ChainMap：将多个字典视为一个，解锁Python超能力。

简而言之ChainMap：将多个字典视为一个，解锁Python超能力。
Python标准库中的集合模块包含许多为性能而设计的实用的数据结构。著名的包括命名元组或计数器。
今天，通过实例，我们来看看鲜为人知的**ChainMap**。通过浏览具体的示例，我希望给你一个提示，关于在更高级的Python工作中使用ChainMap将如何从中受益。
免责声明：这篇文章是关于Python的一个相当高级的特性。如果你刚入门，请等一等！

ChainMap是什么?

ChainMap是由Python标准库提供的一种数据结构，允许你将多个字典视为一个。
ChainMap上的官方文档如下：
ChainMap将多个dict或其他映射组合在一起以创建单个可更新视图。[...]查找基础映射，直到找到key为止。[...]如果其中一个基础映射得到更新，这些更改将反映在ChainMap中。 [...] 支持所有常用的字典方法。
换句话说:**ChainMap是一个基于多dict的可更新的视图，它的行为就像一个普通的dict。**
你以前可能从来没有听说过ChainMap，你可能会认为ChainMap的使用情况是非常特定的。坦率地说，你是对的。
我知道的用例包括：

- 通过多个字典搜索
- 提供链缺省值
- 经常计算字典子集的性能关键的应用程序

我们将通过两个例子来说明。

注意：这两个例子是受到Mike Driscoll在The Mouse vs. The Python写的一篇文章的启发。为了我的目的，我已经调整了它们，但一定要阅读他的帖子另一个关于ChainMap的观点！

示例：购物清单

作为使用ChainMap的第一个例子，让我们考虑一张购物清单。我们的清单可能包含玩具，电脑，甚至衣服。所有这些条目都有价格，所以我们将把我们的条目存储在名称价格映射中。

```
>>> toys = {'Blocks': 30, 'Monopoly': 20}
>>> computers = {'iMac': 1000, 'Chromebook': 800, 'PC': 400}
>>> clothing = {'Jeans': 40, 'T-Shirt': 10}

```

现在我们可以使用ChainMap在这些不同的集合上建立一个单一的视图：

```
>>> from collections import ChainMap
>>> inventory = ChainMap(toys, computers, clothing)

```

这使得我们可以查询清单，**就像它是一个单一的字典**：

```
>>> inventory['Monopoly']
20

```

正如官方文档所述，ChainMap支持所有常用的字典方法。我们可以使用.get()来搜索可能不存在的条目，或者使用 .pop()删除条目。

```
>>> inventory.get('Mario Bros.')
None
>>> inventory.pop('Blocks')
200
>>> inventory['Blocks']  # KeyError: 'Blocks'

```

如果我们现在把玩具添加到toys字典里，它也将在清单中可用。这是ChainMap的可更新的方面。

```
>>> toys['Nintendo'] = 200
>>> inventory['Nintendo']
200

```

Oh和ChainMap有一个恰当的字符串表示形式：

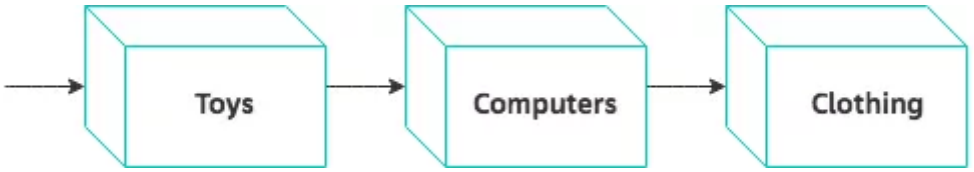
```
>>> str(inventory)
ChainMap({'Monopoly': 20, 'Nintendo': 200}, {'iMac': 1000,
'Chromebook': 800, 'PC': 400}, {'Jeans': 40, 'T-Shirt': 10})

```

一个很好的特点是，在我们的例子中，toys, computers和clothing都是在相同的上下文中（解释器），它们可以来自完全不同的模块或包。这是因为ChainMap**通过引用**存储底层字典。

第一个例子是使用ChainMap一次搜索多个字典。

事实上，当构建ChainMap时，我们所做的就是有效地构建一系列字典。当查找清单中的一个项时，toys首先被查找，然后是computers，最后是clothing。



ChainMap真的只是一个映射链！

实际上，ChainMap的另一个任务是**维护链的默认值**。

我们将以一个命令行应用程序的例子来说明这是什么意思。

示例：CLI配置

让我们面对现实，管理命令行应用程序的配置可能是困难的。

配置来自多个源：命令行参数、环境变量、本地文件等。

我们通常实施**优先级**的概念：如果A和B都定义参数P，A的P值将被使用，因为它的优先级高于B。

例如，如果传递了命令行参数，我们可能希望在环境变量上使用命令行参数。

如何轻松地管理配置源的优先级？

一个答案是将所有配置源存储在ChainMap中。

因为ChainMap中的查找是连续地对每个底层映射执行的（按照他们传给构造函数的顺序），所以我们可以很容易地实现我们寻找的优先级。

下面是一个简单的命令行应用程序。调试参数从命令行参数、环境变量或硬编码默认值中提取：

```python
# cli.py
import argparse
import os
from collections import ChainMap

defaults = {'debug': False}

parser = argparse.ArgumentParser()
parser.add_argument('--debug')
args = parser.parse_args()
cli_args = {key: value for key, value in vars(args).items() if value}

config = ChainMap(cli_args, os.environ, defaults)

print(config.get('debug'))

```

在执行脚本时，我们可以检查是否首先在命令行参数中查找debug，然后是环境变量，最后是默认值：

```
$ python cli.py
False
$ python cli.py --debug 1
1
$ export debug=True
$ python cli.py
True
$ python cli.py --debug yes
yes
```

整洁，对吧?

我为什么关心?

坦率地说，ChainMap是那些你可以忽略的Python特性之一。

还有其他ChainMap的替代方案。例如，使用更新循环——例如创建一个dict并用字典.update()它——可能奏效。但是，这只有在您不需要跟踪项目的起源时才有效，就像我们的多源CLI配置示例中的情况一样。

但是，当你知道ChainMap存在的时候，ChainMap可以让你更轻松，你的代码更优雅。

事实上，我第一次使用ChainMap是在一周前。为什么以前没有呢？我根本没用过。

我使用它是因为我需要频繁地计算字典的子集（基于值的属性），这代价很大。我需要实现**恒定的时间查找**以满足性能要求。

我决定把字典分成两个不同的dict，并在插入时执行分支。然后我用ChainMap把这两个dict组合在一起。这样，我就可以在单个字典中保留最初的视图——但也可以在固定时间内查找每个单独的字典。

总结

总而言之，我们一起看了ChainMap是什么，一些具体的使用示例，以及如何在现实生活中，性能关键的应用程序中使用ChainMap。

如果您想了解更多关于Python的高性能数据容器的信息，请务必从Python的标准库中collections模块中查看其他出色类和函数。