# Python之functools

## 讲师：Wayne

从业十余载，漫漫求知路

# functools模块

- reduce方法
  - reduce方法，顾名思义就是减少
  - reduce(function, sequence[, initial]) -> value
  - 可迭代对象不能为空；初始值没提供就在可迭代对象中取一个元素

```
from functools import reduce

nums = [6, 9, 4, 2, 4, 10, 5, 9, 6, 9]
print(nums)
print(sum(nums))
print(reduce(lambda val, x: val + x, nums))
```

# functools模块

- partial方法
  - 偏函数，把函数部分的参数固定下来，相当于为部分的参数添加了一个固定的默认值，形成一个新的函数并返回
  - 从partial生成的新函数，是对原函数的封装

# functools模块

- partial方法举例

```
import functools

def add(x, y) -> int:
    return x + y

newadd = functools.partial(add, y=5)

print(newadd(7))
print(newadd(7, y=6))
print(newadd(y=10, x=6))

import inspect
print(inspect.signature(newadd))
```

# functools模块

- partial方法举例

```
import functools

def add(x, y, *args) -> int:
    print(args)
    return x + y

newadd = functools.partial(add, 1,3,6,5)

print(newadd(7))
print(newadd(7, 10))
print(newadd(9, 10, y=20, x=26)) #
print(newadd())

import inspect
print(inspect.signature(newadd))
```

# functools模块

- partial函数本质

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords): # 包装函数
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func # 保留原函数
    newfunc.args = args # 保留原函数的位置参数
    newfunc.keywords = keywords # 保留原函数的关键字参数参数
    return newfunc

def add(x,y):
    return x+y

foo = partial(add,4)
foo(5)
```

# functools模块

- partial函数
  - 分析functools.wraps的实现

# functools模块

- @functools.lru_cache(maxsize=128, typed=False)
  - Least-recently-used装饰器。lru，最近最少使用。cache缓存
  - 如果maxsize设置为None，则禁用LRU功能，并且缓存可以无限制增长。当maxsize是二的幂时，LRU功能执行得最好
  - 如果typed设置为True，则不同类型的函数参数将单独缓存。例如，f(3)和f(3.0)将被视为具有不同结果的不同调用

# functools模块

- 举例

```
import functools
import time
@functools.lru_cache()
def add(x, y, z=3):
    time.sleep(z)
    return x + y

add(4, 5)
add(4.0, 5)
add(4, 6)
add(4, 6, 3)
add(6, 4)
add(4, y=6)
add(x=4, y=6)
add(y=6, x=4)
```

思考：缓存的机制是什么？

# functools模块

- lru_cache装饰器
  - 通过一个字典缓存被装饰函数的调用和返回值
  - key是什么？分析代码看看

    functools._make_key((4,6),{'z':3},False)

    functools._make_key((4,6,3),{},False)

    functools._make_key(tuple(),{'z':3,'x':4,'y':6},False)

    functools._make_key(tuple(),{'z':3,'x':4,'y':6}, True)

# functools模块

- lru_cache装饰器
  - 斐波那契数列递归方法的改造

```
import functools


@functools.lru_cache() # maxsize=None
def fib(n):
    return 1 if n<3 else fib(n-1) + fib(n-2)


print([fib(i+1) for i in range(35)])
```

# functools模块

- lru_cache装饰器应用
  - 使用前提
    - 同样的函数参数一定得到同样的结果
    - 函数执行时间很长，且要多次执行
  - 本质是函数调用的参数=>返回值
  - 缺点
    - 不支持缓存过期，key无法过期、失效
    - 不支持清除操作
    - 不支持分布式，是一个单机的缓存
  - 适用场景，单机上需要空间换时间的地方，可以用缓存来将计算变成快速的查询

# 装饰器应用练习

- 一、写一个命令分发器
  - 程序员可以方便的注册函数到某一个命令，用户输入命令时，路由到注册的函数
  - 如果此命令没有对应的注册函数，执行默认函数
  - 用户输入用input(">>")

# 装饰器应用练习

- 二、实现一个cache装饰器，实现可过期被清除的功能
  - 简化设计，函数的形参定义不包含可变位置参数、可变关键词参数和keyword-only参数
  - 可以不考虑缓存大小，也不用考虑缓存满了之后的换出问题
  - 进阶

```
def add(x=4, y=5):
    time.sleep(3)
    return x + y

以下5种，可以认为是同一种调用
print(1, add(4,5))
print(2, add(4))
print(3, add(y=5))
print(4, add(x=4,y=5))
print(5, add(y=5,x=4))
```

# 谢谢

咨询热线 400-080-6560