1.3 Installation

1.3.1 for end user

If you want to use *pyignite* in your project, you may install it from PyPI:

```
$ pip install pyignite
```

1.3.2 for developer

If you want to run tests, examples or build documentation, clone the whole repository:

```
$ git clone git@github.com:nobitlost/ignite.git
$ git checkout ignite-7782
$ cd ignite/modules/platforms/python
```

Then run through the contents of *requirements* folder to install the necessary prerequisites into your working Python environment using

```
$ pip install -r requirements/install.txt
```

You may also want to consult the setuptools manual about using *setup.py*.

1.4 Examples

Some examples of using pyignite are provided in *ignite/modules/platforms/python/examples* folder.

This code implies that it is run in the environment with *pyignite* package installed. If you want to play with the repository clone of *pyignite*, run

```
$ cd ignite/modules/platforms/python
$ pip install -e .
```

to install the package code in your environment without actually copying it to *site-packages* folder.

1.5 Testing

Create and activate virtualenv environment. Run

```
$ cd ignite/modules/platforms/python
$ python ./setup.py pytest
```

This does not require *pytest* and other test dependencies to be installed in your environment.

NB! Some or all tests require Apache Ignite node running on localhost:10800. To override the default parameters, use command line options *–ignite-host* and *–ignite-port*:

```
$ python ./setup.py pytest --addopts "--ignite-host=example.com --ignite-port=19840"
```

You can use each of these options multiple times. All combinations of given host and port will be tested.

1.6 Documentation

To recompile this documentation, do this from your virtualenv environment:

```
$ cd ignite/modules/platforms/python
$ pip install -r requirements/docs.txt
$ cd docs
$ make html
```

Then open *ignite/modules/platforms/python/docs/generated/html/index.html* in your browser.

If you feel that old version is stuck, do

```
$ cd ignite/modules/platforms/python/docs
$ make clean
$ sphinx-apidoc -M -o source/ ../pyignite
$ make html
```

And that should be it.

1.7 Licensing

This is a free software, brought to you on terms of the Apache License v2.

CHAPTER

TWO

MODULE STRUCTURE

The client library consists of several modules.

The most important for the end user are *connection* and *api*.

2.1 datatypes

Apache Ignite uses a sophisticated system of serializable data types to store and retrieve user data, as well as to manage the configuration of its caches through the Ignite binary protocol.

The complexity of data types varies from simple integer or character types to arrays, maps, collections and structures.

Each data type is defined by its code. *Type code* is byte-sized. Thus, every data object can be represented as a payload of fixed or variable size, logically divided into one or more fields, prepended by the *type_code* field.

Most of Ignite data types can be represented by some of the standard Python data type or class. Some of them, however, are conceptually alien, overly complex, or ambiguous to Python dynamic typing system.

The following table summarizes the notion of Apache Ignite data types, as well as their representation and handling in Python. For the nice description, as well as gory implementation details, you may follow the link to the parser/constructor class definition.

Note: you are not obliged to actually use those parser/constructor classes. Pythonic types will suffice to interact with Apache Ignite binary API. However, in some rare cases of type ambiguity, as well as for the needs of interoperability, you may have to sneak one or the other class, along with your data, in to some API function as a *type conversion hint*.

type_code	Apache Ignite docs reference	Python type or class	Parser/constructor class
Primitive data types			
0x01	Byte	int	`ByteObject`
0x02	Short	int	`ShortObject`
0x03	Int	int	`IntObject`
0x04	Long	int	`LongObject`
0x05	Float	float	`FloatObject`
0x06	Double	float	`DoubleObject`
0x07	Char	str	`CharObject`
0x08	Bool	bool	`BoolObject`
0x65	Null	NoneType	`Null`
Standard objects			
0x09	String	Str	`String`
0x0a	UUID	uuid.UUID	`UUIDObject`
0x21	Timestamp	tuple	`TimestampObject`
0x0b	Date	datetime.datetime	`DateObject`

Continued on next page

Table 1 – continued from previous page

type_code	Apache Ignite docs reference	Python type or class	Parser/constructor class
0x24	Time	datetime.timedelta	`TimeObject`
0x1e	Decimal	decimal.Decimal	`DecimalObject`
0x1c	Enum	tuple	`EnumObject`
0x67	Binary enum	tuple	`BinaryEnumObject`
Arrays of primitives			
0x0c	Byte array	iterable/list	`ByteArrayObject`
0x0d	Short array	iterable/list	`ShortArrayObject`
0x0e	Int array	iterable/list	`IntArrayObject`
0x0f	Long array	iterable/list	`LongArrayObject`
0x10	Float array	iterable/list	`FloatArrayObject`
0x11	Double array	iterable/list	`DoubleArrayObject`
0x12	Char array	iterable/list	`CharArrayObject`
0x13	Bool array	iterable/list	`BoolArrayObject`
Arrays of standard objects			
0x14	String array	iterable/list	`StringArrayObject`
0x15	UUID array	iterable/list	`UUIDArrayObject`
0x22	Timestamp array	iterable/list	`TimestampArrayObject`
0x16	Date array	iterable/list	`DateArrayObject`
0x23	Time array	iterable/list	`TimeArrayObject`
0x1f	Decimal array	iterable/list	`DecimalArrayObject`
Object collections, special types, and complex object			
0x17	Object array	iterable/list	`ObjectArrayObject`
0x18	Collection	tuple	`CollectionObject`
0x19	Map	dict, collections.OrderedDict	`MapObject`
0x1d	Enum array	iterable/list	`EnumArrayObject`
0x67	Complex object		Not yet implemented
0x1b	Wrapped data		Not yet implemented

All type codes are stored in module `pyignite.datatypes.type_codes`.

On top of all parser/constructor classes, there is an `AnyDataObject` class. It is an omnivorous data type that has no *type_code*; instead, it picks up the right class on serializing your python data or deserializing the byte stream.

It is not overly smart or omnipotent though: it can not choose CharObject for you; it will use String. It will also use LongArrayObject for represent two-integer tuple, even if you mean Enum or Collection.

This is the summary of its type guessing:

Native data types	Ignite data object
None	`Null`
int	`LongObject`
float	`DoubleObject`
str, bytes	`String`
datetime.datetime	`DateObject`
datetime.timedelta	`TimeObject`
decimal.Decimal	`DecimalObject`
uuid.UUID	`UUIDObject`
iterable	`datatypes` will inspect its contents to find the right *ArrayObject class

Bottom line: use type hints when you need to pick up a certain data type for your data, not just store that data.

2.2 connection

To connect to Ignite server socket, instantiate a `Connection` class with host name and port number. Connection will negotiate a handshake with the Ignite server and raise a `SocketError` in case of client/server API versions mismatch or data flow errors.

You can then pass a `Connection` instance to various API functions.

2.3 api

This is a collection of functions, split into three parts:

- `cache_config` allows you to manipulate caches;
- `key_value` brings a key-value-style data manipulation, similar to *memcached* or *Redis* APIs;
- `sql` gives you the ultimate power of SQL queries.

To construct client queries and process server responses, all API functions uses `Query` and `Response` base classes respectively under their hoods. These classes are a natural extension of the data type parsing/constructing module (`datatypes`) and uses all the power of the indigenous `AnyDataObject`.

Each function returns operation status and result data (or verbose error message) in `APIResult` object.

All data manipulations are handled with native Python data types, without the need for the end user to construct complex data objects or parse blobs.

CHAPTER

THREE

EXAMPLES OF USAGE

3.1 Open connection

```
from pyignite.connection import Connection
from pyignite.api import (
    cache_create, cache_destroy, cache_get, cache_put,
    cache_get_names, hashcode,
)

conn = Connection()
conn.connect('127.0.0.1', 10800)
```

3.2 Create cache

```
cache_name = 'my cache'
hash_code = hashcode(cache_name)

cache_create(conn, cache_name)
```

3.3 Put value in cache

```
result = cache_put(conn, hash_code, 'my key', 42)
print(result.message)  # “Success”
```

3.4 Get value from cache

```
result = cache_get(conn, hash_code, 'my key')
print(result.value)  # “42”

result = cache_get(conn, hash_code, 'non-existent key')
print(result.value)  # None
```

3.5 List keys in cache

```
result = cache_get_names(conn, hash_code)
print(result.value)  # ['my key']
```

3.6 Type hints usage

```
from pyignite.datatypes import CharObject, ShortObject


cache_put(conn, hash_code, 'my key', 42)
# value ‘42’ takes 9 bytes of memory as a LongObject

cache_put(conn, hash_code, 'my key', 42, value_hint=ShortObject)
# value ‘42’ takes only 3 bytes as a ShortObject

cache_put(conn, hash_code, 'a', 1)
# ‘a’ is a key of type String

cache_put(conn, hash_code, 'a', 2, key_hint=CharObject)
# another key ‘a’ of type CharObject was created

# now let us delete both keys at once
cache_remove_keys(conn, hash_code, [
    'a',                # a default type key
    ('a', CharObject),  # a key of type CharObject
])
```

3.7 Scan queries

Scan queries allows you to browse cache contents with pagination.

```
page_size = 10

result = scan(conn, hash_code, page_size)
print(dict(result.value))
# {
#     'cursor': 1,
#     'data': {
#         'key_4': 4,
#         'key_2': 2,
#         'key_8': 8,
#         ... 10 elements on page...
#         'key_0': 0,
#         'key_7': 7
#     },
#     'more': True
# }
```

Subsequent scans could be made using cursor ID.

```
cursor = result.value['cursor']
result = scan_cursor_get_page(conn, cursor)
print(result.value)
# {
#     'data': {
#         'key_15': 15,
#         'key_17': 17,
#         'key_11': 11,
#         ... another 10 elements...
#         'key_19': 19,
#         'key_16': 16
#     },
#     'more': False
# }
```

When cursor have no more data, it automatically destroys.

```
result = scan_cursor_get_page(conn, cursor)
print(result.message)
# Failed to find resource with id: 1
```

If your cursor still holds some data, but you have no use of it anymore, you may destroy it manually.

```
resource_close(conn, cursor)
```

3.8 Inspect cache configuration

```

result = cache_get_configuration(conn, hash_code)
print(result.value)
# OrderedDict([
#     ('length', 122),
#     ('backups_number', 1),
#     ('cache_mode', 0),
#     ('copy_on_read', True),
#     ('data_region_name', None),
#     ('eager_ttl', True),
#     ('statistics_enabled', False),
#     ('group_name', None),
#     ('invalidate', 0),
#     ('default_lock_timeout', 2147483648000),
#     ('max_query_iterators', 1024),
#     ('name', 'my cache'),
#     ('is_onheap_cache_enabled', False),
#     ('partition_loss_policy', 4),
#     ('query_detail_metric_size', 0),
#     ('query_parallelism', 1),
#     ('read_from_backup', True),
#     ('rebalance_batch_size', 524288),
#     ('rebalance_batches_prefetch_count', 2),
#     ('rebalance_delay', 0),
#     ('rebalance_mode', 1),
#     ('rebalance_order', 0),
#     ('rebalance_throttle', 0),
```

(continues on next page)

(continued from previous page)

```
#     ('rebalance_timeout', 10000),
#     ('sql_escape_all', False),
#     ('sql_index_inline_max_size', -1),
#     ('sql_schema', None),
#     ('write_synchronization_mode', 2),
#     ('cache_key_configuration', []),
#     ('query_entity', [])
```

3.9 Create cache with a certain configuration

You must supply at least cache name.

```
cache_name = 'my_special_cache'

cache_create_with_config(conn, {
        PROP_NAME: cache_name,
        PROP_CACHE_KEY_CONFIGURATION: [
            {
                'name': '123_key',
                'type_name': 'blah',
                'is_key_field': False,
                'is_notnull_constraint_field': False,
            }
        ],
    })
```

3.10 Do cleanup

Destroy created cache and close connection.

```
cache_destroy(conn, hash_code)
conn.close()
```

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search