Python函数
进阶用法

本节重点：

- 高阶函数
- 嵌套函数与闭包
- 作用域与命名空间(LEGB)
- 语法糖-装饰器
- 语法糖-迭代器
- 异常处理

高阶函数

思考：传入的参数和返回结果能是函数对象吗？

- 答案：可以的！
- 变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。
- 例子： +-*/的使用。

```
# # 高阶函数示例
# def add(x,y):
#     return x+y
# def min(x,y):
#     return x-y
# def mul(x,y):
#     return x *y
# def div(x,y):
#     return x / y
# # 这是1个高阶函数，返回结果是函数
# def get_func(flag):
#     if flag == "+":
#         return add
#     if flag == "-":
#         return min
#     if flag == "*":
#         return mul
#     if flag == "/":
#         return div
#     return None
# # 这个也是一个高阶函数，传入参数有函数
# def do_calc(func, x, y):
#     return func(x,y)
# def calc(x, y, flag):
#     func = get_func(flag)
#     return do_calc(func, x, y)
# print(calc(1, 2, "+"))
```

```
# # 高阶函数示例
# def add(x,y):
#     return x+y
#
# def min(x,y):
#     return x-y
#
# def mul(x,y):
#     return x *y
#
# def div(x,y):
#     return x / y
#
# #
# # 善于使用 python提供的灵活强大的工具特性
# def get_func(flag):
#     func = globals().get(flag)
#     return func
# #
# # 这个也是一个高阶函数，传入参数有函数
# def do_calc(func, x, y):
#     return func(x,y)
#
# def calc(x, y, flag):
#     func = get_func(flag)
#     return do_calc(func, x, y)
#
# print(calc(1, 2, "add"))
# print(calc(1, 2, "min"))
# print(calc(1, 2, "mul"))
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - map: 生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后放入到新的数组中。
    - 举例：数组的平方，或取绝对值　　(1-1映射的关系)

```
# 数组的平方
#
# # 方法1：
# li = [1,2,3,4]
# nli = []
# for ele in li:
#     nli.append(ele * ele)
#
# # 方法2：
# nli = [ele * ele for ele in li]
```

```
# 方法3：
# def func(ele):
#     return ele * ele
# nli = map(func, li)
# # print(list(mp))
# for ele in nli:
#     print(ele)

# 方法4：
# def func(ele):
#     return ele * ele
# nli = map(lambda ele: ele * ele, li)
# # print(list(mp))
# for ele in nli:
#     print(ele)
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - filter:生成一个新数组，在遍历原数组的时候只将返回值为 true 的元素放入新数组。
    - 举例：过滤出所有的偶数/奇数。

```
# 过滤所有奇数
#
# nli = [ele for ele in li if ele % 2 > 0]
# print(nli)
#
# def func(ele):
#     return ele % 2 > 0

# nli = filter(lambda ele: ele % 2 > 0, li)
#
# print(list(nli))
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - reduce：将数组中的元素通过回调函数最终转换为一个值。
    - 举例：取最大值/最小值/平均值

```
# 求和
#
# from functools import reduce
# #
# # def func(e1,e2):
# #     return e1 + e2
#
# li = [1,10,100,1000]
# val = reduce(lambda e1, e2: e1 + e2, li)
# print(val)
```

```
# 求最大值
#
# from functools import reduce
#
# # def func(e1,e2):
# #     return e1 if e1 > e2 else e2
#
# li = [1,10,100,1000]
# val = reduce(lambda e1, e2: e1 if e1 > e2 else e2, li)
# print(val)
```

课堂练习（1）

1. 编写嵌套函数area()，专门计算图形的面积（兼容rect、circle、squre）
2. 利用map，实现1个函数，返回列表的平方数。
3. 利用map，实现1个函数，返回列表的绝对值。
4. 利用 filter，实现1个函数，返回列表的偶数项。
5. 利用reduce，实现1个函数，返回列表之和。
6. 利用reduce，实现1个函数，返回列表中元素的最大值
7. 人员信息字典：dic=[{‘name’:‘p1’, ‘salary’:200}, {‘name’:’p2’, “salary’:2000} , {‘name’:’p3’, “salary’:1500}]
   - 利用map，实现1个函数，所有人员的工资均增加 500元。
   - 利用 filter，实现1个函数，返回 工资大于 1000的人名列表
   - 利用reduce，实现1个函数，返回工资最高的人员名称
   - 利用reduce，实现1个函数，返回最高工资
   - 利用reduce，实现1个函数，返回平均工资

课堂练习（2）

写函数area()，专门计算图形的面积（兼容rect、circle、squre），按如下要求：

```
# # 方式1:
# def squre(x):
#     return x*x
#
# def rect(x, y):
#     return x * y
#
# def circle(r):
#     return 3.1415926 * r * r
#
# print(squre(10))
# print(rect(10, 20))
# print(circle(30))
#
```

```
# # 方式2:
# def area(tp, *args):
#     if tp == 'squre':
#         return squre(args[0])
#     if tp == 'rect':
#         return rect(args[0], args[1])
#     if tp == 'circle':
#         return circle(args[0])
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
#
```

```
# # 方式3:
# def area(tp, *args):
#     func = None
#     if tp == 'squre':
#         func = squre
#     if tp == 'rect':
#         func = rect
#     if tp == 'circle':
#         func = circle
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
#
#
# def circle2(x, y, z):
#     return x * y * z
#
```

```
# # 方式4:
# def area(tp, *args):
#     func = globals().get(tp, None)
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
# print(area('circle2', 30, 40, 50))
```

```
# # 方式5: 内嵌函数 + 高阶函数 + locals()
# def area(tp, *args):
#     def squre(x):
#         return x * x * 10
#
#     def rect(x, y):
#         return x * y * 10
#
#     def circle(r):
#         return 3.1415926 * r * r * 10
#
#     def circle2(x, y, z):
#         return x * y * z * 10
#
#     func = locals().get(tp, None)
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
# print(area('circle2', 30, 40, 50))
```

课堂练习（3）

```
# map / filter / reduce
# 需求， 返回列表的平方数。
# li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 方式1:
# li_new = []
# # for i in li:
# #     li_new.append(i * i)
# # print(li_new)

# for i in data:
#     pass

# 方式2: map(func, data) data: list/dict/set/str 。。。。
# def sqr(x):
#     return x * x

# 方式3: 匿名函数
# lambda x: x * x
```

课堂练习（4）

```
# # filter 的用法
# filter(func, data) -> 只有func 返回值  为True, data的数据才会插入新的列表
# print(list(filter(lambda a: True if a >= 0 else False, li)))
#
#
# def filter_func(x):
#     return x > 0
#
#
# print(list(filter(filter_func, li)))
# 1. 从 li中取出一个数; x
# 2. 调用 func()，  对数进行判断
# 3. 如果结果为True，则插入新的数组。

# filter 与 map的区别。
# map 不改变长度，但改变数值。
# filter 不改变数值，但改变长度。
```

课堂练习（5）

```
# reduce

# print(li)
# from functools import reduce
# 求和
# def reduct_func(x, y):
#     return x + y
# print(reduce(reduct_func, li))
# li=[5,2,3,4]
# 最小值
# def reduct_func(x, y):
#     print("x",x)
#     print("y",y)
#     return x if x < y else y
# print(reduce(reduct_func, li))
# 平均值
# def add(x, y):
#     return x + y
# print(reduce(add, li)/len(li))
```

课堂练习（6）

```
#
# oper_dict = {'+': lambda x, y: x+y,
#              '-': lambda x, y: x-y,
#              '*': lambda x, y: x*y,
#              '/': lambda x, y: x/y,
#              '**': lambda x, y: x **y}
# def calc(tp, *args):
#     func = oper_dict.get(tp, None)
#     if not func:
#         print("找不到。。。")
#         return None
#     return func(*args)
#
# print(calc('+', 2, 3))
# print(calc('-', 2, 3))
# print(calc('*', 2, 3))
# print(calc('/', 2, 3))
# print(calc('**', 2, 3))
```

嵌套函数与闭包

- 内嵌函数（允许在函数内部创建另一个函数，叫内部函数）
- 内部函数的整个作用域都在外部函数之内，如fun2整个定义和调用的过程都在fun1里面，除了在fun1里可以任意调用，出了fun1之外无法被调用，会系统报错。

嵌套函数示例1：

```
name = '1'
def fun1():
    name = '2'
    print('fun1 is using...', name)
    def fun2():
        name = '3'
        print('fun2 is using...', name)
    fun2()
    print('fun1 end...', name)
    name = '4'
fun1()
print('fun1 end...', name)
```

- 嵌套函数可以访问父函数中声明的所有局部变量、参数，嵌套函数在被外部调用时，就会形成闭包。
- 闭包的作用就是在父函数执行完并返回后，解释器垃圾回收机制不会收回父函数所占用的资源。
- 返回的函数对象，不仅仅是一个函数对象，在该函数外还包裹了一层作用域，该函数无论在何处调用，优先使用自己外层包裹的作用域

闭包示例2：

```
def funX(x):  # 外部作用域 的变量 x
    def funY(y):  # 内部函数（闭包）
        return x * y  # 引用了x变量

    return funY

fun = funX(8)
print(fun)  # <function funX.<locals>.funY at 0x000002889E0E8510>
print(type(fun))  # <class 'function'>
print(funX(8)(5))  # 40
print(fun(5))  # 40
```

命名空间（作用域）

又名name space, 存放变量标签与内存地址的绑定关系，python里面有很多名字空间，每个地方都有自己的名字空间，互不干扰，不同空间中的两个相同名字的变量之间没有任何联系。

名称空间有4种:LEGB

- locals:函数内部，包括函数局部变量以及形式参数
- enclosing function:在嵌套函数中外部函数的名字空间, 若fun2嵌套在fun1里，对fun2来说，fun1的名字空间就enclosing.
- globals: 当前的模块空间，模块就是一些py文件。也就是说，globals()类似全局变量。
- builtins: 内置模块空间，也就是内置变量或者内置函数的名字空间，print(dir(builtins))可查看。

不同变量的作用域不同就是由这个变量所在的名称空间决定的。
不同的空间，有不同的查询方式。

作用域查找顺序
当程序引用某个变量的名字时，就会从当前名字空间开始搜索。搜索顺序规则便是:LEGB。即locals -> enclosing function -> globals ->builtins。一层一层的查找，找到了之后，便停止搜索，如果最后没有找到,则抛出在NameError的异常。

作用域示例（LEGB）

```python
# LEGB local - enclosed - global - buildin

# name = 'global_name'  #  全局
# def func():
#     name = 'enclosed_name'  # 闭包
#     # name2 = 'cc'
#     def func1(): # 内部函数
#         # name = 'local_name'  # 本地
#         return name
#     return func1
#
# a = func() # 高阶函数
# print(a())
#
# print(globals())

#print(a)
# name = 'global_name2'

# print(a())
```

```python
# # python 解析机制，以及 变量/函数的注册流程
# def oper(name):
#     #1.  'name':name -> locals()
#     def add(x, y):
#         z = x + y
#         print(name, locals())
#         return z
#
#     #2. 'calc': calc -> locals()
#     def mul(x, y):
#         z = x * y
#         print(name, locals())
#         return z
#     #3. 'calc': calc -> locals()
#     print("oper::locals()", locals())
#
#     func = locals().get(name, None)
#     if not func:
#         print("error", name)
#     return func
#     # if name == 'add':
#     #     return add
#     # if name == 'mul':
#     #     return mul
#
# # func = oper("mul")
# # val = func(2, 3)
#
# val = oper("add")(2,3)
# print(val)
```

```python
# python 解析的规则
# = / += / -= / *= / /=,
# 符号左边的变量会被python认为是局部变量，除非global 来提前指定。
#
# a = 1
# def foo():
#     def bar():
#         a += 1
#         return a
#     return bar
#
# c = foo()
# val = c()
# print('第1次:', val)
# val = c()
# print('第1次:', val)
```

作用域示例

```
def func():
    # name = 'enclosed_name'
    def sub_func():
        # name = 'local_name'
        return name
    return sub_func
name = "global_name"
a = func()
b = a()
print(b)
```

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
c =foo()
c()
```

- 这段程序的本意是要通过在每次调用闭包函数时都对变量a进行递增的操作。但在实际使用时
- 在执行代码 c = foo()时，python会导入全部的闭包函数体bar()来分析其的局部变量，python规则指定所有在赋值语句左面的变量都是局部变量，则在闭包bar()中，变量a在赋值符号"="的左面，被python认为是bar()中的局部变量。再接下来执行print c()时，程序运行至a = a + 1时，因为先前已经把a归为bar()中的局部变量，所以python会在bar()中去找在赋值语句右面的a的值，结果找不到，就会报错

```
flist = []
for i in range(4):
    print(id(i))
    def foo(x):
        print(id(i))
        print(x + i)
    flist.append(foo)

print("this:", id(i))
i = 2
for f in flist:
    f(2)
```

```
flist = []
for i in range(4):
    print(id(i))
    def foo(x, y=i):
        print(x + y)
    flist.append(foo)

print("this:", id(i))
i = 2
for f in flist:
    f(2)
```

分析一下计算结果，思考为什么？

作用域示例

```
# LEGB 变量引用顺序
# L: LOCAL
# E：ENCLOSED
# G：GLOBALS
# B: buildins
# name = 'global_name'  # 1
# def func():
#     name = 'enclosed_name'  # 2
#     # name2 = 'cc'
#     def func1():
#         name = 'local_name'  # 3
#         # print(name2)
#         return name
#     return func1
#
# a = func()
# print(a())
# print(a)
# name = 'global_name2'
#
# print(a())
```

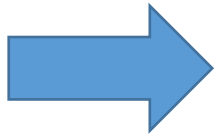
语法糖-装饰器-不推荐用法1

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式1

```
def login():
    print("come to login...")


def func1():
    login()
    print("come to func1...")


func1()
func2()
func3()
```

每个模块都需要修改，直接违反了软件开发中的一个原则“开放-封闭”原则，

简单来说，它规定已经实现的功能代码不应该被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块不应该被修改
- 开放：对现有功能的扩展开放

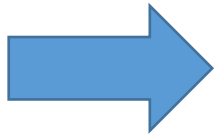
语法糖-装饰器-不推荐用法2

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式2

```
def login():
    print("come to login...")

def func1():
    print("come to func1...")

def login_func1():
    login()
    func1()
    print("come to func1...")

login_func1()
login_func2()
login_func3()
login_func4()
```

模块本身没有动，但需要修改调用该模块的方式（func1 -> login_func1）。同样违反了软件开发中的一个原则“开放-封闭”原则。

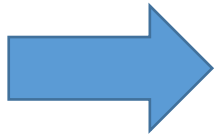
语法糖-装饰器-常规用法

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式3

```
def login():
    print("come to login...")

def func1_nologin():
    print("come to func1...")

def func1():
    login()
    func1_nologin()

func1()
func2()
func3()
func4()
```

工程上，可考虑使用的方式，c/c++可考虑此方法

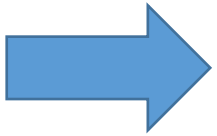
语法糖-装饰器-python推荐用法

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
增加登录功能



方式4

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

def func1():
    print("come to func1...")

func1 = login(func1)

def func2():
    print("come to func2...")

func2 = login(func2)

func1()
func2()
```

方式5

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

@login
def func1():
    print("come to func1...")

@login
def func2():
    print("come to func2...")

func1()
func2()
```

课堂练习（1）- 打印时间

```
# 示例1：
# 给调用的函数 添加 时间戳语句 -> 进入时打印时间，函数执行完成时，也打印时间。

# import time
#
# def timer(func):
#     def inner():
#         print("come to func ... ", time.time())
#         func()
#         print("end func ... ", time.time())
#     return inner
#
# @timer
# def func1():
#     print("func1....")
#
# @timer
# def func2():
#     print("func2....")
#
#
# func1()  -> logger(func1)
# func2()  -> logger(func2)
```

课堂练习（2）-给函数添加运行日志 用装饰器来修饰带参数的函数

```
# 示例2：  给函数添加运行日志 用装饰器来修饰带参数的函数.
# 这里需要使用到了不定参数 *args, **kwargs
# def logger(func):
#     def inner(*args, **kwargs):
#         print("come to func ...")
#         ret = func(*args, **kwargs)
#         print("end func ...")
#         return ret
#     return inner
#
# @logger
# def squre(x):
#     return x * x
#
# @logger
# def rect(x,y):
#     return x * y
#
# @logger
# def circle(r):
#     return 3.1415926 * r * r
#
# print(squre(10)) # -> login(squre)(10) -> inner(10)
# print(rect(10, 20))
# print(circle(10))
#
```

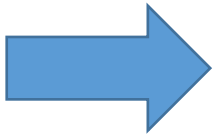
语法糖-装饰器-带参数运行

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

方式5

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

@login
def func1():
    print("come to func1...")

@login
def func2():
    print("come to func2...")

func1()
func2()
```

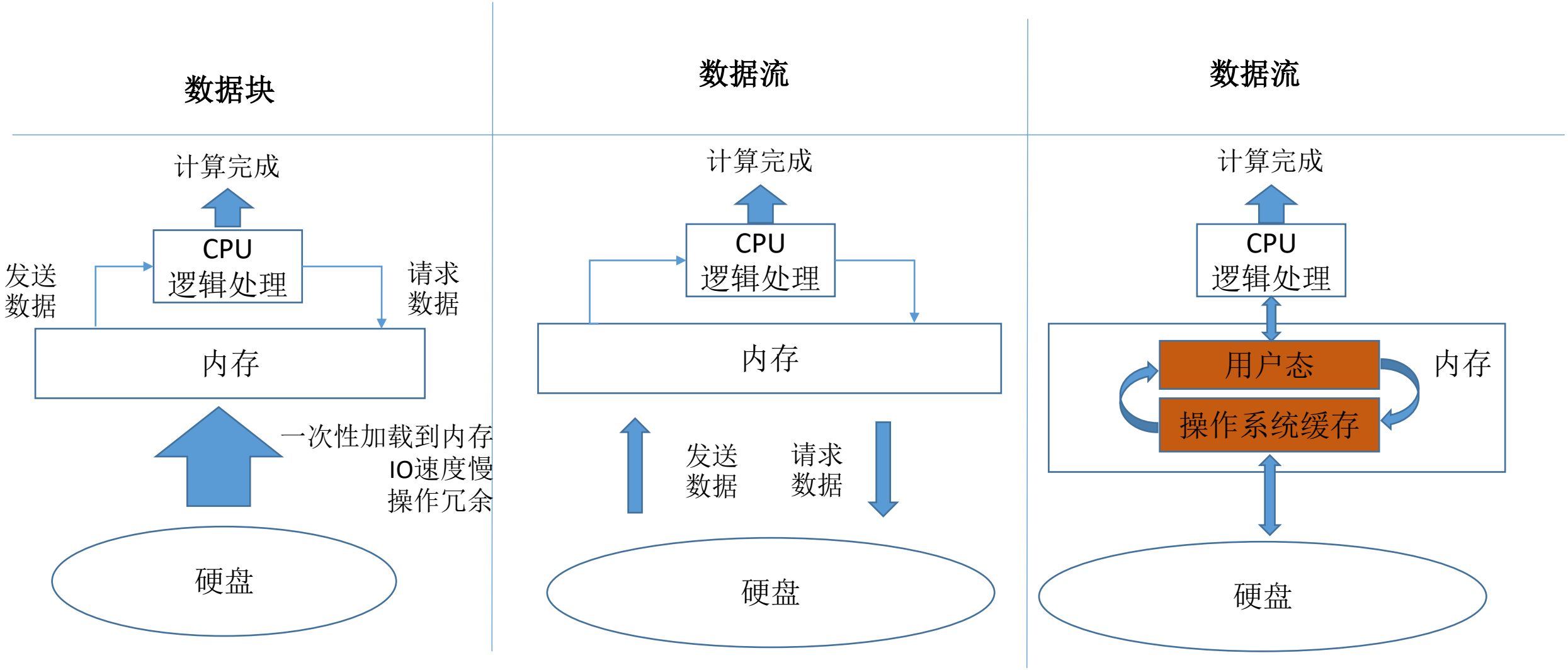
带参数运行



```
def login(func):
    def inter():
        print("come to login...")
        func()
    return inter

@login
def func1(para1):
    print("come to func1...")

@login
def func2(para1, para2):
    print("come to func2...")

@login
def func3(para1, para2, para3):
    print("come to func3...")

@login
def func4(para1, para2, para3,
para4):
    print("come to func4...")

func1(1)
func2(1,2)
func3(1,2,3)
func4(1,2,3,4)
```

```
def login(func):
    def inter(*args, **kwargs):
        print("come to login...")
        func(*args, **kwargs)
    return inter

@login
def func1(para1):
    print("come to func1...", para1)

等价于： login(func1)(para1)

@login
def func2(para1, para2):
    print("come to func2...", para1, para2)

@login
def func3(para1, para2, para3):
    print("come to func3...", para1, para2, para3)

@login
def func4(para1, para2, para3, para4):
    print("come to func4...", para1, para2, para3,
para4)

func1(1)
func2(1,2)
func3(1,2,3)
func4(1,2,3,4)
```

语法糖-生成器


数据块
计算完成
CPU
逻辑处理
发送
数据
请求
数据
内存
一次性加载到内存
IO速度慢
操作冗余
硬盘
数据流
计算完成
CPU
逻辑处理
内存
发送
数据
请求
数据
硬盘
数据流
计算完成
CPU
逻辑处理
用户态
操作系统缓存
内存
硬盘

语法糖-生成器

数据块	数据流
把数据看成数据块，一次性读入到内存，直接使用， 解析和导入时间长，内存压力大 大文件不适用	把硬盘上的数据看成流，一边读入，一边处理，直到逻辑结束
离线视频文件，必须全部下载完成才能看	在线视频，边下载边看。
提前生成数据，用于后续调用	通过generator来生成，用时再生成并调用

语法糖-生成器generator

```
for i in range(100000000):
    if i == 50:
        break
    print(i)
```

问题：

创建一个包含10000万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

怎么办？

```
xr = (x * x for x in range(10))
print(type(xr))
print(xr)
for x in xr:
    print(x)
```

问题：

生成器能否转化成 list or tuple or set?

注意：

python3中，range()已经做了优化

语法糖-可迭代对象 and 迭代器 and 生成器

- 可以直接作用于for循环的对象统称为可迭代对象（Iterable，可遍历、可循环），可以使用isinstance()判断一个对象是否是iterable对象，主要分为两大类：
  - 集合数据类型，如list、tuple、dict、set、str等；
  - generator，包括生成器和带yield的generator function。
- 被next()函数调用并不断返回下一个值的对象称为迭代器：iterator，所有生成器都是Iterator对象。
- list、dict、str虽然是iterable，却不是iterator，但可以通过iter()函数变成iterator。
- 为啥要引入iterator对象？
  - 通过iterator，python就可以处理长度不定（或无限大）的有序序列数据流，处理中，通过不断通过next()函数实现按需计算下一个数据，直到抛出StopIteration异常。
  - Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的（）。
  - Iterator可以处理硬盘数据的读取（长度未知）。
  - Iterator可以读取socket传送的网络报文数据流（长度未知）。

```
x = iter(range(1, 100))

for i in x:
    print(i)

while True:
    try:
        i = next(x)
        print(i)
    except StopIteration:
        break
```

课堂练习（2）

1. 利用装饰器， 添加函数的开始时间，和结束时间
2. 利用装饰器， 实现函数调用前的登录验证（只需要登录1次）。

课堂练习（3）-登录验证

```
# # 示例3， 登录验证
# login_status = False
# def login(func):
#     def inner(*args, **kwargs):
#         global login_status
#         if not login_status:
#             print("come to login...")
#             login_status = True
#         print("come to func...")
#         ret = func(*args, **kwargs)
#         print("end func ...")
#         return ret
#     return inner
#
# @login
# def func1():
#     print("func1...")
#
# @login
# def func2():
#     print("func2...")
#
#
# func1()
# func2()
```

异常处理

异常就是程序运行时发生错误（包括语法错误、逻辑错误）的信号，若程序产生了异常信号，且没有处理它，则会抛出该异常，程序的运行也随之终止。常见的错误异常类型：

- IndentationError 语法错误（的子类）；代码没有正确对齐
- SyntaxError Python代码非法，代码不能编译
- AttributeError 试图访问一个对象没有的树形，比如foo.x，但是foo没有属性x
- IOError 输入/输出异常；基本上是无法打开文件
- ImportError 无法引入模块或包；基本上是路径问题或名称错误
- IndexError 下标索引超出序列边界，比如当x只有三个元素，却试图访问x[5]
- KeyError 试图访问字典里不存在的键
- KeyboardInterrupt Ctrl+C被按下
- NameError 使用一个还未被赋予对象的变量
- TypeError 传入对象类型与要求的不符合
- ValueError 传入一个调用者不期望的值，即使值的类型是正确的

异常处理：几种典型用法

```
try:
    raise TypeError('类型错误')
except Exception as e:
    print(e)
```

```
s1 = 'hello'
try:
    int(s1)
    print( 'a' )
except IndexError as e:
    print(e)
except KeyError as e:
    print(e)
except ValueError as e:
    print(e)
except Exception as e:
    print(e)
```

```
s1 = 'hello'
try:
    int(s1)
except IndexError as e:
    print(e)
except KeyError as e:
    print(e)
except ValueError as e:
    print(e)
#except Exception as e:
#    print(e)
else:
    print( 'try内代码块没有异常则执行我' )
finally:
    print('无论异常与否,都会执行该我')
```

```
class EgonException(BaseException):
    def __init__(self, msg):
        self.msg = msg
    def __str__(self):
        return self.msg
try:
    raise EgonException('类型错误')
except EgonException as e:
    print(e)
```

- 把错误处理和真正的工作分开来
- 代码更易组织，更清晰，复杂的工作任务更容易实现；
- 毫无疑问，更安全了，不至于由于一些小的疏忽而使程序意外崩溃了；

课堂练习（5）-异常处理-实验1

```
# i = 1
# a = []
# b = []
# try:
#     while True:
#         j = i + 1
#         i += 1
#         # a = int('aaa')
#         if i >= 10:
#             break
#         raise ValueError("haaha")
# except NameError as e:
#     print(e)
# except IndentationError as e:
#     print(e)
# except ValueError as e:
#     print(e)
# else:
#     print("没有异常！")
# finally:
#     print("有没有异常都会执行到我！")
# print(i)
#
```

课堂练习（3）- 异常练习

- 用异常来重新封装一下文件copy 程序。
- 编写函数, 接收一个列表(包含30个1~100之间的随机整形数)和一个整形数k, 返回一个新列表.

  函数需求:

  将列表下标k之前对应(不包含k)的元素逆序;

  将下标k及之后的元素逆序;

  例如：

  输入两个参数：[1,2,3,4,5] 2， 返回结果： [2,1,5,4,3]

  注意：异常处理：

  （1）输入非列表

  （2）k不在范围之内

课堂练习（6）-异常处理-实验2-文件测试

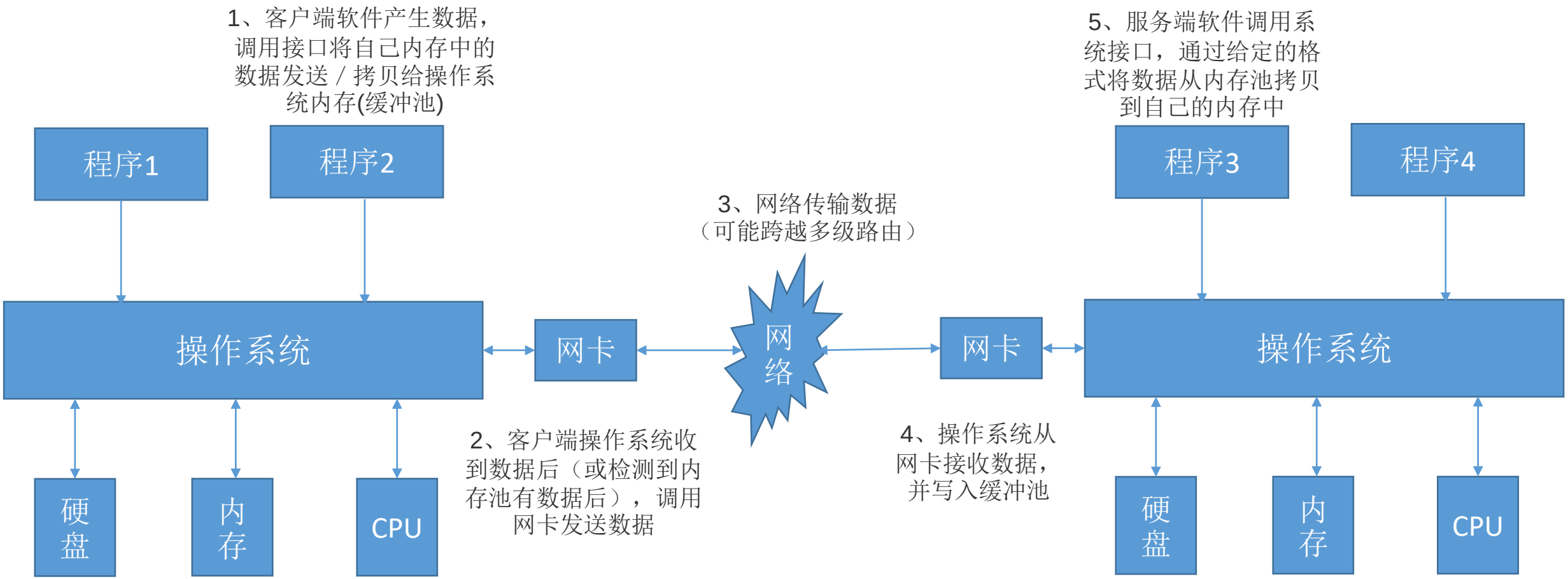
```
# def read_file(file_name):
#     try:
#         f = open(file_name, 'rb')
#         cc = f.read()
#         print(cc)
#         f.close()
#     except FileNotFoundError as e:
#         # add your err del code!
#         print(e)
#
# read_file('abb.txt')


# def read_file(file_name):
#     with open(file_name, 'rb') as f:
#         print(f)
#         cc = f.read()
#         print(cc)
#
# read_file('abb.txt')
```

网络编程

网络编程的重点：

- 理解网络通讯分层协议模型
- 理解socket概念
- 掌握使用python实现socket编程
- 了解TCP原理、粘包问题
- 模拟QQ，模拟FTP，模拟telnet

计算机网络通讯基本流程



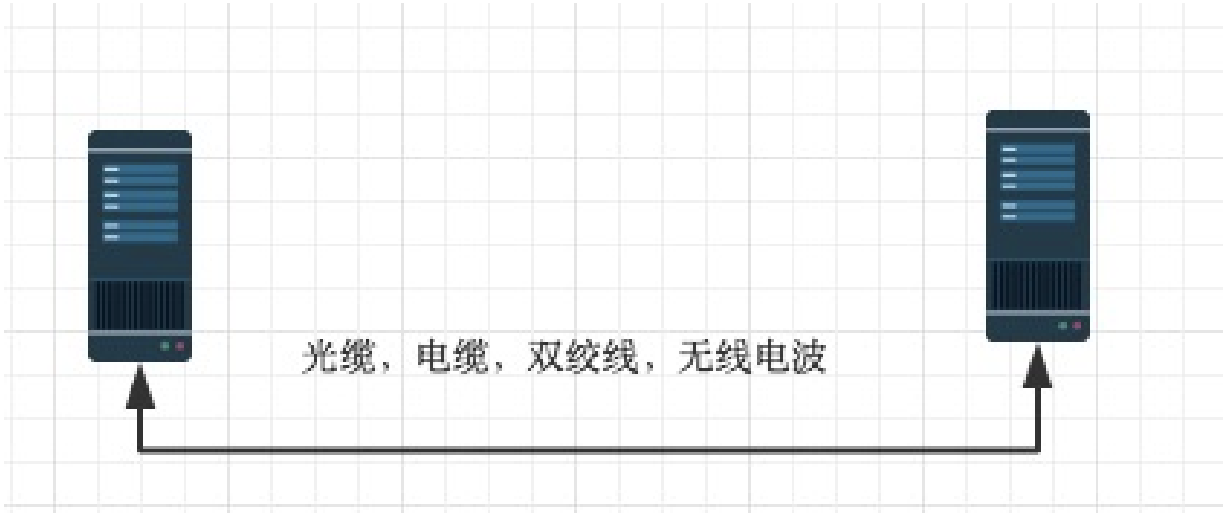
网络通讯分层协议模型

OSI层	功能	协议	TCP/IP五层协议
应用层 (Application layer)	文件传输、电子邮件、文件服务，虚拟终端等	TFTP/HTTP/FTP/SMTP/ DNS/telnet等	应用层 (Application layer)
表示层 (Presentation layer)	数据格式化、代码转换 数据加密	无	
会话层 (Session layer)	解决或建立与其他节点的联系	无	
传输层 (Transport layer)	提供端对端的接口(port)	TCP/UDP	传输层 (Transport layer)
网络层 (NetWork layer)	为数据包选择路由(ip)	IP/ICMP/ARP/RARP	网络层 (NetWork layer)
数据链路层 (Data link layer)	传输带地址的帧、错误监测功能。	Ethernet	数据链路层 (Data link layer)
物理层 (Physical layer)	以二进制形式在物理媒体上传输数据	RS-232、RS-449、V.35、 RJ-45	物理层 (Physical layer)

网络通讯分层协议模型

第1层：物理层（PhysicalLayer)（中继器、集线器、双绞线等，2进制数据）：

- 规定通信设备的机械的、电气的、功能的和过程的特性，用以建立、维护和拆除物理链路连接。在这一层，数据的单位称为比特（bit）。属于物理层定义的典型规范代表包括：EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45等。

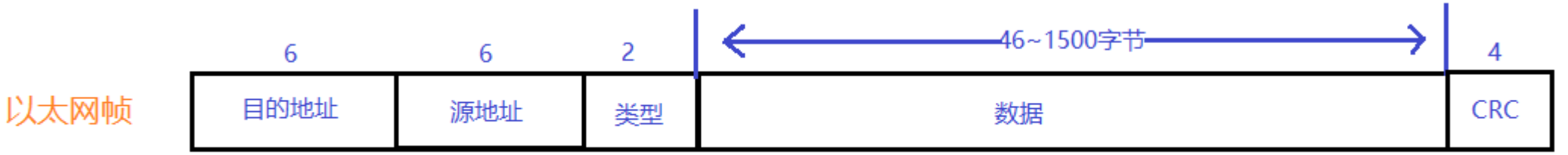


- 主要是基于电器特性发送高低电压(电信号)，高电压对应数字1，低电压对应数字0

网络通讯分层协议模型

第2层：链路层（Data link layer)（网卡，数据帧，ethernet协议）：

- 数据的单位称为帧（frame），通过差错控制提供数据帧在信道上无差错的传输，该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等，每一帧的数据帧构成：报头head和数据data。



- 目的地址与源地址是 MAC 地址。每一个网卡对应唯一的一个 MAC 地址，长度是 48 位。
- 在以太网帧的最后，还有一个 CRC 校验码，来校验数据是否异常。
- 在中间，有一个两个字节的类型标识。这个类型字段有三种值，分别是： IP、ARP、RARP。

IP协议：类型码为 0800， 数据链路层解包完毕后，将该数据交付给网络层的 IP 协议来处理该报文。

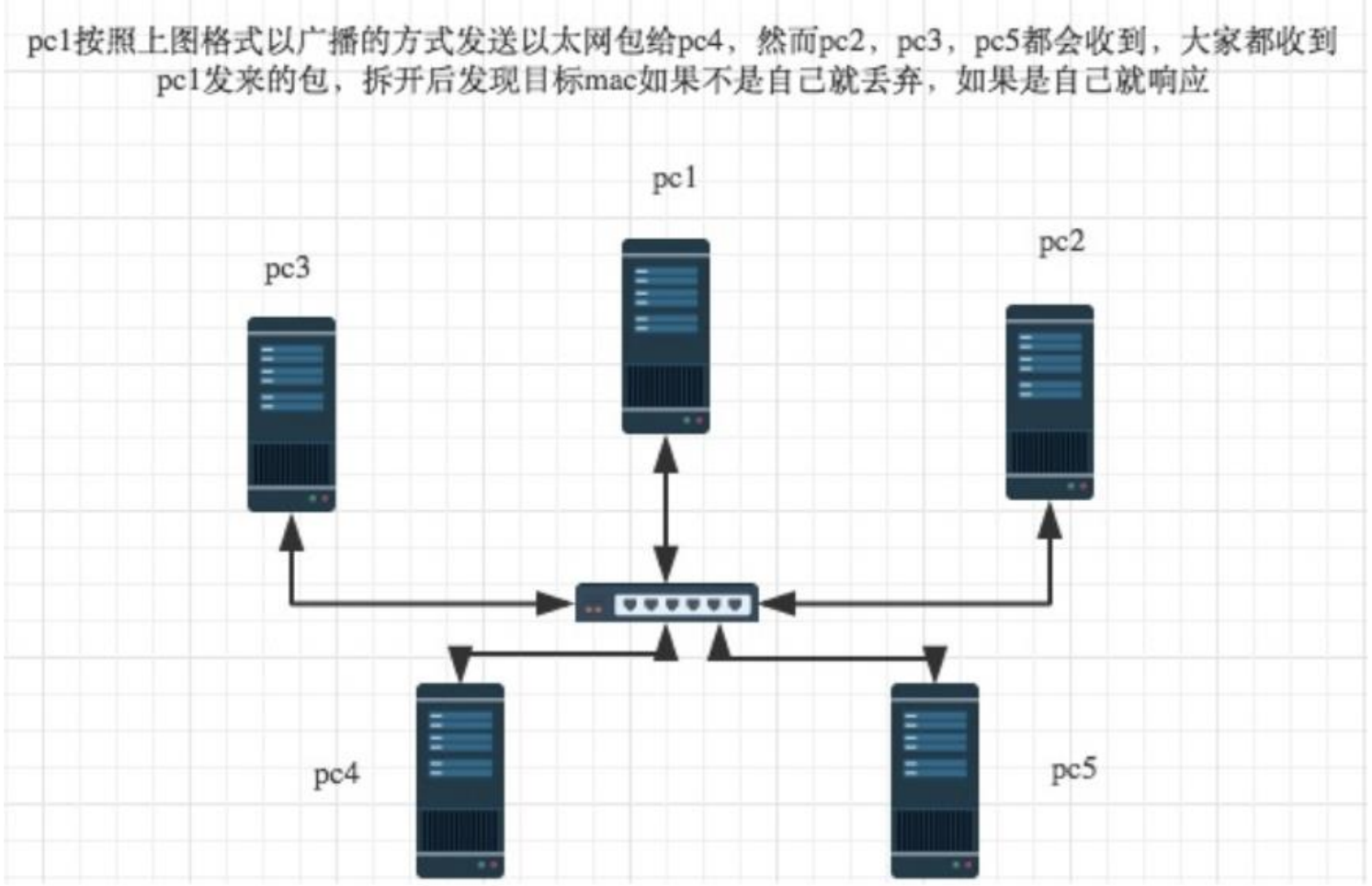
ARP协议：类型码0806 ，处于数据链路层与网络层之间的协议，也叫作地址解析协议，将 IP 地址转换为 MAC 地址。

IP协议：类型码为 0835 ， Reverse ARP，类似于 ARP，是将 MAC 地址转换为 IP 地址的协议。

网络通讯分层协议模型

有了mac地址，同一网络内的两台主机就可以通信了（一台主机通过arp协议获取另外一台主机的mac地址）

ethernet采用最原始的方式，广播的方式进行通信，即计算机通信基本靠吼。

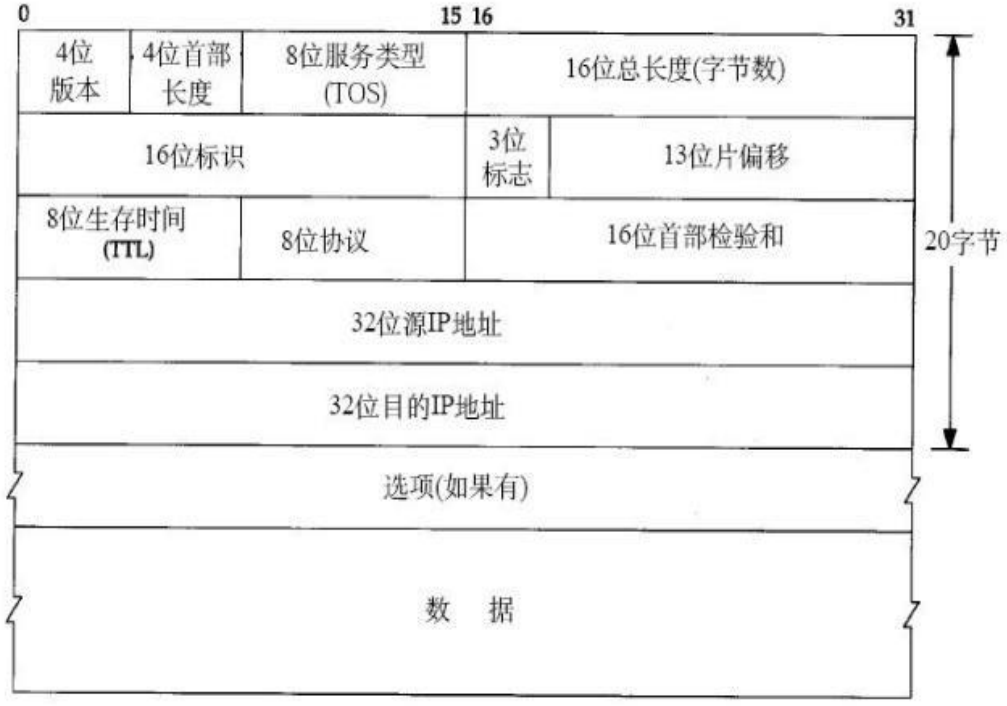


- 有了ethernet、mac地址、广播的发送方式，世界上的计算机就可以彼此通信了。
- 问题是世界范围的互联网是由一个个彼此隔离的小的局域网组成的，那么如果所有的通信都采用以太网的广播方式，那么一台机器发送的包全世界都会收到。
- 这就不仅仅是效率低的问题了，这会是一种灾难

网络通讯分层协议模型

第3层：网络层（Network layer)（主机、路由器，数据包，IP/ICMP/ARP/RARP协议）：

- 把传输层产生的报文段或用户数据封装成分组或包进行传送。在TCP/IP体系中，由于网络层使用IP协议，因此分组也叫做IP数据包，或简称为数据包。
- 选中合适的路由，使源主机运输层所传下来的分组，能够通过网络中的路由器找到目的主机，引入一套新的地址用来区分不同的广播域 / 子网，这套地址即网络地址。

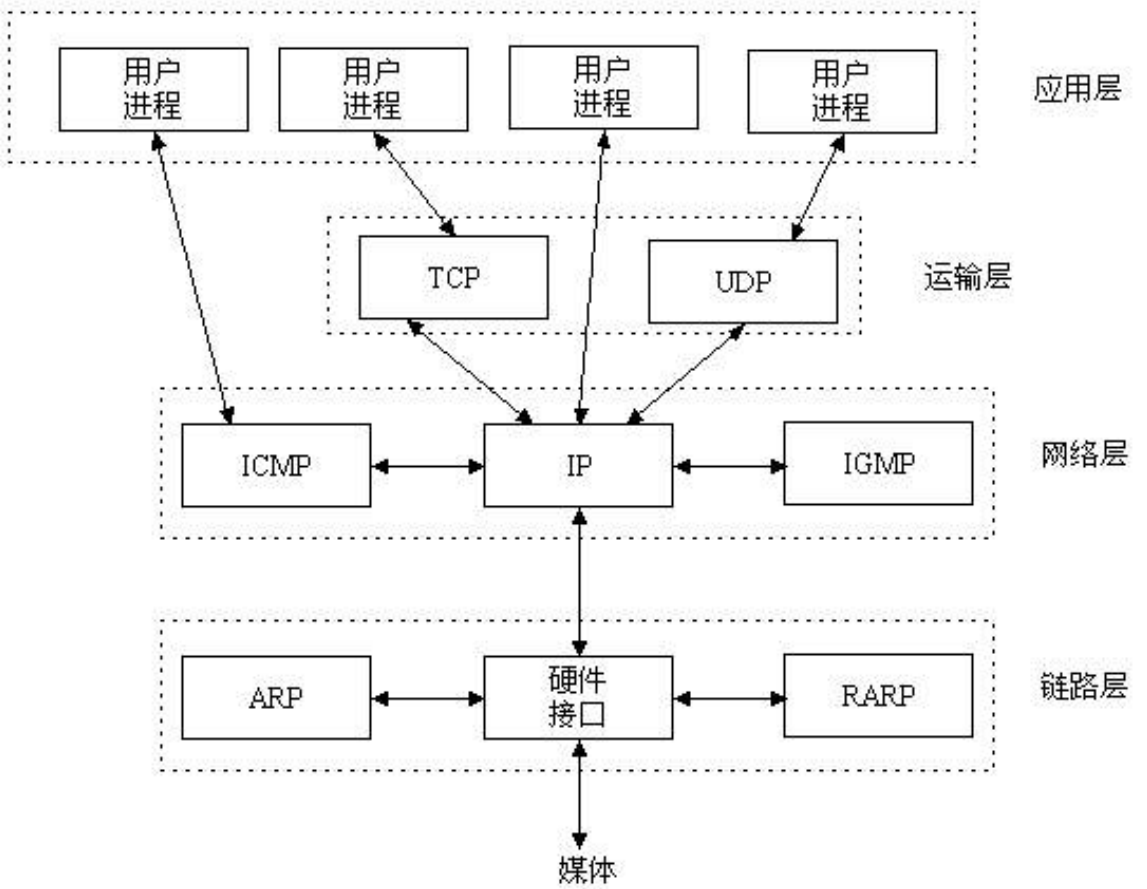


- ip协议规定网络地址，称之为ip地址，广泛采用的v4版本即ipv4
- 子网掩码是用来标识一个IP地址的哪些位是代表网络位，以及哪些位是代表主机位。
- IP协议是TCP/IP协议的核心，所有的TCP，UDP，IMCP，IGCP的数据都以IP数据格式传输
- ARP协议功能：广播发送数据包，获取目标主机的mac地址
- ICMP协议将会把错误信息封包，然后传送回给主机。给主机一个处理错误的机会（ping命令）

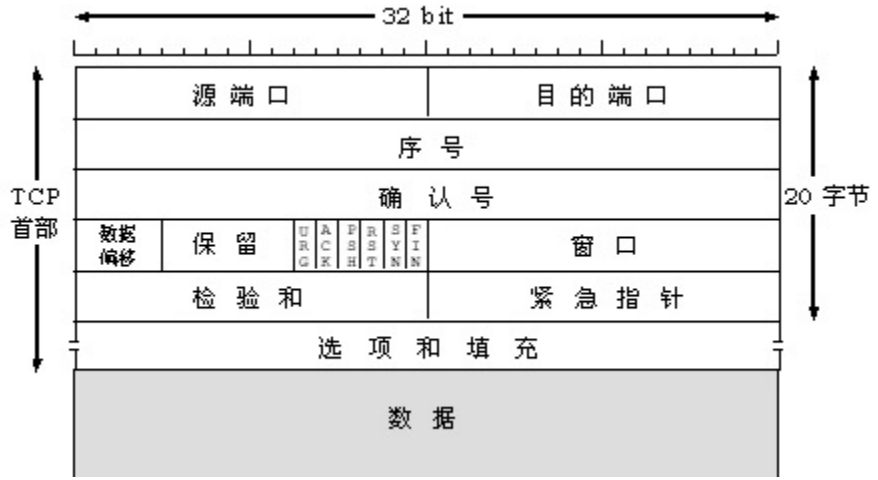
网络通讯分层协议模型

第4层：传输层（Transport layer)（应用程序，端口，TCP/UDP）：

- 网络层的ip帮我们区分子网，以太网层的mac帮我们找到主机，如何标识主机上的应用程序？
- 端口：应用程序与网卡关联的编号，传输层建立端口到端口的通信，端口范围0-65535，0-1023系统占用。



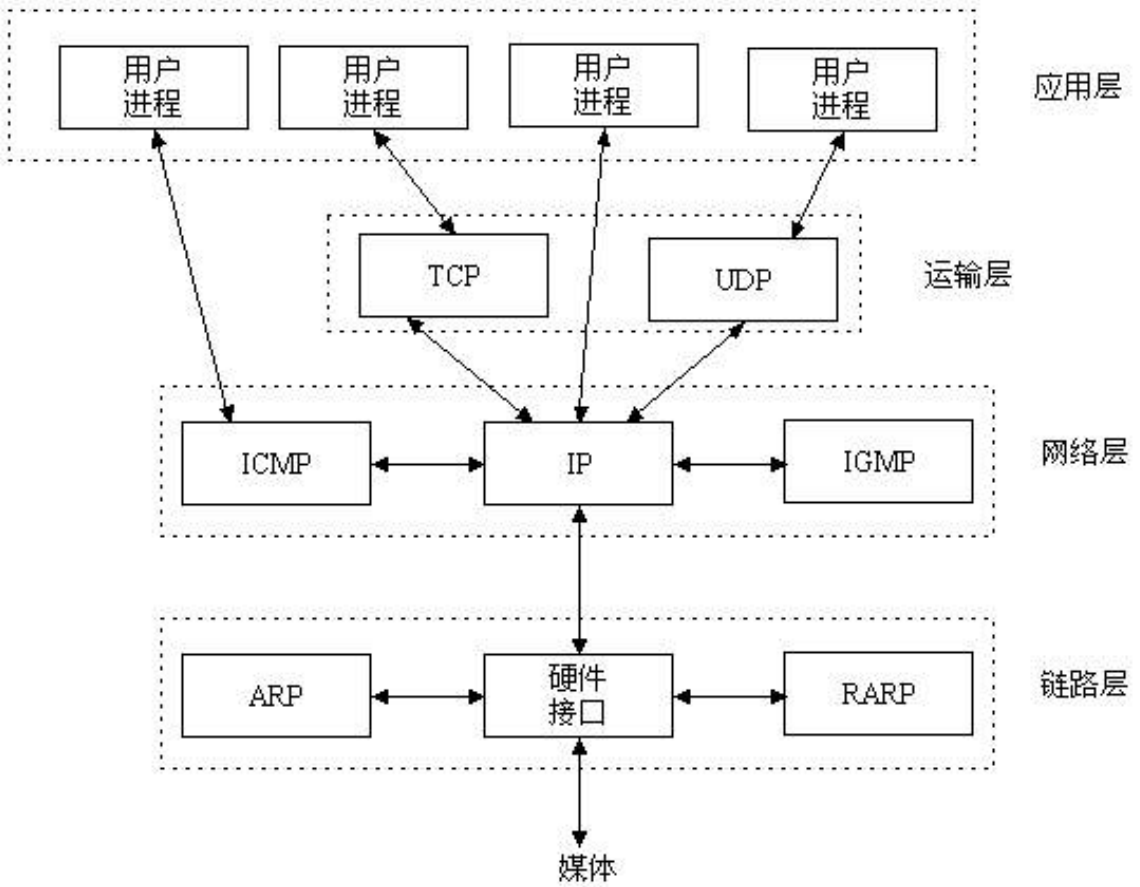
TCP可靠传输，TCP数据包没有长度限制，但是为了保证网络的效率，通常TCP数据包的长度不会超过IP数据包的长度，以确保不分包。所谓可靠传输，**只要不得到确认，就重新发送数据报，直到得到对方的确认为止。**



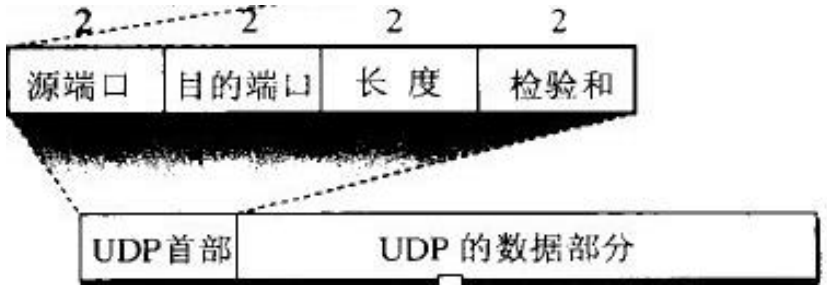
网络通讯分层协议模型

第4层：传输层（Transport layer)（应用程序，端口，TCP/UDP）：

- 网络层的ip帮我们区分子网，以太网层的mac帮我们找到主机，如何标识主机上的应用程序？
- 端口：应用程序与网卡关联的编号，传输层建立端口到端口的通信，端口范围0-65535，0-1023系统占用。

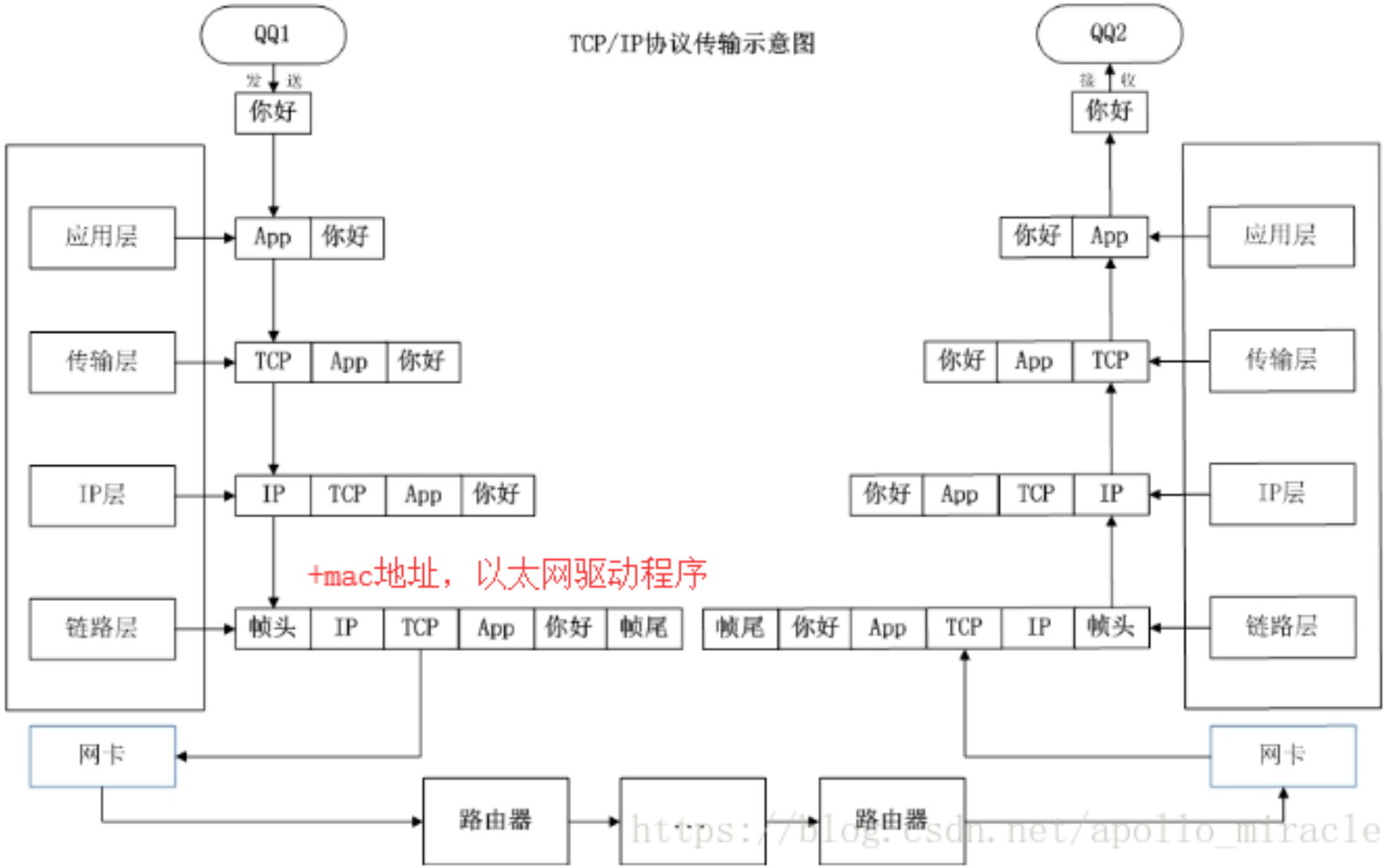


udp协议，不可靠传输，”报头”部分一共只有8个字节，总长度不超过65,535字节，正好放进一个IP数据包。



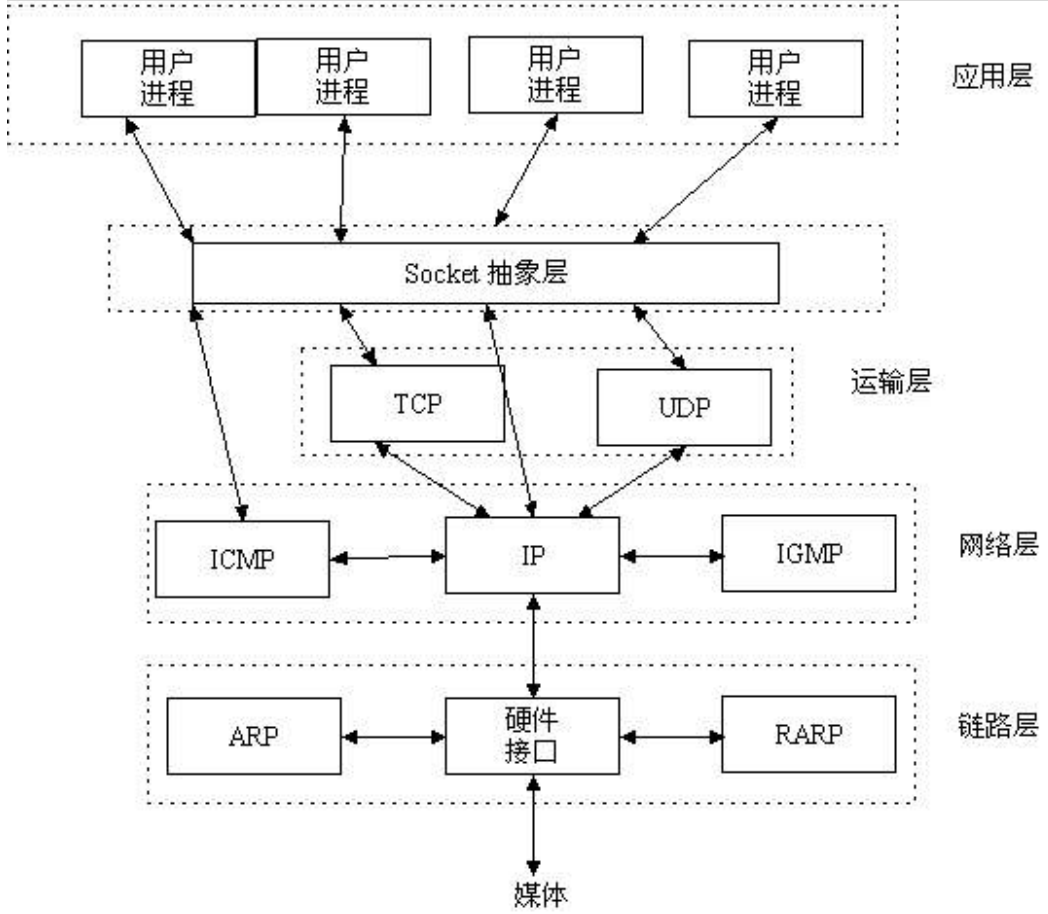
TCP协议虽然安全性很高，但是网络开销大，而UDP协议虽然没有提供安全机制，但是网络开销小。

网络通讯分层协议模型


TCP/IP协议传输示意图
QQ1
发送
你好
应用层
传输层
IP层
链路层
App
你好
TCP
App
你好
IP
TCP
App
你好
帧头
IP
TCP
App
你好
帧尾
+mac地址，以太网驱动程序
网卡
路由器
...
路由器
网卡
QQ2
接收
你好
你好
App
应用层
你好
App
TCP
传输层
你好
App
TCP
IP
IP层
帧尾
你好
App
TCP
IP
帧头
链路层
https://blog.csdn.net/apollo_miracle

网络编程socket

- 假设我现在要写一个程序通过TCP/IP与另外一台服务器上的服务端程序通讯，我应该怎么操作才能把数据封装成tcp/ip的包，又执行什么指令才能把数据发到对端机器上呢?
- Socket：数据封装、数据发送、数据接收、数据解包，只需要调用几行代码，就可以完成通信功能。

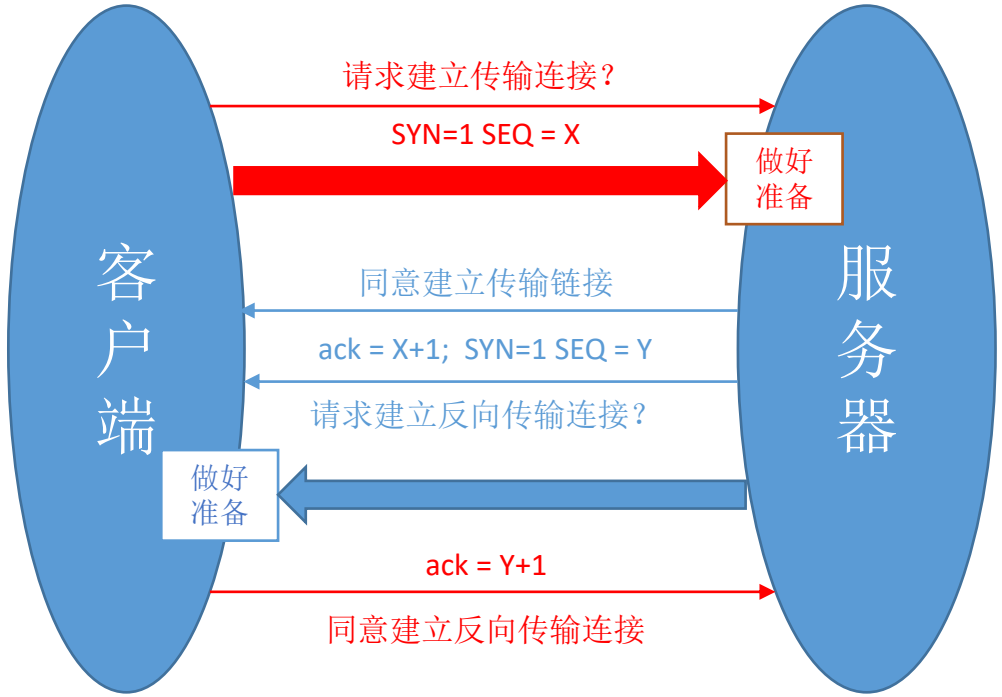


Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

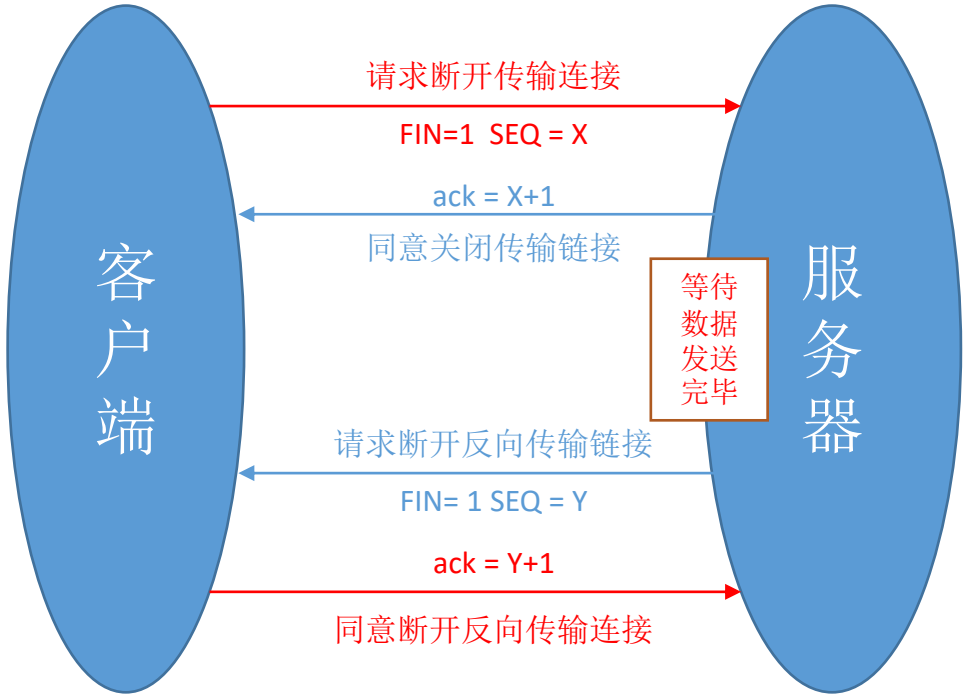
Socket最初是加利福尼亚大学Berkeley分校为Unix系统开发的本机进程通信接口。随着互联网发展，Socket成为在Internet上进行应用开发最为通用的API，后来又扩展到Windows下的网络编程接口。

网络编程socket-tcp

双向数据流 可靠传输机制



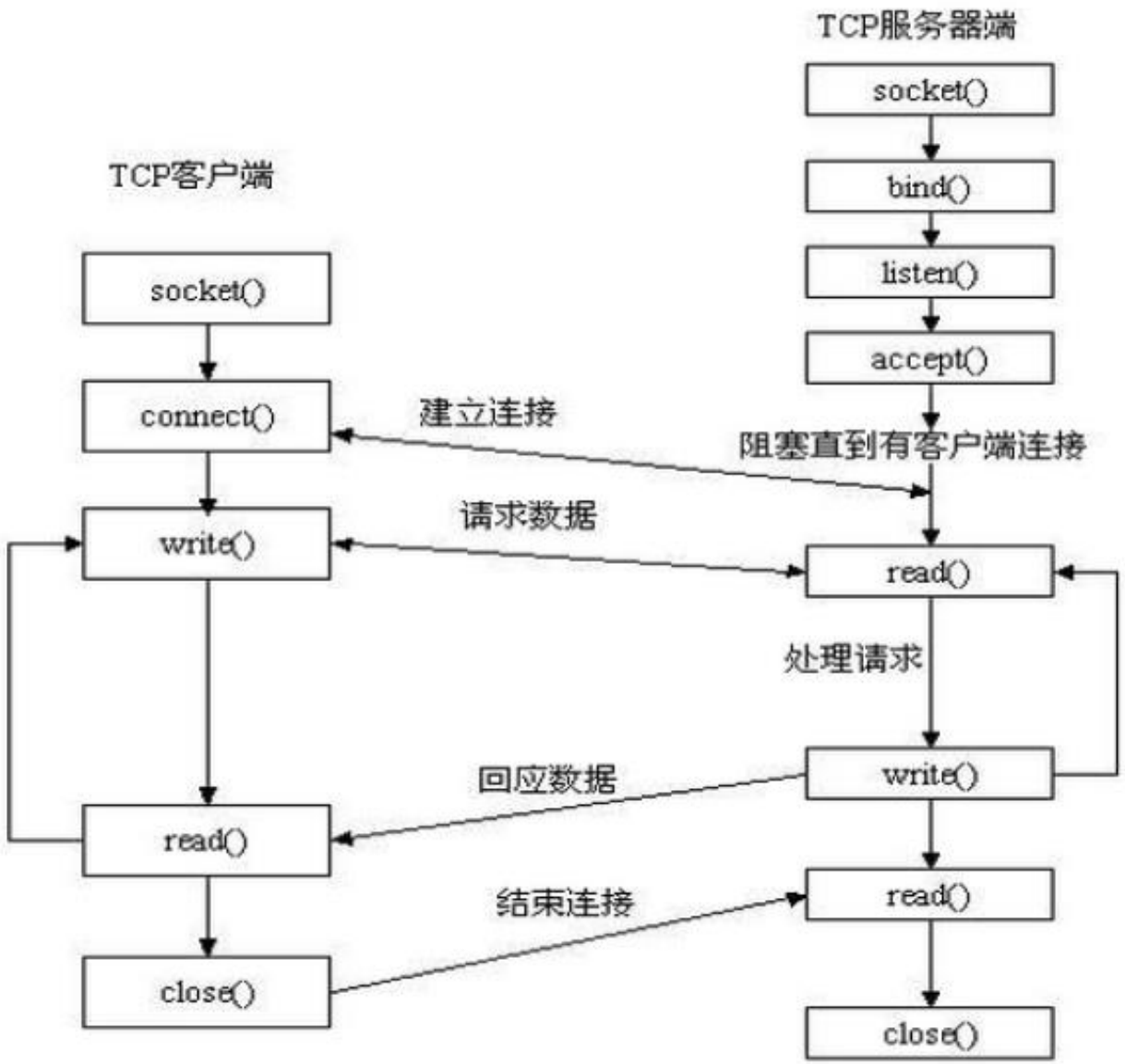
TCP握手的3个过程



TCP挥手的4个过程

为什么要4次挥手？确保数据能够完整传输。

网络编程socket-tcp过程



本质上socket只干2件事，一是收数据，一是发数据，没数据时就等着。类比于打电话为例：

接电话方(服务器端)：

- 首先你得有个电话\(生成socket对象\)
- 你的电话要有号码\(绑定本机ip+port\)
- 开始在家等电话\(开始监听电话listen\)
- 电话铃响了，接起电话，听到对方的声音\(accept\)
- 等待→响应→等待(read/send/read)。。。。

打电话方(客户端)：

- 首先你得有个电话\(生成socket对象\)
- 输入你想拨打的电话\(connect 远程主机ip+port\)
- 等待对方接听
- say “您好，约饭吗？~”\(send\(\) 发消息。。。\)
- 等待→响应→等待(read/send/read)
- 挂电话(close)

网络编程socket-tcp示例

socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)

参数	取值	含义
Family (socket家族)	AF_UNIX	进程间通讯
	AF_INET	网络通讯
socket type类型	SOCK_STREAM	tcp
	SOCK_DGRAM	udp

```python
import socket
sock = socket.socket()
sock.connect((“127.0.0.1”, 8800))
while True:
    data = input("输入信息>>")
    sock.send(data.encode('utf-8'))
    data = sock.recv(1024)
    print(data)
sock.close()
```

```python
import socket

sock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
sock.bind(("127.0.0.1", 8800))
sock.listen(10)

while True:
    conn, status = sock.accept()
    print(conn, status)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        print(data.decode('utf-8'))
        data = input("输入消息 >>").encode('utf-8')
        conn.send(data)
    conn.close()
```

课堂练习（2）

1. 网络通讯，实现两人对话

课堂练习（2）

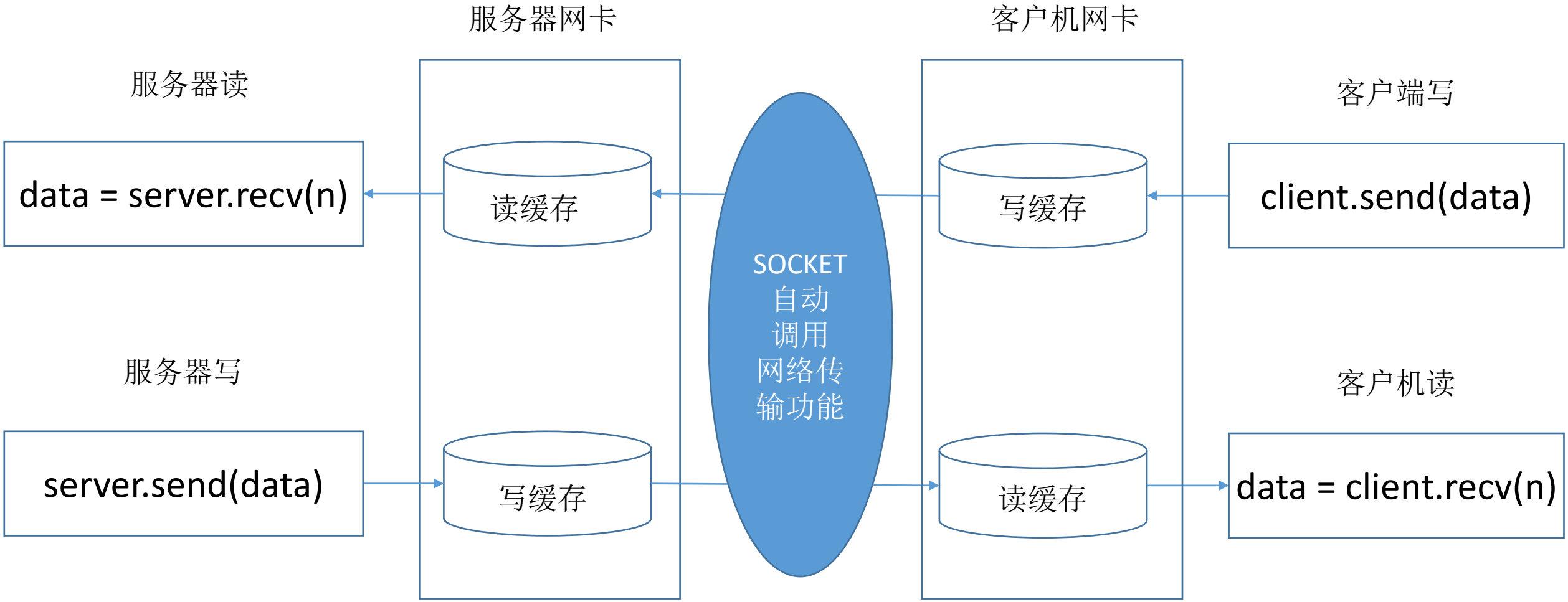
1. 网络通讯，实现两人对话

```python
import socket

sock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)

sock.bind(("127.0.0.1", 8888))
sock.listen(10)

conn, status = sock.accept()

while True:
    data_get = conn.recv(1024)
    print(data_get.decode('utf8'))
    data_send = input("请输入>>")
    conn.send(data_send.encode('utf8'))
```

服务端

```python
import socket

sock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)

sock.connect(("127.0.0.1", 8888))

while True:
    data_in = input("请输入:>>").strip()
    sock.send(data_in.encode('utf8'))
    data = sock.recv(1024)
    print(data.decode('utf8'))
    break

sock.close()
```

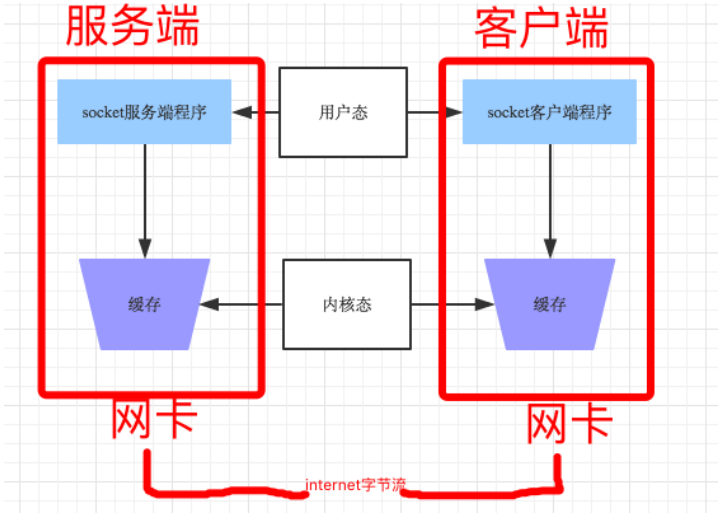
客户端



接收方不知道消息之间的界限，不知道一次性提取多少字节数据，就会造成粘包

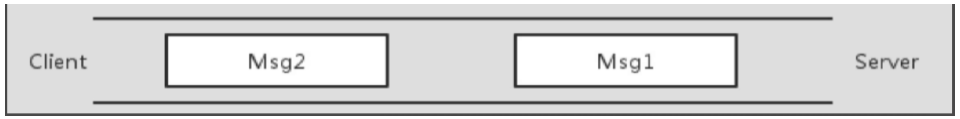
网络编程socket- TCP粘包

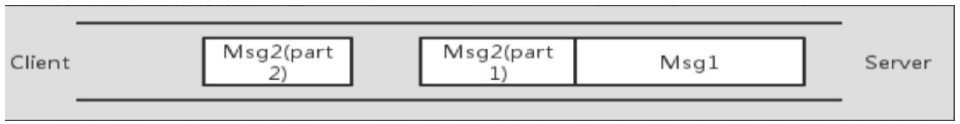
```python
import socket


sock = socket.socket()
sock.connect(("127.0.0.1", 8800))

import string
import time
while True:
    for i in string.ascii_letters:
        sock.send(str(i).encode('utf-8'))
        # time.sleep(0.001)
```

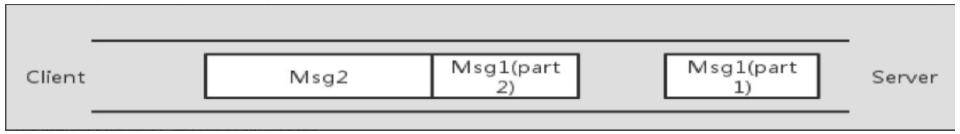


接收方不知道消息之间的界限，不知道一次性提取多少字节的数据所造成的

```python
import socket


sock = socket.socket()
sock.bind(("127.0.0.1", 8800))
sock.listen(10)

while True:
    conn, status = sock.accept()
    while True:
        try:
            data = conn.recv(1024)
        except ConnectionError:
            break
        if not data:
            break
        print(data.decode('utf-8'))
    conn.close()
```







网络编程socket- TCP粘包-解决办法

```python
import socket

sock = socket.socket()
sock.connect(("127.0.0.1", 8800))

str_cmd = """
邱勇认真倾听了每位同学的发言，他首先对同学们来到清华表示祝贺和欢迎。
邱勇说：“大家在发言中谈到，来到清华感觉一见如故，我见到各位新同学也有一见如故的感觉，祝贺大家在清华开启崭新的人生阶段，相信清华的生活会比你们想象中的还要丰富多彩！”
邱勇对新同学的未来发展寄予两点希望。
清华加油
"""
str_list = str_cmd.split('\n')
print(str_list)
import time
import json, struct
while True:
    for i in str_list:
        data_bytes = i.encode('utf-8')
        data_len = len(data_bytes)
        head_buf = {'len': data_len}
        head_bytes = json.dumps(head_buf).encode('utf-8')
        head_len_bytes = struct.pack('i', len(head_bytes))
        # print(head_len_bytes)
        sock.send(head_len_bytes)  # 先发报头的长度,4个bytes
        # print(head_bytes)
        sock.send(head_bytes)  # 再发报头的字节格式
        sock.sendall(data_bytes)  # 然后发真实内容的字节格式
        # time.sleep(1)
```

```python
import socket


sock = socket.socket()
sock.bind(("127.0.0.1", 8800))
sock.listen(10)

import json, struct
while True:
    conn, status = sock.accept()
    while True:
        try:
            data = conn.recv(4)
            # print(data)
            head_len = struct.unpack('i', data)
            # print(head_len)
            head_json = conn.recv(head_len[0]).decode('utf-8')
            # print(head_json)
            head_buf = json.loads(head_json)
            # print(head_buf)
            data_len = head_buf.get('len', 0)
            # print("data_len", data_len)
            if data_len == 0:
                data = ''
            else:
                data = conn.recv(data_len)
                # print(data)
                data = data.decode('utf-8')
                if not data:
                    break
        except ConnectionError:
            break
        print(data)
    conn.close()
```

客户机， 利用json 序列化复杂变量， 并利用struct 得到 报文长度， 先发送报文头，再发送正式报文。

```python
# 客户端程序
import socket

client = socket.socket()
client.connect(("127.0.0.1", 7700))

import time
import json
import struct
while True:

    li = [1,2,3, time.time(), {"a": 1, "b": 2}]
    str = json.dumps(li).encode("utf-8")

    # 首先得到报文的长度，并打成固定长度包并发送
    data = struct.pack("i", len(str))
    client.send(data)

    # 然后再发送数据
    client.send(str)
```

服务器，先recv到4个字节数据并解析，得到报文长度，再recv指定报文长度的数据，并利用json 反序列化，得到数据内容

```python
# 服务器程序
import socket

server = socket.socket()
server.bind(("127.0.0.1", 7700))
server.listen(10)
conn, addr = server.accept()
import time
import json
import struct
while True:

    # 首先获取报文头，得到报文体的长度
    data = conn.recv(4)
    head_len = struct.unpack('i', data)
    # print(head_len)

    # 首先再根据报文体的长度，获取并解析报文体数据。。。
    data = conn.recv(head_len[0])
    obj = json.loads(data.decode('utf-8'))
    print(obj)

    time.sleep(0.5)
```

课堂练习（2）

1. 网络通讯，实现文件传输（类似于FTP功能）

对象	功能点
客户端	1. 支持选择待传输文件 2. 支持显示进度条 3. 支持多次文件传输
服务器	1. 支持接收并存储文件

课堂练习（2）

1. 网络通讯，实现文件传输（类似于FTP功能）

```python
import socket
import struct
import json

sock = socket.socket()
sock.bind(("127.0.0.1", 8889))
sock.listen(10)

conn, status = sock.accept()
while True:
    head_recv = conn.recv(4)
    len = struct.unpack("i", head_recv)[0]

    head_recv = conn.recv(len)
    head_data = json.loads(head_recv)

    file_name, file_len = head_data['name'], head_data['size']
    with open(file_name, "wb") as fp:
        recv_len = 0
        while recv_len < file_len:
            this_len = 1024 if file_len - recv_len > 1024 else file_len - recv_len
            data = conn.recv(this_len)
            fp.write(data)
            recv_len += this_len
```

服务端

```python
sock = socket.socket()
sock.connect(("127.0.0.1", 8889))

while True:
    # 获取文件信息
    file_name = input("请输入文件名称").strip()
    data_len = os.path.getsize(file_name)
    data_head = {"name": "new_" + file_name, "size": data_len}
    json_head = json.dumps(data_head).encode('utf8')

    # 发送报文头
    json_head_lens = struct.pack("i", len(json_head))
    sock.send(json_head_lens)
    sock.send(json_head)

    # 发送文件内容
    with open(file_name, "rb") as fp:
        send_len = 0
        while True:
            data = fp.read(10240)
            send_len += len(data)
            if not data:
                break
            process = int(send_len * 100.0 / data_len)
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')
            sock.send(data)

    print("上传成功")
```

客户端