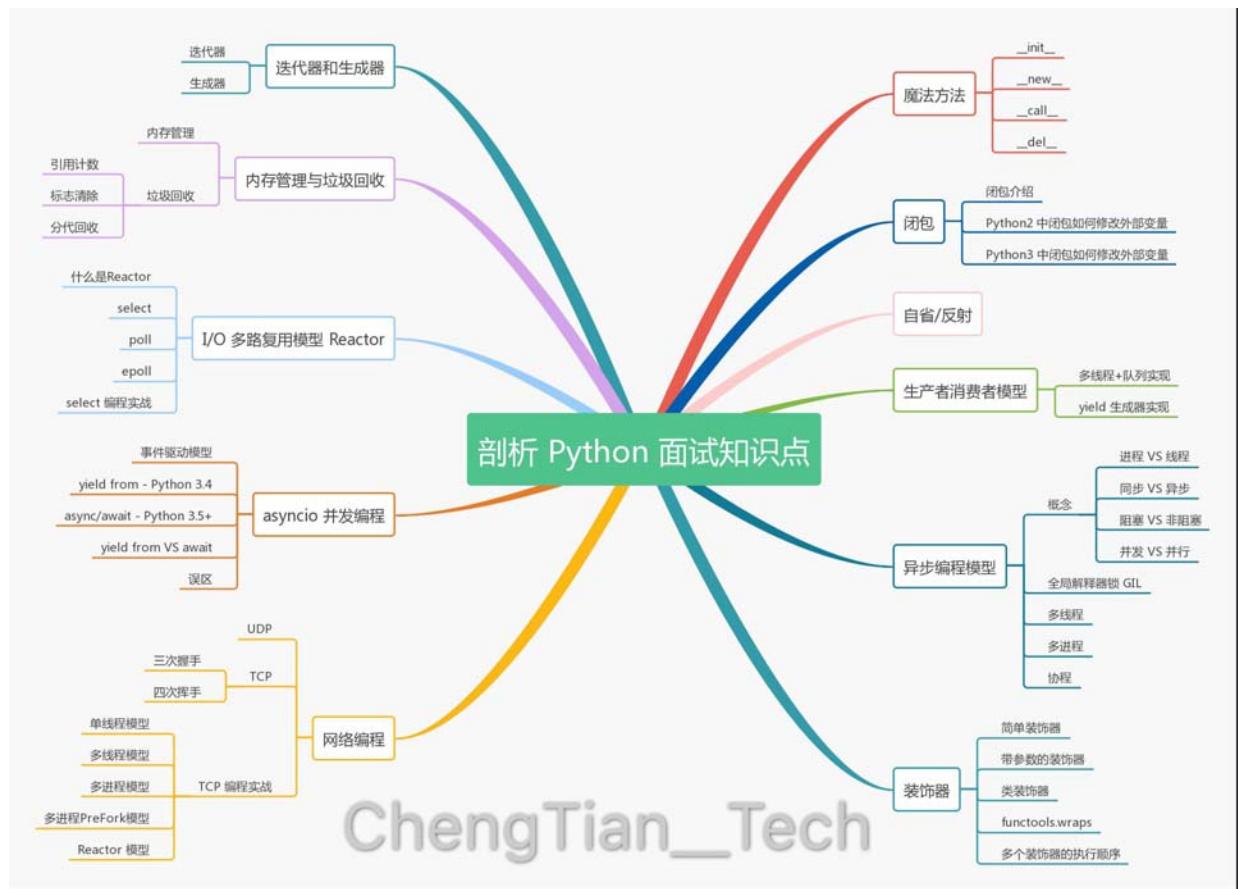


全面剖析 Python 面试知识点

知识点整理基于 Python3。文章篇幅较长，先看下面图了解全文。



Python 魔法方法

在 Python 中用双下划线 `__` 包裹起来的方法被称为魔法方法，可以用来给类提供算术、逻辑运算等功能，让这些类能够像原生的对象一样用更标准、简洁的方式进行这些操作。

下面介绍常常被问到的几个魔法方法。

1.1 `__init__`

`__init__` 方法做的事情是在对象创建好之后初始化变量。很多人以为 `__init__` 是构造方法，其实不然，真正创建实例的是 `__new__` 方法，下面会讲它，先来看看 `__init__` 方法。

GitChat 用户专享，请尊重版权

```
class Person(object):
    def __init__(self, name, age):
        print("in __init__")
        self.name = name
        self.age = age

p = Person("TianCheng", 27)
print("p:", p)
```

输出

```
in __init__
p: <__main__.Person object at 0x105a689e8>
```

明白 `__init__` 负责初始化工作，平常也是我们经常用到的。。

1.2 `__new__`

构造方法: `__new__ (cls, [...])`

`new` 是 Python 中对象实例化时所调用的第一个函数，在`init`之前被调用。`new` 将 class 作为他的第一个参数，并返回一个这个 class 的 instance。而`init`是将 instance 作为参数，并对这个 instance 进行初始化操作。每个实例创建时都会调用`new`函数。下面来看一个例子：

```
class Person(object):
    def __new__(cls, *args, **kwargs):
        print("in __new__")
        instance = super().__new__(cls)
        return instance

    def __init__(self, name, age):
        print("in __init__")
        self._name = name
        self._age = age

p = Person("TianCheng", 27)
print("p:", p)
```

输出结果

```
in __new__
in __init__
p: <__main__.Person object at 0x106ed9c18>
```

GitChat 用户专享，请尊重版权

可以看到先执行 new 方法创建对象，然后 init 进行初始化。假设将new方法中不返还该对象，会有什么结果了？

```
class Person(object):
    def __new__(cls, *args, **kwargs):
        print("in __new__")
        instance = super().__new__(cls)
        #return instance

    def __init__(self, name, age):
        print("in __init__")
        self._name = name
        self._age = age

p = Person("TianCheng", 27)
print("p:", p)

# 输出:
in __new__
p: None
```

发现如果 new 没有返回实例化对象，init 就没法初始化了。

如何使用 new 方法实现单例（高频考点）

```
class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, "_instance"):
            cls._instance = cls.__new__(cls, *args, **kwargs)
        return cls._instance

s1 = Singleton()
s2 = Singleton()
print(s1)
print(s2)
```

输出结果

```
<__main__.Singleton object at 0x1031cf8>
<__main__.Singleton object at 0x1031cf8>
```

s1、s2 内存地址一致，实现单例效果。

1.3 __call__

GitChat 用户专享，请尊重版权

`__call__` 方法，先需要明白什么是可调用对象，平时自定义的函数、内置函数和类都属于可调用对象，但凡是可以把一对括号（）应用到某个对象身上都可称之为可调用对象，判断对象是否为可调用对象可以用函数 `callable`。举例如下：

```
class A(object):
    def __init__(self):
        print("__init__ ")
        super(A, self).__init__()

    def __new__(cls):
        print("__new__ ")
        return super(A, cls).__new__(cls)

    def __call__(self): # 可以定义任意参数
        print('__call__ ') 

a = A()
a()
print(callable(a)) # True
```

输出：

```
__new__
__init__
__call__
True
```

执行 `a()` 才会打印出 `__call__`。`a` 是一个实例化对象，也是一个可调用对象。

1.4 `__del__`

`__del__` 析构函数，当删除一个对象时，则会执行此方法，对象在内存中销毁时，自动会调用此方法。举例：

```
class People:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def __del__(self): # 在对象被删除的条件下，自动执行
        print('__del__')

obj=People("Tiancheng", 27)
#del obj #obj.__del__() #先删除的情况下，直接执行__del__
```

输出结果

-- del --

闭包和自省

2.1 闭包

2.1.1 什么闭包

简单的说，如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包（closure）。来看一个简单的例子：

```
>>>def addx(x):
>>>     def adder(y): return x + y
>>>     return adder
>>> c = addx(8)
>>> type(c)
<type 'function'>
>>> c.__name__
'adder'
>>> c(10)
18
```

其中 adder(y) 函数就是闭包。

2.1.2 实现一个闭包并可以修改外部变量

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
c = foo()
print(c())
```

有上面一个小例子，目的是每次执行一次，a 自增 1，执行后是否正确了？显示会报下面错误。

```
local variable 'a' referenced before assignment
```

GitChat 用户专享，请尊重版权

原因是 bar() 函数中会把 a 作为局部变量，而 bar 中没有对 a 进行声明。

如果面试官问你，在 Python2 和 Python3 中如何修改 a 的值了。

Python3 中只需引入 nonlocal 关键字即可：

```
def foo():
    a = 1
    def bar():
        nonlocal a
        a = a + 1
        return a
    return bar
c = foo()
print(c()) # 2
```

而在 Python2 中没有 nonlocal 关键字，该如何实现了：

```
def foo():
    a = [1]
    def bar():
        a[0] = a[0] + 1
        return a[0]
    return bar
c = foo()
print(c()) # 2
```

需借助可变变量实现，比如 dict 和 list 对象。

闭包的一个常用场景就是装饰器。后面会讲到。

2.2 自省（反射）

自省，也可以说是反射，自省在计算机编程中通常指这种能力：检查某些事物以确定它是什么、它知道什么以及它能做什么。

与其相关的主要方法：

- **hasattr(object, name)**
检查对象是否具体 name 属性。返回 bool
- **getattr(object, name, default)**
获取对象的 name 属性。
- **setattr(object, name, value)**
给对象设置 name 属性
- **delattr(object, name)**
给对象删除 name 属性

GitChat 用户专享，请尊重版权

- **dir([object])**
获取对象大部分的属性
- **isinstance(name, object)**
检查 name 是不是 object 对象
- **type(object)**
查看对象的类型
- **callable(object)**
判断对象是否是可调用对象

```
>>> class A:  
...     a = 1  
...  
>>> hasattr(A, 'a')  
True  
>>> getattr(A, 'a')  
1  
>>> setattr(A, 'b', 1)  
>>> getattr(A, 'b')  
1  
>>> delattr(A, 'b')  
>>> hasattr(A, 'b')  
False  
>>> dir(A)  
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',  
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',  
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
'__repr__', '__setattr__', '__sizeof__', '__str__',  
'__subclasshook__', '__weakref__', 'a']  
>>> isinstance(1, int)  
True  
>>> type(A)  
<class 'type'>  
>>> type(1)  
<class 'int'>  
>>> callable(A)  
True
```

装饰器和迭代器

装饰器

装饰器本质上是一个 Python 函数或类，它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能（设计模式中的装饰器模式），装饰器的返回值也是一个函数/类

GitChat 用户专享，请尊重版权

对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。

3.1.1 简单装饰器

先来看一个之前闭包的例子：

```
def my_logging(func):  
  
    def wrapper():  
        print("{} is running.".format(func.__name__))  
        return func() # 把 foo 当做参数传递进来时，执行func()就相当于  
        执行foo()  
    return wrapper  
  
def foo():  
    print("this is foo function.")  
  
foo = my_logging(foo) # 因为装饰器 my_logging(foo) 返回的时函数对象  
wrapper, 这条语句相当于 foo = wrapper  
foo() # 执行foo相当于执行wrapper
```

但在 Python 里有 @语法糖，则可以直接这样做：

```
def my_logging(func):  
  
    def wrapper():  
        print("{} is running.".format(func.__name__))  
        return func()  
    return wrapper  
  
@my_logging  
def foo():  
    print("this is foo function.")  
  
foo()
```

上面二者都会有如下打印结果：

```
foo is running.  
this is foo function.
```

my_logging 就是一个装饰器，它一个普通的函数，它把执行真正业务逻辑的函数 func 包裹在其中，看起来像 foo 被 my_logging 装饰了一样 my_logging 返回的也是一个函数，这个函数的名字叫 wrapper。在这个例子中，函数进入和退出时，被称为一个横切面，这种编程方式被称为面向切面的编程（AOP）。

GitChat 用户专享，请尊重版权

如果 foo 带有参数，如何将参数带到 wrapper 中了？

```
def my_logging(func):

    def wrapper(*args, **kwargs):
        print("{} is running.".format(func.__name__))
        return func(*args, **kwargs)
    return wrapper

@my_logging
def foo(x, y):
    print("this is foo function.")
    return x + y

print(foo(1, 2))
```

可以通过 *args, *kwargs 接收参数，然后带入 func 中执行，上面执行结果为：

```
foo is running.
this is foo function.
3
```

3.1.2 带参数的装饰器

装饰器的语法允许我们在调用时，提供其它参数，比如 @decorator(a)。这样就大大增加了灵活性，比如在日志告警场景中，可以根据不同的告警定告警等级：info/warn 等。

```
def my_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "info":
                print("{} is running. level:
".format(func.__name__), level)
            elif level == "warn":
                print("{} is running. level:
".format(func.__name__), level)
            return func(*args, **kwargs)
        return wrapper
    return decorator

@my_logging(level="info")
def foo(name="foo"):
    print("{} is running".format(name))

@my_logging(level="warn")
def bar(name="bar"):
    print("{} is running".format(name))
```

```
foo()  
bar()
```

结果输出：

```
foo is running. level: info  
foo is running  
bar is running. level: warn  
bar is running
```

上面的 my_logging 是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器。我们可以将它理解为一个含有参数的闭包。当使用 @my_logging(level="info") 调用的时候，Python 能够发现这一层的封装，并把参数传递到装饰器的环境中。

@my_logging(level="info") 等价于 @decorator

3.1.3 类装饰器

装饰器不仅可以是函数，还可以是类，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的 __call__ 方法，当使用 @ 形式将装饰器附加到函数上时，就会调用此方法。

```
class MyLogging(object):  
  
    def __init__(self, func):  
        self._func = func  
  
    def __call__(self, *args, **kwargs):  
        print("class decorator starting.")  
        a = self._func(*args, **kwargs)  
        print("class decorator end.")  
        return a  
  
@MyLogging  
def foo(x, y):  
    print("foo is running")  
    return x + y  
  
print(foo(1, 2))
```

输出结果

```
class decorator starting.  
foo is running
```

```
class decorator end.  
3
```

3.1.4 functools.wraps

Python 中还有一个装饰器的修饰函数 `functools.wraps`, 先来看看它的作用是什么? 先来看看有一个问题存在, 因为原函数被装饰函数装饰后, 发生了一下变化:

```
def my_logging(func):  
  
    def wrapper(*args, **kwargs):  
        print("{} is running.".format(func.__name__))  
        return func(*args, **kwargs)  
    return wrapper  
  
@my_logging  
def foo(x, y):  
    """  
    add function  
    """  
    print("this is foo function.")  
    return x + y  
  
print(foo(1, 2))  
print("func name:", foo.__name__)  
print("doc:", foo.__doc__)
```

打印结果:

```
foo is running.  
this is foo function.  
3  
func name: wrapper  
doc: None
```

问题出来了, `func name` 应该打印出 `foo` 才对, 而且 `doc` 也不为 `None`。由此发现原函数被装饰函数装饰之后, 元信息发生了改变, 这明显不是我们想要的, Python 里可以通过 `functools.wraps` 来解决, 保持原函数元信息。

```
from functools import wraps  
  
def my_logging(func):  
  
    @wraps(func)  
    def wrapper(*args, **kwargs):  
        print("{} is running.".format(func.__name__))
```

GitChat 用户专享，请尊重版权

```
return func(*args, **kwargs)
return wrapper

@my_logging
def foo(x, y):
    """
    add function
    """
    print("this is foo function.")
    return x + y

print(foo(1, 2))
print("func name:", foo.__name__)
print("doc:", foo.__doc__)
```

输出结果：

```
foo is running.
this is foo function.
3
func name: foo
doc:
    add function
```

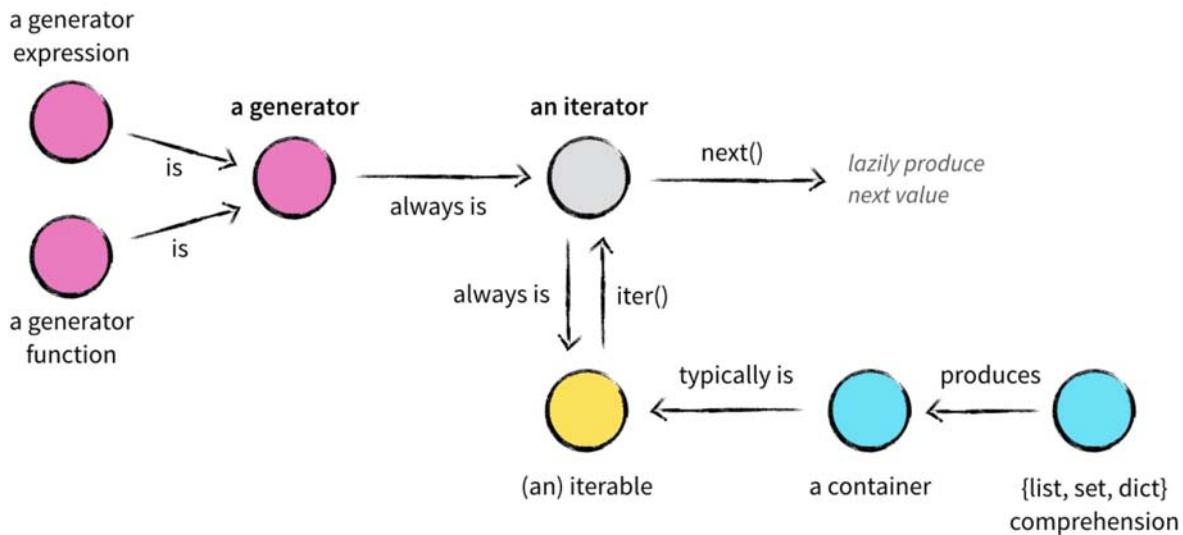
3.1.5 多个装饰器的执行顺序

```
@a
@b
@c
def f():
    pass
```

执行顺序为 `f = a(b(c(f)))`

3.2 迭代器 VS 生成器

先来看一个关系图：



3.2.1 container(容器)

container 可以理解为把多个元素组织在一起的数据结构，container 中的元素可以逐个地迭代获取，可以用 in, not in 关键字判断元素是否包含在容器中。在 Python 中，常见的 container 对象有：

```
list, deque, ....  
set, frozensets, ....  
dict, defaultdict, OrderedDict, Counter, ....  
tuple, namedtuple, ...  
str
```

举例：

```
>>> assert 1 in [1, 2, 3]      # lists  
>>> assert 4 not in [1, 2, 3]  
>>> assert 1 in {1, 2, 3}      # sets  
>>> assert 4 not in {1, 2, 3}  
>>> assert 1 in (1, 2, 3)      # tuples  
>>> assert 4 not in (1, 2, 3)
```

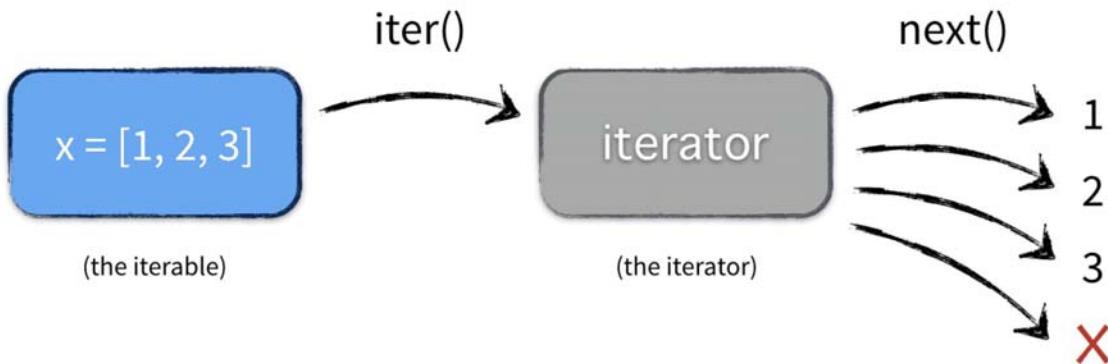
3.2.2 可迭代对象(iterables) vs 迭代器(iterator)

大部分的 container 都是可迭代对象，比如 list or set 都是可迭代对象，可以说只要是可以通过一个迭代器的都可以称作可迭代对象。下面看一个例子：

```
>>> x = [1, 2, 3]  
>>> y = iter(x)  
>>> next(y)  
1
```

```
>>> next(y)
2
>>> type(x)
<class 'list'>
>>> type(y)
<class 'list_iterator'>
```

可见，`x`是可迭代对象，这里也叫 container。`y`则是迭代器，且实现了`__iter__`和`__next__`方法。它们之间的关系是：



那什么是迭代器了？上面例子中有 2 个方法 iter and next。可见通过 iter 方法后就是迭代器。

它是一个带状态的对象，调用 next 方法的时候返回容器中的下一个值，可以说任何实现了 iter 和 next 方法的对象都是迭代器，iter 返回迭代器自身，next 返回容器中的下一个值，如果容器中没有更多元素了，则抛异常。

迭代器就像一个懒加载的工厂，等到有人需要的时候才给它生成值返回，没调用的时候就处于休眠状态等待下一次调用。

3.2.3 生成器(generator)

生成器一定是迭代器，是一种特殊的迭代器，特殊在于它不需要再像上面的 iter() 和 next 方法了，只需要一个 yield 关键字。下面来看一个例子：

用生成器实现斐波拉契

```
# content of test.py
def fib(n):
    prev, curr = 0, 1
    while n > 0:
        yield curr
        prev, curr = curr, curr + prev
        n -= 1
```

到终端执行 fib 函数

GitChat 用户专享，请尊重版权

```
>>> from test import fib
>>> y = fib(10)
>>> next(y)
1
>>> type(y)
<class 'generator'>
>>> next(y)
1
>>> next(y)
2
```

fib 就是一个普通的 Python 函数，它特殊的地方在于函数体中没有 return 关键字，函数的返回值是一个生成器对象(通过 yield 关键字)。当执行 f=fib() 返回的是一个生成器对象，此时函数体中的代码并不会执行，只有显示或隐示地调用 next 的时候才会真正执行里面的代码。

假设有千万个对象，需要顺序调取，如果一次性加载到内存，对内存是极大的压力，有生成器之后，可以需要的时候去生成一个，不需要的则也不会占用内存。

平常可能还会遇到一些生成器表达式，比如：

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x102d79a20>
>>> next(a)
0
>>> next(a)
1
>>> a.close()
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

这些小技巧也是非常有用的。close 可以关闭生成器。生成器中还有一个 send 方法，其中 send(None) 与 next 是等价的。

```
>>> def double_inputs():
...     while True:
...         x = yield
...         yield x * 2
...
>>> generator = double_inputs()
>>> generator.send(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

```
>>> generator.send(None)
>>> generator.send(10)
20
>>> next(generator)
>>> generator.send(20)
40
```

从上面的例子中可以看出，生成器可以接收参数，通过 `send(value)` 方法，且第一次不能直接 `send(value)`，需要 `send(None)` 或者 `next()` 执行之后。也就是说调用 `send` 传入非 `None` 值前，生成器必须处于挂起状态，否则将抛出异常。

3.2.4 迭代器和生成器的区别

可能你看完上面的，有点好奇到底他们二者有什么区别了？

- 迭代器是一个更抽象的概念，任何对象，如果它有 `next` 方法（`next` python3，python2 是 `next`方法）和`iter`方法，则可以称作迭代器。
- 每个生成器都是一个迭代器，但是反过来不行。通常生成器是通过调用一个或多个 `yield` 表达式构成的函数 `s` 生成的。同时满足迭代器的定义。
- 生成器能做到迭代器能做的所有事，而且因为自动创建了 `iter()` 和 `next()`方法，生成器显得特别简洁，而且生成器也是高效的。

内存管理和垃圾回收

内存管理

Python 中一切皆对象，对象又可以分为可变对象和不可变对象。二者可以通过原地修改，如果修改后地址不变，则是可变对象，否则为不可变对象，地址信息可以通过 `id()` 进行查看。

```
>>> a = 10
>>> id(a)
4339392960
>>> a = 11
>>> id(a)
4339392992
>>> a = [1, 2]
>>> id(a)
4342877192
>>> a.append(3)
>>> a
[1, 2, 3]
```

```
>>> id(a)
4342877192
```

Python 有内存池机制，Pymalloc 机制，用于对内存的申请和释放管理。先来看一下为什么有内存池：

当创建大量消耗小内存的对象时，c 中频繁调用 new/malloc 会导致大量的内存碎片，致使效率降低。

内存池的概念就是预先在内存中申请一定数量的，大小相等的内存块留作备用，当有新的内存需求时，就先从内存池中分配内存给这个需求，不够了之后再申请新的内存。这样做最显著的优势就是能够减少内存碎片，提升效率。

查看源码，可以看到 Pymalloc 对于小的对象，Pymalloc 会在内存池中申请空间，一般是少于236kb，如果是大的对象，则直接调用 new/malloc 来申请新的内存空间。

有了内存的创建，那就需要回收，垃圾回收机制，也是 Python 面试当中必问的一个知识点，接下来看看垃圾回收机制是什么？

4.2 垃圾回收机制

垃圾回收机制，Python 采用 GC 作为自动内存管理机制，GC 要做的有 2 件事，一是找到内存中无用的垃圾对象资源，二是清除找到的这些垃圾对象，释放内存给其他对象使用。

如何实现上述 2 点了，Python 采用了引用计数为主，标志清除和分代回收为辅的策略。

4.2.1 引用计数

查看源码，每一个对象，在源码里就是一个结构体表示，都会有一个计数字段。

```
typedef struct_object {
    int ob_refcnt;
    struct_typeobject *ob_type;
} PyObject;
```

PyObject 是每个对象必有的内容，其中 ob_refcnt 就是做为引用计数。当一个对象有新的引用时，它的 ob_refcnt 就会增加，当引用它的对象被删除，它的 ob_refcnt 就会减少。

一旦对象的引用计数为 0，该对象立即被回收，对象占用的内存空间将被释放。

此算法的优点和缺点都是非常明显的：

优点

- 简单

GitChat 用户专享，请尊重版权

- 实时性：一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。

缺点

- 需要额外的空间维护引用计数。
- 不能解决对象的**循环引用**。(主要缺点)

接下来说明一下什么是循环引用：

A 和 B 相互引用而且没有外部引用 A 与 B 中的任何一个。也就是对象之间互相应用，导致引用链形成一个环。

```
>>>>>a = {} #对象A的引用计数为 1
>>>b = {} #对象B的引用计数为 1
>>>a['b'] = b #B的引用计数增1
>>>b['a'] = a #A的引用计数增1
>>>del a #A的引用减 1, 最后A对象的引用为 1
>>>del b #B的引用减 1, 最后B对象的引用为 1
```

执行 del 后，A、B 对象已经没有任何引用指向这两个对象，但是这两个对象各包含一个对方对象的引用，虽然最后两个对象都无法通过其它变量来引用这两个对象了，这对 GC 来说就是两个非活动对象或者说是垃圾对象。理论上是需要被回收的。

按上面的引用计数原理，要计数为 0 才会回收，但是他们的引用计数并没有减少到零。因此如果是使用引用计数法来管理这两对象的话，他们并不会被回收，它会一直驻留在内存中，就会造成了内存泄漏（内存空间在使用完毕后未释放）。

为了解决对象的循环引用问题，Python 引入了标记清除和分代回收两种 GC 机制。

4.2.2 标记清除

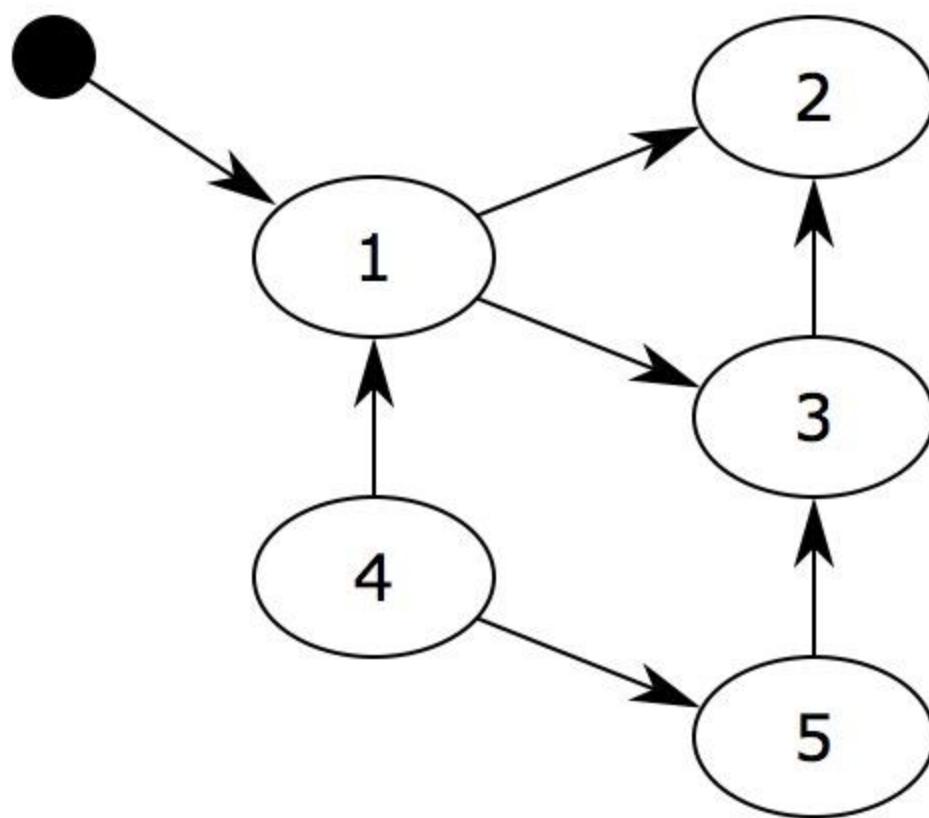
标记清除主要是解决循环引用问题。

标记清除算法是一种基于追踪回收（tracing GC）技术实现的垃圾回收算法。

它分为两个阶段：第一阶段是标记阶段，GC 会把所有的 活动对象 打上标记，第二阶段是把那些没有标记的对象 非活动对象 进行回收。那么 GC 又是如何判断哪些是活动对象哪些是非活动对象的呢？

对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象（root object）出发，沿着有向边遍历对象，可达的（reachable）对象标记为活动对象，不可达的对象就是要被清除的非活动对

象。根对象就是全局变量、调用栈、寄存器。



在上图中，我们把小黑圈视为全局变量，也就是把它作为 root object，从小黑圈出发，对象 1 可直达，那么它将被标记，对象 2、3 可间接到达也会被标记，而 4 和 5 不可达，那么 1、2、3 就是活动对象，4 和 5 是非活动对象会被 GC 回收。

标记清除算法作为 Python 的辅助垃圾收集技术主要处理的是容器对象(container，上面讲迭代器有提到概念)，比如 list、dict、tuple 等，因为对于字符串、数值对象是不可能造成循环引用问题。Python 使用一个双向链表将这些容器对象组织起来。

Python 这种简单粗暴的标记清除算法也有明显的缺点：清除非活动的对象前它必须顺序扫描整个堆内存，哪怕只剩下小部分活动对象也要扫描所有对象。

4.2.3 分代回收

分代回收是一种以空间换时间的操作方式。

Python 将内存根据对象的存活时间划分为不同的集合，每个集合称为一个代，Python 将内存分为了 3“代”，分别为年轻代（第 0 代）、中年代（第 1 代）、老年代（第 2 代），他们对应的是 3 个链表，它们的垃圾收集频率与对象的存活时间的增大而减小。新创建的对象都会分配在年轻代，年轻代链表的总数达到上限时，Python 垃圾收集机制就会被触发，把那些可以被回收的对象回收掉，而那些不会回收的对象就会被移到中年代去，依此类推，老年代中的对象是存活时间最久的对象，甚至是存活于整个系统的生命周期内。同时，分代回收是建立在标记清除技术基础之上。

分代回收同样作为 Python 的辅助垃圾收集技术处理容器对象

生产者消费者模型

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

有些模块负责生产数据，这些数据由其他模块来负责处理(此处的模块可能是：函数、线程、进程等)。产生数据的模块称为生产者，而处理数据的模块称为消费者。在生产者与消费者之间的缓冲区称之为仓库。

可以说生产者负责往仓库运输商品，而消费者负责从仓库里取出商品，这就构成了生产者消费者模式。



此模型经常会在实际生产中遇到，有以下优点：

- 解耦
- 并发
- 支持忙闲不均

当生产者制造数据快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中，慢慢处理掉。而不至于因为消费者的性能造成数据丢失或影响生产者生产。

生产者消费者模型在 Python 中一般有 2 种实现：

- 多线程和队列
- 生成器 yield

GitChat 用户专享，请尊重版权

多线程和队列实现

以经典的生产包子为例

```
# content of queue_test.py
from threading import Thread
from time import sleep
from queue import Queue

class Producer(Thread):
    def __init__(self, worker, queue):
        super().__init__()
        self._worker = worker
        self._queue = queue

    def run(self):
        while True:
            if 0 <= self._queue.qsize() <= 10:
                queue.put('baozi')
                print('{} 生产了1个包子，一共{}个包子'.format(self._worker,
                                                               self._queue.qsize()))
                sleep(0.5)
            elif 10 < self._queue.qsize() <= 20:
                queue.put('baozi')
                print('{} 生产了1个包子，一共{}个包子'.format(self._worker, self._queue.qsize()))
                sleep(1)
            else:
                print('仓库较多，生产者休息3秒钟。')
                sleep(3)
```

上面是生产者代码，Queue 为队列，作为仓库角色。接下来看消费者代码

```
# content of queue_test.py
class Consumer(Thread):
    def __init__(self, client, queue):
        super().__init__()
        self._client = client
        self._queue = queue

    def run(self):
        while True:
            if self._queue.empty():
                print('仓库没有包子了。。。')
                sleep(0.5)
            else:
                result = self._queue.get()
```

GitChat 用户专享，请尊重版权

```
    print('{} 消费了1个包子，还剩{}个包子'.format(self._client, self._queue.qsize()))
    sleep(0.5)

queue = Queue(maxsize=20)

for item in ['LiBao', 'YangBao']:
    temp = Producer(item, queue)
    temp.start()

for item in ['ChengBaoConsumer', 'TianBaoConsumer']:
    temp = Consumer(item, queue)
    temp.start()
```

上面是消费者相关代码

运行输出结果

```
LiBao 生产了1个包子，一共1个包子
YangBao 生产了1个包子，一共2个包子
ChengBaoConsumer 消费了1个包子，还剩1个包子
TianBaoConsumer 消费了1个包子，还剩0个包子
LiBao 生产了1个包子，一共1个包子
YangBao 生产了1个包子，一共2个包子
TianBaoConsumer 消费了1个包子，还剩1个包子
ChengBaoConsumer 消费了1个包子，还剩0个包子
YangBao 生产了1个包子，一共1个包子
ChengBaoConsumer 消费了1个包子，还剩0个包子
仓库没有包子了。。。
```

5.2 yield 实现

yield 实现主要是利用生成器的 send 和 next 方法， send 发送信息， next 消费信息。

```
import time

def consumer(name):
    print('{}准备吃包子了！'.format(name))
    while True:
        baozi = yield #在它就收到内容的时候后就把内容传给baozi
        print('包子【{}】来了，被【{}】吃了'.format(baozi, name))

def producer():
    c1 = consumer('A') #它只是把c1变成一个生成器
    c2 = consumer('B')
    c1.__next__() #第一个next只是会走到yield然后停止
    c2.__next__()
```

```
print('开始做包子了')
for i in range(1,10):
    time.sleep(0.5)
    print('三秒做了两个包子')
    c1.send(i)
    c2.send(i+1)

producer()
```

运行输出结果

```
A准备吃包子了!
B准备吃包子了!
开始做包子了
三秒做了两个包子
包子【1】来了，被【A】吃了
包子【2】来了，被【B】吃了
```

异步编程模型

6.1 相关概念

6.1.1 进程 vs 线程

- **进程**

进程是表示资源分配的基本单位，又是调度运行的基本单位。程序并不能单独运行，只有将程序装载到内存中，系统为它分配资源才能运行，而这种执行的程序就称之为进程。程序和进程的区别就在于：程序是指令的集合，它是进程运行的静态描述文本；进程是程序的一次执行活动，属于动态概念。

- **线程**

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

6.1.2 同步 vs 异步

- **同步**

不同程序单元为了完成某个任务，在执行过程中需靠某种通信方式以协调一致，称这些程序单元是同步执行的。同步意味着有序，按顺序执行。

- **异步**

为完成某个任务，不同程序单元之间过程中无需通信协调，即多个任务之间没有先后顺序。

6.1.3 阻塞 vs 非阻塞

- 阻塞

阻塞是当请求不能满足的时候就将进程挂起，使调用者不能继续往下执行。

- 非阻塞

非阻塞则不会阻塞当前进程，可以继续执行，就是说非阻塞的。

6.1.3 并行 vs 并发

- 并发

并发描述的是程序的组织结构。指程序要被设计成多个可独立执行的子任务。以利用有限的计算机资源使多个任务可以被实时或近实时执行为目的。

- 并行

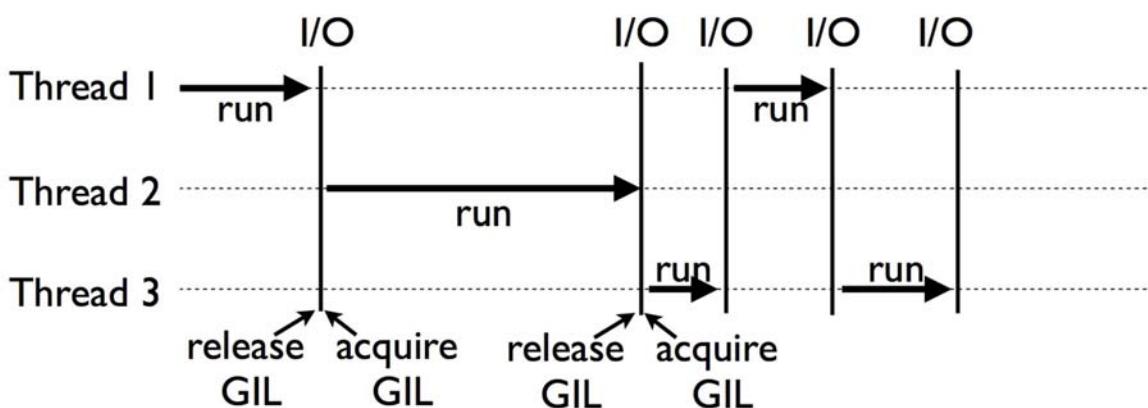
并行描述的是程序的执行状态。指多个任务同时被执行。以利用富余计算资源（多核 CPU）加速完成多个任务为目的。

6.2 全局解释器锁 (GIL)

全局解释器锁 (GIL) 表示在同一时刻只有一个线程对共享资源进行存取。

GIL 并不是 Python 的特性，Python 完全可以不依赖于 GIL。

先明确的一点是 GIL 并不是 Python 的特性，它是在实现 Python 解析器(CPython)时所引入的一个概念。就好比 C++ 是一套语言（语法）标准，但是可以用不同的编译器来编译成可执行代码。Python 也一样，同样一段代码可以通过 CPython、PyPy、Psyco 等不同的 Python 执行环境来执行。像其中的 JPython 就没有 GIL。然而因为 CPython 是大部分环境下默认的 Python 执行环境。所以在很多人的概念里 CPython 就是 Python，也就想当然的把 GIL 归结为 Python 语言的缺陷。



线程何时切换？一个线程无论何时开始睡眠或等待网络 I/O，其他线程总有机会获取 GIL 执行 Python 代码。这是协同式多任务处理。CPython 也还有抢占式多任务处理。如果一个线程不间断地在 Python 2 中运行 1000 字节码指令，或者不间断地在 Python 3 运行 15 毫秒，那么它便会放弃 GIL，而其他线程可以运行。

GitChat 用户专享，请尊重版权

Python 的多线程在多核 CPU 上，只对于 IO 密集型计算产生正面效果；而当有至少有一个 CPU 密集型线程存在，那么多线程效率会由于 GIL 而大幅下降。

如何避免 GIL 对性能的影响：

- 多进程
- 使用别的解析器，比如 Jpython

6.3 多线程

多线程在 Python3 中 2 种实现方式：

- Threading 模块
- concurrent.futures

下面举几个小例子：

```
# 直接调用
import threading
import time

def sayhi(num): #定义每个线程要运行的函数
    print("running on number:%s" %num)
    time.sleep(3)

if __name__ == '__main__':
    t1 = threading.Thread(target=sayhi,args=(1,)) #生成一个线程实例
    t2 = threading.Thread(target=sayhi,args=(2,)) #生成另一个线程实例
    t1.start() #启动线程
    t2.start() #启动另一个线程

    print(t1.getName()) #获取线程名
    print(t2.getName())
```

```
# 继承调用
import threading
import time

class MyThread(threading.Thread):
    def __init__(self,num):
        threading.Thread.__init__(self)
        self.num = num

    def run(self):#定义每个线程要运行的函数
        print("running on number:%s" %self.num)
        time.sleep(3)
```

GitChat 用户专享，请尊重版权

```
if __name__ == '__main__':
    t1 = MyThread(1)
    t2 = MyThread(2)
    t1.start()
    t2.start()

# 线程池
from concurrent.futures import ThreadPoolExecutor
import time

def return_future_result(message):
    time.sleep(0.5)
    return message

pool = ThreadPoolExecutor(max_workers=2) # 创建一个最大可容纳2个task的线程池
future1 = pool.submit(return_future_result, ("hello")) # 往线程池里面加入一个task
future2 = pool.submit(return_future_result, ("world")) # 往线程池里面加入一个task
print(future1.done()) # 判断task1是否结束
time.sleep(1)
print(future2.done()) # 判断task2是否结束
print(future1.result()) # 查看task1返回的结果
print(future2.result()) # 查看task2返回的结果
```

关于多线程还有一些知识点，由于篇幅问题，就不展开了，主要是线程锁、信号量、定时器。线程间的通信可以使用 Queue 实现。

6.4 多进程

多进程在 Python3 中 2 种实现方式：

- Threading 模块
- concurrent.futures

下面举几个例子

```
from multiprocessing import Process
import time
def f(name):
    time.sleep(2)
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
```

GitChat 用户专享，请尊重版权

```
p.start()  
p.join()
```

为了更加清楚的展示父进程和子进程的 ID，下面再来一个例子

```
from multiprocessing import Process  
import os  
  
def info(title):  
    print(title)  
    print('module name:', __name__)  
    print('parent process:', os.getppid())  
    print('process id:', os.getpid())  
    print("\n\n")  
  
def f(name):  
    info('\033[31;1mfunction f\033[0m')  
    print('hello', name)  
  
if __name__ == '__main__':  
    info('\033[32;1mmain process line\033[0m')  
    p = Process(target=f, args=('bob',))  
    p.start()  
    p.join()
```

进程池

```
from concurrent.futures import ProcessPoolExecutor  
import os, time, random  
def task(n):  
    print('%s is running' % os.getpid())  
    time.sleep(0.2)  
    return n**2  
  
if __name__ == '__main__':  
    p=ProcessPoolExecutor() #不填则默认为cpu的个数  
    l=[]  
    start=time.time()  
    start = time.time()  
    with ProcessPoolExecutor() as p:  
        future_tasks = [p.submit(task, i) for i in range(10)]  
    print('=' * 30)  
    print([obj.result() for obj in future_tasks])  
    print(time.time() - start)
```

运行上面代码，输出结果

```
57058 is running
57059 is running
57060 is running
57061 is running
57062 is running
57063 is running
57064 is running
57065 is running
57058 is running
57060 is running

=====
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
0.4237039089202881
```

进程间的通信可以通过 Queue 和管道实现，这里不详细展开。实际生产中用进程池的场景会比较多。

6.5 协程

协程，又称微线程。协程是一种用户态的轻量级线程。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

优点：

- 无需线程上下文切换的开销
 - 无需原子操作锁定及同步的开销
- “原子操作(atomic operation)是不需要 synchronized”，所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch（切换到另一个线程）。原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序是不可以被打乱，或者切割掉只执行部分。视作整体是原子性的核心。
- 方便切换控制流，简化编程模型
 - 高并发+高扩展性+低成本：一个 CPU 支持上万的协程都不是问题。所以很适合用于高并发处理。

缺点：

- 无法利用多核资源：协程的本质是个单线程，它不能同时将单个 CPU 的多个核用上，协程需要和进程配合才能运行在多 CPU 上。
- 进行阻塞（Blocking）操作（如 IO 时）会阻塞掉整个程序。

特性：

- 必须在只有一个单线程里实现并发
- 修改共享数据不需加锁
- 用户程序里自己保存多个控制流的上下文栈
- 一个协程遇到 IO 操作自动切换到其它协程

在 Python3 中 asyncio 模块没有出来之前，协程可以通过 yield、gevent 来实现，asyncio 模块出来之后，基本都是用 asyncio 来开发，下一节会仔细讲解。

```
import gevent

def func1():
    print("我在玩游戏...")
    gevent.sleep(2)
    print("切换回去继续玩游戏...")

def func2():
    print('我要去上厕所。。.')
    gevent.sleep(1)
    print('上完厕所了。。.')

gevent.joinall([
    gevent.spawn(func1),
    gevent.spawn(func2),
])
```

运行上面代码，打印结果

```
我在玩游戏.....
我要去上厕所.....
上完厕所了.....
切换回去继续玩游戏.....
```

I/O 多路复用模型 (Reactor)

7.1 什么是 I/O 多路复用模型

I/O 多路复用技术是为实现单线程处理多请求连接，减少系统因频繁的创建线程或进程而产生的资源消耗，这里的复用特指同时使用单一线程。

linux 下的 select、poll、epoll 为 I/O 多路复用的具体实现。

GitChat 用户专享，请尊重版权

当客户端与服务端的 socket 连接建立之后，程序将该 socket 文件描述符注册到 epoll，然后返回，最终交由 epoll 去管理。epoll 可以同时监听多个文件描述符，当某个或某些文件描述符就绪，则通知程序进行相应的读写操作，否则会一直阻塞知道有文件描述符就绪。我们使用 epoll 编程时，会设置 socket 非阻塞模式。

7.1.1 select

select 目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

缺点

- 每次调用 select 都需要把文件描述符 (FD) 从用户态拷贝到内核，开销比较大。
- 每次都需要在内核遍历传入的文件描述符 (FD)。
- select 支持文件数量比较小，默认是 1024。当然，也可以通过修改宏定义改掉，但这会造成效率的降低。

7.1.2 poll

poll 及轮训，poll 和 select 本质上是一样的，只是描述 fd 集合的方式不同。poll 使用的是 pollfd 结构，select 使用的是 fd_set 结构。

poll 和 select 同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

7.1.3 epoll

epoll 是对 select 和 poll 的改进，而且改正了 select、poll 的三个缺点和不足。

相对于 select 和 poll 来说，epoll 更加灵活，没有描述符限制。epoll 使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的 copy 只需一次。

优点

- 每次注册新事件到 epoll 句柄都会把所有的 fd 拷贝进来，而不是在 epoll_wait 中重複拷贝，这样确保 fd 只会被拷贝一次。
- epoll 不是像 select/poll 那样每次都把 fd 加入等待队列，epoll 把每个 fd 指定一个回调函数，当设备就绪时，唤醒等待队列的等待者就会调用其它的回调函数，这个回调函数会把就绪的 fd 放入一个就绪链表。epoll_wait 就是在这个就绪链表中查看有没有就绪 fd。
- epoll 没有 fd 数目限制。

缺点

GitChat 用户专享，请尊重版权

- 如果没有大量的 idle-connection 或者 dead-connection，epoll 的效率并不会比 select/poll 高很多，但是当遇到大量的 idle-connection，就会发现 epoll 的效率大大高于 select/poll。

模式

- 水平触发 (level-triggered)：满足状态时触发

当被监控的文件描述符上有可读写事件发生时，`epoll_wait()` 会通知处理程序去读写。如果这次没有把数据一次性全部读写完（如读写缓冲区太小），那么下次调用 `epoll_wait()` 时，它还会通知你在上次没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你。如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率。

- 边缘触发 (edge-triggered)：状态改变时触发

当被监控的文件描述符上有可读写事件发生时，`epoll_wait()` 会通知处理程序去读写。如果这次没有把数据全部读写完（如读写缓冲区太小），那么下次调用 `epoll_wait()` 时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你去读写余下的数据。这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符。

例如一个 socket 经过长时间等待后接收到一段 100k 的数据，两种触发方式都会向程序发出就绪通知。假设程序从这个 socket 中读取了 50k 数据，并再次调用监听函数，水平触发依然会发出就绪通知，而边缘触发会因为 socket“有数据可读”这个状态没有发生变化而不发出通知且陷入长时间的等待。

7.2 select 实战编程

接下来实战编写一个单线程实现处理多个非阻塞 socket 连接。

因代码量较大，我分段解释，先解释 server 端代码。

7.2.1 Server 端代码

```
# content of socket_server.py part 1
import select
import socket
import sys
import queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(False)

# Bind the socket to the port
server_address = ('localhost', 9000)
print(sys.stderr, 'starting up on %s port %s' % server_address)
```

GitChat 用户专享，请尊重版权

```
server.bind(server_address)

# Listen for incoming connections
server.listen(5)

# Sockets from which we expect to read
inputs = [server]
# Sockets to which we expect to write
outputs = []

message_queues = {}
```

第一部分，`select()` 方法接收并监控3个通信列表，第一个是所有的输入的 data，就是指外部发过来的数据，第 2 个是监控和接收所有要发出去的 data(outgoing data)，第 3 个监控错误信息，接下来我们需要创建 2 个列表来包含输入和输出信息来传给 `select()`。

这里创建 `inputs` 接收外部数据，监控错误信息这里和 `inputs` 共用，`outputs` 为需要发出去的数据。

所有客户端的进来的连接和数据将会被 server 的主循环程序放在上面的list中处理，我们现在的 server 端需要等待连接可写 (writable) 之后才能过来，然后接收数据并返回（因此不是在接收到数据之后就立刻返回），因为每个连接要把输入或输出的数据先缓存到 queue 里，然后再由 `select` 取出来再发出去。

下面开始主循环部分代码

```
# content of socket_server.py part 2
while True:
    # Wait for at least one of the sockets to be ready for
    # processing
    print('\nwaiting for the next event')
    readable, writable, exceptional = select.select(inputs,
                                                    outputs, inputs)
    # Handle inputs
    for s in readable:
        if s is server:
            # A "readable" server socket is ready to accept a
            # connection
            connection, client_address = s.accept()
            print('new connection from', client_address)
            connection.setblocking(False)
            inputs.append(connection)

            # Give the connection a queue for data we want to
            # send
            message_queues[connection] = queue.Queue()
        else:
            data = s.recv(1024)
            if data:
```

GitChat 用户专享，请尊重版权

```
# A readable client socket has data
print(sys.stderr, 'received "{}" from {}'.format(
    data, s.getpeername()))
message_queues[s].put(data)
# Add output channel for response
if s not in outputs:
    outputs.append(s)
else:
    # Interpret empty result as closed connection
    print('closing', client_address, 'after reading
no data')
    # Stop listening for input on the connection
    if s in outputs:
        outputs.remove(s)
        #既然客户端都断开了，我就不用再给它返回数据了，所以这
时候如果这个客户端的连接对象还在outputs列表中，就把它删掉
        inputs.remove(s) #inputs中也删除掉
        s.close() #把这个连接关闭掉
        # Remove message queue
        del message_queues[s]
```

这里有几个逻辑，下面解释一下：

- 把 inputs,outputs,exceptional (这里跟 inputs 共用) 传给 select() 后，它返回 3 个新的 list，我们上面将他们分别赋值为 readable、writable、exceptional，所有在 readable list 中的 socket 连接代表有数据可接收 (recv)，所有在 writable list 中的存放着你可以对其进行发送(send)操作的 socket 连接，当连接通信出现 error 时会把 error 写到 exceptional 列表中。
- 先处理 readable 的 socket。有 3 种可能状态。
 - 第一种是如果这个 socket 是 main “server” socket，它负责监听客户端的连接，如果这个 main server socket 出现在 readable 里，那代表这是 server 端已经 ready 来接收一个新的连接进来了，为了让这个 main server 能同时处理多个连接，在上面的代码里，我们把这个 main server 的 socket 设置为非阻塞模式。
 - 第二种情况是这个 socket 是已经建立了的连接，它把数据发了过来，这个时候你就可以通过 recv() 来接收它发过来的数据，然后把接收到的数据放到 queue 里，这样你就可以把接收到的数据再传回给客户端了。
 - 第三种情况就是这个客户端已经断开了，所以你再通过 recv() 接收到的数据就为空了，所以这个时候你就可以把这个跟客户端的连接关闭了。

处理完 Readable list 的数据，接下来处理 Writeable 中的 socket 和异常信息

```
# content of socket_server.py part 3
# 还在主循环 while True 里
```

GitChat 用户专享，请尊重版权

```
# Handle outputs
for s in writable:
    try:
        next_msg = message_queues[s].get_nowait()
    except queue.Empty:
        # No messages waiting so stop checking for
        writability.
        print('output queue for', s.getpeername(), 'is
empty')
        outputs.remove(s)
    else:
        print('sending "%s" to %s' % (next_msg,
s.getpeername()))
        s.send(next_msg)
    # Handle "exceptional conditions"
    for s in exceptional:
        print('handling exceptional condition for',
s.getpeername())
        # Stop listening for input on the connection
        inputs.remove(s)
        if s in outputs:
            outputs.remove(s)
            s.close()

    # Remove message queue
del message_queues[s]
```

对于 writable list 中的 socket，也有几种状态，如果这个客户端连接在跟它对应的 queue 里有数据，就把这个数据取出来再发回给这个客户端，否则就把这个连接从 output list 中移除，这样下一次循环 select() 调用时检测到 outputs list 中没有这个连接，那就会认为这个连接还处于非活动状态。

最后，如果在跟某个 socket 连接通信过程中出了错误，就把这个连接对象在 inputs\outputs\message_queue 中都删除，再把连接关闭掉。

7.2.2 Client 端代码

客户端代码就相对简单点。

```
# content of socket_client.py
import socket
import sys

messages = [
    'This is the message. ',
    'It will be sent ',
    'in parts.',
]
server_address = ('localhost', 9000)
```

GitChat 用户专享，请尊重版权

```
# Create a TCP/IP socket
socks = [
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
]

# Connect the socket to the port where the server is listening
print('connecting to {} port {}'.format(server_address[0],
server_address[1]))
for s in socks:
    s.connect(server_address)

for message in messages:
    # Send messages on both sockets
    for s in socks:
        print(sys.stderr, '{}: sending "{}"'.format(s.getsockname(), message))
        s.send(message.encode("utf8"))

    # Read responses on both sockets
    for s in socks:
        data = s.recv(1024)
        print(sys.stderr, '{}: received "{}"'.format(s.getsockname(), data))
        if not data:
            print('closing socket', s.getsockname())
            s.close()
```

客户端展示了如何通过 select() 对 socket 进行管理并与多个连接同时进行交互，通过循环通过每个 socket 连接给 server 发送和接收数据。

打开两个终端，先执行 python socket_server.py，后执行 python socket_client.py 输出结果如下：

```
# content of socket_client cmd
connecting to localhost port 9000
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57700): sending "This is the message. "
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57701): sending "This is the message. "
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57700): received "b'This is the message. ''"
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57701): received "b'This is the message. ''"
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57700): sending "It will be sent "
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
('127.0.0.1', 57701): sending "It will be sent "
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
```

GitChat 用户专享，请尊重版权

```
('127.0.0.1', 57700): received "b'It will be sent '"  
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>  
('127.0.0.1', 57701): received "b'It will be sent '"  
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>  
('127.0.0.1', 57700): sending "in parts."  
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>  
('127.0.0.1', 57701): sending "in parts."  
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>  
('127.0.0.1', 57700): received "b'in parts.'"  
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>  
('127.0.0.1', 57701): received "b'in parts.'"
```

从打印结果看，完美的实现了一个单线程处理多个非阻塞 socket 连接进行接收发送数据。

理解 select, epoll 就相对容易了，这部分代码作为思考题大家之后自己扩展。

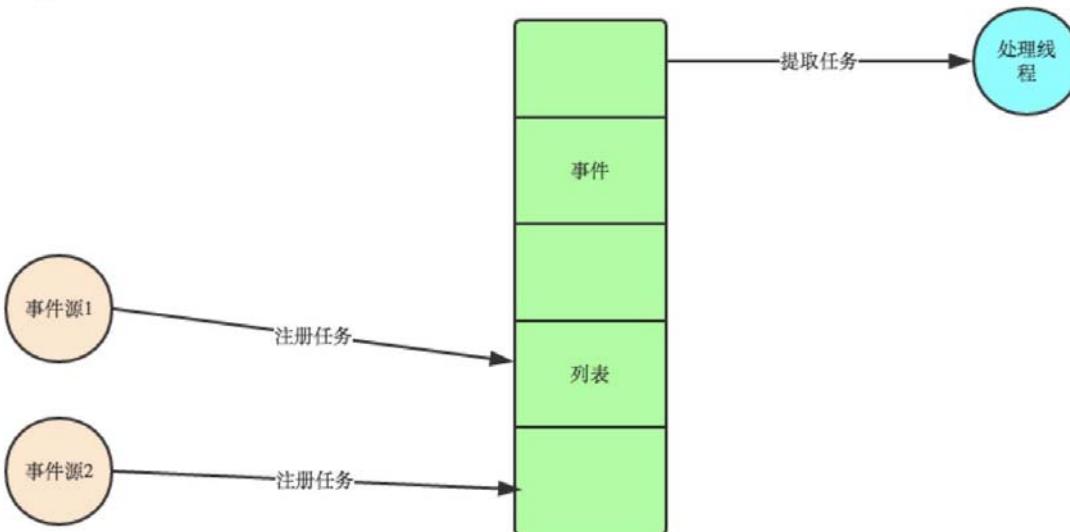
asyncio 并发编程

下面会详细讲解其原理和事件驱动模型，一些细节上的如何使用，请查阅官方文档，这里由于篇幅问题，不会细讲如何使用，只会描述核心原理。

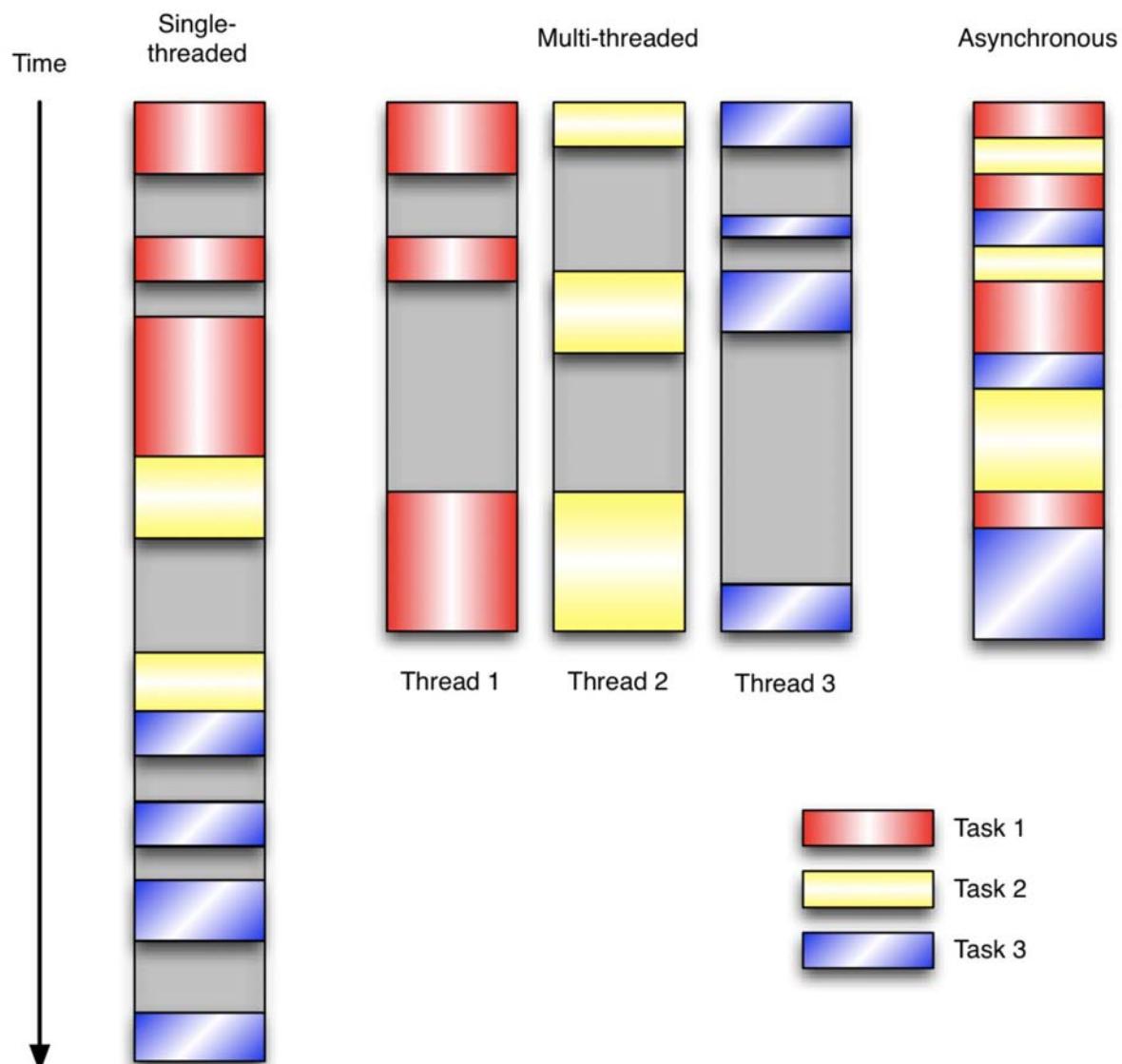
8.1 事件驱动模型

事件驱动模型是什么了，下面一张图展示了，简单的说，有以下几条思路（UI 编程很多都是事件驱动）：

- 有一个事件（消息）队列；
- 鼠标按下时，往这个队列中增加一个点击事件（消息）；
- 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如 onClick()、onKeyDown() 等；
- 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数。



下面图为多线程、单线程、事件驱动模型的性能对比：



其实事件驱动模型还有另外一个名字，可以理解为就是 I/O 多路复用模型，这个第 7 节已经讲解了，所以事件驱动模型就是通过 I/O 多路复用原理实现的。

GitChat 用户专享，请尊重版权

而 Python3 中的黑魔法协程 asyncio 也是得意于此。

Python3.4 之后，asyncio 模块算是比较火了，同时也引入了 await/async 关键字实现协程。由于 GIL 的存在，使得协程成为 Python 并发编程的最佳模型。

8.2 yield from - python 3.4

下面 Python 3.4 的代码分别以异步和并发的函数调用实现按秒倒计时。

```
# python 3.4
import asyncio

@asyncio.coroutine
def countdown(number, n):
    while n > 0:
        print('T-minus', n, '( {})'.format(number))
        yield from asyncio.sleep(1)
        n -= 1

loop = asyncio.get_event_loop()
tasks = [
    asyncio.ensure_future(countdown("A", 2)),
    asyncio.ensure_future(countdown("B", 3))]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

Python 3.4 中，`asyncio.coroutine` 修饰器用来标记作为协程的函数，这里的协程是和 `asyncio` 及其事件循环一起使用的。`asyncio` 要求所有要用作协程的生成器必须由 `asyncio.coroutine` 修饰。

然后可以对任何 `asyncio.Future` 对象使用 `yield from`，从而将其传递给事件循环，暂停协程的执行来等待某些事情的发生（`future` 对象并不重要，只是 `asyncio` 细节的实现）。一旦 `future` 对象获取了事件循环，它会一直在那里监听，直到完成它需要做的一切。当 `future` 完成自己的任务之后，事件循环会察觉到，暂停并等待在那里的协程会通过 `send()` 方法获取 `future` 对象的返回值并开始继续执行。

运行此代码，需要开启事件循环，有两种方法，一种方法就是通过调用 `run_until_complete`，另外一种就是调用 `run_forever`。`run_until_complete` 内置 `add_done_callback`，使用 `run_forever` 的好处是可以通过自己自定义 `add_done_callback`。

以上面代码为例，再次描述其过程：

事件循环启动每一个 `countdown()` 协程，一直执行到遇见其中一个协程的 `yield from` 和 `asyncio.sleep()`。这样会返回一个 `asyncio.Future` 对象并将其传递给事件循环，同时暂停这一协程的执行。事件循环会监控这一 `future` 对象，直到倒计时 1 秒钟之后（同时也会检查其它正在监控的对象，比如像其它协程）。1 秒钟的时间一到，事件循环会选择刚

GitChat 用户专享，请尊重版权

刚传递了 future 对象并暂停了的 countdown() 协程，将 future 对象的结果返回给协程，然后协程可以继续执行。这一过程会一直持续到所有的 countdown() 协程执行完毕，事件循环也被清空。

8.3 async/await - python 3.5+

在 Python3.5+ 后，yield from 变成了 await，同时添加了 types.coroutine 修饰器，也可以像 asyncio.coroutine 一样将生成器标记为协程。之后可以用 async def 来定义一个协程函数，但是这个函数不能包含任何形式的 yield 语句；只有 return 和 await 可以从协程中返回值。

```
# python3.5
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    print("print sum starting...")
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

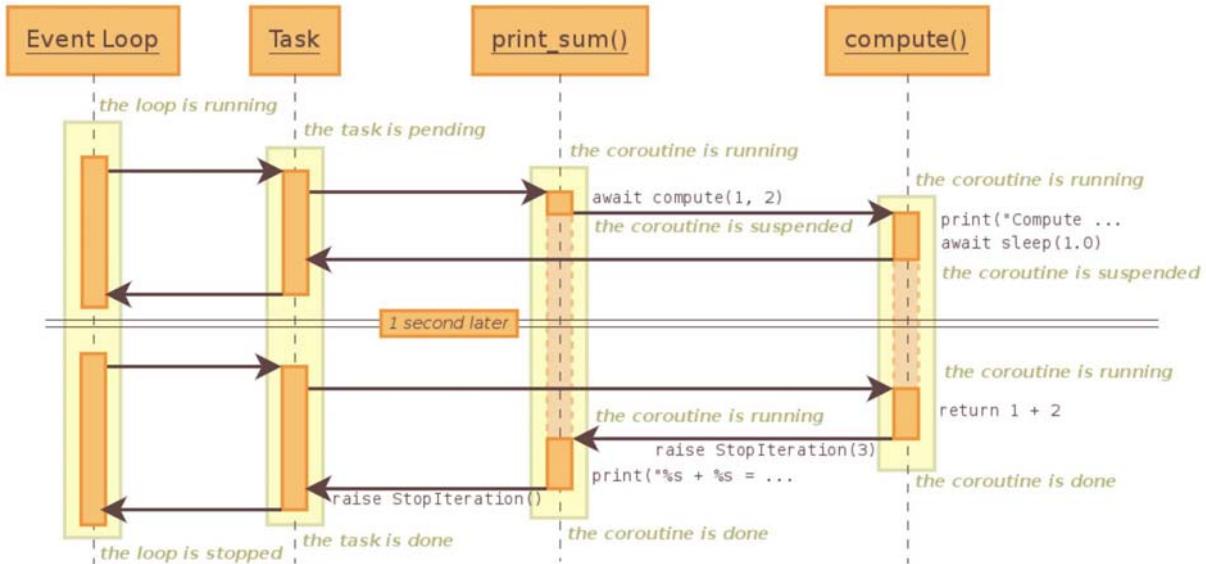
loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

运行上面代码，输出结果

```
print sum starting...
Compute 1 + 2 ...
1 + 2 = 3
```

GitChat 用户专享，请尊重版权

执行流程：



图中可以看到，`asyncio`自带的事件循环负责所有协程的调度。在首先执行 `print_sum` 协程时，内部使用 `await` 等待另外一个协程 `compute` 的返回结果，于是本地协程被挂起来执行协程 `compute`。而协程 `compute` 执行过程中调用了 `await asyncio.sleep(1.0)`，于是本地协程挂起 1 秒、将程序控制权交回事件循环，1 秒后再恢复协程 `compute` 的执行直到整个 stack 执行结束。

8.4 yield from vs await

来看看 `yield from` 和 `await` 的区别

虽然 `await` 的使用和 `yield from` 很像，但 `await` 可以接受的对象却是不同的。`await` 可以接受协程，因为协程的概念是所有这一切的基础。但是当你使用 `await` 时，其接受的对象必须是 `awaitable` 对象：必须是定义了 `await()` 方法且这一方法必须返回一个不是协程的迭代器。协程本身也被认为是 `awaitable` 对象。

`yield from` 和 `await` 在底层的差别是什么？让我们看一下上面两则 Python 3.5 代码的例子所产生的字节码在本质上有何差异。`py34_coro()` 的字节码是：

```
>>> dis.dis(py34_coro)
  2           0  LOAD_GLOBAL               0  (stuff)
  3           3  CALL_FUNCTION            0  (0 positional, 0
keyword pair)
  6           6  GET_YIELD_FROM_ITER
  7           7  LOAD_CONST                0  (None)
 10          10  YIELD_FROM
 11          11  POP_TOP
 12          12  LOAD_CONST                0  (None)
 15          15  RETURN_VALUE
```

GitChat 用户专享，请尊重版权

`py35_coro()` 的字节码是:

```
>>> dis.dis(py35_coro)
  1           0 LOAD_GLOBAL              0 (stuff)
              3 CALL_FUNCTION         0 (0 positional, 0
keyword pair)
              6 GET_AWAITABLE
              7 LOAD_CONST               0 (None)
              10 YIELD_FROM
              11 POP_TOP
              12 LOAD_CONST               0 (None)
              15 RETURN_VALUE
```

两者之间唯一可见的差异是 `GET_YIELD_FROM_ITER` 操作码对比 `GET_AWAITABLE` 操作码。两个函数都被标记为协程，因此在这里没有差别。`GET_YIELD_FROM_ITER` 只是检查参数是生成器还是协程，否则将对其参数调用 `iter()` 方法。

但是 `GET_AWAITABLE` 的做法不同，其字节码像 `GET_YIELD_FROM_ITER` 一样接受协程，但是不接受没有被标记为协程的生成器。就像前面讨论过的一样，除了协程以外，这一字节码还可以接受 `awaitable` 对象。这使得 `yield from` 和 `await` 表达式都接受协程，但 `yield from` 还可以接受一般的生成器，而 `await` 只能接受 `awaitable` 对象。

为什么基于 `async` 的协程和基于生成器的协程会在对应的表达式上面有所不同？主要原因是出于最优化 Python 性能的考虑，确保你不会将刚好有同样 API 的不同对象混为一谈。由于生成器默认实现协程的 API，因此很有可能在你希望用协程的时候错用了一个生成器。而由于并不是所有的生成器都可以用在基于协程的控制流中，你需要避免错误地使用生成器。但是由于 Python 并不是静态编译的，它最好也只能在用基于生成器定义的协程时提供运行时检查。这意味着当用 `types.coroutine` 时，Python 的编译器将无法判断这个生成器是用作协程还是仅仅是普通的生成器，因此编译器只能基于当前的情况生成有着不同限制的操作码。

8.5 误区

看完上面那些，如果你觉得随便什么外部函数用 `await` 修饰后，就会变成非阻塞，那就会让你失望了。上面讲到 `await` 接受 `awaitable` 对象，而不是普通的生成器。如果你将 `await asyncio.sleep(1)` 换成 `sleep(1)` 看看有什么效果？

如果 `async def` 协程里有数据库操作或者 `request` 请求，都是会阻塞的，效果不会是你想象的一样。所以有一个 GitHub 仓库是你必须需要了解的。

<https://github.com/aio-libs>

- 异步的数据库操作: `aiomysql`
- 异步的请求: `aiohttp`
- 异步的 redis: `aioredis`
- 异步的 I/O 操作: `aiofiles`

GitChat 用户专享，请尊重版权

如果是确实用到非 awaitable 对象，又想用 asyncio。可以考虑多线程和多进程。

这个 asyncio 也是支持的。下面来看怎么使用：

```
# 线程池
from concurrent import futures

@asyncio.coroutine
def loop(self, loop):
    """
    function: loop
    """
    with futures.ThreadPoolExecutor(max_workers=8) as executor:
        loop = asyncio.get_event_loop()
    ...

# 进程池
from concurrent import futures

@asyncio.coroutine
def loop(self, loop, indexes=""):
    """
    function: loop
    """
    with futures.ProcessPoolExecutor(max_workers=8) as executor:
        loop = asyncio.get_event_loop()
    ...
```

利用 futures 模块提供的进程池和线程池可以用构建多进程和多线程，其余代码与单独使用 asyncio 无差别。

网络编程

9.1 TCP 协议

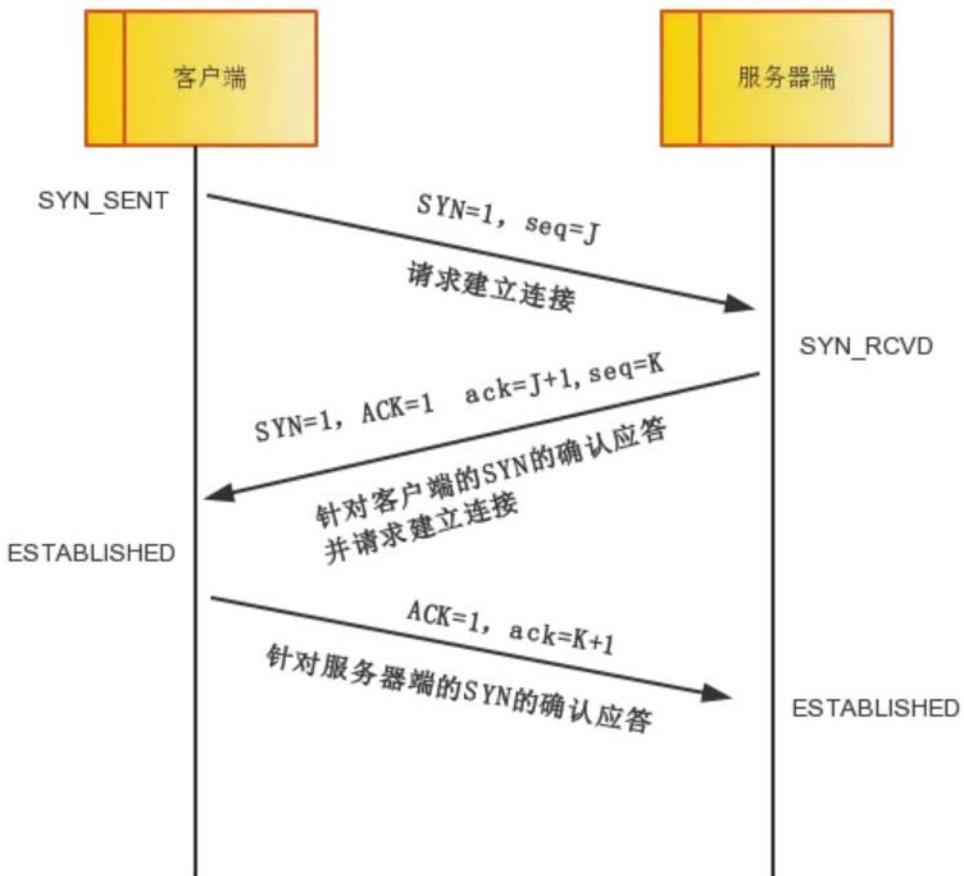
TCP 提供面向有连接的通信传输。面向有连接是指在数据通信开始之前先做好两端之间的准备工作。

面试过程中必问的知识点就是三次握手和四次挥手。

下面主要描述什么是三次握手和四次挥手：

GitChat 用户专享，请尊重版权

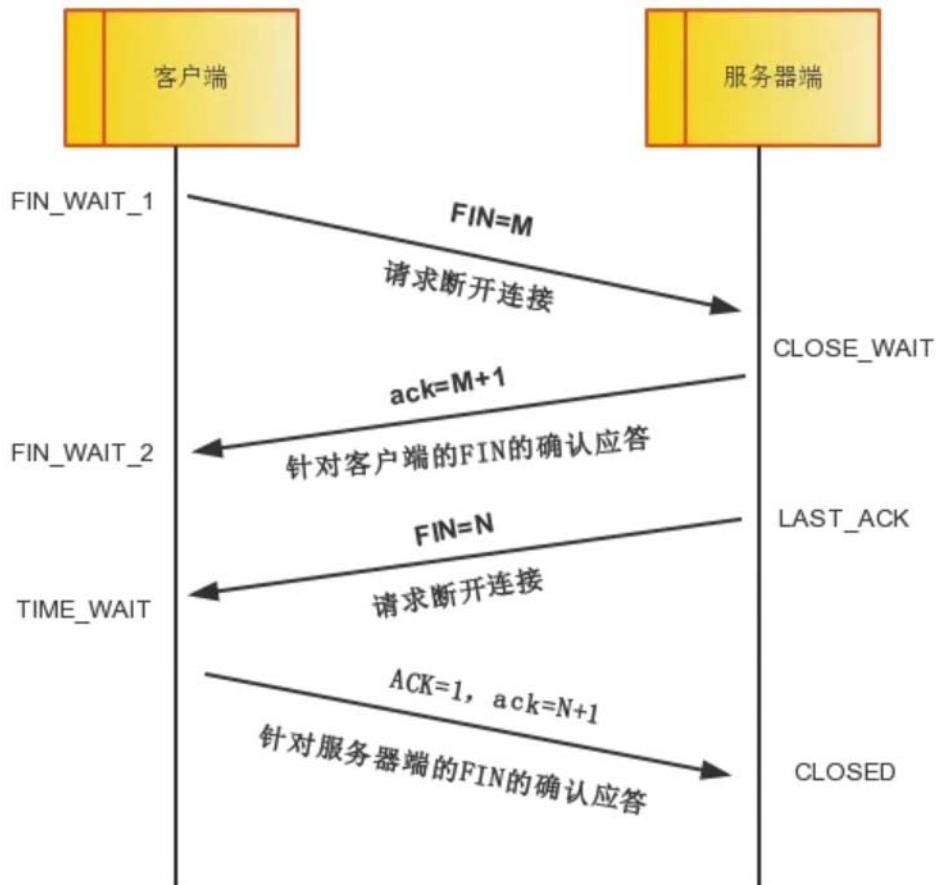
- 三次握手



- 第一次握手：客户端将标志位 SYN 置为 1，随机产生一个值 seq=J，并将该数据包发送给服务器端，客户端进入 SYN_SENT 状态，等待服务器端确认。
- 第二次握手：服务器端收到数据包后由标志位 SYN=1 知道客户端请求建立连接，服务器端将标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个值 seq=K，并将该数据包发送给客户端以确认连接请求，服务器端进入 SYN_RCVD 状态。
- 第三次握手：客户端收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1，并将该数据包发送给服务器端，服务器端检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，客户端和服务器端进入 ESTABLISHED 状态，完成三次握手，随后客户端与服务器端之间可以开始传输数据了。

GitChat 用户专享，请尊重版权

- 四次挥手



- 第一次挥手：客户端发送一个 $FIN=M$ ，用来关闭客户端到服务器端的数据传送，客户端进入 `FIN_WAIT_1` 状态。意思是说”我客户端没有数据要发给你了”，但是如果你服务器端还有数据没有发送完成，则不必急着关闭连接，可以继续发送数据。
- 第二次挥手：服务器端收到 FIN 后，先发送 $ack=M+1$ ，告诉客户端，你的请求我收到了，但是我还没准备好，请继续你等我的消息。这个时候客户端就进入 `FIN_WAIT_2` 状态，继续等待服务器端的 FIN 报文。
- 第三次挥手：当服务器端确定数据已发送完成，则向客户端发送 $FIN=N$ 报文，告诉客户端，好了，我这边数据发完了，准备好关闭连接了。服务器端进入 `LAST_ACK` 状态。
- 第四次挥手：客户端收到 $FIN=N$ 报文后，就知道可以关闭连接了，但是他还是不相信网络，怕服务器端不知道要关闭，所以发送 $ack=N+1$ 后进入 `TIME_WAIT` 状态，如果 Server 端没有收到 ACK 则可以重传。服务器端收到 ACK 后，就知道可以断开连接了。客户端等待了 $2MSL$ 后依然没有收到回复，则证明服务器端已正常关闭，那好，我客户端也可以关闭连接了。最终完成了四次握手。

一般面试过程中还是问为什么是三次握手，而挥手则需要四次？

GitChat 用户专享，请尊重版权

首先 TCP 的定位是全双工的、支持半关闭的、可靠的传输协议。三次握手是可以最低限度地确定双方的信息是双向可用的（全双工）。

假设是 A 向 B 发起请求。

第二次握手成功表明 $A \Rightarrow B$ 没问题。

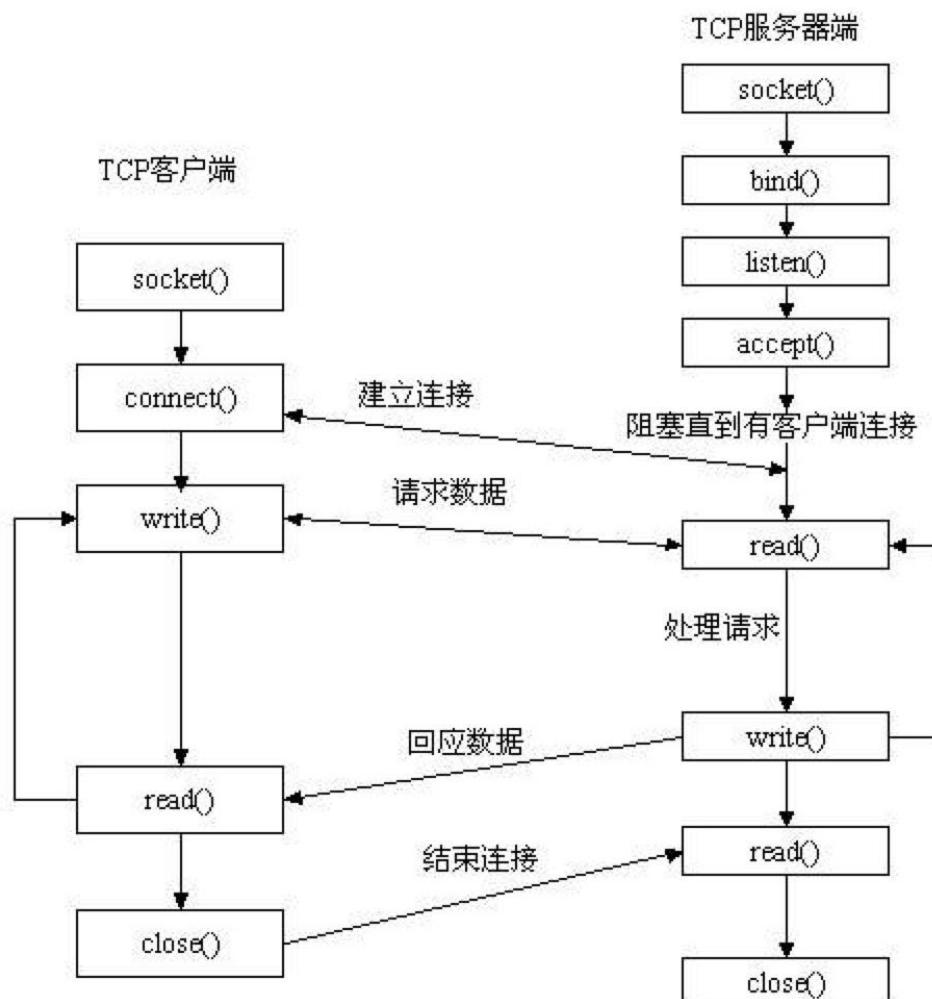
第三次握手成功表明 $B \Rightarrow A$ 没问题。

同时服务端在 LISTEN 状态下，收到建立连接请求的 SYN 报文后，把 ACK 和 SYN 放在一个报文里发送给客户端。

而四次挥手，TCP 要支持半关闭连接。建立的连接是全双工的， $A \Leftrightarrow B$ 双方都可以读写。支持半关闭意味着，TCP 支持 A 和 B 双方独立关闭通道。因此会有两次独立的关闭写通道的请求。一次关闭请求（FIN），对应一个 ACK。

9.2 编程实战

socket 编程，先来看看流程：



- 服务器流程

- 建立 socket
- 绑定 socket 的 ip 地址和端口
- listen 监听客户端的连接请求

GitChat 用户专享，请尊重版权

- accept 接受客户端的连接请求
 - read/write 与客户端进行数据对话
 - close 关闭连接
- 客户端流程
 - 创建 socket
 - connect 连接服务器 socket
 - write/read 与服务器进行对话
 - close 关闭连接

socket 编程模型有单线程，多线程，多进程，以及 Reactor(在第 7 节已经实现过)，接下来代码演示这几个模型。

9.2.1 单线程模型

```
# content of single_server.py
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('0.0.0.0', 8000))
server.listen()
sock, addr = server.accept()

while True:
    data = sock.recv(1024)
    print(data.decode('utf8'))
    data = "server: {}".format(data)
    sock.send(data.encode("utf8"))

# content of single_client.py
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('0.0.0.0', 8000))
while True:
    data = input()
    client.send(data.encode('utf8'))
    data = client.recv(1024)
    print(data.decode('utf8'))
```

分别运行客户端和服务端代码，输出结果为

```
→ py3 git:(master) ✘ python3 single_server.py
hello

→ py3 git:(master) ✘ python3 single_client.py
```

GitChat 用户专享，请尊重版权

```
hello
server: b'hello'
```

单线程同步的完全按照上面流程来执行，但是只能接受一个客户端，生产环境肯定是不能接受的。接下来看看多线程

9.2.2 多线程模型

客户端代码与上面一致，下面看看服务器代码

```
import socket
import threading

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("0.0.0.0", 8000))
server.listen(1)

def handle_sock(sock, addr):
    while True:
        data = sock.recv(1024).decode("utf8")
        print(data)
        if data == "q":
            break
        send_data = "server: {}".format(data)
        sock.send(send_data.encode("utf8"))
    sock.close()

while True:
    sock, addr = server.accept()
    thread_client = threading.Thread(target=handle_sock, args=[sock, addr])
    thread_client.start()
```

不同的区别是用了 `threading.Thread` 模块去启动一个线程处理客户端连接。

9.2.3 多进程模型

客户端代码与上面一致，下面看看服务器代码

```
import socket
import multiprocessing

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("0.0.0.0", 8002))
server.listen(1)

def handle_sock(sock, addr):
    while True:
```

GitChat 用户专享，请尊重版权

```
data = sock.recv(1024).decode("utf8")
print(data)
if data == "q":
    break
send_data = "server: {}".format(data)
sock.send(send_data.encode("utf8"))

sock.close()

while True:
    sock, addr = server.accept()
    thread_client = multiprocessing.Process(
        target=handle_sock, args=[sock, addr])
    thread_client.start()
```

代码跟多线程很像，区别在于把 Thread 模块换成了 Process 模块。

9.2.4 多进程 preforking 模型

采用 PreForking 模型可以对子进程的数量进行了限制。PreForking 是通过预先产生多个子进程，共同对服务器套接字进行竞争性的 accept，当一个连接到来时，每个子进程都有机会拿到这个连接，但是最终只会有一个进程能 accept 成功返回拿到连接。子进程拿到连接后，进程内部可以继续使用单线程或者多线程同步的形式对连接进行处理。

因为进程要比线程更加吃资源，如果来一个连接就开一个进程，当连接比较多时，进程数量也会跟着多起来，操作系统的调度压力也就会比较大。所以我们要对服务器开辟的进程数量进行限制，避免系统负载过重

客户端代码依然不变，看服务器代码

```
import os
import socket
import multiprocessing

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("0.0.0.0", 8002))
server.listen(100)

def handle_sock(sock, addr):
    while True:
        data = sock.recv(1024).decode("utf8")
        print(data)
        if data == "q":
            break
        send_data = "server: {}".format(data)
        sock.send(send_data.encode("utf8"))

    sock.close()

def loop(server):
    while True:
```

GitChat 用户专享，请尊重版权

```
sock, addr = server.accept()
handle_sock(sock, addr)

def prefork(n):
    for i in range(n):
        pid = os.fork()
        if pid < 0: # fork error
            return
        if pid > 0: # parent process
            continue
        if pid == 0:
            break # child process

prefork(10)
loop(server)
```

上面代码，会预先生产 10 个子进程。

9.2.5 Reactor 模型

请看第 7 章内容

这次分享就到这里，欢迎大家读者圈交流。

参考文章

1. <https://pyzh.readthedocs.io/en/latest/python-magic-methods-guide.html>
2. <https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/>
3. <https://foofish.net/python-gc.html>