



School of Economics and Management, Beihang University

# 现代程序设计技术

赵吉昌

[jichang@buaa.edu.cn](mailto:jichang@buaa.edu.cn)

- 面向对象编程
  - 异步IO
  - 协程

- 同步IO

- 在IO过程中当前线程被挂起，当前线程其他需要CPU计算的代码无法执行

- 一般的io是同步的

- 多线程可解决该问题

- 计算和IO任务可以由不同的线程负责

- 但会带来线程创建、切换的成本，而且线程数不能无上限地增加

- 异步IO

- 当前线程只发出IO指令，但不等待其执行结束，而是先执行其他代码，避免线程因IO操作而阻塞

- 事件驱动模型

- 一种编程范式，程序执行流由外部事件决定
- 包含一个事件循环，当外部事件发生时使用**回调机制**来触发相应的处理
- 可能的实现机制
  - 每收到一个请求，创建一个新的进程来处理该请求；
  - 每收到一个请求，创建一个新的线程来处理该请求；
  - **每收到一个请求，放入一个事件列表让主进程通过非阻塞IO方式来处理请求**
- 一般场景
  - **当程序中有许多任务，任务之间高度独立（不需要互相通信或等待彼此等）**，并且在等待事件到来时，某些任务会阻塞

- 事件列表模型

- 主线程不断重复“读取请求-处理请求”这一过程
- 进行IO操作时相关代码只发出IO请求，不等待IO结果，然后直接结束本轮事件处理，进入下一轮事件处理
- 当IO操作完成后，将收到IO完成消息，在处理该消息时再获取IO操作结果
- 在发出IO请求到收到IO完成消息期间，主线程并不阻塞，而是在循环中继续处理其他消息
- 对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力

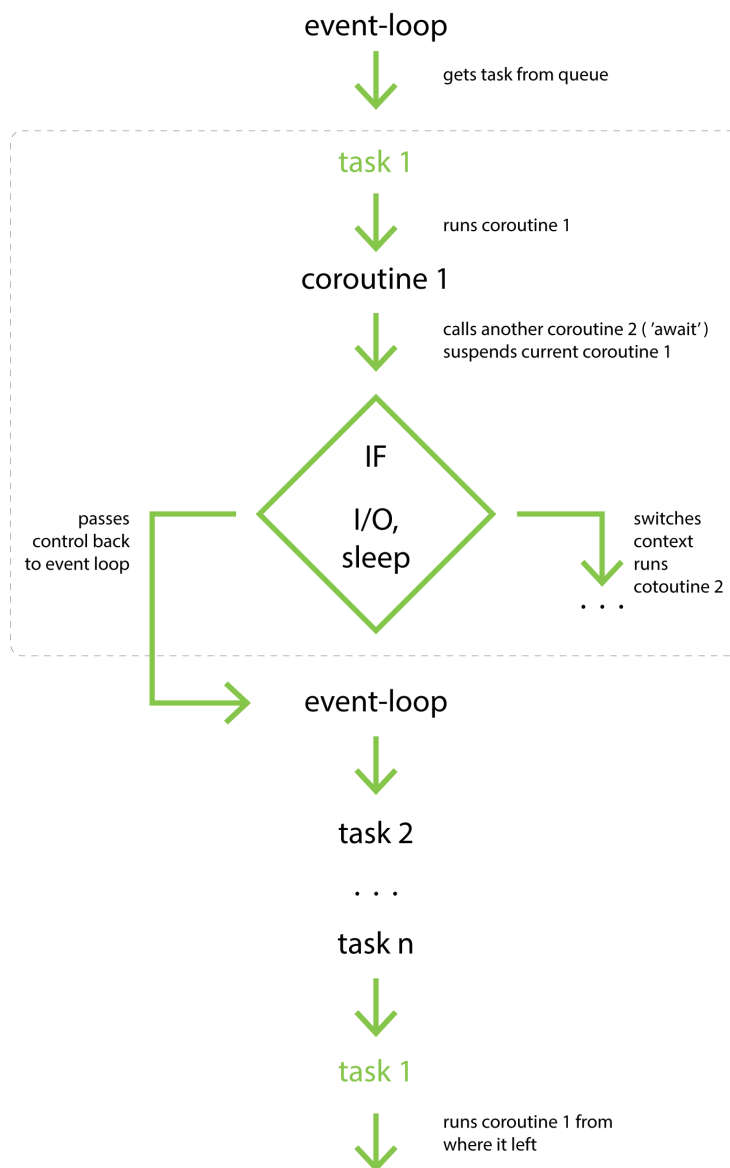
- 协程

- **Coroutine**, pseudo-thread, micro-thread
- 微线程
- 在一个线程中会有很多函数，一般将这些函数称为子程序，在子程序执行过程中可以**中断**去执行别的子程序，而别的子程序也可以**中断回来继续执行**之前的子程序，这个过程就称为协程
  - 执行函数A时，可以随时中断，进而执行函数B，然后中断B并继续执行A，且上述切换是自主可控的
  - **但上述过程并非函数调用（没有调用语句）**
- 表象上类似多线程，但**协程本质上只有一个线程**在运行

- Event Loop

- The event loop is running **in a thread**
- It gets tasks from the queue
- Each task calls the next step of a coroutine
- If coroutine calls another coroutine (await `<coroutine_name>`), **the current coroutine gets suspended and context switch occurs. Context of the current coroutine (variables, state) is saved and context of a called coroutine is loaded**
- If coroutine comes across a blocking code (I/O, sleep), the current coroutine gets suspended and **control is passed back to the event loop**
- Event loop gets next tasks from the queue 2, ...n
- Then the event loop goes back to task 1 from where it left off

- Event Loop





- 协程的优点

- 无需线程上下文切换的开销，协程避免了无意义的调度，由此可以提高性能
- 无需原子操作锁定及同步的开销
- 方便切换控制流，简化编程模型
  - 线程由操作系统调度，而协程则是在程序级别由程序员自己调度
- 高并发+高扩展性+低成本
  - 一个CPU可以支持上万协程
  - 在高并发场景下的差异会更突出

- 协程的缺点

- 程序员必须自己承担调度的责任
- 协程仅能提高IO密集型程序的效率，但对于CPU密集型程序无能为力
- Python3的相关生态还在不断成熟
- 在CPU密集型程序中要充分发挥CPU利用率**需  
要结合多进程和协程**

- 通过yield关键字实现协程
  - 生成器的send() 函数
    - 与next() 作用类似，但可以发送值给对应的yield 表达式
    - 支持外部程序与生成器的交互
  - next(g) 就相当于g.send(None)
  - 注意第一次调用next() 或send(None) 相当于启动生成器，不能使用send() 发送一个非None的值
    - 利用装饰器来解决该问题
    - 在装饰器中先调用一次next
  - Demo: ysend.py, dys.py, cpc.py, search.py

- 通过gevent实现协程
  - 基于greenlet
  - spawn构建新协程
  - monkey.pach\_all将第三方库标记为IO非阻塞
  - 通过协程池控制协程数目
  - Demo :
    - `geventc.py`
    - `gevent_urlhost.py`
    - `gevent_urllenth.py`
    - `geventc_pool.py`

- 通过`asyncio`实现协程
  - python3.4引入
  - 用`asyncio`提供的`@asyncio.coroutine`将任务标记为`coroutine`类型，然后在`coroutine`内部用`yield from`调用另一个`coroutine`实现异步操作
  - Python3.5开始引入了`async`和`await`进一步简化语法
    - 把`@asyncio.coroutine`替换为`async`
    - 把`yield from`替换为`await`
  - Python3.7进一步变化...

- 通过asyncio实现

- run/create\_task/await/gather/get\_event\_loop/run\_until\_complete

- Demo:

- `async.py`, `async_url.py` ( 过时写法, 仅示例 )
    - `async_run.py`, `async_gather.py`
    - `async_url2.py`
    - `async_ret.py`
    - `aiocrawl.py`

- 通过aiofiles实现文件的异步读写
  - `pip install aiofiles`
    - `async with aiofiles.open(path, mode='r') as f:`
      - `contents = await f.read()`
  - Demo: `aiof.py`

# 本周作业



- 协程有时被称为“微线程”，因为二者有类似的使用场景。和线程相比，协程占用资源更少、切换迅速，且实现简单（如协程是协作式调用，一般不用考虑资源的抢占，所以大部分情况下不需要通过同步原语来避免冲突）。通常协程被用在高并发的IO场景中，因此本周要求在第十二次作业的基础上，利用协程重写并扩充已有爬虫的功能
  - 注意：可以使用[aiohttp](#)，[aiofiles](#)等第三方库。