Web 编程入门

上期讲到我们把斐波那契数列功能，改写成了面向对象的版本，实现了一个终端接口。

server.py:

```
class BaseHandler:
  def handle(self, request:str):
    pass

class Server:
  def __init__(self, handlerClass):
    self.handlerClass = handlerClass

  def run(self):
    while True:
      request = input()
      self.handlerClass().handle(request)
```

fib_handle.py:

```
from fib import f
from server import BaseHandler, Server

class FibHandler(BaseHandler):
  def handle(self, request:str):
    n = int(request)
    print('f(n)=', f(n))
    pass

server = Server(FibHandler)
server.run()
```

简单 Web 服务器

那如何改成 Web 接口呢。

我们把上面的 Server 换成 HTTP 协议的 Server 就行了。先来看看 Python 中的 HTTP 服务器是怎么样的。

Python 的标准库中提供了一个网页服务器。

```
python -m http.server
```

在终端中运行它。

```
$ python -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

在浏览器中打开便可以看到效果。

这把当前目录列举出来了。接着当浏览这个网页时，再回去看终端。这会，很有意思。

```
$ python -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [07/Mar/2021 15:30:35] "GET / HTTP/1.1" 200 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /favicon.ico HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /apple-touch-icon.png HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:38] "GET / HTTP/1.1" 200 -
```

这是网页访问日志。其中 `GET` 表示网页服务的一种数据访问操作。`HTTP/1.1` 表示使用了 `HTTP` 的 `1.1` 版本的协议。

如何用它来打造我们的斐波那契数列服务。先网上找找样例代码，稍微改改，写一个最简单的 Web 服务器：

```
from http.server import SimpleHTTPRequestHandler, HTTPServer

class Handler(SimpleHTTPRequestHandler):
    def do_GET(self):
      self.send_response(200)
      self.send_header('Content-type', 'text')
      self.end_headers()
      self.wfile.write(bytes("hi", "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)

server.serve_forever()
```

这些是不是很眼熟。几乎跟上面我们使用 `Server` 是一样的。注意到 `SimpleHTTPRequestHandler` 不

是基础类，还有一个叫 BaseHTTPRequestHandler。SimpleHTTPRequestHandler 相对于多处理了一些内容。这些加上斐波那契数列处理功能是容易的。

这里的 127.0.0.1 表示本机的地址，8000 表示本机的端口。端口怎么理解呢。就好像家里的一个窗户一样，是家里跟外界沟通的一个端口。bytes 表示把字符串变成字节。utf-8 是一种字符串编码方式。send_response、send_header 和 end_headers 都是在输出一些内容，来输出 HTTP 协议所规定的内容，好能被浏览器所理解。这样我们在网页里就看到了 hi。

接着试试再从请求从得到参数。

```
from http.server import SimpleHTTPRequestHandler, HTTPServer
from fib import f
from urllib.parse import urlparse,parse_qs

class Handler(SimpleHTTPRequestHandler):
    def do_GET(self):
      self.send_response(200)
      self.send_header('Content-type', 'text')
      self.end_headers()
      parsed = urlparse(self.path)
      qs = parse_qs(parsed.query)
      result = ""
      if len(qs) > 0:
        ns = qs[0]
        if len(ns) > 0:
          n = int(ns)
          result = str(f(n))
      self.wfile.write(bytes(result, "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)

server.serve_forever()
```

有点复杂吧。这里就是在解析一些参数。

```
self.path=/?n=3
parsed=ParseResult(scheme='', netloc='', path='/', params='', query='n=3', fragment='')
qs={'n': ['3']}
ns=['3']
```

```
n=3
```

递归进阶

让我们稍稍重构一下代码。

```
from http.server import SimpleHTTPRequestHandler, HTTPServer
from fib import f
from urllib.parse import urlparse,parse_qs

class Handler(SimpleHTTPRequestHandler):

    def parse_n(self, s):
      parsed = urlparse(s)
      qs = parse_qs(parsed.query)
      if len(qs) > 0:
        ns = qs['n']
        if len(ns) > 0:
          n = int(ns[0])
          return n
      return None

    def do_GET(self):
      self.send_response(200)
      self.send_header('Content-type', 'text')
      self.end_headers()

      result = ""
      n = self.parse_n(self.path)
      if n is not None:
        result = str(f(n))

      self.wfile.write(bytes(result, "utf-8"))
      self.wfile.write(bytes(result, "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)
```

```
server.serve_forever()
```

引入 parse_n 的函数来把从请求路径中解析得到 n 封装在一块。

现在程序有这样的问题。小王请求了斐波那契数列的第 10000 位，过了一些天，小明又请求了斐波那契数列的第 10000 位。两次，小王和小明都等待了半天，才得到结果。我们该如何提高这个 Web 服务的效率呢。

注意到如果 n 相同，f(n) 的值总是一样的。我们进行了一番实验。

```
127.0.0.1 - - [10/Mar/2021 00:33:01] "GET /?n=1000 HTTP/1.1" 200 -
----------------------------------------
Exception occurred during processing of request from ('127.0.0.1', 50783)
Traceback (most recent call last):
    ...
    if v[n] != -1:
IndexError: list index out of range
```

原来数组不够大，那就把 v 数组改成 10000 吧。

```
v = []
for x in range(10000):
   v.append(-1)
```

然而当 n 为 2000 时，出现了递归深度溢出错误：

```
127.0.0.1 - - [10/Mar/2021 00:34:00] "GET /?n=2000 HTTP/1.1" 200 -
----------------------------------------
Exception occurred during processing of request from ('127.0.0.1', 50821)
Traceback (most recent call last):
    ...
    if v[n] != -1:
RecursionError: maximum recursion depth exceeded in comparison
```

然而这一切都还挺快的。

为什么。因为 f(1) 到 f(1000)，都只需要算一次。这意味着当在算 f(1000) 的时候，+ 运算也许只被执行了 1000 次左右。我们知道 Python 的递归深度大约在 1000 左右。这意味着我们可以这样优化程序，如果要算 2000，那我先算 1000 的。不，这样还是可能会出现递归深度溢出错误。如果要算 2000，先算 1200 吧。如果要算 1200，先算 400 吧。

这样算完 400 和 1200 之后，再算 2000，递归深度大概在 800 左右，就不会出现递归深度溢出错误了。

```
v = []
for x in range(1000000):
   v.append(-1)

def fplus(n):
   if n > 800:
      fplus(n-800)
      return f(n)
   else:
      return f(n)

def f(n):
   if v[n] != -1:
      return v[n]
   else:
      a = 0
      if n < 2:
         a = n
      else:
         a = f(n-1) + f(n-2)
      v[n] = a
      return v[n]
```

增加了 `fplus` 函数。

然而不禁让人想，`fplus` 被递归调用 `1000` 次怎么样。1000 * 800 = 800000。当我把 n 设为 80 万之后，又出现递归深度错误了。继续试探了一下，发现事情更复杂。然而这样优化之后，算 2000 是非常轻松的了。

文件读写

似乎把话题岔开了。回到 Web 开发的话题上。第一次请求 `f(400)`，第二次请求 `f(600)`。那么第二次请求时，第一次请求所产生的 v 数组的值，我们是能用上的。然而当我们把程序退出。再启动就用不上了。按我们的方法，斐波那契数列计算是很快的。然而设想，如果很慢怎么办。尤其就如当我们没有引入 v 数组的时候，有着大量重复的计算。这时我们希望能把好不容易得到的结果保存起来。

这时，就引入缓存的概念了。v 数组这里就是一个缓存。不过它只存在于程序生命周期里。程序关闭后，它就没了。怎么办呢。很自然，我们会想到存到文件里去。

如何把 v 数组保存到文件呢。

```
0 0
1 1
2 1
3 2
4 3
...
```

我们的 v 数组可以这样保存。每一行保存为 n f(n)。既然 n 是自然增长的。或许我们可以只保存 f(n) 值。

```
0
1
1
2
3
...
```

来试试吧。

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

open 的第二个参数可以是 a，表示会加在文件末尾；或者是 w，表示会覆盖掉文件。

```
file = open('fib_v', 'a')
file.write('hi')
file.close()
```

运行一下，果然有文件 fib_v。

```
fib_v:

hi
```

当我们再运行一次的时候，变成了这样。

```
hihi
```

如何换行呢。

```
file = open('fib_v', 'a')
file.write('hi\n')
file.close()
```

这会打印一次，出现了 hihihi，没看见换行呢。然而再打印一次，换行了。可见第一次已经打印了换行符，只是在末尾，看不见。

如何读取呢。

```
file = open('fib_v', 'r')
print(file.read())
```

```
$ python fib.py
hihihi
hi
```

接下来，改改我们的斐波那契程序。

```
v = []
for x in range(1000000):
   v.append(-1)


def read():
   file = open('fib_v', 'r')
   s = file.read()
   if len(s) > 0:
      lines = s.split('\n')
      if (len(lines) > 0):
        for i in range(len(lines)):
           v[i] = int(lines[i])


def save():
   file = open('fib_v', 'w')
   s = ''
   start = True
   for vv in v:
      if vv == -1:
         break
      if start == False:
```

```
            s += '\n'
        start = False
        s += str(vv)
    file.write(s)
    file.close()

def fcache(n):
    x = fplus(n)
    save()
    return x

def fplus(n):
    if n > 800:
        fplus(n-800)
        return f(n)
    else:
        return f(n)

def f(n):
    if v[n] != -1:
        return v[n]
    else:
        a = 0
        if n < 2:
            a = n
        else:
            a = f(n-1) + f(n-2)
        v[n] = a
        return v[n]

read()
fcache(10)
save()
```

终于我们写好程序了。程序运行后，fib_v 文件是这样的。

fib_v:

```
0
1
1
2
3
5
8
13
21
34
55
```

看到上面的解析有点麻烦。`\n` 是换行符。有没有更简单统一的解析方式。人们发明了 `JSON` 这件数据格式。

JSON

JSON 的全名是 `JavaScript Object Notation`。以下是 `JSON` 的例子。

```
{"name":"John", "age":31, "city":"New York"}
```

以上这样来表示一种映射。

JSON 有这样基本元素：

1. 数字或字符串
2. 列表
3. 映射

而这些基本元素又可以任意嵌套。就是列表里可以有列表。映射里也可以有列表。等等

```
{
  "name":"John",
  "age":30,
  "cars":[ "Ford", "BMW", "Fiat" ]
}
```

写成一行，和这样写得好看点是意义上的差别的。或许我们可以想象它们的计算图。空格不会影响他们的计算图。

接着我们要把 v 数组变成 `json` 格式的字符串。

```
import json
```

```
v = []
for x in range(1000000):
   v.append(-1)

def fplus(n):
   if n > 800:
      fplus(n-800)
      return f(n)
   else:
      return f(n)

def f(n):
   if v[n] != -1:
      return v[n]
   else:
      a = 0
      if n < 2:
         a = n
      else:
         a = f(n-1) + f(n-2)
      v[n] = a
      return v[n]

fplus(100)
s = json.dump(v)
file = open('fib_j', 'w')
file.write(s)
file.close()
```

当我们这么写的时候。报错了。`TypeError: dump() missing 1 required positional argument: 'fp'`。在 vscode 上可以这样来看到函数定义。

可以用鼠标移动到 dump 上就行。很方便吧。

```
fplus(10)
file = open('fib_j', 'w')
json.dump(v, file)
```

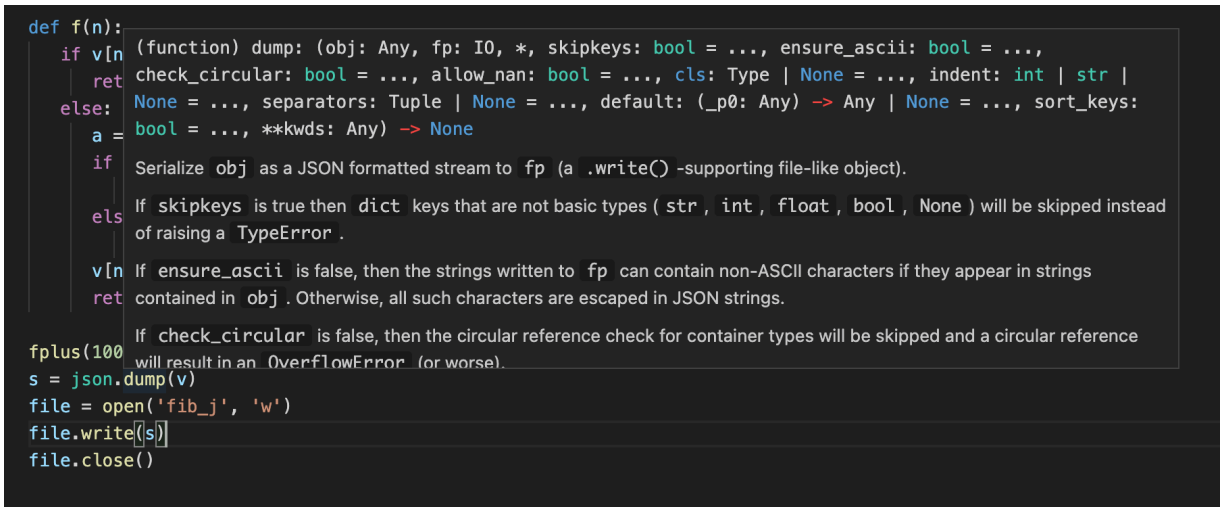

Figure 1: json

```
file.close()
```

计算到 100 显示的数有点多，这里改为 10。原来 dump 的第二个参数传如 `file` 对象就行。

这样可以看到文件:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, -1, -1, -1]
```

注意后面省略了很多-1。

```
def read():
    file = open('fib_j', 'r')
    s = file.read()
    sv = json.loads(s)
    for i in range(len(sv)):
        if sv[i] != -1:
            v[i] = sv[i]
def save():
    file = open('fib_j', 'w')
    json.dump(v, file)
    file.close()

read()

for vv in v:
    if vv!=-1:
```

```
        print(vv)
```

当这样时，可见打印出了：

```
0
1
1
2
3
5
8
13
21
34
55
```

这几个函数一起检查一下：

```
def read():
    file = open('fib_j', 'r')
    s = file.read()
    sv = json.loads(s)
    for i in range(len(sv)):
        v[i] = sv[i]



def save():
    sv = []
    for i in range(len(v)):
        if v[i] != -1:
            sv.append(v[i])
        else:
            break
    file = open('fib_j', 'w')
    json.dump(sv, file)
    file.close()

read()
fplus(100)
```

```
save()
```

然后到文件查看，果然保存了正确的值，而且很整齐。

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711
```

数据库

如果数据很大结构很复杂怎么办。用文件保存的方式会变得很慢很繁琐。这会引入了数据库。也就相当于可编程的 Excel 表。可以很方便用代码进行增删改查的 Excel 表。

在官网文档找到例子。

```
import sqlite3
con = sqlite3.connect('example.db')

cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
               (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
con.close()

for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)
```

cursor 表示游标，也就像光标一样。上面是连接数据库、建表、插入数据、提交更改、关闭连接的意思。最后面的例子则是查询数据的一个示例。

```
import sqlite3

v = []
for x in range(1000000):
```

```
    v.append(-1)

def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE vs(v text)')

def read():
    pass

def save():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    for vv in v:
        if vv != -1:
            cur.execute('INSERT INTO vs VALUES(' + str(vv) + ')')
        else:
            break
    con.commit()
    con.close()

fplus(10)
save()
```

写好了。试试看。

我电脑上已经有了 sqlite3。

```
$ sqlite3
SQLite version 3.32.3 2020-06-18 14:16:19
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

sqlite> .help
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
```

```
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.databases               List names and files of attached databases
.dbconfig ?op? ?val?     List or change sqlite3_db_config() options
.dbinfo ?DB?             Show status information about the database
.dump ?TABLE?            Render database content as SQL
.echo on|off             Turn command echo on or off
.eqp on|off|full|...     Enable or disable automatic EXPLAIN QUERY PLAN
.excel                   Display the output of next command in spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.expert                  EXPERIMENTAL. Suggest indexes for queries
.explain ?on|off|auto?   Change the EXPLAIN formatting mode.  Default: auto
.filectrl CMD ...        Run various sqlite3_file_control() operations
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.import FILE TABLE       Import data from FILE into TABLE
.imposter INDEX TABLE    Create imposter table TABLE on index INDEX
.indexes ?TABLE?         Show names of indexes
.limit ?LIMIT? ?VAL?     Display or change the value of an SQLITE_LIMIT
.lint OPTIONS            Report potential schema issues.
.log FILE|off            Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?       Set output mode
.nullvalue STRING        Use STRING in place of NULL values
.once ?OPTIONS? ?FILE?   Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.parameter CMD ...       Manage SQL parameter bindings
.print STRING...         Print literal STRING
.progress N              Invoke progress handler after every N opcodes
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Exit this program
.read FILE               Read input from FILE
.recover                 Recover as much data as possible from corrupt db.
.restore ?DB? FILE       Restore content of DB (default "main") from FILE
```

```
.save FILE               Write in-memory database into FILE
.scanstats on|off        Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
.selftest ?OPTIONS?      Run tests defined in the SELFTEST table
.separator COL ?ROW?     Change the column and row separators
.session ?NAME? CMD ...  Create or control sessions
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
.stats ?on|off?          Show stats or turn stats on or off
.system CMD ARGS...      Run CMD ARGS... in a system shell
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
.testcase NAME           Begin redirecting output to 'testcase-out.txt'
.testctrl CMD ...        Run various sqlite3_test_control() operations
.timeout MS              Try opening locked tables for MS milliseconds
.timer on|off            Turn SQL timer on or off
.trace ?OPTIONS?         Output each SQL statement as it is run
.vfsinfo ?AUX?           Information about the top-level VFS
.vfslist                 List all available VFSes
.vfsname ?AUX?           Print the name of the VFS stack
.width NUM1 NUM2 ...     Set column widths for "column" mode
```

可以看到有很多的命令。其中.quit 表示退出。

没有的话可以到官网下载，或者运行 brew install sqlite 来安装。

```
$ sqlite3 fib.db
```

```
sqlite> show tables
   ...> ;
Error: near "show": syntax error
sqlite> tables;
Error: near "tables": syntax error
sqlite> .schema
CREATE TABLE vs(v text);
```

一开始我以为像 MySQL 一样。可以用 show tables 来看看有哪些表。后来发现在 SQLite 是这样。MySQL 是另外一种数据库，也是未来我们要学的。

```
sqlite> select * from vs;
```

```
0
1
1
2
3
5
8
13
21
34
55
```

果然，我们正确写入了数据。注意我们用的是 `text`，因为我们数字很大，可能数据库的整数类型保存不了。

```
import sqlite3

v = []
for x in range(1000000):
   v.append(-1)

def fplus(n):
   if n > 800:
      fplus(n-800)
      return f(n)
   else:
      return f(n)

def f(n):
   if v[n] != -1:
      return v[n]
   else:
      a = 0
      if n < 2:
         a = n
      else:
         a = f(n-1) + f(n-2)
```

```
        v[n] = a
        return v[n]

def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE vs(v text)')

def read():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    i = 0
    for row in cur.execute('SELECT * from vs'):
         v[i] = int(row)
    con.close()

def save():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    for vv in v:
        if vv != -1:
            cur.execute('INSERT INTO vs VALUES(' + str(vv) + ')')
        else:
            break
    con.commit()
    con.close()

read()
for i in range(10):
    print(v[i])
```

我们继续加上 `read` 函数。然而运行后，出现了错误。

```
$ python fib_db.py
  ...
  File "fib_db.py", line 27, in create_table
    cur.execute('CREATE TABLE vs(v text)')
```

```
sqlite3.OperationalError: table vs already exists
```

我们无法再创建表，表已经存在了。将语法稍稍改下。

```
def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE IF NOT EXISTS vs(v text)')
```

然而出现了错误。

```
    v[i] = int(row)
TypeError: int() argument must be a string, a bytes-like object or a number, not 'tuple'
```

tuple 是什么。意思是 row 返回了 tuple。让我们打印一下。

```
    for row in cur.execute('SELECT * from vs'):
        print(row)
        v[i] = int(row)
```

结果为：

```
('0',)
```

其实 tuple 和数组差不多。只不过它的元素可以是彼此不一样的，不像数组里的元素都得是同一类型。

```
def read():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    i = 0
    for row in cur.execute('SELECT * from vs'):
        v[i] = int(row[0])
    con.close()
```

这么改。然而很奇怪。输出是这样：

```
55
-1
-1
-1
-1
-1
-1
-1
```

```
-1
-1
```

原来是我们的 `i` 没有自增。

```
    for row in cur.execute('SELECT * from vs'):
        v[i] = int(row[0])
        i += 1
```

这样就对了。

```
0
1
1
2
3
5
8
13
21
34
```

然而我们注意到，当数字很大的时候，在数据库里保存的样子是这样的：

```
4660046610375530309
7540113804746346429
1.22001604151219e+19
1.97402742198682e+19
3.19404346349901e+19
```

再运行一下是这样的。

```
$ python fib_db.py
Traceback (most recent call last):
  File "fib_db.py", line 35, in read
    v[i] = int(row[0])
ValueError: invalid literal for int() with base 10: '1.22001604151219e+19'
```

改一改：

```
cur.execute("INSERT INTO vs VALUES('" +str(vv) + "')")
```

原来注意到这里我们把 `INSERT` 语句两边的单引号改成了双引号，同时给我们的数字字符串加了引

号。如果之前这样写，数据库把我们的字符串当成了数字，而如今，这样用引号括起来，则表示是字符串。

然后就正确了。然而如何把之前的错误数据清空掉。

```
$ sqlite3 fib.db
SQLite version 3.32.3 2020-06-18 14:16:19
Enter ".help" for usage hints.
sqlite> delete * from vs;
```

接下来可以试试其他语句。增删改查。我们这里举了增删查的例子。

练习

- 学生像上面这样类似探索一遍。