# 编程语言

2022年9月22日 16:18

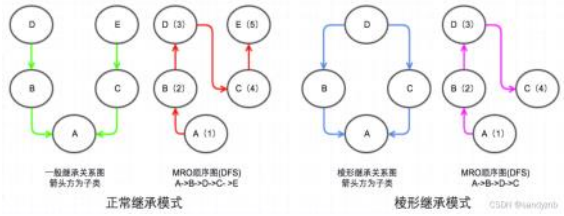
# Python

2022年4月23日 21:35

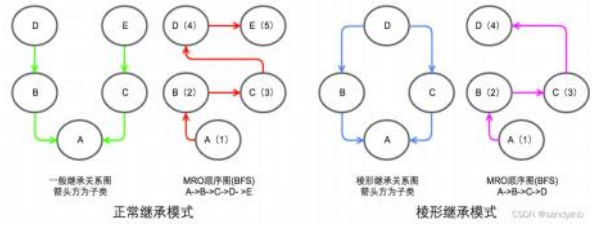
1. 什么是python生成器
   - 生成器概念
     - 一种一边循换一边计算的机制
     - 对象后续的元素是通过某种算法推算出来的
   - 生成器返回一个可迭代对象，但一次只能返回一个值（可以用next（）），减少内存消耗
   - 函数生成器
     - 使用了yield关键字的函数
2. python字典的底层原理
   - 字典的底层实现是哈希表
   - 哈希表
     - 可以看做一张带索引和存储空间的表
     - 对键值进行哈希函数映射，得到对应的索引，然后根据索引对应的空间进行值的读取
     - 

| 哈希运算结果 | 取余运算结果 | 索引 | 键值对 |
| --- | --- | --- | --- |
| hash('小王')=2360347816510736229 | 2360347816510736229%3=0 | 0 | '小王':26 |
| hash('大熊')=4284897975392025871 | 4284897975392025871%3=1 | 1 | '大熊':28 |
| hash('牛牛')=-7069010861127204901 | -7069010861127204901%3=2 | 2 | '牛牛':3 |

3. python中 is 和 == 的区别
   - Is：判断两个对象是否是同一个引用
   - ==：判断两个对象的值是否相等
   - Id()：查看对象的地址
4. 主数据与元数据
   - 元数据：数据的数据
     - 对数据的属性信息进行具体描述
       - 175：含义，获取时间，存储位置，是否公开
   - 主数据：业务实体数据
     - 系统间的共享数据，整个业务系统中价值最大的数据
       - 对于二手房平台，房源信息就是主数据
5. python方法的解析顺序（MRO）
   - MRO：需要满足 本地优先级+单调性顺序
     - Python的方法解析顺序优先级，从高到低为：实例本身 -> 类 -> 继承类 (继承关系越近，越先定义，优先级越高)
   - 常见MRO方法
     - 深度优先搜索
       - 正常继承没有问题，菱形继承顺序错误

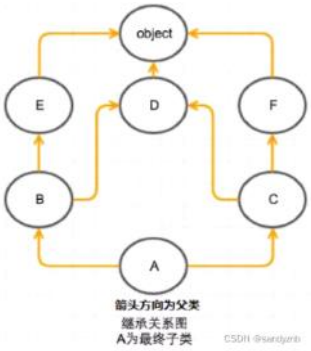



     - 广度优先搜索
       - 菱形继承没有问题，正常继承出现问题




     - C3算法
       - 正常继承和菱形继承都没有问题
       - 核心：不断的合并 “入度为0的点”

- 举例：




```
mro(E) = [E,O]
mro(D) = [D,O]
mro(F) = [F,O]
mro(B) = [B] + merge(mro(E),mro(D),[E,D])
 = [B] + merge([E,O],[D,O],[E,D]) E符合merge条件
 = [B,E] + merge([O],[D,O],[D]) D符合merge条件
 = [B,E,D] + merge([O],[O],[]) O符合merge条件
 = [B,E,D,O]
mro(C) = [C] + merge(mro(D),mro(F),[D,F])
 = [C] + merge([D,O],[F,O],[D,F]) D符合merge条件
 = [C,D] + merge([O],[F,O],[F]) F符合merge条件
 = [C,D,F] + merge([O],[O],[]) O符合merge条件
 = [C,D,F,O]
mro(A) = [A] + merge(mro(B),mro(C),[B,C])
 = [A] + merge([B,E,D,O] ,[C,D,F,O] ,[B,C]) B符合merge条件
 = [A,B] + merge([E,D,O] ,[C,D,F,O] ,[C]) E符合merge条件
 = [A,B,E] + merge([D,O] ,[C,D,F,O] ,[C]) C符合merge条件
 = [A,B,E,C] + merge([D,O] ,[D,F,O] ,[]) D符合merge条件
 = [A,B,E,C,D] + merge([O] ,[F,O] ,[]) F符合merge条件
 = [A,B,E,C,D,F] + merge([O] ,[O] ,[]) O符合merge条件
 = [A,B,E,C,D,F,O] 与程序结果一致
```

    - 经典类：深度优先搜索
    - 新式类：广度优先搜索
    - 新式类：在声明类时需要加上object关键字（Python3默认全是新式类）
6. Python中List和Dict的区别
    - Dict
        - 查找速度快，占用内存也较大
        - 键值必须是不可变类型
        - Dict是无序的
        - Dict是通过hash表实现的，Dict本身是数组，数组索引通过hash处理后得到，hash使键值均匀分布在数组中
    - List
        - 查找速度较慢，占用内存小
7. Python中Dict排序
    - 按照value从大到小：sorted(dict.items(), key=lambda x:x[-1], reverse=True)
8. Python多进程实现
    - 进程：程序的一次执行过程，是资源调度的基本单位
    - 线程：一个进程可以对应于多个现程
    - 区别
        - 进程：操作系统资源分配的基本单位
        - 线程：任务调度和执行的基本单位
    - 实现多进程
        - 使用multiprocessing模块中的Process类
        - 继承multiprocessing模块中的Process类，重写run方法
        - OS.fork()
        - 进程池：当进程数很多时，可以采用multiprocessing模块中的Pool()方法
9. Python中深拷贝和浅拷贝
    - 不可变对象
        - 深拷贝

- 不可变数据类型的深浅拷贝，其结果是相同的

```
In [35]: # 针对字符串

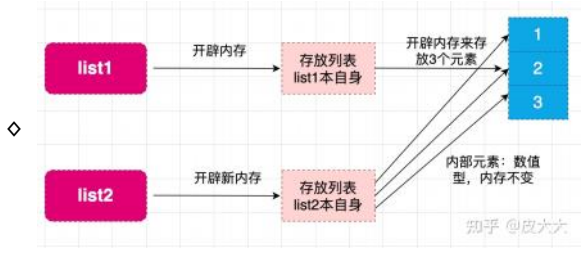
         a = "Peter"
         b = copy.copy(a)  #  浅拷贝
         c = copy.deepcopy(a)  # 深拷贝
```

```
In [36]: print("原数据的地址:",id(a))
         print("浅拷贝的地址:",id(b))
         print("深拷贝的地址:",id(c))
```

```
原数据的地址: 4579746672
浅拷贝的地址: 4579746672
深拷贝的地址: 4579746672
```



  - 浅拷贝
    - 浅拷贝的对象和原数据对象是相同的内存地址
- 可变对象
  - 深拷贝
    - 深拷贝是变量对应的值复制到新的内存地址中
  - 浅拷贝：
    - 不存在嵌套类型的可变类型数据（列表、字典、集合）
      - 浅拷贝对象的地址和原对象的地址是不同的，但列表中的元素（第1个元素为例）和浅拷贝对象中的第一个元素的地址是相同的，因为元素本身是数值型，是不可变的




    - 存在嵌套类型的可变类型数据（列表、字典、集合）
      - 浅拷贝只复制最外层的数据，导致内存地址发生变化，里面数据的内存地址不会变


- 深拷贝
  - 重新开辟新的空间，原始对象的改变不会引起新对象的改变
  - copy.deepcopy()
- 浅拷贝
  - 两个对象指向同一块数据空间，原始对象的改变会引起新对象的改变
  - copy.copy()
- 不可变对象不存在深浅拷贝问题

10. Eval()函数
    - eval() 函数用来执行一个字符串表达式，并返回表达式的值
11. 为什么Python多线程不起作用
    - 因为Python中有全局解释器锁
    - 步骤
      - 线程获取全局解释器锁
      - 执行代码直到sleep  或者 Python解释器将其挂起

        - 释放全局解释器锁
    - 一个线程只有获得全局解释器的使用权限，才能被执行
    - 一个Python进程中只有一个GIL
    - 因此同一时刻只有一个线程在执行
12. Python中的各种锁
    - 全局解释器锁GIL
        - 概念
            - 对于同一个进程中的多个线程来说，同一时间内只有一个线程可以获取全局解释器（CPU）的使用权限，其他线程只有等待该线程的使用权限消失后才能使用全局解释器，多个线程之间不会互相影响
        - 好处
            - 使数据更加安全，解决了多线程之间的数据完整性和状态同步问题
            - 避免了大量的加锁解锁步骤
        - 缺点
            - 多核处理器退化成单核处理器，只能并发不能并行
    - 同步锁
        - 概念
            - 确保一个线程下的程序在一段时间内被CPU执行
        - 原因
            - 当一个线程在使用CPU时，该线程下的程序会有IO操作，此时CPU可能会切换到别的线程上，这样会影响该程序结果的完整性
        - 使用
            - 在使用公共数据操作的前后添加 上锁 和 释放锁 操作
    - 死锁
        - 概念
            - 两个或两个以上线程或进程在执行程序过程中，因争夺资源而互相等待的现象
        - 产生死锁的必要条件
            - 互斥条件
            - 请求和保持条件
            - 不剥夺条件
            - 环路等待条件
        - 处理死锁的方法
            - 预防死锁
            - 避免死锁：银行家算法
            - 检测死锁：资源分配
            - 解除死锁：剥夺资源、撤销进程
    - 递归锁
        - 为了支持同一线程多次请求同一资源而设置
        - 只有当同一线程的所有请求都被释放后，其他线程才能获得此资源
    - 乐观锁
        - 假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性
    - 悲观锁
        - 假设会发生冲突，屏蔽一切可能违反数据完整性的操作
    - Python常用的加锁方式
        - 互斥锁，可重入锁，迭代死锁，相互调用死锁，自旋锁
13. Python的函数参数传递
    - 参数是可变对象
        - 相当于地址传递：直接改变参数的值
    - 参数是不可变对象
        - 相当于值传递：函数外面的值不会发生改变
14. Python能否重载
    - 重载：多个函数名相同的函数，根据不同的参数个数以及参数类型而执行不同的功能
    - 重载的实质：解决编程中的参数可变不统一问题
    - Python不需要重载
        - Python是动态语言，不需要声明变量类型，函数可以接收任意类型的参数，只需要在函数内部判断类型即可，不需要重新去写一个函数

15. Python和其他语言的区别
    - Python属于解释型语言：边解释边执行
    - C/Java/C#属于编译型语言：先编译再执行
    - Python由C开发而来
    - Python有强大的标准库和第三方库
16. Python解释器
    - Cpython：官方版本的解释器，由C语言开发的
    - IPython：基于Cpython的交互式解释器，只在交互方式上有所增强，内核依旧是Cpython
    - PyPy：由Python写的解释器，执行速度最快，它对代码进行动态编译
    - Jython：运行在Java平台的解释器
17. 常见的哈希碰撞解决方法
    - 开放寻址法
        - 如果当前的hash映射存在冲突，利用线性探测函数计算出下一个候选位置，如果候选位置依旧冲突，那么就继续利用线性探测函数进行计算，直到找到一个空位置来存放元素
    - 再哈希法
        - 定义多个哈希函数，每次查询时按照顺序调用哈希函数
            - 当调用到最后一个哈希函数都返回空时，返回不存在
            - 当调用到键值时，返回索引值
    - 链地址法
        - 将键值对应的哈希值相同的记录存储在同一链表中，按照顺序查找
18. 位和字节的关系
    - 1 Byte = 8 bit
    - 1 KB  = 1024 B
    - 1 MB = 1024 KB
    - 1 GB = 1024 MB
19. Python递归的最大层数
    - 998
20. ASCII、Unicode、UTF-8、GBK的区别
    - ASCII：一个字符只能用8位表示，因此ASCII最多只能表示256个字符
    - Unicode：任何一个字符用两个字节表示，也就是16位
    - UTF-8：中文字符用三个字节，英文字符用一个字节
    - GBK：中文用两个字节，英文用一个字节
21. 字节码和机器码的区别
    - 机器码：是CPU可以直接解读的数据
    - 字节码：是一种中间码，需要直译器转译后才能成为机器码
22. xrange和range的区别
    - xrange与range用法完全相同
    - 区别
        - range：生成数组
        - xrange: 生成生成器，性能更优
23. 文件操作：readlines和xreadlines的区别
    - readlines：返回列表
    - xreadlines：返回生成器
24. global语句的作用
    - global可以实现定义全局变量的作用
25. lambda匿名函数的作用
    - 精简代码，省去了定义函数的过程
26. Python 错误处理
    - Python内置了try…except…finally错误处理机制
        - 机制：当我们感觉某一段代码可能会出错时，就用try来运行这段代码，如果执行出错，则后续代码不再执行，直接跳转至异常处理代码，即except语句块，执行完except语句块后，如果包含finally语句块，则继续执行finally语句块
    - Python内置错误类型

- `IOError`：输入输出异常
- `AttributeError`：试图访问一个对象没有的属性
- `ImportError`：无法引入模块或包，基本是路径问题
- `IndentationError`：语法错误，代码没有正确的对齐
- `IndexError`：下标索引超出序列边界
- `KeyError`：试图访问你字典里不存在的键
- `SyntaxError`：Python 代码逻辑语法出错，不能执行
- `NameError`：使用一个还未赋予对象的变量

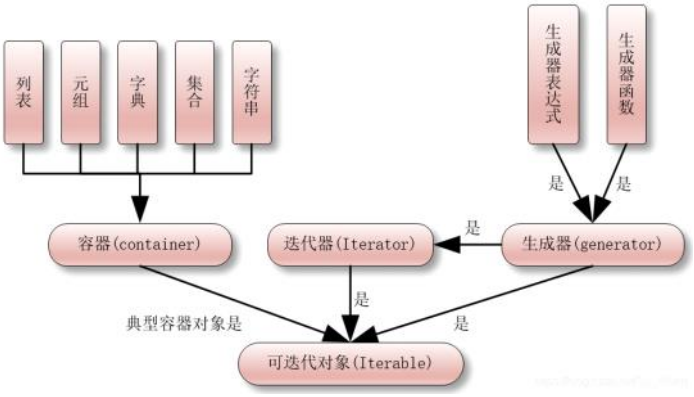
27. 布尔值为False的常见值
    - 0、''、{}、[]、()、set()、None、不成立的表达式
28. 匿名函数lambda
    - 格式：函数名=lambda 参数1，参数2：返回值
    - 匿名函数不管多复杂，只能写一行
29. pass的作用
    - pass是空语句，一般用作占位语句
    - 为了保持程序结构的完整性
30. any()和all()方法的区别
    - any()：只要迭代器中有一个元素为真就为真
    - all()：迭代器中所有判断项返回都为真，结果才为真
31. map()和reduce()函数区别
    - map()
        - 用法：接收两个参数（函数，可迭代对象），map将函数依次作用到可迭代对象Iterable的每个元素，结果返回为新的Iterator（惰性序列），通过list转化为常用的列表结构
        - 举例：

```
# 示例1
def square(x):
    return x ** 2
r = map(square, [1, 2, 3, 4, 5, 6, 7])
squareed_list = list(r)
print(squareed_list)  # [1, 4, 9, 16, 25, 36, 49]
# 使用lambda匿名函数简化为一行代码
list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
# 示例2
list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))#['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

    - reduce()
        - 用法：接收两个参数（函数，可迭代对象），reduce将前一个元素的结果继续和序列的下一个元素做累积计算（`reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)`）
32. 提高Python运行效率的方法
    - 使用生成器，节约大量内存
    - 循换代码优化，避免过多重复代码的执行
    - 核心代码模块用Cpython、PyPy解释器，提高效率
    - 多进程，多线程，协程
    - 当有多个if elif条件判断时，将最优可能发生的条件写在前面，这样可以减少程序判断的次数，提高效率
33. *arg和**kwarg的作用
    - *arg：返回的是元祖，接受任意多个无名参数，并把这些参数作为元祖传递给函数
    - **kwarg：返回的是字典，接收任意多个关键字参数
    - *arg和**kwarg同时使用时，*arg参数列要在**kwarg之前
34. Python垃圾回收机制
    - 垃圾
        - 当一个变量已经调用完毕，且后续不再需要时，成为垃圾
        - 当原本指向该内存地址的变量名指向另一个地址时，原本内存地址无法访问，此时内存变量成为垃圾
    - 垃圾回收机制
        - 引用计数
            - 为每一个对象定义一个内部跟踪变量，来记录当前对象被引用的次数
            - 一旦引用计数器的值为0，解释器会在适当的时机将垃圾对象占用的空间回收
            - 引用计数的缺陷
                - 循环引用：两个对象互相引用，但没有其他变量引用他们
                - 此时这两个对象的引用计数不会为0

        - 标记清除
            - 用于解决循环引用问题
            - 步骤：
                - 寻找根对象的集合，根对象就是一些全局引用和函数栈的引用
                - 遍历根对象集合，将每一个引用可以直接访问和间接访问的对象标记为存活对象，其余为非存活对象
                - 将非存活对象清除
        - 分代回收
            - 基于引用计数器的回收机制，在每一次内存回收过程中，都需要将对象的引用计数遍历一遍，非常耗时
            - 分代回收通过“空间换时间”的策略提升效率
            - 原理
                - 如果一个对象在经历过多次扫描之后都没有被回收，那么该变量被认为是常用变量，对其的扫描频率会降低
                - 分代：根据变量的存活时间对变量划分等级，不同的代扫描的频率不同
            - 步骤：
                - Python将对象分为三个代，每个代可以看做是一条可回收对象形成的链表
                - 第0代：最多容纳700个对象，一旦超过700阈值，就会触发垃圾回收机制
                - 第1代：当第0代触发了10次垃圾回收机制时，第1代的垃圾回收被触发，清理第1代链表时会顺带清理第0代
                - 第2代：当第1代触发了10次垃圾回收机制时，第2代的垃圾回收被触发，清理第2代链表时会顺带清理第1代和第0代
35. Python的可变类型和不可变类型
    - 可变类型：字典，列表， 可变集合（set()：集合中的元素可以动态的增加或删除）
    - 不可变类型：字符串，数字，元祖，不可变集合（frozenset():集合中的元素不可改变）
36. 安装第三方库的方法
    - pip包管理器
    - 源码下载
        - 下载+解压
        - 执行语句：python setup.py build -python setup.py install
37. 正则表达式的贪婪匹配
    - 匹配一个字符串没有限制，能匹配多少就去匹配多少，直到没有匹配的为止
38. Logging模块的作用
    - 应用场景：模块中定义的函数和类，可以灵活的为应用程序引入日志系统
    - 作用
        - 了解程序运行情况是否正常
        - 当程序出现故障时，快速定位出错位置以及故障分析
39. 常用的字符串格式化方式
    - 占位符
        - 整数：%d
        - 浮点数：%f
        - 字符串：%s
        - `print('Hello,%s' % 'Python')`
        - `print('Hello,%d%s%.2f' % (666, 'Python', 9.99)) # 打印：Hello,666Python10.00`
    - format
        - `print('{k} is {v}'.format(k='python', v='easy')) # 通过关键字`
        - `print('{0} is {1}'.format('python', 'easy')) # 通过关键字`
40. 简述容器、生成器、迭代器、可迭代对象
    - 容器
        - 容器就是一个用来存储多个元素的数据结构
        - 特点：
            - 容器中的元素可通过迭代获取。
            - 所有容器中的元素被存储在内存中
    - 可迭代对象Iterable
        - 一般为数据容器：list、set、dict、string、tuple
        - 特点
            - 定义了可返回迭代器的__iter__()方法
        - iterable必须有__iter__()函数，这个函数的作用就是返回迭代器iterator
    - 迭代器Iterator
        - 迭代器是一个带状态的对象

        - 特点
            - 实现了__iter__()和__next__()函数、调用__next__()函数会返回下一个元素
            - 迭代器有具体的迭代器类型：list_iterator、set_iterator
            - 迭代器不会一次性把所有元素加载到内存，而是需要时才生成返回结果
    - 生成器
        - 一种特殊的迭代器
        - 特点
            - 生成器拥有迭代器的迭代传出数据的功能、生成器通过yeild代替迭代器中的__next__()函数
            - 与迭代器的不同：生成器可以传入数据进行计算，并根据变量内容计算结果后返回
            - 生成器可以通过for循环迭代
    - 图解




| 对象 | 特点 | 功能 |
| --- | --- | --- |
| Iterable | 实现 `__iter__()` 方法 | 返回一个Iterator |
| Iterator | ①实现 `__iter__()` 和 `__next__()` 方法<br>②数据需要的时候才存储到内存中 | 逐个传出数据 |
| generator | ①通过 `yield` 替换迭代器两个方法并有中断功能<br>②数据需要的时候才存储到内存中 | ①逐个传出数据<br>**②逐个传入数据** |
| container | ①容器中的元素可通过迭代获取<br>②所有容器中的元素被存储在内存中 | —— |

41. OS模块与Sys模块的作用
    - OS模块：负责程序与操作系统的交互，提供了访问操作系统底层的接口
    - Sys模块：负责程序与Python解释器的交互，提供了一系列的函数和变量，用于操控Python运行时的环境
42. 生成随机数

```
import random
print(random.random()) # 用于生成一个0到1的随机浮点数: 0 <= n < 1.0
print(random.randint(1, 1000)) # 用于生成一个指定范围内的整数
```

43. 面向对象的理解
    - 面向对象是一种编程思想，将有共同属性和方法的事物封装到同一个类下面
    - 三大特性
        - 继承：将多个类的共同属性和方法封装到父类中，然后子类继承父类的属性和方法
        - 封装：将共同的属性和方法封装到同一个类中
        - 多态：基类的同一个方法在不同的派生类中有不同的功能
44. Python中的继承
    - 继承的实现方式
        - 实现继承：使用基类的属性和方法，无需额外的编码
        - 接口继承：子类仅使用父类的属性和方法名称，子类提供方法和属性的实现（子类重构父类方法）
45. 面向对象中super的作用
    - 用于子类继承基类的方法
46. functool函数

- 用于修复装饰器

47. Python的魔法方法
    - 会在特定的情况下自动调用，且方法名通常被双下划线包裹
48. 列举面向对象中带双下划线的方法
    - __new__：生成实例
    - __init__：生成实例的属性
    - __call__：实例对象加（）会执行def __call__函数中的内容
    - __del__：析构函数，当对象在内存中被释放时，自动触发执行
    - __get__：调用一个属性时触发
    - __set__：为一个属性赋值时触发
    - __delete__：采用del删除属性时触发
49. @staticmethod和@classmethod的区别
    - @classmethod：类方法的第一个参数必须是指向自身的 cls 参数
    - @staticmethod：可以没有任何参数
    - 实例方法：实例方法的第一个参数必须是指向对象自身的self参数，只能被实例对象调用
    - 静态方法和类方法可以被 类 或者 类的实例对象 调用
50. __new__和__init__方法的区别
    - __new__方法是类的构造函数，用于创建对象并返回对象
    - 当返回对象时自动调用__init__方法进行初始化
    - __new__方法比__init__方法更早执行
    - __new__方法是静态方法，而__init__方法是实例方法
51. 为什么说Python是动态语言
    - Python中的变量本身没有类型，可以反复赋值为不同类型的变量
    - 静态语言：定义变量时必须指明变量类型，如果赋值时类型不匹配，就会报错
52. metaclass作用以及应用场景
    - metaclass用于指定类是谁创建的
53. Python装饰器的理解
    - 本质：装饰器本质上是Python的一个类或者函数
    - 作用
        - 让其他函数或者类在不添加任何代码修改的情况下增加额外的功能（为已经存在的对象添加额外功能）
        - 抽离出大量与函数本身功能无关的雷同代码封装到装饰器中，可以继续重用，装饰器的返回值也是一个函数/类对象
    - 应用场景
        - 插入日志，缓存处理，权限校验，性能测试，事务处理
    - 实现
        - @符号
            - 用于装饰器中，修饰一个函数，把被修饰的函数作为参数传递给装饰器
            - 用作类方法及静态方法
54. 装饰器的写法以及应用场景
    - 装饰器：装饰器本质是函数，为其他函数添加附加功能
    - 原则
        - 不修改被修饰函数的代码
        - 不修改被修饰函数的调用方式
    - 应用场景
        - 无参数装饰器在用户登录认证中常见
        - 有参数装饰器在flask的路由系统中使用
55. assert断言
    - 作用：当条件成立时，则继续往下执行，否则抛出异常
56. 简述OSI七层协议
    - 物理层：实现终端信号的传输，码流通过物理介质传输
    - 数据链路层：把位元合并为字节，然后将字节合并到帧中，以使媒体能够通过数据链路层访问
    - 网络层：IP选址及路由选择
    - 传输层：建立、维护、管理端到端的通信
    - 会话层：建立客户端与服务端连接
    - 表示层：数据格式转化，对数据进行加密

- 应用层：为应用程序提供服务

57. 什么是C/S架构和B/S架构
    - C/S架构：客户端与服务端的架构
    - B/S架构：浏览器端和服务端架构
    - 优点：统一了所有应用程序的入口，方便，轻量级
58. 三次握手和四次挥手流程
    - 三次握手（建立连接）
        - 第一次握手：客户端向服务器端发起的一次建立连接请求，并随机生成一个值作为标识
        - 第二次握手：服务器向客户端回应第一个标识，再重新发一个确认标识
        - 第三次握手：客户端确认标识，建立连接，开始传输数据
    - 四次挥手（断开连接）
        - 第一次挥手：客户端向服务器发起断开连接请求
        - 第二次挥手：服务器向客户端确认请求
        - 第三次挥手：服务器向客户端发起断开连接请求
        - 第四次挥手：客户端向服务器确认断开连接请求
59. TCP和UDP区别
    - TCP协议：流式协议、面向连接，保证高可靠性传输层协议、可靠传输
    - UDP协议：数据报协议、存在数据丢失、无秩序的传输层协议、不可靠传输
    - TCP通信比UDP通信更可靠
        - TCP：只有对方回复了确认收到信息，才发下一个，如果没收到确认信息就重发
        - UDP：不可靠，一直发数据，不需要对方回应
60. 防火墙以及其作用
    - 防火墙：监控内部网与Internet之间的任何活动，保证内部网络的安全
    - 作用：
        - 防火墙是网络安全的屏障
        - 对网络存取和访问进行监控审计
        - 防止内部信息的外泄
61. 进程，线程以及协程的区别
    - 进程
        - 进程有自己独立的堆和栈，堆和栈都不共享，进程由操作系统调度
    - 线程
        - 线程有自己独立的栈和共享堆，共享堆不共享栈，线程也由操作系统调度
    - 协程
        - 协程避免了无意义的调度，可以提高性能，但协程也失去了线程使用多CPU的能力
    - 进程与线程区别
        - 地址空间：线程是程序的一个执行单元，进程内至少有一个线程，多个线程共享进程的地址空间，而进程有自己独立的地址空间
        - 资源拥有：进程是资源分配的基本单元，同一个进程内的多个线程共享进程的资源
        - 线程是处理器调度的基本单位
        - 进程和线程都可以并发执行
    - 协程和线程的区别
        - 一个线程可以有多个协程，一个进程也可以单独拥有多个协程，这样Python就可以使用多核CPU
        - 线程和进程都是同步机制，而协程是异步
        - 协程可以保留上一次调用时的状态
62. 并行和并发
    - 并发：同一时刻只能处理一个任务，但一个时间段内可以对多个任务交替处理（一个处理器同时处理多个任务）
    - 并行：同一时刻可以处理多个任务（多个处理器或者多核处理器同时处理多个不同的任务）
63. yeild和yeild from
    - yeild：当一个函数中出现yeild关键字时，该函数就是一个生成器，可以用for循换或者next()函数来迭代
    - yeild from
        - yeild from后面一定是iterable
        - `yield from iterable`就是`for item in iterable: yield item`的语法糖
64. Python中with语句的用法
    - 动机
        - 有一些任务，执行结束后需要做清理工作

        - 举例
            - 文件读写完毕后，需要调用close()方法关闭文件（因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的）
            - 由于文件读写时有可能产生IOError，一旦出错，后面的file.close()不会被调用

```
file = open("/tmp/foo.txt")
data = file.read()
file.close()
```

            - 为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally捕捉异常、处理异常（若能保证文件打开没有异常的情况下，每次都这么写很繁琐）

```
file = open("/tmp/foo.txt")
try:
    data = file.read()
finally:
    file.close()
```

    - 目的：简化Try...Finally模式，提供更方便的处理方式，引入with语句自动调用close()方法
    - with所求值的对象必须有一个__enter__()方法，一个__exit()__方法
        - 当with语句在开始执行时，会在上下文管理器对象上调用 __enter__()方法
        - 当with语句执行结束后，会在上下文管理器对象上调用 __exit__()方法

```
with open("/tmp/foo.txt") as file:
    data = file.read()
```

65. Python内存管理机制
    - 内存管理包括：内存分配+内存回收两部分
    - 内存池（内存分配）
        - 动机
            - 当创建大量的消耗小容量内存的对象时，频繁调用new/malloc将会导致大量的内存碎片，降低执行效率
        - 作用
            - 内存池预先在内存中申请一定数量的，大小相等的内存块留作备用
            - 当有新的内存申请需求时，先从内存池中分配内存，不够再申请新的内存
            - 从而减少内存碎片，提高效率
    - 内存池工作机制
        - 对象管理主要涉及Level+1-Level+3层
        - Level+3
            - 主要存储Python内置对象（int，list，dict，string等）
            - 每个内置对象都有属于自己的独立的私有内存池，且不共享
        - Level+2
            - 当申请内存容量<256KB时，内存分配主要由Python对象分配器实施
        - Level+1
            - 当申请内存容量>256KB时，内存分配由Python原生的内存分配器进行分配
            - 本质上是调用C标准库中的malloc函数


66. None、False、0、''的区别
    - None是Python的一个关键字
    - None也是一个类，None本身也是个一个数据类型
    - 0、"" 以及 False 都只是对象

67. Python设计模式
    - 设计模式
        - 代码设计经验的总结
        - 为了实现代码重用
    - 实现方式
        - 接口
            - 一种特殊的类
            - 要求继承该接口的类必须实现这种方法
        - 单例模式
            - 确保某一个类只有一个实例存在
            - 实现
                - 模块
                    - 需要实现的单例功能放到一个.py 文件中
                - 装饰器
        - 工厂模式
            - 简化对象的创建
            - 不同于类实例化直接创建对象，工厂模式通过一个中心化函数来实现
            - 实现
                - 工厂方法
                    - Python中的一个函数，对不同的输入参数返回不同的对象
                - ◇ 

        - 建造者模式
            - 建造者模式将所有细节都交由子类实现
                - Eg：画人物，要求画一个人的头，左手，右手，左脚，右脚和身体，画一个瘦子，一个胖子
                - 依次得画六个部位
            - 实现
                - 运用了抽象方法的特性，父类定义了几个抽象的方法，子类必须要实现这些方法
                    - Python本身不提供抽象类和接口机制
                    - 要想实现抽象类，可以借助abc模块（Abstract Base Class）
                    - 被@abstractmethod装饰为抽象方法后，该方法不能被实例化
                    - 除非子类实现了基类的抽象方法，才能实例化
68. Python的name属性
    - __name__
        - Python的内置属性，表示应用程序的名称
    - 两种情况
        - 当Python程序自己执行时，__name__变量的值就是：__main__
        - 当python程序是作为模块被导入时，那么__name__变量的值就是：程序的文件名，也就是.py前面的文件名称
        - 举例
            - 假设模块A、B，模块A自己定义了功能C，模块B调用模块A，现在功能C被执行
            - 如果C被A自己执行，也就是说模块执行了自己定义的功能，那么 __name__ == '__main__'
            - 如果C被B调用执行，也就是说当前模块调用执行了别的模块的功能，那么__name__ == 'A'
69. Python面向对象
    - 类变量
        - 定义在__init__()函数之前的变量
        - 类变量的值在类的所有实例之间共享
    - 类的私有属性
        - 双下划线开头，声明属性为私有属性
        - 允许这个类本身进行访问

- 可以使用 object._className__attrName（对象名._类名__私有属性名）访问属性
  - ```
    class Runoob:
        __site = "www.runoob.com"
    ```
  - ```
    runoob = Runoob()
    print runoob._Runoob__site
    ```
- 类的私有方法
  - 双下划线开头，声明方法为私有方法
  - 不能在类的外部调用
- 下划线方法
  - 头尾双下划线方法：魔法方法
  - 头双下划线方法/属性：私有类型变量, 只能是允许这个类本身进行访问
  - 头单下划线方法/属性：protected 类型的变量，只能允许其本身与子类进行访问
- 重写
  - 在子类中直接重写

70. OS模块
    - 当前路径及路径下的文件
      - `os.getcwd()`：查看当前所在路径。【具体到当前脚本的上一级】

        `os.listdir(path)`：列举path目录下的所有文件。返回的是列表类型。
      - ```
        import os

        os.getcwd() # 'D:\\pythontest\\ostest'

        os.listdir(os.getcwd()) #['hello.py', 'test.txt']
        ```
    - 绝对路径
      - `os.path.abspath(path)`：返回当前文件位置的绝对路径。

        `os.path.realpath(path)`：返回当前文件位置的绝对路径。

        举例1：
      - ```
        import os

        print(os.getcwd())

        print(os.path.abspath('.'))

        print(os.path.abspath('..'))

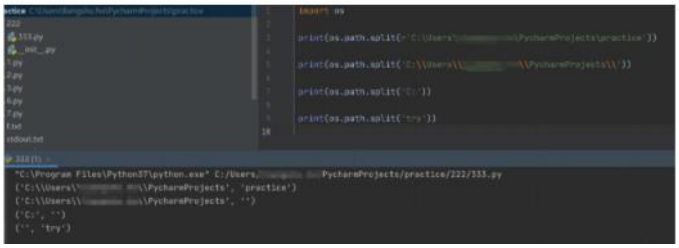
        print(os.path.abspath('../..'))
        ```

        运行结果：

        ```
        "C:\Program Files\Python37\python.exe" C:/Users/[redacted]/PycharmProjects/practice/222/333.py
        C:\Users\[redacted]\PycharmProjects\practice\222
        C:\Users\[redacted]\PycharmProjects\practice\222
        C:\Users\[redacted]\PycharmProjects\practice
        C:\Users\[redacted]\PycharmProjects
        ```
    - 文件夹的路径和文件名
      - `os.path.split(path)`：将指定文件的路径分解为(文件夹路径, 文件名)，返回的数据类型是**元组**类型。

        ①若文件夹路径字符串最后一个字符是\，则只有文件夹路径部分有值；

        ②若路径字符串中均无\，则只有文件名部分有值。

        ③若路径字符串有\，且不在最后，则文件夹和文件名均有值。且返回的文件名的结果不包含\.
      - 
    - 路径拼接
      - ```
        os.path.join('D:\\pythontest', 'ostest') #  'D:\\pythontest\\ostest'

        os.path.join('D:\\pythontest\\ostest', 'hello.py')  #  'D:\\pythontest\\ostest\\hello.p

        os.path.join('D:\\pythontest\\b', 'D:\\pythontest\\a') #  'D:\\pythontest\\a'
        ```
    - 返回路径中包含的文件夹路径部分
        ```

`os.path.dirname(path)` ：返回path中的文件夹路径部分，且路径结尾不包含'\'。【即返回文件的路径（此路径不包含文件名）】

```
os.path.dirname('D:\\pythontest\\ostest\\hello.py')  #  'D:\\pythontest\\ostest'

os.path.dirname('.')  #  ''

os.path.dirname('D:\\pythontest\\ostest\\')  #  'D:\\pythontest\\ostest'

os.path.dirname('D:\\pythontest\\ostest')  #  'D:\\pythontest'
```

- 获取路径中的文件名

`os.path.basename(path)` ：返回path中的文件名。

```
os.path.basename('D:\\pythontest\\ostest\\hello.py')  #  'hello.py'

os.path.basename('.')  #  '.'

os.path.basename('D:\\pythontest\\ostest\\')  #  ''

os.path.basename('D:\\pythontest\\ostest')  #  'ostest'
```

- 查看文件大小

`os.path.getsize(path)` ：返回文件的大小；若path入参值是一个文件夹路径则返回0。

```
os.path.getsize('D:\\pythontest\\ostest\\hello.py')  #  58L

os.path.getsize('D:\\pythontest\\ostest')  #  0L
```

- 查看文件是否存在

`os.path.exists(path)` ：判断文件或者文件夹是否存在，返回True 或 False。【文件或文件夹的名字不区分大小写】

```
os.listdir(os.getcwd())  #  ['hello.py', 'test.txt']

os.path.exists('D:\\pythontest\\ostest\\hello.py')  #  True

os.path.exists('D:\\pythontest\\ostest\\Hello.py')  #  True

os.path.exists('D:\\pythontest\\ostest\\Hello1.py')  #  False
```

- 操作目录以及文件的函数

`os.mkdir('文件夹名')` ：新建文件夹；入参为目录路径，不可为文件路径；（父目录必须存在的情况下创建下一级文件夹）

`os.rmdir('文件夹名')` ：删除文件夹；入参为目录路径，不可为文件路径

`os.remove('文件路径')` ：删除文件；入参为文件路径，不可谓目录路径

`os.makedirs('路径及文件')` ：递归新建文件夹；可以连续创建该文件夹的多级目录

`os.path.isdir('路径')` ：判断入参路径是否为文件夹，返回值为布尔值；是文件夹返回True，不是文件夹返回False

`os.path.isfile('路径')` ：判断入参路径是否为文件，返回值为布尔值；是文件返回True，不是文件返回False

- OS遍历目录数
  - os.listdir()：当一个目录下只有文件时使用

```
path = r'C:\Users\XXN\Desktop\test_file'

for each_file in os.listdir(path):

    print(os.path.join(path,each_file))
```

  - os.walk()：当一个文件下面既有文件、又有目录时使用，返回所有文件

```
path = r'C:\Users\XXN\Desktop\test_file'

for parent,dirnames,filenames in os.walk(path):

    print(parent,dirnames,filenames)
```

- 总结
  - `# python--函数 os.sep ： 主要用于系统路径中的分隔符`

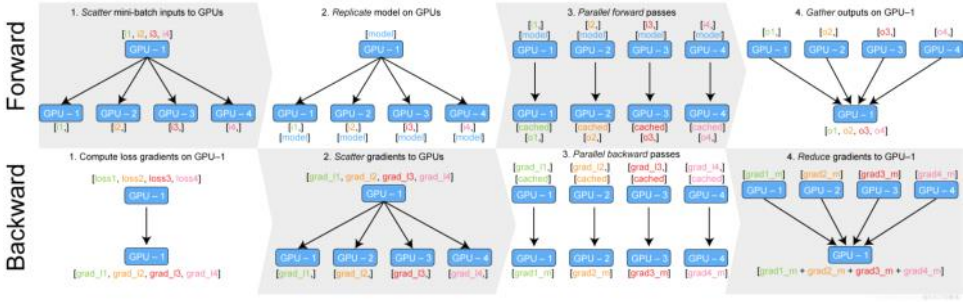
```
# Windows系统通过是“\\”，Linux类系统如Ubuntu的分隔符是“/”，而苹果Mac OS系统中是“:”

# 常用的os模块命令：
# 1、os.name——name顾名思义就是'名字'，这里的名字是指操作系统的名字，主要作用是判断目前正在使用的平台，并给出操作系统的名字，如Windows 返回 'nt'; Linux 返回'posix'。注意该命令不带括号。
# 2、os.getcwd()——全称应该是'get current work directory'，获取当前工作的目录
# 3、os.listdir(path)——列出path目录下所有的文件和目录名。Path参数可以省略。
# 4、os.remove(path)——删除path指定的文件，该参数不能省略。
# 5、os.rmdir(path)——删除path指定的目录，该参数不能省略。
# 6、os.mkdir(path)——创建path指定的目录，该参数不能省略。
# 7、os.unlink() 方法用于删除文件,如果文件是一个目录则返回一个错误

# os.remove()         #删除文件
# os.rename()         #重命名文件
# os.walk()         #生成目录树下的所有文件名
# os.chdir()         #改变目录
# os.mkdir/makedirs()       #创建目录/多层目录
# os.rmdir/removedirs       #删除目录/多层目录
# os.listdir()          #列出指定目录的文件
# os.getcwd()         #取得当前工作目录
# os.chmod()         #改变目录权限
# os.path.basename()       #去掉目录路径，返回文件名
# os.path.dirname()       #去掉文件名，返回目录路径
# os.path.join()        #将分离的各部分组合成一个路径名
# os.path.split()        #返回(dirname(),basename())元组
# os.path.splitext()       #返回filename,extension)元组
# os.path.getatime\ctime\mtime    #分别返回最近访问、创建、修改时间
# os.path.getsize()       #返回文件大小
# os.path.exists()        #是否存在
# os.path.isabs()        #是否为绝对路径
# os.path.isdir()        #是否为目录
# os.path.isfile()        #是否为文件
# os.system('command') 会执行括号中的命令，如果命令成功执行，这条语句返回0，否则返回1
```

# PyTorch

2022年5月11日 21:37

1. PyTorch中cuda()的作用
   - 将张量放在GPU显存中加速
   - CPU中的张量与GPU中的张量计算会报错
   - 数据默认存放在CPU上
2. 训练神经网络为什么用GPU
   - 神经网络中的计算大都是矩阵的加法和乘法运算，而GPU主要负责图形运算，图形运算主要基于矩阵运算
   - CPU可以支持很多复杂的逻辑运算，但核心数往往较少，运行矩阵运算的话需要较长的时间
3. PyTorch加快训练速度的方法
   - Dataloader数据加载时，设置numworker>0，多线程读取数据
   - 最大化Batchsize
   - 使用更优的优化器
   - 避免CPU和GPU之间频繁的数据传输
     - 小心使用tensor.cpu()和tensor.cuda()频繁地将张量在GPU和CPU之间相互转换
     - 使用关键字参数device=torch.device('cuda:0')直接将它分配给你的GPU
   - 多GPU分布式数据并行
     - 使用torch.nn.DistributedDataParallel而不是torch.nn.DataParallel
     - ```
       model = model.cuda()
       device_ids = [0, 1]          # id为0和1的两块显卡
       model = torch.nn.DataParallel(model, device_ids=device_ids)
       ```
4. Model.eval()的作用
   - 在模型测试阶段，需要在计算推理之前添加Model.eval()语句
   - 作用
     - 不进行Dropout操作
     - 不进行BN层中均值和方差的更新
     - 不再进行梯度回传（但梯度仍然会计算）
5. Torch.no_grad()的作用
   - 不反向回传，不进行梯度计算，节省算力
   - 但会进行Dropout和BN的计算
   - ```
     with torch.no_grad():
         # your code
     ```
6. Pytorch多GPU训练
   - 
     
   - 多卡训练流程
     - 指定主GPU节点
     - 主GPU划分数据，一个batch数据平均分到每个GPU上
     - 将模型从主GPU拷贝到各个GPU
     - 每个GPU分别进行前向传播，将结果汇总到主GPU
     - 主GPU计算loss损失，并回传计算梯度，进行参数更新
     - 将参数更新后模型拷贝给各个GPU
   - 实例
     - 假设模型输入为(32, input_dim)，模型输出为(32, output_dim)
     - 使用 4 个GPU训练。nn.DataParallel将这 32 个样本拆成 4 份，发送给 4 个GPU 分别做 forward，生成 4 个大小为(8, output_dim)的输出
     - 将这 4 个输出都收集到cuda:0上并合并成(32, output_dim)。
     - 弊端：后续的loss计算只会在cuda:0上进行，没法并行，因此会导致负载不均衡的问题。
     - 举例
       - ```
         import os
         import torch
         args.gpu_id="2,7" ; #指定gpu id
         args.cuda = not args.no_cuda and torch.cuda.is_available() #作为是否使用cpu的判定
         #配置环境  也可以在运行时临时指定 CUDA_VISIBLE_DEVICES='2,7' Python train.py
         os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu_id #这里的赋值必须是字符串，list会报错
         device_ids=range(torch.cuda.device_count())  #torch.cuda.device_count()=2
         #device_ids=[0,1] 这里的0 就是上述指定 2，是主gpu，  1就是7,模型和数据由主gpu分发

         if arg.cuda:
             model=model.cuda()  #这里将模型复制到gpu ,默认是cuda('0'), 即转到第一个GPU 2
         if len(device_id)>1:
             model=torch.nn.DaraParallel(model);#前提是model已经.cuda() 了

         #前向传播时数据也要cuda(),即复制到主gpu里
         for batch_idx, (data, label) in pbar:
             if args.cuda:
                 data,label= data.cuda(),label.cuda();
             data_v = Variable(data)
             target_var = Variable(label)
             prediction= model(data_v,target_var,args)
             #这里的prediction 预测结果是由两个gpu合并过的，并行计算只存在在前向传播里
             #前向传播每个gpu计算量为 batch_size/len(device_ids),等前向传播完了将结果和到主gpu里
             #prediction length=batch_size

             criterion = nn.CrossEntropyLoss()
             loss = criterion(prediction,target_var) #计算loss
             optimizer.zero_grad()
             loss.backward()
             optimizer.step()
         ```
   - 多GPU负载不均衡问题
         ```

- 通过在模型内置loss计算可以解决上述负载不均衡的情况，最后所得loss进行取平均
- 按照提到的模型并行逻辑，在每个GPU上会计算出一个loss，这些loss会被收集到cuda:0上并合并成长度为 4 的张量。这个时候在做backward的之前，必须对将这个loss张量合并成一个标量，一般直接取mean就可以。
- 举例：
  - ```
    class Net:
        def __init__(self,...):
            # code

        def forward(self, inputs, labels=None)
            # outputs = fct(inputs)
            # loss_fct = ...
            if labels is not None:
                loss = loss_fct(outputs, labels)  # 在训练模型时直接将labels传入模型，在forward过程中计算loss
                return loss
            else:
                return outputs
    ```

7. PyTorch多个loss的backward问题
   - 原则上，PyTorch的一张计算图中只能有一次梯度回传，每次梯度回传时，中间变量会被直接释放掉
   - 解决办法：
     - 在backward中加入retain_grad=True【backward(retain_graph=True)】
       - 暂时不释放计算图，在后续的训练过程中计算图不会被释放掉，而是会一直累积
     - 需要在最后一个loss计算时，把(retain_graph=True)去掉，只使用backward()，
       - 即除了最终的loss要释放资源、计算梯度，前面若干个的loss都不进行此步骤
   - ```
     for j in range(len(output)):
         loss += criterion(output[j], target_var)
     loss.backward()
     ```
8. PyTorch网络创建，模型加载
   - 构建网络
     - 添加子模型到网络中
       - add_module()方法：单独添加一个 Module，可以使用属性或者方法
       - ModuleList()方法：将一个 Module 的 List 加入到网络中，自由度较高，但是需要手动的遍历 ModuleList 进行 forward
       - Sequential()方法：按照顺序将 Module 加入到网络中，也可以处理字典。 相比于 ModuleList 不需要自己实现 forward
       - 
       - 
       - 
       - ```
         # Example of using Sequential
               model = nn.Sequential(
                         nn.Conv2d(1,20,5),
                         nn.ReLU(),
                         nn.Conv2d(20,64,5),
                         nn.ReLU()
                       )

         # Example of using Sequential with OrderedDict
         model = nn.Sequential(OrderedDict([
             ('conv1', nn.Conv2d(1,20,5)),
             ('relu1', nn.ReLU()),
             ('conv2', nn.Conv2d(20,64,5)),
             ('relu2', nn.ReLU())
             ]))
         ```
     - 遍历网络
       - modules()方法：递归的返回网络中的所有Module
         - ```
           m = Model()
           for idx,m in enumerate(m.modules()):
               print(idx,"-",m)
           ```
       - children()方法：只返回当前模块的子模块，不会递归子模块
         - ```
           for idx,m in enumerate(m.children()):
               print(idx,"-",m)
           ```
     - 网络参数
       - parameters()方法：返回一个包含模型所有参数的迭代器
         - ```
           for p in m.parameters():
               print(type(p.data),p.size())
           ```
       - named_parameters()方法：返回参数的名称及参数本身
         - ```
           for k,v in m1.named_parameters():
               print(k,v.size())
           ```
     - 模型的保存与加载
       - 保存模型结构及参数
         - ```
           torch.save(model,'model.pth') # 保存
           model = torch.load("model.pth") # 加载
           ```
       - 只加载模型参数，网络结构从代码中创建
         - ```
           torch.save(model.state_dict(),"model.pth") # 保存参数
           model = model() # 代码中创建网络结构
           params = torch.load("model.pth") # 加载参数
           model.load_state_dict(params) # 应用到网络结构中
           ```
       - 加载预训练模型
           ```

```
resnet152 = models.resnet152(pretrained=True)
pretrained_dict = resnet152.state_dict()
"""加载torchvision中的预训练模型和参数后通过state_dict()方法提取参数
   也可以直接从官方model_zoo下载:
   pretrained_dict = model_zoo.load_url(model_urls['resnet152'])"""
model_dict = model.state_dict()
# 将pretrained_dict里不属于model_dict的键剔除掉
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}
# 更新现有的model_dict
model_dict.update(pretrained_dict)
# 加载我们真正需要的state_dict
model.load_state_dict(model_dict)
```

9. 数据加载以及预处理
   - Dateset类

     `torch.utils.data.Dataset`是一个抽象类，用户想要加载自定义的数据只需要继承这个类，并且覆写其中的两个方法即可：

     1. `__len__`： 覆写这个方法使得`len(dataset)`可以返回整个数据集的大小
     2. `__getitem__`： 覆写这个方法使得`dataset[i]`可以返回数据集中第`i`个样本

     ```
     from torch.utils.data import DataLoader, Dataset
     from skimage import io, transform
     import matplotlib.pyplot as plt
     import os
     import torch
     from torchvision import transforms
     import numpy as np

     class AnimalData(Dataset):
         def __init__(self, root_dir, transform=None):
             self.root_dir = root_dir
             self.transform = transform

         def __len__(self):
             return 20

         def __getitem__(self, idx):
             filenames = os.listdir(self.root_dir)
             filename = filenames[idx]
             img = io.imread(os.path.join(self.root_dir, filename))
             # print(filename[:-5]
             if (int(filename[:-5]) > 10):
                 lable = np.array([0])
             else:
                 lable = np.array([1])
             sample = {'image': img, 'lable':lable}

             if self.transform:
                 sample = self.transform(sample)
             return sample
     ```

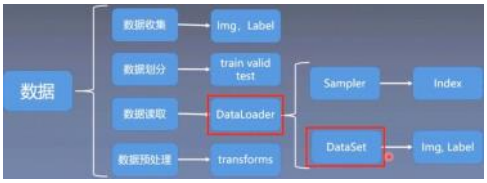
   - DataLoader类
     - 提供了对Dataset的读取工作，构建可迭代的数据加载器

   - 读取顺序
     - 原始数据 -> Dataset -> 通过Dataloader配置数据的读取条件
   - torchvision库
     - PyTorch专门用来处理图像的库
     - torchvision.datasets：包含了常用数据集
     - torchvision.models：包含了常用的基线网络模块
     - torchvision.transforms：包含了常用的图像转换操作，用于图像处理和图像增强

     ```
     print('########## 利用torchvision创建数据加载器 #########')
     data_transform = transforms.Compose([
         transforms.RandomSizedCrop(224),
         transforms.RandomHorizontalFlip(),
         transforms.ToTensor(),
         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0
     # ants_bees_dataset = datasets.ImageFolder(root='ants_bees_data/train', t
     # dataset_loader = DataLoader(ants_bees_dataset, batch_size=4, shuffle=Tr
     ```

   - PyTorch数据处理步骤

10. 反向传播

```
# 模型加载
tudui = Tudui()
# 损失函数
loss = nn.CrossEntropyLoss()
# 优化器
optim = torch.optim.Adam(tudui.parameters(), lr=0.0001)
for epoch in range(20):
    # 每一轮损失
    running_loss = 0.0
    for data in test_loader:
        # 加载数据
        imgs, targets = data
        # 将数据放入网络
        outputs = tudui(imgs)

        # 损失函数：网络输出(预测)、标签
        result_loss = loss(outputs, targets)

        # 优化器 梯度清零
        optim.zero_grad()
        # 反向传播
        result_loss.backward()
        # 调用优化器
        optim.step()

        # 累计损失
        running_loss += result_loss
        # \r() 可不换行直接显示，加上[]是为了好看一点
        print("\r[{}]".format(result_loss), end="")
    print(running_loss)
```

11. PyTorch中张量初始化
    - ```
      import numpy as np
      x = torch.randn((2, 3))
      x = np.zeros((5, 5))
      print(x)
      print(x.dtype)

      x_torch = torch.from_numpy(x)    # 从numpy变成torch张量类型
      print(x_torch)
      print(x_torch.dtype)

      x_back = x_torch.numpy()    # 从torch张量变回numpy类型
      print(x_back)
      print(x_back.dtype)
      ```
12. PyTorch模型部署
    - 基于Flask框架部署模型
      - Flask是一种Python编写的轻量级Web应用框架，它是Python应用程序和Web服务器之间的接口
        - Web应用框架：将不同Web应用程序中的共性部分抽象出来，提供一系列通用的接口，从而避免开发者做重复性工作
      - Flask框架的安装：`pip install flask`
      - Flask处理请求流程：
        - 客户端发送请求
        - 服务器端根据客户端URL选择相应的处理函数
        - 对应函数进行处理操作，得到响应数据
        - 由Flask返回响应内容
    - 举例
      - 任务：客户端发送一个样本数据给服务端，然后服务端再返回给客户端这个样本数据的检测结果（模型已经用pytorch框架离线训练好了）
      - 步骤（A作服务器，B作客户端）
        - 在A服务器上编写服务代码
          - ```
            from flask import Flask
            from flask import request

            # 创建一个flask类的实例，程序实例
            app = Flask(__name__)
            # 使用 route() 装饰器来告诉 Flask 触发函数的URL
            @app.route("/v1/", methods=["POST"])
            # "/v1/" 表示请求服务路径
            ```
        - 在A服务器上部署Flask服务
          - ```
            # -w 代表开启的进程数，我们只开启一个进程
            # -b 服务的IP地址和端口
            # app:app 是指执行的主要对象位置，在app.py中的app对象

            # 服务部署
            nohup gunicorn -w 1 -b 0.0.0.0:8001 app:app > ./ret.log 2>&1 &
            ```
        - 在B客户端调用部署好的服务
          - ```
            import requests
            import json
            import time

            url = " http://0.0.0.0:8001/v1/"
            data = {
                "id": 1234,
                "data": "这是一条测试数据！"
            }

            start = time.time()
            res = requests.post(url, json=data, timeout=200)
            end = time.time()
            ```
13. CV2函数
    - Cv2.imread()：读取图像
    - Cv2.imwrite()：保存图像
    - Cv2.split()：将图像数据分离为3个通道
    - Cv2.merge()：将三个颜色通道合并
    - Cv2.flip()：反转图像
    - Cv2.resize()：图像缩放
    - Cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)：RGE转灰度图
    - Cv2.cvtColor(img,cv2.COLOR_GRAY2RGB)：灰度图转RGB
14. Tensorboard模块
    - 可视化工具，可以用于记录训练数据、评估数据、网络结构、图像等
    - Tensorboard大致使用流程
      - 导入Tensorboard，实例化SummaryWriter类，指明记录路径等信息
        - ```
          from torch.utils.tensorboard import SummaryWriter
          writer = SummaryWriter(log_dir = '/logs')
          #调用实例
          writer.add_xxx()
          #关闭writer
          writer.close()
          ```
      - 调用相应的API，接口一般格式为：
          ```

        - `add_xxx(tag_name, object, iteration-number)`
    - 启动tensorboard，在命令行中输入
        - `tensorboard --logdir=路径`
    - 复制网址在浏览器中打开
  - 各种API
    - add_scalar()：绘制单条曲线
    - add_scalars()：绘制多条曲线
    - add_histogram()：绘制直方图
    - add_image()：加载图像
    - add_graph(model)：绘制模型结构
15. PyTorch模型部署
  - Web应用部署
    - 基于Flask框架进行封装，做一个简单的HTTP接口来对外提供服务
  - PyTorch提供的部署工具TorchServe
  - 腾讯的NCNN（适用于移动端部署）
  - NVIDIA的TensorRT
  - 考虑因素
    - 模型训练阶段：单机单卡，单机多卡
    - 部署阶段：多机多卡，并发
16. 多卡训练的时候Batchsize变大但精度下降的原因
  - 原因
    - 神经网络通常是非凸优化问题，损失函数包含多个局部最优点
    - 小的Batchsize有噪声的干扰，可能容易跳出局部最优点
    - 而大的Batchsize有可能停在局部最优点跳不出来
  - 解决方法
    - 增大学习率，跳出局部最优点，但可能导致模型不收敛
17. Torch.transpose和Tensor.permute的区别
  - Tensor.permute可以实现多个维度的交换
  - Torch.transpose只能进行两位维度的交换
  - 连续使用transpose也可实现permute的效果
  - ```
    In [1]: dat = torch.tensor(
                    [[[ 1,  2,  3,  4],
                      [ 5,  6,  7,  8],
                      [ 9, 10, 11, 12]],

                     [[13, 14, 15, 16],
                      [17, 18, 19, 20],
                      [21, 22, 23, 24]]])
    In [2]: dat.shape
    Out[2]: torch.Size([2, 3, 4])

    In [3]: dat_transpose = torch.transpose(dat, 1,2)
    In [4]: dat_transpose.shape
    Out[4]: torch.Size([2, 4, 3])

    In [5]: dat_permute = dat.permute((1,2,0))
    In [6]: dat_permute.shape
    Out[6]: torch.Size([3,4,2])
    ```
18. model中的forward函数是如何自动执行的
  - 条件
    - 我们在定义model的时候会继承nn.Module类，使得我们自定义的model中包含了父类的一些变量和方法
    - nn.Module类中定义了_call_impl()方法，包含result = self.forward(*input, **kwargs)语句
  - 调用步骤
    - 当父类nn.Module被调用执行时，首先调用self.__call__方法，触发_call_impl()的执行，从而执行result = self.forward(*input, **kwargs)
    - 从而使得自定义model，自动执行forward函数
  - __call__（）函数
    - 作用
      - 可以使得类的实例对象，以函数一样的形式被调用
      - 形式：对象名（）
      - 等价于：对象名.__call__()
    - 示例
      - ```
        class CLanguage:
            # 定义__call__方法
            def __call__(self,name,add):
                print("调用__call__()方法",name,add)

        clangs = CLanguage()
        clangs("C语言中文网","http://c.biancheng.net")
        ```
19. PyTorch动态图与Tensorflow静态图的区别
  - 区分方式：根据图的搭建方式不同，分为静态图和动态图
  - 静态图
    - 先搭建图的整个框架，再输入数据，一步一步按照流程进行计算
    - 高效但不灵活
  - 动态图
    - 运算与搭建同时进行，一步一步运算，不断的完善图结构
    - 灵活易调节
20. Tensor和Numpy区别
  - 两者都是矩阵形式
  - Tensor支持GPU计算，Numpy只能在CPU计算
  - Tensor默认数据类型float32，Numpy默认数据类型是Int64

# Linux

2022年5月11日 11:54

1. 文件和目录
    - cd命令
        - 作用：用于切换当前目录
        - 格式：cd+路径
    - pwd命令
        - 作用：显示工作路径
    - ls命令
        - 作用：查看目录中的文件
        - 格式：ls+目录名称
    - cp命令
        - 作用：复制文件
    - mv命令
        - 作用：用于移动文件或修改文件名
    - rm命令
        - 作用：用于删除文件或目录
    - mkdir命令
        - 创建文件夹
    - rmdir命令
        - 删除文件夹
2. 查看文件内容
    - cat命令
        - 作用：用于查看文本文件的内容
        - 格式：cat+文件名
3. 文件搜索
    - find命令
        - 作用：搜索文件以及目录
4. 文本处理
    - grep命令
        - 作用：全局正则表达式搜索
    - paste命令
        - 作用：合并两个文件
        - 格式：paste file1 file2
    - sort命令
        - 作用：排序两个文件内容取两个文件的并集
        - 格式：sort file1 file2
    - comm命令

        - 作用：比较两个文件删除共有内容
        - 格式：comm file1 file2
    - awk命令
        - 将文本按照指定的格式输出
        - awk [选项] '正则表达式 ' + 路径
5. 打包和压缩文件
    - tar命令
        - 作用：对文件进行压缩
        - 格式：
            - 压缩：tar -jcv -f filename.tar.bz2
            - 解压：tar -jxv -f filename.tar.bz2 -C
            - 压缩：bzip2 file1
            - 解压：bunzip2 file1.bz2
6. 系统和关机
    - 关闭系统：shutdown -h now
    - 重启：shutdown -r now
    - 注销：logout
7. 进程相关命令
    - jps命令
        - 作用：显示当前的java进程情况以及id号
    - ps命令
        - 作用：显示所有运行中的进程
    - kill命令
        - 用于向某个进程传送信号
        - 杀死进程：kill -9 pid
    - top命令
        - 实时动态显示系统中各个进程的资源占用情况
8. 内存相关命令
    - 磁盘空间使用情况
        - df -h命令
            - 显示文件系统容量，已用，可用，使用率
        - du -h命令
            - 显示当前位置文件夹容量大小
    - 内存空间使用情况
        - free -h命令
            - 显示内存使用情况
9. 显卡相关
    - 查看显卡使用情况：
        - nvidia-smi
        - watch -n 1 nvidia-smi
    - 查看显卡驱动

        - Lshw -c video | grep configuration
        - navidia -smi
    - 查看CUDA版本
        - cat /usr/local/cuda/version.txt
10. Anaconda相关命令
    - 创建虚拟环境
        - 创建环境：conda create --name 文件名 python=3.4
        - 激活环境：（source）activate 文件名
        - 显示python版本：python --version
        - 关闭环境：（source）deactivate 文件名
        - 删除环境：conda remove --name 文件名 --all
    - 安装第三方包
        - conda install 名称
        - pip install 名称
    - 卸载第三方包
        - conda remove 名称
        - pip uninstall 名称
    - 更新第三方包
        - conda update 名称
    - 设置镜像源
        - 添加镜像源：conda config --add channels +镜像源网址
        - 修改源：conda config --set show_channels_urls_yes
    - 查看conda安装的镜像源
        - conda config --show
    - 查看当前虚拟环境中第三方包的信息
        - conda list
    - 导入导出虚拟环境
        - 导出：conda env export > 文件名.yaml
        - 创建相同环境：conda env create -f 文件名.yaml
11. Github上传代码
    - 在文件夹根目录下右击打开 git bash
    - 文件夹初始化为仓库：Git init
    - 在github上新建repositories并复制SSH地址（没有SSH的网上百度安装）
    - 利用 lfs 过滤>100MB的文件
        - git lfs track "*.pth"
        - git add .gitattributes
        - git commit -m "lfs"
        - git remote add origin + github账户地址
        - git push -u origin master
    - git add 文件/文件夹 （将文件上传到暂存区）
    - git commit -m “说明信息” 上传到仓库
    - git push -u origin master 完成

# C/C++

2022年7月5日 9:06

1. C++中static关键字的作用
    - 定义静态函数和静态变量
    - 添加static关键字的静态函数和静态变量只能在本源文件中使用，保持变量内容的持久性
    - 未添加static的全局变量或函数具有全局可见性
2. 静态局部变量，全局变量和局部变量的区别
    - 作用域
        - 全局变量：全局作用域，可以作用于其他源文件（通过extern关键字引用后使用）
        - 静态全局变量：全局作用域+文件作用域，无法在其他文件中使用
        - 局部变量：局部作用域，函数内部等
        - 静态局部变量：局部作用域，只能被初始化一次，直到程序结束
    - 所在空间
        - 栈：局部变量
            - 出了作用域就回收内存
        - 静态存储区：全局变量，静态全局变量，静态局部变量
            - 直到程序结束才会回收内存
3. 虚函数和纯虚函数的区别
    - 含有纯虚函数的类称为抽象类（只包含虚函数的类不是抽象类）
    - 虚函数可以直接被使用，也可以被子类重载或者重写后以多态的形式调用
    - 纯虚函数只有在子类中实现之后，才可以使用（纯虚函数在基类中只有声明，没有定义）
4. 纯虚函数怎么定义
    - 纯虚函数是一种特殊的虚函数
    - 某些情况下，基类无法对虚函数进行有意义的实现，因此将它声明纯虚函数
    - 纯虚函数由基类的派生类实现
    - ```
      class <类名>
      {
      virtual <类型><函数名>(<参数表>)=0;
      …
      };
      ```
5. 在程序里面，智能指针的类型
    - Shared_ptr：允许多个指针指向同一个对象
    - Unique_ptr：独占所指向的对象
    - Weak_ptr：弱引用，指向shared_ptr所指向的对象
6. new和malloc的区别
    - 相同点
        - 两者都用于动态申请内存
    - 不同点
        - malloc和free是C/C++的标准库函数
        - new和delete是C++的运算符
        - new在使用时，先分配内存，后调用构造函数，释放时调用析构函数；malloc没有构造和析构函数
        - malloc需要给定申请内存的大小，new不需要指定大小
7. 函数后面接const是什么含义
    - 将整个函数修饰为const
    - 函数体内不能对成员函数做任何改动
8. const和define的区别
    - 相同点
        - const和define都可以用于定义常量
    - 不同点
        - const定义的常量是带类型的，define定义的常量是不带类型的
      ```

    - const定义的常量本质上是变量，拥有内存空间，define定义的常量不占用内存
9. 什么是常函数
    - 类的成员函数后边加const，表示该函数不会对类对象的数据成员做任何改变，只能读取数据，不能改变数据
    - 没有const修饰的成员函数，对数据成员则是可读可写
10. C++语言特点
    - C++在C语言的基础上引入了面向对象机制，同时兼容C语言
    - C++三大特性：继承，封装和多态
    - C++相比其他语言来说，运行效率更高
11. C与C++的区别
    - C是C++的子集，C++可以很好的兼容C，同时新增了很多特性：引用，智能指针，auto变量等
    - C++是面向对象的编程语言，C是面向过程的编程语言
    - C存在一些安全风险（强制类型转换的不确定性，内存泄露等），C++新增了const常量，引用，智能指针等，来改善安全性
12. C++中struct和class的区别
    - struct：用于描述数据结构的集合，默认访问权限是public
    - class：用于实现对象数据的封装，默认访问权限是private
13. include头文件中<>和“”的区别
    - <>：表示头文件是系统文件
        - 查找路径：编译器设置的头文件路径 -> 系统变量
    - ""：表示头文件是自定义文件
        - 查找路径：当前头文件目录 -> 编译器设置的头文件路径 -> 系统变量
14. C的结构体与C++结构体的区别
    - 继承
        - C的结构体不可以继承
        - C++结构体可以继承自其他结构体或者类
    - 使用
        - C中使用结构体时需要加上struct关键字，或者通过typedef取别名
        - C++可以省略struct关键字，直接使用
15. C++与C语言从代码到可执行二进制文件的转换过程
    - 预编译：对各种预处理指令（#include、#define）进行处理，删除多余的空白字符和注释，生成一份新的代码
    - 编译：对代码进行语法、语义分析和错误判断，生成汇编代码文件
    - 汇编：将汇编代码转换成二进制文件
    - 链接：将多个二进制文件转换成一个可执行文件
16. C++中各种变量存储位置
    - 全局数据区：全局变量，静态变量、常量
    - 代码区：成员函数代码、非成员函数代码
    - 栈区：局部变量，函数参数，返回数据
    - 堆区：由程序员自己决定变量生存期的区间
17. C++中的哈希Map
    - 使用Map时，根据key返回value值时需要判断，key是否在Map中
    - C++的Map不会返回NULL
    - 如果key不在Map中，会产生值对Map[key]=默认值
18. C++代码中常见问题
    - 野指针，数组越界，内存泄漏，浮点数不能判断是否相等
19. 数组和指针的区别
    - 数组：用于存储多个相同类型数据的集合，数组名是首元素地址
    - 指针：指针相当于一个变量，但他存放的是其他变量在内存空间中的地址，指针名指向内存的首地址
20. 函数指针
    - 函数指针是指向函数的指针变量
    - 每个函数都有一个入口地址，函数指针指向函数的入口地址
21. 静态变量什么时候初始化

- C语言中：静态变量在所有代码执行之前初始化
- C++：静态对象在对象首次使用前才进行构造

22. 野指针
    - 野指针：指针指向的位置是不可知的，随机的，不正确的，没有明确限制的
    - 产生原因
        - 释放内存后，指针没有及时置空，依然指向该内存
    - 避免方法
        - 申请内存后判断是否为空
        - 内存释放后，指针置为空
23. 内联函数和宏定义函数的区别
    - 宏定义函数
        - 不是函数，但使用起来像函数
        - 宏定义在预编译期将所有的宏名用宏体替换
        - 宏定义没有类型检查，无论对还是错都是直接替换
    - 内联函数
        - 本质上是函数，内联函数不包含复杂的控制语句（while，switch等）
        - 内联函数在编译阶段进行代码插入，编译器在调用内联函数的地方直接把内联函数的内容展开
        - 内联函数在编译时进行安全检查（返回值类型，参数列表等）
        - 若内联函数本身的函数体过大，编译器会自动将其转换为普通函数
        - 内联函数不能调用自身
        - 省去函数调用的开销，提高效率
24. 内联函数和函数的区别
    - 普通函数
        - 普通函数在被调用的时候首先要找到函数的入口地址
            - 执行函数体，执行完成之后再回到函数调用的地方继续执行
    - 内联函数
        - 内联函数比普通函数多了关键字inline
        - 内联函数要求代码简单，不包含复杂的控制语句
        - 内联函数不需要寻址
            - 当执行内联函数时，将内联函数展开
            - 如果程序中有N次调用内联函数，则会有N次展开函数代码
25. 运算符i++和++i的区别
    - i++：先赋值，后加1
    - ++i：先加1，后赋值
26. 函数指针和指针函数的区别
    - 函数指针：本质是指针，指向一个函数
    - 指针函数：本质是函数，返回值是指针
27. 使用指针时的注意事项
    - 定义指针时，先初始化为NULL
    - 使用free或者delete释放内存后，应该指针置空，避免野指针
28. C++的传值方式
    - 值传递：函数体内形参值的变化，不会影响函数体外实参的值
    - 引用传递：形参在函数体内值的变化，会引起实参值的变化
    - 指针传递：在指针指向不变的情况下，形参的变化，会引起实参的变化
29. C++中堆和栈的区别
    - 空间分配不同
        - 堆一般由程序员分配和释放（由new分配的内存块）
        - 栈由操作系统分配和释放，存储函数参数值，局部变量等
    - 缓存方式不同
        - 栈使用一级缓存，调用完毕立即释放

    - 堆使用二级缓存，速度慢一些
30. 内存泄露
    - 申请的内存空间，使用完毕后没有释放掉
    - 避免方法
        - 申请一块内存，使用完毕后，利用相应的函数释放
31. 程序的组成
    - 代码段、数据段、BSS段、堆、栈、共享区
    - 数据段：存放已初始化的全局变量和静态变量的内存区域
    - BSS段：存放未初始化的全局变量和静态变量的一块区域
    - 代码段：存放程序执行代码的一块内存区域
    - 共享区：位于堆区和栈区之间
32. 面向过程和面向对象的区别
    - 面向过程：根据业务逻辑，从上到下编写代码
    - 面向对象：将数据和函数进行绑定和封装，减少了重复代码的编写
33. 继承方式
    - public继承
    - protected继承
    - private继承
34. 继承中，派生类对不同关键字修饰的基类方法的访问权限
    - 类中成员的类型：public成员、protected成员、private成员
        - public成员：父类和子类可以访问
        - protected成员：父类和子类可以访问
        - private成员：父类可以访问
        - 类可以直接访问自己的所有成员
    - public继承
        - 子类的实例化对象只能访问父类的public成员，不可以访问父类的protected和private成员
    - protected继承
        - 子类的实例化对象不可以访问父类的public，protected，private成员
    - private继承
        - 子类的实例化对象不可以访问父类的public，protected，private成员
35. C++重载和重写的区别
    - 重写
        - 派生类中重新定义的函数，函数名，参数类型，参数个数，返回值类型，都与基类相同，只有函数体不同
        - 基类中被重写的函数，必须由virtual关键字修饰
    - 重载
        - 函数名相同、参数类型，参数个数、顺序不同，根据参数调用相应的同名函数
        - 重载不关心函数返回值类型
36. C语言如何实现C++的函数重载
    - C语言不支持同名函数，在编译时，不需要参数类型和返回值类型
    - C++支持重载，在编译时，会添加参数类型和返回值类型作为函数编译后的名称
    - C实现重载
        - 可以使用函数指针实现
37. C++构造函数的类型及作用
    - 默认构造函数：定义类的对象时，完成对象初始化
    - 初始化构造函数：定义类的对象时，完成对象初始化
    - 移动构造函数：将其他类型变量，隐式转化为本类的对象
    - 拷贝构造函数：默认实现值拷贝
38. 一个类默认会生成那些函数
    - 构造函数
    - 析构函数

    - 赋值运算符
39. C++类对象的初始化顺序
    - 构造函数的作用就是对对象进行初始化
    - 构造函数=（如果存在继承先调用基类的构造函数）+（然后对成员变量初始化）+（自身构造函数函数体）
    - 父类构造函数 -> 成员类对象构造函数 -> 自身构造函数
    - 构造函数的调用顺序是类派生列表的顺序
    - 析构顺序和派生顺序相反
40. C++中变量的初始化顺序
    - 基类的静态变量或全局变量
    - 派生类的静态变量或全局变量
    - 基类的成员变量
    - 派生类的成员变量
41. 向上转换和向下转换
    - 向上转换：子类转换为父类
    - 向下转换：父类转换为子类，可以使用强制转换
42. 什么是虚继承
    - 虚继承是解决C++多重继承的一种手段
    - 多重继承：子类从不同途径，多次重复继承同一基类，基类在子类中存在多份拷贝
43. 纯虚函数可以实例化吗
    - 纯虚函数不能实例化，但可以在其子类中实例化
    - 纯虚函数是在基类中声明的虚函数，它要求任何子类都要定义自己的实现方法，以实现多态性
44. 构造函数中能否调用虚方法
    - 在语法上上可以调用，但构造函数调用虚函数没有意义
    - 子类对象在构造期间，进入父类的构造函数时，对象类型转换为父类类型，同样，进入父类的析构函数时，对象也是父类类型
    - 因此，调用虚函数时，始终只是调用父类的虚函数，不能达到多态的效果
45. 如何理解抽象类
    - 基类中包含纯虚函数的类，称为抽象类
    - 纯虚函数的实现：virtual关键字+函数原形后加"=0"
46. 虚析构函数
    - 虚析构函数：将基类的析构函数，声明为virtual
    - 作用：防止内存泄露
47. 虚基类
    - 在基类的前面加上virtual关键字，这时被继承的类称为虚基类
    - 虚基类可以被实例化
48. 哪些函数不能被声明为虚函数
    - 普通函数（非成员函数）
        - 普通函数只能被重载，不能被重写，声明为虚函数没有意义
    - 构造函数
        - 虚函数是通过父类的指针在调用相应的成员函数时，根据变量类型来调用子类相应的成员函数
        - 构造函数是在创建对象时调用的，不能通过父类的指针或引用来调用
        - 因此，构造函数不能是虚函数
49. C++中类模板和模板类的区别
    - 类模板是模板的定义，不是实实在在的类
    - 模板类是类模板的实例化
50. STL
    - STL：标准模板库（一些常用数据结构和算法模板的集合）
    - 分为3大类：
        - 容器、算法和迭代器
        - 容器
            - 一种数据结构

        - 如list、vector、deque，以模板类的方法提供
    - 算法
        - 用来操作容器中的数据的模板函数
    - 迭代器
        - 提供了访问容器中对象的方法

51. STL常见容器
    - Vector：动态数组
        - 特点
            - 内存中连续存储结构
            - 支持高效的随机访问+尾端插入/删除元素
            - 其他位置插入/删除，效率较低下
        - vector与数组的区别
            - 支持动态的内存空间扩展（push_back()、pop_back()）
            - 定义时不需要指定vector的大小
            - 数组的扩展需要程序员自己定义
    - List：双向链表
        - 内存中非连续存储结构
        - 每个元素拥有前指针和后指针（pre，post）
        - 可以在两端进行push和pop
        - 方便删除和插入、查找效率低下
        - 内存占用量较高
    - Deque：双向队列
        - 连续存储结构
        - 功能上合并了vector和list
        - 利用两级数组结构（第一级：实际容器、第二级：容器的首地址）
        - 高效的插入、删除、查询
        - 支持双端pop和push
    - Set：集合
        - 用于存储同类型的唯一元素
        - 元素插入后不能被修改
        - set支持插入与删除
    - Map：类似于字典
        - Map和set都属于关联容器
        - Map内部自建红黑二叉树、对数据进行自动排序，Map中的数据是有序的
        - 关联容器与序列容器
            - 序列容器：容器中的元素是按照位置存储顺序进行保存和访问的
            - 关联容器：支持高效的关键字查找和访问
52. STL的空间适配器
    - 空间适配器是用于实现内存空间分配的工具
    - 与容器联系紧密，每一种容器的空间分配，都是通过空间适配器实现的
53. STL容器的查找时间复杂度
    - Vector：动态数组
        - 插入：O(N)
        - 查看：O(1)
        - 删除：O(N)
    - Deque：双向队列
        - 插入：O(N)
        - 查看：O(1)
        - 删除：O(N)
    - List：双向链表

        - 插入：O(1)
        - 查看：O(N)
        - 删除：O(1)
    - Map、Set
        - 插入：O(logN)
        - 查看：O(logN)
        - 删除：O(logN)
54. C++中智能指针和指针的区别
    - 智能指针
        - 程序中用new申请一块内存，使用结束后，需要delete将其释放
        - C++采用智能指针auto_ptr自动实现这个过程
    - 指针
        - C语言中，指针专门用于存放地址
    - 区别
        - 智能指针是对普通指针的一种封装，他负责自动释放所指对象
55. C++中右值引用
    - 左值
        - 当对象被用作左值的时候，用的是对象在内存中的位置
    - 右值
        - 当一个对象被用作右值的时候，用的是对象的值
        - 右值要么是字面常量，要么是在表达式求值过程中创建的临时对象
    - 右值引用
        - 只能绑定到右值的引用，通过&&获得，只能绑定到即将销毁的对象上
    - 左值引用
        - 引用是变量的别名，指向左值
        - 左值引用的对象是持久的，不会马上销毁
56. 指针和引用的区别
    - 引用是定义一个变量的别名，指针存储一个变量的地址
    - 引用在定义时必须要初始化，不能为NULL，指针可以为NULL
    - 引用 加1 相当于 引用对应的实体值 加1，指针 加1 相当于 指针向后偏移 一个类型单位
57. C++的Main()函数
    - C++在main()函数之前和之后进行的操作
        - Main()之前
            - 进行初始化栈，堆
            - 打开标准输入，输出，错误流，
            - 把参数压入栈
            - 全局变量、对象和静态变量的空间分配和赋初始值
        - Main()之后
            - 销毁堆内存
            - 关闭标准输入，输出，错误流
    - Main()函数参数
        - C++允许主函数main()有或者没有参数列表
        - 主函数main()中可以使用一个或更多的参数
        - 常见：int main(int argc, char *argv[])
            - 参数argc
                - 指明有多少个参数将被传递给主函数main()
                - main()函数本身是索引为0的第一个参数，argc总是至少为1
                - argc的总数是argv[]中元素的数目
            - 参数*argv[]
                - 真正的参数以字符串数组（argv[]）的形式来传递

- 每一个字符串均有自己含义

58. 静态多态与动态多态
    - 动态多态
        - 也称为运行期多态
        - 主要通过虚函数实现
    - 静态多态
        - 也称为编译期多态
        - 通过函数重载实现
59. 静态库与动态库
    - 一个程序编译成可执行文件的过程
        - 预编译-编译-汇编-链接-可执行文件
    - 静态库
        - 概念：在汇编结束后生成目标文件.o文件，链接阶段会将目标文件.o文件与需要的库一起链接到可执行文件当中
        - 优点
            - 静态库对函数库的链接是在编译期完成的，执行期间代码装载速度快
            - 如果多个进程需要引用到【静态库】，在内存中就会存在多份拷贝
        - 缺点
            - 使可执行文件变大，浪费空间和资源
    - 动态库
        - 概念：动态库在编译时并不会链接到目标代码中，而是在程序运行时才会载入
        - 特点
            - 不同的应用程序如果依赖同一个动态库，那么动态库在内存中只存在一份拷贝
            - 避免了静态库浪费空间和资源的问题
60. C++11新特性
    - 智能指针、lambda表达式、空指针、右值引用等
    - nullptr空指针
        - 解决C++中NULL的二义性问题，因为NULL实际上代表的是0
        - 以往我们使用NULL表示空指针，它实际上是为0的int值
    - lambda表达式
        - 定义匿名函数，简化编程工作
        - 

61. sizeof运算符
    - 作用：用于判断变量或者数据类型所占用的内存空间的字节数
    - 变量
        - sizeof（变量）
        - 求的是变量所占用的内存空间的字节数
    - 数据类型
        - Sizeof（int）
        - 求的是这个object类型的空间大小
    - 数组
        - 求的是定义时声明的数组大小空间，不管你这个空间用完没有
            - char a[100]='1281'
            - Sizeof(a) = 100
    - 指针
        - 求的是指针里面的内容到底占几个字节，与计算机的位数有关
        - 32位=4字节
        - 64位=8字节
            - char *p='1281'
            - 32位：Sizeof(p)=4

        - 64位：Sizeof(p)=8
        - Sizeof(*p)=1 等价于 sizeof(p[0])
62. Auto变量
    - auto变量具有自动匹配类型功能
    - 使用auto变量时不需要考虑变量是否被释放，auto变量离开作用域后就会被程序自动释放，一般不会发生内存溢出
63. Volatile关键字作用
    - 作用
        - 保证变量写操作的可见性
        - 防止编译器优化
        - 防止在共享的区间发生读取错误
    - 编译器优化
        - 如果在两次读取变量之间不改变变量的值，编译器会发生优化，将RAM中的值赋值到寄存器中
        - 访问寄存器的效率要高于RAM
    - Volatile操作
        - 被Volatile修饰的变量被修改后，会将修改后的变量直接写入内存中，并且将其他线程中该变量的缓存置为无效
        - 从而让其它线程直接从主存中获取数据
64. 宏定义的作用
    - 宏的作用：主要在于字符串替换
    - 宏定义常量
        - # define MAXSIZE 200
    - 宏定义函数
        - # define MAX(a,b) ((a)<(b)?(b):(a))
65. 数据类型占用空间
    -

| 类型 | 16位平台 | 32位平台 | 64位平台 |
| --- | --- | --- | --- |
| char | 1个字节 | 1个字节 | 1个字节 |
| short | 2个字节 | 2个字节 | 2个字节 |
| int | 2个字节 | 4个字节 | 4个字节 |
| unsigned int | 2个字节 | 4个字节 | 4个字节 |
| float | 4个字节 | 4个字节 | 4个字节 |
| double | 8个字节 | 8个字节 | 8个字节 |
| long | 4个字节 | 4个字节 | 8个字节 |
| long long | 8个字节 | 8个字节 | 8个字节 |
| unsigned long | 4个字节 | 4个字节 | 8个字节 |
| 指针 | 2个字节 | 4个字节 | 8个字节 |
| **最大存储空间** | 2^16 | 2^32 | 2^64 |