计算缓存、优化算法和加速 Python 执行 第一部分

从斐波那契数列谈起

这里 https://robert-mcdermott.gitlab.io/posts/speeding-up-python-with-nim/讨论通过一种称之为 Nim 的技术框架来进行 Python 的加速（后面会对 Nim 技术详细介绍）。文章从计算斐波那契数列开始举例，用递归方式来计算，并且可以看到同样的计算方式，Python 和其他语言的速度上有不小的差异。

上述文章中提到，的确 Python 是一种优秀的编程语言，针对程序员的工作效率进行了优化；令人惊讶的是，你可以非常快速实现的从创意到最低可工作的一个解决方案。它通过其非常灵活的特性和易于编写和阅读的语法，大大缩短了代码开发时间。某种程度上，我们一直说 Python 的实现方式非常接近大脑的思考方式。（从机器学习的发展史来看，大脑的运转速度，比如计算加法的能力远远比不上现在的电脑，但是大脑在复杂推理的场景下有很大的优势。所以很多时候我觉得 Python 的直观易学要比速度快重要的多，如果必须牺牲其中一个特性的话。）

虽然 Python 具有很低的“代码开发时间”，但它具有很高的“代码执行时间”。为了解决Python非常低的执行性能，Python 的许多扩展模块都是用c / c++ 等高性能语言编写的。像 c / c++ 这样的语言与 Python 完全相反;，它们有很高的“代码开发时间”和非常低的“代码执行时间”。对于每种可能需要的计算密集型任务，不可能都有现成的扩展模块，并且在c / c++ 中编写自己的扩展模块以加速Python代码的慢速部分对于大多数Python程序员来说是遥不可及的。好在我们有不少方式可以改变这点。

为了了解 Python 如何执行 CPU 密集型任务，我们使用一个非常耗时的递归 Fibonacci 算法来确定序列中的第 47 个数字，以模拟计算密集型任务，计算复杂度是 (O(2^n)) 。

斐波那契数列： 根据高德纳（Donald Ervin Knuth）的《计算机程序设计艺术》（The Art of Computer Programming），1150年印度数学家 Gopala 和金月在研究箱子包装对象长宽刚好为1和2的可行方法数目时，首先描述这个数列。在西方，最先研究这个数列的人是比萨的列奥那多（意大利人斐波那契 Leonardo Fibonacci），他描述兔子生长的数目时用上了这数列。斐波那契数列就是这样：0, 1, 1, 2, 3, 5, 8, 13, 21……

摘自 https://zh.wikipedia.org/wiki/斐波那契数列

斐波那契数列可以表示如下：

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

用递归方式非常容易实现：

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

if __name__ == "__main__":
    x = 47
    start = time.time()
    res = fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.4f seconds" % (x, res,
elapsed))
```

这里的 47 是指计算到斐波那契梳理的第 47 位，在我的电脑上计算结果是

```
Python Computed fib(47)=2971215073 in 667.7838 seconds
```

将近 11 分钟，电脑配置 Macbook Pro, 2.5 GHz Intel Core i7，16 GB 1600 MHz DDR3，这台电脑是2015年年中的，算是中等计算水平吧。Python 版本 3.7.2。

原文作者的机器性能要更加好一点，Ubuntu 16.04LTS，Intel Xeon E5-2667v3 CPUs 3.20GHz.

如下表对比，我们可以看到 Python 3 在速度上要比 C 语言慢了将近 100 倍，比 Java 也慢了将近70倍。即便是 PyPy，依然和 C 、Java 语言相比不是一个数量级的。

C Computed fib(47)=2971215073 in 4.58 seconds

Java Computed fib(47)=2971215073 in 7.74 seconds

Go Computed fib(47)=2971215073 in 10.94 seconds

JavaScript Computed fib(47)=2971215073 in 21.384 seconds

PyPy Computed fib(47)=2971215073 in 93.63 seconds

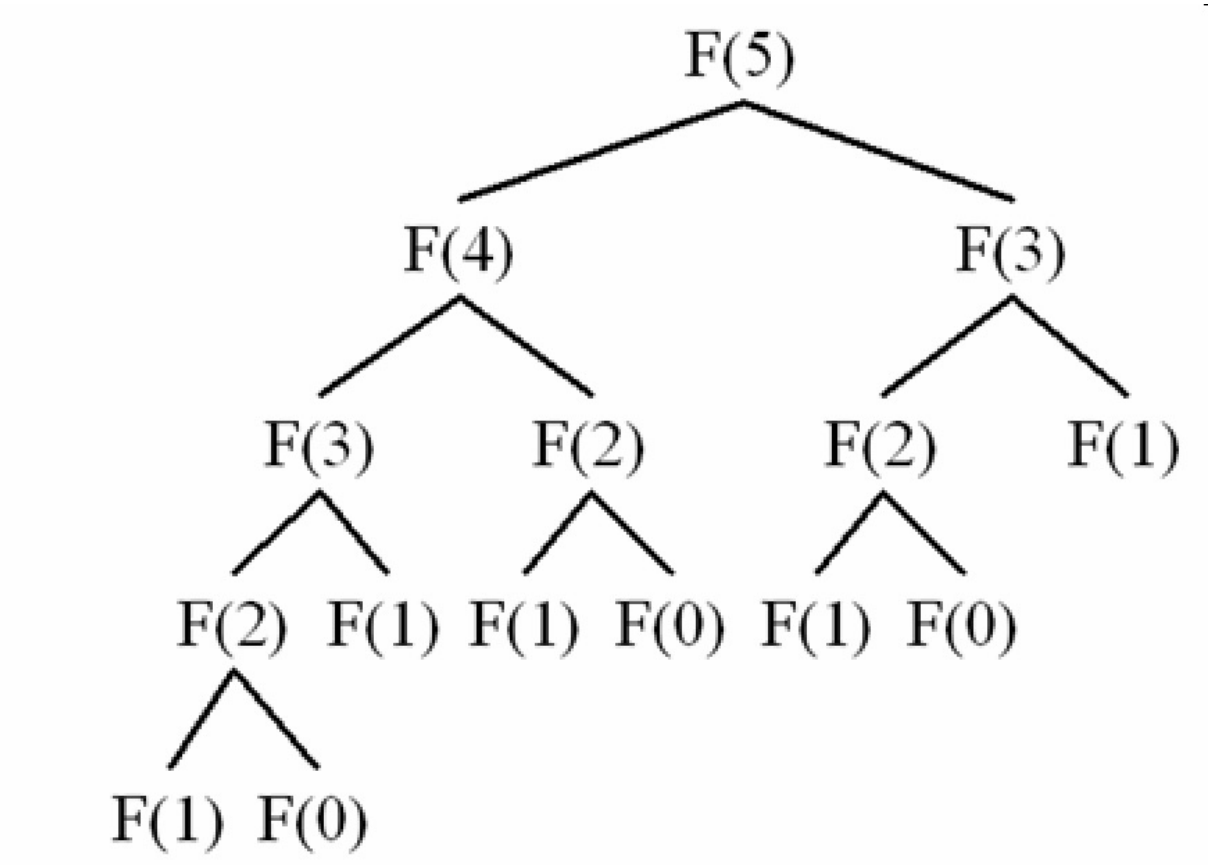
Ruby Computed fib(47)=2971215073 in 191.57 seconds

Python3 Computed fib(47)=2971215073 in 504.55 seconds

Perl5 Computed fib(47)=2971215073 in 980.24 seconds

R Computed fib(47)=2971215073 in 2734.70 seconds

递归计算虽然简洁明了，实际上有包含大量的重复计算，因此称之为计算密集型，下图可以说明递归过程计算的重复性：



利用计算缓存进行优化

我们先用一个非常有效的方式来进行优化，可以让 Python 程序计算斐波那契数列立刻达到 C 语言的水平。

在 stackoverflow 上有专门讨论 python 实现 斐波那契数列的一个帖子。里面有很多实现方式，有的非常巧妙执行速度也非常快。https://stackoverflow.com/questions/494594/how-to-write-the-fibonacci-sequence

我称这个方法是计算缓存，因为递归时候有大量的都是重复计算之前计算过的步骤，我们把每一次的计算输入和输出都存储下来，形成一个缓存，这样一个(O(2^n)) 的复杂度就成了 (O(n))，比如当计算第 5 个数列中的数字时，第 3 个和 第 4 个都已经在缓存中，这样就变成了简单的加法，而不需要真正的递归计算了。并且并不失递归的优雅本质。

我们来看一下代码：

```
def cache_fib(n, _cache={}):

    if n in _cache:
        return _cache[n]
    elif n > 1:
        return _cache.setdefault(n, cache_fib(n-1) + cache_fib(n-2))
    return n

if __name__ == "__main__":
    x = 47
    start = time.time()
    res = cache_fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.8f seconds" % (x, res,
elapsed))
```

这样修改后的执行速度就是极速了，为此我把计算时间的代码精确到了小数点后八位，否则显示的就是 0 。

```
Python Computed fib(47)=2971215073 in 0.00016499 seconds
```

虽然我们用缓存的方式打败了所有其他语言有点胜之不武，但是在真正的业务系统开发过程中，这样做无可厚非，且应该大力推广。

“Fibonacci Numbers in Python” https://mortada.net/fibonacci-numbers-in-python.html 这篇文章也专门讨论了在 Python 中如何实现斐波那契数列，并且展示了如何使用 pandas 和 matplotlib 技术来可视化的分析执行效率。

计算缓存

缓存技术不是新技术，只是其概念在实际使用中再发生着变化。我们可以学习一下标准的缓存的定义。 https://zh.wikipedia.org/wiki/缓存

Cache一词来源于1967年的一篇电子工程期刊论文。其作者将法语词“cache”赋予“safekeeping storage”的涵义，用于计算机工程领域。CPU的缓存曾经是用在超级计算机上的一种高级技术，不过现今计算机上使用的的AMD或Intel微处理器都在芯片内部集成了大小不等的数据缓存和指令缓存，通称为L1缓存（L1 Cache即Level 1 On-die Cache，第一级片上高速缓冲存储器）；而比L1更大容量的L2缓存曾经被放在CPU外部（主板或者CPU接口卡上），但是现在已经成为CPU内部的标准组件；更昂贵的CPU会配备比L2缓存还要大的L3缓存（level 3 On-die Cache第三级高速缓冲存储器）。

主存容量远大于CPU缓存，磁盘容量远大于主存，因此无论是哪一层次的缓存都面临一个同样的问题：当容量有限的缓存的空闲空间全部用完后，又有新的内容需要添加进缓存时，如何挑选并舍弃原有的部分内容，从而腾出空间放入这些新的内容。解决这个问题的算法有几种，如最久未使用算法（LFU）、先进先出算法（FIFO）、最近最少使用算法（LRU）、非最近使用算法（NMRU）等，这些算法在不同层次的缓存上执行时拥有不同的效率和代价，需根据具体场合选择最合适的一种。

在现代开发系统中，由于数据吞吐量太大了，并且重复访问的情况也非常多，为了有效的节约算力、提升响应速度、减少对系统的依赖，缓存技术大大发展。比如 `Redis` 就是广泛使用的一种缓存技术。刚才我们看到在递归计算中使用了缓存，对于性能可以有千万倍的提升。

下面介绍一下 Python 中的一些缓存技术。

lru_cache

Python 语言就是一个瑞士军刀，绝大多数需要的功能都已经整装待发。lru_cache 就是 Python 3.2 开始在 functools 中增加一个函数，通过装饰器的方式来缓存一个函数的执行结果。https://docs.python.org/3/library/functools.html#functools.lru_cache

在上面的文档中，我们可以看到 Python 官方同样是用斐波那契数列作为例子，可见这个斐波那契数列的递归其实是多么的不深入人心啊。

```
import time
from functools import lru_cache


@lru_cache(maxsize=None)
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


if __name__ == "__main__":
    x = 47
    start = time.time()
    res = fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.8f seconds" % (x, res,
elapsed))
    print(fib.cache_info())
```

我们在最后一行增加了显示缓存击中的情况。

```
Python Computed fib(47)=2971215073 in 0.00003409 seconds


CacheInfo(hits=44, misses=47, maxsize=None, currsize=47)
```

执行速度上可以看到比刚才我们自己实现的缓存算法还要快。

DiskCache

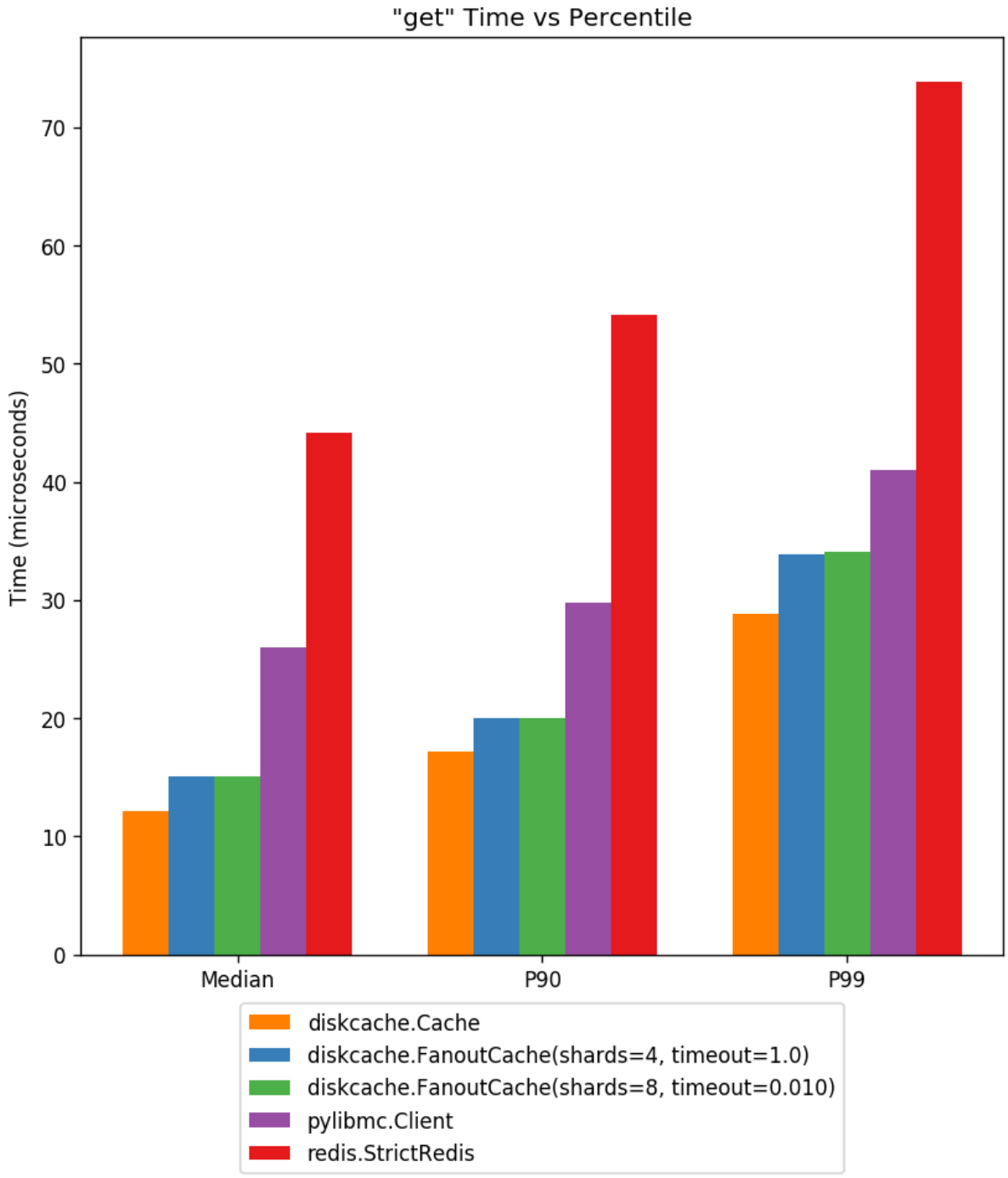
DiskCache 是一个纯 Python 的缓存包，http://www.grantjenks.com/docs/diskcache/

DiskCache 可以有效的只用上 G 空间用于缓存，通过利用坚如磐石的数据库和内存映射文件，缓存性能可以匹配并超越行业标准解决方案。（放在这里解决斐波那契数列问题有点杀鸡用牛刀了！）

DiskCache 的主要功能如下：

- 纯 Python 构造
- 完整的文档
- 100% 单元测试覆盖
- 数小时的压力测试
- Django 兼容的 API
- 线程安全和进程安全
- 支持多种缓存算法 (包括LRU 和 LFU)
- Keys 支持标签、元数据等
- 基于 Python 2.7 开发，在 CPython 2.7, 3.4, 3.5, 3.6 和 PyPy 上测试
- 支持 Linux, Mac OS X 和 Windows
- 通过 Travis CI 和 AppVeyor CI 的集成测试

DiskCache 的功能更像是 `Redis` 和 `MemCached`，并且性能优异，我们可以看看下面的性能对照表。



DiskCache 功能非常多，我们用文档中的一个例子修改一下，来继续刚才的斐波那契数列的 demo，前面的计算缓存是将相关缓存代码写在了函数的逻辑中，通过 DiskCache 的 FanoutCache 来沟通一个函数的装饰器，同样且更加通用的达到计算缓存的效果。

```
from diskcache import FanoutCache
import time
```

```
cache = FanoutCache('/tmp/diskcache/fanoutcache')


@cache.memoize(typed=True, expire=1, tag='fib')
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


if __name__ == "__main__":
    x = 47
    start = time.time()
    res = fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.8f seconds" % (x, res,
elapsed))
```

执行效果如下：

```
Python Computed fib(47)=2971215073 in 0.02766585 seconds
```

装饰器中的 expire 参数是多少毫秒后失效，使用 DiskCache 的话，在装饰器发挥作用前定义了磁盘缓存文件的位置。如果将参数 expire 调整到比较大的数值或者 None 的话，会发现再次执行的话，速度大大提升。

```
Python Computed fib(47)=2971215073 in 0.00032520 seconds
```

DiskCache 功能强大，值得用专门的章节来完整的介绍。

cache.py

如果想既有 lru_cache 这样简单，又暂时不想用 DiskCache 这样的大家伙，但是其文件可以实例化还是不错的一种解决方案，我们还可以尝试用一下 cache.py，出处在这里 https://github.com/bwasti/cache.py

只要 `import cache` 之后，就可以直接使用了。

```
import cache
import time


@cache.cache(timeout=20, fname="my_cache.pkl")
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


if __name__ == "__main__":
    x = 47
    start = time.time()
    res = fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.8f seconds" % (x, res,
elapsed))
```

执行结果如下：

```
Python Computed fib(47)=2971215073 in 0.02948809 seconds
```

列出四种缓存方式的执行速度：

- 内置缓存 0.00016499 seconds
- Python 自带 lru_cache 0.00003409 seconds
- DiskCache 0.00032520 seconds
- cache.py 0.02948809 seconds

现在在这类计算缓存的场景下，Python 自带的 lru_cache 速度最快，而 DiskCache 包性能均衡，考虑到其强大的功能，值得一试，cache.py 的性能一般，但是代码非常简洁，可以学习。