# 面试经验

我叫卫宝，本科毕业于哈尔滨工程大学，专业是计算机科学与技术。2017年7月-2021年7月在中国航天科工二院706所担任科研助理岗位，我在学校有一年项目开发经验，主要进行的是后台开发和基于OpenStack的二次开发。

## Python基础

### Python 中的可变对象和不可变对象

不可变对象，该对象所指向的内存中的值不能被改变。当改变某个变量时候，由于其所指的值不能被改变，相当于把原来的值复制一份后再改变，这会开辟一个新的地址，变量再指向这个新的地址。

可变对象，该对象所指向的内存中的值可以被改变。变量（准确的说是引用）改变后，实际上是其所指的值直接发生改变，并没有发生复制行为，也没有开辟新的出地址，通俗点说就是原地改变。

Python 中，数值类型（int 和 float）、字符串 str、元组 tuple 都是不可变类型。而列表 list、字典 dict集合 set 是可变类型。

### is和==

is 判断的是 a 对象是否就是 b 对象，是通过 id 来判断的。

==判断的是 a 对象的值是否和 b 对象的值相等，是通过 value 来判断的。

### 赋值、浅拷贝和深拷贝的区别

1赋值：

在 Python 中，对象的赋值就是简单的对象引用，is，id()函数来查看两个变量是相同的，

2浅copy

浅拷贝会创建新对象，其内容非原对象本身的引用，而是原对象内第一层对象的引用。浅拷贝有三种形式:切片操作、工厂函数、copy 模块中的 copy 函数。

用is，id()函数二者不一样，浅拷贝之所以称之为浅拷贝，是它仅仅只拷贝了一层，如果变量中有一个嵌套的 list，如果我们修改了它，情况就不一样了。

3．深拷贝

深拷贝就是copy 模块中的 deepcopy() 函数，它拷贝了对象的所有元素，包括多层嵌套的元素。因此，它的时间和空间开销要高，深拷贝出来的对象根本就是一个全新的对象，不再与原来的对象有任何的关联。

4.对于非容器类型，如数字、字符，以及其他的“原子”类型，没有拷贝一说，产生的都是原对象的引用。如果元组变量值包含原子类型对象，即使采用了深拷贝，也只能得到浅拷贝。

（容器是用来存储和组织其他对象的对象。那python里面也一样，有列表(list)，元祖(tuple)，集合(set)，字典(dict)，队列(queue)等）容器通常配有迭代器（iter），迭代器是用来遍历容器的利器）

### 说明init 和new的区别

\_\_new\_\_是在实例创建之前被调用的，因为它的任务就是创建实例然后返回该实例对象，是个静态方法，参数cls。\_\_init\_\_是当实例对象创建完成后被调用的，然后设置对象属性的一些初始值，通常用在初始化一个类实例的时候。是一个实例方法。

当我们使用「类名()」创建对象的时候，Python 解释器会帮我们做两件事情：第一件是为对象在内存分配空间用new方法，第二件是为对象进行初始化，用init方法。new 方法在内部其实做了两件事情：第一件事是为「对象分配空间」，第二件事是「把对象的引用返回给 Python 解释器」。当 Python 的解释器拿到了对象的引用之后，就会把对象的引用传递给 init 的第一个参数 self，init 拿到对象的引用之后，就可以在方法的内部，针对对象来定义实例属性。当使用「类名()」创建对象的时候，无论执行多少次，在内存中永远只会创造出一个对象的实例。

### Python变量对象和引用

变量是到内存空间的一个指向，也就是拥有指向对象连接的空间；

对象是一块内存，表示它们所代表的值；

引用就是自动形成的从变量到对象的指向。

### Python可变对象和不可变对象

可变对象：list dict set

不可变对象：tuple string int float bool

### Python无题

**Python 是强类型的动态脚本语言强类型**：不允许不同类型相加。 动态：不使用显示数据类型声明，且确定一个变量的类型是在第一次给它赋值的时候。 脚本语言：一般也是解释型语言，运行代码只需要一个解释器，不需要编译。

内建函数封装了各种转换函数，可以使用目标类型关键字强制类型转换，int(‘str’，base=’n’)将特定进制的字符串转换为十进制，再用相应的进制转换函数将十进制转换为目标进制。

Python 中有日志，Python 自带 logging 模块，调用 logging.basicConfig()方法，配置需要的日志等级和相应的参数，Python 解释器会按照配置的参数生成相应的日志。

**dict 的 items() 方法与 iteritems() 方法的不同**

items方法将所有的字典以列表方式返回，其中项在返回时没有特殊的顺序

iteritems方法有相似的作用，但是返回一个迭代器对象

**os.path和sys.path的区别**

os.path是module，包含了各种处理长文件名(路径名)的函数。

sys.path是由目录名构成的列表，Python 从中查找扩展模块( Python 源模块, 编译模块,或者二进制扩展). 启动 Python 时,这个列表从根据内建规则,PYTHONPATH 环境变量的内容, 以及注册表( Windows 系统)等进行初始化

os.remove() 删除文件

os.rename() 重命名文件

os.chmod() 改变目录权限

os.mkdir/rmdir创建目录/删除目录

os.getcwd() 取得当前工作目录

os.path.join() 将分离的各部分组合成一个路径名

os.path.exists() 是否存在

**Python字符串中\字符**1、转义字符2、路径名中用来连接路径名3、编写太长代码手动软换行。

**read、readline 和 readlines 的区别？**read:读取整个文件。readline：读取下一行，使用生成器方法。readlines：读取整个文件到一个迭代器以供我们遍历。

**range 和 xrange 的区别？** range 返回的结果是一个列表，而 xrange 的结果是一个生成器，前者是直接开辟一块内存空间来保存列表，后者是边循环边使用，只有使用时才会开辟内存空间，所以当列表很长时，使用 xrange 性能要比 range 好。

**print 方法默认调用 sys.stdout.write 方法，即往控制台打印字符串。**

sys.argv 命令行参数 List，第一个元素是程序本身路径。

sys.path 返回模块的搜索路径，初始化时使用 PYTHONPATH 环境变量的值。

sys.modules.keys() 返回所有已经导入的模块列表。

在 Python 中，模块是搭建程序的一种方式。每一个 Python 代码文件都是一个模块，并可以引用 其他的模块，比如对象和属性。

assert断言——声明其布尔值必须为真判定，发生异常则为假。

\*args收集到的是元组 \*\*kwargs是字典

回调函数是把函数的指针(地址)作为参数传递给另一个函数，将整个函数当作一个对象，赋值给调用的函数。

shutil 模块有一个 copyfile 函数可以实现文件拷贝，python程序用encode和decode

monkey在运行期间动态修改一个类或模块

### 正则

贪婪匹配：正则表达式一般趋向于最大长度匹配，也就是所谓的贪婪匹配。

非贪婪匹配：就是匹配到结果就好，就少的匹配字符。

面向对象是相对于面向过程而言的。

面向过程语言是一种基于功能分析的、以算法为中心的程序设计方法

面向对象是一种基于结构分析的、以数据为中心的程序设计思想。在面向对象语言中有一个有很重要东西，叫做类。面向对象有三大特性：封装、继承、多态。

### With的用法

with语句的作用是通过某种方式简化异常处理，它是所谓的上下文管理器的一种

当你要成对执行两个相关的操作的时候，这样就很方便，以上便是经典例子，with语句会在嵌套的代码执行之后，自动关闭文件。

这种做法的还有另一个优势就是，无论嵌套的代码是以何种方式结束的，它都关闭文件。如果在嵌套的代码中发生异常，它能够在外部exception handler catch异常前关闭文件。如果嵌套代码有return/continue/break语句，它同样能够关闭文件。

### 解释型语言和编译型语言

计算机不能直接理解高级语言，只能直接理解机器语言，所以必须要把高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。解释性语言在运行程序的时候才会进行翻译。

编译型语言写的程序在执行之前，需要一个专门的编译过程，把程序编译成机器语言（可执行文件）。

### Python作用域和命名空间

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定。当 Python 遇到一个变量的话它会按照这的顺序进行搜索：

本地作用域(Local)—>当前作用域被嵌入的本地作用域(Enclosing locals)—>全局/模块作用域 (Global)—>内置作用域(Built-in)。LEGB原则

命名空间：比如公司 A 张三公司 B也叫张三，公司就表示一个独立的命名空间，locals，global，built-in

### 说一下字典和 json 的区别

字典是一种数据结构，json 是一种数据的表现形式，字典的 key 值只要是能 hash 的就行，json 的必须是字符串。字典中存入的数据不会自动排序，可以使用 sort 函数对字典进行排序。

**json序列化时，默认遇到中文会转换成unicode，如果想要保留中文怎么办？**

### Python标准库

os 操作系统、time 时间datetime日期、random math随机、pymysql 连接数据库、threading 线程、multiprocessing 进程、queue 队列hashlib 加密 socket 通信gc垃圾回收

### Python异常

在 except 中 return 后还会不会执行 finally 中的代码？怎么抛出自定义异常？会继续处理 finally 中的代码；用 raise 方法可以抛出自定义异常。

### Python字符串格式化

% 格式化字符串操作符，字典形式的字符串格式化方法，字符串格式化（format）(1) 使用位置参数(2) 使用关键字参数

### Python 函数调用的时候参数的传递方式是值传递还是引用传递

Python 的参数传递有：位置参数、默认参数、可变参数、关键字参数。函数的传值到底是值传递还是引用传递，要分情况：

**不可变参数用值传递**

像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都**不可能在原处改变不可变对象**

可变参数是引用传递的比如像列表，字典这样的对象是通过引用传递、和 C 语言里面的用指针传递数组很相似，可变对象能在函数内部改变。

### map 函数和 reduce 函数 filter函数 lambda函数

map()包含两个参数，第一个参数是一个函数，第二个是序列（列表 或元组）。其中，函数（即 map 的第一个参数位置的函数）可以接收一个或多个参数。

reduce()第一个参数是函数，第二个是序列（列表或元组）。但是，其函数必须接收两个参数。两个向量加成一个向量

map()是将传入的函数依次作用到序列的每个元素，每个元素都是独自被函数“作用”一次 。两个向量相加reduce()是将传人的函数作用在序列的第一个元素得到结果后，把这个结果继续与下一个元素作用（累积计算）。

lambda 函数是一个可以接收任意多个参数(包括可选参数)并且返回单个表达式值的函数。

1、lambda 函数比较轻便，即用即仍，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下

2、匿名函数，一般用来给 filter， map 这样的函数式编程服务

3、作为回调函数，传递给某些应用，比如消息处理

### 面向对象

是相对于面向过程而言的。

面向过程语言是一种基于功能分析的、以算法为中心的程序设计方法

面向对象是一种基于结构分析的、以数据为中心的程序设计思想。在面向对象语言中有一个有很重要东西，叫做类。面向对象有三大特性：封装、继承、多态。

### Python 中类方法、类实例方法、静态方法有何区别

类方法：是类对象的方法，在定义时需要在上方使用“@classmethod”进行装饰，形参为 cls，表示类对象，类对象和实例对象都可调用

类实例方法：是类实例化对象的方法，只有实例对象可以调用，形参为 self，指代对象本身

静态方法：是一个任意函数，在其上方使用“@staticmethod”进行装饰，可以用对象直接调用，静态方法实际上跟该类没有太大关系静态方法没有类似self、cls 这样的特殊参数，因此 Python 解释器不会对它包含的参数做任何类或对象的绑定。也正因为如此，类的静态方法中无法调用任何类属性和类方法。态方法只是名义上归属类管理，但是不能使用类变量和实例变量，是类的工具包放在函数前（该函数不传入self或者cls），所以不能访问类属性和实例属性

### 面向对象中super的作用

super() 函数是用于调用父类(超类)的一个方法。

super 是用来解决多重继承问题的，直接用类名调用父类方法在使用单继承的时候没问题，但是如果使用多继承，会涉及到查找顺序、重复调用（钻石继承）等种种问题。

### 是否使用过functools中的函数，其作用是什么

Python的functools模块用以为可调用对象（callable objects）定义高阶函数或操作。简单地说，就是基于已有的函数定义新的函数。所谓高阶函数，就是以函数作为输入参数，返回也是函数。

### setattr，getattr，delattr函数使用详解？

都是\_\_func\_\_

1.setattr(self,name,value)：如果想要给 name 赋值的话，就需要调用这个方法。

2.getattr(self,name)：如果 name 被访问且它又不存在，那么这个方法将被调用。

3.delattr(self,name)：如果要删除 name 的话，这个方法就要被调用了。

### 请描述抽象类和接口类的区别和联系？

(1) 抽象类

规定了一系列的方法，并规定了必须由继承类实现的方法。由于有抽象方法的存在，所以抽象类不能实例化。可以将抽象类理解为毛坯房，门窗、墙面的样式由你自己来定，所以抽象类与作为基类的普通类的区别在于约束性更强。

(2) 接口类

与抽象类很相似，表现在接口中定义的方法，必须由引用类实现，但他与抽象类的根本区别在于用途：与不同个体间沟通的规则（方法），你要进宿舍需要有钥匙，这个钥匙就是你与宿舍的接口，你的同室也有这个接口，所以他也能进入宿舍，你用手机通话，那么手机就是你与他人交流的接口。

(3) 区别和关联

接口是抽象类的变体，接口中所有的方法都是抽象的。而抽象类中可以有非抽象方法。抽象类是声明方法的存在而不去实现它的类。

接口可以继承，抽象类不行。

接口定义方法，没有实现的代码，而抽象类可以实现部分方法。

接口中基本数据类型为 static 而抽类象不是。

接口可以继承，抽象类不行。

可以在一个类中同时实现多个接口。

接口的使用方式通过 implements 关键字进行，抽象类则是通过继承 extends 关键字进行。

### 请描述方法重载与方法重写

1)方法重载

是在一个类里面，方法名字相同，而参数不同。返回类型呢？可以相同也可以不同。重载是让类以统一的方式处理不同类型数据的一种手段。

(2) 方法重写

子类不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

## Python进阶

### Python内存管理

内存管理机制：引用计数、垃圾回收、内存池。

**引用计数**

引用计数是一种非常高效的内存管理手段， 当一个 Python 对象被引用时其引用计数增加 1， 当其不再被一个变量引用时则计数减 1. 当引用计数等于 0 时对象被删除。

**垃圾回收**

小整数和字符常驻内存Pyhton内存用链表来管理

(1) 引用计数

简单，高效。但引用存在循环引用，导致内存泄漏Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如创建、引用、传参、存在容器里，对象的引用计数+1；如果引用被del，销毁，离开作用域，对象的引用计数-1。但出现循环引用的话，引用计数机制就不再起有效的作用了

(2)标记清除

如果两个对象的引用计数都为 1，但是仅仅存在他们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，它们的引用计数虽然表现为非 0，但实际上有效的引用计数为 0。所以先将循环引用摘掉，就会得出这两个对象的有效计数。

(3) 分代回收

从前面“标记-清除”这样的垃圾收集机制来看，这种垃圾收集机制所带来的额外操作实际上与系统中总的内存块的数量是相关的，当需要回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作。

举个例子：当某些内存块 M 经过了 3 次垃圾收集的清洗之后还存活时，我们就将内存块 M 划到一个集合 A 中去，而新分配的内存都划分到集合 B 中去。当垃圾收集开始工作时，大多数情况都只对集合 B 进行垃圾回收，而对集合 A 进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合 B 中的某些内存块由于存活时间长而会被转移到集合 A 中，当然，集合 A 中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

**内存池**

(1) Python 的内存机制呈现金字塔形状，-1，-2 层主要有操作系统进行操作

(2) 第 0 层是 C 中的 malloc，free 等内存分配和释放函数进行操作

(3)第 1 层和第 2 层是内存池，有 Python 的接口函数 PyMem\_Malloc 函数实现，当对象小于256K 时有该层直接分配内存

(4) 第 3 层是最上层，也就是我们对 Python 对象的直接操作

Python 在运行期间会大量地执行 malloc 和 free 的操作，频繁地在用户态和核心态之间进行切换，这将严重影响 Python 的执行效率。为了加速 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。

Python 内部默认的小块内存与大块内存的分界点定在 256 个字节，当申请的内存小于 256 字节时，PyObject\_Malloc 会在内存池中申请内存；当申请的内存大于 256 字节时，PyObject\_Malloc 的行为将蜕化为 malloc 的行为。当然，通过修改 Python 源代码，我们可以改变这个默认值，从而改变 Python 的默认内存管理行为

### 闭包

闭包是一个可以由另一个函数动态生成的函数，并且**可以改变和存储函数外创建的变量的值**。事务做更高层次的抽象，用闭包会相当舒服。比如我们要写一个二元一次函数，如果不使用闭包的话相信你可以轻而易举的写出来，下面让我们来用闭包的方式完成这个一元二次方程。

### 迭代器和生成器

迭代器是一个更抽象的概念，任何对象，如果它的类有 next 方法和 iter 方法返回自己本身，对于 string、list、dict、tuple 等这类容器对象，使用 for 循环遍历是很方便的。在后台 for 语句对容器对象调用 iter()函数，iter()是 python 的内置函数。iter()会返回一个定义了 next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是 python 的内置函数。在没有后续元素时，next()会抛出一个 StopIteration 异常。

生成器（Generator）是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，只是在需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）

区别：生成器能做到迭代器能做的所有事,而且因为自动创建了 iter()和 next()方法,生成器显得特别简洁,而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序状态的自动方法,当发生器终结时,还会自动抛出 StopIteration 异常。

yield 就是保存当前程序执行状态。你用 for 循环的时候，每次取一个元素的时候就会计算一次。用yield 的函数叫 generator，和 iterator 一样，它的好处是不用一次计算所有元素，而是用一次算一次，可以节省很多空间。generator每次计算需要上一次计算结果，所以用 yield，否则一 return，上次计算结果就没了

### 装饰器

装饰器作为 Python 高级语言特性中的重要部分，是修改函数的一种超级便捷的方式，适当使用能够有效提高代码的可读性和可维护性，非常的便利灵活。

「装饰器」本质上就是一个函数，这个函数的特点是可以接受其它的函数当作它的参数，并将其替换成一个新的函数（即返回给另一个函数）。函数可以赋值给变量，函数可嵌套，函数对象可以作为另一个函数的参数。

装饰器本质上是一个 Python 函数，它可以在让其他函数在不需要做任何代码的变动的前提下增加额外的功能。装饰器的返回值也是一个函数的对象，它经常用于有切面需求的场景。比如：插入日志、性能测试、事务处理、缓存、权限的校验等场景 有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码并发并继续使用。

### 是否使用过functools中的函数？其作用是什么？

Python的functools模块用以为可调用对象（callable objects）定义高阶函数或操作。简单地说，就是基于已有的函数定义新的函数。

所谓高阶函数，就是以函数作为输入参数，返回也是函数。

### 内存泄漏

而是应用程序分配某段内存后，由于设计错误，程序不能释放内存，致程序运行速度减慢甚至系统崩溃等严重后果。

del() 函数的对象间的循环引用是导致内存泄漏的主凶。不使用一个对象时使用:del object 来删除一个对象的引用计数就可以有效防止内存泄漏问题。通过 Python 扩展模块 gc 来查看不能回收的对象的详细信息。可以通过 sys.getrefcount(obj) 来获取对象的引用计数，并根据返回值是否为 0 来判断是否内存泄漏。

当退出 Python 时，是否释放全部内存？不是。

循环引用其它对象或引用自全局命名空间的对象的模块，在Python退出时并非完全释放。另外，也不会释放C库保留的内存部分

### Python 程序的运行方面，有什么手段能提升性能

1、使用多进程，充分利用机器的多核性能

2、对于性能影响较大的部分代码，可以使用 C 或 C++ 编写

3、对于 IO 阻塞造成的性能影响，可以使用 IO 多路复用来解

4、尽量使用 Python 的内建函数

5、尽量使用局部变量

### 4G 内存怎么读取一个 5G 的数据

迭代器，多次读取

可以通过 linux 命令 split 切割成小文件，然后再对数据进行处理，此方法效率比较高。可以按照行 数切割，可以按照文件大小切割。

### 如何在Python中管理内存

python中的内存管理由Python私有堆空间管理。所有Python对象和数据结构都位于私有堆中。程序员无权访问此私有堆。Python解释器负责处理这个问题。

Python对象的堆空间分配由Python的内存管理器完成。核心API提供了一些程序员编写代码的工具。

Python还有一个内置的垃圾收集器，它可以回收所有未使用的内存，并使其可用于堆空间。

### GIL进程线程协程

多进程适合在 CPU 密集型操作(cpu 操作指令比较多，如位数多的浮点运算)。

多线程适合在 IO 密集型操作(读写数据操作较多的，比如爬虫)。

进程是资源分配的最小单位，一个程序至少有一个进程。线程是程序执行的最小单位，一个进程至少有一个线程

Python实现并发的库：线程，进程，协程，threading。GIL 是python的全局解释器锁，同一进程中假如有多个线程运行，一个线程在运行python程序的时候会霸占python解释器（加了一把锁即GIL），使该进程内的其他线程无法运行，等该线程运行完后其他线程才能运行。如果线程运行过程中遇到耗时操作，则解释器锁解开，使其他线程运行。所以在多线程中，线程的运行仍是有先后顺序的，并不是同时进行。

Process语法结构如下：

Process([group [, target [, name [, args [, kwargs]]]]])

target：如果传递了函数的引用，可以任务这个子进程就执行这里的代码

args：给target指定的函数传递的参数，以元组的方式传递

kwargs：给target指定的函数传递命名参数

name：给进程设定一个名字，可以不设定

Process创建的实例对象的常用方法：

start()：启动子进程实例（创建子进程）

is\_alive()：判断进程子进程是否还在活着

join([timeout])：是否等待子进程执行结束，或等待多少秒

terminate()：不管任务是否完成，立即终止子进程

Process创建的实例对象的常用属性：

name：当前进程的别名，默认为Process-N，N为从1开始递增的整数

pid：当前进程的pid（进程号）

线程threading 函数模块

activeCount() 获取当前活动中的Thread对象个数

currentThread() 获取当前的Thread对象

enumerate() 获取当前活动的Thread对象列表

settrace(func) 为所有线程设置一个跟踪（trace）函数

setprofile(func) 为所有线程设置配置文件（profile）函数

stack\_size(size=None) 获取新创建线程的栈大小，也可设置线程栈的大小为size

协程是一种比线程更加轻量级的存在，最重要的是，协程不被操作系统内核管理，协程是完全由程序控制的。

运行效率极高，协程的切换完全由程序控制，不像线程切换需要花费操作系统的开销,线程数量越多，协程的优势就越明显。协程不需要多线程的锁机制，因为只有一个线程，不存在变量冲突。对于多核CPU，利用多进程+协程的方式，能充分利用CPU，获得极高的性能

### 你有哪些多线程开发良好的实践

1.给线程命名

2.最小化同步范围

3.优先使用volatile

4.尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore

5.优先使用并发容器而非同步容器.

6.考虑使用线程池

10.线程池

### 什么是线程池

线程池是一种多线程处理形式，处理过程中将任务提交到线程池，任务的执行交由线程池来管理。

如果每个请求都创建一个线程去处理，那么服务器的资源很快就会被耗尽，使用线程池可以减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

#### 几种常见的线程池及使用场景

1、newSingleThreadExecutor

单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。（一个单线程的线程池，可以用于需要保证顺序执行的场景，并且只有一个线程在执行。）

2、newFixedThreadPool

定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。可以用于已知并发压力的情况下，对线程数做限制。

3、newCachedThreadPool

可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。比较适合处理执行时间比较小的任务。（一个可以无限扩大的线程池， ）

4、newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。可以延时启动，定时启动的线程池，适用于需要多个后台线程执行周期任务的场景。

5. newWorkStealingPool：一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行。

#### 为什么要用线程池？

创建线程和销毁线程的花销是比较大的，这些时间有可能比处理业务的时间还要长。这样频繁的创建线程和销毁线程，再加上业务工作线程，消耗系统资源的时间，可能导致系统资源不足。使用线程池可以把创建和销毁的线程的过程去掉

使用线程池的好处：

（1）降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

（2）提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

（3）提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

#### 几种重要的参数

corePoolSize：线程池中的核心线程数量，这几个核心线程，只是在没有用的时候，也不会被回收

maximumPoolSize：线程池中可以容纳的最大线程的数量

keepAliveTime：就是线程池中除了核心线程之外的其他的最长可以保留的时间，因为在线程池中，除了核心线程即使在无任务的情况下也不能被清 除，其余的都是有存活时间的，意思就是非核心线程可以保留的最长的空闲时间，

Util：就是计算这个时间的一个单位。

workQueue：就是等待队列，任务可以储存在任务队列中等待被执行，执行的是先进先出原则。

threadFactory：创建线程的线程工厂。

Handler：一种拒绝策略，我们可以在任务满了之后，拒绝执行某些任务。

#### 线程池的拒绝策略

当请求任务不断的过来，而系统此时又处理不过来的时候，我们需要采取的策略是拒绝服务。RejectedExecutionHandler接口提供了拒绝任务处理的自定义方法的机会。在ThreadPoolExecutor中已经包含四种处理策略。

AbortPolicy策略： 当任务添加到线程池中被拒绝时，它将抛出 RejectedExecutionException 异常。(异常)

CallerRunsPolicy 策略：只要线程池未关闭，该策略直接在调用者线程中，运行当前的被丢弃的任务。

DiscardOleddestPolicy策略： 线程池会放弃等待队列中最旧的未处理任务，然后将被拒绝的任务添加到等待队列中。

DiscardPolicy策略：线程池将丢弃被拒绝的任务，不予任何处理。

除了JDK默认提供的四种拒绝策略，我们可以根据自己的业务需求去自定义拒绝策略，自定义的方式很简单，直接实现RejectedExecutionHandler接口即可。

#### 线程池都有哪几种工作队列

1、ArrayBlockingQueue

是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2、LinkedBlockingQueue

一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列

3、SynchronousQueue

一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

4、PriorityBlockingQueue

一个具有优先级的无限阻塞队列。

### sleep wait join怎么用的

sleep:强制当前线程休眠.当休眠时间到期后恢复到可运行状态.不一定会立即执行,具体取决于线程调度器.(不释放锁)

wait:使当前线程,进入到一个和对象相关的等待池中,同时失去对象的锁,其他线程可以访问，通过notify(),notifyAll(),或者指定超时时间来唤醒当前等待池中线程.进入到锁池，用在synchronized代码块中,否则会抛出异常.

join：Python中join所完成的工作就是线程同步，即主线程任务结束之后，进入阻塞状态，一直等待其他的子线程执行结束之后，主线程在终止

## Python后端

### 中间件Redis

#### redis的过期策略以及内存淘汰机制

redis采用的是定期删除+惰性删除策略。

为什么不用定时删除策略?

定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略.

定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。

1.

于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。

在redis.conf中有一行配置

# maxmemory-policy allkeys-lru

该配置就是配内存淘汰策略:当内存不足以容纳新写入数据时

1）noeviction：新写入操作会报错。应该没人用吧。

2）allkeys-lru：在键空间中，移除最近最少使用的key。推荐使用。

3）allkeys-random：在键空间中，随机移除某个key。应该也没人用吧，你不删最少使用Key,去随机删。

4）volatile-lru：在设置了过期时间的键空间中，移除最近最少使用的key。这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐

5）volatile-random：在设置了过期时间的键空间中，随机移除某个key。依然不推荐

6）volatile-ttl：在设置了过期时间的键空间中，有更早过期时间的key优先移除。不推荐

ps:如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

#### 缓存穿透 击穿 雪崩

这几种情况都是从缓存没有获取到数据，大量的并发请求到了数据源，给数据源造成很大压力，从而可能引发问题

1.缓存穿透：key对应的数据在数据源并不存在，每次针对此key的请求从缓存获取不到，请求都会到数据源，从而可能压垮数据源。比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。

采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2. 缓存击穿：

key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试。

3.缓存雪崩：当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

#### **Redis优势和特点**

特点1.内存数据库，速度快，也支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

2.Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

3.Redis支持数据的备份，即master-slave模式的数据备份。

4支持事务

优势：

1.性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

2.丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

3.原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。（事务）

4.丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

#### 知道Redis的持久化吗？都有什么缺点和优点？具体底层实现呢？

RDB:

优点：最大化发挥redis的性能，主进程不需要进行任何I/O操作，只需要派生一个子进程来处理，文件非常紧凑，恢复数据的效率高。

缺点：数据恢复时一致性和完整性较差，因为也许最后一次备份前就宕机了，那么在最后一次备份到宕机中间的数据是不会存在的

AOF:

优点：能保持数据的一致性和完整性

缺点：AOF文件比RDB文件大，在读取过程中，会比RDB更慢一些。

底层实现

RDB:是在达到指定的时间或者操作次数后，自动将在内存中的数据写入磁盘

AOF:是日志形式，当数据写入内存中的时候，在日志文件下记录下所有写操作。

### 消息队列

#### 什么是消息队列

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。目前使用较多的消息队列有ActiveMQ，RabbitMQ，Kafka，RocketMQ。

另外，我们知道队列 Queue 是一种先进先出的数据结构，所以消费消息时也是按照顺序来消费的。比如生产者发送消息1,2,3...对于消费者就会按照1,2,3...的顺序来消费。但是偶尔也会出现消息被消费的顺序不对的情况，比如某个消息消费失败又或者一个 queue 多个consumer 也会导致消息被消费的顺序不对，我们一定要保证消息被消费的顺序正确。

除了上面说的消息消费顺序的问题，使用消息队列，我们还要考虑如何保证消息不被重复消费？如何保证消息的可靠性传输（如何处理消息丢失的问题）？......等等问题。所以说使用消息队列也不是十全十美的，使用它也会让系统可用性降低、复杂度提高，另外需要我们保障一致性等问题。

把数据放到消息队列叫做生产者 从消息队列里边取数据叫做消费者

#### 为什么使用消息队列

1. 通过异步处理提高系统性能（削峰、减少响应所需时间）;

2.降低系统耦合性。如果在面试的时候你被面试官问到这个问题的话，一般情况是你在你的简历上涉及到消息队列这方面的内容，这个时候推荐你结合你自己的项目来回答。

(1) 通过异步处理提高系统性能（削峰、减少响应所需时间）

通过异步处理提高系统性能

如上图，在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

通过以上分析我们可以得出消息队列具有很好的削峰作用的功能——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。举例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图所示：

合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击

因为用户请求数据写入消息队列之后就立即返回给用户了，但是请求数据在后续的业务校验、写数据库等操作中可能失败。因此使用消息队列进行异步处理之后，需要适当修改业务流程进行配合，比如用户在提交订单之后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功，以免交易纠纷。这就类似我们平时手机订火车票和电影票。

(2) 降低系统耦合性

我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响较小，这样系统的可扩展性无疑更好一些。

我们最常见的事件驱动架构类似生产者消费者模式，在大型网站中通常用利用消息队列实现事件驱动结构。如下图所示：

利用消息队列实现事件驱动结构

消息队列使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。 从上图可以看到消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计。

消息接受者对消息进行过滤、处理、包装后，构造成一个新的消息类型，将消息继续发送出去，等待其他消息接受者订阅该消息。因此基于事件（消息对象）驱动的业务架构可以是一系列流程。

另外为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。

备注： 不要认为消息队列只能利用发布-订阅模式工作，只不过在解耦这个特定业务环境下是使用发布-订阅模式的。除了发布-订阅模式，还有点对点订阅模式（一个消息只有一个消费者），我们比较常用的是发布-订阅模式。 另外，这两种消息模型是 JMS 提供的，AMQP 协议还提供了 5 种消息模型。

#### 使用消息队列带来的一些问题

1.系统可用性降低： 系统可用性在某种程度上降低，为什么这样说呢？在加入MQ之前，你不用考虑消息丢失或者说MQ挂掉等等的情况，但是，引入MQ之后你就需要去考虑了！

2.系统复杂性提高： 加入MQ之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！

3.一致性问题： 我上面讲了消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了!

#### 常见的消息队列对比

对比方向 概要

吞吐量 万级的 ActiveMQ 和 RabbitMQ 的吞吐量（ActiveMQ 的性能最差）要比 十万级甚至是百万级的 RocketMQ 和 Kafka 低一个数量级。

可用性 都可以实现高可用。ActiveMQ 和 RabbitMQ 都是基于主从架构实现高可用性。RocketMQ 基于分布式架构。 kafka 也是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用

时效性 RabbitMQ 基于erlang开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。其他三个都是 ms 级。

功能支持 除了 Kafka，其他三个功能都较为完备。 Kafka 功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准

消息丢失 ActiveMQ 和 RabbitMQ 丢失的可能性非常低， RocketMQ 和 Kafka 理论上不会丢失。

#### 总结：

• ActiveMQ 的社区算是比较成熟，但是较目前来说，ActiveMQ 的性能比较差，而且版本迭代很慢，不推荐使用。

• RabbitMQ 在吞吐量方面虽然稍逊于 Kafka 和 RocketMQ ，但是由于它基于 erlang 开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。但是也因为 RabbitMQ 基于 erlang 开发，所以国内很少有公司有实力做erlang源码级别的研究和定制。如果业务场景对并发量要求不是太高（十万级、百万级），那这四种消息队列中，RabbitMQ 一定是你的首选。如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

• RocketMQ 阿里出品，Java 系开源项目，源代码我们可以直接阅读，然后可以定制自己公司的MQ，并且 RocketMQ 有阿里巴巴的实际业务场景的实战考验。RocketMQ 社区活跃度相对较为一般，不过也还可以，文档相对来说简单一些，然后接口这块不是按照标准 JMS 规范走的有些系统要迁移需要修改大量代码。还有就是阿里出台的技术，你得做好这个技术万一被抛弃，社区黄掉的风险，那如果你们公司有技术实力我觉得用RocketMQ 挺好的

• kafka 的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms 级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展。同时 kafka 最好是支撑较少的 topic 数量即可，保证其超高吞吐量。kafka 唯一的一点劣势是有可能消息重复消费，那么对数据准确性会造成极其轻微的影响，在大数据领域中以及日志采集中，这点轻微影响可以忽略这个特性天然适合大数据实时计算以及日志收集。