УДК 004.42

К.Ю. Островська, Є.В. Островський, І.В. Кліопа

# РЕАЛІЗАЦІЯ REMOTE DICTIONARY SERVER З ВИКОРИСТАННЯМ МОВИ PYTHON

*Анотація. В результаті роботи було програмно реалізовано Redis використовуючи мову Python. Завдяки високій швидкості і простоті Redis часто використовується для мобільних і інтернет-додатків, ігор, рекламних платформ та ін. В тих випадках, коли необхідна максимально можлива продуктивність.*

*Ключові слова: сховище, структура даних, Server, оперативна пам'ять, клас, команда, база даних, мова Python.*

Redis (англ. Remote Dictionary Server) - швидке сховище в пам'яті з відкритим вихідним кодом для структур даних «ключ-значення». Redis поставляється з набором різноманітних структур даних в пам'яті, що спрощує створення різних спеціальних додатків.

Зберігає базу даних в оперативній пам'яті, забезпечена механізмами знімків і журналювання для забезпечення постійного зберігання на диску. Також надає операції для реалізації механізму обміну повідомленнями в паттерне publish - subscribe. З його допомогою додатка можуть створювати канали, підписуватися на них і поміщати в канали повідомлення, які будуть отримані всіма передплатниками (як IRC-чат). Підтримує реплікацію даних з основних вузлів на кілька підлеглих. Також Redis підтримує транзакції і пакетну обробку команд.

Redis працює на більшості POSIX систем, таких як Linux, *.BSD, Mac OS X без будь-яких доповнень. Linux і Mac OS X - дві операційні системи, в яких був розроблений і в більшій мірі протестований Redis, тому VMware рекомендує використовувати саме їх для розгортання. Офіційною підтримки для збірок Windows немає, але доступні деякі опції, що дозволяють забезпечити роботу Redis на цій ОС. Компанія Microsoft активно працює над перенесенням Redis на Windows.



 ISSN 1562-9945

Безліч мов програмування мають бібліотеки для роботи з Redis: C, C ++, C #, Clojure, Lisp, Erlang, Java, JavaScript, Haskell, Lua, Perl, PHP, Python, Ruby, Scala, Go, Tcl, Rust.

Всі дані Redis зберігає у вигляді словника, в якому ключі пов'язані зі своїми значеннями. Одне з ключових відмінностей Redis від інших сховищ даних полягає в тому, що значення цих ключів не обмежуються рядками. Підтримуються наступні абстрактні типи даних:

- Рядки (strings). Базовий тип даних Redis. Рядки в Redis бінарнобезпечні, можуть використовуватися так само як числа.

- Списки (lists). Класичні списки рядків, впорядковані в порядку вставки, яка можлива як з боку голови, так і з боку хвоста списку.

- Безлічі (sets). Безлічі рядків в математичному розумінні: не впорядковані, підтримують операції вставки, перевірки входження елемента, перетину і різниці множин.

- Хеш-таблиці (hashes). Класичні хеш-таблиці або асоціативні масиви.

Тип даних значення визначає, які операції (команди) доступні для нього. Redis підтримує такі високорівневі операції, як об'єднання і різницю наборів, а також їх сортування.

Redis корисний як сховище в якому можна зберігати дані з певним терміном дії. Час дії може бути зазначено в секундах або в форматі Unix timestamp (кількість секунд з 1.1.1970).

Redis підтримує атомарний інкремент строкових даних. При роботі інкремента доступ до даних блокується, таким чином здійснюється цілісність даних.

Redis також підтримує виконання транзакцій, які повинні дотримуватися двох принципів:

1. Команди повинні виконуватися по порядку. Поки не буде перервано іншими запитами протягом всього процесу.

2. Повинна бути забезпечена цілісність транзакції. Транзакції починаються з команди MULTI, а запускаються командою EXEC. Якщо з яких-небудь причин транзакція переривається, Redis заблокує її виконання до тих пір, поки не буде виконана команда redis-check-aof і скасовані всі зміни.

Redis працює з п'ятьма типами даних: Strings (рядки), Sets (безлічі), Sorted Sets (сортовані безлічі), Lists (списки), Hashes (хеш).

Відновлення даних проводиться двома різними способами. Перший - це механізм знімків, в якому дані асинхронно переносяться з оперативної пам'яті в файл формату *.RDB (розширення дампов Redis). Другий спосіб - файл, доступний тільки для дозапису, в якому зберігається лог всіх операцій, що змінювали дані в пам'яті.

Redis підтримує реплікацію типу master-slave. Дані з будь-якого сервера Redis можуть реплікуватись довільну кількість разів. Реплікація корисна для масштабування читання (але не записи) або при дуже великих обсягах даних. Всі дані, які потрапляють на один вузол Redis (який називається master) будуть потрапляти також на інші вузли (називаються slave). Для конфігурації slave-вузлів можна змінити опцію slaveof або аналогічну по написанню команду (вузли, запущені без подібних опцій є master-вузлами).

Реплікація допомагає захистити дані, копіюючи їх на інші сервера. Реплікація також може бути використана для збільшення продуктивності, так як запити на читання можуть обслуговуватися slave-вузлами. Ці вузли можуть відповісти злегка застарілими даними, але для більшості додатків це прийнятно.

Мета проекту полягала в тому, щоб написати простий сервер, який міг би працювати з чергою завдань. Сервер використовує Redis як механізм зберігання за замовчуванням для запису завдань в черзі, результатів виконання та інших речей.

Сервер, який ми будемо створювати, зможе відповідати на такі команди:

- *GET <key>*
- *SET <key> <value>*
- *DELETE <key>*
- *FLUSH*
- *MGET <key1> ... <keyn>*
- *MSET <key1> <value1> ... <keyn> <valuen>*

Також будемо підтримувати такі типи даних: рядки і виконавчі дані, числа, null, масиви, словники, повідомлення про помилки.

Щоб обробляти кілька клієнтів асинхронно, будемо використовувати gevent, але можливо використовувати модуль SocketServer

стандартної бібліотеки за допомогою ForkingMixin або ThreadingMixin.

Нам знадобиться сам сервер і функція зворотного виклику, яка буде виконуватися при підключенні нового клієнта. Крім того, знадобиться якась логіка для обробки запиту від клієнта і відправки йому відповіді.

```
class ProtocolHandler (object):
    def handle_request (self, socket_file):
        # Parse a request from the client into it's component parts.
        pass
    def write_response (self, socket_file, data):
        # Serialize the response data and send it to the client.
        pass
class Server (object):
    def __init __ (self, host = '127.0.0.1', port = 31337, max_clients = 64):
        self._pool = Pool (max_clients)
        self._server = StreamServer (
            (Host, port),
            self.connection_handler,
            spawn = self._pool)
        self._protocol = ProtocolHandler ()
        self._kv = {}…
```

Поділивши завдання так, щоб обробка протоколу знаходиться у власному класі з двома загальнодоступними методами: handle_request і write_response, а сервер використовує обробник протоколу для розпакування клієнтських запитів і повторення відповідей сервера на клієнтський сервер. Метод get_response () буде використовуватися для виконання команди, ініційованої клієнтом.

Більш детально розглядаючи код методу connection_handler (), бачимо, що ми отримуємо оболонку навколо об'єкта сокета, подібну файлу. Ця оболонка дозволяє абстрагуватися від деяких особливостей, з якими зазвичай стикаються при роботі з чистими сокетами. Функція входить в нескінченний цикл, зчитує запити від клієнта, відправляє відповіді і, нарешті, виходить з циклу, коли клієнт відключається (відзначається методом read (), який повертає порожній рядок).

Використовуємо типізовані виключення для обробки випадків, коли клієнти відключаються, і для повідомлення користувачів про командах обробки помилок. Наприклад, якщо користувач відправляє

невірно складений запит на сервер, викидаємо CommandError, який серіалізуются у відповідь на помилку і відправляється клієнту.

Заповнимо клас обробника протоколу, щоб він реалізовував протокол Redis.

```
class ProtocolHandler (object):
    def __init __ (self):
        self.handlers = {
            '+': Self.handle_simple_string,
            '-': self.handle_error,
            ':': Self.handle_integer,
            '$': Self.handle_string,
            '*': Self.handle_array,
            '%': Self.handle_dict}
    def handle_request (self, socket_file):
        first_byte = socket_file.read (1)
        if not first_byte:
            raise Disconnect ()…
```

У частині сериализації протоколу зробимо протилежне: перемістимо об'єкти Python в їх серіалізовані копії.

```
class ProtocolHandler (object):
    # ... above methods omitted ...
    def write_response (self, socket_file, data):
        buf = BytesIO ()
        self._write (buf, data)
        buf.seek (0)
        socket_file.write (buf.getvalue ())
        socket_file.flush ()
    def _write (self, buf, data):
        if isinstance (data, str):
            data = data.encode ('utf-8')…
```

Додатковою перевагою обробки протоколу в своєму класі є те, що можливо повторно використовувати методи handle_request і write_response для створення клієнтської бібліотеки.

Клас Server, повинен мати метод get_response (). Команди будуть вважатися поданими клієнтом як прості рядки або масив аргументів команди, тому параметр даних, переданий в get_response (), буде або рядком байтів, або списком. Щоб спростити обробку, якщо дані виявляться простий рядком, перетворимо її в список шляхом поділу на прогалини.

Першим аргументом буде ім'я команди з будь-якими додатковими аргументами, які належать зазначеній команді. Як і при зіставленні першого байта з обработчиками в ProtocolHandler, давайте створимо зіставлення команд з функціями зворотного виклику в класі Server:

```
class Server (object):
  def __init __ (self, host = '127.0.0.1', port = 31337, max_clients = 64):
    self._pool = Pool (max_clients)
    self._server = StreamServer (
      (Host, port),
      self.connection_handler,
      spawn = self._pool)
    self._protocol = ProtocolHandler ()
    self._kv = {}
    self._commands = self.get_commands ()
  def get_commands (self):
    return {
      'GET': self.get,
      'SET': self.set,
      'DELETE': self.delete,
      'FLUSH': self.flush,
      'MGET': self.mget,
      'MSET': self.mset}…
```

Щоб взаємодіяти з сервером, необхідно повторно використовувати клас ProtocolHandler для реалізації простого клієнта. Клієнт буде підключатися до сервера і відправляти команди, закодовані у вигляді списків. Ми будемо повторно використовувати як write_response (), так і логіку handle_request () для запитів на кодування і обробку відповідей сервера відповідно.

```
class Client (object):
  def __init __ (self, host = '127.0.0.1', port = 31337):
    self._protocol = ProtocolHandler ()
    self._socket = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
    self._socket.connect ((host, port))
    self._fh = self._socket.makefile ( 'rwb')
  def execute (self, * args):
    self._protocol.write_response (self._fh, args)
    resp = self._protocol.handle_request (self._fh)…
```

За допомогою методу execute () є можливість передати довільний список параметрів, які будуть закодовані як масив і

відправлені на сервер. Відповідь сервера аналізується і повертається як Python-об'єкт. Для зручності необхідно написати клієнтські методи для окремих команд:

```
class Client (object):
  # ...
  def get (self, key):
    return self.execute ( 'GET', key)
  def set (self, key, value):
    return self.execute ( 'SET', key, value)
  def delete (self, key):
    return self.execute ( 'DELETE', key)
  def flush (self):
    return self.execute ( 'FLUSH')
  def mget (self, * keys):
    return self.execute ( 'MGET', * keys)…
```

Щоб перевірити клієнт, необхідно сконфігурувати Python-скрипт для запуску сервера безпосередньо з командного рядка:

```
# Add this to bottom of module:
if __name__ == '__main__':
  from gevent import monkey; monkey.patch_all ()
  Server (). Run ()
```

Щоб перевірити сервер, просто запустіть серверний Python-модуль з командного рядка. В іншому терміналі відкрийте інтерпретатор Python і імпортуйте клас Client з модуля сервера. При створенні екземпляра клієнта буде відкритим з'єднання, і будемо мати можливість запускати команди!

Завдяки високій швидкості і простоті Redis часто використовується для мобільних і інтернет-додатків, ігор, рекламних платформ ін. В тих випадках, коли необхідна максимально можлива продуктивність.

## ЛІТЕРАТУРА

1. Redis [Электронный ресурс]: Официальный сайт/ Режим доступа: http://redis.io/

2. An interview with Salvatore Sanfilippo, creator of Redis, working out of Sicily, January 4, 2011, by Stefano Bernardi, EU-Startups

3. Хабрахабр [Электронный ресурс]: Redis 2.0 на Хабрахабре / Режим доступа: http://habrahabr.ru/post/105022/

4. ruseller.com [Электронный ресурс]:работа с Redis и PHP / Режим доступа: http://ruseller.com/lessons.php?rub=37&id=2289

5. beget.ru [Электронный ресурс]: Использование Redis / Режим доступа:https://beget.ru/articles/redis