23. 并发程序

- 顺序程序和并发程序
- 并发程序的意义
- 并发的情况和历史发展

  实例，硬件，相关概念
- **Python并发库threading包**

  一些问题和实例，并发程序的一些现象
- 线程间通讯和**queue包**

顺序程序和并发程序

- 至今为止，我们用**Python**写出的程序都是顺序程序：在程序运行中只有一个活动着的执行进程
  - 一个语句执行完后，接着执行根据控制流要求的下一语句
  - 调用一个函数时，调用代码段的执行暂时停在函数调用点，被调用的函数开始执行。被调用函数执行完成后，调用代码段从调用点之后继续
  - 顺序执行是命令式程序（以操作命令作为基本计算元素的语言）的基本特性，**Python**程序也属于命令式程序
- 在并发程序执行中，可能同时出现多个独立的执行流
  - 这些执行流有独立的生命周期，有自己的开始和结束，一般称为执行进程（**process**）或活动（**activity**）
  - 每个进程执行自己的代码，也可能需要相互合作和传递信息

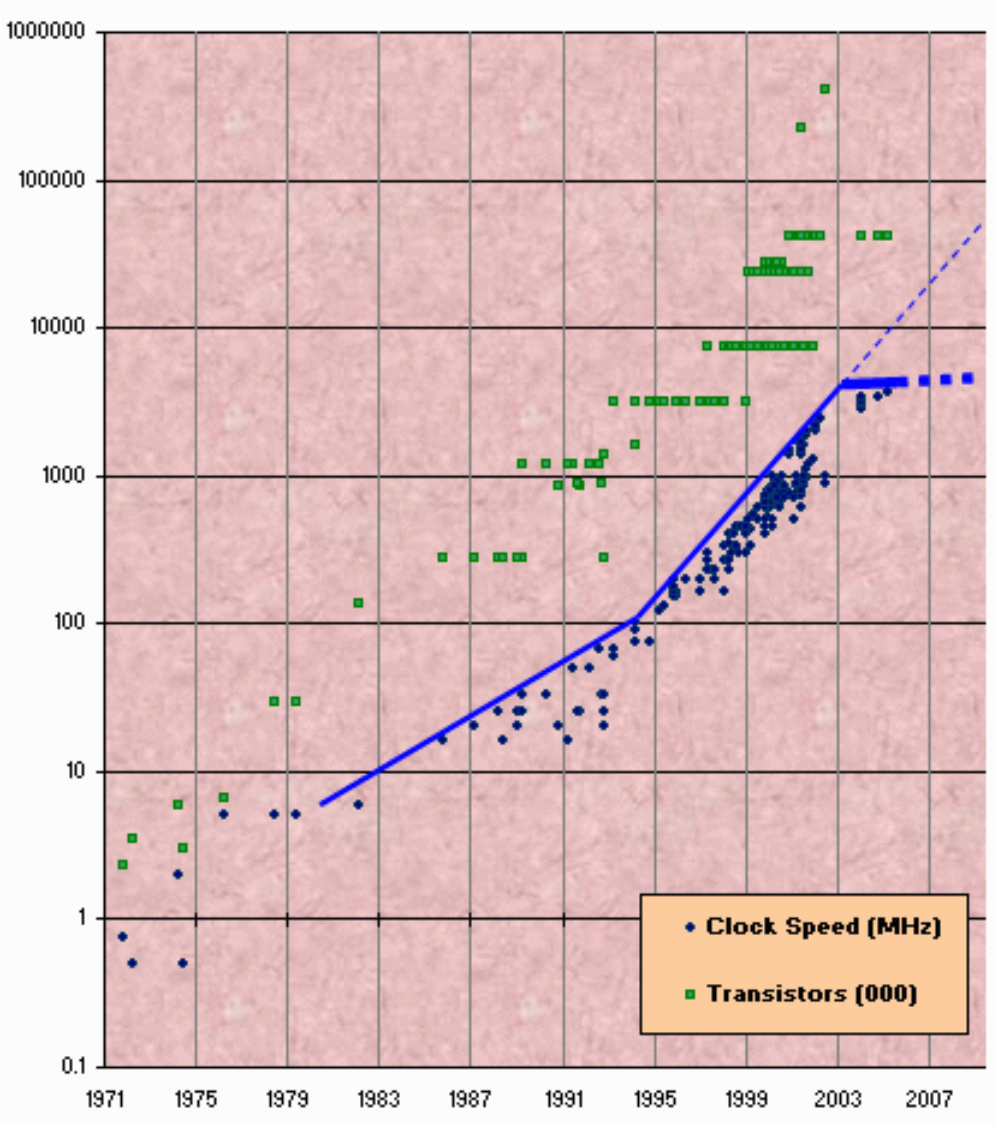
并发程序

- 需要并发程序和并发执行，主要基于下面一些理由：
  - 更好地反映问题的逻辑结构。许多程序里，特别是各种服务器、图形应用、计算机模拟等，需要做很多基本上相互独立的工作。构造这种程序，最简单也最合逻辑的方式，就是用一个独立的执行进程实现一个工作
  - 计算机硬件通常控制着一批独立设备和部件，例如键盘、显示器、打印机，还有各种内部组件。这些设备/部件的特性和操作速度不同，相应控制程序最好作为独立的进程，根据需要开始/结束或暂时中断。实时控制系统通常需要指挥控制多台独立的外部设备，最好用独立的控制线程，还需要与处理器上运行的其他线程交互，实现系统的整体行为
  - 通过并发执行有可能提高计算的性能。有些程序本质上并不需要并行执行，但如果能使用多个处理器同时计算，可能得到很大的速度提升，或者满足实际的需要

并发性

- 并发性的思想和技术已经开发多年
  - 并发程序设计的最重要的基础理论工作从 20 世纪 60 年代开始提出和研究，其基础理论框架在70年代已基本完成
  - Algol 68编程语言已经包含了与并发性有关的程序特征
  - 但对并发性的广泛兴趣是近年的新现象，主要原因是廉价的多处理器系统的开发和普及，以及图形、多媒体和互联网应用等的发展（用并发线程描述特别自然）等
- 今天，几乎所有硬件计算机都是多处理器计算机
  - 个人计算机（笔记本）都有2个或更多计算核心
  - 新型手机的CPU包含4个/8个或更多核心
  - 目前最快的天河二号计算机共包含32000颗Ivy Bridge处理器和48000颗Xeon Phei处理器，总计312万个计算核心

摩尔定律和多核CPU



- 通过提高主频提高计算机性能的时代已经结束。大约到 2003 年，常规CPU的主频增长雅然而止
- 新处理器的性能提高完全依靠片上多核技术
- 目前流行的高端商品处理器上单片处理器数为 8-16 核，研究或作为原型处理器有几十核到数千核，称为多核/众核处理器
- 多核发展方兴未艾，对并行程序设计技术和语言提出许多新要求

并发性的情况

- 计算机系统的许多不同层次都大量着出现各种并发活动
  - 在计算机基础逻辑电路层面，大量的动作都在并行地发生，硬件信号在芯片和电路中的大量连线上同时传播
  - CPU执行一条指令需要做一系列动作：取指令/指令解码/取数据/执行指令/保存数据等，新型处理器用一组执行部件分别完成这些动作，采用流水线方式并行工作
  - 专用向量处理器实现数据的并行处理；多处理器系统中很多进程在并行地运行；在互联网上一切事情都并发地发生
- 程序语言（或库）提供可见的并发性控制，程序员可以在代码中描述并发性，多处理器的机器可以利用这种并发性
  - 后面将简单介绍Python标准库的并发执行包
  - 说明如何写并发程序，展示并发程序的一些重要特征

并发性的简单历史

- 早期计算机是单用户机器，一个用户占用整个计算机硬件
- 单用户模式成本高。计算机系统管理改用另一方式：用户离线构造出作业；操作员接收用户作业并启动其执行。每个程序最后把控制交回系统监控程序，监控程序装入下一程序并执行之
- 商务应用需输入输出（IO）大量数据但其中的计算比较简单。如果程序做IO时CPU给设备发出命令后进入等待，就会浪费大量的CPU时间。为了利用这种闲置计算能力，人们开发了中断驱动的输入输出和多道程序技术（允许多个程序同时驻留内存）
  - 一旦程序IO就将CPU给操作系统（OS）。OS找出一个可执行程序并令其执行。OS维持多个程序的执行轨迹，知道哪些进程正在等待相应的I/O完成，哪些可运行
  - 设备完成时发中断使执行控制回到OS，OS将等待该IO的进程标记为可执行进程，从可执行进程里选出一个去执行

分时系统和分布式系统

- 随着内存的增大和虚存技术的发展，一个系统里已经可以让任意多个程序同时处于运行状态，系统里存在多个执行进程
  - 批处理系统里，只有一个进程因I/O阻塞时才转到另一进程
  - 强占式分时系统每秒做几次常规切换，防止计算量大的程序长时间连续占据CPU，保证各进程都有平等的运行机会
- 1970 年代早期分时系统已广泛使用。增加了数据共享机制和其他进程间的通信机制，允许用户在应用层面进行并发编程
- 计算机网络是分布式的多机系统，支持真正的并行，程序在物理上独立的不同机器里运行，程序之间通过消息相互通信

  多数分布式系统通过并发以有效利用多台物理设备

  多处理器系统在1960/70年代出现，但很少见。近年大量出现在个人计算机里，主要是因为单处理器的性能已无法提高

个人计算机领域并发性的发展

- 早期个人计算机（PC）只能同时运行一个程序，程序IO时整个计算机等待。微软的Windows和MacOS/Multifinder允许在PC的内存里同时装入多个程序，在程序IO时切换执行

  Windows 3.1/MacOS v7支持合作式并发，一个程序可以占用CPU任意长时间。由于只有一个用户，这种情况尚可接受

- 随着PC功能增强和应用越来越复杂，人们希望更好的并发效果，如希望浏览器在后台更新窗口、检查电子邮件、照顾打印机等
  - 程序不能常规地交出CPU，就无法保证系统的良好响应效果
  - Windows NT和MacOS X加入强占机制，OS定期轮换式地把CPU分配给当时可运行的进程。这样，即使一个进程死循环，其他进程也可能执行，不会导致整个系统垮台
  - 如果所用PC中存在多个执行部件（多核或多CPU），就可以把这些进程分配给它们执行

多线程程序的实例

- 许多实际应用的逻辑结构中存在着对并行的需求，例如：
- 基于万维网的应用对多线程程序的需求特别明显
  - 浏览器：点击链接创建与服务器通讯的新线程。线程接收消息包并将其展示到屏幕，需要访问字型，拼装单词序列并分割成行等。可能派生许多线程：网页中图像/背景/表格常用独立线程处理，框架可能用多个线程。派生线程也能与服务器通信，获取所需信息（如图像内容）。访问菜单/创建窗口或标签页/编辑书签等也常用独立线程，与页面绘制同时进行
  - 服务器端需要同时为许多客户服务，自然需要多线程
- 视频游戏：需要在更新屏幕图像的同时处理击键/鼠标/游戏棒动作。常用独立线程处理输入，一个或多个线程更新屏幕
- 离散事件模拟：用并发线程表示真实世界的一批主动物体，事件在特定时间点自动发生导致哪个线程的并发运行

并行执行硬件

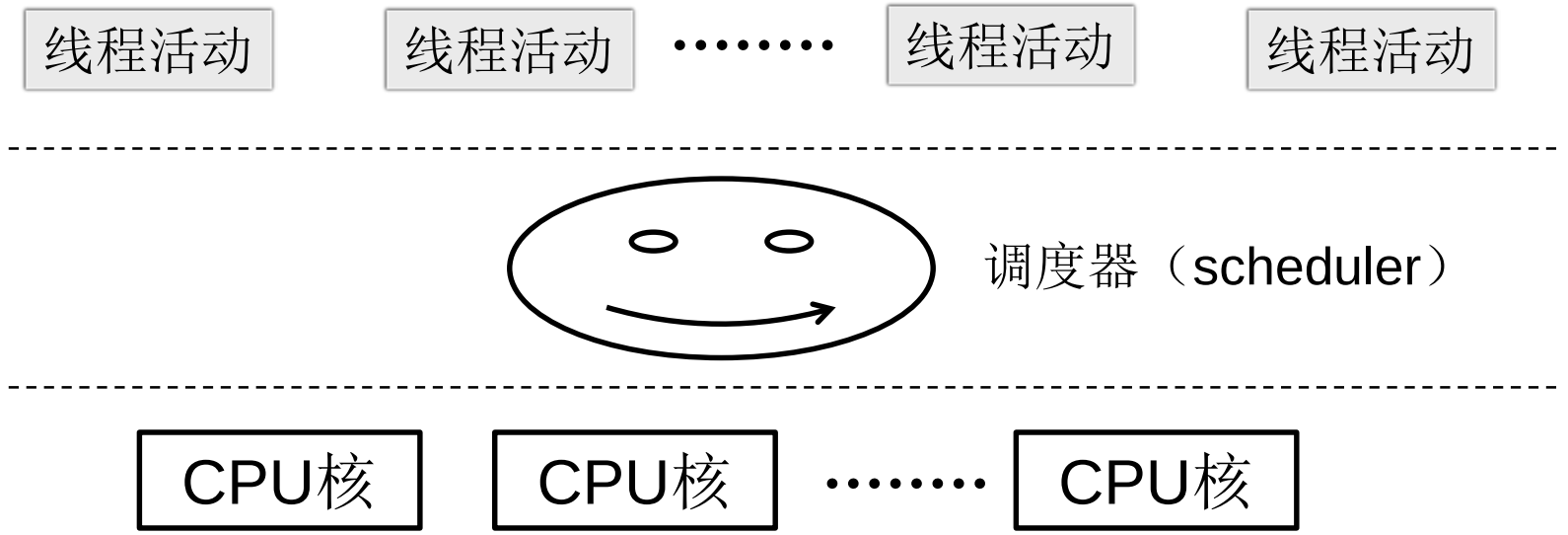
- 并行计算机硬件有许多不同情况：
- 单机并行系统可分为两类：处理器共享公共存储器（多处理器系统；处理器有独立的存储器，只能通过消息相互通信
- 计算机集群：一批物理上独立的单处理器或小型多处理器硬件，安装在一组机架，通过高速网络连接，作为整体管理（**Google**、**Amazon**、**eBay**、百度等都使用成万台处理器构成的集群 ）
- 多机系统：采用消息传递的单机柜多机机器。**1980**年代在科学计算与数据库应用领域流行，已基本被集群和多处理器系统取代
- 向量处理器：一组结构相同的较简单**CPU**，执行一条指令时同步操作。向量指令很容易流水线化，在科学计算中非常有用。向量机的思想现在也在发挥重要作用，如**Intel CPU**的**MMX**多媒体扩充，目前广泛使用的图形处理器和**GPU**等
- 主要问题是处理器之间的通讯网络（拓扑结构，路由），共享和分布式存储器的有效利用（存储器一致性问题）等

并发程序的概念

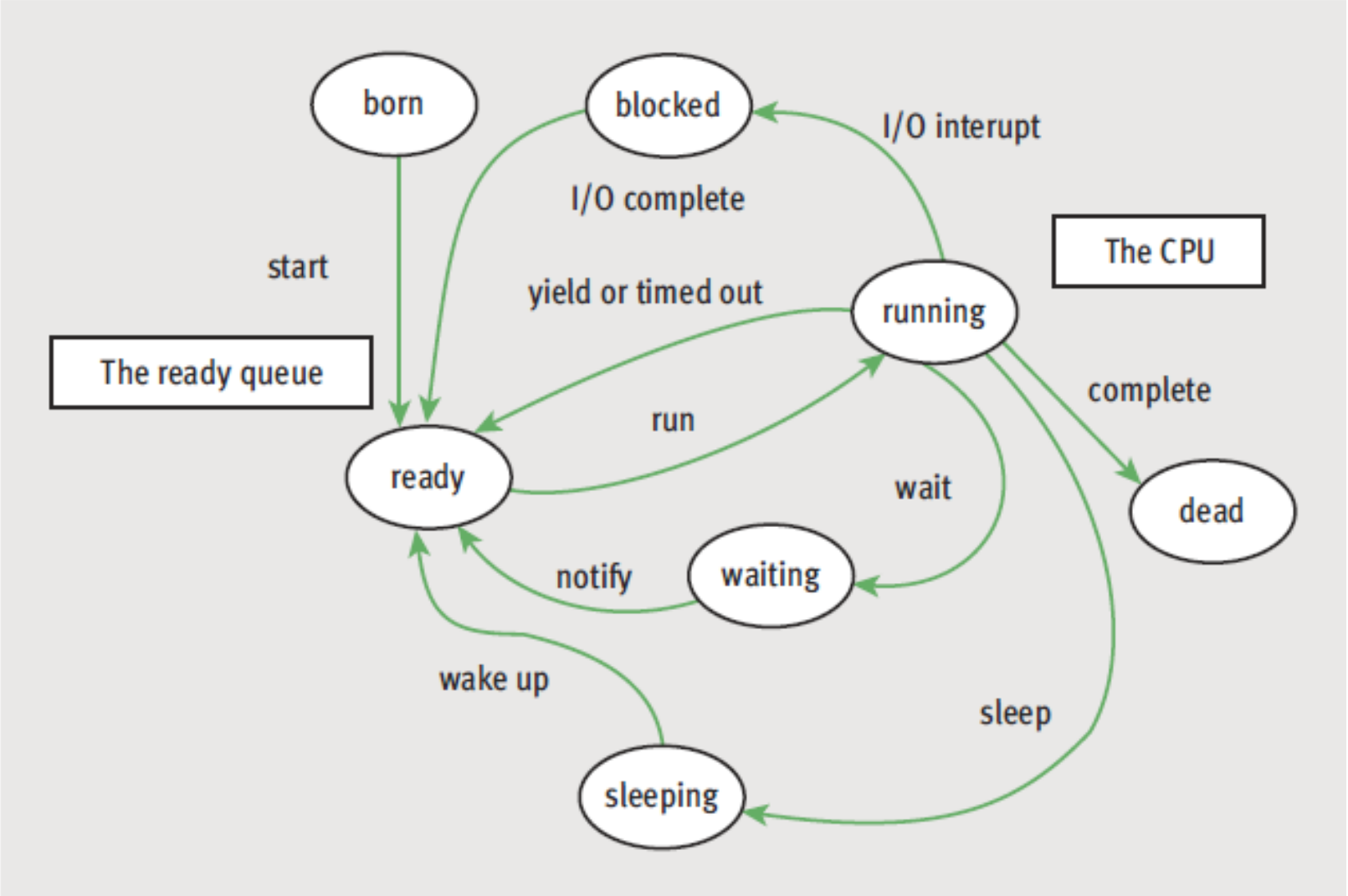
- 常见到并行（**parallelism**）和并发（**concurrency**）两个概念
  - 并行强调多个执行活动同时处于运行状态中，强调其相互独立性。关心并行算法、并行系统、并行体系结构等
  - 并发强调多个执行活动之间的关联和相互作用，关心执行的启动/结束、同步、通讯、资源共享和竞争、互斥等
- 有关并发程序的概念体系不统一（比较乱），下面采用**Python**标准库**threading**线程库的术语
  - **Thread**（线程）指一个可以并发运行的对象（静态概念）。线程的一个执行称为活动（**activity**，动态概念）
  - 基于**threading**库构造并发程序，主要是定义各种线程对象类（作为类**Thread**的派生类），生成线程对象，启动其执行
  - **threading**还提供了一批与并发有关的其他对象类

Python程序的并发执行

- Python语言本身没有并发的概念，通过库支持并发执行
  - 标准库手册第17章，Concurrent Execution
  - 下面介绍基于threading库的并发编程，其他库支持不同功能
- 基本编程方式：定义Thread类的派生类；生成线程对象；调用这种对象的start()方法，启动该对象执行（对象的活动）



线程的活动期和状态转换



Python的threading库

- threading库定义了一组与并发程序有关的类，其中最基本的是Thread类，这个类的对象可以并发执行
- 创建Thread对象时，可用关键字参数target指定对象的执行代码（必须是一个可执行对象，例如一个无参函数对象）
  - 调用线程对象的start()方法就会以并发方式执行target代码
  - 调用线程对象的join()方法导致调用代码阻塞至该线程结束
  - 看两个例子。注意：一个线程对象只能start一次
- 有用操作（方法）
  - 调用线程对象的is_alive()方法时，如果线程正在活动将会返回True，否则返回False
  - 调用threading.enumerate()得到当时处于活动状态的线程表

从Thread派生

- Thread类的对象调用了start方法，就把自己加入线程调度器管理下的就绪线程队列，调度器将在适当的适合为其安排执行

  所有Thread对象都可以被调度，从而得到执行的机会

- 使用Thread类的另一重要方式是建立其派生类
  - 注意，Thread派生类的对象也是Thread对象，创建时需重新定义 __init__ 方法建立派生类所需的对象，但它必须首先调用Thread的初始化函数，使这个对象可能被调度
  - 为定制通过派生定义的线程类实例对象的行为，应该在派生类里重新定义（覆盖）Thread的run方法，这是无参方法
  - 一旦对自定义线程类的对象调用start()方法，就会自动执行该对象所属的类里定义的run()方法
- 下面看两个例子

从Thread派生

- Thread类的__init__方法有几个参数，其target/name参数已经介绍。另一形参args默认值是空序对，用于接受任意多个普通实参；形参kwargs默认值是空字典，用于接受其他关键字实参
- 通过派生定义新线程类时，只允许覆盖（重新定义）__init__和run，其他方法只能调用不能覆盖
- threading包定义的另一个类是Timer，定时器类，可用于生成任意多个计时器对象，两个参数：

  tm = Timer(3.0, *executable*)

  创建一个定时器，这是一种延时线程，用start启动，经过指定时间（以秒计）后启动由*executable*给定的可执行代码
- Timer没有可以重新定义的__init__和run方法，只能用于定义延时动作，具体执行的动作需要另外定义，通过参数给定

线程和并发

- 程序在运行中可以创建任意多个线程对象并激活它们
  - 这些线程活动在调度器的管理下自主运行，其进展情况无法控制，可以等待其终止，或获取它的一些信息
  - 一个线程对象只能启动（和运行）一次，即使前次执行已经终止，也不能再次启动运行
  - 如果进程对象本身的代码是顺序程序，其操作总是按程序描述的控制流顺序进行
  - 同时处于运行中的不同线程的执行速度是不确定的，不仅相对速度不确定，同一程序再次运行的情况也可能不同。整个多线程程序的执行，表现为其中线程的线性执行产生的操作序列的某种相互交错，而且每次交错的情况可能不同
- 如果程序中的各个线程可以任意执行，而且其不同线程的行为相互影响，整个程序的综合行为将很难看清楚

并发线程

- 由于不同的线程的速度可能不同。即使在一次执行中发生错误，后来的多次执行中这种错误也可能根本不发生
  - 所以，并发程序很难通过测试保证正确性
  - 需要按照一定的规矩进行编程
- Python系统一启动，就创建了一个主线程对象，在其中执行当前的最高层代码（脚本）
  - 通过将一个函数（或具有函数性质的其他对象）作为Thread的target参数的方式创建线程对象，适合处理具体的特殊计算过程。可以方便地以线程方式启动任何计算
  - 用从Thread类派生的方式定义的是特殊的线程类。这种做法适合需要创建多个同类线程的情况。可以利用初始化函数建立线程对象的内部状态，采用定义类的局部方法等方式，实现复杂的线程，并将其良好封装

线程之间的信息交换

- 线程是独立的可运行对象，实现一种特定的计算功能。活动的线程处于运行之中，有其内部状态，正在进行自己的计算
  - 前面讨论的各种对象都是被动对象，被其他对象使用。函数对象也是被动的对象，顺序程序里的函数被主线程使用
  - 线程对象则不同，它是一种主动对象，启动之后有自己的行为，可以使用各种被动对象（数据对象、函数对象等）
- 同时运行的一组线程可以各自独立工作（前面线程都是这样），我们也常希望它们能合作完成计算，各自完成计算的一部分。要实现这种合作，一种可能性是让不同线程去操作公共变量

  一个程序里的线程都在同一个存储空间里，有共同的全局名字空间，可以通过全局变量合作工作。下面看一个例子
- 但两个线程操作同一个全局变量，任由它们随便操作，就可能相互干扰，未必能正确完成工作。因此，线程合作需要有控制

共享控制：锁

- 被多个线程操作的变量称为共享变量
  - 任由多个线程随意操作共享变量，由于操作相互交错，以及一些变动操作的“非原子性”，可能造成不正确的操作效果
  - 共享变量操作是一种“临界操作”，有危险。出现这种操作的代码段称为“临界区”（**critical region**，危险区）。对临界操作（临界区）的执行需要控制
- 并发语言或库都要提供一些线程控制机制（同步机制）。现介绍称为“锁”的常用机制，**threading**包中的类**Lock**实例就是锁
  - 锁是一种对象，可以请求和释放（**acquire/release**，只有这两个操作），但任何时刻只能有一个线程获得这个锁
  - 如果一个线程请求一个锁，该操作成功时这个线程继续，如果不能获得（锁被其他线程占用），它就必须等待（称为阻塞，**block**），直至获得了这个锁才能继续

共享控制：锁

- 显然，可以利用锁的性质控制临界区代码的执行
  - 要求线程在进入临界区之前获得锁，保证“互斥”
  - 线程必须在退出临界区时释放锁，以便其他线程能进入
- 这是一个有开始和结束的操作过程（协议），标准的写法是（假设lock是已定义的锁对象）：

```
try:
    lock.acquire()
    … … # 临界代码段
finally:
    lock.release()
```

符合Python with语句模式，Lock对象也实现为管理器，可写

```
with lock:
    … … # 临界代码段
```

共享控制和同步机制

- 有了锁机制，就可能在多个线程共同工作完成一个复杂全局对象的构造过程中，保证它们的操作不相互干扰
- 举个例子：假设现在要构造一个10000个元素的表，每个元素的计算都比较费时间。可以考虑下面方式：
  - 定义一种线程类，其 run 方法完成一个个元素的计算，并将计算结果逐个存入全局表的 [m, n) 段。其中的存入操作是临界操作，用一个锁保护这个共享的全局变量（及其操作）
  - 构造10个线程，分别计算元素段 [0, 1000), [1000, 2000), …
  - 主线程启动这10个线程后等待它们结束
- threading库还提供了另一些同步机制，实现为几个类：Event（事件），Semaphone（信号量），Condition（条件变量），Barrier（栅栏），各有不同用途，用于实现线程间的各种不同的同步需要，这里不再介绍

线程间数据传输

- 通过共享变量实现线程间数据交换，要求不同线程有公共的数据空间，例如在一个全局存储环境中执行，存储可共享的全局数据对象。对在不同存储空间里执行的线程，这种方式就不能用了

  例如，在一个集群中不同计算机上运行的线程，各有存储空间，一台有多个CPU的机器，也可能每个CPU有自己的存储器

- 在一个计算环境中，不同活动线程之间交换数据的另一方式称为线程间通讯，一个线程通过某种通讯媒介把数据送给另一线程
  - 线程间通讯可以通过合适的机制实现，例如通过不同计算机之间的通讯网络，或通过同一台计算机中的共享内存机制
  - 与共享变量的数据交换方式不同，线程间通讯需要一种专门的通讯媒介，有线程作为数据发送方，有线程作为数据接收方。通讯可以是单向或者双向的（由媒介的性质决定）
- 下面简单介绍Python标准库中提供的通讯媒介包queue.py

线程间数据传输

- queue包最常用的类是Queue（称为“队列”），队列对象中可以存放一些数据（默认为任意多项数据），主要操作：
  - q.put(item) 把数据 item 放入队列 q
  - q.get() 从队列 q 中取出一项数据，如果操作时队列里当时没有数据，调用线程就阻塞等待数据的到来
- Queue作为线程间数据传输类，已经内在地实现了操作的互斥。使用线程可以随意使用，Queue保证数据放入和取出的正确性

  如果一个线程取数据时，当时的队列存在多项数据，Queue的实现保证实际得到的是最早放入的数据

  Queue类保证被放入的数据都能按放入顺序逐一取出

  如果存在多个等待数据的线程，某个线程可以取得下一项数据
- 看个例子，其中包括两个产生数据的线程和一个使用数据的线程

并发计算和程序的概念

- 并发和并行
- 线程和活动（进程，计算进程等）
- 线程（进程）同步
- 共享变量，临界代码和临界区
- 同步和锁，请求和释放
- 阻塞/等待/唤醒
- 共享内存和进程间通讯
- 通讯通道（队列）。queue包的其他功能，Queue的实例对象的其他功能，put和get操作的其他可选参数，请参考标准库手册

- 对一个（自定义）线程类，可以创建任意多个线程对象
  - 可能把一件工作划分为一些部分，每部分用一个线程处理
  - 如果硬件有多个执行部件，这样做就可能提高效率
- 举例：**Google**开发的**map-reduce**处理模式
  - 把一个大计算裁剪为一大批同类工作分给一大批处理器，令其并发处理，可以大大缩短完成所有处理的时间
  - **reduce**合并结果，如果分项少可以用一台机器处理，如果很多也可以安排一组处理器处理，合并的结果再进一步合并
- 举例：处理一个大型图像文件，或者大型矩阵，可能将其划分为较小的数据块，启动若干个线程对象，分别处理

  在数值计算和复杂的模拟工作中，这种情况非常多。其中每个子部分的处理是独立的，整个计算是其中各子部分计算的重复