PostgreSQL & Python: Asynchronous Queries and Tornado Web Server

Lightning Talk For Pyvo

Michal Šiška

November 2013

Outline

Differences and Little Remark about Psycopg2

In examples I will use `psycopg2` database adapter (mostly implemented in C as a `libpq` wrapper).

- Synchronous Command Processing
  - `cursor.execute()` resp. `PQexec` waits for the command to be completed
  - It blocks application that might have other work to do (e.g. maintaining a user interface)
- Asynchronous Command Processing
  - Connect to a database server in a nonblocking manner via `connect(async=1)` resp. `PQconnectStart`
  - Polls for connection state via `connection.poll()`
  - When an asynchronous query is being executed, `connection.isexecuting()` returns `True`.

Simple Useless Example

Using `fileno()`, `poll()` and `select()` for asyn. operation.

```
def wait(conn):
    while 1:
        state = conn.poll()
        if state == psycopg2.extensions.POLL_OK:
            break
        elif state == psycopg2.extensions.POLL_WRITE:
            select.select([], [conn.fileno()], [])
        elif state == psycopg2.extensions.POLL_READ:
            select.select([conn.fileno()], [], [])

conn = psycopg2.connect(database='test', async=1)
wait(conn)
curs = conn.cursor()
curs.execute("SELECT pg_sleep(5); SELECT 42;")
wait(curs.connection)
row = curs.fetchone()
```

Overview of Tornado Web Server

Tornado is an open source Python web framework and asynchronous networking library.

- Non-blocking network I/O
- Uses `epoll` or `kqueue`
- Can scale to tens of thousands of open connections
- Ideal for long polling, WebSockets, etc.
- Originally developed at FriendFeed

`epoll` is a scalable I/O event notification mechanism for Linux. It is meant to replace the older POSIX `select(2)` and `poll(2)` system calls, to achieve better performance.

Another Simple Useless Example

Hello World Web Application

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

Putting It All Together: `class Poller`

This has same functionality as the `wait()` function.

```
def update_handler(self):
    state = self.conn.poll()
    if state == psycopg2.extensions.POLL_OK:
        for callback in self.callbacks:
            callback()
    elif state == psycopg2.extensions.POLL_READ:
        ioloop.add_handler(self.conn.fileno(), io_callback,
                IOLoop.READ)
    elif state == psycopg2.extensions.POLL_WRITE:
        ioloop.add_handler(self.conn.fileno(), io_callback,
                IOLoop.WRITE)

def io_callback(self, *args):
    ioloop.remove_handler(self.conn.fileno())
    self.update_handler()
```

Putting It All Together: `class ConnectionPool`

Connection pool is a cache of database connections maintained so that connections can be reused when future requests to the database are required. It is useful for dynamic database-driven website application.

```
def execute(query, params=(), callback=None, conn=None):
    if not conn or conn.closed:
        conn = get_free_conn()
    cursor = conn.cursor()
    cursor.execute(query, params)
    callback = functools.partial(callback, cursor)
    Poller(cursor.conn, (callback,),
           ioloop=ioloop).update_handler()
```

Finally The Useful Example

Handler for URL given to `tornado.web.Application()`.

```
class BaseHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        self.pool.execute(
            "SELECT * FROM something WHERE id=%(id)s;",
            dict(id=id),
            callback=self.on_response)

    def on_response(self, cursor):
        rows = cursor.fetchall()
        self.write(render_tmpl(self.request.path, rows))
        self.finish()
```