استفاده از `async` و `await` در برنامه‌نویسی غیرهمزمان

در پایتون، برای مدیریت برنامه‌های غیرهمزمان، از کلمات کلیدی `async` و `await` استفاده می‌شود. این دو کلمه به شما امکان می‌دهند که به راحتی توابع غیرهمزمان بنویسید و آن‌ها را به صورت غیرهمزمان اجرا کنید.

1. مفهوم و نحوه استفاده از `async` برای تعریف توابع غیرهمزمان

کلمه کلیدی `async` برای تعریف توابع غیرهمزمان (coroutines) در پایتون استفاده می‌شود. یک تابع که با `async` تعریف می‌شود به طور پیش‌فرض یک **coroutine** است که می‌تواند به طور غیرهمزمان اجرا شود.

مثال:

```
import asyncio

# تعریف تابع غیرهمزمان
async def my_function():
    print("Start")
    await asyncio.sleep(2)  # شبیه‌سازی عملیات غیرهمزمان
    print("End")
```

در اینجا:

- `async def` برای تعریف یک تابع غیرهمزمان استفاده می‌شود.
- `await` برای فراخوانی توابعی که به صورت غیرهمزمان هستند، استفاده می‌شود.

2. استفاده از `await` برای فراخوانی توابع غیرهمزمان و بلوکه کردن برنامه تا دریافت نتیجه

کلمه کلیدی `await` برای فراخوانی توابع غیرهمزمان و منتظر ماندن برای نتیجه آن‌ها استفاده می‌شود. در هنگام استفاده از `await`، کد جاری تا زمانی که نتیجه حاصل نشود، متوقف نمی‌شود، بلکه دیگر توابع غیرهمزمان می‌توانند اجرا شوند.

مثال:

```
import asyncio

async def my_function():
    print("Start")
    await asyncio.sleep(2)  # منتظر ماندن برای 2 ثانیه
    print("End")

# اجرای تابع غیرهمزمان
asyncio.run(my_function())
```

در این مثال:

- وقتی `await asyncio.sleep(2)` فراخوانی می‌شود، برنامه برای مدت زمان 2 ثانیه متوقف می‌شود، اما در این مدت، هیچ بخشی از برنامه متوقف نمی‌شود و برنامه می‌تواند سایر کارهای غیرهمزمان را انجام دهد.

3. مدیریت چندین کار غیرهمزمان با استفاده از `()asyncio.gather` و `()asyncio.create_task`

برای مدیریت چندین کار غیرهمزمان، می‌توان از `()asyncio.gather` یا `()asyncio.create_task` استفاده کرد. این توابع به شما این امکان را می‌دهند که چندین کار غیرهمزمان را هم‌زمان اجرا کرده و نتایج آن‌ها را جمع‌آوری کنید.

- `()asyncio.gather` : این متد به شما امکان می‌دهد که چندین coroutine را هم‌زمان اجرا کنید و نتایج آن‌ها را به صورت یک لیست دریافت کنید.

```
import asyncio

async def task1():
    await asyncio.sleep(1)
    return "Task 1 Complete"

async def task2():
    await asyncio.sleep(2)
    return "Task 2 Complete"

async def main():
    result = await asyncio.gather(task1(), task2())  # اجرای همزمان دو وظیفه
    print(result)

# اجرای برنامه
asyncio.run(main())
```

- `()asyncio.create_task` : این متد برای شروع یک coroutine به صورت غیرهمزمان و مدیریت آن در یک `Task` استفاده می‌شود.

```
import asyncio

async def task1():
    await asyncio.sleep(1)
    return "Task 1 Complete"

async def task2():
    await asyncio.sleep(2)
    return "Task 2 Complete"

async def main():
    task1_instance = asyncio.create_task(task1())  # ایجاد task برای task1
    task2_instance = asyncio.create_task(task2())  # ایجاد task برای task2
    result1 = await task1_instance
    result2 = await task2_instance
    print(result1, result2)

# اجرای برنامه
asyncio.run(main())
```

4. استفاده از `async with` برای مدیریت منابع در توابع غیرهمزمان

کلمه کلیدی `async with` برای استفاده از context managers در توابع غیرهمزمان استفاده می‌شود. این قابلیت به شما اجازه می‌دهد تا منابعی مانند فایل‌ها، اتصالات شبکه و پایگاه‌داده را در یک قالب غیرهمزمان مدیریت کنید.

مثال:

```
import asyncio

class MyAsyncContextManager:
    async def __aenter__(self):
        print("Entering the context")
        await asyncio.sleep(1)
        return "Resource"

    async def __aexit__(self, exc_type, exc, tb):
        print("Exiting the context")
        await asyncio.sleep(1)

async def main():
    async with MyAsyncContextManager() as resource:
        print(f"Using {resource}")

# اجرای برنامه
asyncio.run(main())
```

در اینجا:

- `__aenter__` و `__aexit__` مشابه `__enter__` و `__exit__` در context managers معمولی هستند، اما به صورت غیرهمزمان (asynchronous) اجرا می‌شوند.
- `async with` برای استفاده از این context managerها در توابع غیرهمزمان به کار می‌رود.

5. مدیریت خطاها و استثناها در توابع غیرهمزمان با استفاده از `try-except`

در توابع غیرهمزمان نیز می‌توان از ساختار `try-except` برای مدیریت خطاها و استثناها استفاده کرد. همانطور که در توابع همزمان می‌توان از این ساختار برای مدیریت خطاها استفاده کرد، در توابع غیرهمزمان نیز به همین شیوه عمل می‌شود.

```
import asyncio

async def my_function():
    try:
        print("Start")
        await asyncio.sleep(1)
        raise ValueError("An error occurred!")  # شبیه‌سازی خطا
    except ValueError as e:
        print(f"Caught an error: {e}")
    print("End")

# اجرای تابع غیرهمزمان
asyncio.run(my_function())
```

در اینجا:

- خطای `ValueError` در هنگام اجرای تابع غیرهمزمان مدیریت می‌شود و در صورت وقوع، پیام خطا چاپ می‌شود.

6. مقایسه عملکرد `asyncio` با استفاده از `Threading` و `Multiprocessing` در پردازش‌های I/O-bound و CPU-bound

- **I/O-bound operations**: برای کارهای ورودی/خروجی مانند خواندن از فایل‌ها، شبکه و پایگاه‌داده، `asyncio` می‌تواند گزینه‌ای بهینه باشد زیرا نیاز به مصرف منابع اضافی ندارد و عملکرد بهتری ارائه می‌دهد.
  - **`asyncio`**: برای پردازش‌های I/O-bound که منتظر پاسخ از منابع خارجی هستند، مناسب است.
  - **`Threading`** و **`Multiprocessing`**: می‌توانند در پردازش‌های I/O-bound هم‌زمان استفاده شوند، اما در مقایسه با `asyncio` کارآمدی کمتری دارند.
- **CPU-bound operations**: برای پردازش‌های محاسباتی سنگین که نیاز به پردازش زیاد از CPU دارند، استفاده از `asyncio` بهینه نیست.
  - **`Multiprocessing`**: برای پردازش‌های **CPU-bound** به دلیل اینکه هر فرآیند فضای حافظه مستقل دارد، مناسب‌تر است.
  - **`Threading`**: برای پردازش‌های **CPU-bound** به دلیل وجود **Global Interpreter Lock (GIL)** در پایتون، کارایی کمتری دارد.

نتیجه‌گیری

- **`async`** و **`await`** ابزارهای قدرتمندی برای نوشتن برنامه‌های غیرهمزمان در پایتون هستند.
- **`asyncio`** برای مدیریت عملیات‌های **I/O-bound** غیرهمزمان بسیار مناسب است و می‌تواند به طور موثری منابع را استفاده کند.
- برای **CPU-bound operations** از **Multiprocessing** و برای **I/O-bound** از **asyncio** استفاده کنید.