user

Python3 迭代器与生成器教程

assistant

当然！下面是一份 Python 3 中迭代器（Iterator）与生成器（Generator）的详细教程，适合初学者和有一定基础的开发者。

一、什么是可迭代对象（Iterable）？

在 Python 中，可迭代对象 是指可以被 `for` 循环遍历的对象。例如：列表、元组、字符串、字典、集合等。

```
for i in [1, 2, 3]:
    print(i)
```

这些对象之所以能被遍历，是因为它们实现了 `__iter__()` 方法（或 `__getitem__()` 方法），返回一个 迭代器（Iterator）。

二、什么是迭代器（Iterator）？

迭代器 是一个实现了 迭代器协议 的对象，即同时具有：

- `__iter__()`：返回自身（self）
- `__next__()`：返回下一个元素，若没有更多元素则抛出 `StopIteration` 异常

示例：手动创建一个迭代器

```
class MyIterator:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration
        value = self.data[self.index]
        self.index += 1
        return value
```

```
# 使用
my_iter = MyIterator([10, 20, 30])
for x in my_iter:
    print(x)
```

> 注意：迭代器只能被遍历一次！再次使用会抛出 StopIteration 。

三、什么是生成器（Generator）？

生成器 是一种特殊的迭代器，由函数定义，使用 yield 关键字。它更简洁、内存友好。

1. 用函数 + yield 创建生成器

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

# 使用
gen = count_up_to(3)
for num in gen:
    print(num)  # 输出: 1, 2, 3
```

- 每次调用 next(gen) 时，函数从上次 yield 处继续执行。
- 函数状态（局部变量、执行位置）被自动保存。

2. 生成器表达式（Generator Expression）

类似列表推导式，但用圆括号 ()，返回生成器对象：

```
gen = (x**2 for x in range(5))
print(type(gen))  # `<class 'generator'>`

for val in gen:
    print(val)  # 0, 1, 4, 9, 16
```

> 对比： [x**2 for x in range(5)] 返回列表（一次性生成所有值，占用内存）；而生成器是“按需生成”，节省内存。

四、迭代器 vs 生成器

特性	迭代器	生成器
实现方式	类实现 __iter__ 和 __next__	函数中使用 yield 或生成器表达式

特性	迭代器	生成器
内存占用	通常较低（但取决于实现）	极低（惰性求值）
可读性	较复杂	更简洁
是否可重复使用	否（用完即废）	否（用完即废）
本质	所有生成器都是迭代器	生成器是迭代器的子集

五、常见用途

1. 处理大文件（避免一次性加载）

```
def read_large_file(file_path):
    with open(file_path, 'r') as f:
        for line in f:
            yield line.strip()

for line in read_large_file('huge.log'):
    process(line)
```

2. 无限序列

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
for _ in range(10):
    print(next(fib))
```

六、内置工具

- iter(obj)：将可迭代对象转为迭代器
- next(iterator)：获取下一个元素
- itertools 模块：提供高级迭代器工具（如 chain，cycle，islice 等）

```
import itertools

# 无限循环
cycler = itertools.cycle([1, 2, 3])
print(next(cycler))  # 1
```

```
print(next(cycler))  # 2
print(next(cycler))  # 3
print(next(cycler))  # 1
```

七、小结

- 可迭代对象 → 能被 `for` 遍历（如 list, str）
- 迭代器 → 实现了 `__iter__` 和 `__next__`，可逐个取值
- 生成器 → 用 `yield` 或 `(x for x in ...)` 创建，是最简单的迭代器写法
- 生成器适合处理大数据、流式数据、无限序列等场景

如果你希望我提供练习题、性能对比或进阶技巧（如 `send()`，`throw()`，`close()`），也可以告诉我！

是否需要我生成一份 PDF 或 Markdown 笔记？

user

如何用生成器优化大文件处理？

assistant

使用 生成器（Generator） 是优化大文件处理的核心技巧之一，因为它能实现 “惰性加载”（Lazy Loading） —— 一次只读取和处理一小部分数据，而不是把整个文件加载到内存中。

下面从原理、示例、最佳实践三个方面详细说明如何用生成器高效处理大文件。

一、为什么需要优化？

假设你有一个 10GB 的日志文件：

- 如果用 `readlines()` 或 `list(open(...))`，Python 会一次性把所有行读入内存 → 内存爆炸！
- 而使用生成器，每次只读一行（或一批），处理完就释放 → 内存占用恒定（几 KB 到几 MB）

二、基础：逐行读取（最常见场景）

✅ 推荐写法（本身就是生成器行为）

```
def read_lines(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        for line in f:        # 文件对象本身是迭代器！
```

```
        yield line.strip()  # 可选：去除换行符

# 使用
for line in read_lines('huge.log'):
    if 'ERROR' in line:
        print(line)
```

> 💡 实际上，文件对象 `f` 本身就是迭代器，所以你甚至可以不用显式写生成器函数：
>
> ```
> with open('huge.log') as f:
>     for line in f:
>         process(line)
> ```
>
> 但封装成生成器函数更利于复用和组合。

三、进阶：按块（chunk）或按记录读取

有些文件不是按行组织的（比如 JSON 数组、二进制数据、多行日志条目），这时需要自定义生成逻辑。

示例 1：读取多行组成的日志条目（以时间戳开头为新条目）

```
import re

def parse_multiline_logs(filename):
    buffer = []
    time_pattern = re.compile(r'^\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}')

    with open(filename, 'r') as f:
        for line in f:
            if time_pattern.match(line):  # 新日志开始
                if buffer:
                    yield ''.join(buffer)  # 返回上一条完整日志
                buffer = [line]
            else:
                buffer.append(line)
        if buffer:
            yield ''.join(buffer)  # 最后一条

# 使用
for log_entry in parse_multiline_logs('app.log'):
    if 'CRITICAL' in log_entry:
        alert(log_entry)
```

示例 2：按固定字节数读取（适合二进制文件）

```
def read_in_chunks(file_path, chunk_size=8192):
    """每次读取 chunk_size 字节"""
    with open(file_path, 'rb') as f:
```

```
        while True:
            chunk = f.read(chunk_size)
            if not chunk:
                break
            yield chunk

# 用于计算文件哈希
import hashlib
def file_md5(file_path):
    hash_md5 = hashlib.md5()
    for chunk in read_in_chunks(file_path):
        hash_md5.update(chunk)
    return hash_md5.hexdigest()
```

四、结合生成器表达式 + 管道处理

你可以将多个生成器串联成数据处理管道，像流水线一样高效处理：

```
def read_lines(filename):
    with open(filename) as f:
        yield from f  # 等价于 for line in f: yield line

def filter_errors(lines):
    for line in lines:
        if 'ERROR' in line:
            yield line

def extract_timestamps(error_lines):
    import re
    pattern = re.compile(r'\[(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\]')
    for line in error_lines:
        match = pattern.search(line)
        if match:
            yield match.group(1)

# 组合使用（内存始终很低）
timestamps = extract_timestamps(
    filter_errors(
        read_lines('server.log')
    )
)

for ts in timestamps:
    print(ts)
```

> 这种“生成器管道”模式非常适合 ETL（抽取-转换-加载）任务。

五、性能与内存优势对比

方法	内存占用	适用场景
open().readlines()	高（整个文件）	小文件（<100MB）
for line in open()	低（一行）	文本日志、CSV
自定义生成器（按块/记录）	低（可控）	大文件、非标准格式
mmap + 生成器	极低（虚拟内存）	超大文件随机访问（进阶）

六、注意事项

1. 不要在生成器里打开文件后不关闭 → 务必用 with open
2. 避免在生成器中积累大量中间数据（如不断 append 到 list）
3. 编码问题：大文件常含非 UTF-8 编码，需指定 encoding='latin1' 或 errors='ignore'
4. Windows 换行符： strip() 可统一处理 \r\n 和 \n

七、总结

✅ 用生成器优化大文件处理的关键点：

- 利用文件对象本身的迭代器特性（逐行读）
- 对复杂格式，自定义生成器按“逻辑记录”读取
- 用生成器管道组合处理步骤
- 始终保持低内存占用，避免 list() 强制展开

如果你有具体的文件格式（如 CSV、JSON、XML、日志等），我可以提供针对性的生成器处理方案！是否需要？

user

assistant

user

assistant

user

assistant

user

assistant

user