一文带你了解并发编程：线程、进程与协程

在 Python 中，**并发编程**让程序能够同时执行多个任务，显著提高效率。主要的并发方案包括【多线程】、【多进程】和【协程】。本文将深入浅出地介绍这些概念、适用场景，并提供优化后的代码示例，帮助你轻松掌握并发编程。

1. 进程与线程基础

1.1 进程与线程的关系

- **进程**：操作系统资源分配的最小单位，拥有独立的内存空间，各进程相互隔离。
- **线程**：CPU 调度的最小执行单位，同一进程内的多个线程共享该进程的资源。

形象比喻：进程好比一个工厂，线程则是工厂里的工人。多线程就像多个工人在同一工厂协作，多进程则是开设多个工厂并行工作。

1.2 GIL（全局解释器锁）

CPython 解释器的 GIL 机制使得即使启用多线程，同一时刻也只有一个线程在执行 Python 字节码。因此：

- **计算密集型任务**（大量数学运算、数据处理）更适合使用多进程，充分利用多核 CPU。
- **IO 密集型任务**（文件读写、网络请求）适合使用多线程或协程，因为在等待 IO 时可以切换到其他任务。

2. 多线程编程

多线程特别适合处理 IO 密集型任务，例如并行下载多个文件。下面展示一个优化的多线程下载示例：

```
import threading
import requests
import time
import logging
from concurrent.futures import ThreadPoolExecutor
from typing import List, Tuple
# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(threadName)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class DownloadManager:
    """文件下载管理器，支持多线程并行下载"""
    def __init__(self, max_workers: int = 5):
        """
        初始化下载管理器
        参数:
            max_workers: 最大工作线程数
        """
        self.max_workers = max_workers
        # 使用线程池管理线程资源
        self.executor = ThreadPoolExecutor(max_workers=max_workers, thread_name_prefix="Downloader")

    def download_file(self, file_name: str, url: str, timeout: int = 30) -> bool:
        """
        下载单个文件的方法
```

```
    下载单个文件的方法

    参数:
        file_name: 保存的文件名
        url: 下载文件的URL地址
        timeout: 请求超时时间（秒）
    返回:
        bool: 下载是否成功
    """
        try:
        logger    .info(f"开始下载 {file_name}")
        start_time     = time.time()

            # 添加请求头模拟浏览器行为
        headers     = {
                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
            }
            # 使用超时和流式下载处理大文件
            with requests.get(url, headers=headers, stream=True, timeout=timeout) as response:
            response    .raise_for_status()  # 确保请求成功
                # 获取文件大小（如果服务器提供）
            total_size     = int(response.headers.get('content-length', 0))

                with open(file_name, 'wb') as f:
                    # 分块下载，避免一次性加载大文件到内存
                chunk_size     = 8192  # 8KB
                downloaded     = 0
                    for chunk in response.iter_content(chunk_size=chunk_size):
                        if chunk:  # 过滤保持连接活跃的空块
                        f    .write(chunk)
                        downloaded     += len(chunk)

                            # 记录下载进度
                            if total_size > 0:
                            progress     = (downloaded / total_size) * 100
                                if downloaded % (5 * chunk_size) == 0:  # 每下载约40KB更新一次进度
                                logger    .debug(f"{file_name} - 下载进度: {progress:.1f}%")

        elapsed     = time.time() - start_time
        logger    .info(f"{file_name} 下载完成 - 耗时: {elapsed:.2f}秒")
            return True

        except requests.exceptions.RequestException as e:
        logger    .error(f"下载 {file_name} 失败: {str(e)}")
            return False
        except IOError as e:
        logger    .error(f"文件处理错误 {file_name}: {str(e)}")
            return False
        except Exception as e:
        logger    .error(f"下载 {file_name} 时发生未知错误: {str(e)}")
            return False

    def download_multiple(self, url_list: List[Tuple[str, str]]) -> List[bool]:
        """
    并行下载多个文件

    参数:
        url_list: 包含(文件名, URL)元组的列表
```

```
        返回:
            List[bool]: 每个下载任务的成功状态列表
        """
        logger.info(f"开始下载 {len(url_list)} 个文件，使用 {self.max_workers} 个线程")

        # 提交所有下载任务到线程池
        futures = [
            self.executor.submit(self.download_file, file_name, url)
            for file_name, url in url_list
        ]

        # 等待所有任务完成并收集结果
        results = []
        for future in futures:
            try:
                result = future.result()
                results.append(result)
            except Exception as e:
                logger.error(f"执行任务时发生异常: {str(e)}")
                results.append(False)

        # 统计成功率
        success_count = sum(results)
        logger.info(f"所有下载任务完成! 成功: {success_count}/{len(url_list)}")

        return results

    def shutdown(self):
        """关闭线程池，释放资源"""
        self.executor.shutdown(wait=True)
        logger.info("下载管理器已关闭")

# 使用示例
if __name__ == "__main__":
    # 测试URL列表 (替换为实际可用的URL)
    url_list = [
        ("video1.mp4", "https://example.com/video1"),
        ("video2.mp4", "https://example.com/video2"),
        ("video3.mp4", "https://example.com/video3")
    ]
    # 创建下载管理器并开始下载
    downloader = DownloadManager(max_workers=3)
    try:
        results = downloader.download_multiple(url_list)
    finally:
        # 确保资源被正确释放
        downloader.shutdown()
```

线程管理的关键点

- **线程池**：使用 `ThreadPoolExecutor` 管理线程资源，避免频繁创建和销毁线程的开销。
- **异常处理**：对每个下载任务进行完善的异常处理，确保单个任务失败不会影响整体程序。
- **资源管理**：通过 `with` 语句和 `shutdown()` 方法确保资源正确释放。
- **日志记录**：使用 `logging` 模块替代简单的 `print`，方便调试和问题追踪。

线程锁保证数据安全

当多个线程操作共享数据时，需要使用锁机制防止数据竞争问题：

```
import threading
import logging
import time
from typing import List
# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(threadName)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class ThreadSafeCounter:
    """线程安全的计数器实现"""
    def __init__(self, initial_value: int = 0):
        """
        初始化计数器
        参数:
            initial_value: 计数器初始值
        """
        self._value = initial_value
        self._lock = threading.RLock()  # 可重入锁，支持同一线程多次获取
    def increment(self, amount: int = 1) -> int:
        """
        增加计数器值并返回新值
        参数:
            amount: 增加的数量

        返回:
            int: 增加后的计数器值
        """
            with self._lock:
            self._value += amount
            current = self._value
            return current

    def decrement(self, amount: int = 1) -> int:
        """
        减少计数器值并返回新值

        参数:
            amount: 减少的数量
        返回:
            int: 减少后的计数器值
        """
            with self._lock:
            self._value -= amount
            current = self._value
            return current
    @property
    def value(self) -> int:
        """
        获取当前计数器值
```

```
        返回:
            int: 当前计数器值
        """
            with self._lock:
                return self._value

def     worker(counter: ThreadSafeCounter, iterations: int, worker_id: int):
        """
    工作线程函数，执行指定次数的计数器增加操作
    参数:
        counter: 线程安全的计数器对象
        iterations: 迭代次数
        worker_id: 工作线程ID
    """
    logger    .info(f"工作线程 {worker_id} 开始执行")

        for i in range(iterations):
            # 模拟一些随机工作量
            if i % 10000 == 0:
            logger    .debug(f"工作线程 {worker_id} 已完成 {i} 次操作")
            # 增加计数器
        counter    .increment()

    logger    .info(f"工作线程 {worker_id} 完成，共执行 {iterations} 次操作")

def     run_threaded_counter_test(num_threads: int = 5, iterations_per_thread: int = 100000):
        """
    运行多线程计数器测试
    参数:
        num_threads: 线程数量
        iterations_per_thread: 每个线程执行的迭代次数
    """
        # 创建线程安全计数器
    counter     = ThreadSafeCounter()

        # 创建线程列表
    threads    : List[threading.Thread] = []
    logger    .info(f"开始测试: {num_threads} 个线程，每个线程 {iterations_per_thread} 次操作")
    start_time     = time.time()

        # 创建并启动所有线程
        for i in range(num_threads):
        t     = threading.Thread(
            target    =worker,
            args    =(counter, iterations_per_thread, i),
            name    =f"Worker-{i}"
              )
        threads     .append(t)
        t     .start()

          # 等待所有线程完成
          for t in threads:
        t     .join()

    elapsed      = time.time() - start_time

        # 验证结果
```

```
    expected       = num_threads * iterations_per_thread
    actual       = counter.value

    logger      .info(f"测试完成! 耗时: {elapsed:.2f}秒")
    logger      .info(f"计数器最终值: {actual}")
    logger      .info(f"期望值: {expected}")
    logger      .info(f"结果{'正确' if actual == expected else '不正确'}")

if      __name__ == "__main__":
    run_threaded_counter_test      (num_threads=5, iterations_per_thread=100000)
```

3. 多进程编程

多进程能够绕过 GIL 限制，充分利用多核 CPU，特别适合计算密集型任务。以下是一个优化的多进程计算示例：

```
import    multiprocessing as mp
import    time
import    logging
import    os
import    math
from    typing import List, Tuple, Dict, Any
# 配置日志
logging   .basicConfig(
    level   =logging.INFO,
    format     ='%(asctime)s - %(processName)s - %(levelname)s - %(message)s'
)
logger      = logging.getLogger(__name__)

class      ParallelProcessor:
        """并行任务处理器，基于多进程实现"""
        def __init__(self, num_processes: int = None):
            """
        初始化并行处理器
        参数:
            num_processes: 进程数量，默认为CPU核心数
        """
            # 若未指定进程数，则使用CPU核心数
        self     .num_processes = num_processes or mp.cpu_count()
        logger     .info(f"初始化并行处理器，使用 {self.num_processes} 个进程")

        def _worker_calc_partial_sum(self, task_id: int, start: int, end: int,
                                     result_queue     : mp.Queue) -> None:
            """
        工作进程函数：计算部分和
        参数:
            task_id: 任务ID
            start: 起始值（包含）
            end: 结束值（不包含）
            result_queue: 结果队列
        """
            try:
            process_id      = os.getpid()
            logger     .info(f"任务 {task_id} 开始于进程 {process_id}，计算范围: [{start}, {end})")
                # 用数学公式计算区间和，比循环更高效
                # sum(range(start, end)) = (end-1 + start) * (end - start) / 2
```

```
    result      = (end - 1 + start) * (end - start) // 2
        # 也可以用内置sum函数，但在特大范围时可能效率较低
        # result = sum(range(start, end))

        # 将结果放入队列
    result_queue     .put((task_id, result))
    logger     .info(f"任务 {task_id} 完成，结果: {result}")

     except Exception as e:
    logger     .error(f"任务 {task_id} 出错: {str(e)}")
        # 放入错误结果
    result_queue     .put((task_id, None))
 def calculate_sum(self, start: int, end: int) -> int:
     """
并行计算从start到end-1的整数和
参数:
    start: 起始值（包含）
    end: 结束值（不包含）
返回:
    int: 计算结果
"""
     if end <= start:
         return 0
     # 创建通信队列
result_queue      = mp.Queue()

     # 计算每个进程的工作量
total_range      = end - start
chunk_size      = math.ceil(total_range / self.num_processes)

     # 创建进程列表
processes      = []
logger     .info(f"开始计算从 {start} 到 {end-1} 的和，分为 {self.num_processes} 个子任务")
start_time      = time.time()

     # 创建并启动所有进程
     for i in range(self.num_processes):
    task_start      = start + i * chunk_size
    task_end      = min(task_start + chunk_size, end)
         # 若已超出范围，跳过创建进程
         if task_start >= end:
             continue
    p      = mp.Process(
        target     =self._worker_calc_partial_sum,
        args     =(i, task_start, task_end, result_queue),
        name     =f"Calculator-{i}"
        )
    processes     .append(p)
    p     .start()
     # 收集结果
results      = {}
     for _ in range(len(processes)):
    task_id     , value = result_queue.get()
         if value is not None:
        results     [task_id] = value

     # 等待所有进程结束
```

```
            # 等待所有进程结束
            for p in processes:
                p.join()

            # 检查是否所有任务都成功完成
            if len(results) != len(processes):
                logger.warning(f"部分任务失败! 只收到 {len(results)}/{len(processes)} 个结果")

            # 计算总和
            total_sum = sum(results.values())
            elapsed = time.time() - start_time
            logger.info(f"计算完成! 耗时: {elapsed:.2f}秒")
            logger.info(f"结果: {total_sum}")
            return total_sum

# 验证函数，确认并行计算结果正确性
def verify_sum(start: int, end: int, result: int) -> bool:
    """验证计算结果是否正确"""
    # 使用数学公式计算正确答案
    expected = (end - 1 + start) * (end - start) // 2
    logger.info(f"验证结果 - 计算值: {result}, 期望值: {expected}")
    return result == expected

if __name__ == "__main__":
    # 计算从0到1亿的和
    START = 0
    END = 100_000_000

    processor = ParallelProcessor()
    result = processor.calculate_sum(START, END)
    # 验证结果
    is_correct = verify_sum(START, END, result)
    print(f"\n计算结果: {result}")
    print(f"结果验证: {'✓ 正确' if is_correct else '✗ 错误'}")
```

进程间数据共享

不同进程间内存空间隔离，数据不能直接共享。可通过以下方式实现共享：

```
import multiprocessing as mp
import logging
import time
from typing import Dict, List, Any
# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(processName)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class SharedDataProcessor:
    """演示多进程间数据共享的处理器"""
    def __init__(self):
        """初始化处理器"""
        logger.info("初始化共享数据处理器")
    def using_value_and_array(self):
```

```
    """使用共享内存Value和Array示例"""
    # 创建共享整数和数组
shared_counter = mp.Value('i', 0)  # 'i'表示有符号整数
shared_array = mp.Array('d', [0.0] * 5)  # 'd'表示双精度浮点数
processes = []
    # 定义工作函数
    def worker(counter, array, worker_id):
    pid = mp.current_process().pid
    logger.info(f"工作进程 {worker_id} (PID: {pid}) 开始")

        # 修改共享计数器
        with counter.get_lock():
        counter.value += 1
        logger.info(f"工作进程 {worker_id} 增加计数器值到 {counter.value}")
        # 修改共享数组
        with array.get_lock():
            for i in range(len(array)):
            array[i] = array[i] + worker_id
        logger.info(f"工作进程 {worker_id} 更新数组: {list(array)}")
    logger.info(f"工作进程 {worker_id} 完成")
    # 创建并启动进程
    for i in range(3):
    p = mp.Process(target=worker, args=(shared_counter, shared_array, i))
    processes.append(p)
    p.start()
    # 等待进程完成
    for p in processes:
    p.join()
logger.info(f"所有进程完成。最终计数器值: {shared_counter.value}")
logger.info(f"最终数组值: {list(shared_array)}")

def using_manager(self):
    """使用 Manager 共享复杂数据结构示例"""
    # 创建Manager对象
    with mp.Manager() as manager:
        # 创建共享字典和列表
    shared_dict = manager.dict()
    shared_list = manager.list()

        # 创建共享锁
    lock = manager.RLock()
    processes = []
        # 定义工作函数
        def worker(shared_dict, shared_list, lock, worker_id):
        pid = mp.current_process().pid
        logger.info(f"工作进程 {worker_id} (PID: {pid}) 开始")

            # 安全地修改共享字典
            with lock:
            shared_dict[f'key_{worker_id}'] = worker_id * worker_id
            shared_dict['total'] = shared_dict.get('total', 0) + worker_id
            logger.info(f"工作进程 {worker_id} 更新字典: {dict(shared_dict)}")

            # 安全地修改共享列表
            with lock:
            shared_list.append(f"来自进程 {worker_id} 的数据")
            logger.info(f"工作进程 {worker_id} 更新列表: {list(shared_list)}")
```

```
            logger.info(f"工作进程 {worker_id} 更新列表: {list(shared_list)}")
            # 模拟一些工作
            time.sleep(0.5)
            logger.info(f"工作进程 {worker_id} 完成")
        # 创建并启动进程
        for i in range(5):
            p = mp.Process(
                target=worker,
                args=(shared_dict, shared_list, lock, i),
                name=f"Manager-Worker-{i}"
            )
            processes.append(p)
            p.start()
        # 等待进程完成
        for p in processes:
            p.join()
        logger.info("所有进程完成")
        logger.info(f"最终共享字典: {dict(shared_dict)}")
        logger.info(f"最终共享列表: {list(shared_list)}")

if __name__ == "__main__":
    processor = SharedDataProcessor()
    print("\n=== 使用 Value 和 Array 共享数据 ===")
    processor.using_value_and_array()

    print("\n=== 使用 Manager 共享数据 ===")
    processor.using_manager()
```

进程池的最佳实践

对于需要并行处理的大量独立任务，进程池是更高效的选择：

```
import multiprocessing as mp
from concurrent.futures import ProcessPoolExecutor
import time
import logging
import random
from typing import List, Tuple, Any
# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(processName)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class AdvancedTaskProcessor:
    """高级任务处理器，演示进程池最佳实践"""
    def __init__(self, max_workers: int = None):
        """
        初始化处理器
        参数:
            max_workers: 最大工作进程数，默认为CPU核心数
        """
        self.max_workers = max_workers or mp.cpu_count()
        logger.info(f"初始化高级任务处理器，最大进程数: {self.max_workers}")
```

```
def process_single_task(self, task_data: Tuple[int, int]) -> Tuple[int, float]:
    """
处理单个任务的函数
参数:
    task_data: 任务数据，格式为 (任务ID, 任务复杂度)

返回:
    Tuple[int, float]: (任务ID, 处理结果)
"""
task_id    , complexity = task_data
process_id     = mp.current_process().pid

logger    .info(f"进程 {process_id} 开始处理任务 {task_id}，复杂度: {complexity}")

    # 模拟计算密集型工作
start_time     = time.time()
    # 根据任务复杂度决定计算量
iterations     = complexity * 1000000
result     = 0
    for i in range(iterations):
    result     += i % 10
        # 避免过度优化
        if i % 1000000 == 0:
        result     = result * 0.99999

    # 计算处理时间
processing_time     = time.time() - start_time
logger    .info(f"进程 {process_id} 完成任务 {task_id}，耗时: {processing_time:.2f}秒")
    return task_id, result
def process_tasks_with_pool(self, tasks: List[Tuple[int, int]]) -> List[Tuple[int, float]]:
    """
使用进程池处理多个任务
参数:
    tasks: 任务列表，每个任务为 (任务ID, 任务复杂度) 元组

返回:
    List[Tuple[int, float]]: 处理结果列表
"""
logger    .info(f"开始处理 {len(tasks)} 个任务，使用 {self.max_workers} 个进程")
start_time     = time.time()

results     = []

    # 使用ProcessPoolExecutor进行并行处理
    with ProcessPoolExecutor(max_workers=self.max_workers) as executor:
        # 提交所有任务
    futures     = [executor.submit(self.process_single_task, task) for task in tasks]
        # 收集结果
        for future in futures:
            try:
            result     = future.result()
            results    .append(result)
             except Exception as e:
            logger    .error(f"任务执行出错: {str(e)}")
elapsed     = time.time() - start_time
logger    .info(f"所有任务处理完成，总耗时: {elapsed:.2f}秒")

    return results
```

```
    def run_demo(self, num_tasks: int = 10, seed: int = 42):
        """
    运行演示

    参数:
        num_tasks: 任务数量
        seed: 随机种子，确保可重复性
    """
    random.seed(seed)

        # 创建任务列表，复杂度在1到5之间随机
    tasks = [(i, random.randint(1, 5)) for i in range(num_tasks)]
    logger.info(f"创建了 {num_tasks} 个任务")
    logger.info(f"任务列表: {tasks}")

        # 处理任务
    results = self.process_tasks_with_pool(tasks)
        # 输出结果
    logger.info(f"处理结果: {results}")

if __name__ == "__main__":
    processor = AdvancedTaskProcessor()
    processor.run_demo(num_tasks=8)
```

4. 协程编程（续）

基于 asyncio 的协程示例（续）

```
import asyncio
import logging
import time
import random
from typing import List, Dict, Any

# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class AsyncTaskManager:
    """基于协程的异步任务管理器"""

    def __init__(self):
        """初始化异步任务管理器"""
    logger.info("初始化异步任务管理器")
    self.results = {}
    async def task_simulator(self, task_id: int, duration: float) -> Dict[str, Any]:
        """
    模拟一个异步任务
    参数:
        task_id: 任务ID
        duration: 模拟的任务执行时间（秒）
    返回:
```

```
        Dict: 任务结果数据
    """
    logger.info(f"任务 {task_id} 开始，预计耗时 {duration:.2f}秒")

        # 模拟不同任务阶段
    stages = ['初始化', '数据加载', '处理中', '完成']
    result_data = {
        'task_id': task_id,
        'stages': {},
        'total_time': 0
    }
    start_time = time.time()
        # 分阶段执行任务
    stage_time = duration / len(stages)
    for i, stage in enumerate(stages):
        # 模拟每个阶段的工作
        stage_start = time.time()
        # 使用asyncio.sleep模拟异步IO操作
        variation = random.uniform(0.8, 1.2)  # 增加一些随机性
        await asyncio.sleep(stage_time * variation)
        stage_duration = time.time() - stage_start
        result_data['stages'][stage] = stage_duration

        logger.debug(f"任务 {task_id}: {stage} 阶段完成，耗时 {stage_duration:.2f}秒")

        # 计算总耗时
    total_time = time.time() - start_time
    result_data['total_time'] = total_time

    logger.info(f"任务 {task_id} 完成，总耗时 {total_time:.2f}秒")
        return result_data
async def run_tasks(self, num_tasks: int = 5, min_duration: float = 1.0,
                    max_duration: float = 5.0) -> List[Dict[str, Any]]:
        """
    并发运行多个异步任务

    参数:
        num_tasks: 任务数量
        min_duration: 最小任务持续时间
        max_duration: 最大任务持续时间
    返回:
        List[Dict]: 所有任务结果列表
    """
    logger.info(f"准备运行 {num_tasks} 个异步任务")

        # 生成随机任务持续时间
    durations = [random.uniform(min_duration, max_duration) for _ in range(num_tasks)]
        # 创建任务列表
    tasks = [
        self.task_simulator(i, durations[i]) for i in range(num_tasks)
        ]

        # 并发执行所有任务
    start_time = time.time()
    results = await asyncio.gather(*tasks)
    total_time = time.time() - start_time
        # 分析结果
```

```
        theoretical_sequential_time = sum(durations)
        speedup = theoretical_sequential_time / total_time
        logger.info(f"所有任务完成! 总耗时: {total_time:.2f}秒")
        logger.info(f"如果顺序执行预计需要: {theoretical_sequential_time:.2f}秒")
        logger.info(f"加速比: {speedup:.2f}x")
            return results

async def main():
    """主函数"""
    manager = AsyncTaskManager()

    # 运行并发任务
    results = await manager.run_tasks(num_tasks=10, min_duration=1.0, max_duration=3.0)
    # 输出结果摘要
    print("\n任务执行摘要:")
    for result in results:
        task_id = result['task_id']
        total_time = result['total_time']
            print(f"任务 {task_id}: 总耗时 {total_time:.2f}秒")

if __name__ == "__main__":
    # 在Python 3.7+中，可以直接使用asyncio.run()
    asyncio.run(main())
```

使用 aiohttp 进行高效网络请求

协程在网络 IO 场景中特别高效。下面是使用 aiohttp 实现并发下载多个文件的示例：

```
import asyncio
import aiohttp
import logging
import time
import os
from typing import List, Tuple, Dict, Any
from aiohttp import ClientSession
import aiofiles  # 需要先安装: pip install aiofiles
# 配置日志
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class AsyncDownloader:
    """异步HTTP下载器，基于aiohttp和asyncio实现"""
    def __init__(self, download_dir: str = "./downloads", chunk_size: int = 8192):
        """
        初始化下载器
        参数:
            download_dir: 下载文件保存目录
            chunk_size: 下载时的块大小（字节）
        """
        self.download_dir = download_dir
        self.chunk_size = chunk_size

        # 确保下载目录存在
```

```
        os.makedirs(download_dir, exist_ok=True)
        logger.info(f"异步下载器初始化完成，下载目录: {download_dir}")
    async def download_file(self, session: ClientSession, url: str,
                            file_name: str) -> Dict[str, Any]:
        """
        异步下载单个文件

        参数:
            session: aiohttp会话对象
            url: 下载URL
            file_name: 保存的文件名
        返回:
            Dict: 下载结果信息
        """
        file_path = os.path.join(self.download_dir, file_name)
        start_time = time.time()
        # 记录下载信息
        result = {
            'url': url,
            'file_name': file_name,
            'file_path': file_path,
            'success': False,
            'file_size': 0,
            'download_time': 0,
            'speed': 0,
            'error': None
        }
        try:
            logger.info(f"开始下载: {file_name} 从 {url}")

            # 发送请求并流式下载
            async with session.get(url, ssl=False) as response:
                if response.status != 200:
                    error_msg = f"HTTP错误: {response.status} - {response.reason}"
                    logger.error(error_msg)
                    result['error'] = error_msg
                    return result

                # 获取文件大小（如果服务器提供）
                content_length = response.headers.get('Content-Length')
                if content_length:
                    total_size = int(content_length)
                    result['file_size'] = total_size
                    logger.info(f"{file_name} 大小: {total_size/1024:.1f} KB")
                # 异步写入文件
                async with aiofiles.open(file_path, 'wb') as f:
                    downloaded = 0
                    last_log_time = time.time()

                    # 流式下载，避免内存溢出
                    async for chunk in response.content.iter_chunked(self.chunk_size):
                        await f.write(chunk)
                        downloaded += len(chunk)
                        # 每秒最多记录一次进度
                        current_time = time.time()
                        if current_time - last_log_time >= 1.0 and total_size > 0:
```

```
                    progress = (downloaded / total_size) * 100
                    logger.debug(f"{file_name} - 下载进度: {progress:.1f}%")
                    last_log_time = current_time

            # 计算下载统计信息
        download_time = time.time() - start_time
        speed = result['file_size'] / download_time if download_time > 0 else 0

        result.update({
            'success': True,
            'download_time': download_time,
            'speed': speed
        })

        logger.info(f"{file_name} 下载完成 - "
                f"耗时: {download_time:.2f}秒, "
                f"速度: {speed/1024:.1f} KB/s")

            return result

        except aiohttp.ClientError as e:
        error_msg = f"网络错误: {str(e)}"
        logger.error(f"下载 {file_name} 失败: {error_msg}")
        result['error'] = error_msg
            return result

        except asyncio.CancelledError:
        error_msg = "下载被取消"
        logger.warning(f"{file_name} 下载被取消")
        result['error'] = error_msg
            return result

        except Exception as e:
        error_msg = f"未知错误: {str(e)}"
        logger.error(f"下载 {file_name} 时发生异常: {error_msg}")
        result['error'] = error_msg
            return result

async def download_batch(self, urls: List[Tuple[str, str]],
                         max_concurrent: int = 5) -> List[Dict[str, Any]]:
        """
    批量并发下载多个文件
    参数:
        urls: 包含(文件名, URL)元组的列表
        max_concurrent: 最大并发下载数
    返回:
        List[Dict]: 下载结果列表
    """
        if not urls:
        logger.warning("没有提供下载URL")
            return []

    logger.info(f"开始批量下载 {len(urls)} 个文件，最大并发数: {max_concurrent}")
    start_time = time.time()
        # 创建限制并发数的信号量
    semaphore = asyncio.Semaphore(max_concurrent)

        # 定义带信号量的下载函数
```

```
        async def bounded_download(session, file_name, url):
            async with semaphore:
                return await self.download_file(session, url, file_name)
        # 创建会话并发送请求
        async with aiohttp.ClientSession() as session:
            # 创建所有下载任务
            tasks = [
                bounded_download(session, file_name, url)
                for file_name, url in urls
            ]
            # 并发执行所有任务
            results = await asyncio.gather(*tasks, return_exceptions=True)

            # 处理结果
        processed_results = []
        success_count = 0

            for result in results:
                # 检查是否发生异常
                if isinstance(result, Exception):
                logger.error(f"下载任务异常: {str(result)}")
                processed_results.append({
                    'success': False,
                    'error': str(result)
                })
                else:
                processed_results.append(result)
                    if result['success']:
                    success_count += 1
            # 统计信息
        total_time = time.time() - start_time
        success_rate = (success_count / len(urls)) * 100 if urls else 0
        logger.info(f"批量下载完成! 总耗时: {total_time:.2f}秒")
        logger.info(f"成功: {success_count}/{len(urls)} ({success_rate:.1f}%)")

            return processed_results

async def main():
        """主函数"""
        # 创建下载器实例
    downloader = AsyncDownloader(download_dir="./downloads")

        # 准备下载列表（替换为实际可用的URL）
    urls = [
            ("image1.jpg", "https://example.com/image1.jpg"),
            ("image2.jpg", "https://example.com/image2.jpg"),
            ("image3.jpg", "https://example.com/image3.jpg"),
            ("document1.pdf", "https://example.com/document1.pdf"),
            ("document2.pdf", "https://example.com/document2.pdf")
        ]
        # 开始批量下载
    results = await downloader.download_batch(urls, max_concurrent=3)

        # 打印下载结果摘要
        print("\n下载结果摘要:")
        for result in results:
```

```
            if result.get('success'):
                file_name = result.get('file_name')
                size_kb = result.get('file_size', 0) / 1024
                time_s = result.get('download_time', 0)
                speed_kbps = result.get('speed', 0) / 1024
                print(f" {file_name}: {size_kb:.1f} KB, {time_s:.2f}秒, {speed_kbps:.1f} KB/s")
            else:
                url = result.get('url', 'Unknown URL')
                error = result.get('error', 'Unknown error')
                print(f" {url}: {error}")

if __name__ == "__main__":
    # 在Python 3.7+中可以直接使用asyncio.run()
    asyncio.run(main())
```

5. 并发模式选择指南

根据任务类型选择合适的并发模式非常重要，下面是选择指南：

5.1 多线程适用场景

- **IO 密集型任务**：文件读写、网络请求、数据库操作
- **需要共享内存**：线程间可直接共享变量和对象
- **与外部库集成**：许多第三方库不支持异步，但支持多线程

5.2 多进程适用场景

- **CPU 密集型任务**：数学计算、图像处理、数据分析
- **需要隔离安全**：各进程独立运行，一个进程崩溃不影响其他进程
- **充分利用多核 CPU**：绕过 GIL 限制，实现真正的并行计算

5.3 协程适用场景

- **高并发 IO 操作**：网络爬虫、API 调用、微服务通信
- **资源消耗敏感**：协程开销远小于线程和进程
- **需要精细控制任务调度**：协程可以精确控制任务切换时机

6. 总结与最佳实践

6.1 主要概念对比

特性	多线程	多进程	协程
适用任务类型	IO 密集型	CPU 密集型	IO 密集型
资源消耗	中等	高	低
数据共享	直接共享	需特殊机制	直接共享
并行执行	受 GIL 限制	真正并行	单线程内并发
开发复杂度	中等	中等	较高（需理解异步）
切换开销	中等	高	极低

6.2 并发编程最佳实践

1. **明确问题类型**：首先确定任务是 CPU 密集型还是 IO 密集型，再选择并发方案。
2. **避免共享状态**：尽量减少线程/进程间的数据共享，必要时使用锁或消息传递。
3. **合理使用池化**：使用线程池、进程池管理资源，避免频繁创建销毁的开销。
4. **善用异步库**：使用协程时，优先选择支持异步 IO 的库（如 aiohttp、asyncpg）。
5. **异常处理**：确保捕获并妥善处理所有子任务中的异常，防止静默失败。
6. **超时控制**：设置合理的超时时间，避免任务无限等待。
7. **监控与日志**：实现良好的日志记录，便于调试和性能分析。
8. **资源管理**：正确释放资源，使用上下文管理器（ `with` 语句）自动清理。