functools模块

reduce

- functools.reduce(function, iterable[, initial])
- 就是减少的意思
- 初始值没有提供就在可迭代对象中取一个

```
from functools import reduce

s = sum(range(10))
print(s)

s = reduce(lambda x: x, range(10))
print(s) # TypeError: <lambda>() takes 1 positional argument but 2 were given
```

从上面的异常推断lambda应该2个参数

```
s = reduce(lambda x,y: print(x,y), range(10))
print(s)

返回结果如下
0 1
None 2
None 3
None 4
None 5
None 6
None 7
None 8
None 9
None
```

上一次lambda函数返回值会成为下一次的x

```
from functools import reduce

s = sum(range(10))
print(s)

s = reduce(lambda x,y: x + y, range(10), 100)
print(s)
```

sum只能求和，reduce能做更加复杂的迭代计算。

思考：5的阶乘怎么做?

partial

偏函数

- 把函数部分参数固定下来，相当于为部分的参数添加了固定的默认值，形成一个新的函数，并返回这个新函数
- 这个新函数是对原函数的封装

```python
from functools import partial

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

newadd = partial(add, y=5)

print(newadd(4))
print(newadd(4, y=15))
print(newadd(x=4))
print(newadd(4, 6)) # 可以吗
print(newadd(y=6, x=4))

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

```python
from functools import partial

def add(x, y, *args):
    return x + y + sum(args)

newadd = partial(add, 1, 2, 3, 4, 5)

print(newadd())
print(newadd(1))
print(newadd(1, 2))
print(newadd(x=1)) #
print(newadd(x=1, y=2)) #

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

partial本质

```python
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)
```

尝试分析functools.wraps的实现。类别下面的实现

```
from functools import partial, wraps
import inspect

def add(a, b, c, d):
    return a + b + c + d

newadd = partial(add, b=2, c=3, d=4)
print(inspect.signature(newadd))
```

lru_cache

`@functools.lru_cache(maxsize=128, typed=False)`

- lru即Least-recently-used，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
- Python 3.8 简化了调用，可以使用

```
@functools.lru_cache
def add():
    pass

# 等价于
@functools.lru_cache(128)
def add():
    pass
```

```
from functools import lru_cache
import time

@lru_cache()
def add(x, y=5):
    print('-' * 30)
    time.sleep(3)
    return x + y

print(1, add(4, 5))
print(2, add(4, 5))
print(3, add(x=4, y=5))
print(4, add(y=5, x=4))
print(5, add(4.0, 5))
print(6, add(4))
```

到底什么调用才能用缓存呢?

lru_cache本质

- 内部使用了一个字典
- key是由_make_key函数构造出来

```
from functools import _make_key

print(_make_key((4, 5), {}, False))
print(_make_key((4, 5), {}, True))
print(_make_key((4,), {'y':5}, False))
print(_make_key((), {'x':4, 'y':5}, False))
print(_make_key((), {'y':5, 'x':4}, False))
```

应用

```
# 斐波那契数列lru_cache版
from functools import lru_cache

@lru_cache()
def fib(n):
    return 1 if n < 3 else fib(n-1) + fib(n-2)

print(fib(101))
```

总结

lru_cache装饰器应用

- 使用前提
    1. 同样的函数参数一定得到同样的结果，至少是一段时间内，同样输入得到同样结果
    2. 计算代价高，函数执行时间很长
    3. 需要多次执行，每一次计算代价高
- 本质是建立函数调用的参数到返回值的映射
- 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
- lru_cache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

学习lru_cache可以让我们了解缓存背后的原理。