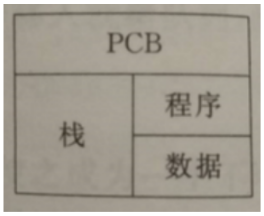
操作系统

进程管理

PCB

- PCB(Process Control Block) 是进程控制块。
- PCB 是进程的唯一标识，操作系统调度进程时就是根据每个进程 PCB 中的信息进行调度的，当决定执行 某个进程后，会根据该进程PCB 中保存的信息去恢复上次执行的现场，当分配到的CPU时间片用完 后，需要将当前状态保存到PCB中，以便下次恢复。
- PCB 组织方式： 通过链表的方式组织成一个个队列，拥有相同状态的进程组成一个队列。 比如就绪进程就会组成就绪队列、因为某些事件而阻塞的进程组成阻塞队列。 也可以将相同状态的PCB按照其他策略排成多个链表

- PCB主要包含下面几部分的内容：
  - 进程的描述信息，比如进程的名称，标识符，
  - 处理机的状态信息，当程序中断是保留此时的信息，以便 CPU 返回时能从断点执行
  - 进程调度信息，比如阻塞原因，状态，优先级等等
  - 进程控制和资源占用，同步通信机制，链接指针（指向队列中下一个进程的 PCB 地址）
- PCB 的作用
  - PCB是进程实体的一部分，是操作系统中最重要的数据结构
  - 由于它的存在，使得多道程序环境下，不能独立运行的程序成为一个能独立运行的基本单位，使得程序可以并发执行
  - 系统通过 PCB 来感知进程的存在。（换句话说，PCB 是进程存在的唯一标识）
  - 进程的组成可以用下图来表示，PCB 就是他唯一标识符。



2.1 线程、进程、协程的区别

答：

- 进程是资源分配的最基本的单位，运行一个程序会创建一个或多个进程，进程就是运行起来的可执行程序。
- 线程是程序执行的最基本的单位，是轻量级的进程，每个进程里都有一个主线程，且只能有一个，和进程是相互依存的关系，生命周期和进程一样。
- 协程是用户态的轻量级线程，是线程内部的基本单位。无需线程上下文切换的开销、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。

进程和线程的区别的话

- 首先从资源来说，**进程是资源分配的基本单位**，但是线程不拥有资源，线程可以访问隶属进程的资源。
- 然后从调度来说，**线程是独立调度的基本单位**，在同一进程中线程切换的话不会引起进程的切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程的切换。
- 从系统开销来讲，由于创建或撤销进程，系统都要分配回收资源，所付出的开销远大于创建或撤销线程时的开销。类似的，在进行进程切换的时候，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境设置，而线程切换只需保存和设置少量寄存器的内容，开销很小。
- 通信方面来说，线程间可以通过直接读写同一进程的数据进行通信，但是进程通信需要借助一些复杂的方法。

死锁

28 什么是死锁，

死锁就是多个进程间彼此持有对方需要的资源，但又在等待对方释放自己需要的资源的状态。

29 为什么产生死锁

- 竞争资源引起死锁：竞争非剥夺性资源（CPU和内存是可剥夺资源，IO设备是非剥夺资源）和临时性资源。非剥夺性资源分配不当、使用未加限制都会产生死锁
- 进程推进顺序不当：相似逻辑的程序，指令顺序不相同。

2.1 讲讲死锁发生的条件是什么?

1. 互斥条件：是资源分配是互斥的，资源要么处于被分配给一个进程的状态，要么就是可用状态。
2. 等待和占有条件：进程在请求资源得不到满足的时候，进入阻塞等待状态，且不释放已占有的资源。
3. 不剥夺条件：已经分配给一个进程的资源不能强制性地被抢占，只能等待占有他的进程释放。
4. 环路等待：有两个或者两个以上的进程组成一条环路，该环路中的每个进程都在等待下一个进程释放所占有的资源。

31 解决死锁的基本策略

- 预防死锁。从根本上破坏死锁出现的四个条件之一，但会降低资源利用率
- 避免死锁。使用某种机制防止系统进入死锁状态，可以有较高的资源利用率
- 检测死锁。允许发生死锁，系统可以检测出死锁的发生，然后消除死锁
- 解除死锁。与检测死锁配套使用，常用方法是撤销或挂起一些进程，释放出一些资源，这些资源分配给阻塞的进程，等于是疏通了环路。

CFS(Completely Fair Scheduler)

内存管理

内存管理的机制

1. 分块管理

   是连续管理的一种，把内存分为几个大小相等且固定的块，每个进程占用其中一个，如果进程很小的话，会浪费大量的空间。已经淘汰。

2. 分页管理

   把内存分为若干个很小的页面，相对比分块的划分力度更大一些。提高内存利用率。减少碎片，页式管理通过页表对应逻辑地址和物理地址。

3. 分段管理

   把内存分为几个大小不定的有实际意义的段，比如 main 函数段，局部变量段，通过管理段表来把逻辑地址转为物理地址。

4. 段页式管理

   结合了段式管理和页面管理的优点，把主存先分为若干个段，每个段又分为若干个页，也就是说段页式管理的段与段以及段的内部都是离散的。

3.4 分页和分段有什么区别呢?

- 共同点的话：
  - 首先都是离散分配的，单每个页和每个段的内存是连续的。
  - 都是为了提高内存利用率，减少内存碎片。
- 不同点：
  - 分页式管理的页面大小是固定的，由操作系统决定；分段式管理的页面是由用户程序所决定的。
  - 分页是为了满足操作系统内存管理的需求，每一页是没有实际的意义的；而段是有逻辑意义的，在程序中可认为是代码段、数据段。
  - 分页的内存利用率高，不会产生外部碎片；而分段如果单段长度过大，为其分配很大的连续空间不方便，会产生外部碎片。

3.5 讲讲分页管理的快表和多级页表（按照why how的方式来回答，即为什么出现快表，是如何解决痛点的）

3.5.1. 快表

- why? 首先快表的引入是为了加快逻辑地址到物理地址的访问速度的，在引入快表之前，由逻辑地址访问到内存的过程是这样的：

  a) 首先根据逻辑地址的高位拿到页号

  b) 根据页号访问内存中页表，根据页表的映射拿到实际的内存块儿号。（一次访问）

  c) 把内存块儿号和逻辑地址的低位拼接，得到物理地址

  d) 访问对应的内存物理地址。（二次访问）

  这样是需要有两次直接访问内存的过程的，所以为了加快这个速度，引入了快表，快表可以认为是一个 Cache，内容是页表的一部分或者全部内容。和页表的功能是一样的，只不过比在内存中的页表的访问速度要快很多。

- how? 根据局部性原理，被访问后的内存块儿很可能在短时间内再次被访问，可能程序在一段时间内会多次访问同一个页表项。所以在每次访问页表项时，先在快表里查询是否有该页表项，如果没有再去页表中查询，并把查到的页表项放入快表。如果快表满了，就根据一些策略把里面的页表项淘汰掉，再把新查询的页表加入进去。

3.5.2 多级页表

- why? 多级页表主要是为了解决页表在内存中占用空间太大的问题的，典型的时间换空间。
- how? 讲个例子即可：在引入多级页表之前，我们使用单级页表来进行存储页表项，假如虚拟内存为 4GB，每个页大小为 4KB，那么需要的页表项就为 4GB / 4KB = 1M 个！每个页表项一般为 4B，那么就需要 4MB 的空间，大概需要占用 1000 个页来存页表项。

  所以如果引入两级页表，让一级页表的每个页表项不再映射 4KB，而是映射 4MB，那么需要的一级页表项的个数为 4GB / 4MB = 1K 个，再让每个一级的页表项映射 1K 个二级页表项。当一级页表的某个页表项被用到时，再把该一级页表项对应的所有 1K 个二级页表项加载到内存中，这样可以节省大量的空间！

3.9 讲讲页面置换算法

答：当使用请求分页存储来管理内存时，发生缺页中断，就是要访问的页面不在内存中，这时就需要操作系统把其调入主存后再进行访问。

而在发生缺页中断时，内存中没有空闲的页面，就必须在内存中根据一定的策略挪出一些不用的页面，可以把页面置换算法看成是淘汰机制。

- OPT 页面置换法，最佳页面置换：不可实现，不可预测哪个是不用的。
- FIFO 先到先出算法，把在内存中停留时间最长的页面置换出去
- LRU 最近最久未使用页面置换算法：LRU 算法赋予每一个页面一个访问字段，来记录一个页面最近一次访问到现在所经历的时间 T，需要淘汰一个页面时，把最久没有使用的页面淘汰掉就可以了。
- LFU 最少使用算法：把使用最少的页面淘汰掉。

IO 管理

IO 模型

- **阻塞 I/O**：你站在柜台前等餐，直到食物准备好。
- **非阻塞 I/O**：你时不时去柜台看看食物是否准备好。
- **I/O 多路复用**：你坐在座位上，服务员会通知你食物准备好。
- **信号驱动 I/O**：你给服务员留了电话，食物准备好时会打电话通知你。
- **异步 I/O**：你点餐后继续做其他事情，食物准备好时会直接送到你面前。

1. 阻塞 I/O (Blocking I/O)

- **定义**：进程发起 I/O 操作后，必须等待操作完成才能继续执行。
- **特点**：
  - 简单易用，编程模型直观。
  - 效率低，进程在等待期间无法执行其他任务。
  - 适用于简单的 I/O 操作场景。

```
┌──────────┐
│ 发起 I/O 请求│
├──────────┤
│ 阻塞等待     │
├──────────┤
│ I/O 完成     │
└──────────┘
```

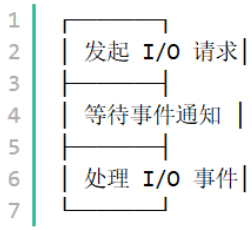
2. 非阻塞 I/O (Non-blocking I/O)

- **定义**：进程发起 I/O 操作后立即返回，若操作未完成则返回错误，进程可继续执行其他任务。
- **特点**：
  - 需要反复检查 I/O 状态，增加编程复杂度。
  - 提高了 CPU 利用率，适合需要高并发的场景。
  - 适用于需要快速响应的应用。

```
┌──────────┐
│ 发起 I/O 请求│
├──────────┤
│ 立即返回     │
├──────────┤
│ 检查 I/O 状态│
└──────────┘
```

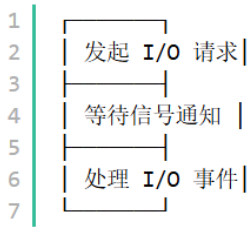
3. I/O 多路复用 (I/O Multiplexing)

- **定义**：通过select、poll或epoll等系统调用，进程可以同时监控多个 I/O 事件。
- **特点**：
  - 适合处理大量 I/O 连接的场景，如网络服务器。
  - 通过单个线程管理多个 I/O，提高资源利用率。
  - 编程复杂度较高，需要处理事件循环。



4. 信号驱动 I/O (Signal-driven I/O)

- **定义**：进程发起 I/O 操作后，内核通过信号通知进程 I/O 事件的发生。
- **特点**：
  - 进程无需阻塞等待，可以处理其他任务。
  - 需要处理信号，编程复杂度较高。
  - 适用于需要异步通知的场景。



5. 异步 I/O (Asynchronous I/O)

- **定义**：进程发起 I/O 操作后立即返回，内核在操作完成后通知进程。
- **特点**：
  - 进程无需等待，完全异步处理。
  - 提高了系统的并发能力和响应速度。
  - 编程复杂度最高，需要处理回调或事件通知。

```
┌──────────┐
│ 发起 I/O 请求 │
├──────────┤
│继续执行其他任务│
├──────────┤
│ I/O 完成通知  │
└──────────┘
```

6. I/O 模型对比

模型	优点	缺点	适用场景
阻塞 I/O	简单易用	效率低	简单 I/O 操作
非阻塞 I/O	提高 CPU 利用率	编程复杂	高并发应用
I/O 多路复用	高效管理多连接	编程复杂	网络服务器
信号驱动 I/O	异步通知	复杂度高	异步通知场景
异步 I/O	完全异步	复杂度最高	高性能应用

IO 多路复用

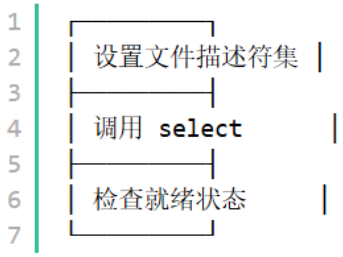
1. I/O 多路复用的基本概念

- **定义**：I/O 多路复用允许一个或多个线程同时监控多个文件描述符（如套接字、管道）的状态变化（如可读、可写、异常），并在状态变化时进行相应处理。
- **优点**：
  - 提高了系统的并发性和资源利用率。
  - 通过单个线程管理多个 I/O，减少了线程切换和资源消耗。
  - 适用于需要处理大量 I/O 连接的场景。

	select	poll	epoll
性能	随着连接数的增加，性能急剧下降，处理成千上万的并发连接数时，性能很差	随着连接数的增加，性能急剧下降，处理成千上万的并发连接数时，性能很差	随着连接数的增加，性能基本没有变化
连接数	一般1024	无限制	无限制
内存拷贝	每次调用select拷贝	每次调用poll拷贝	fd首次调用epoll_ctl拷贝，每次调用epoll_wait不拷贝
数据结构	bitmap	数组	红黑树
内在处理机制	线性轮询	线性轮询	FD挂在红黑树，通过事件回调callback
时间复杂度	O(n)	O(n)	O(1)

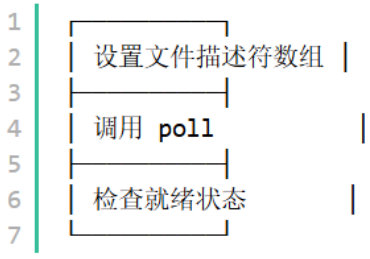
2.1 select

- **特点**：
  - 使用固定大小的位图表示文件描述符集，最大支持 1024 个文件描述符。
  - 每次调用都需要重新设置文件描述符集，效率较低。
- **适用场景**：适合小规模 I/O 监控，简单易用。



2.2 poll

- **特点**：
  - 使用动态数组表示文件描述符集，支持任意数量的文件描述符。
  - 每次调用都需要遍历整个数组，效率较低。
- **适用场景**：适合中等规模 I/O 监控，灵活性较高。



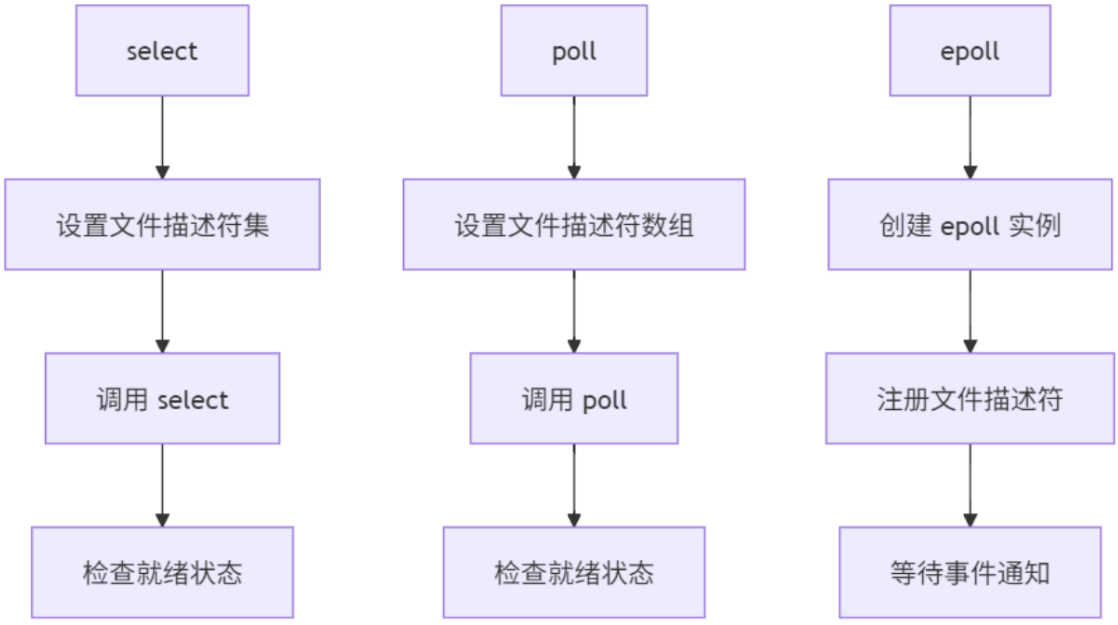
2.3 epoll

- **特点**：
  - 使用事件通知机制，支持大规模文件描述符监控。
  - 只需在文件描述符状态变化时进行回调，效率高。
- **适用场景**：适合大规模高并发 I/O 监控，性能优越。

```
┌──────────┐
| 创建 epoll 实例 |
├──────────┤
| 注册文件描述符  |
├──────────┤
| 等待事件通知    |
└──────────┘
```

3. 适用场景

- **高并发网络服务器**：如 Web 服务器、聊天服务器，需同时处理大量客户端连接。
- **实时数据处理**：如金融交易系统，需快速响应数据变化。
- **多设备监控**：如监控系统，需同时监控多个传感器或设备的状态。



问题	解答
select、poll 和 epoll 的主要区别是什么?	select 使用固定大小位图，poll 使用动态数组，epoll 使用事件通知机制，epoll 性能最佳。
为什么 epoll 适合高并发场景?	因为 epoll 只在文件描述符状态变化时进行回调，减少了不必要的遍历和检查。
如何选择合适的 I/O 多路复用方式?	根据应用的并发需求和文件描述符数量选择，epoll 适合大规模高并发场景。

epoll 为什么比select、poll更高效?

- epoll 采用红黑树管理文件描述符
  从上图可以看出，epoll使用红黑树管理文件描述符，红黑树插入和删除的都是时间复杂度O(logN)，不会随着文件描述符数量增加而改变。
  select、poll采用数组或者链表的形式管理文件描述符，那么在遍历文件描述符时，时间复杂度会随着文件描述的增加而增加。
- epoll 将文件描述符添加和检测分离，减少了文件描述符拷贝的消耗
  select&poll 调用时会将全部监听的 fd 从用户态空间拷贝至内核态空间并线性扫描一遍找出就绪的 fd 再返回到用户态。下次需要监听时，又需要把之前已经传递过的文件描述符再读传递进去，增加了拷贝文件的无效消耗，当文件描述很多时，性能瓶颈更加明显。
  而epoll只需要使用epoll_ctl添加一次，后续的检查使用epoll_wait，减少了文件拷贝的消耗。

状态处理

1. 水平触发 (Level Triggered, LT)

- **定义**：在水平触发模式下，只要文件描述符的状态满足条件（如可读、可写），就会反复通知。
- **特点**：
  - 简单易用，适合初学者。
  - 只要条件满足，事件会持续触发，直到状态改变。
  - 可能导致重复处理同一事件，效率较低。

```
┌──────────┐
│ 状态满足条件 │
├──────────┤
│ 反复通知     │
├──────────┤
│ 状态改变     │
└──────────┘
```

2. 边缘触发 (Edge Triggered, ET)

- **定义**：在边缘触发模式下，只有当文件描述符的状态发生变化时（如从不可读到可读），才会通知。
- **特点**：
  - 高效，减少了不必要的事件通知。
  - 需要一次性处理所有数据，避免遗漏。
  - 编程复杂度较高，适合高性能应用。

```
┌──────────┐
│ 状态变化     │
├──────────┤
│ 仅通知一次   │
├──────────┤
│ 处理所有数据 │
└──────────┘
```

3. ET 与 LT 的对比

维度	水平触发 (LT)	边缘触发 (ET)
通知机制	状态满足时反复通知	状态变化时通知一次
处理效率	可能低，重复处理	高效，减少通知
编程复杂度	简单	较高
适用场景	简单应用	高性能应用

4. 应用场景

- **水平触发 (LT)**：
  - 适合简单的 I/O 处理场景，如小型服务器或应用。
  - 易于实现，适合初学者和简单项目。
- **边缘触发 (ET)**：
  - 适合高性能、高并发的网络服务器和应用。
  - 需要一次性处理所有数据，避免遗漏。

水平触发(LT)

关注点是数据是否有无，只要读缓冲区不为空，写缓冲区不满，那么epoll_wait就会一直返回就绪，水平触发是epoll的默认工作方式。

边缘触发(ET)

关注点是变化，只要缓冲区的数据有变化，epoll_wait就会返回就绪。
这里的数据变化并不单纯指缓冲区从有数据变为没有数据，或者从没有数据变为有数据，还包括了数据变多或者变少。即当buffer长度有变化时，就会触发。
假设epoll被设置为了边缘触发，当客户端写入了100个字符，由于缓冲区从0变为了100，于是服务端epoll_wait触发一次就绪，服务端读取了2个字节后不再读取。这个时候再去调用epoll_wait会发现不会就绪，只有当客户端再次写入数据后，才会触发就绪。
这就导致如果使用ET模式，那就必须保证要「一次性把数据读取&写入完」，否则会导致数据长期无法读取/写入。

Reactor 模式

计算机网络

维度	TCP（Transmission Control Protocol）	UDP（User Datagram Protocol）
是否连接	**面向连接**：三次握手建立、四次挥手释放	**无连接**：直接发报文
可靠性	序列号 + ACK + 重传 → **可靠、有序、不丢不重**	Best-Effort：**可能丢、乱序、重复**，不保证送达
传输粒度	字节流（Stream） 应用看不到分段	报文（Datagram） 一次send()= 一次完整包
流量 / 拥塞控制	有窗口 + 拥塞算法（慢启动、拥塞避免、BBR...）	无流控、无拥塞控制，**发得快丢得多**
首部开销	20 Byte 起，带选项可到 40-60 Byte	8 Byte 固定，结构简单
传输速度	稳定但握手 + 拥塞控制 → **首包慢、稳中求胜**	**低延迟、抖动小**，但需应用自行容错
组播 / 广播	不支持	原生支持单播 / 组播 / 广播
适用场景	HTTP/HTTPS、FTP、数据库、电子邮件...	DNS、VoIP、直播、在线游戏、DHCP...
稳定性成本	端口占用多、FD 常驻、握手耗 RTT	资源占用低、无状态，服务端易做并发
消息边界	无，需要应用层自行划分	天然保留边界，一包一消息

计算机组成原理

数据结构

概率论和数理统计

7、协方差、相关系数

协方差

期望值分别为$E(X)$与$E(Y)$的两个实随机变量X与Y之间的协方差$Cov(A,Y)$定义为：

$$\begin{aligned} Cov(X,Y) &= E[(X-E[X])(Y-E[Y])] \\ &= E[XY]-2E[Y]E[X]+E[X]E[Y] \\ &= E[XY]-E[X]E[Y] \end{aligned}$$

即X，Y的协方差等于每一个X减去X的平均值乘上每一个Y减去Y的平均值的乘积的和的平均值。

相关系数

（皮尔逊相关系数）

$$p_{xy}=\frac{Cov(X,Y}{\sqrt{D(X)}\sqrt{D(Y)}}$$

即，用X，Y的协方差除以X的标准差和Y的标准差

概率、似然

1、概率（发生前推测）

- 某件事情发生的可能性，在结果没有产生之前依据环境所对应的参数来预测某件事情发生的可能性
- 例如抛硬币之前，推测正面朝上的概率为50%

2、似然（发生后推测，参数）

- 是在确定的结果之后推测产生这个结果的可能环境（参数）
- 例如抛一枚硬币1000次，其中500次正面朝上，推测这是一枚标准硬币，正面吵上的概率为50%

θ 是模型的参数，$x_1, x_2 \ldots x_n$ 是 n 次采样得到的结果。

而一个数理统计模型是由果溯因，即求解一下问题，**p是多大时，事件A发生k次的概率最大，实际上就是一个求参数问题。**

- **概率质量函数**（Probability Mass Function，PMF）是离散型随机变量在个特定取值上的概率
- **概率密度函数**（Probability Density Function，PDF）是统计学中常用的参数估计方法

最大似然估计（Maximum Likelihood Estimation，MLE）是统计学中常用的参数估计方法，用于根据已观测到的样本数据，选择使得观测数据出现概率最大的参数值。

- 对于离散型随机变量，似然函数是概率质量函数的乘积：

$$L(\theta) = P(X = x_1) \times P(X = x_2) \times \cdots \times P(X = x_n)$$

- 对于连续型随机变量，似然函数是概率密度函数的乘积：

$$L(\theta) = f(x_1|\theta) \times f(x_2|\theta) \times \cdots \times f(x_n|\theta)$$

最大似然估计的目标是找到使得似然函数最大化的参数值。

线性代数

特征值和特征向量（Eigenvalue & Eigenvector）

今天和大家聊一个非常重要，在机器学习领域也广泛使用的一个概念——矩阵的特征值✦与特征向量。

我们先来看它的定义，定义本身很简单，假设我们有一个n阶的矩阵A以及一个实数λ，使得我们可以找到一个非零向量x，满足：

$$Ax = \lambda x$$

如果能够找到的话，我们就称λ是矩阵A的特征值，非零向量x是矩阵A的特征向量。

几何意义

光从上面的式子其实我们很难看出来什么，但是我们可以结合矩阵变换的几何意义，就会明朗很多。

我们都知道，对于一个n维的向量x来说，如果我们给他乘上一个n阶的方阵A，得到Ax。从几何角度来说，是对向量x进行了一个线性变换✦。变换之后得到的向量y和原向量x的方向和长度都发生了改变。

但是，对于一个特定的矩阵A来说，总存在一些特定方向的向量x，使得Ax和x的方向没有发生变化，只是长度发生了变化。我们令这个长度发生的变化当做是系数λ，那么对于这样的向量就称为是矩阵A的特征向量，λ就是这个特征向量对应的特殊值。

$$\lambda_1 + \lambda_2 + \ldots + \lambda_n = a_{11} + a_{22} + \ldots + a_{nn} = trace(A)$$
$$\lambda_1 \lambda_2 \ldots \lambda_n = |A| = det(A)$$

复数特征值

值得注意的是，特征值未必是实数，比如下面的矩阵：

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$
$$\det(A - \lambda I) = \begin{vmatrix} 0-\lambda & -1 \\ 1 & 0-\lambda \end{vmatrix} = \lambda^2 + 1 = 0$$

此时特征值是复数，λ=±i

使用Python⁺求解特征值和特征向量

在我们之前的文章当中，我们就介绍过了Python在计算科学上的强大能力，这一次在特征值和特征矩阵的求解上也不例外。通过使用numpy当中的库函数，我们可以非常轻松，一行代码，完成特征值和特征向量的双重计算。

我们一起来看代码：

```
import numpy as np

a = np.mat([[3, 1], [1, 3]])
lam, vet = np.linalg.eig(a)
```

np.linalg.eig 方法会返回两个值，第一个返回值是矩阵的特征值，第二个返回值是矩阵的特征向量，我们看下结果：

```
In [2]: vet
Out[2]: matrix([[ 0.70710678, -0.70710678],
                [ 0.70710678,  0.70710678]])

In [3]: lam
Out[3]: array([4., 2.])
```


这里的特征向量为什么是0.707呢？因为Python自动帮我们做好了单位化，返回的向量都是单位向量，不得不说实在是太贴心了。

二次型和正定性

矩阵的二次型+

任意一个正方矩阵的二次型定义为 $x^H Ax$ ，其中 x是任意非零向量 。

下面来看一个例子，以实矩阵以及实向量为例：

$$x^T Ax = \begin{bmatrix} x_1, x_2, x_3 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 \\ -1 & 7 & 5 \\ -1 & 6 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$= x_1^2 + 7x_2^2 + 3x_3^2 + 3x_1x_2 + x_1x_3 + 11x_2x_3$$

这就是关于变元 x 的二次型函数，故称 $x^T Ax$ 为矩阵 A 的二次型。

那么二次型的计算有没有什么规律呢?
还是以上式为例。（ a_{ij} 代表 A 中第 i 行第 j 列的元素）

1. 首先对于 x_1^2, x_2^2, x_3^2 这三项，我们可以看出，系数分别对应着矩阵 A 主对角线上的元素。
2. 对于剩下的三项， x_1x_2 的系数为 $a_{12} + a_{21}$ ， x_1x_3 的系数为 $a_{13} + a_{31}$,x_2x_3 的系数为 $a_{23} + a_{32}$

因此，推而广之，我们可以得到**计算二次型的一般公式**：

$$x^T Ax = \sum_{i=1}^{n} a_{ii}x_i^2 + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (a_{ij} + a_{ji})x_i x_j$$

那么，我们是否能将二次型作为刻画矩阵特征的性能指标呢?

暂时不行。因为对于任意一个二次型函数，我们会发现，与其对应的矩阵不止一个！也就是说，存在许多矩阵，他们的二次型是相同的。举个例子：

$$D = \begin{bmatrix} 1 & 114 & 52 \\ -111 & 7 & 2 \\ -51 & 9 & 3 \end{bmatrix}, F = \begin{bmatrix} 1 & 114 & 52 \\ -111 & 7 & 4 \\ -51 & 7 & 3 \end{bmatrix}$$

对于上面这两个不同的矩阵来说，他们的二次型是相同的。

因此，为了保证定义的唯一性，我们在讨论矩阵 A 的二次型时，通常假定 A 为实对称矩阵或复共轭对称矩阵(即 $Hermitian$ 矩阵)。通过这个约束条件，我们还能够保证二次型函数一定是实值函数。

通过二次型 $x^T Ax$ 我们就可以得出正定矩阵的定义：对于任意一个 $Hermitian$ 矩阵 A

1. $\forall x \neq 0, x^H Ax > 0$ ，则称 A 为正定矩阵
2. $\forall x \neq 0, x^H Ax \geq 0$ ，则称 A 为半正定矩阵
3. $\forall x \neq 0, x^H Ax < 0$ ，则称 A 为负定矩阵
4. $\forall x \neq 0, x^H Ax \leq 0$ ，则称 A 为半负定矩阵
5. 若二次型可能取正值，也可能取负值，则称 A 为不定矩阵

总结一下：作为一个性能指标，矩阵的二次型刻画矩阵的正定性。

(1)正定矩阵：所有特征值取正实数的矩阵。
(2)半正定矩阵：各个特征值取非负实数的矩阵。
(3)负定矩阵：全部特征值为负实数的矩阵。
(4)半负定矩阵：每个特征值取非正实数的矩阵。
(5)不定矩阵：特征值有些取正实数，另外一些取负实数的矩阵。

Hermitian 矩阵

定义

厄米特矩阵✦（Hermitian Matrix，又译作“埃尔米特矩阵”或“厄米矩阵”），指的是自共轭矩阵。矩阵中每一个第 i 行第 j 列的元素都与第 j 行第 i 列的元素的共轭相等。显然厄米特矩阵主对角线上的元素都是实数。

复数向量下的向量转置

我们发现， $\bar{x}^T$ 将会经常出现，它意味着对x里面的每个元素取其共轭复数，然后将整个矩阵转置。我们将这两个操作合起来称为共轭转置（conjugate transpose), 并且用 $x^\dagger$ 表示。

为什么要将两个操作合起来，因为我们往往不只需要转置，只需要再转置的时候取其共轭。

所以两个复数向量x和y的内积可以表示为 $x^\dagger y$, 如果 $x^\dagger y = 0$ ， 这两个向量正交。

复数向量x的模长为 $||x|| = (x^\dagger x)^{\frac{1}{2}}$

$(AB)^\dagger = B^\dagger A^\dagger$

Hermitian Matirces✦

对于实数矩阵，如果 $A = A^T$ ， 我们称A这个矩阵是对称矩阵。

对于复数矩阵，也有类似对称的概念。如果对于复数矩阵A,有 $A = A^\dagger$, 我们则称这个矩阵为Hermitian Matirces

我们将会发现，如果这个复数矩阵A的虚部全部为0，那么 $A = A^\dagger$ 就会变成 $A = A^T$ ，因为其共轭等于其自身。这个时候，我们会发现对称矩阵只是Hermitian Matirces虚部为0 的情况。

格拉姆矩阵（Gram matrix)

4.1 Gram矩阵和度量矩阵的定义

设 $\beta_1, \beta_2, \ldots, \beta_s$ 是内积空间的一个**向量组**，矩阵

$$G = \begin{bmatrix} < \beta_1, \beta_1 > & \cdots & < \beta_1, \beta_n > \\ \vdots & \ddots & \vdots \\ < \beta_n, \beta_1 > & \cdots & < \beta_n, \beta_n > \end{bmatrix}$$

称为向量组 $\beta_1, \beta_2, \ldots, \beta_s$ 的**Gram矩阵**

若 $\beta_1, \beta_2, \ldots, \beta_s$ 是**一组基**，则将Gram矩阵称为该基的**度量矩阵**

令 $A = [\beta_1, \beta_2 \ldots \beta_n]$，则 $G = A^T A$

海塞矩阵（Hessian matrix)

假设有一实值函数$f(x_1, x_2, \ldots, x_n)$，如果 f 的所有二阶偏导数都存在并在定义域内连续，那么函数f 的黑塞矩阵为

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \frac{\partial^2 f}{\partial x_2\, \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\, \partial x_1} & \frac{\partial^2 f}{\partial x_n\, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

或使用下标记号表示为

$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$