3行代码，4倍提速你的Python数据处理脚本

文-景略集智船长

Python是一门非常适合处理数据和自动化完成重复性工作的编程语言，我们在用数据训练机器学习模型之前，通常都需要对数据进行预处理，而Python就非常适合完成这项工作，比如需要重新调整几十万张图像的尺寸，用Python没问题！你几乎总是能找到一款可以轻松完成数据处理工作的Python库。

然而，虽然Python易于学习，使用方便，但它并非运行速度最快的语言。默认情况下，Python程序使用一个CPU以单个进程运行。不过如果你是在最近几年配置的电脑，通常都是四核处理器，也就是有4个CPU。这就意味着在你苦苦等待Python脚本完成数据处理工作时，你的电脑其实有75%甚至更多的计算资源就在那闲着没事干！

今天我（作者Adam Geitgey——译者注）就教大家怎样通过并行运行Python函数，充分利用你的电脑的全部处理能力。得益于Python的 concurrent.futures 模块，我们只需3行代码，就能将一个普通数据处理脚本变为能并行处理数据的脚本，提速4倍。

普通Python处理数据方法

比方说，我们有一个全是图像数据的文件夹，想用Python为每张图像创建缩略图。

下面是一个短暂的脚本，用Python的内置glob函数获取文件夹中所有JPEG图像的列表，然后用Pillow图像处理库为每张图像保存大小为128像素的缩略图：

```
import globimport osfrom PIL import Imagedef make_image_thumbnail(filename):
    # 缩略图会被命名为"<original_filename>_thumbnail.jpg"
    base_filename, file_extension = os.path.splitext(filename)
    thumbnail_filename = f"{base_filename}_thumbnail{file_extension}"
    # 创建和保存缩略图
    image = Image.open(filename)
    image.thumbnail(size=(128, 128))
    image.save(thumbnail_filename, "JPEG")    return thumbnail_filename# 循环文件夹中所有JPEG图像，为每张图像创建缩略图for image_file in glob.glob("*.jpg"):
```

```
        thumbnail_file = make_image_thumbnail(image_file)

print(f"A thumbnail for {image_file} was saved as {thumbnail_file}")
```

这段脚本沿用了一个简单的模式，你会在数据处理脚本中经常见到这种方法：

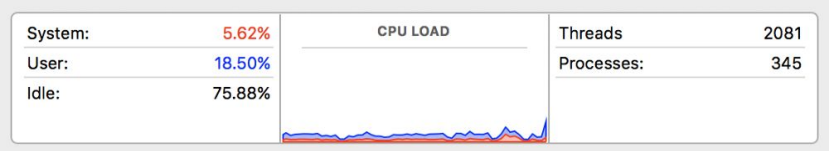
- 首先获得你想处理的文件（或其它数据）的列表
- 写一个辅助函数，能够处理上述文件的单个数据
- 使用for循环调用辅助函数，处理每一个单个数据，一次一个。

咱们用一个包含1000张JPEG图像的文件夹测试一下这段脚本，看看运行完要花多长时间：

```
$ time python3 thumbnails_1.py

A thumbnail for 1430028941_4db9dedd10.jpg was saved as 1430028941_4db9dedd10_thumbnail.jpg

[... about 1000 more lines of output ...]real 0m8.956s

user 0m7.086s

sys 0m0.743s
```

运行程序花了8.9秒，但是电脑的真实工作强度怎样呢？

我们再运行一遍程序，看看程序运行时的活动监视器情况：



电脑有75%的处理资源处于闲置状态！这是什么情况？

这个问题的原因就是我的电脑有4个CPU，但Python只使用了一个。所以程序只是卯足了劲用其中一个CPU，另外3个却无所事事。因此我需要一种方法能将工作量分成4个我能并行处理的单独部分。幸运的是，Python中有个方法很容易能让我们做到！

试试创建多进程

下面是一种可以让我们并行处理数据的方法：

1.将JPEG文件划分为4小块。 2.运行Python解释器的4个单独实例。 3.让每个Python实例处理这4块数据中的一块。 4.将这4部分的处理结果合并，获得结果的最终列表。

4个Python拷贝程序在4个单独的CPU上运行，处理的工作量应该能比一个CPU大约高出4倍，对吧？

最妙的是，Python已经替我们做完了最麻烦的那部分工作。我们只需告诉它想运行哪个函数以及使用多少实例就行了，剩下的工作它会完成。整个过程我们只需要改动3行代码。

首先，我们需要导入concurrent.futures库，这个库就内置在Python中：

```
import concurrent.futures
```

接着，我们需要告诉Python启动4个额外的Python实例。我们通过让Python创建一个Process Pool来完成这一步：

```
with concurrent.futures.ProcessPoolExecutor() as executor:
```

默认情况下，它会为你电脑上的每个CPU创建一个Python进程，所以如果你有4个CPU，就会启动4个Python进程。

最后一步是让创建的Process Pool用这4个进程在数据列表上执行我们的辅助函数。完成这一步，我们要将已有的for循环：

```
for image_file in glob.glob("*.jpg"):thumbnail_file = make_image_thumbnail(image_file)
```

替换为新的调用executor.map()：

```
image_files = glob.glob("*.jpg")for image_file, thumbnail_file in zip(image_files, executor.map(make_image_thumbnai
```

该executor.map()函数调用时需要输入辅助函数和待处理的数据列表。这个函数能帮我完成所有麻烦的工作，包括将列表分为多个子列表、将子列表发送到每个子进程、运行子进程以及合并结果等。干得漂亮！

这也能为我们返回每个函数调用的结果。Executor.map()函数会按照和输入数据相同的顺序返回结果。所以我用了Python的zip()函数作为捷径，一步获取原始文件名和每一步中的匹配结果。

这里是经过这三步改动后的程序代码：

```
import globimport osfrom PIL import Imageimport concurrent.futuresdef make_image_thumbnail(filename):
        # 缩略图会被命名为 "<original_filename>_thumbnail.jpg"
        base_filename, file_extension = os.path.splitext(filename)
        thumbnail_filename = f"{base_filename}_thumbnail{file_extension}"
        # 创建和保存缩略图
        image = Image.open(filename)
        image.thumbnail(size=(128, 128))
        image.save(thumbnail_filename, "JPEG")        return thumbnail_filename# 创建Process Pool，默认为电脑的每个CPU创建一个with concurrent.futures.ProcessPoolExecutor() as executor:        # 获取需要处理的文件列表
        image_files = glob.glob("*.jpg")        # 处理文件列表，但通过Process Pool划分工作，使用全部CPU!
        for image_file, thumbnail_file in zip(image_files, executor.map(make_image_thumbnail, image_files)):
                print(f"A thumbnail for {image_file} was saved as {thumbnail_file}")
```

我们来运行一下这段脚本，看看它是否以更快的速度完成数据处理：

```
$ time python3 thumbnails_2.py
A thumbnail for 1430028941_4db9dedd10.jpg was saved as 1430028941_4db9dedd10_thumbnail.jpg
[... about 1000 more lines of output ...]real 0m2.274s
user 0m8.959s
```

```
sys  0m0.951s
```

脚本在2.2秒就处理完了数据！比原来的版本提速4倍！之所以能更快的处理数据，是因为我们使用了4个CPU而不是1个。

但是如果你仔细看看，会发现“用户”时间几乎为9秒。那为何程序处理时间为2.2秒，但不知怎么搞得运行时间还是9秒？这似乎不太可能啊？

这是因为“用户”时间是所有CPU时间的总和，我们最终完成工作的CPU时间总和一样，都是9秒，但我们使用4个CPU完成的，实际处理数据时间只有2.2秒！

注意：启用更多Python进程以及给子进程分配数据都会占用时间，因此靠这个方法并不能保证总是能大幅提高速度。如果你要处理非常大的数据集，这里有篇设置将数据集切分成多少小块的文章，可以读读，会对你帮助甚大.

这种方法总能帮我的数据处理脚本提速吗？

如果你有一列数据，并且每个数据都能单独处理时，使用我们这里所说的Process Pools是一个提速的好方法。下面是一些适合使用并行处理的例子：

- **从一系列单独的网页服务器日志里抓取统计数据。**
- **从一堆XML，CSV和JSON文件中解析数据。**
- **对大量图片数据做预处理，建立机器学习数据集。**

但也要记住，Process Pools并不是万能的。使用Process Pool需要在独立的Python处理进程之间来回传递数据。如果你要处理的数据不能在处理过程中被有效地传递，这种方法就行不通了。简而言之，你处理的数据必须是Python知道怎么应对的类型。

同时，也无法按照一个预想的顺序处理数据。如果你需要前一步的处理结果来进行下一步，这种方法也行不通。

那GIL的问题呢？

你可能知道Python有个叫全局解释器锁（Global Interpreter Lock）的东西，即GIL。这意味着即使你的程序是多线程的，每个线程也只能执行一个Python指令。GIL确保任何时候都只有一个Python线程执行。换句话说，多线程的Python代码并不能真正地并行运行，从而无法充分利用多核CPU。

但是Process Pool能解决这个问题！因为我们是运行单独的Python实例，每个实例都有自己的GIL。这样我们获得是真正能并行处理的Python代码！

不要害怕并行处理！

有了concurrent.futures库，Python就能让你简简单单地修改一下脚本后，立刻让你电脑上所有CPU投入到工作中。不要害怕尝试这种方法，一旦你掌握了，它就跟一个for循环一样简单，却能让你的数据处理脚本快到飞起。

转自：　　优达学城Udacity