QCon
SHANGHAI 2015

探索Python 3.5中async/await特性的实现

广州齐昌网络科技有限公司
赖勇浩

自我介绍

赖勇浩

http://laiyonghao.com

创业者，程序员，社区控

齐昌网络
gzqichang.com

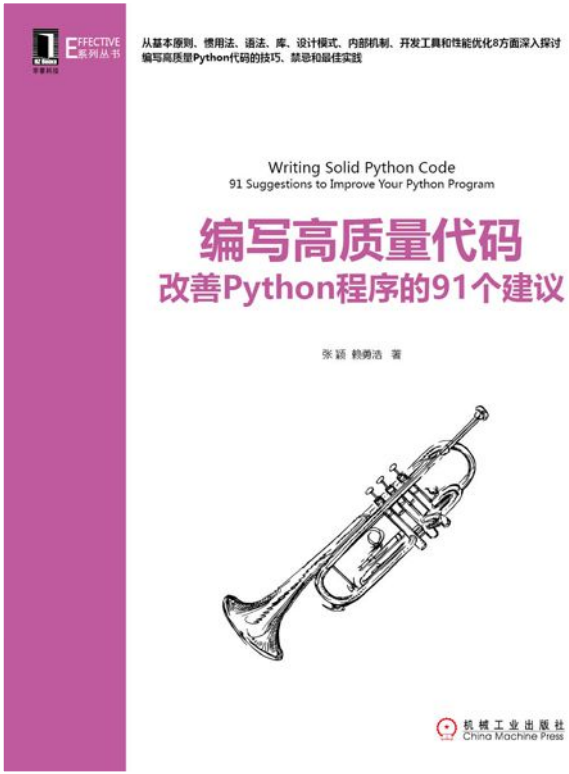
2014 -

python

2005 -

珠三角技术沙龙
http://techparty.org

2009 -



合著有《编写高质量代码：改善Python程序的91个建议》

7

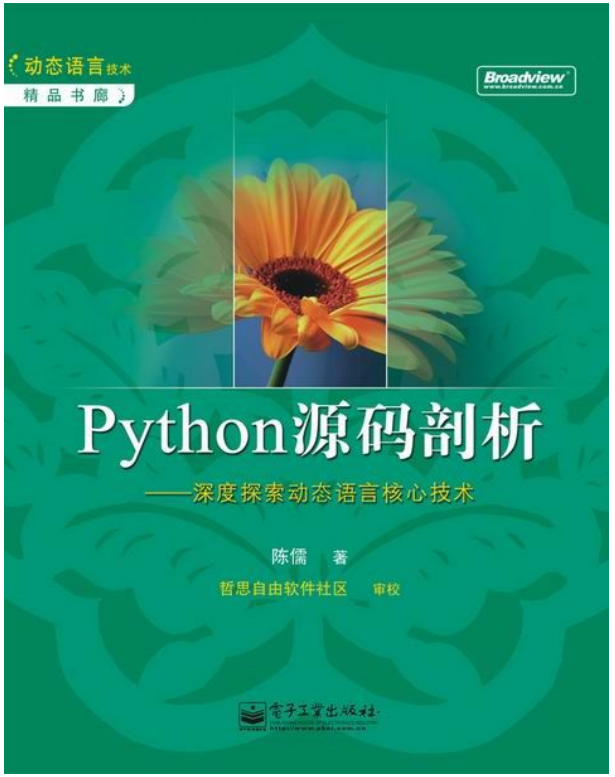
2009年

珠三角技术沙龙
http://techparty.org

8

2015年7月

ArchSummit



背景知识

10

Python
C语言
虚拟机实现
协程概念

Python协程的演化

12

- Python 2.2, 2001.12
  - PEP 255 - Simple Generators

```
def generate_ints(N):
    for i in range(N):
        yield i
```


14

- Python 2.5 2006.8
  - PEP 342 -- Coroutines via Enhanced Generators
  - In 2.5, yield is now an expression
  - Add send()/throw()/close()

- Python 3.3 2012.9
  - PEP 0380 -- Syntax for Delegating to a Subgenerator

16

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```


17

- Python 3.5 2015.9
  - PEP 0492 -- Coroutines with async and await syntax

```
async def read_data(db):
    data = await db.fetch('SELECT ...')
    ...
```


19

async/await的意义

- 定义了原生协程，与生成器彻底区分开来
- 解决with/for的异步需求。

async with

async with EXPR as VAR:
    BLOCK

async with

- 同步的 __enter__/__exit__
  - 无法实现获取、释放资源时复杂耗时的操作异步
- 异步的 __aenter__/__aexit__

async with

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```


23

async for

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

async for

- 同步的 __iter__/__next__
  - 无法实现获取、释放资源时复杂耗时的操作异步
- 异步的 __aiter__/__anext__

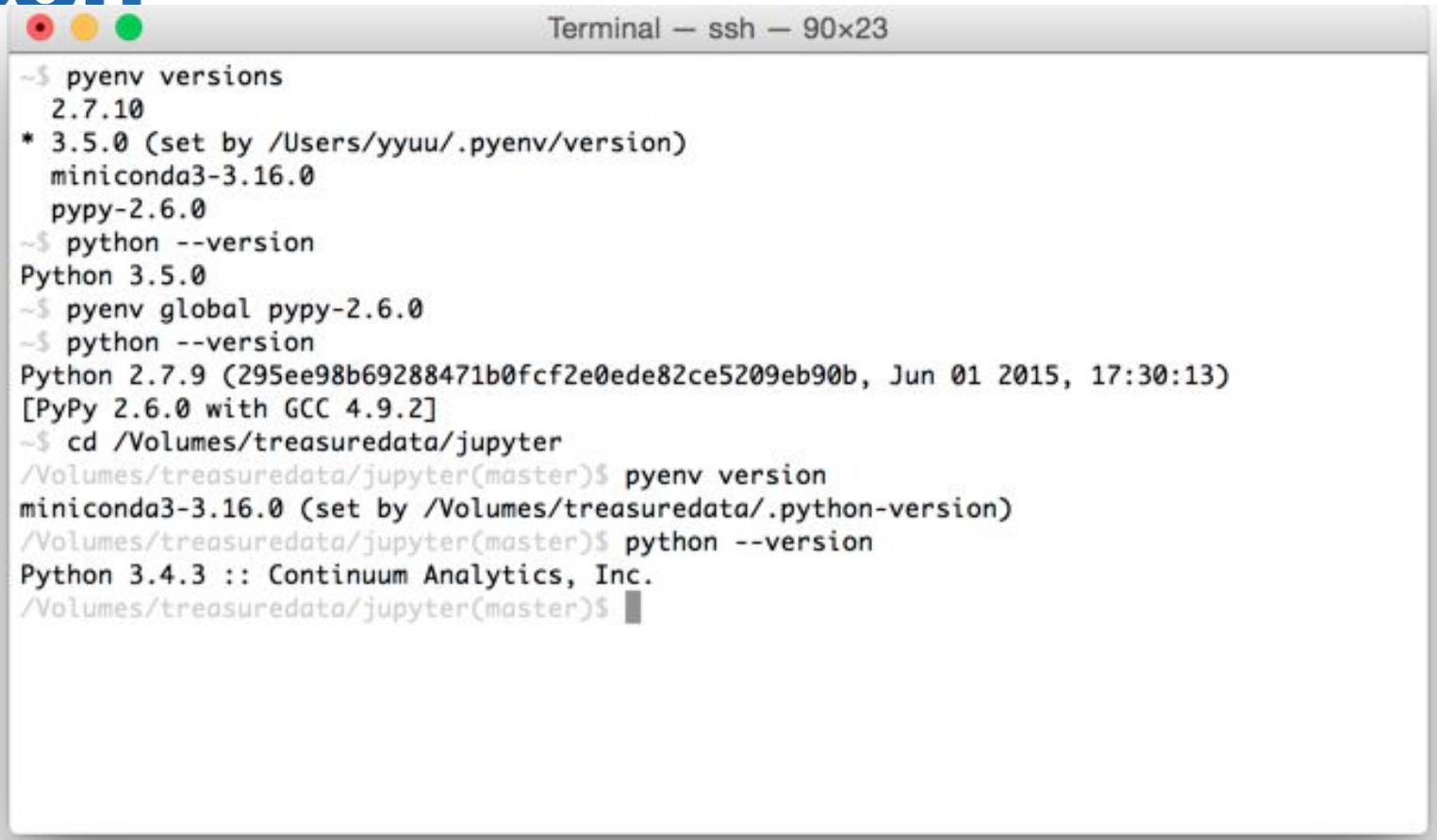
async for

```
class AsyncIterable:
    async def __aiter__(self):
        return self
    async def __anext__(self):
        data = await self.fetch_data()
        if data:
            return data
        else:
            raise StopAsyncIteration
    async def fetch_data(self):
        ...
```

体验Python协程

27

pyenv install 3.5.0



神器！

https://github.com/yyuu/pyenv

```
mkdir py35lab
cd py35lab
pyenv local 3.5.0
```

python –version
Python 3.5.0

31

探索async/await的实现

32

async/await的字节码

```
async def foo():
    return 42
async def bar():
    print(await foo())

import dis
dis.dis(bar)
```

async/await的字节码

```
9           0 LOAD_GLOBAL              0 (print)
            3 LOAD_GLOBAL              1 (foo)
            6 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
            9 GET_AWAITABLE
           10 LOAD_CONST               0 (None)
           13 YIELD_FROM
           14 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
           17 POP_TOP
           18 LOAD_CONST               0 (None)
           21 RETURN_VALUE
```

GET_AWAITABLE

```
TARGET(GET_AWAITABLE) {
    PyObject *iterable = TOP();
    PyObject *iter = _PyCoro_GetAwaitableIter(iterable);

    Py_DECREF(iterable);

    SET_TOP(iter); /* Even if it's NULL */

    if (iter == NULL) {
        goto error;
    }

    DISPATCH();
}
```

QCon SHANGHAI 2015

_PyCoro_GetAwaitableIter

```
PyObject *
_PyCoro_GetAwaitableIter(PyObject *o)
{
    unaryfunc getter = NULL;
    PyTypeObject *ot;

    if (PyCoro_CheckExact(o) || gen_is_coroutine(o
        /* 'o' is a coroutine. */
        Py_INCREF(o);
        return o;
    }

    ot = Py_TYPE(o);
    if (ot->tp_as_async != NULL) {
        getter = ot->tp_as_async->am_await;
    }
    if (getter != NULL) {
        PyObject *res = (*getter)(o);
        if (res != NULL) {
            if (PyCoro_CheckExact(res) || gen_is_c
                /* __await__ must return an *itera
                   a coroutine or another awaitab
                PyErr_SetString(PyExc_TypeError,
                                "__await__() retu
                Py_CLEAR(res);
            } else if (!PyIter_Check(res)) {
                PyErr_Format(PyExc_TypeError,
                             "__await__() returned
                             "of type '%.100s'",
                             Py_TYPE(res)->tp_name
```

_PyCoro_GetAwaitableIter

* This helper function returns an awaitable for `o`:
* - `o` if `o` is a coroutine-object;
* - `type(o)->tp_as_async->am_await(o)`

am_await

```
static PyObject * coro_await(PyCoroObject *coro)
{
    PyCoroWrapper *cw = ...New(PyCoroWrapper,
&_PyCoroWrapper_Type);
    cw->cw_coroutine = coro;
    return (PyObject *)cw;
}
```

_PyCoroWrapper_Type

```
PyTypeObject _PyCoroWrapper_Type = {
    ...
    PyObject_SelfIter,                  /* tp_iter */
    (iternextfunc)coro_wrapper_iternext, /* tp_iternext */
    coro_wrapper_methods,               /* tp_methods */
    ...
};
```

GET_AWAITABLE让coroutine的返回值（Awaitable）入栈

async/await的字节码

```
9           0 LOAD_GLOBAL              0 (print)
            3 LOAD_GLOBAL              1 (foo)
            6 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
            9 GET_AWAITABLE
           10 LOAD_CONST               0 (None)
           13 YIELD_FROM
           14 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
           17 POP_TOP
           18 LOAD_CONST               0 (None)
           21 RETURN_VALUE
```


41

QCon SHANGHAI 2015

YIELD_FROM

```
TARGET(YIELD_FROM) {
    PyObject *v = POP();
    PyObject *reciever = TOP();
    int err;
    if (PyGen_CheckExact(reciever) || PyCoro_Che
        retval = _PyGen_Send((PyGenObject *)rec:
    } else {
        _Py_IDENTIFIER(send);
        if (v == Py_None)
            retval = Py_TYPE(reciever)->tp_iter
        else
            retval = _PyObject_CallMethodIdObjA
    }
    Py_DECREF(v);
    if (retval == NULL) {
        PyObject *val;
        if (tstate->c_tracefunc != NULL
                && PyErr_ExceptionMatches(PyExc_
            call_exc_trace(tstate->c_tracefunc,
        err = _PyGen_FetchStopIterationValue(&va
        if (err < 0)
            goto error;
        Py_DECREF(reciever);
        SET_TOP(val);
        DISPATCH();
    }
    /* x remains on stack, retval is value to be
    f->f_stacktop = stack_pointer;
    why = WHY_YIELD;
    /* and repeat...  */
```

YIELD_FROM

- 获取栈顶元素, v
- 调用_PyGen_Send(..., v)，并返回结果。

async/await真相

- Native Corotine 就是换了马甲的 generator

QCon SHANGHAI 2015

PythonVM中的协程

Corotine

```
typedef struct {
    _PyGenObject_HEAD(cr)
} PyCoroObject;
```

Generator

```
typedef struct {
_PyGenObject_HEAD(gi)
} PyGenObject;
```

_PyGenObject_HEAD

```
#define _PyGenObject_HEAD(prefix) \
PyObject_HEAD                     \
struct _frame *prefix##_frame;    \
char prefix##_running;            \
PyObject *prefix##_code;          \
PyObject *prefix##_weakreflist;   \
PyObject *prefix##_name;          \
PyObject *prefix##_qualname;
```

协程从何处来？

- 代码编译、执行，就是一个 PyCodeObject* co
- co->co_flag 标识了类型
- Py3.5: CO_COROUTINE, CO_ITERABLE_COROUTINE
- async def 使 co_flag 具有 CO_COROTINE

```
if (is_coro) {
        gen = PyCoro_New(f, name, qualname);
    }
```

协程的运行与终止

- 解释器遇到 YIELD_FROM，调用 _PyGen_Send，主要逻辑在 gen_send_ex

gen_send_ex

- 做好参数、状态的检查工作
- 参数压栈
- 保存要返回栈帧（PyFrameObject)
- 设置运行状态标志
- 调用 PyEval_EvalFrameEx 从自己的栈帧执行代码
- 重置运行状态标志
- 恢复现场，异常处理，释放资源，返回结果

Q&A

50