高并发下场景下 Python的性能挑战

概要

- Python的性能问题
  - GIL
  - 解释器
  - 动态语言
- 服务选型
- 性能瓶颈分析
- 优化方法
- 稳定性建议

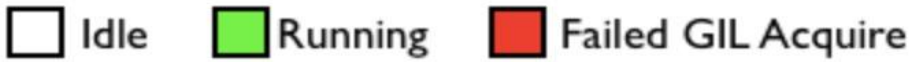
全局解释器锁(Global Interpreter Lock)

当CPython创建变量时，它会分配内存然后计算对该变量的引用数量，大家通常称之为“引用计数”。如果引用计数变为0，则从系统中释放该内存。引用计数变量时需要保护竞争条件，多个线程同时增加或减少变量引用计数时，可能导致内存泄漏或者错误的内存释放。

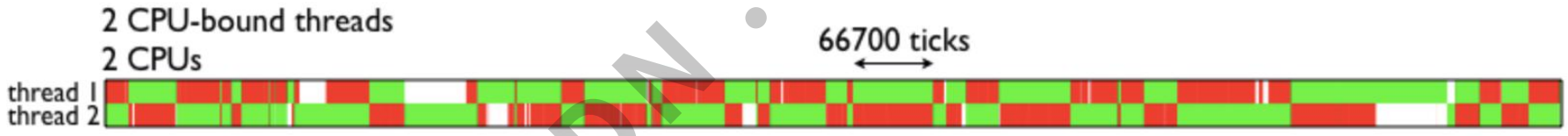
CPython引入了GIL，线程在执行代码时，必须首先获得解释器的使用权，虽然保证了数据安全，也意味着单进程下Python多线程的性能没有那么好。

[经典的测试结果]

状态说明



2CPU，2CPU-bound Threads



GIL的影响

2CPU, 1I/O-bound , 1CPU-bound

```
def thread_1(port):
    s = socket(AF_INET,SOCK_DGRAM)
    s.bind(("",port))
    while True:
        msg, addr = s.recvfrom(1024)
        s.sendto(msg,addr)
```

Python解释器

在执行一段py文件时，CPython解释器首先将py文件编译成pyc 字节码序列，并且缓存在本地中；大多数情况（除非刻意的删除掉pyc文件），Cpython在运行代码时都是在解释字节码序列。

预编译语言（C,C++）本身保证在执行时就是CPU可理解的代码，不过JAVA,.NET实际上也是即时翻译中间字节码，它们之所以比Python快的原因是因为使用了JIT(即时编译)，好的JIT可以检测哪些部分执行次数比较多，这些部分被称为“hot spots”。意味着当计算机应用程序需要重复做一件事情的时候，它就会更加地快。

pypy

动态类型语言

```
Python 2.7.10 (default, Aug 17 2018, 17:41:52)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> v = 1
>>> v = "v"
>>> v = {}
>>>
```

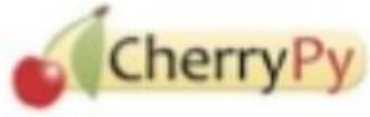
比较和转换类型是耗时的，因为每次读取、写入变量或引用变量类型时都会进行检查；

以灵活性换取了性能；静态类型语言没有这么高的灵活性；

像JIT编译器，可以通过预判数据类型对代码做更多的准备；

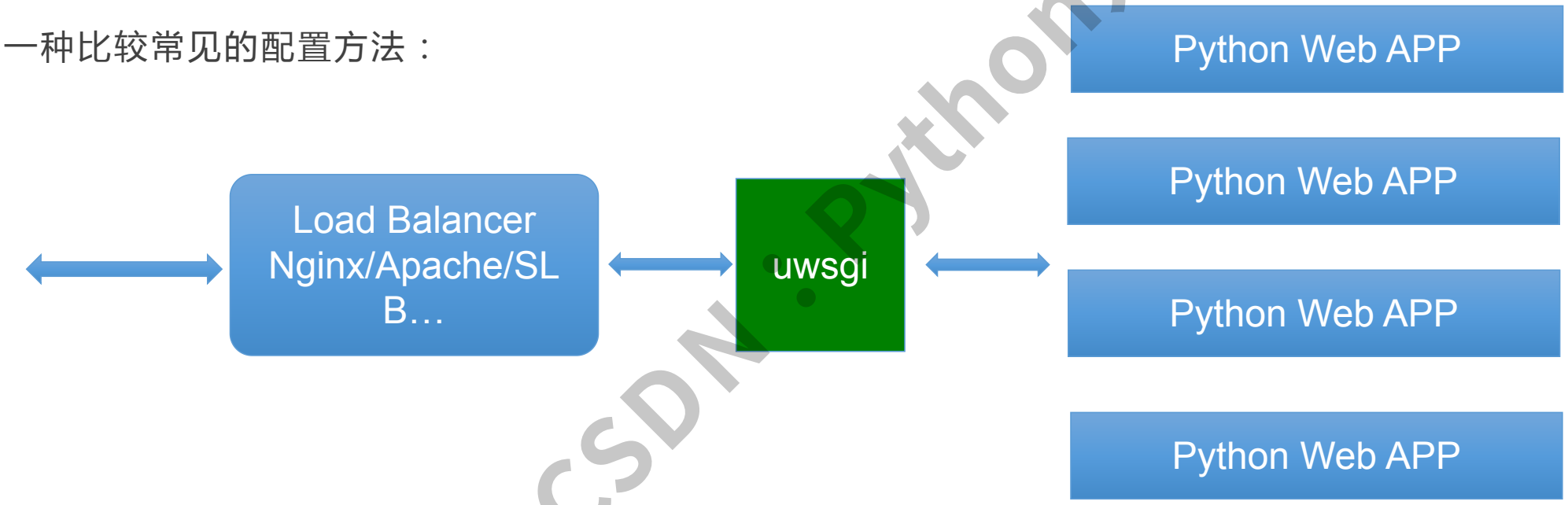
人生苦短，

我用Python



不论使用哪种web框架，我们都尽量要使用多进程；一方面是考虑到服务的可用性，另一方面是尽量减少GIL带来的性能影响；

一种比较常见的配置方法：



做好准备选择异步IO

Tornado

- 纯python；
- 封装程度不足，改造难度大；

Gevent

- 不是纯python，不支持pypy；
- 简单的monkey patch;

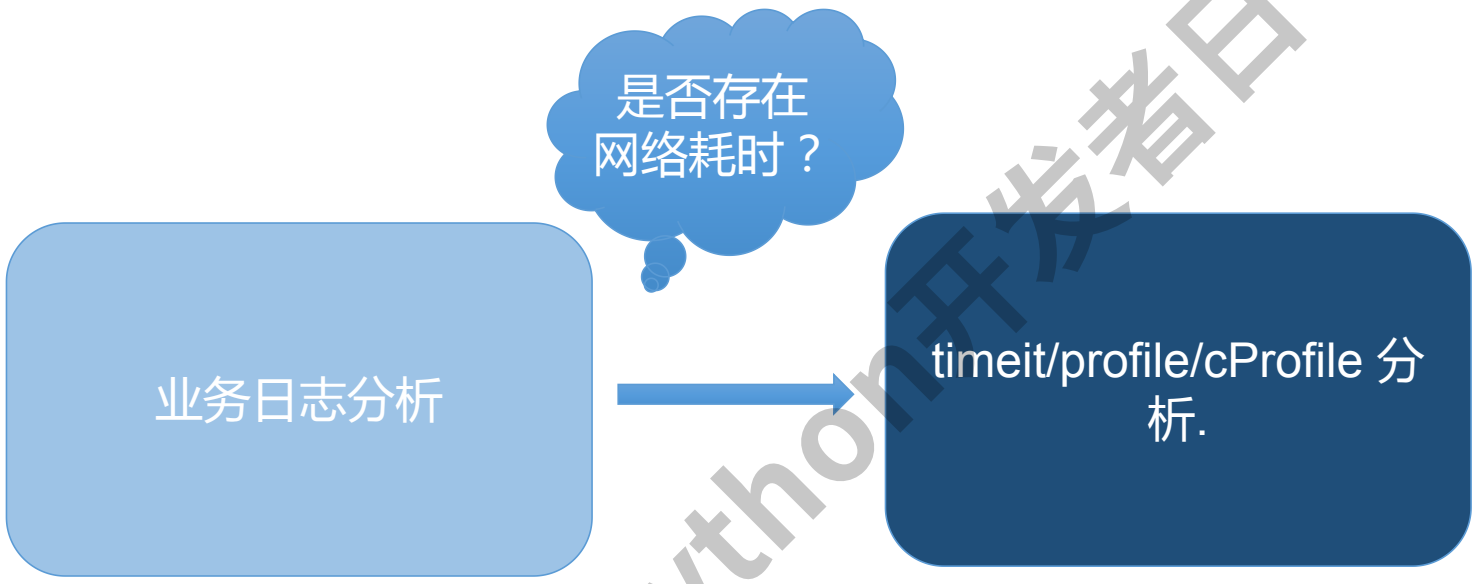
```
from gevent.monkey import patch_all
patch_all()
```

但是只支持封装Python Socket，所以在选型时要注意避开非纯Python的连接库；

Python的Just In Time 编译器，性能一般要比CPython解释器至少好3倍；

但是同样和其他JIT编译器一样有启动慢的缺点，适合对重启不是很敏感的服务；

性能瓶颈分析



即时服务选型正确，因为代码差异和持续的迭代，难免会遇到框架解决不了的问题；未避免后续的优化无迹可寻，我们的性能分析入口最好首先是根据业务日志耗时来判断，然后通过profile/cProfile工具进行某块代码性能分析；之后再寻找合适的优化方向；

性能瓶颈分析

profile/cProfile

test,py

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def func1():
    fib(10)


def func2():
    fib(15)


def func3():
    fib(30)

if __name__ == "__main__":
    func1()
    func2()
    func3()
```

python -m cProfile test.py

性能瓶颈分析

```
         2694692 function calls (8 primitive calls) in 0.710 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 test.py:11(func1)
        1    0.000    0.000    0.001    0.001 test.py:15(func2)
        1    0.000    0.000    0.709    0.709 test.py:19(func3)
        1    0.000    0.000    0.710    0.710 test.py:2(<module>)
2694687/3    0.710    0.000    0.710    0.237 test.py:2(fib)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

也可以输出pstat格式的数据：python -m cProfile –o test.pstat test.py，使用https://github.com/jrfonseca/gprof2dot可视化分析结果

优化方法 – 目录

- 原则
- IO-Bound
- CPU-Bound
- 缓存
- 懒加载
- 代码技巧

优化方法-原则

- 用数据说话；
- 不要过早优化或过度优化；
- 深入理解业务；
- 选择好的衡量标准；

优化方法 – IO密集型

如果是IO密集型的服务，使用多线程实际上比单线程还是要提高很多的；

条件允许的情况下，使用协程和Event Loop的编程模型是最佳的；

IO密集型如果在大量IO操作耗时都比较差时，性能也是很差的，此时改为协程，

或者批量处理IO操作，尽量减少IO操作都是可行方案；

- 批量操作

  我们在写代码时，也切记要规避这样的点：

  单个循环循环操作IO并且等待结果返回；

  合并成批量操作，虽然会增加网络带宽消耗，但往往能大幅降低因为IO延时带来的损耗；

优化方法 – CPU密集型

多线程显然已经不适用于CPU密集型的服务，因为频繁的GIL争抢已经导致程序性能大幅下降；

多进程意味着多解释器，很大程度减少了GIL的争抢，其实很适合CPU密集型服务；

对于CPU密集型的服务，为了减少解释器的损耗，最好可以适用C的扩展库来提高程序性能，

一定程度缓解类型转换带来的性能损耗，而且可以大幅提高基础库的运行速度；

缓存一直是系统性能优化的利器，如果一层缓存不够，那就两层；

Python的编程方法decorator，对于缓存代码改造有着天然的优势；

优化方法 – 缓存

一个Django Model 缓存例子

```python
import cache_store
from django.db import models

CACHE_TIMEOUT = 30
CACHE_KEY_PREFIX = "person:%d"

def cache_id(cls_method):
    def wraps(cls, _id):
        key = CACHE_KEY_PREFIX % _id
        instance = cache_store.get(key, CACHE_TIMEOUT)
        if instance == None:
            instance = cls_method(cls, _id)
            cache_store.set(key, instance, CACHE_TIMEOUT)
        return instance


class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

    @cache_id
    @classmethod
    def get_by_id(cls, _id):
        return cls.objects.get(pk=_id)

```

优化方法 – 缓存

一个有缓存开关的函数

```python
def func_cache(key_prefix, timeout=30, mode=0):
    """
    Func Cache Decorator
    :param timeout: cache timeout
    :param mode: 0 close cache, >0 open cache
    :return: wrapped method
    """
    def wrapped(origin_method):
        def func(*args, **kwargs):
            cache_mode = mode or get_current_switch_mode()
            log.info("Cache Mode For '%s': %s" % (origin_method.func_name, cache_mode))
            if cache_mode == 0:
                result = origin_method(*args, **kwargs)
            else:
                key = get_cache_key(key_prefix, *args, **kwargs)
                result = cache_store.get(key, timeout)
                if result == None:
                    result = origin_method(*args, **kwargs)
                    cache_store.set(key, result, timeout)
            return result

        return func

    return wrapped

@func_cache("FIB", timeout=60 * 60, mode=1)
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) * fib(n - 12)
```

优化方法 – 缓存

序列化方式最好使用高性能的库，比如cjson, cpickle等，性能更好；

在Python应用程序之外，我们也有很多假设高速缓存的方法，比如：

nginx的lua插件扩展；

varnish，squid等等；

缓存更多是一种架构上的东西，我们可以给存储层、DB层、函数层、应用层都加上缓存，但是Python的开发方法在做AOP编程时的用户体验是极佳的。

常用的Lazy单例

```
class eager_meta(type):
    def __init__(clz, name, bases, dic):
        super(eager_meta, clz).__init__(name, bases, dic)
        clz._instance = clz()

class singleton_eager(object):
    __metaclass__ = eager_meta

    @classmethod
    def instance(clz):
        return clz._instance


class singleton_lazy(object):
    __instance = None
    @classmethod
    def instance(clz):
        if clz.__instance is None:
            clz.__instance = singleton_lazy()
        return clz.__instance
```

不要盲目迷信generator一定是较快的

```
>>> timeit.timeit('a = (i for i in range(100000))',number=100)
0.09974312782287598
>>> timeit.timeit('a = [i for i in range(100000)]',number=100)
0.4275360107421875
```

a = (i for i in range(100000))返回的是generator，和列表空间无关，所以性能会好一些；具体应用上，set(i for i in range(100000))会比set([i for i in range(100000)])快。

然而对于需要循环遍历的情况，generator也未必快；下面的例子使用generator反而更慢

```
>>> timeit.timeit('for x in [i for i in range(100000)]:  pass', number=100)
0.4937741756439209
>>> timeit.timeit('for x in (i for i in range(100000)):  pass', number=100)
0.5356199741363525
```

命名空间的问题

```
import time
start_time = time.time()
b = 20
for i in range(1000000): z = 10 *b
print(z)
print("Time: ", time.time() - start_time)
```

```
200
('Time: ', 0.147619009017944 34)
```

```
def main():
    b = 20
    for i in range(1000000): z = 10 * b
    return z

start_time = time.time()
print(main())
print("Time: ", time.time() - start_time)
```

```
200
('Time: ', 0.07180309295654297)
```

命名空间的问题

import dis

dis.dis(main)

main函数因为包含了range, range不再走LOAD_GLOABL

```
4           0 LOAD_CONST               1 (20)
            3 STORE_FAST               0 (b)

5           6 SETUP_LOOP              30 (to 39)
            9 LOAD_GLOBAL              0 (range)
           12 LOAD_CONST               2 (1000000)
           15 CALL_FUNCTION            1
           18 GET_ITER
      >>   19 FOR_ITER                16 (to 38)
           22 STORE_FAST               1 (i)
           25 LOAD_CONST               3 (10)
           28 LOAD_FAST                0 (b)
           31 BINARY_MULTIPLY
           32 STORE_FAST               2 (z)
           35 JUMP_ABSOLUTE           19
      >>   38 POP_BLOCK

6     >>   39 LOAD_FAST                2 (z)
           42 RETURN_VALUE
```

CSDN

2019
Python开发者日

让开发者紧跟
技术潮流

谢谢