写在前面

- 这些都是平时逛技术社区时看到的我认为不错的内容，顺手记录在印象笔记中，所以分类总结下来。能够找到来源出处的都有所注明，如果涉及侵权可以联系我删除或进行来源标注。

公司选择

- 写 Python 可供选择的公司本来不多，但中小公司还是很多，但找到小而美的就很少，说说个人心中“小而美”的标准😐：
    - 团队互补能力强，虽小但是能够cover的业务能力广；
    - 有明锐的嗅觉和创新能力，并且能够专注于某个行业领域或者发展方向；
    - 快或缓，有清晰的发展规划；
    - 最后一条，商业模式应该在很小的规模下就能验证，据说这句话是雷军说的
    - 多抓鱼，长亭科技，待补充。。。（欢迎提交issue或推荐公司😂）
- 一个学长告诉我找工作很重要，更重要的是找准一个行业。

简历投递

- 岗位对上三条及以上就可以投了，要不要实习都可以尝试一下！😂
- 自己设计一张简历投递状态记录表，投递中 -> 笔试-> 一面 -> 二面-> hr面 -> 面试结果(被拒绝/get offer)
- 找正确的投递渠道，拉勾上找到几页满足岗位要求的公司，然后去官网，一般都有加入我们，没有的就不用考虑啦(官网连这些基本都做不到，还招啥人，可以看看一面数据，连招过的实习生都有展示)，当然还有没有官网的公司😂，不想放过的话，去V2EX、知乎搜关键字会发现惊喜。

- thank you letter！面试结束后发感谢信！

面试策略

- 项目回答
  - 任何一个公司的面试，都一定会涉及到作为一个工程师最核心的价值——解决问题的能力，具体来说就是你做过的项目，这块是面试准备时的重中之重，应该作为最高优先级来对待。面试官反复的追问项目的各个地方的技术实现细节，就想看看有没有哪个地方是有一定的技术难度的，可以体现出这个候选人的一些项目上的亮点，但是如果说来说去总是从业务的角度去说，就说有哪些子系统组成，如何交互的，没有技术含量的东西在项目里体现出来，就会有点本末倒置了。
  - 自己至少应该反复思考，你目前负责的系统应该引入什么样的技术架构，采用何种技术方案，才能抗住各种冲击。同时搜一下国内大型互联网公司的技术架构，他们使用了哪些技术，对于某个技术难点采用了什么技术方案。可以对面试官阐述一下你对这个项目一些问题的思考，这样设计可以解决什么技术问题，有没有更好的方案选择。
- 投递策略
  - 第一阶段练手：当你觉得自己充分的准备好了面试之后，你应该先找几个感觉对自己会有一定技术挑战的公司，但是并不是自己特别意向想要去的公司去投递简历面试一下，这个阶段一般会暴露出来很多问题。只要面试官的技术实力比你强，那么在面试的过程中，一定会问到你一些问题，是你之前没注意以及没准备好的。此时你很可能会发现，刚开始面试的头三四家公司，每家公司聊的都不太顺畅，每家公司总有那么几个问题没回答好，然后都没拿到offer。但是这个阶段的好处是，你发现了自己很多薄弱环节，这个时候你应该尽快通过上网查资料的方式填补好自己对一些薄弱问题的弱项，然后迅速总结，内化为自己的语言，并且能落地到纸上画图实现。
  - 第二个阶段冲刺：经过了第一个阶段的被虐之后，每个人的面试能力都会加强很多，而且找到了一些面试的感觉，对面试的节奏、对答都有了更好的把控。这个时候就可以尝试去冲刺一下心仪的大厂了，在这个阶段里，需要全力以赴去面试。

面试总结

- 本来我认为将自己使用过的技术掌握扎实就行，技术面试靠“面试总结”有点死记硬背应付的赶脚，但实习时的 mentor 告诉我针对面试常见问题进行记忆也是为了提升面试效果，再说技术广度如此大，要是都用过会被严重怀疑仅仅是个码农，不能从底层发现问题，只会照着 api 和框架 code，这也是为自己写这份总结的原因。
- 总之面试是一场持久战，要软硬兼备。硬性条件：面试官对候选人的技术广度、技术深度、基础功底、系统设计、项目经验几个角度来进行的考察，软性条件：一些软素质比如说学习能力、表达能力、沟通能力都比较重要。哦，对了，绝对不能忽略运气成分😁

技术部分

语言特性篇

- 学习书籍推荐：《流畅的Python》

谈谈 **Python** 和其他语言的区别

- Python是强类型、动态类型的语言

Python 2和**3**的区别

- python2中，print是个特殊语句，python3中print是函数。
- python中有两种字符类型：字节字符串和文本字符串。

版本	**python2**	**python3**
字节字符串	str	bytes
文本字符串	Unicode	str

- python2中默认的字符串类型默认是ASCII，python3中默认的字符串类型是Unicode。
- python2中除法 / 的结果是整型，python3中是浮点类型。
- python2中默认类是旧式类，需要显式继承新式类（object）来创建新式类。python3中完全移除旧式类，所有类都是新式类，但仍可显式继承object类。
- 兼容问题：six库，2to3，__future__包

闭包

- 闭包指延伸(延伸的意思是seriers在average函数用，但在average_nums中定义的)了作用域的函数，访问定义体之外定义的非全局变量。创建一个闭包必须满足以下几点:
  - 必须有一个内嵌函数
  - 内嵌函数必须引用外部函数中的变量
  - 外部函数的返回值必须是内嵌函数

```
def average_num():
    series=[]
    def average(num):
        # series自由变量,未在本地作用域绑定
        series.append(num)
        return sum(series)/len(series)
    return average
aver=average_num()
```

- nonlocal 关键字将变量count，total 手动标记为自由变量

```
def average_num():
    count=0
    total=0
    def average(num):
        nonlocal total,count
        total+=num
        count+=1
        return total/count
    return average
aver=average_num()
print(aver(10))
```

- 顺便提下 global 关键字，Python 默认函数定义体中赋值的变量是局部变量，所以要在函数中使用全局变量，须用global声明。(下面这句去掉global就会报错， `local variable 'b' referenced before assignment`)

```
b=9
def test(a):
    print(a)
    global b
    print(b)
    b=6
test(3)
```

装饰器

- 主要参考
  - 理解 Python 装饰器看这一篇就够了
- 闭包的一个重要应用就是装饰器，函数装饰器在导入模块时立即执行，被装饰的函数只在明确调用时执行。装饰器采用语法糖，使用方便，用途多样，既可以自定义在装饰器中指定日志级别，也可以使用官方提供的预激协程装饰器等等。

```
# 手动实现一个带参数的装饰器
def use_arg(argument):
    def decorate(fun):
        def wrapper(*args,**kwargs):
            print('%s is running'%fun.__name__)
            print('传入的参数 %s'%argument)
            return fun()
        return wrapper
    return decorate
@use_arg('hahhaa')
def demo():
    pass
demo()
```

谈谈GC

- 主要参考
  - Python 进阶：浅析「垃圾回收机制」(上篇)
- 主要使用**引用计数**进行垃圾回收
- 通过**标记-清理**解决容器对象产生循环引用的问题
- 通过**分代回收**以空间换时间的方式来提高垃圾回收的效率

GIL的理解

- 主要参考
  - python中的GIL详解
- GIL是一个防止多线程并发执行机器码的Mutex：cpython解释器的内存管理不是线程安全的，保护多线程情况下对python对象的访问，cpython使用简单的锁机制避免多个线程同时执行字节码。

- 缺陷：限制程序的多核执行，同一时间只能有一个线程执行字节码，CPU密集型程序（大量时间花在计算上）难以利用多核优势；但是IO期间会释放GIL，对IO密集型程序(大量时间花在网络传输上，资源调度)影响小。
- 规避影响策略：CPU密集型使用多进程+进程池，IO密集型采用多线程/协程的策略；cython扩展（将python转换为c代码）
- GIL释放：Python会计算当前已执行的微代码数量，达到一定阈值后强制释放GIL；系统分配的时间片用完释放；IO操作时释放。
- 一个操作如果是一个字节码指令可以完成就是原子的，原子操作是可以保证线程安全的，有了GIL之后仍然要关注线程安全，必要时候需人为加锁。

```
import threading
lock=threading.Lock()
n=[0]
def demo():
    #with lock:
    n[0]=n[0]+1
threads=[]
for i in range(5000):
    t=threading.Thread(target=demo)
    threads.append(t)
for t in threads:
    t.start()
print(n)
```

- 剖析程序性能：内置profile/cprofile工具

Python传参

- 主要参考
  - Python传入参数的几种方法
- Python中参数传递方式：位置传递、默认参数、可变参数、关键字参数、命名关键字参数
- Python 函数参数传递实际叫传对象（call by object），可以把不可变对象传参理解为传值，可变对象参数理解为传引用。

深拷贝和浅拷贝

- 浅拷贝(copy)拷贝父对象，不会拷贝对象的内部的子对象。深拷贝(deepcopy)完全拷贝了父对象及其子对象。

```
import copy
a = [1, 2, 3, 4, ['a', 'b']] #原始对象
b = a                        #赋值，传对象的引用
c = copy.copy(a)             #对象拷贝，浅拷贝
d = copy.deepcopy(a)         #对象拷贝，深拷贝
a.append(5)                  #修改对象a
a[4].append('c')             #修改对象a中的['a', 'b']数组对象
```

鸭子类型

- 鸭子类型就是：如果走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子（If it walks like a duck and quacks like a duck, it must be a duck）。鸭子类型是编程语言中动态类型语言中的一种设计风格，一个对象的特征不是由父类决定，而是通过对象的方法决定的。（重点关注接口，不关注对象）

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')
```

猴子补丁

- Monkey patch就是在运行时对已有的代码进行修改，达到hot patch的目的。运行时动态替换模块的方法，运行时动态增加模块的方法（慎用）

```
class Foo(object):
    def bar(self):
        print('Foo.bar')
def bar(self):
    print('Modified bar')
Foo().bar()
Foo.bar = bar
Foo().bar()
```

Python中的作用域

- Python的作用域一共有4种: L （Local） 局部作用域，E （Enclosing） 闭包函数外的函数中，G （Global） 全局作用域，B （Built-in） 内建作用域。以 L –> E –> G –>B 的规则查找，即：在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内建中找。

函数式编程

lambda函数

- lambda 定义了一个匿名函数，lambda 并不会带来程序运行效率的提高，只会使代码更简洁。使用lambda内不要包含循环，如果有，定义函数来完成，使代码获得可重用性和更好的可读性。lambda 是为了减少单行函数的定义而存在的。

```
 list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
# [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map函数

- map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

reduce函数

- reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：
  `reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)`

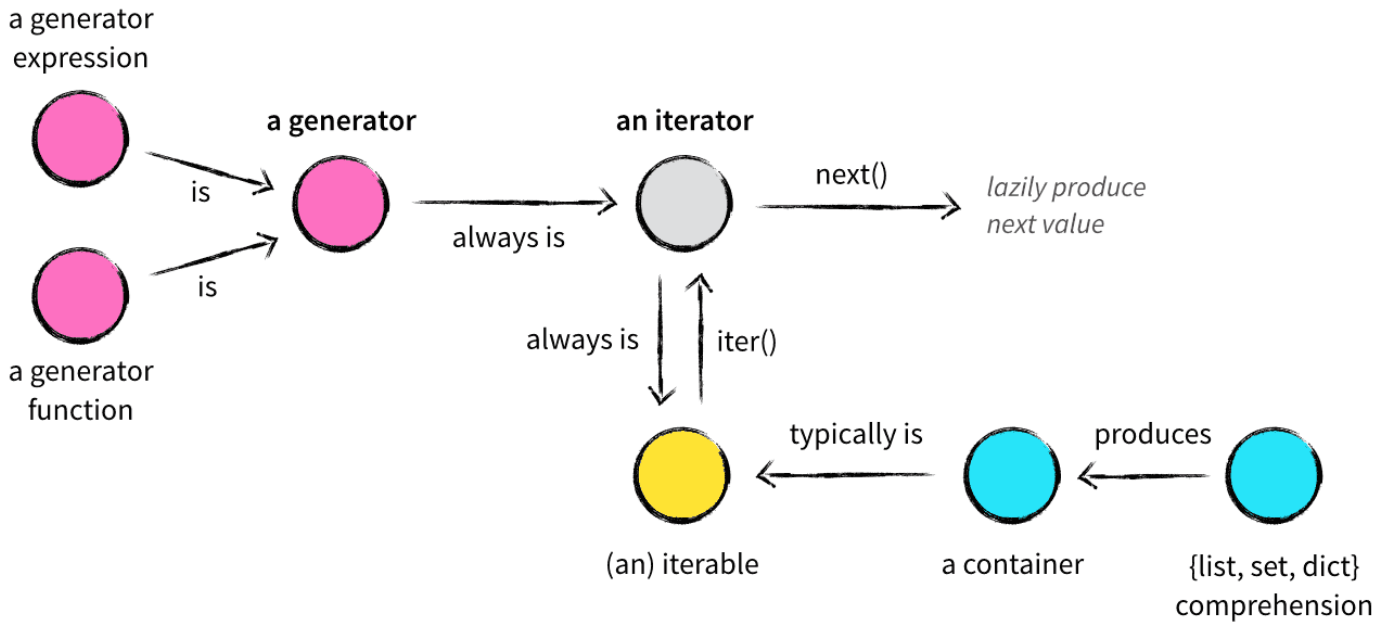
filter函数

- filter()函数用于过滤序列。filter()接收一个函数和一个序列，把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

```
def is_odd(n):
    return n % 2 == 1
list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
# 结果: [1, 5, 9, 15]
```

迭代器和生成器

- 主要参考
  - Python中的可迭代对象、迭代器和生成器的异同点



- 可迭代对象和容器一样是一种通俗的叫法，并不是指某种具体的数据类型，可迭代对象实现了__iter__方法，该方法返回一个迭代器对象。
- 迭代器有一种具体的迭代器类型，它是一个带状态的对象，他能在你调用next()方法的时候返回容器中的下一个值，迭代器不会一次性把所有元素加载到内存，而是需要的时候才生成返回结果(不同于容器)。任何实现了__iter__和__next__()方法的对象都是迭代器，__iter__返回迭代器自身，__next__返回容器中的下一个值，如果容器中没有更多元素了，则抛出StopIteration异常。迭代器每次调用next()方法的时候做两件事：为下一次调用next()方法修改状态，生成当前调用的返回结果。
- 生成器其实是一种特殊的迭代器，这种迭代器更加优雅，它不需要写__iter__()和__next__()方法了，只需要一个yiled关键字， 生成器一定是迭代器（反之不成立）。

协程

- 主要参考

- Python协程深入理解

- 协程调用细节：实际上next()和send()在一定意义上作用是相似的，区别是send()可以传递yield表达式的值进去，而next()不能传递特定的值，只能传递None进去。因此，我们可以看做c.next() 和 c.send(None) 作用是一样的。第一次调用时，请使用next()语句或是send(None)，不能使用send发送一个非None的值，否则会出错的，因为没有Python yield语句来接收这个值。
- send执行的顺序。n1 = yield r这句话是从右往左执行的。当第一次send（None）时，启动生成器，从生成器函数的第一行代码开始执行，直到第一次执行完yield后，跳出生成器函数。这个过程中，n1一直没有定义。运行到send（1）时，进入生成器函数，此时，将yield r看做一个整体，赋值给它并且传回。此时即相当于把1赋值给n1，但是并不执行yield部分。下面继续从yield的下一语句继续执行，然后重新运行到yield语句，执行后，跳出生成器函数。即send和next相比，只是开始多了一次赋值的动作，其他运行流程是相同的。

```
def consumer():
    r = 'hello'
    while True:
        n1=yield r  #这里的等式右边相当于一个整体，接受回传值
        if not n1:
            return
        print('[CONSUMER] Consuming %s...' % n1)
        r='%d00 OK' % n1
c = consumer()
c.__next__()
n=0
while n<3:
    n=n+1
    r1=c.send(n)
    print('[PRODUCER] Consumer return: %s' % r1)
c.close()
```

- 预激装饰器，使用协程之前必须预激，为了避免忘记，可以在协程上使用一个特殊的装饰器。

```
from functools import wraps
def corountine(func):
    @wraps(func)
    # functools.wraps 则可以将原函数对象的指定属性复制给包装函数对象，默认有 __module__、__name__、__doc__,或者通过参数选择
    def primer(*args, **kwargs):       # 把被装饰的生成器函数替换成这里的 primer 函数；调用 primer 函数时，返回预激后的生成器
        gen = func(*args, **kwargs)    # 调用被装饰的函数，获取生成器对象。
        next(gen)                      # 预激生成器
        return gen                     # 返回生成器
    return primer

@corountine               # 预激装饰器
def coro_average():
    total = 0.0
    count = 0
    average = None
    while 1:
        term = yield average
        total += term
        count += 1
        average = total/count
coro3 = coro_average()
print (coro3.send(5))
print (coro3.send(7))
print (coro3.send(10))
coro3.close()
```

Python 面对对象编程

封装

- 将内容封装到某处
- 从某处调用被封装的内容

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age

# 初始化一个实例时调用__init__方法
tim=Person('Tim',18)
# 将Bob,16分布封装到bob的name和age属性中
bob=Person('Bob',16)
```

继承

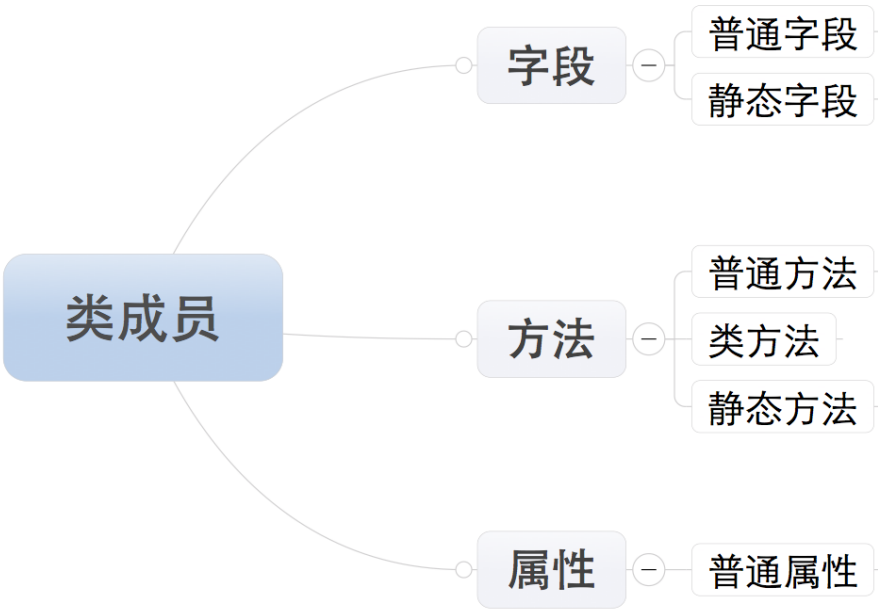
- 对于面向对象的继承来说，其实就是将多个类共有的方法提取到父类中，子类仅需继承父类而不必一一实现每个方法。Python的类如果继承了多个类，那么其寻找方法的方式（即MRO,method resolution order (方法解析顺序)，在上述查找过程中，一旦找到，则寻找过程立即中断，便不会再继续找了）有两种，可以分别粗略理解为：深度优先和广度优先。具体实现细节可以看这篇python多重继承C3算法
  - 当类是经典类时，多继承情况下，会按照深度优先方式查找，Python 2.x中默认都是经典类，只有显式继承了object才是新式类。
  - 当类是新式类时，多继承情况下，会按照广度优先方式查找Python 3.x中默认都是新式类，不必显式的继承object。
- 继承中super的调用顺序是与MRO-C3的类方法查找顺序一样的，super作用如下：
  - 如果子类继承父类不做初始化，那么会自动继承父类属性。
  - 如果子类继承父类做了初始化，且不调用super初始化父类构造函数，那么子类不会自动继承父类的属性。
  - 如果子类继承父类做了初始化，且调用了super初始化了父类的构造函数，那么子类也会继承父类的属性。

```
class A:
    def __init__(self):
        print('A')
class B(A):
    def __init__(self):
        print('B')
        super().__init__()
class C(A):
    def __init__(self):
        print('C')
        super().__init__()
class D(B, C):
    def __init__(self):
        print('D')
        super().__init__()
```

多态

- Pyhon不支持多态并且也用不到多态，多态的概念是应用于Java和C#这一类强类型语言中，而Python崇尚“鸭子类型”。

类成员



- 字段
  - 静态字段在内存中只保存一份
  - 普通字段在每个实例对象中都要保存一份

- 应用场景： 通过类创建实例对象时，如果每个实例对象都具有相同的字段，那么就使用静态字段

```
class Province:
    # 静态字段
    country = 'China'
    def __init__(self,name):
        # 普通字段
        self.name = name
obj1 =Province('BeiJing')
print(obj1.name)
print(Province.country)
```

- 方法
  - 普通实例方法：由实例对象调用；至少一个self参数；执行普通方法时，自动将调用该方法的实例对象赋值给self；
  - 类方法：由类调用；至少一个cls参数；执行类方法时，自动将调用该方法的类复制给cls；
  - 静态方法：由类调用；无默认参数；

```
class Foo:
    def __init__(self,name):
        self.name =name
    # 定义普通实例方法,至少一个self参数
    def ord_func(self):
        print(self.name)
    # 定义类方法，至少一个cls参数
    @classmethod
    def class_func(cls):
        print('类方法')
    # 定义静态方法，无默认参数
    @staticmethod
    def static_fun():
        print('静态方法')
f = Foo('ord_func')
f.ord_func()
Foo.class_func()
Foo.static_fun()
```

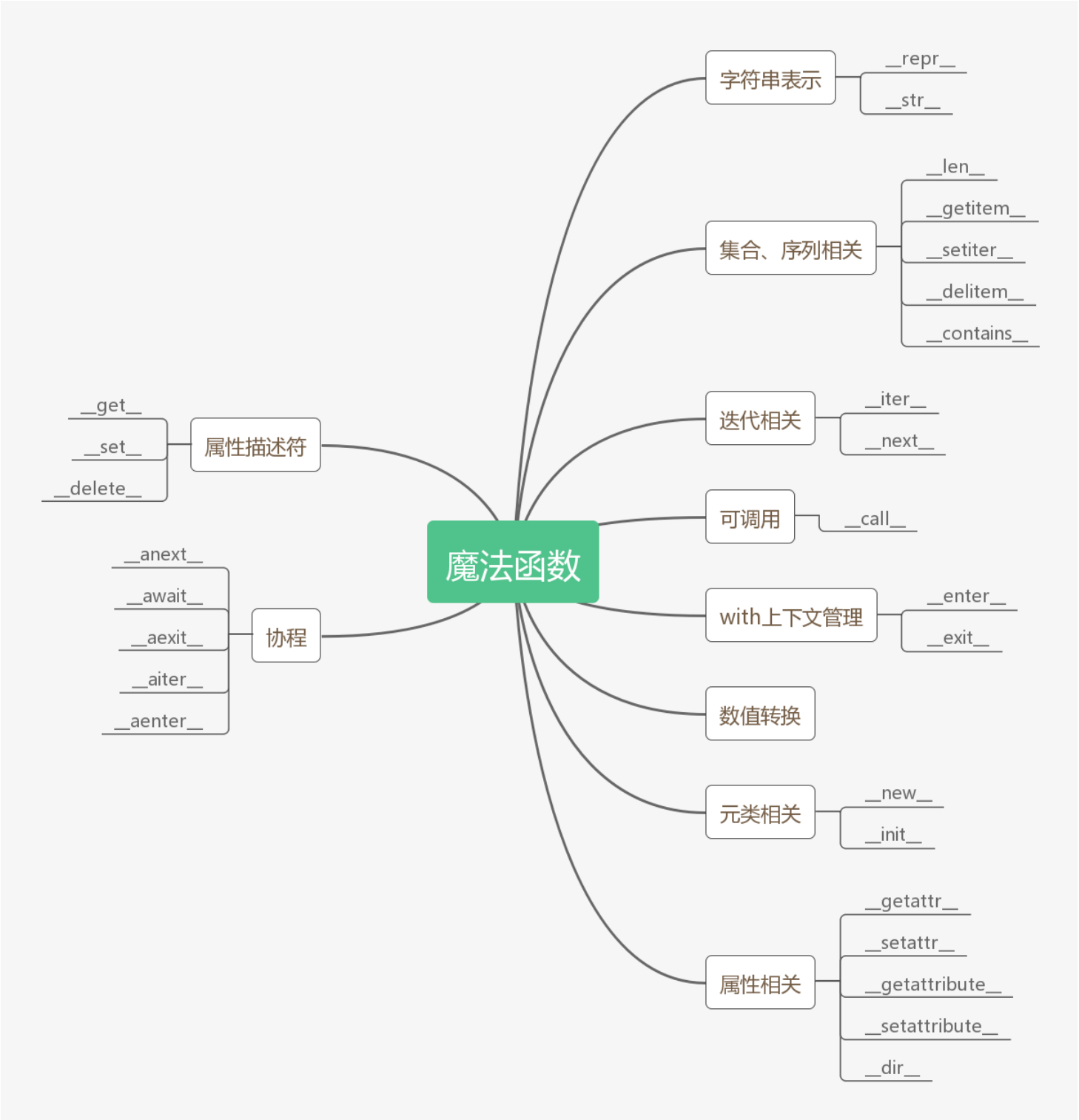
- 属性
  - 定义时，在普通方法的基础上添加@property装饰器，属性仅有一个self参数，调用时无需括号。@property装饰器可以实现其他语言所拥有的getter，setter和deleter的功能（例如实现获取，设置，删除隐藏的属性）；通过@property装饰器可以对属性的取值和赋值加以控制,提高代码的稳定性。

```
class Goods:
    # 查看属性值
    @property
    def price(self):
        print('@property')
    # 修改、设置属性
    @price.setter
    def price(self, value):
        print('@price.setter')
    # 删除属性
    @price.deleter
    def price(self):
       print('@price.deleter')
obj = Goods()
# 自动执行 @property 修饰的 price 方法，并获取方法的返回值
obj.price
# 自动执行 @price.setter 修饰的 price 方法，并将2000赋值给方法的参数
obj.price = 2000
# 自动执行 @price.deleter 修饰的 price 方法
del obj.price
```

类成员的修饰符

- _xxx 表示这是一个保护成员（属性或者方法），它不能用from module import * 导入，其他方面和公有一样访问；
- __xxx 这表示这是一个私有成员，它无法直接像公有成员一样随便访问，但可以通过 实例对象名._类名__xxx 这样的方式可以访问；
- __xxx__表示这是一个特殊成员，用来区别其他用户自定义的命名以防冲突，就是例如 __init__()、 __del__() 这些特殊方法

类的特殊成员



1. __doc __输出类的描述信息
2. __module __输出当前操作对象所在模块
3. __class __输出当前操作对象的类
4. __init __构造方法，通过类创建实例对象时，自动触发执行
5. __del __析构方法，当对象在内存中被释放时，自动触发执行，此方法一般无须定义，因为Python是一门高级语言，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给Python解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。
6. __call__对象后面加括号，触发执行，构造方法的执行是由创建对象触发的，即：对象 = 类名() ；而对于 **call** 方法的执行是由对象后加括号触发的，即：对象() 或者 类()()

```
class Foo:
    def __init__(self):
        pass
    def __call__(self,*args,**kwargs):
        print('__call__')
# 执行 __init__
obj6 = Foo()
# 执行 __call__
obj6()
```

7. __dict__类或对象中的所有成员

```
class Province:
    country = 'China'
    def __init__(self, name, count):
        self.name = name
        self.count = count
    def func(self, *args, **kwargs):
        print('func')
# 获取类的成员，即：静态字段、方法、
# 输出：{'__module__': '__main__', 'country': 'China', '__init__': <function Province.__init__ at 0x00000264B20F1E18>,
# 'func': <function Province.func at 0x00000264B4638D08>, '__dict__': <attribute '__dict__' of 'Province' objects>,
# '__weakref__': <attribute '__weakref__' of 'Province' objects>, '__doc__': None}
print(Province.__dict__)
# 输出：{'name': 'HeBei', 'count': 1000}
obj1 = Province('HeBei',1000)
print(obj1.__dict__)
obj2 = Province('HeNan', 3)
# 输出：{'name': 'HeNan', 'count': 3}
print(obj2.__dict__)
```

8. __str__如果一个类中定义了__str__方法，那么在打印对象时，默认输出该方法的返回值。

```
class Foo:
    def __init__(self):
        pass
    def __str__(self):
        return '__str__'
obj=Foo()
print(obj)
```

9. `__getitem__`、`__setitem__`、`__delitem__` 用于索引操作，如字典。以上分别表示获取、设置、删除数据。
10. `__getslice__`、`__setslice__`、`__delslice__` 该三个方法用于分片操作，如列表。
11. __iter__用于迭代器，之所以列表、字典、元组可以进行for循环。
12. `__new__`和`__init__`

    `__new__` 是一个静态方法，而 `__init__` 是一个实例方法， `__new__` 方法会返回一个创建的实例，而 `__init__` 什么都不返回，当创建一个新实例时调用 `__new__` ，初始化一个实例时用 `__init__` 。

Python自省指南

- 自省（Introspection）指运行时判断一个对象类型的能力，常见type，id，isinstance 等获取对象类型信息或 ispect 模块中的函数获取详细信息。

Python 中的元编程

- 元类是创建类的类，元类允许我们控制类的生成（修改类的属性等），用type来定义元类，元类最常见的使用场景是 ORM 框架。

```
# 通过元编程实现类自定义属性大写
class Base(type):
    def __new__(cls,name,bases,attrs):
        upper_attrs={}
        for k,v in attrs.items():
            if not k.startswith("__"):
                upper_attrs[k.upper()]=v
            else:
                upper_attrs[k]=v
        return type.__new__(cls,name,bases,upper_attrs)
class Foo(metaclass=Base):
    foo=True
    def hello(self):
        print("world")
def hello():
    print("world")
print(dir(Foo))
```

设计模式

单例模式

- 单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。当你希望在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。（None就是单例）

1. metaclass 实现

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Foo(metaclass=Singleton):
    pass
foo1=Foo()
foo2=Foo()
print(id(foo1)==id(foo2))
```

2. 函数装饰器实现

```
def singleton(cls):
    _instance = {}
    def wrapper():
        if cls not in _instance:
            _instance[cls] = cls()
        return _instance[cls]
    return wrapper

@singleton
class Cls(object):
    def __init__(self):
        pass
cls1 = Cls()
cls2 = Cls()
print(id(cls1) == id(cls2))
```

3. import 方法

- Python 的模块就是天然的单例模式

工厂模式

- 工厂模式是说调用方可以通过调用一个简单函数就可以创建不同的对象。工厂模式一般包含工厂方法和抽象工厂两种模式。

数据结构与算法篇

- 主要参考
  - problem-solving-with-algorithms-and-data-structure-using-python 中文版
  - 五分钟算法公众号
- 常见数据结构戳这里

排序算法

- 快排，堆排，归并排序详细原理参考这篇这或许是东半球分析十大排序算法最好的一篇文章，下面我将用 Python 快速实现出来。话不多说，Show Me Code!
- 冒泡排序

```
def bubble_sort(alist):
    for i in range(len(alist)-1):
        for j in range(len(alist)-1-i):
            if alist[j]>alist[j+1]:
                alist[j],alist[j+1]=alist[j+1],alist[j]
alist=[54,26,93,17,77,31,44,55,20]
bubble_sort(alist)
print(alist)
```

- 选择排序

```
def select_sort(alist):
    for i in range(len(alist)-1):
        min_index=i
        for j in range(i+1,len(alist)):
            if alist[j]<alist[min_index]:
                min_index=j
        if i!=min_index:
            alist[i],alist[min_index]=alist[min_index],alist[i]
alist=[54,26,93,17,77,31,44,55,20]
select_sort(alist)
print(alist)
```

- 插入排序

```
def insert_sort(alist):
    for i in range(1,len(alist)):
        for j in range(i,0,-1):
            if alist[j]<alist[j-1]:
                alist[j],alist[j-1]=alist[j-1],alist[j]
            else:
                break
alist=[54,26,93,17,77,31,44,55,20]
insert_sort(alist)
print(alist)
```

- 希尔排序

```
# 变换步长（增量）的插入排序->(n ~ n^2,不稳定)
def shell_sort(alist):
    gap=len(alist)//2
    while gap>=1:
        for i in range(gap,len(alist)):
            j=i
            while(j-gap)>=0:
                if alist[j]<alist[j-gap]:
                    alist[j],alist[j-gap]=alist[j-gap],alist[j]
                    j-=gap
                else:
                    break
        gap//=2
alist=[54,26,93,17,77,31,44,55,20]
shell_sort(alist)
print(alist)
```

- 归并排序

```
#分而治之，递归分解/递归合并 （nlogn）
def merge_sort(alist):
    if len(alist)<=1:
        return alist
    mid = len(alist)//2
    left_alist = merge_sort(alist[:mid])
    right_alist = merge_sort(alist[mid:])
    return merge(left_alist,right_alist)
def merge(left_list,right_list):
    result=[]
    j=0
    i=0
    while i<len(left_list) and j < len(right_list):
        if left_list[i]<=right_list[j]:
            result.append(left_list[i])
            i+=1
        else:
            result.append(right_list[j])
            j+=1
    result+=left_list[i:]
    result+=right_list[j:]
    return result
alist=[54,26,93,17,77,31,44,55,20]
print(merge_sort(alist))
```

- 快速排序

```
# 快速排序使用分而治之来获得与归并排序相同的优点，而不使用额外的存储。
# 有可能列表不能被分成两半。当这种情况发生时，性能降低
def quick_sort(alist, start, end):
    if start>= end:
        return
    #基准
    mark = alist[start]
    left = start
    right = end
    while left < right:
        while left<right  and alist[right]>mark:
            right-=1
        alist[left]=alist[right]
        while left<right and alist[left]<mark:
            left+=1
        alist[right]=alist[left]
    alist[right]=mark
    quick_sort(alist,start,left-1)
    quick_sort(alist,left+1,end)
alist=[54,26,93,17,77,31,44,55,20]
quick_sort(alist,0,len(alist)-1)
print(alist)
```

链表算法

- 输出/删除 单链表倒数第 K 个节点

```
# leetcode 19：快慢双指针
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        second = fist = head
        for i in range(n):  # 第一个指针先走 n 步
            fist = fist.next

        if fist == None:  # 如果现在第一个指针已经到头了，那么第一个结点就是要删除的结点。
            return second.next

        while fist.next:  # 然后同时走，直到第一个指针走到头
            fist = fist.next
            second = second.next
        second.next = second.next.next  # 删除相应的结点
        return head
```

- 有序链表合并

```
# leetcode 21
# 解题思路：合并后的链表仍然是有序的，可以同时遍历两个链表，
# 每次选取两个链表中较小值的节点，依次连接起来，就能得到最终的链表
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        if not l1 or not l2:
            return l1 or l2
        head=cur=ListNode(0)
        while l1 and l2:
            if l1.val<l2.val:
                cur.next=l1
                l1=l1.next
            else:
                cur.next=l2
                l2=l2.next
            cur=cur.next
        cur.next=l1 or l2
        return head.next
```

- 链表有环问题

```
# leetcode 142：判断链表是否有环/定位环入口/环长度，快慢双指针
    class Solution(object):
    def detectCycle(self, head):
        if head is None:
            return
        slow=fast=head
        has_cycle=False
        while fast and fast.next:
            slow=slow.next
            fast=fast.next.next
            if slow==fast:
                has_cycle=True
                break
        if has_cycle:
            slow_p=head
            fast_p=fast
            while slow_p !=fast_p:
                fast_p=fast_P.next
                slow_p=slow_p.next
            return slow_p
        return
        # 环长度
        slow=slow.next
        fast=fast.next.next
        length=1
        while slow!=fast:
            slow=slow.next
            fast=fast.next.next
            length+=1
        return length
```

- 使用链表实现大数加法

```
# leetcode 2
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        flag=0
        root=n=ListNode(0)
        while l1 or l2 or flag:
            v1=v2=0
            if l1:
                v1=l1.val
                l1=l1.next
            if l2:
                v2=l2.val
                l2=l2.next
            flag,val=divmod(v1+v2+flag,10)
            n.next=ListNode(val)
            n=n.next
        return root.next
```

- 删除链表中节点，要求时间复杂度为O(1)

```
# leetcode 237
class Solution:
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        next_node=node.next
        next_nextnode=next_node.next
        node.val=next_node.val
        node.next=next_nextnode
```

- 从尾到头打印链表

```
# 根据栈的特性
class Solution:
    def printListFromTailToHead(self, listNode):
        if listNode is None:
            return []
        sta=list()
        res=list()
        while listNode:
            sta.append(listNode.val)
            listNode=listNode.next
        while sta:
            res.append(sta.pop())
        return res
# 递归
class Solution:
    def printListFromTailToHead(self, listNode):
        if listNode is None:
            return []
        return self.printListFromTailToHead(listNode.next)+[listNode.val]
```

- 反转链表

```
# leetcode 206
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre=None
        # 不断取出和向后移动头节点,并将头节点连接到新头节点后面
        while head:
            next_node=head.next
            head.next=pre
            pre=head
            head=next_node
        return pre
```

- LRU缓存机制

```python
# leetcode 146:
# 字典（哈希）+双端链表
class Node(object):
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.next = None
        self.prev = None

class LRUCache(object):

    def __init__(self, capacity):
        self.dic = {}
        self.capacity = capacity
        self.dummy_head = Node(0, 0)
        self.dummy_tail = Node(0, 0)
        self.dummy_head.next = self.dummy_tail
        self.dummy_tail.prev = self.dummy_head

    def get(self, key):
        if key not in self.dic:
            return -1
        node = self.dic[key]
        self.remove(node)
        self.append(node)
        return node.val

    def put(self, key, value):
        if key in self.dic:
            self.remove(self.dic[key])
        node = Node(key, value)
        self.append(node)
        self.dic[key] = node

        if len(self.dic) > self.capacity:
            head = self.dummy_head.next
            self.remove(head)
            del self.dic[head.key]

    def append(self, node):
        tail = self.dummy_tail.prev
        tail.next = node
        node.prev = tail
        self.dummy_tail.prev = node
        node.next = self.dummy_tail

    def remove(self, node):
        prev = node.prev
        next = node.next
        prev.next = next
        next.prev = prev
```

栈和队列算法

- 以下几个算法原理详解
- 有效的括号

```python
# leetcode 20
class Solution:
    def isValid(self, s: str) -> bool:
        chars={'(':')','[':']','{':'}'}
        stack=[]
        for i in s:
            if i in chars:
                stack.append(i)
            else:
                if not stack or chars[stack.pop()]!=i:
                    return False
        return not stack
```

- 用两个栈实现队列

```
# leetcode 232: 一个栈作为缓存区
class MyQueue:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.stack1=[]
        self.stack2=[]


    def push(self, x: int) -> None:
        """
        Push element x to the back of queue.
        """
        self.stack1.append(x)


    def pop(self) -> int:
        """
        Removes the element from in front of queue and returns that element.
        """
        if len(self.stack2)==0:
            while(self.stack1):
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop()

    def peek(self) -> int:
        """
        Get the front element.
        """
        if len(self.stack2)==0:
            while(self.stack1):
                self.stack2.append(self.stack1.pop())
        return self.stack2[-1]

    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        return not self.stack1 and not self.stack2
```

- 栈的压入、弹出序列

```
# leetcode 946:借用一个辅助的栈tmp，遍历压栈顺序
class Solution:
    def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
        tmp=[]
        while pushed:
            tmp.append(pushed.pop(0))
            while tmp and tmp[-1]==popped[0]:
                popped.pop(0)
                tmp.pop()
        return not tmp
```

- 包含 min 函数的栈

```
# leetcode 155：实际上就是实现最小栈（重点是连续弹出最小值时的问题）
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack=[]
        self.minstack=[]

    def push(self, x: int) -> None:
        self.stack.append(x)
        if self.minstack:
            self.minstack.append(x)
        else:
            if x<self.minstack[-1]:
                self.minstack.append(x)
            else:
                self.minstack.append(self.minstack[-1])

    def pop(self) -> None:
        self.stack.pop()
        self.minstack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.minstack[-1]
```

- 验证栈序列

```
# leetcode 946
class Solution:
    def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
        tmp=[]
        while pushed:
            tmp.append(pushed.pop(0))
            while tmp and tmp[-1]==popped[0]:
                popped.pop(0)
                tmp.pop()
        return not tmp
```

- 约瑟夫环（双端队列实现）

```
from pythonds.basic.queue import Queue
def hotPotato(namelist,num):
    simple_queue = Queue()
    for name in namelist:
        simple_queue.enqueue(name)
    while simple_queue.size()>1:
        for i in range(num):
            simple_queue.enqueue(simple_queue.dequeue())
        simple_queue.dequeue()
    return simple_queue.dequeue()

print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],2))
```

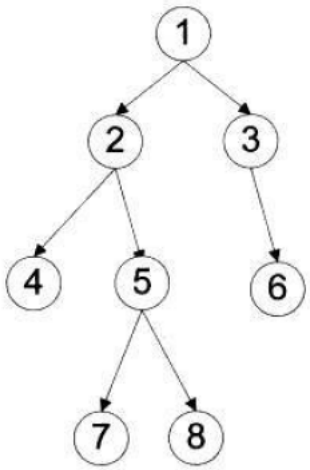
- 合并K个排序链表（我刚开始用的暴力解法2333）
    - 这个解析不错，我直接贴了其中采用最优队列的方式。

```python
# leetcode 23
from Queue import PriorityQueue
class Solution(object):
    def mergeKLists(self, lists):
        head = point = ListNode(0)
        q = PriorityQueue()
        for l in lists:
            if l:
                q.put((l.val, l))
        while not q.empty():
            val, node = q.get()
            point.next = ListNode(val)
            point = point.next
            node = node.next
            if node:
                q.put((node.val, node))
        return head.next
```

二叉树算法

四种遍历方式

- 二叉树的四种遍历方式分别是：前序、中序、后序和层次。它们的时间复杂度都是O(n)，其中n是树中节点个数，因为每一个节点在递归的过程中，只访问了一次。
- 三种深度优先遍历方法的空间复杂度是O(h)，其中h是二叉树的深度，额外空间是函数递归的调用栈产生的，而不是显示的额外变量。 空间复杂度，通常是指完成算法所用的辅助空间的复杂度，而不用管算法前置的空间复杂度。比如在树的遍历算法中，整棵树肯定要占O(n)的空间，n是树中节点的个数，
- 层次遍历的空间复杂度是O(w)，其中w是二叉树的宽度（拥有最多节点的层的节点数）。



前序遍历: 1 2 4 5 7 8 3 6

中序遍历: 4 2 7 5 8 1 3 6

后序遍历: 4 7 8 5 2 6 3 1

层次遍历: 1 2 3 4 5 6 7 8

- 前序遍历

```
# 递归解法
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        ret=[]
        if root:
            ret=ret+[root.val]
            ret=ret+self.preorderTraversal(root.left)
            ret=ret+self.preorderTraversal(root.right)
        return ret
# 迭代算法思路：使用栈的思想，从根节点开始以此使用ret添加根节点值，stack添加右节点，
# curr=左节点，如果左节点为None，则获取其上一个右节点（一直输出根节点，添加其右节点，
# 遍历左节点，右节点的输出顺序为从下往上）
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        ret=[]
        if root==None:
            return ret
        stack=[]
        curr=root
        while curr or stack:
            if curr:
                ret.append(curr.val)
                stack.append(curr.right)
                curr=curr.left
            else:
                curr=stack.pop()
        return ret
```

- 中序遍历

```
# 递归解法同上
# 迭代解法
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        res=[]
        if root==None:
            return res
        stack=[]  #添加根节点
        curr=root
        while curr or stack:
            if curr:
                stack.append(curr)
                curr=curr.left
            else:
                curr=stack.pop()
                res.append(curr.val)
                curr=curr.right
        return res
```

- 后序遍历

```
# 递归解法同上
# 迭代算法思路：后序遍历方式为：左右中，将其进行反转 中右左，
# 那么我们可以实现一个中右左，其原理与前序遍历一样
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        ret=[]
        if root==None:
            return ret
        stack=[]
        curr=root
        while curr or stack:
            if curr:
                ret.append(curr.val)
                stack.append(curr.left)
                curr=curr.right
            else:
                curr=stack.pop()
        return ret[::-1]
```

- 层次遍历

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        level=[root]
        ret=[]
        if root==None:
            return []
        while level:
            ret.append([i.val for i in level])
            level=[kid for node in level for kid in (node.left,node.right) if kid]
        return ret
```

二叉树中的一些重要属性

- 二叉树的最小深度

```
class Solution:
    def minDepth(self, root: TreeNode) -> int:
        if not root:
            return 0
        if root.left==None or root.right==None:
            return self.minDepth(root.left)+self.minDepth(root.right)+1
        else:
            return min(map(self.minDepth,(root.left,root.right)))+1
```

- 二叉树的层平均值

```
class Solution:
    def averageOfLevels(self, root: TreeNode) -> List[float]:
        average=[]
        if root:
            level=[root]
            while level:
                average.append(sum(i.val for i in level)/len(level))
                level=[kid for node in level for kid in (node.left,node.right) if kid]
        return average
```

- 二叉树的直径
  - 采用分治和递归的思想：根节点为root的二叉树的直径 = max(root-left的直径, root->right的直径，root->left的最大深度+root->right的最大深度+1)
  - 分两种情况，1，最大直径经过根节点，则直径为左子树最大深度+右子树最大深度 2.如果不经过根节点，则取左子树或右子树的最大深度

```
class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.diameter=0
        def dfs(root):
            if root==None:
                return 0
            left=dfs(root.left)
            right=dfs(root.right)
            self.diameter=max(self.diameter,left+right)
            return max(left,right)+1
        dfs(root)
        return self.diameter
```

- 两个二叉树的最低公共祖先节点

```
'''
思路：递归，如果当前节点就是p或q，说明当前节点就是最近的祖先；如果当前节点不是p或p，
就试着从左右子树里找pq；如果pq分别在一左一右被找到，那么当前节点还是最近的
祖先返回root就好了；否则，返回它们都在的那一边。
'''
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if root==None:
            return
        if root==p or root==q:
            return root
        left=self.lowestCommonAncestor(root.left,p,q)
        right=self.lowestCommonAncestor(root.right,p,q)
        if left and right:
            return root
        if left:
            return left
        if right:
            return right
        else:
            return
```

字符串算法

- 最长公共前缀

```
# leetcode 14
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        temp_str=""
        zip_obj = zip(*strs)
        for obj in zip_obj:
            if(len(set(obj)))>1:
                break
            temp_str+=obj[0]
        return temp_str
```

- 最长回文子串

```
# leetcode 5
class Solution:
    def longestPalindrome(self, s: str) -> str:
        start=0
        maxstr=0
        for i in range(len(s)):
            if i-maxstr>=1 and s[i-maxstr-1:i+1]==s[i-maxstr-1:i+1][::-1]:
                start = i-maxstr-1
                maxstr+=2
                continue
            if  i-maxstr>=0 and s[i-maxstr:i+1]==s[i-maxstr:i+1][::-1]:
                start = i-maxstr
                maxstr+=1
        return s[start:start+maxstr]
```

- 无重复字符的最长子串

```
# leetcode 3 : 滑动窗口法
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        max_str=0
        for i in range(len(s)):
            if len(s[i-max_str:i+1])==len(set(s[i-max_str:i+1])):
                max_str+=1
        return max_str
```

哈希算法

- 哈希函数特性：确定性、哈希冲突，不可逆性，混淆特性
  - 哈希表的 load factor（负载因子），越大表明哈希表中填的元素越多，越容易发生冲突
  - 解决哈希冲突：开放寻址法（线性探测、二次探测、双重哈希），链表法
  - 哈希洪水攻击

- 实现哈希表

```
class HashMap:
    def __init__(self):
        self.size = 11
        self.slots = [None]*self.size
        self.data = [None]*self.size

    def put(self, key, data):
        hash_value = self.hash(key, len(self.slots))
        if self.slots[hash_value] == None:
            self.slots[hash_value] = key
            self.data[hash_value] = data
        else:
            # 键一样时替换
            if self.slots[hash_value] == key:
                self.data[hash_value] = data
            else:
                # 重新散列，+1线性探测
                next_slot = self.rehash(hash_value, len(self.slots))
                while self.slots[next_slot] != None and self.slots[next_slot] != key:
                    next_slot = self.rehash(next_slot, len(self.slots))
                if self.slots[next_slot] != None:
                    self.slots[next_slot] = key
                    self.data[next_slot] = data
                else:
                    self.data[next_slot] = data

    def get(self, key):
        start_slot = self.hash(key, len(self.slots))
        found = False
        data = None
        stop = False
        position = start_slot
        while self.slots[position] != None and not stop and not found:
            if self.slots[position] == key:
                data = self.data[position]
                found = True
            else:
                position = self.rehash(position,len(self.slots))
                if position==start_slot:
                    stop = True
        return data

    def hash(self, key, size):
        return key % size

    def rehash(self, oldhash, size):
        return (oldhash+1) % size

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, data):
        return self.put(key, data)
hashmap=HashMap()
hashmap[0]='cat'
hashmap[1]='dog'
hashmap[2]='bird'
print(hashmap.slots)
print(hashmap.data  )
```

- 两数之和

```
# leetcode 1
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        #enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)
        # 组合为一个索引序列，同时列出数据和数据下标
        demo_dict = dict((num,i)for i,num in enumerate(nums))
        for i,num in enumerate(nums):
            #get()方法和[key]方法的主要区别在于[key]在遇到不存在的key时会抛出KeyError错
            j=demo_dict.get(target-num)
            if j and i!=j:
                return [i,j]
```

- 三数之和

```
# leetcode 15
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        res = []
        # 排序后遍历
        for t in range(len(nums)- 2):
            # 跳过相同的情况
            if t > 0 and nums[t] == nums[t - 1]:
                    continue
            # i向后判断，j向前判断
            i, j = t + 1, len(nums) - 1
            while i < j:
                _sum = nums[t] + nums[i] + nums[j]
                if _sum == 0:
                    res.append([nums[t], nums[i], nums[j]])
                    i += 1
                    j -= 1
                    # 跳过相同的情况
                    while i < j and nums[i] == nums[i - 1]:
                        i += 1
                    while i < j and nums[j] == nums[j + 1]:
                        j -= 1
                elif _sum < 0:
                    i += 1
                else:
                    j -= 1
        return res
```

- 重复的 DNA 序列

```
# leetcode 187
class Solution:
    def findRepeatedDnaSequences(self, s: str) -> List[str]:
        res=dict()
        for i in range(len(s)-9):
            res[s[i:i+10]]=res.get(s[i:i+10],0)+1
        return [k for k,v in res.items() if v>=2]
```

- 两个数组的交集

```
# leetcode 350
class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
        res=[]
        num_dict={}
        for num in nums1:
            if num not in num_dict:
                num_dict[num]=1
            else:
                num_dict[num]+=1
        for num in nums2:
            if num in num_dict and num_dict[num]>0:
                num_dict[num]-=1
                res.append(num)
        return res
```

常见算法基本思想

回溯

- 主要参考
  - Backtracking回溯法(又称DFS,递归)全解
  - leetcode 22 / leetcode 39 / leetcode 40 /leetcode 46 / leetcode 79

```
# leetcode 17
class Solution(object):
    def letterCombinations(self, digits):
        """
        :type digits: str
        :rtype: List[str]
        """
        if not digits:
            return []
        d = {'2':"abc",
             '3':"def",
             '4':"ghi",
             '5':"jkl",
             '6':"mno",
             '7':"pqrs",
             '8':"tuv",
             '9':"wxyz"
            }
        res = []
        self.dfs(d, digits, "", res)
        return res

    def dfs(self, d, digits, tmp , res):
        if not digits:
            res.append(tmp)
        else:
            for num in d[digits[0]]:
                self.dfs(d, digits[1:], tmp + num, res)
```

分而治之

- leetcode 4 / leetcode 215 / leetcode 241

```
# leetcode 215
'''
任意选定数组内一个数mark，将比其大和比其小的值分别放在左右两个数组里，接着判断两边数字的数量，如果左边的数量大于k个，说明第k大的数字存在于mark及mark左边的区域，对左半区执行partition函数；如果左边的数量小于k个，说明第k大的数字在mark和mark右边的区域之内，对右半区执行partition函数。直到左半区刚好有k-1个数，那么第k大的数就已经找到了。
'''
class Solution(object):
    def findKthLargest(self, nums, k):
        mark = random.choice(nums)
        num1,num2=list(),list()
        for num in nums:
            if num>mark:
                num1.append(num)
            if num<mark:
                num2.append(num)
        if k<=len(num1):
            return self.findKthLargest(num1,k)
        if k>len(nums)-len(num2):
            return self.findKthLargest(num2,k-(len(nums)-len(num2)))
        return mark
```

动态规划

- 硬币找零

```
# leetcode 322
class Solution(object):
    def coinChange(self, coins, amount):
    # 建一个长度是 amount + 1的dp数组，构成面额从0到 amount需要使用的最少硬币数量。
        dp = [0] + [-1] * amount
        for x in range(amount):
            if dp[x] < 0:
                continue
            for c in coins:
                if x + c > amount:
                    continue
                if dp[x + c] < 0 or dp[x + c] > dp[x] + 1:
                    dp[x + c] = dp[x] + 1
        return dp[amount]
```

分支界限

操作系统篇

- 学习资料推荐
  - 哈工大李治军老师的OS
  - 配套的实验环境
  - gitbook和源码
  - Linux内核完全注释

进程与线程区别

- 进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；线程是进程的子任务，是CPU调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发。
- 一个程序至少有一个进程，一个进程至少有一个线程，线程依赖于进程而存在；进程在执行过程中拥有独立的内存单元，而多个线程线程是共同享有进程占有的资源和地址空间的(*线程共享的环境包括：进程代码段、进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID*)。**进程让操作系统的并发性成为可能，而线程让进程的内部并发成为可能。**

谈谈多线程并发

- 将程序的执行逻辑与调度机制的细节，交替执行的操作，异步 I/O 以及资源等待等问题分离开来。通过使用线程，可以将复杂并且异步的工作流进一步分解为一组简单并且同步的工作流，每个工作流在一个单独的线程中运行，并在特定的同步位置进行交互。
- 提高资源利用率，多处理器系统的出现，使得同一个程序的多个线程可以被同时调度到多个 CPU 上运行。因此，多线程程序可以通过提高处理器资源的利用率来提升系统的吞吐率。当然多线程程序也有助于在单处理器系统上获得更高的吞吐率（如果程序的一个线程在等待 I/O 操作）
- 多线程的性能不一定优于单线程，对于单核CPU，如果是 CPU密集型任务，如解压文件，多线程的性能反而不如单线程性能，因为解压文件需要一直占用 CPU资源，如果采用多线程，线程切换导致的开销反而会让性能下降。但是对于 IO密集型任务，肯定是需要使用多线程的。而对于多核CPU，对于解压文件来说，多线程肯定优于单线程，因为多个线程能够更加充分利用每个核的资源。
- 使用 queue 进行线程通信。当线程之间如果要共享资源或数据的时候，可能变的非常复杂。Python的threading模块提供了很多同步原语，包括信号量，条件变量，事件和锁。如果可以使用这些原语的话，应该优先考虑使用这些，而不是使用queue（队列）模块。队列操作起来更容易，也使多线程编程更安全，因为队列可以将资源的使用通过单线程进行完全控制，并且允许使用更加整洁和可读性更高的设计模式。

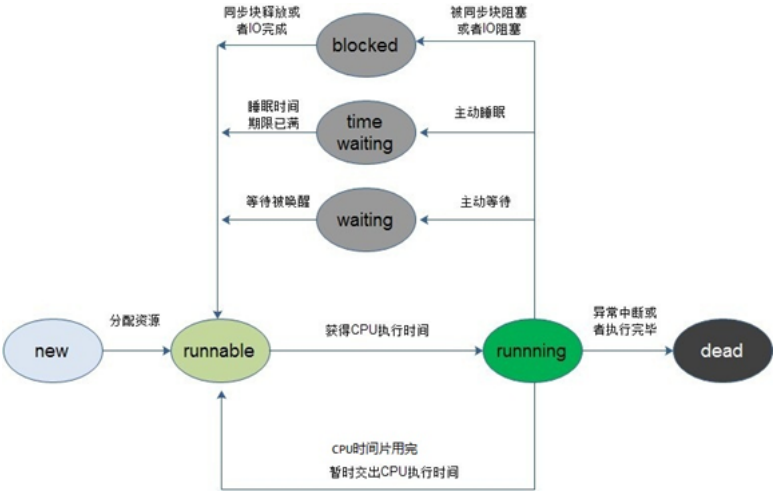
```
import threading
import time
class GetHtml(threading.Thread):
    def __init__(self,name):
        super().__init__(name=name)
    def run(self):
        print('get html start')
        time.sleep(2)
        print('get html end')
class GetUrl(threading.Thread):
    def __init__(self,name):
        super().__init__(name=name)
    def run(self):
        print('get url start')
        time.sleep(4)
        print('get url end')
if __name__ == "__main__":
    thread1=GetHtml(name='html')
    thread2=GetUrl(name='url')
    start_time=time.time()
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    end_time=time.time()
    total_time=end_time-start_time
    print('total running tim:{}'.format(total_time))
```

线程同步方式(Python代码实现)

- 主要参考
  - 四种线程同步（或互斥）方式小结
- 线程同步就是协同步调，按预定的先后次序进行运行。如：你说完，我再说。这里的同步千万不要理解成那个同时进行，应是指协同、协助、互相配合。线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步）也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间是各自运行各自的！

- 线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步（下文统称为同步）。
  - **互斥量 Mutex**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问，当前拥有互斥对象的线程处理完任务后必须将互斥对象交出，以便其他线程访问该资源。(RLock 和 Lock)
  - **信号量 Semaphare**：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1 ，只要当前可用资源计数是大于0 的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离 开的同时通过Release将当前可用资源计数加1，在任何时候当前可用资源计数决不可能大于最大资源计数。
  - **事件对象 Event**：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。事件是内核对象，事件分为手动置位事件和自动置位事件。事件Event内部它包含一个使用计数（所有内核对象都有），一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。

线程状态切换



分页机制

- 主要参考
  - 分页机制图文详解
- 一级页表的弊端：
  - 每存取一个数据，需两次访问内存，第一次访问内存中的页表，第二次访问内存中的数据，效率较低。改进：增设一个具有并行查寻能力的特殊高速缓冲寄存器，称为“联想寄存器”或“快表”，IBM中称为TLB，用于存放当前访问的那些页表项。
  - IA-32体系结构中，处理器为32位，可寻址 2^32 =4GB的虚拟地址空间，若每页大小为4KB，则共分为4GB/4KB= 2^20 =1048576页，因此页表中应有1048576项，每个页表项为4B，则一个页表需要4MB的连续的物理内存，每个进程都需要自身的页表占4MB，将导致大量内存用于保存进程的页表。
  - 改进：采用两级页表，每页中存2^10 项，共分为2^10 页，并新增一个页目录表来记录这 2^10 页表的地址与信息，因此页目录表大小为 2^10*4B=4KB放在内存中，需要具体的表再由此读入。采用离散分配方式代替原来页表需要的连续物理内存，将当前需要的部分页表项调入内存，其余页表项仍驻留在磁盘上，需要时再调入。

分页和分段有什么区别（内存管理）

- 主要参考
  - 分段和分页内存管理
- 段式存储管理是一种符合用户视角的内存分配管理方案。在段式存储管理中，将程序的地址空间划分为若干段（segment），如代码段，数据段，堆栈段；这样每个进程有一个二维地址空间，相互独立，互不干扰。段式管理的优点是：没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片（比如 4k 的段换5k 的段，会产生 1k 的外碎片）
- 页式存储管理方案是一种用户视角内存与物理内存相分离的内存分配管理方案。在页式存储管理中，将程序的逻辑地址划分为固定大小的页（page），而物理内存划分为同样大小的帧，程序加载时，可以将任意一页放入内存中任意一个帧，这些帧不必连续，从而实现了离散分离。页式存储管理的优点是：没有外碎片（因为页的大小固定），但会产生内碎片（一个页可能填充不满）
- 两者的不同点：
  - 目的不同：分页是由于系统管理的需要而不是用户的需要，它是信息的物理单位；分段的目的是为了能更好地满足用户的需要，它是信息的逻辑单位，它含有一组其意义相对完整的信息；
  - 大小不同：页的大小固定且由系统决定，而段的长度却不固定，由其所完成的功能决定；
  - 地址空间不同： 段向用户提供二维地址空间；页向用户提供的是一维地址空间；
  - 信息共享：段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制；

- 内存碎片：页式存储管理的优点是没有外碎片（因为页的大小固定），但会产生内碎片（一个页可能填充不满）；而段式管理的优点是没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片（比如 4k 的段换 5k 的段，会产生1k 的外碎片）。

什么是虚拟内存？

- 主要参考
  - BAT面试之操作系统内存详解
- 内存的发展历程：没有内存抽象 (单进程，除去操作系统所用的内存之外，全部给用户程序使用) —> 有内存抽象（多进程，进程独立的地址空间，交换技术 (内存大小不可能容纳下所有并发执行的进程)）—> 连续内存分配 (固定大小分区 (多道程序的程度受限)，可变分区 (首次适应，最佳适应，最差适应)，碎片) —> 不连续内存分配（分段，分页，段页式，虚拟内存）。
- 虚拟内存允许执行进程不必完全在内存中。虚拟内存的基本思想是：每个进程拥有独立的地址空间，这个空间被分为大小相等的多个块，称为页 (Page)，每个页都是一段连续的地址。这些页被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻进行必要的映射；当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的命令。这样，对于进程而言，逻辑上似乎有很大的内存空间，实际上其中一部分对应物理内存上的一块 (称为帧，通常页和帧大小相等)，还有一些没加载在内存中的对应在硬盘上。注意，请求分页系统、请求分段系统和请求段页式系统都是针对虚拟内存的，通过请求实现内存与外存的信息置换。
- 局部性原理
  - 时间上的局部性：最近被访问的页在不久的将来还会被访问；
  - 空间上的局部性：内存中被访问的页周围的页也很可能被访问。
- 页面置换算法
  - FIFO 先进先出算法：在操作系统中经常被用到，比如作业调度（主要实现简单，很容易想到）；
  - LRU（Least recently use）最近最少使用算法：根据使用时间到现在的长短来判断；
  - LFU（Least frequently use）最少使用次数算法：根据使用次数来判断；
  - OPT（Optimal replacement）最优置换算法：理论的最优，理论；就是要保证置换出去的是不再被使用的页，或者是在实际内存中最晚使用的算法。
- 虚拟内存的应用与优点：虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待它的一部分读入内存时，可以把 CPU 交给另一个进程使用。虚拟内存的使用可以带来以下好处：
  - 在内存中可以保留多个进程，系统并发度提高
  - 解除了用户与内存之间的紧密约束，进程可以比内存的全部空间还大

颠簸(抖动)

- 颠簸本质上是指频繁的页调度行为，具体来讲，进程发生缺页中断，这时，必须置换某一页。然而，其他所有的页都在使用，它置换一个页，但又立刻再次需要这个页。因此，会不断产生缺页中断，导致整个系统的效率急剧下降，这种现象称为颠簸（抖动）。
- 内存颠簸的解决策略包括：
  - 如果是因为页面替换策略失误，可以修改替换算法来解决这个问题；
  - 如果是因为运行的程序太多，造成程序无法同时将所有频繁访问的页面调入内存，则要降低多道程序的数量；

进程调度算法

- FCFS(先来先服务，队列实现，非抢占的)：先请求CPU的进程先分配到CPU
- SJF(最短作业优先调度算法)：平均等待时间最短，但难以知道下一个CPU区间长度
- 优先级调度算法(可以是抢占的，也可以是非抢占的)：优先级越高越先分配到CPU，相同优先级先到先服务，存在的主要问题是：低优先级进程无穷等待CPU，会导致无穷阻塞或饥饿；
- 时间片轮转调度算法(可抢占的)：队列中没有进程被分配超过一个时间片的CPU时间，除非它是唯一可运行的进程。如果进程的CPU区间超过了一个时间片，那么该进程就被抢占并放回就绪队列。
- 多级队列调度算法：将就绪队列分成多个独立的队列，每个队列都有自己的调度算法，队列之间采用固定优先级抢占调度。其中，一个进程根据自身属性被永久地分配到一个队列中。
- 多级反馈队列调度算法：与多级队列调度算法相比，其允许进程在队列之间移动：若进程使用过多CPU时间，那么它会被转移到更低的优先级队列；在较低优先级队列等待时间过长的进程会被转移到更高优先级队列，以防止饥饿发生。

经典进程同步问题

- 主要参考
  - 进程同步的基本概念
- 信号量（PV操作）、管程

进程的状态转换

- 主要参考
  - 操作系统之进程的状态和转换详解
- 进程的三种状态情况：运行态（占用CPU）;就绪态（可运行，但暂时没有CPU分配给它）；阻塞态(除非某种外部条件发生，就算CPU空闲，也不能执行)。unix用block阻塞。状态转换表

- 就绪状态 ：一个进程获得了除处理机外的一切所需资源，一旦得到处理机即可运行，则称此进程处于就绪状态。
- 执行状态：当一个进程在处理机上运行时，则称该进程处于运行状态。
- 阻塞状态：一个进程正在等待某一事件发生（例如请求I / O而等待I / O完成等）而暂时停止运行，这时即使把处理机分配给进程也无法运行，故称该进程处于阻塞状态。

进程间通信方式

- 主要参考
  - 进程间通信（IPC）介绍
- 进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。进程间通信（Inter-Process Communication）方式：管道（匿名管道/命名管道），消息队列，**共享内存、信号量、套接字（socket），后三种较常用**
  - **管道**一般指匿名管道，是 UNIX 系统IPC最古老的形式,只能用于具有亲缘关系的进程之间的通信（父子进程或者兄弟进程之间），是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存。**命名管道（FIFO）**除了具有匿名管道所具有的功能外，它还允许无亲缘关系进程间的通信。
  - **消息队列**是消息的链接表，它克服了上两种通信方式中信号量有限的缺点，具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息；消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。
  - **共享内存**可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等；共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。特点:共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。因为多个进程可以同时操作，所以需要进行同步。信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。
  - **信号量**主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段；信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。特点：信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。支持信号量组。
  - **套接字**，这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

僵尸进程和孤儿进程

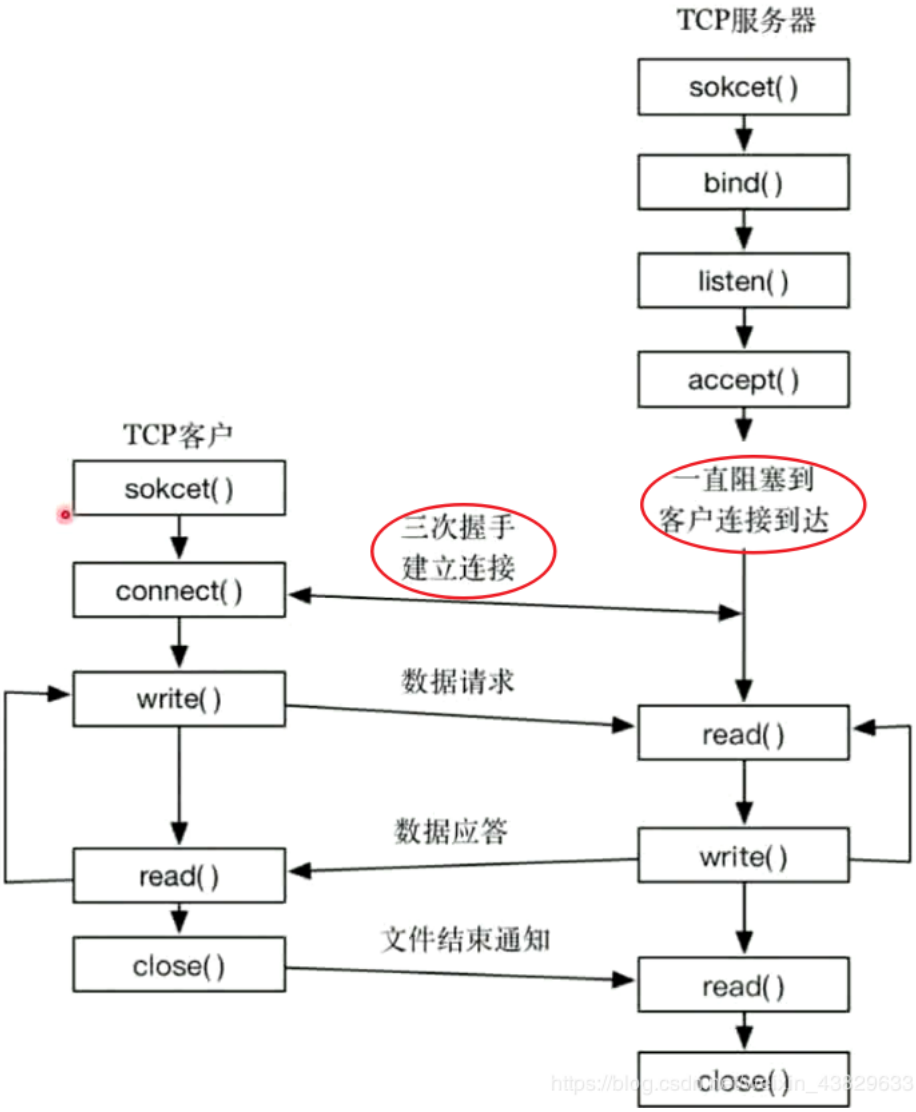
- 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

fork进程

- 由fork创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新进程（子进程）的进程 id。将子进程id返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的进程id。对子进程来说，之所以fork返回0给它，是因为它随时可以调用getpid()来获取自己的pid；也可以调用getppid()来获取父进程的id。(进程id 0总是由交换进程使用，所以一个子进程的进程id不可能为0)。
- fork之后，操作系统会复制一个与父进程完全相同的子进程，虽说是父子关系，但是在操作系统看来，他们更像兄弟关系，这2个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置。

Socket编程

- Socket Programming in Python (Guide)



Linux的五种IO模型

- 主要参考
  - 聊聊IO多路复用之select、poll、epoll详解
  - 漫话：如何给女朋友解释什么是Linux的五种IO模型
- 在Linux(UNIX)操作系统中，共有五种IO模型，分别是：阻塞IO模型、非阻塞IO模型、IO复用模型、信号驱动IO模型以及异步IO模型。 I/O操作，以一次磁盘文件读取为例，读取的文件是存储在磁盘上的，我们的目的是把它读取到内存中，可以把这个步骤简化成（真正的文件读取还涉及到缓存等细节）把数据从硬件（硬盘）中读取到用户空间中。关于用户空间、内核空间以及硬件等的关系如可以通过钓鱼的例子理解。鱼塘就可以映射成磁盘，中间过渡的鱼钩可以映射成内核空间，最终放鱼的鱼篓可以映射成用户空间。一次完整的钓鱼（IO）操作，是鱼（文件）从鱼塘（硬盘）中转移（拷贝）到鱼篓（用户空间）的过程。

1. **阻塞IO模型**。应用进程通过系统调用 recvfrom 接收数据，但由于内核还未准备好数据报，应用进程就会阻塞住，直到内核准备好数据报，recvfrom 完成数据报复制工作，应用进程才能结束阻塞状态。（并发低，时效性要求低）

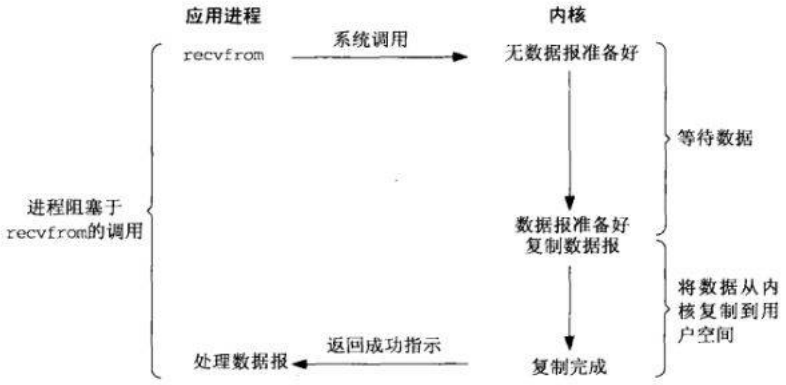


图6-1 阻塞式I/O模型

2. **非阻塞IO模型**。应用进程通过 recvfrom 调用不停的去和内核交互，直到内核准备好数据。如果没有准备好，内核会返回error，应用进程在得到error后，过一段时间再发送recvfrom请求。在两次发送请求的时间段，进程可以先做别的事情。

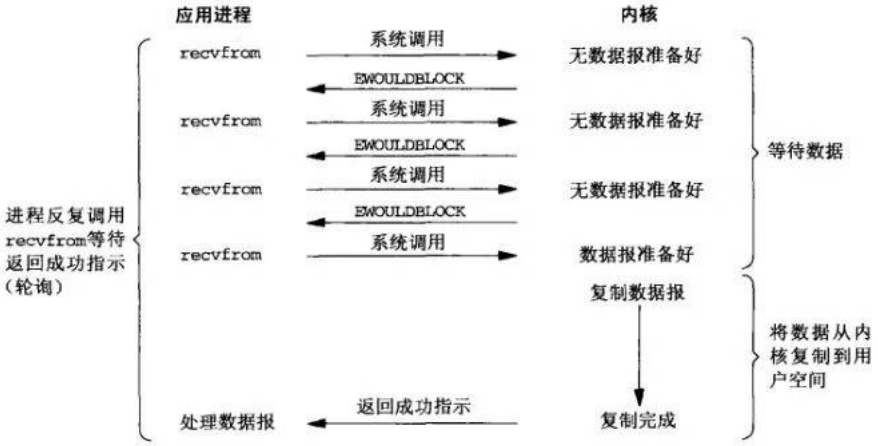


图6-2 非阻塞式I/O模型

3. **IO多路复用模型**。IO多路转接是多了一个select函数，多个进程的IO可以注册到同一个select上，当用户进程调用该select，select会监听所有注册好的IO，如果所有被监听的IO需要的数据都没有准备好时，select调用进程会阻塞。当任意一个IO所需的数据准备好之后，select调用就会返回，然后进程在通过recvfrom来进行数据拷贝。这里的IO复用模型，并没有向内核注册信号处理函数，所以，他并不是非阻塞的。进程在发出select后，要等到select监听的所有IO操作中至少有一个需要的数据准备好，才会有返回，并且也需要再次发送请求去进行文件的拷贝。PS: 这种方式的钓鱼，通过增加鱼竿的方式，可以有效的提升效率。

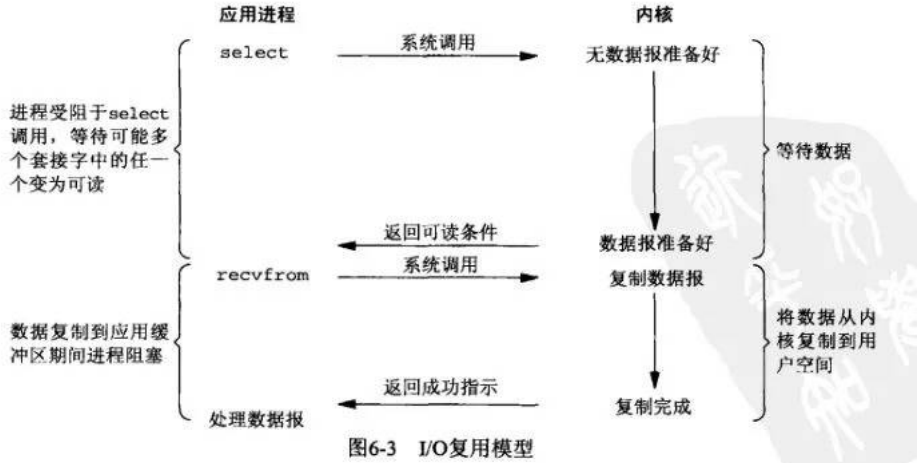


图6-3 I/O复用模型

4. **信号驱动IO模型**。应用进程预先向内核注册一个信号处理函数，然后用户进程返回，并且不阻塞，当内核数据准备就绪时会发送一个信号给进程，用户进程便在信号处理函数中开始把数据拷贝的用户空间中。PS: 这种方式钓鱼，和前几种相比，所使用的工具有了一些变化，需要有一些定制（实现复杂）。但是钓鱼的人就可以在鱼儿咬钩之前彻底做别的事儿去了。等着报警器响就行了。

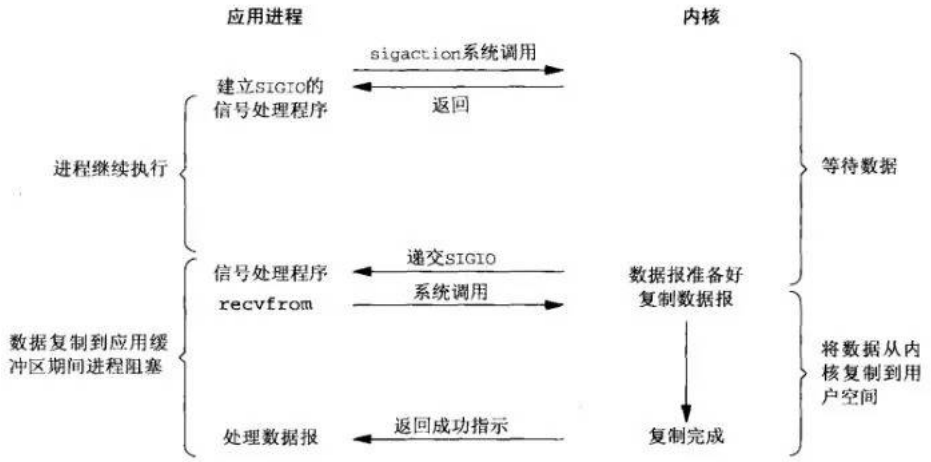


图6-4　信号驱动式I/O模型

5. **异步IO模型**。用户进程发起aio_read操作之后，给内核传递描述符、缓冲区指针、缓冲区大小等，告诉内核当整个操作完成时，如何通知进程，然后就立刻去做其他事情了。当内核收到aio_read后，会立刻返回，然后内核开始等待数据准备，数据准备好以后，直接把数据拷贝到用户控件，然后再通知进程本次IO已经完成。(应用进程把IO请求传给内核后，完全由内核去操作文件拷贝。内核完成相关操作后，会发信号告诉应用进程本次IO已经完成。)

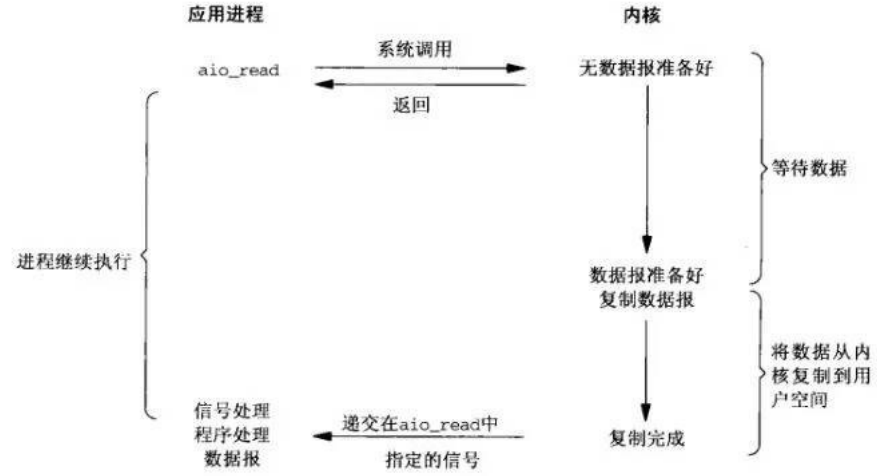
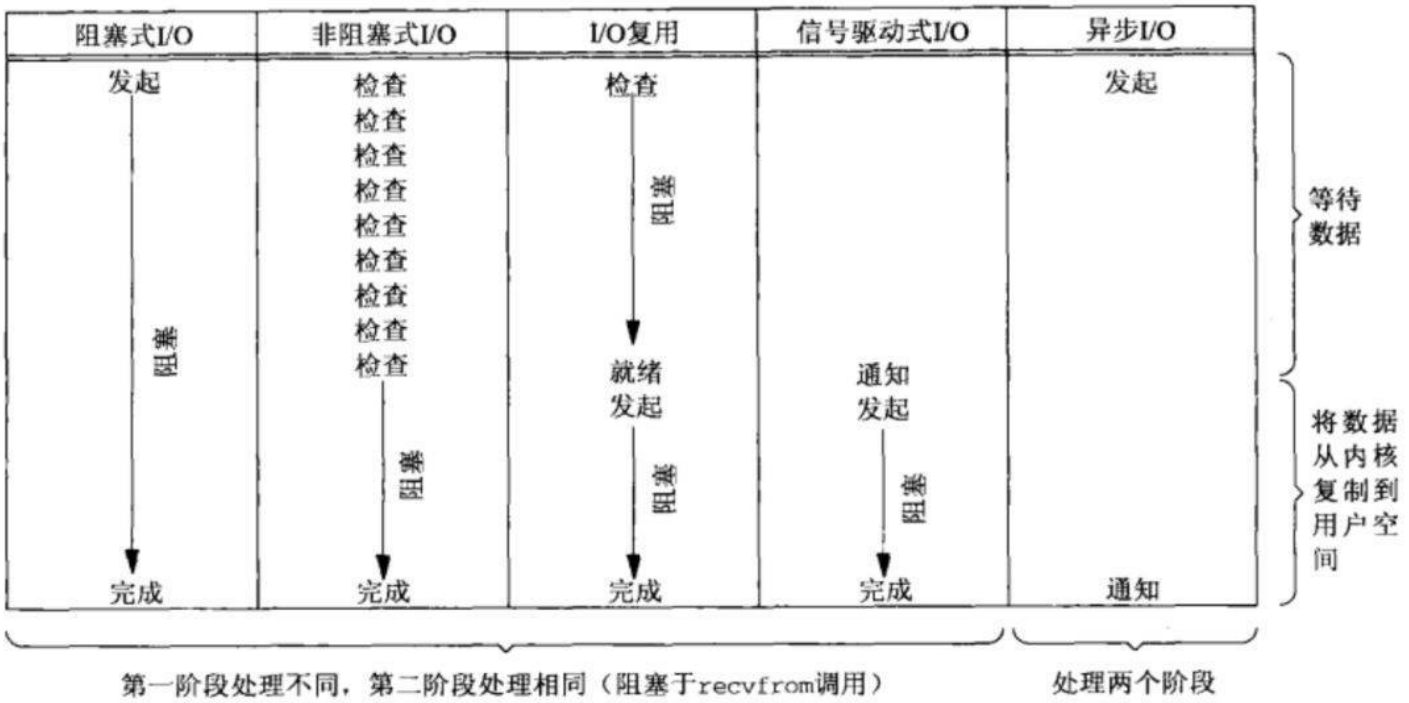


图6-5　异步I/O模型

- 五种IO模型比较:以上前四种都是同步的IO模型。因为，无论以上那种模型，真正的数据拷贝过程，都是同步进行的。信号驱动难道不是异步的么？ 信号驱动，内核是在数据准备好之后通知进程，然后进程再通过recvfrom操作进行数据拷贝。我们可以认为数据准备阶段是异步的，但是，数据拷贝操作是同步的。所以，整个IO过程也不能认为是异步的。



IO 多路复用模型

- I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，pselect，poll，epoll本质上都是同步I/O(epoll是Linux所特有，而select则应该是POSIX所规定的)，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。
- IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：
  1. 当客户处理多个描述符时（一般是交互式输入和网络套接口），必须使用I/O复用。
  2. 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。
  3. 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用
  4. 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。
  5. 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

select

- select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪（有数据 可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。

poll

- poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

epoll

- epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。

select、poll、epoll区别

- 支持一个进程所能打开的最大连接数

select	单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32*32，同理64位机器上FD_SETSIZE为32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。
poll	poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的
epoll	虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接

- FD剧增后带来的IO效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。
poll	同上
epoll	因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

- 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	同上
epoll	epoll通过内核和用户空间共享一块内存来实现的。

- 最佳实践：表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善。

谈谈死锁

死锁的概念

- 在两个或者多个并发进程中，如果每个进程持有某种资源而又等待其它进程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组进程产生了死锁。通俗的讲，就是两个或多个进程无限期的阻塞、相互等待的一种状态。

死锁产生的原因

- 系统中的资源可以分为两类：
- 可剥夺资源，是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺，CPU和主存均属于可剥夺性资源；
- 不可剥夺资源，当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如磁带机、打印机等。
- 产生死锁中的竞争资源之一指的是竞争不可剥夺资源（例如：系统中只有一台打印机，可供进程P1使用，假定P1已占用了打印机，若P2继续要求打印机打印将阻塞）
- 产生死锁中的竞争资源另外一种资源指的是竞争临时资源（临时资源包括硬件中断、信号、消息、缓冲区内的消息等），通常消息通信顺序进行不当，则会产生死锁。

死锁产生的四个必要条件

- 互斥：至少有一个资源必须属于非共享模式，即一次只能被一个进程使用；若其他申请使用该资源，那么申请进程必须等到该资源被释放为止；
- 占有并等待：一个进程必须占有至少一个资源，并等待另一个资源，而该资源为其他进程所占有；
- 非抢占：进程不能被抢占，即资源只能被进程在完成任务后自愿释放
- 循环等待：若干进程之间形成一种头尾相接的环形等待资源关系

解决死锁的基本方法

- 解决死锁的基本方法主要有 预防死锁、避免死锁、解除死锁
- 预防死锁的基本思想是 只要确保死锁发生的四个必要条件中至少有一个不成立，就能预防死锁的发生，具体方法包括：
  - 打破互斥条件：允许进程同时访问某些资源。但是，有些资源是不能被多个进程所共享的，这是由资源本身属性所决定的，因此，这种办法通常并无实用价值。
  - 打破占有并等待条件：可以实行资源预先分配策略(进程在运行前一次性向系统申请它所需要的全部资源，若所需全部资源得不到满足，则不分配任何资源，此进程暂不运行；只有当系统能满足当前进程所需的全部资源时，才一次性将所申请资源全部分配给该线程)或者只允许进程在没有占用资源时才可以申请资源（一个进程可申请一些资源并使用它们，但是在当前进程申请更多资源之前，它必须全部释放当前所占有的资源）。但是这种策略也存在一些缺点：在很多情况下，无法预知一个进程执行前所需的全部资源，因为进程是动态执行的，不可预知的；同时，会降低资源利用率，导致降低了进程的并发性。
  - 打破非抢占条件：允许进程强行从占有者哪里夺取某些资源。也就是说，但一个进程占有了一部分资源，在其申请新的资源且得不到满足时，它必须释放所有占有的资源以便让其它线程使用。这种预防死锁的方式实现起来困难，会降低系统性能。
  - 打破循环等待条件：实行资源有序分配策略。对所有资源排序编号，所有进程对资源的请求必须严格按资源序号递增的顺序提出，即只有占用了小号资源才能申请大号资源，这样就不回产生环路，预防死锁的发生。
- 避免死锁的基本思想是动态地检测资源分配状态，以确保循环等待条件不成立，从而确保系统处于安全状态。所谓安全状态是指：如果系统能按某个顺序为每个进程分配资源（不超过其最大值），那么系统状态是安全的，换句话说就是，如果存在一个安全序列，那么系统处于安全状态。资源分配图算法和银行家算法是两种经典的死锁避免的算法，其可以确保系统始终处于安全状态。其中，资源分配图算法应用场景为每种资源类型只有一个实例(申请边，分配边，需求边，不形成环才允许分配)，而银行家算法应用于每种资源类型可以有多个实例的场景。
- 解除死锁
  - 终止进程（简单粗暴），就是字面上的，你们死锁了，我就把你们一起杀掉，缺点就是如果一个进程跑了很长时间，但是被杀了，还得从头来。
  - 逐个终止进程，按照某种顺序，挨个杀死进程，每杀一个进程就去看看死锁解除了没有（每杀一个进程都会释放一些资源，如果释放好粗来的资源解决了死锁问题，就没必要再滥杀无辜了），没解除就继续杀。

计算机网络篇

- 主要参考
  - 协议森林
  - HTTP协议详解

从**URL**输入到页面展现到底发生什么？

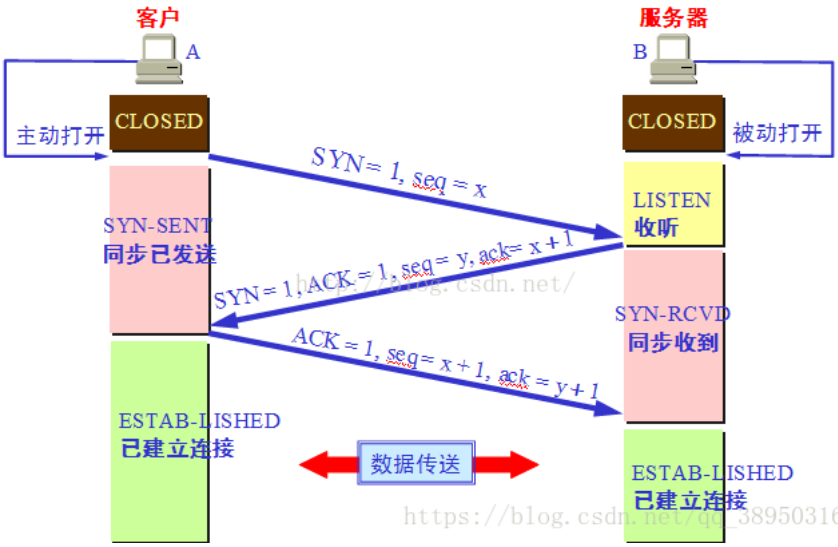
- DNS查询，ARP解析，TCP三次握手连接，HTTP请求，反向代理Nginx，uwsgi/gunicorn（WSGI服务器）,服务器响应，浏览器页面渲染，TCP四次挥手。

ARP 协议工作原理

- 网络层的 ARP 协议完成了 IP 地址与物理地址的映射。首先，每台主机都会在自己的 ARP 缓冲区中建立一个 ARP列表，以表示 IP 地址和 MAC 地址的对应关系。当源主机需要将一个数据包要发送到目的主机时，会首先检查自己 ARP 列表中是否存在该 IP 地址对应的 MAC 地址：如果有，就直接将数据包发送到这个 MAC 地址；如果没有，就向本地网段发起一个 ARP 请求的广播包，查询此目的主机对应的 MAC 地址。此 ARP 请求数据包里包括源主机的 IP 地址、硬件地址、以及目的主机的 IP 地址。网络中所有的主机收到这个 ARP 请求后，会检查数据包中的目的 IP 是否和自己的 IP 地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的 MAC 地址和 IP 地址添加到自己的 ARP 列表中，如果 ARP 表中已经存在该 IP 的信息，则将其覆盖，然后给源主机发送一个 ARP 响应数据包，告诉对方自己是它需要查找的 MAC 地址；源主机收到这个 ARP 响应数据包后，将得到的目的主机的 IP 地址和 MAC 地址添加到自己的 ARP 列表中，并利用此信息开始数据的传输。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败。

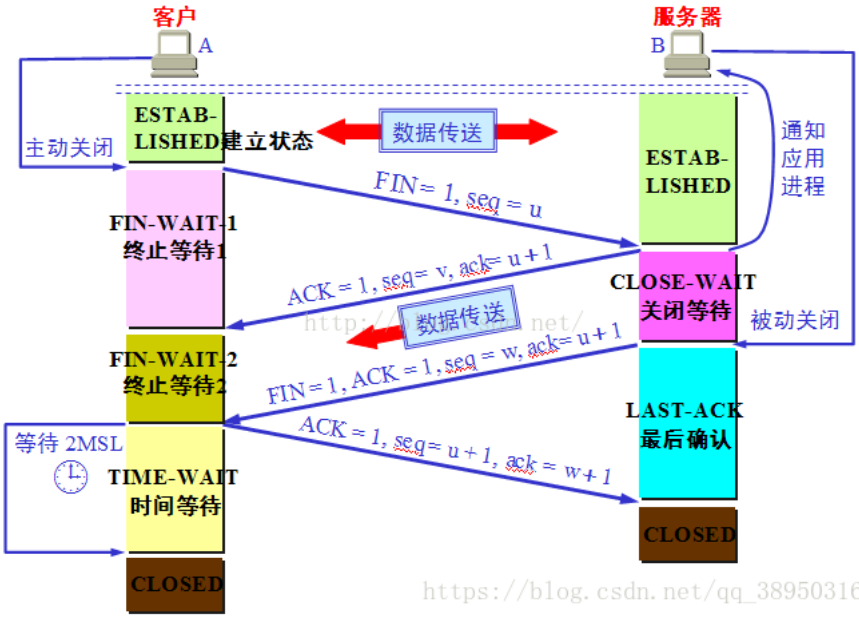
TCP 三次握手

- 重点记住每个阶段以后客户和服务器的状态变化



- 为什么不能用两次握手进行连接？
  答：为了防止 已失效的链接请求报文突然又传送到了服务端，因而产生错误。客户端发出的连接请求报文并未丢失，而是在某个网络节点长时间滞留了，以致延误到链接释放以后的某个时间才到达 Server。这是，Server 误以为这是 Client 发出的一个新的链接请求，于是就向客户端发送确认数据包，同意建立链接。若不采用 “三次握手”，那么只要 Server 发出确认数据包，新的链接就建立了。由于 client 此时并未发出建立链接的请求，所以其不会理睬 Server 的确认，也不与 Server通信；而这时 Server 一直在等待 Client 的请求，这样Server 就白白浪费了一定的资源。若采用 “三次握手”，在这种情况下，由于 Server 端没有收到来自客户端的确认，则就会知道 Client 并没有要求建立请求，就不会建立链接。

TCP 四次挥手



- 为什么TIME_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？
  答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。Server如果没有收到ACK，将不断重复发送FIN片段。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

TCP 怎样保证可靠性

- 数据包校验：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
- 对失序数据包重排序：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
- 丢弃重复数据：对于重复数据，能够丢弃重复数据；
- 应答机制：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；
- 超时重发：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；

- 流量控制：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议是可变大小的滑动窗口协议。

流量控制和拥塞控制

- 主要参考
  - TCP的流量控制和拥塞控制
- 区别注意，拥塞控制和流量控制不同，前者是一个全局性的过程，而后者指点对点通信量的控制。
  - 拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：慢开始、拥塞避免；快重传、快恢复。
  - 流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。

TCP 与 UDP 的区别

- TCP 是面向连接的，UDP 是无连接的；
- TCP 是可靠的，UDP 是不可靠的；
- TCP 只支持点对点通信，UDP 支持一对一、一对多、多对一、多对多的通信模式；
- TCP 是面向字节流的，UDP 是面向报文的；
- TCP 有拥塞控制机制; UDP 没有拥塞控制，适合媒体通信；
- TCP 首部开销 (20 个字节) 比 UDP 的首部开销 (8 个字节) 要大

HTTP协议

- 超文本传输协议(HTTP)是一种通信协议，它允许将超文本标记语言(HTML)文档从Web服务器传送到客户端的浏览器，目前我们使用的是HTTP/1.1 版本。
- HTTP协议是无状态的，同一个客户端的这次请求和上次请求是没有对应关系，对http服务器来说，它并不知道这两个请求来自同一个客户端。 为了解决这个问题， Web程序引入了Cookie机制和session机制来维护状态。
- HTTP/1.1起，默认都开启了Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。

HTTP 和 HTTPS 的区别

1. https协议需要到CA申请证书，一般免费证书较少，因而需要一定费用。
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl/tls加密传输协议。
3. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. http的连接很简单，是无状态的；HTTPS协议是由SSL/TLS+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

URL详解

- URL(Uniform Resource Locator) 地址用于描述一个网络上的资源, 基本格式如下：

```
schema://host[:port#]/path/.../[?query-string][#anchor]
>schema                 指定低层使用的协议(例如：http, https,ftp)
host                    HTTP服务器的IP地址或者域名
port#          HTTP服务器的默认端口是80，这种情况下端口号可以省略。如果使用了别的端口，必须指明
path                    访问资源的路径
query-string        发送给http服务器的数据
anchor              锚
```

HTTP 常见方法（GET/PUT）

- Http协议定义了很多与服务器交互的方法，最基本的有4种，分别是GET,POST,PUT,DELETE. 一个URL地址用于描述一个网络上的资源，而HTTP中的GET, POST, PUT, DELETE就对应着对这个资源的查，改，增，删4个操作。

安全性和幂等性

- 安全意味着该操作用于获取信息而非修改信息。幂等意味着对同一 URL 的多个请求应该返回同样的结果。

HTTP方法的安全性和幂等性见下表：

方法名	安全性	幂等性	
GET	是	是	请求指定的页面信息，并返回实体主体
HEAD	是	是	只请求页面的首部
OPTIONS	是	是	允许客户端查看服务器的性能
DELETE	否	是	请求服务器删除指定的页面
PUT	否	是	从客户端向服务器传送的数据取代指定的文档的内容
POST	否	否	请求服务器接受所指定的文档作为对所标识的URI的新的从属实体

GET和POST 区别

- 最常见的就是GET和POST了。GET一般用于获取/查询资源信息，而POST一般用于更新资源信息.
- GET提交的数据会放在URL之后，以?分割URL和传输数据，参数之间以&相连?name=test1&id=123456. POST方法是把提交的数据放在HTTP包的Body中.
- GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制。
- GET方式提交数据，会带来安全问题，比如一个登录页面，通过GET方式提交数据时，用户名和密码将出现在URL上，如果页面可以被缓存或者其他人可以访问这台机器，就可以从历史记录获得该用户的账号和密码。

常见 **HTTP** 状态码

- Response 消息中的第一行叫做状态行，由HTTP协议版本号，状态码，状态消息三部分组成。状态码用来告诉HTTP客户端,HTTP服务器是否产生了预期的Response。HTTP/1.1中定义了5类状态码，状态码由三位数字组成，第一个数字定义了响应的类别

```
1XX  提示信息 - 表示请求已被成功接收，继续处理
2XX  成功 - 表示请求已被成功接收，理解，接受
    200 OK :  表明该请求被成功地完成，所请求的资源发送回客户端
3XX  重定向 - 要完成请求必须进行更进一步的处理
    302 Found :  重定向，新的URL会在response 中的Location中返回，浏览器将会
    自动使用新的URL发出新的Request
    304 Not Modified :  代表上次的文档已经被缓存了，  还可以继续使用
    （提示：如果你不想使用本地缓存可以用Ctrl+F5 强制刷新页面）
4XX  客户端错误 -  请求有语法错误或请求无法实现
    400 Bad Request :  客户端请求与语法错误，不能被服务器所理解
    403 Forbidden :  服务器收到请求，但是拒绝提供服务
    404 Not Found : 请求资源不存在（还有可能是输错了URL)
5XX  服务器端错误 -  服务器未能实现合法的请求
    500 Internal Server Error:服务器发生了不可预期的错误
    503 Server Unavailable:服务器当前不能处理客户端的请求，一段时间后可能恢复正常
```

HTTP 长连接与短连接

长连接

- **长连接定义**：client方与server方先建立连接，连接建立后不断开，然后再进行报文发送和接收。这种方式下由于通讯连接一直存在。此种方式常用于P2P点对点的通信。长连接的操作步骤是：建立连接——数据传输...（保持连接）...数据传输——关闭连接
- **长连接适用场景**：以下这些连接，如果每次操作都要建立连接然后再操作的话处理速度会降低。所以操作时第一次连接上以后，以后每次直接发送数据就可以了，不用再建立TCP连接。
  - 即时通信系统：其它用户登录、发送信息；
  - 即时报价系统：后台数据库内容发生变化；
  - 数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，频繁的socket创建也是对资源的浪费。

短连接

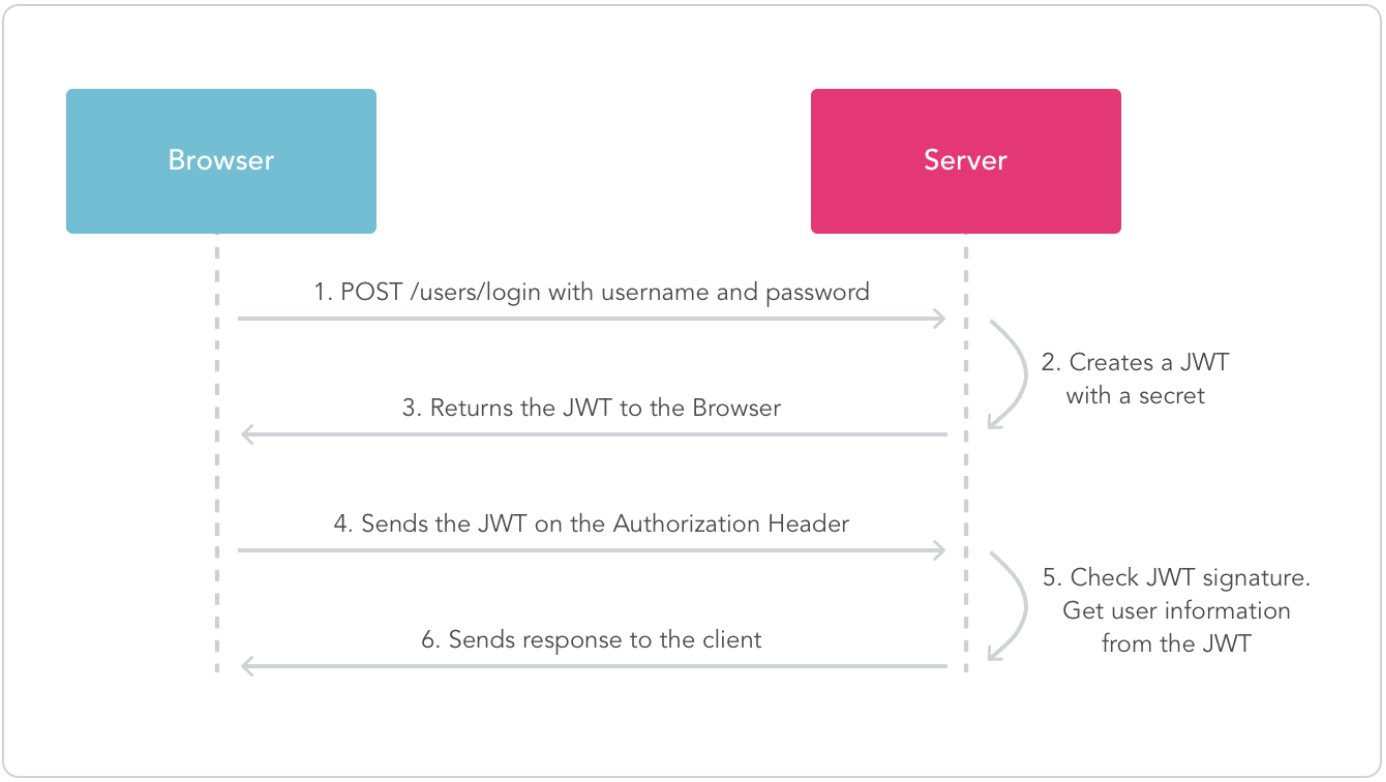
- **短连接定义**: Client方与server每进行一次报文收发交易时才进行通讯连接，交易完毕后立即断开连接。此方式常用于一点对多点通讯。短连接的操作步骤是：建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接
- **短连接的适用场景**：短连接多用于操作频繁，点对点的通讯，而且连接数不能太多的情况。每个TCP连接的建立都需要三次握手，每个TCP连接的断开要四次握手。web网站的http服务一般都用短连接。因为长连接对于服务器来说要耗费一定的资源。像web网站这么频繁的成千上万甚至上亿客户端的连接用短连接更省一些资源。

cookie 和 **session** 的区别

- 会话状态保存在客户端。客户端请求服务器，如果服务器需要记录该用户状态，就向客户端浏览器颁发一个 Cookie，而客户端浏览器会把Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器，服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie 的内容。
- 会话状态也可以保存在服务器端。客户端请求服务器，如果服务器记录该用户状态，就获取 Session 来保存状态，这时，如果服务器已经为此客户端创建过 session，服务器就按照 sessionid 把这个 session 检索出来使用；如果客户端请求不包含 sessionid，则为此客户端创建一个session 并且生成一个与此 session 相关联的 sessionid，并将这个 sessionid 在本次响应中返回给客户端保存。保存这个 sessionid 的方式可以采用 cookie 机制 ，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器；若浏览器禁用 Cookie 的话，可以通过 URL 重写机制将sessionid 传回服务器。
- 实现机制：Session 的实现常常依赖于 Cookie 机制，通过 Cookie 机制回传 SessionID；
- 安全性：Cookie 存在安全隐患，通过拦截或本地文件找得到 cookie 后可以进行攻击，而 Session 由于保存在服务器端，相对更加安全；
- 大小限制：Cookie 有大小限制并且浏览器对每个站点也有 cookie 的个数限制，Session 没有大小限制，理论上只与服务器的内存大小有关；
- 服务器资源消耗：Session 是保存在服务器端上会存在一段时间才会消失，如果 session 过多会增加服务器的压力。

JSON Web Token

- 主要参考
  - JSON Web Token 入门教程



- JWT 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准，该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

IP 地址的分类

- IP 地址是指互联网协议地址，是 IP 协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。IP 地址编址方案将 IP 地址空间划分为 A、B、C、D、E 五类，其中 A、B、C 是基本类，D、E 类作为多播和保留使用，为特殊地址。
- 每个 IP 地址包括两个标识码（ID），即网络 ID 和主机ID。同一个物理网络上的所有主机都使用同一个网络 ID，网络上的一个主机（包括网络上工作站，服务器和路由器等）有一个主机 ID 与其对应。A~E 类地址的特点如下：
  - A 类地址：以 0 开头，第一个字节范围：0~127；
  - B 类地址：以 10 开头，第一个字节范围：128~191；
  - C 类地址：以 110 开头，第一个字节范围：192~223；
  - D 类地址：以 1110 开头，第一个字节范围为: 224~239；
  - E 类地址：以 1111 开头，保留地址

OSI七层参考模型

数据库篇

- 学习书籍推荐
  - 知乎龚子捷的回答
  - 《高性能MySQL》

数据库系统概念

数据库范式

- 第一范式：列不可分，eg:【联系人】（姓名，性别，电话），一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF；
- 第二范式：有主键，保证完全依赖。eg:订单明细表【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName），Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID，不符合 2NF；
- 第三范式：无传递依赖 (非主键列 A 依赖于非主键列B，非主键列 B 依赖于主键的情况)，eg: 订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID），CustomerName，CustomerAddr，CustomerCity直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

视图

- 视图是一种虚拟的表，通常是有一个表或者多个表的行或列的子集，具有和物理表相同的功能，可以对视图进行增，删，改，查等操作。特别地，对视图的修改不影响基本表。相比多表查询，它使得我们获取数据更容易。

游标

- 游标是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行，可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。 MySQL 检索操作返回一组称为结果集的行，这组返回的行都是与 SQL 语句相匹配的行（零行或多行），使用简单的 SELECT 语句，不存在每次一行地处理所有行的简单方法（相对于成批地处理它们）。有时需要在检索出来的行中前进或后退一行或多行，这就是使用游标的原因。游标（cursor）是一个存储在 MySQL 服务器上的数据库查询，它不是一条 SELECT 语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

触发器

- 触发器是与表相关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据库的完整性。

存储过程

- 存储过程是事先经过编译并存储在数据库中的一段 SQL语句的集合。存储过程是由一些 T-SQL 语句组成的代码块，这些 T-SQL 语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。存储过程具有以下特点：

- 存储过程只在创建时进行编译，以后每次执行存储过程都不需再重新编译，而一般 SQL 语句每执行一次就编译一次，所以使用存储过程可提高数据库执行效率；
- 当 SQL 语句有变动时，可以只修改数据库中的存储过程而不必修改代码；
- 减少网络传输，在客户端调用一个存储过程当然比执行一串 SQL 传输的数据量要小；
- 通过存储过程能够使没有权限的用户在控制之下间接地存取数据库，从而确保数据的安全。

MySQL 基础问题

常用 **SQL** 语句

- 内连接，左连接，全连接
- union 和union all的区别：union会对结果集进行处理排除掉相同的结果，union all 不会对结果集进行处理，不会处理掉相同的结果。
- MySQL 常用数据类型：字符串、数值、日期
- 数据库查询语言分类
  - DQL（Data Query Language）数据查询语言DQL由SELECT子句，FROM子句，WHERE子句组成
  - DML（Data Manipulation Language）数据操纵语言DML包含INSERT，UPDATE，DELETE
  - DDL（Data Definition Language）数据定义语言DDL用来创建数据库中的各种对象-----表、视图、 索引、同义词、聚簇等，如：CREATE TABLE/VIEW/INDEX/SYN/CLUSTER。DDL操作是隐性提交的！不能rollback
  - DCL（Data Control Language）数据控制语言（DCL）是用来设置或者更改数据库用户或角色权限的语句，这些语句包括GRANT、DENY、REVOKE等语句，在默认状态下，只有sysadmin、dbcreator、db_owner或db_securityadmin等角色的成员才有权利执行数据控制语言。

in与not in,exists与not exists的区别

- exist会针对子查询的表使用索引
- not exist会对主子查询都会使用索引
- in与子查询一起使用的时候,只针对主查询使用索引
- not in则不会使用任何索引
- 如果查询的两个表大小相当，那么用in和exists差别不大；如果两个表中一个较小一个较大，则子查询表大的用exists，子查询表小的用in，所以无论哪个表大，用not exists都比not in 要快。

drop、delete 与 truncate 的区别

- delete 用来删除表的全部或者一部分数据行，执行delete 之后，用户需要提交 (commmit) 或者回滚(rollback) 来执行删除或者撤销删除， delete 命令会触发这个表上所有delete 触发器；
- truncate 删除表中的所有数据，这个操作不能回滚，也不会触发这个表上的触发器，truncate 比 delete 更快，占用的空间更小；
- drop 命令从数据库中删除表，所有的数据行，索引和权限也会被删除，所有的 DML 触发器也不会被触发，这个命令也不能回滚。
- 因此，在不再需要一张表的时候，用 drop；在想删除部分数据行时候，用 delete；在保留表而删除所有数据的时候用 truncate。

数据库事务

- 事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。

事务的特征**(ACID)**

- 原子性 (Atomicity)：事务所包含的一系列数据库操作要么全部成功执行，要么全部回滚；
- 一致性 (Consistency)：事务的执行结果必须使数据库从一个一致性状态到另一个一致性状态；
- 隔离性 (Isolation)：并发执行的事务之间不能相互影响；
- 持久性 (Durability)：事务一旦提交，对数据库中数据的改变是永久性的。

事务并发带来的问题

- 脏读：一个事务读取了另一个事务未提交的数据；（eg：事务A：张三妻子给张三转账100元。事务B：张三查询余额。事务A转账后（还未提交），事务B查询多了100元。事务A由于某种问题，比如超时，进行回滚。事务B查询到的数据是假数据。）
- 不可重复读：不可重复读的重点是修改，同样条件下两次读取结果不同；（eg：事务A：张三妻子给张三转账100元。事务B：张三两次查询余额。事务B第一次查询余额，事务A还没有转账，第二次查询余额，事务A已经转账了，导致一个事务中，两次读取同一个数据，读取的数据不一致。）
- 幻读：幻读的重点在于新增或者删除，一个事务两次读取一个范围的记录，两次读取的记录数不一致。（eg：事务A：张三妻子两次查询张三有几张银行卡。事务B：张三新办一张银行卡。事务A第一次查询银行卡数的时候，张三还没有新办银行卡，第二次查询银行卡数的时候，张三已经新办了一张银行卡，导致两次读取的银行卡数不一样。）

事务的隔离级别

- 参考数据库事务隔离级别-- 脏读、幻读、不可重复读（清晰解释）
- 数据库事务的隔离级别有4个，由低到高依次为Read uncommitted 、Read committed 、Repeatable read 、Serializable（最高级别的隔离，只允许事务串行执行。），后三个级别可以逐个解决脏读 、不可重复读 、幻读这几类问题，MySQL 的 InnoDB存储引擎都支持，MySQL 默认的隔离级别是 Repeatable read。

√: 可能出现　×: 不会出现

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

MySQL 的事务支持

- MySQL 的事务支持不是绑定在 MySQL 服务器本身，而是与存储引擎相关：
  - MyISAM：不支持事务，用于只读程序提高性能；
  - InnoDB：支持 ACID 事务、行级锁、并发；
  - Berkeley DB：支持事务。

数据库索引

- 索引是对数据库表中一个或多个列的值进行排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B树和B+树，索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。
- 在数据结构中，我们最为常见的搜索结构就是二叉搜索树和 AVL 树 (高度平衡的二叉搜索树，为了提高二叉搜索树的效率，减少树的平均搜索长度)了。然而，无论二叉搜索树还是 AVL 树，当数据量比较大时，都会由于树的深度过大而造成 I/O 读写过于频繁，进而导致查询效率低下，因此对于索引而言，B 树（平衡多路查找树）各操作能使 B 树保持较低的高度，从而保证高效的查找效率。

索引的优点和缺点

- 优点
  - 大大加快数据的检索速度，这也是创建索引的最主要的原因；
  - 加速表和表之间的连接；
  - 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间；
  - 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性；
- 缺点
  - 时间方面：创建索引和维护索引要耗费时间，具体地对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度；
  - 空间方面：索引需要占物理空间；

B 树和 B+ 树

- 参考重温数据结构：理解 B 树、B+ 树特点及使用场景
- B+ 树相比 B 树的优势
  - B+ 树的磁盘读写代价更低：B+树的内部结点并没有指向关键字具体信息的指针，因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多，相对来说 IO 读写次数也就降低了；
  - B+ 树的查询效率更加稳定：由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引，所以，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；
  - B+ 树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而 B 树只能中序遍历所有节点，效率太低。

索引的分类

- 普通索引和唯一性索引：索引列的值的唯一性
- 主键索引：指的就是主键，主键是索引的一种，是唯一索引的特殊类型（（唯一索引允许空值，主键不允许有空值）。创建主键的时候，数据库默认会为主键创建一个唯一索引；InnoDB 作为 MySQL 存储引擎时，默认按照主键进行聚集，如果没有定义主键，InnoDB 会试着使用唯一的非空索引来代替。如果没有这种索引，InnoDB 就会定义隐藏的主键然后在上面进行聚集。所以，对于聚集索引来说，你创建主键的时候，自动就创建了主键的聚集索引。
- 单个索引和复合索引：索引列所包含的列数
- 聚集索引按照数据的物理存储进行划分的。对于一堆记录来说，使用聚集索引就是对这堆记录进行堆划分，即主要描述的是物理上的存储。聚集索引可以帮助把很大的范围，迅速减小范围，然后查找指定记录；而非聚集索引是把一个很大的范围，转换成一个小的地图，然后你需要在这个小地图中找你要寻找的信息的位置，最后通过这个位置，再去找你所需要的记录。参考自快速理解聚集索引和非聚集索引
- 聚集和非聚集指：B+树叶节点存的指针还是数据记录，myISAM 非聚集（指针）innoDB聚集（指针+记录）

MySQL 中的锁

乐观锁和悲观锁

- 悲观锁的特点是先获取锁，再进行业务操作，即 “悲观” 的认为所有的操作均会导致并发安全问题，因此要先确保获取锁成功再进行业务操作。通常来讲，在数据库上的悲观锁需要数据库本身提供支持，即通过常用的 select … for update 操作来实现悲观锁。当数据库执行 select … for update 时会获取被 select 中的数据行的行锁，因此其他并发执行的 select … for update 如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update 获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。另外还有个问题是: select… for update 语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此，如果在 mysql 中用悲观锁务必要确定使用了索引，而不是全表扫描。（先上锁，再操作，一锁二查三更新）
- 乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部 update 同一行的时候是不允许并发的，即数据库每次执行一条 update 语句时会获取被 update 行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这其间没有发生并发的修改。如果更新失败，即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。（先修改，更新时发现数据变化就会滚）
- 悲观锁与乐观锁的应用场景：一般情况下，读多写少更适合用乐观锁，读少写多更适合用悲观锁。乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能。

MySQL 行级锁

- MyISAM 只支持表级锁，用户在操作MyISAM 表时，select、update、delete 和 insert语句都会给表自动加锁，如果加锁以后的表满足insert 并发的情况下，可以在表的尾部插入新的数据。InnoDB 支持事务和行级锁，行锁大幅度提高了多用户并发操作的新能，但是 InnoDB 的行锁只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表。
- 共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改。如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排它锁。SELECT ... LOCK IN SHARE MODE 会对查询出的每一条数据加共享锁，如果其它线程再加排它锁就会阻塞。
- 排他锁又称写锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的锁，获准排他锁的事务既能读数据，又能修改数据。SELECT ... FOR UPDATE 中会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

存储引擎 **MyISAM** 和 **InnoDB** 区别

- 事务支持：MyISAM 强调的是性能，每次查询具有原子性，其执行速度比 InnoDB 更快，但是不支持事务。InnoDB 提供事务、外键等高级数据库功能，具有事务提交、回滚能力。
- AUTO_INCREMENT：对AUTO_INCREMENT的处理方式不一样。如果将某个字段设置为INCREMENT，InnoDB中规定必须包含只有该字段的索引，如果是组合索引也必须是组合索引的第一列。但是在MyISAM中，可以将该字段和其他字段一起建立组合索引，可以不是第一列。
- 表主键：MyISAM 允许没有任何索引和主键的表存在，索引都是保存行的地址。对于 InnoDB，如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键 (用户不可见)，数据是主索引的一部分。
- 表的具体行数：MyISAM内置了一个计数器来存储表的行数。执行 select count(*)时直接从计数器中读取，速度非常快。而 InnoDB不保存这些信息，如果使用 select count(*) from table 就会遍历整个表，消耗相当大。（在加了wehre 条件后，myisam 和 innodb 处理的方式都一样）
- 全文索引：MyISAM支持 FULLTEXT类型的全文索引，InnoDB不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。
- 外键：MyISAM 不支持外键，而 InnoDB 支持外键。
- 存储结构：每个 MyISAM 在磁盘上存储成三个文件：第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm 文件存储表定义，数据文件的扩展名为. MYD (MYData)，索引文件的扩展名是. MYI(MYIndex)。InnoDB 所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB。
- CURD操作：如果执行大量的SELECT，MyISAM是更好的选择。如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

实现**MVCC**

- MVCC 全称是Multi-Version Concurrent Control，即多版本并发控制，在MVCC协议下，每个读操作会看到一个一致性的snapshot，并且可以实现非阻塞的读。MVCC允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务ID，在同一个时间点，不同的事务看到的数据是不同的。
- innodb会为每一行添加两个字段，分别表示该行创建的版本和删除的版本，填入的是事务的版本号，这个版本号随着事务的创建不断递增。在repeated read的隔离级别下，具体各种数据库操作的实现：
  - select：满足以下两个条件innodb会返回该行数据：该行的创建版本号小于等于当前版本号，用于保证在select操作之前所有的操作已经执行落地。该行的删除版本号大于当前版本或者为空。删除版本号大于当前版本意味着有一个并发事务将该行删除了。
  - insert：将新插入的行的创建版本号设置为当前系统的版本号。
  - delete：将要删除的行的删除版本号设置为当前系统的版本号。
  - update：不执行原地update，而是转换成insert + delete。将旧行的删除版本号设置为当前版本号，并将新行insert同时设置创建版本号为当前版本号。
- 其中，写操作（insert、delete和update）执行时，需要将系统版本号递增。由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做purge。通过MVCC很好的实现了事务的隔离性，可

以达到repeated read级别，要实现serializable还必须加锁。

实践中如何优化 **MySQL**

- 实践中，MySQL 的优化主要涉及 SQL 语句的优化，索引的优化，数据表结构的优化。其他包括系统配置和硬件的优化。

SQL 语句的优化

- 发现有问题的 SQL：MySQL 的慢查询日志是 MySQL 提供的一种日志记录，用来记录在 MySQL 中响应时间超过阀值的语句，具体指运行时间超过 long_query_time(long_query_time 的默认值为 10，意思是运行10s 以上的语句。) 值的 SQL，则会被记录到慢查询日志中。通过 MySQL 的慢查询日志可以查询出执行次数多、占用时间长的 SQL，用 pt_query_disgest(一种 mysql 慢日志分析工具) 分析 Rows examine(MySQL 执行器需要检查的行数) 项去找出 IO 大的 SQL 以及发现未命中索引的 SQL，对其进行优化。
- 分析 SQL 的执行计划：使用 EXPLAIN 关键字可以知道 MySQL 是如何处理 SQL 语句的，以便分析查询语句或是表结构的性能瓶颈。通过 explain 命令可以得到表的读取顺序、哪些索引被实际使用、表之间的引用以及每张表有多少行被优化器查询等问题。当扩展列 extra 出现 Using filesort 和 Using temporay，则往往表示SQL 需要优化了。具体参考MySQL 性能优化神器 Explain 使用分析
- 优化 SQL 语句：
  - 优化 insert 语句：一次插入多值
  - 应尽量避免在 where 子句中使用!= 或 <> 操作符，则将引擎放弃使用索引而进行全表扫描；
  - 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描；
  - 优化嵌套查询：子查询可以被更有效率的连接 (Join)替代；
  - 很多时候用 exists 代替 in 是一个好的选择。

索引的优化

- 适合创建索引的字段
  - 经常作查询选择的字段
  - 经常作表连接的字段
  - 经常出现在 order by, group by, distinct 后面的字段
- 创建索引时需要注意什么
  - 非空字段：应该指定列为 NOT NULL，除非你想存储NULL。在 MySQL 中含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。应该用 0、一个特殊的值或者一个空串代替空值；
  - 取值离散大的字段（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过 count() 函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
  - 索引字段越小越好：数据库的数据存储以页为单位，一页存储的数据越多，一次 IO 操作获取的数据越多，效率越高。
- 索引失效的情况
  - 以 “%(表示任意 0 个或多个字符)” 开头的 LIKE 语句，模糊匹配；
  - OR 语句前后没有同时使用索引；
  - 数据类型出现隐式转化（如 varchar 不加单引号的话可能会自动转换为 int 型）；
  - 对于多列索引，必须满足 最左匹配原则 (eg：多列索引 col1、col2 和 col3，则 索引生效的情形包括 col1或 col1，col2 或 col1，col2，col3)。

数据表结构的优化

- 选择合适数据类型
  - 使用较小的数据类型解决问题；
  - 使用简单的数据类型 (mysql 处理 int 要比 varchar容易)；
  - 尽可能的使用 not null 定义字段；
  - 尽量避免使用 text 类型，非用不可时最好考虑分表。
- 表的范式的优化，一般情况下表的设计应该遵循三大范式。
- 表的垂直拆分
  - 把不常用的字段单独放在同一个表中；
  - 把大字段独立放入一个表中；
  - 把经常使用的字段放在一起；
- 表的水平拆分
  - 表的水平拆分用于解决数据表中数据过大的问题，水平拆分每一个表的结构都是完全一致的。一般地，将数据平分到 N 张表中的常用方法。包括以下两种：对 ID 进行 hash 运算，如果要拆分成 5 个表，mod(id,5) 取出 0~4 个值；针对不同的 hashID 将数据存入不同的表中；
  - 表分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度；
  - 表中的数据本来就有独立性，例如表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常用，而另外一些数据不常用。

Redis

- 参考 面试中关于Redis的问题看这篇就够了
- Redis 是一个使用 C 语言写成的，开源的 key-value 数据库。和Memcached类似，但支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set –有序集合)和hash（哈希类型），这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这

些操作都是原子性的。在此基础上，redis支持各种不同方式的排序，与memcached一样，为了保证效率，数据都是缓存在内存中，区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

- 适合存储在非关系型数据库中的数据：关系不是很密切的的数据，比如用户信息，班级信息，评论数量等等；量比较大的数据，如访问记录等；访问比较频繁的数据，如用户信息，访问数量，最新微博等。

Redis数据类型

- string 常用命令: set,get,decr,incr,mget ，是最基本的数据类型，一个键对应一个值，需要注意是一个键值最大存储512MB。用于常规计数：微博数，粉丝数等。
- hash 常用命令：hget,hset,hgetall 等，redis hash是一个键值对的集合，是一个string类型的field和value的映射表，适用于存储用户信息，商品信息。
- List 常用命令: lpush,rpush,lpop,rpop,lrange等，是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，最新消息排行等功能都可以用Redis的list结构来实现。Redis list的实现为一个双向链表，可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。
- Set常用命令：sadd,spop,smembers,sunion 等，对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动去重的，并且set提供了判断某个成员是否在一个set集合内的重要接口。在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常方便的实现如共同关注、共同喜好、二度好友等功能。
- Sorted Set常用命令： zadd,zrange,zrem,zcard 等，和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按 score 进行有序排列。（直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用Redis中的SortedSet结构进行存储。）

Redis的应用场景

- 访问热点数据，减少响应时间，提升吞吐量
  - 会话缓存（最常用）
  - 消息队列，比如支付
  - 活动排行榜或计数
  - 发布，订阅消息（消息通知）
  - 商品列表，评论列表等

Redis 持久化

- 参考redis的持久化和缓存机制
- 快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久化的方式。可以配置 redis在 n 秒内如果超过 m 个 key 被修改就自动做快照，下面是默认的快照保存配置：

```
保存 900 1   # 900秒内如果超过1个Key被修改，则启动快照保存
保存300 10   # 300秒内如果超过10个Key被修改，则启动快照保存
保存60 10000 # 60秒内如果超过10000个Key被修改，则启动快照保存
```

- aof 比快照方式有更好的持久化性，是由于在使用 aof 持久化方式时,redis 会将每一个收到的写命令都通过 write 函数追加到文件中(默认是 appendonly.aof)。当 redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容，AOF持久化存储方式参数说明

```
appendonly yes  # 开启AOF持久化存储方式
appendfsync always  # 收到写命令后就立即写入磁盘，效率最差，效果最好
appendfsync everysec  # 每秒写入磁盘一次，效率与效果居中
appendfsync no  # 完全依赖操作系统，效率最佳，效果没法保证
```

缓存使用过程中的坑

- 引用自 缓存穿透，缓存击穿，缓存雪崩解决方案分析
- 缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。解决方案：如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。
- 缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。解决方案：将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。
- 缓存击穿，对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，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。

其他问题

- 高并发场景下数据重复插入的问题

    - 使用关系型数据库的唯一索引
    - Redis 实现分布式锁：
- Redis常见性能问题和解决方案
    - Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
    - 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
    - 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
    - 尽量避免在压力很大的主库上增加从库

最佳实践

- Phodal的全栈增长工程师实战

编码规范

- PEP8
- Python 之禅
- 了解 dosctring
- 了解类型注解
- Python 对象的命名规范，例如方法或者类
- Python 中的注释
- 优雅的给一个函数加注释
- 给变量加注释
- Python 代码缩进中是否支持 Tab 键和空格混用
- 是否可以在一句 import 中导入多个库
- 在给 Py 文件命名的时候需要注意什么
- 例举几个规范 Python 代码风格的工具

正确的流程开发

1. 用GitHub或类似的现代平台
2. 平台上设置禁止直接push到主干，所有的修改必须fork后走Pull Request
3. 启用CI（持续集成），提PR时平台自动执行CI步骤，失败的不能被合并（不准开任何后门）
4. CI加入linter，确保代码规范；所有代码规范必须要可由linter检测，代码规范/linter配置规则也要针对实践中发现的问题不断补充细化和更新
5. CI加入单元测试，代码的测试覆盖率至少60%以上，核心模块测试覆盖率必须90%以上；所有发现的bug必须由造成bug的人负责补上单元测试
6. 每个PR强制要求改动代码行数小于100行，新人要求小于60行，以利code review
7. 每个PR在CI通过后必须有其他人进行过code review并approve，否则不能被merge，新人的代码必须至少有两人review和approve（比如新人的mentor和相关代码文件或目录的owner）
8. 针对每个PR自动部署一份到测试环境，方便自测或提供给测试团队进行必要的测试
9. 每2周检查近期bug，总结经验教训，特别是重复犯的错误一定要建立机制去防范

单元测试

- 单元测试可以防止回归的时候出现问题,一个函数，一个类的逐次开始验证，自底向上测试，易测试的代码往往是高内聚低耦合的。
- Django测试
- 单元测试相关的库
    - nose/pytest较为常用
    - mock模块用来模拟替换网络请求
    - coverage 统计测试覆盖率
- 测试用例设计原则：
    - 正常值功能测试
    - 边界值（最大最小，或者最左最右）
    - 异常值（none，空值，非法值）

CI/CD工具篇

- 谁才是世界上最好的 CI/CD 工具？

Git飞行规则(Flight Rules)

Docker 篇

- 用Docker部署一个Web应用

Web 扩展

Web安全篇

DDos 攻击

- 客户端向服务端发送请求链接数据包，服务端向客户端发送确认数据包，客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认。

XSS 攻击

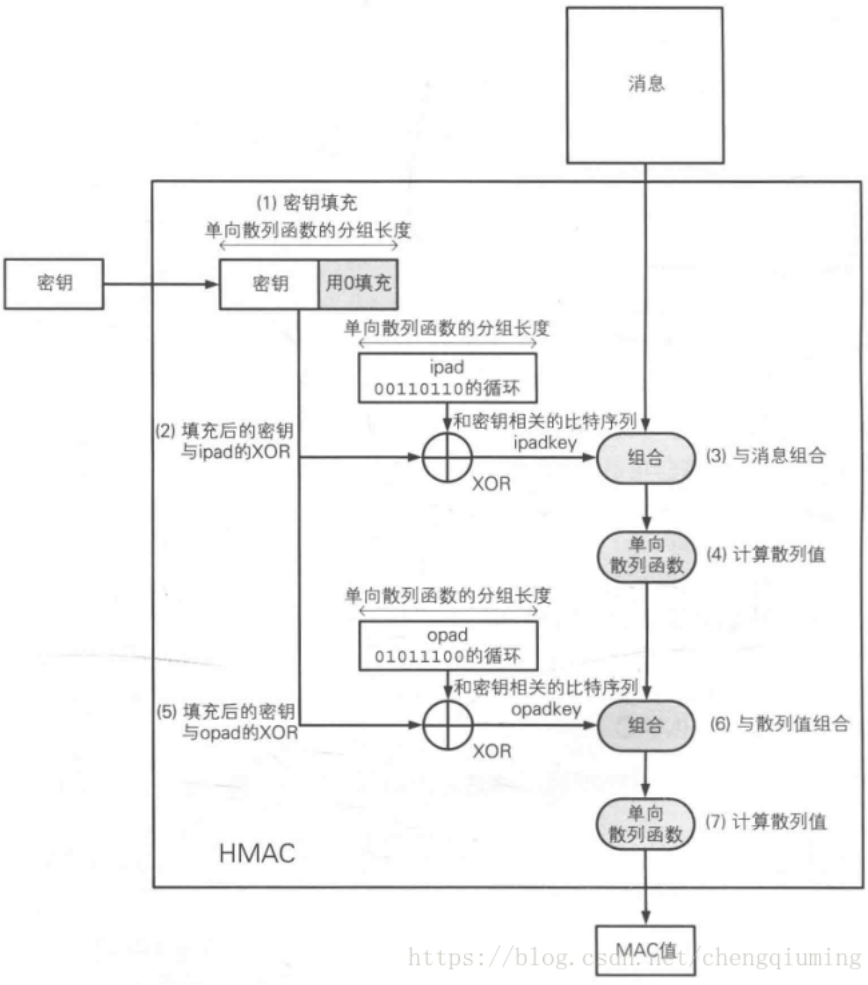
- XSS 是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到 web 页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。
- 主要原因：过于信任客户端提交的数据！解决办法：不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。
- 反射性 XSS 攻击 (非持久性 XSS 攻击)和持久性 XSS 攻击 (留言板场景)

SQL 注入攻击

- SQL 注入就是通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。
- SQL 注入攻击的总体思路：寻找到 SQL 注入的位置，判断服务器类型和后台数据库类型，针对不同的服务器和数据库特点进行 SQL 注入攻击。

加密

- 对称密钥加密是指加密和解密使用同一个密钥的方式，这种方式存在的最大问题就是密钥发送问题，即如何安全地将密钥发给对方；而非对称加密是指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息后，使用自己的私钥进行解密。
- 由于非对称加密的方式不需要发送用来解密的私钥，所以可以保证安全性；但是和对称加密比起来，它非常的慢，所以我们还是要用对称加密来传送消息，但对称加密所使用的密钥我们可以通过非对称加密的方式发送出去。
- 用于消息验证的hash算法：HMAC



Nginx篇

- 主要参考

- nginx源码分析
- 深入浅出搞懂Nginx

Vue篇

- Vue

Django篇

- Django初步使用Celery
- URL Dispatcher
- Class-based View

WSGI、uwsgi、uWSGI 区别

- 主要参考
  - Web 开发规范 — WSGI
- WSGI：全称是Web Server Gateway Interface，WSGI不是服务器，python模块，框架，API或者任何软件，只是一种规范，描述web server如何与web application通信的规范。server和application的规范在PEP 3333中有具体描述。要实现WSGI协议，必须同时实现web server和web application，当前运行在WSGI协议之上的web框架有Bottle, Flask, Django。
- uwsgi：与WSGI一样是一种通信协议，是uWSGI服务器的独占协议，用于定义传输信息的类型(type of information)，每一个uwsgi packet前4byte为传输信息类型的描述，与WSGI协议是两种东西，据说该协议是fcgi协议的10倍快。
- uWSGI：是一个web服务器，实现了WSGI协议、uwsgi协议、http协议等。

正则表达式篇

- 可能是最好的正则表达式的教程笔记了吧...

其他

- 优质 Python 播客推荐
- PYCON CHINA 2019
- JetBrains IDE 基本快捷键