1. **列表，元组，集合和字典:**

**列表：是 Python 中使用 最频繁 的数据类型，在其他语言中通常叫做 数组**

**hzit\_course = ['Python', 'docker', 'linux', 'mysql', 'docker', 'acp', 'rhce']**

**print(hzit\_course)**

**元组： 如何查看变量的类型 type 列表可变 元组不可变 括号表示**

**info\_tuple = ("hzit\_0405", 18, 1.75, "python")**

**集合：是一个无序和无索引的集合。用{ }花括号包裹。没有重复的成员。**

**要定义一个空集合不能用空的{ }，要用set()函数。**

**a= {"apple", "banana", "cherry"}**

**a=set(("apple", "banana", "cherry"))**

**字典：字典用 {key: value, key: value} 定义 可修改**

**person = {**

**"name" : "hzit\_0405",**

**"age" : 18,**

**"gender": True,**

**"height": 1.75**

**}**

**二、在Python中完成异常处理**

**在程序执行时，可能会遇到 不同类型的异常，并且需要 针对不同类型的异常，做出不同的响应，这个时候，就需要捕获错误类型了**

**try:**

**import demo\_syntax # 导入模块**

**except IndentationError:**

**print('xxxx缩进错误')**

**except SyntaxError:**

**print('语法错误')**

**except AttributeError:**

**print('属性错误')**

**except ZeroDivisionError:**

**print('0不能做分母')**

**except Exception as e: # 将异常信息给变量 e**

**print(type(e))**

1. **Python模块用来衡量应用程序代码的性能**

**import time**

**start = time.time() #开始时间**

**print("Hello World") #输出 “Hello World”**

**time.sleep(2) #字面意思 sleep停留2秒之后运行**

**end = time.time() #结束时间**

**print(end - start) #结束时间-开始时间-停留时间=运行时间**

1. **在Python中实现线程**

**进程就是一个应用程序在处理机上的一次执行过程，它是一个动态的概念，而线程是进程中的一部分，进程包含多个线程在运行。**

**python主要是通过thread和threading这两个模块来实现多线程支持。**

**单线程：（只能做一件事）**

**Import time**

**def sayHello():**

**print(“6666”)**

**time.sleep(1)**

**if \_\_name\_\_==”main”:**

**for p in range(11):**

**sayHello()**

**多线程：（唱歌的同时可以跳舞）**

**import threading *# 线程模块***

**from time import sleep**

**def sing():**

**for s in range(1,50):**

**print("正在唱歌...... {}".format(s))**

**sleep(1)**

**def dance():**

**for d in range(1,50):**

**print("正在跳舞...... {}".format(d))**

**sleep(1)**

**def main():**

***# target=函数名 不是调用函数 千万不要加小括号***

**t1=threading.Thread(target=sing) *#***

**t2=threading.Thread(target=dance) *#***

***# 运行 开始 start才开始运行***

**t1.start()**

**t2.start()**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

1. **在Python应用程序的主流程中启动子流程**

**import time**

**from multiprocessing import Process**

**import os**

**def run(name):**

**while True:**

**time.sleep(2)**

**print("子进程启动ID号:{}，run：{}".format((os.getpid(), name)) # os.getpid()进程ID**

**if \_\_name\_\_ == "\_\_main\_\_":**

**print("主进程启动：{}".format(os.getpid())**

**# 创建子进程**

**p = Process(target=run, args=("Ail",)) # target进程执行的任务, args传参数（元祖）**

**p.start() # 启动进程**

**while True:**

**print("死循环")**

**time.sleep(1)**

**# 输出结果**

**主进程启动：36684**

**死循环**

**死循环**

**死循环**

**子进程启动ID号:40228，run：Ail**

**死循环**

**六、调用显式方法来销毁Python中分配的内存**

**import garbage collector**

**a=range(10000\*10000) # 内存会占的很大，并不会马上释放。**

**del a**

**gc.collect() #内存瞬间释放**

**七、列出Python中可用的一些重要模块用法及案例**

**Pymysql模块：实现python连接数据库并且操作数据库。**

**import pymysql #导入模块**

**conn=pymysql.connect(host:’127.0.0.1’,**

**user=:’root’,**

**password=’123’,**

**port=’3306’,**

**charset:’utf8’)**

**cur=conn.cursor() #使用cursor创建一个对象**

**cur.execute(‘create database 库名’) #创建库名**

**conn.commit()#提交请求**

**cur.close()**

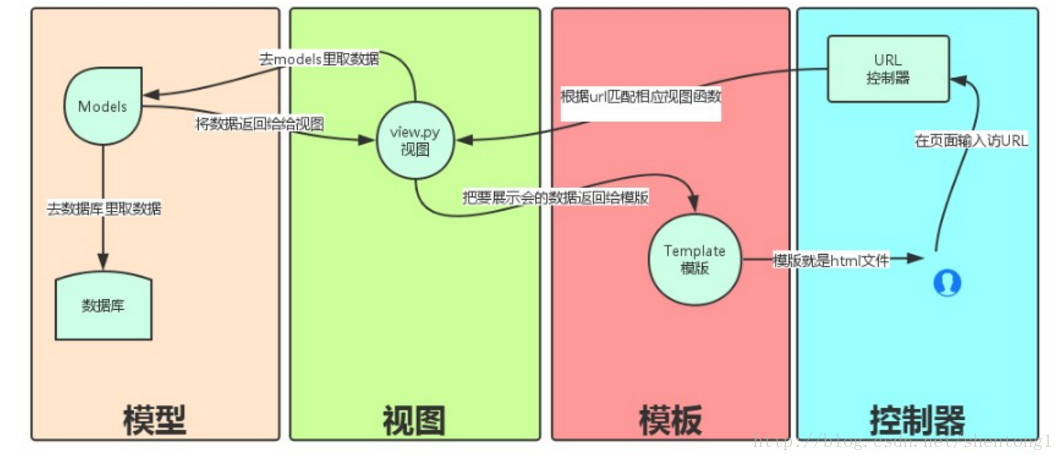
**conn.close()**

**time模块用来输出时间**

**flask模块用来创建web应用框架（轻量）**

**requeste和parsel模块 用来访问网络的模块和请求后的字符串格式解析成re,xpath,css进行内容的匹配**

**Django模块（框架）大而全，框架本身集成了ORM、模型绑定、模板引擎、缓存、Session等诸多功能**

****

**Pandas模块 对于数据的导入和处理的模块。**

**假如需要导入csv文件数据 就需要：**

**import pandas as pd**

**data\_file=pd.read\_csv(“csv文件的路径”)**

**Numpy模块 支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库.**

**# 导入模块**

**import numpy as np**

**# 创建一维array对象**

**a1 = np.array([1, 3, 2])**

**# 创建一个2 x 3 矩阵 , 输入参数实际为列表或元组，列表或元组内部元素为单个数值**

**# 创建二维array对象**

**a1 = np.array([1, 3, 2], [4, 3, 6])**

**# 创建一个2 x 3 矩阵 , 输入参数实际为列表或元组，列表或元组内部元素又是列表或元组数值**

**八、排序算法用法及案例**

**冒泡排序：一种常见的排序算法，以列表[10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]为例，比较相邻的两个元素。如果第一个比第二个大则交换他们的位置(升序排列，降序则反过来)**

**冒泡排序后是 [10,17,7,30,24,27,45,15,5,36,21,50]，在下一轮“冒泡”中，不需要再将50进行比较，需要比较的元素个数减1，然后直到从小到大顺序排列。**

**def bubble\_sort(array):**

**for i in range(1, len(array)):**

**for j in range(0, len(array)-i):**

**if array[j] > array[j+1]:**

**array[j], array[j+1] = array[j+1], array[j]**

**return array**

**if \_\_name\_\_ == '\_\_main\_\_':**

**array = [10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]**

**print(bubble\_sort(array))**

**#输出结果为[5, 7, 10, 15, 17, 21, 24, 27, 30, 36, 45, 50]**

**选择排序：简单直观的排序算法，选择排序首先从待排序列表中找到最小(大)的元素，存放到元素列表的起始位置(与起始位置进行交换)，作为已排序序列，第一轮排序完成。然后，继续从未排序序列中找到最小(大)的元素，存放到已排序序列的末尾。直到所有元素都存放到了已排序序列中，列表排序完成。**

**选择排序每次都是去找最小(大)的元素，隐含了一种挑选的过程，所以被称为选择排序。**

**以列表[10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]为例，找到最小或者最大的元素和启示位置交互位置，最小的是5，启示为10。最小的是已经完成的，其他的是未完成的。然后以此类推。**

**def selection\_sort(array):**

**for i in range(len(array)-1):**

**min\_index = i**

**for j in range(i+1, len(array)):**

**if array[j] < array[min\_index]:**

**min\_index = j**

**if min\_index != i:**

**array[i], array[min\_index] = array[min\_index], array[i]**

**return array**

**if \_\_name\_\_ == '\_\_main\_\_':**

**array = [10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]**

**print(selection\_sort(array))**

**#输出结果为[5, 7, 10, 15, 17, 21, 24, 27, 30, 36, 45, 50]**

**插入排序：插入排序的原理类似于玩扑克牌时，手动抓牌和排序，每抓一张新牌都按顺序插入到已有的牌中。**

**将待排序列表的第一个元素当做已排序序列，第二个元素到最后一个元素当成未排序序列。**

**取未排序序列中的第一个数据，插入到已排序序列中顺序正确的位置。将未排序的第一个数据与相邻的前一个数据(已排序序列的最后一个数)据进行比较，如果顺序错误则交换位置，交换位置后继续与相邻的前一个数据进行比较，直到不需要交换则插入完成。每次插入数据后，已排序序列都是排好序的。**

**重复上一步，继续插入下一个数据。每进行一次插入，已排序序列的长度加1，未排序序列的长度减1，直到列表中的所有数据都插入到已排序序列了，则列表排序完成。**

**以列表 [10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]为例：**

**def insertion\_sort(array):**

**for i in range(len(array)):**

**cur\_index = i**

**while array[cur\_index-1] > array[cur\_index] and cur\_index-1 >= 0:**

**array[cur\_index], array[cur\_index-1] = array[cur\_index-1], array[cur\_index]**

**cur\_index -= 1**

**return array**

**if \_\_name\_\_ == '\_\_main\_\_':**

**array = [10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21]**

**print(insertion\_sort(array))**

**#输出结果为[5, 7, 10, 15, 17, 21, 24, 27, 30, 36, 45, 50]**

**希尔排序：插入排序的进阶版，以[5,4,3,2,1]为例子，因为是要移动到较远的位置，即增大移动间距，那么可以从中抽出几个子序列（间隔为2，间隔自己决定），[5,3,1]和[4,2]，当然下标不变，还是原来的。那么对这两个子序列分别进行插入排序，可以看出来，5只需要两次移动就到了最终的位置，---->[3,5,1]---->[3,1,5]---->[1,3,5]，同理4也是。**

**def shell\_sort(alist):**

**gap = len(alist) // 2**

**while gap > 0:**

**for i in range(0, len(alist) - gap, gap):**

**temp = alist[i+gap]**

**for j in range(i, -1, -gap):**

**if alist[j] > temp:**

**alist[j+gap] = alist[j]**

**else:**

**alist[j+gap] = temp**

**break**

**else:**

**alist[j] = temp**

**print(alist)**

**gap = gap // 2**

**alist = [2,7,5,84,10,1,3,9,8,6]**

**shell\_sort(alist)**

**print(alist)**

**#输出结果**

**[1, 7, 5, 84, 10, 2, 3, 9, 8, 6]**

**[1, 7, 3, 84, 5, 2, 8, 9, 10, 6]**

**[1, 2, 3, 5, 6, 7, 8, 9, 10, 84]**

**[1, 2, 3, 5, 6, 7, 8, 9, 10, 84]**

**快速排序：在partion函数里面，我们初始化i=-1，p=0,r=7,j从0至7，这里有一个循环过程，就是拿序列里面的八个元素与第八个元素（基准元素）进行比较，如果元素小于基准元素，i+=1，置换list[i]和list[j]的顺序，当j=0时，就是list[0]与list[0]置换，这里不变，但是第一个子空间已经增大了一个元素，list[0]<=list[r]；当j=1时，list[1]>=list[r]，此时不交换任何位置，但第二个子空间会随着j的增大而增加一个元素，而此时的第三个子空间已经减少了两个元素了。依次的当j=6时，list[6]>=list[r]，不交换任何位置，而第三个子空间已经没有任何元素了。当j=7时，list[7]>=list[r]，不交换任何位置。此时循环结束，调换list[r]与list[i+1]，使得list[r]可以获取正确的位置，并返回i+=1为划分边界q。**

**def quicksort(list,p,r):**

**if p<r:**

**q=partion(list,p,r)**

**quicksort(list,p,q-1)**

**quicksort(list,q+1,r)**

**def partion(list,p,r):**

**i=p-1**

**for j in range(p,r):**

**if list[j]<list[r]:**

**i+=1**

**list[i],list[j]=list[j],list[i]**

**print(list)**

**list[i+1],list[r]=list[r],list[i+1]**

**return i+1**

**list1=[2,8,7,1,3,5,6,4]**

**quicksort(list1,0,7)**

**print (list1)**

**#输出结果**

**[2, 8, 7, 1, 3, 5, 6, 4]**

**[2, 8, 7, 1, 3, 5, 6, 4]**

**[2, 8, 7, 1, 3, 5, 6, 4]**

**[2, 1, 7, 8, 3, 5, 6, 4]**

**[2, 1, 3, 8, 7, 5, 6, 4]**

**[2, 1, 3, 8, 7, 5, 6, 4]**

**[2, 1, 3, 8, 7, 5, 6, 4]**

**[2, 1, 3, 4, 7, 5, 6, 8]**

**[2, 1, 3, 4, 7, 5, 6, 8]**

**[2, 1, 3, 4, 7, 5, 6, 8]**

**[1, 2, 3, 4, 7, 5, 6, 8]**

**[1, 2, 3, 4, 7, 5, 6, 8]**

**[1, 2, 3, 4, 7, 5, 6, 8]**

**[1, 2, 3, 4, 7, 5, 6, 8]**

**[1, 2, 3, 4, 5, 7, 6, 8]**

**[1, 2, 3, 4, 5, 6, 7, 8]**

**二分法查找：二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。**

**原理：将表中间位置（mid）的数字与待查数字（data）做比较，**

* **如果相等：返回true，结束。**
* **如果不相等：则使用中间位置的记录将表分为前后两个子表。**
  1. **若data>mid 则进一步查找后一个表。**
  2. **若data<mid 则进一步查找前一个表。**

**直到找到与data相等的值。如果没有则返回false。**

**def binary\_search(alist, data):**

**n = len(alist)**

**first = 0**

**last = n - 1**

**while first <= last:**

**mid = (last + first) // 2**

**if alist[mid] > data:**

**last = mid - 1**

**elif alist[mid] < data:**

**first = mid + 1**

**else:**

**return True**

**return False**

**lis = [2, 4, 5, 12, 14, 23]**

**if binary\_search(lis, 16):**

**print('ok')**

**#什么都不返回 因为数字16不在列表内**

**九、单元测试工具Junit**

**首先junit也是最常见的Python单元测试框架，他是基于Java的JUnit框架。**

**用来对最小可测试单元（如一个函数、一个类或者一个模块）进行检查和验证。**

**尤其是在对代码进行修改或重构后，可以快速准确的定位出现问题的问题。**

**import unittest**

**class MyTests(unittest.TestCase):**

**def test\_print(self):**

**print("test1")**

**def test\_print2(self):**

**print("test2")**

**def atest\_print2(self):**

**print("test3")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**unittest.main()**

**#运行结果：**

**..**

**test1**

**----------------------------------------------------------------------**

**test2**

**Ran 2 tests in 0.000s**

**十、单元测试工具TestNG**

**TestNG（Next Generation）单元测试框架比JUnit单元测试框架更强大，它提供了更多的扩展功能，可以通过注解、分组、序列和参数化组织和执行自动化测试脚本，因此它适合运行更复杂的自动化测试用例。**

**十一、Socket编程用法及案例**

**平常我们会接触到两种架构，分别是：b/s架构与c/s架构**

**b/s架构(brownser浏览器/server服务器)：建立在广域网的基础上，通过浏览器与服务器进行数据交互，如：购物等网页数据，我们只需要通过浏览器就可以访问了。**

**c/s架构(client客户端/server服务端)：建立在局域网的基础上，通过网络向服务器发送数据请求，然后服务端回应客户端的请求数据。如：平常所使用的软件，基本都是客户端的形式**

**Socket也可以称为：套接字，应用程序通常使用套接字通过网络，实现与另一台计算机进行通讯。**

**socket套接字：集合了OSI模型七层，传输层以下的所有功能**

**先执行服务端**

**import socket**

**# 创建套接字对象，AF\_INET基于TCP/UDP通信，SOCK\_STREAM以数据流的形式传输数据，这里就可以确定是TCP了**

**server = socket.socket(family=socket.AF\_INET,type=socket.SOCK\_STREAM)**

**# 绑定ip地址和端口，127.0.0.1代表回环地址，只能当前计算机访问**

**server.bind(('127.0.0.1',8080))**

**# 建立半链接池，允许存放5个请求**

**server.listen(5)**

**# 等待建立连接请求，会返回两个值，一个是连接会话，一个是连接的客户端IP与端口**

**conn,ip\_addr = server.accept()**

**while True:**

**# 接收客户端传递的数据，不能超过1024字节**

**res = conn.recv(1024)**

**# 将客户端的数据接收到以后，转换成大写再编码后发送给客户端**

**conn.send(res.decode('utf-8').upper().encode('utf-8'))**

**# 注意：close不能放在这里面，因为在这里面的话，只能一次后就关闭了**

**# 关闭与客户端的连接**

**conn.close()**

**# 关闭套接字**

**server.close()**

**在执行客户端**

**import socket**

**# 创建套接字对象，AF\_INET基于TCP/UDP通信，SOCK\_STREAM以数据流的形式传输数据，这里就可以确定是TCP了**

**client = socket.socket(family=socket.AF\_INET,type=socket.SOCK\_STREAM)**

**# 连接服务端**

**client.connect(('127.0.0.1',8080))**

**while True:**

**inp = input('>>>：').strip()**

**# 向服务端发送数据，需要转换成Bytes类型发送**

**client.send(inp.encode('utf-8'))**

**# 接收服务端回应给客户端的数据，不能超过1024字节**

**res = client.recv(1024)**

**print(res.decode('utf-8'))**

**# 套接字关闭**

**client.close()**

**十二、浅拷贝和深拷贝用法及案例**

**浅拷贝：为了解决函数传递后被修改的问题，就需要拷贝一份副本，将副本传递给函数使用，就算是副本被修改，也不会影响原始数据**

**import copy**

**a = 10**

**b = copy.copy(a)**

**id(a)**

**1730306496**

**id(b)**

**1730306496**

**深拷贝：区别于浅拷贝只拷贝顶层引用，深拷贝会逐层进行拷贝，直到拷贝的所有引用都是不可变引用为止。copy.deepcopy(你要复制的对象) : 这个是深拷贝，他除了和浅拷贝一样，会新生成一份对象的引用，另外对于内部的元素，都会新生成引用，以独立分开。**

**import copy**

**a = 10**

**b = copy. deepcopy (a)**

**十三、\* args和\* kwargs用法及案例**

**\*args：表示的是实参中按照位置传值，将多出来的值都给args，以元祖的方式呈现**

**def foo(x,\*args):**

**print (x,args);**

**foo(1,2,3,4,5,6,7)**

**#运行结果为 (1, (2, 3, 4, 5, 6, 7))**

**\*Kwargs：表示的就是形参中按照关键字传值把多余的传值以字典的方式呈现**

**def foo(a,\*\*kwargs):**

**print (a,kwargs)**

**// 1的值传给了a 将x=1,y=2,z=3以字典的方式传给了kwargs**

**foo(1,x=1,y=2,z=3)**

**#运行的结果为 (1, {'y': 2, 'x': 1, 'z': 3})**

**十四、迭代器用法及案例**

**for循环：用于遍历一个迭代对象的所有元素**

**sum=0**

**for I in range(1,101):**

**sum+=i**

**print(sum)**

**#运行结果为5050**

**可以使用isinstance()判断一个对象是否是Iterable对象：**

**>>>> from collections import Iterable**

**>>> isinstance([], Iterable)**

**True**

**>>> isinstance({}, Iterable)**

**True**

**>>> isinstance('abc', Iterable)**

**True**

**>>> isinstance((x for x in range(10)), Iterable)**

**True**

**>>> isinstance(range(10), Iterable)**

**True**

**>>> isinstance(100, Iterable)**

**False**

**十五、生成器用法及案例**

**但是我们在实现一个迭代器时，关于当前迭代到的状态需要我们自己记录，进而才能根据当前状态生成下一个数据。**为了达到记录当前状态，并配合next()函数进行迭代使用，**所以说**生成器(generator)其实是一类特殊的迭代器**。**

**L = [ x\*2 for x in range(5)]**

**print(L)**

**运行结果： [0, 2, 4, 6, 8]**

**def use(name):**

**print("%s开始捡玩具了。。。" % name)**

**while True:**

**res = yield**

**print("%s捡了%s个玩具" % (name,res))**

**def product(name\_mon):**

**mon = use(name\_mon)**

**mon.\_\_next\_\_()**

**for i in range(10):**

**print("儿子又扔了一个玩具")**

**i+=1**

**mon.send(i)**

**product("妈妈")**

**ls = [x\*2 for x in range(10)]**

**generator1 =(x\*2 for x in range(10)) #这是一个生成器generator**

**print(ls)**

**print(generator1) #注意，打印生成器，不会像列表一样打印他的值，而是地址。**

**''’**

**[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]**

**<generator object <genexpr> at 0x00000239FE00A620>**

**'''**

**十六、闭包用法及案例**

如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包（closure）。它只不过是个“内层”的函数，由一个名字（变量）来指代，而这个名字（变量）对于“外层”包含它的函数而言，是本地变量。

def make\_adder(addend):

def adder(augend):

return augend + addend

return adder

p = make\_adder(23)

q = make\_adder(44)

print(p(100))

print(q(100))

#运行结果为：

123

144

make\_adder是一个函数,包括一个参数addend,比较特殊的地方是这个函数里面又定义了一个新函数,这个新函数里面的一个变量正好是外部make\_adder的参数.也就是说,外部传递过来的addend参数已经和adder函数绑定到一起了,形成了一个新函数,我们可以把addend看做新函数的一个配置信息,配置信息不同,函数的功能就不一样了,也就是能得到定制之后的函数，再看看运行结果,我们发现,虽然p和q都是make\_adder生成的,但是因为配置参数不同,后面再执行相同参数的函数后得到了不同的结果.这就是闭包.

**十七、内置模块用法及案例**

**Re：**

# . 匹配任意1个字符（除了\n）

# [ ] 匹配[ ]中列举的字符

# \d 匹配数字,即0-9

# \D 匹配非数字,即不是数字

# \s 匹配空白,即 空格,tab键

# \S 匹配非空白

# \w 匹配非特殊字符,即a-z A-Z 0-9 \_ 汉字

# \W 匹配特殊字符,即非字母 非数字 非汉字 非下划线

# . 匹配任意1个字符（除了\n）

# 1. 正则表达式

# 2. 要匹配的字符串

# match\_obj返回匹配对象

import re

match\_obj = re.match("\d", "123456") #只匹配第一个数字相关的

if match\_obj:

# 获取匹配结果

result = match\_obj.group()

print(result)

else:

# 匹配失败match\_obj是一个None

print("匹配失败")

**Math：**与数学相关的函数，

import math #先导入math包

print(math.pi) #三角函数

------------------------

Print(math.floor(3.14))#向下取整

3.0

------------------------

Print(math.ceil(3.14))#向上取整

4.0

**Os：**

**Datetime：**

**Sys：**

**Json：**

json是 JavaScript Object Notation 的首字母缩写,翻译过来就是javascript对象表示法,这里说的json就是类似于javascript对象的字符串,它同时是一种数据格式,目前这种数据格式比较流行,逐渐替换掉了传统的xml数据格式

类似于javascript对象的字符串:

它同时是一种数据格式:

对象格式:

数组格式:

json数据 类似JavaScript对象 可以进行转换 对象.name 对象.age就能获取到对应的值

http://www.weather.com.cn/data/sk/101010100.html (编码非UTF-8) 天气预报json接口

{"weatherinfo":{"city":"北京","cityid":"101010100","temp":"27.9","WD":"南风","WS":"小于3级","SD":"28%","AP":"1002hPa","njd":"暂无实况","WSE":"<3","time":"17:55","sm":"2.1","isRadar":"1","Radar":"JC\_RADAR\_AZ9010\_JB"}}

总结:

json就是一个javascript对象表示法,json本质上是一个字符串 (字符串格式的字典)

json有两种格式: 1.对象格式, 2.数组格式

json格式: 必须使用双引号

{

"class": {

"name": "提高班",

"no": "1801",

"students": {

"student": [

{

"name": "宋海娇",

"sex": "女"

},

{

"name": "毛伟伟",

"sex": "男"

}

]

}

}

}

xml格式:

<!--注释1：申明是xml报文,指名版本和编码-->

<?xml version="1.0" encoding="UTF-8" ?>

<class>

<name>提高班</name>

<no>1801</no>

<students>

<student>

<name>宋海娇</name>

<sex>女</sex>

</student>

<student>

<name>毛伟伟</name>

<sex>男</sex>

</student>

</students>

</class>