第十一次作业总结

- 共性问题
  - 文件读写进程不安全
    - 用多个reduce同时读写同一个文件不安全
  - 加锁的范围过大
    - 锁加在整个map的run函数范围内，导致进程实际上没有并行
  - reduce进程不建议采用size()作为队列已空的判断依据
    - 应采用在 队列末端加入一个flag量(比如None)作为判断

关于上次示例的一个问题

School of Economics ang Management, Beihang University

现代程序设计技术

赵吉昌
jichang@buaa.edu.cn

本周内容

- 面向对象编程
  - 异步IO
  - 协程

IO模型

- 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

异步IO

- 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

异步IO

- 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

异步IO-协程

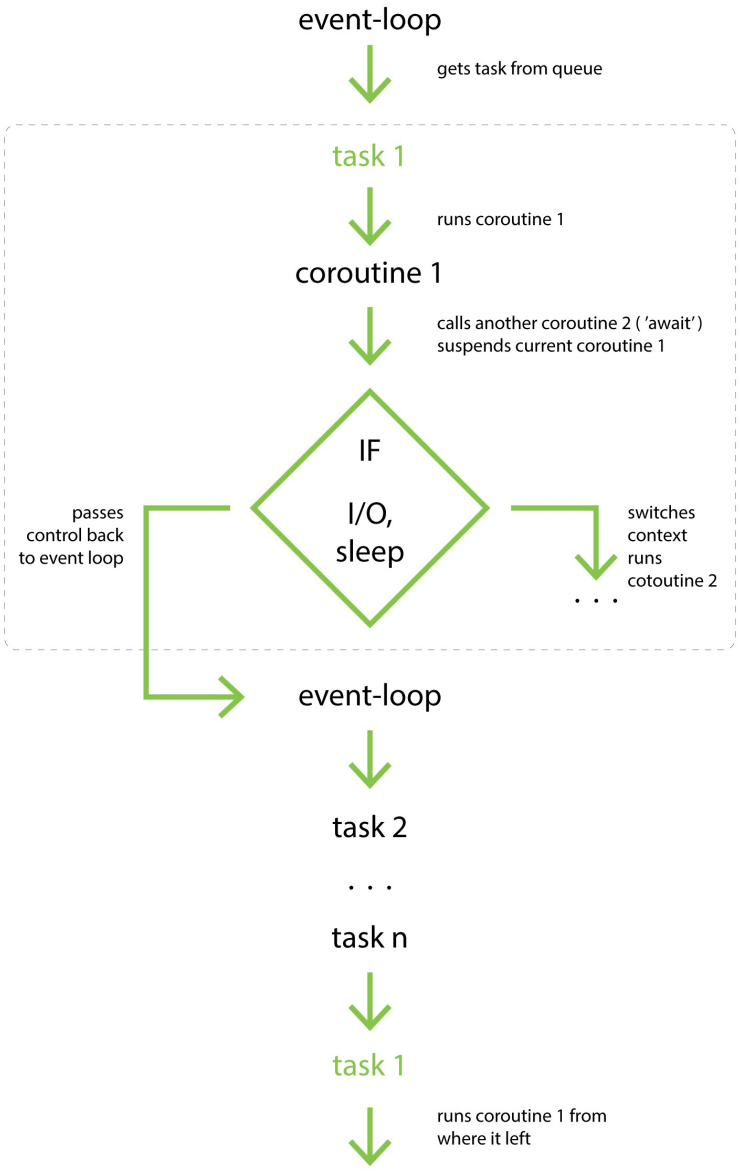
- 协程
  - Coroutine, peusdo-thread, micro-thread
  - 微线程
  - 在一个线程中会有很多函数，一般将这些函数称为子程序，在子程序执行过程中可以中断去执行别的子程序，而别的子程序也可以中断回来继续执行之前的子程序，这个过程就称为协程
    - 执行函数A时，可以随时中断，进而执行函数B，然后中断B并继续执行A，且上述切换是自主可控的
    - 但上述过程并非函数调用（没有调用语句）
  - 表象上类似多线程，但协程本质上只有一个线程在运行

异步IO-协程

- 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

异步IO-协程

- Event Loop



协程

- 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

- 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
  - Python2和Python3中实现有一定差别
    - 所用模块有区别
    - 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

协程的实现

- 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

协程的实现

- 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

协程的实现

- 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...

协程的实现

- 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

协程的实现

- 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，具体要求如下：
  - 1. 利用协程改写第十二次作业的歌单获取功能。
  - 2，对获取的歌单信息进行简单分析，筛选出想要进一步分析的歌单。这里可以设置分享数或播放数阈值来确定目标歌单。要求利用yield或aiofiles逐行地分析并返回筛选到的歌单url，以便进行歌单内音乐信息的爬取。
  - 3. 在歌单详细信息页面，利用协程实现访问并获取歌单前十首音乐的信息，如作者，所属专辑，播放量等。音乐页面url的构成是https://music.163.com/#/playlist?id=6813607109 ，其中"id"字段后是音乐id，该url可从歌单信息页面获取。
  - 注意：可以使用aiohttp，aiofiles等第三方库。