

电网SCADA原型开发

（后台通信）

# 课堂练习（1）模拟FTP

## 第1步， 客户端 put 文件， 服务端接收文件

```
def put(sock, file_name):
    data_len = os.path.getsize(file_name)
    data_head = {"name": file_name, "size": data_len, "cmd": "put"}
    json_head = json.dumps(data_head).encode('utf8')

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

client.py

```
try:
    while True:
        head_recv = conn.recv(4)
        head_len = struct.unpack("i", head_recv)[0]

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

        print("recv", head_data)

        file_cmd = head_data['cmd']

        func = globals().get(file_cmd, None)
        if func is None:
            raise ValueError(f"cmd error: {head_recv}")

        func(conn, head_data)
except Exception as e:
    print("Error:", e)
```

```
def put(conn, head_data):
    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 = 10240 if file_len - recv_len > 10240 else file_len - recv_len
            data = conn.recv(this_len)
            fp.write(data)
            recv_len += this_len
```

server.py

# 课堂练习（1） 模拟FTP

## 第2步， 客户端 get 文件， 服务端发送文件

```
def get(sock, file_name):  
    data_head = {"name": file_name, "cmd": "get"}  
    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)  
  
    # 接受报文头  
    head_recv = sock.recv(4)  
    head_len = struct.unpack("i", head_recv)[0]  
    head_recv = sock.recv(head_len)  
    head_data = json.loads(head_recv)  
    file_name, file_len = head_data['name'], head_data['size']  
  
    # 接受文件信息  
    with open(file_name, "wb") as fp:  
        recv_len = 0  
        while recv_len < file_len:  
            this_len = 10240 if file_len - recv_len > 10240 else file_len - recv_len  
            data = sock.recv(this_len)  
            fp.write(data)  
            fp.flush()  
            recv_len += this_len  
            process = int(recv_len * 100.0 / file_len)  
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')  
  
    print("\n下载成功")
```

client.py

```
try:  
    while True:  
        head_recv = conn.recv(4)  
        head_len = struct.unpack("i", head_recv)[0]  
  
        head_recv = conn.recv(head_len)  
        head_data = json.loads(head_recv)  
  
        print("recv", head_data)  
  
        file_cmd = head_data['cmd']  
  
        func = globals().get(file_cmd, None)  
        if func is None:  
            raise ValueError(f"cmd error: {head_recv}")  
  
        func(conn, head_data)  
  
except Exception as e:  
    print("Error:", e)
```

```
def get(conn, head_data):  
    # 获取文件信息  
    file_name = head_data['name']  
    file_len = os.path.getsize(file_name)  
    send_head = {"name": file_name, "size": file_len, "cmd": "get"}  
    json_head = json.dumps(send_head).encode('utf8')  
  
    # 发送报文头  
    json_head_lens = struct.pack("i", len(json_head))  
    conn.send(json_head_lens)  
    conn.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)  
            process = int(send_len * 100.0 / file_len)  
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')  
            if not data:  
                break  
            conn.send(data)  
    print("上传成功")
```

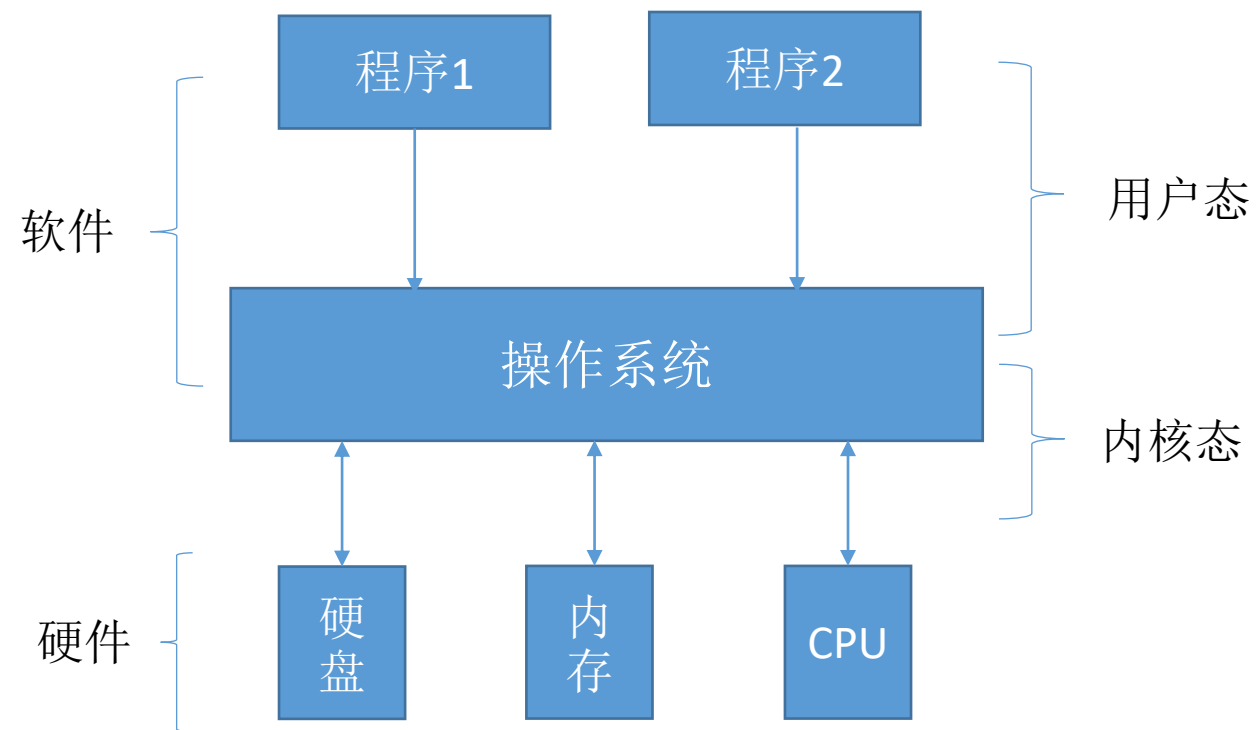
server.py

# 并发编程

## 并发编程的重点：

- 了解操作系统的基本概念及发展历史。
- 理解进程的概念，熟练掌握python进程
- 理解线程的概念，熟练掌握python线程。
- 理解进程与线程的主要区别。

# 操作系统的基本概念



操作系统位于计算机硬件与应用软件之间，本质也是一个软件。操作系统由操作系统的内核（运行于内核态，管理硬件资源）以及系统调用（运行于用户态，为应用程序员写的应用程序提供系统调用接口）两部分组成。

主要功能：

- 隐藏了丑陋的硬件调用接口，为应用程序员提供调用硬件资源的更好，更简单，更清晰的模型（系统调用接口）。应用程序员有了这些接口后，就不用再考虑操作硬件的细节，专心开发自己的应用程序即可。
- 将应用程序对硬件资源的竞态请求变得有序化。

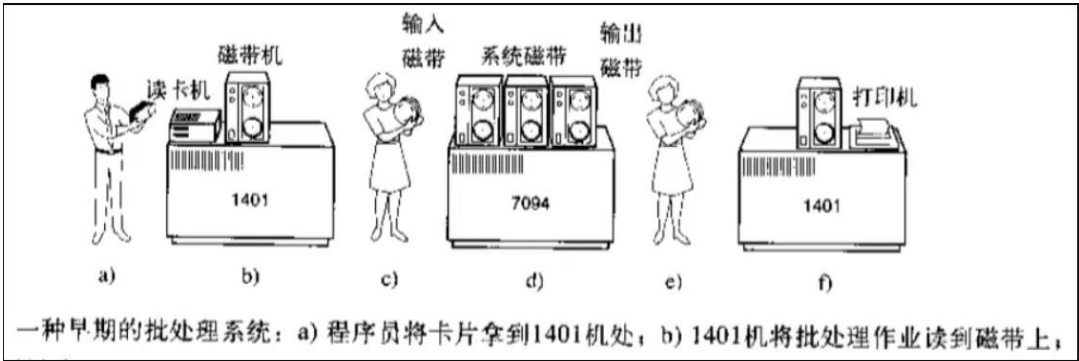
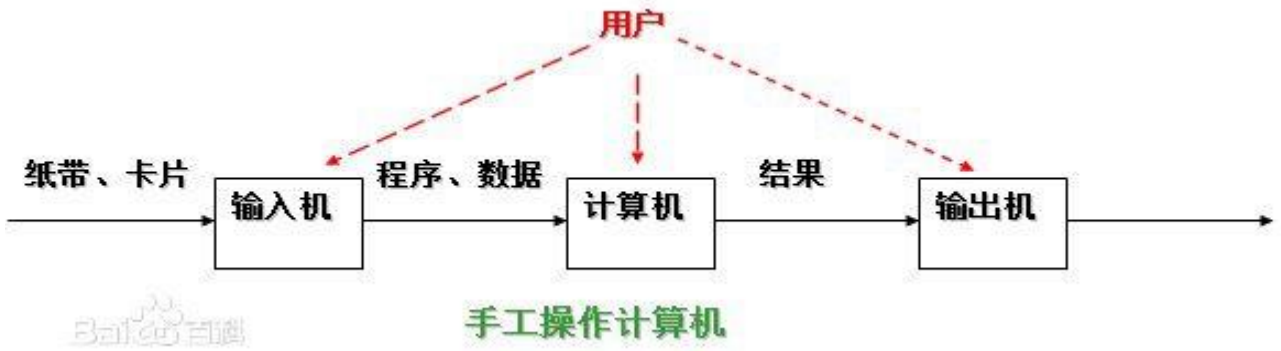
与普通软件的区别：

- 直接运行在“裸机”上的最基本软件，其他软件都必须在操作系统的支持下才能运行，由硬件保护，不能被用户修改。
- 协调管理计算机系统中各种独立的硬件，其他软件可不需要顾及到底层每个硬件是如何工作。

# 操作系统发展历史

## 1、20世纪50年代中期之前，采用手工操作方式：

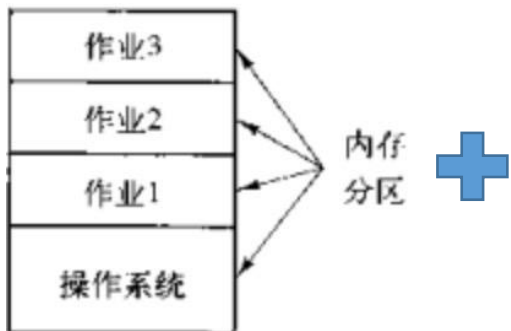
- 用户独占全机。不会出现因资源已被其他用户占用而等待的现象，但资源的利用率低。
- CPU 等待手工操作。CPU的利用不充分，手工操作的慢速度和计算机的高速度之间形成了尖锐矛盾。



## 3、批处理多道技术，引入多进程系统，通过多路利用解决多进程竞争或者说共享同一个资源（比如cpu）的有序调度问题，多路复用分为空间复用（内存分片）和时间上复用（进程切换）。这里，多道即为多进程。

## 2、20世纪50年代起，为了CPU使用效率，人们提出了批处理操作系统。用户（生成磁带）+计算机操作员（导出磁带），主要问题：

- 用户程序和监控程序的交替执行，使得一部分内存要交付给监控程序使用，监控程序消耗了一部分时间。
- 每次只能执行一道程序，I/O速度较处理器速度太慢，导致处理器空闲。



进程之间去争抢cpu的执行权限。这种切换不仅会在一个进程遇到io时进行，一个进程占用cpu时间过长也会被操作系统夺走cpu的执行权限

# 操作系统发展历史

第3代技术（批处理多道），存在的问题：

- 程序之间的内存必须分割，否则存在安全性和鲁棒稳定性的问题。这种分割需要在硬件层面实现，必须由操作系统控制。。
- 程序员原来独享一段时间的计算机，现在必须被统一规划到一批作业中，等待结果和重新调试的过程都需要等同批次的其他程序都运作完才可以（这极大的影响了程序的开发效率，无法及时调试程序）



分时操作系统：

- 多个联机终端
- 多道技术

分时操作系统将系统处理器时间与内存空间按一定的时间间隔，轮流地切换给各终端用户的程序使用。由于时间间隔很短，每个用户的感受就像他独占计算机一样。分时操作系统的特点是可有效增加资源的使用率。

- 1961年，CTSS兼容分时系统， IBM 709（最后一款电子管机） → IBM 7094
- 1965年，贝尔实验室和通用电气开发MULTICS，目标是满足波士顿地区所有用户需求，最终失败。
- 1969年，贝尔实验室Ken Thompson(UNIX之父) 简化了MULTIC，形成UNIX系统，并衍生相当多版本（IBM-AIX，HP-UX, SUN-Solaris, APPLE-MAC）
- 1985年，美国微软公司研发的WINDOWS操作系统，多用户多任务操作系统。
- 1991年，芬兰学生Linus（Linux之父）在minix(UNIX变种)的基础上，编写了Linux，开源的成功典型。



# 操作系统发展历史



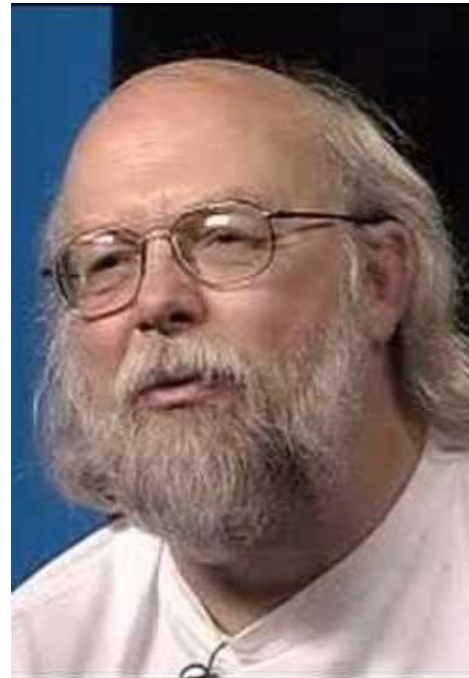
Ken Thompson  
UNIX之父  
(1943-)



Linus  
Linux之父  
(1969年-)



Dennis Ritchie  
C语言发明者  
(1941-2011)



James Gosling  
JAVA之父  
(1955-)



Guido Van Rossum  
Python之父  
(1956-)

论程序员职业选择与头发之间的关系。。。。

# 操作系统知识小结

即使可以利用的cpu只有一个（早期的计算机确实如此），也能保证支持（伪）并发的能力。将一个单独的cpu变成多个虚拟的cpu（多道技术：时间多路复用和空间多路复用+硬件上支持隔离），没有进程的抽象，现代计算机将不复存在。

## ■ 操作系统的作用：

- ✓ 隐藏丑陋复杂的硬件接口，提供良好的抽象接口
- ✓ 管理、调度进程，并且将多个进程对硬件的竞争变得有序

## ■ 多道技术：

- ✓ 产生背景：针对单核，实现并发。
- ✓ 现在的主机一般是多核，那么每个核都会利用多道技术。
- ✓ 空间上的复用：如内存中同时有多道程序
- ✓ 时间上的复用：复用一個cpu的时间片，遇到io切，占用cpu时间过长也切，核心在于切之前将进程的状态保存下来，这样才能保证下次切换回来时，能基于上次切走的位置继续运行

# 并发编程-进程

## ■ 进程与程序的区别：

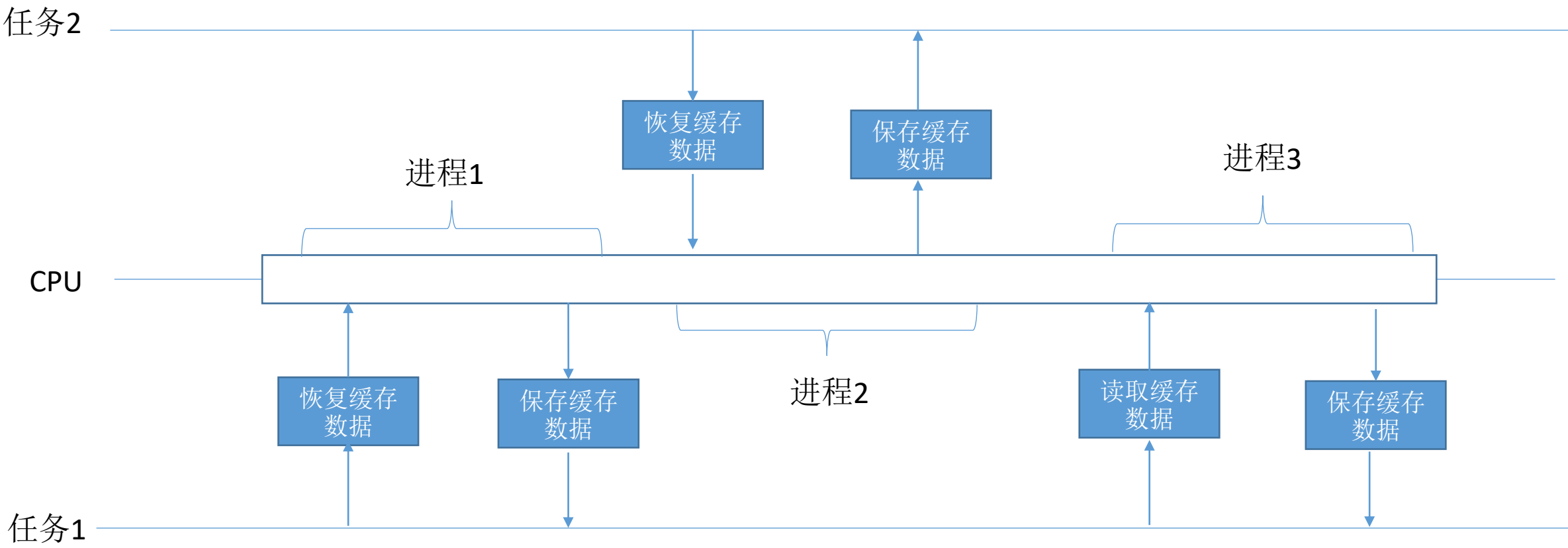
- ✓ 程序：一堆代码的集合，进程是操作系统调用程序执行的运行过程。
- ✓ 进程：正在进行的一个过程或者说一个任务。而负责执行任务则是cpu。
- ✓ 需要强调的是：同一个程序执行两次，那也是两个进程，比如打开暴风影音，虽然都是同一个软件，但是一个可以播放本地视频，一个可以播放网络视频。

## ■ 并发与并行的区别：

- ✓ 并发：伪并行，看起来是同时运行。单个cpu+多道技术就可以实现并发，实际上一个cpu在同一时刻只能执行一个任务。
- ✓ 并行：同时运行，只有具备多个cpu才能实现并行，
- ✓ 无论是并行还是并发，在用户看来都是‘同时’运行的，不管是进程还是线程，都只是一个任务而已，真是干活的是cpu，cpu来做这些任务，而一个cpu同一时刻只能执行一个任务。

思考：通过并行计算，一定可以提高计算速度吗？

# 并发编程-实现原理



进程并发：操作系统维护一张表格，即进程表（**process table**），每个进程占用一个进程表项（或进程控制块），存放进程状态信息：程序计数器、堆栈指针、内存分配状况、所有打开文件的状态，以及其他必须保存的信息，当状态切出时保存进程实时状态；当状态切入时再加载进程控制块，就像从未被中断过一样。

# 并发编程-进程

进程的创建：由一个已经存在的进程执行了系统调用命令而创建的新进程：

- 操作系统初始化
- 一个进程在运行过程中开启了子进程（如：`os.system`, `subprocess.Popen`等）
- 通过用户的交互式请求创建一个新进程：
  - ✓ 双击调用。
  - ✓ `cmd`窗口命令调用。

不同操作系统的进程调用方法：

- UNIX: `fork`。
- windows: `CreateProcess`, `CreateProcess`。

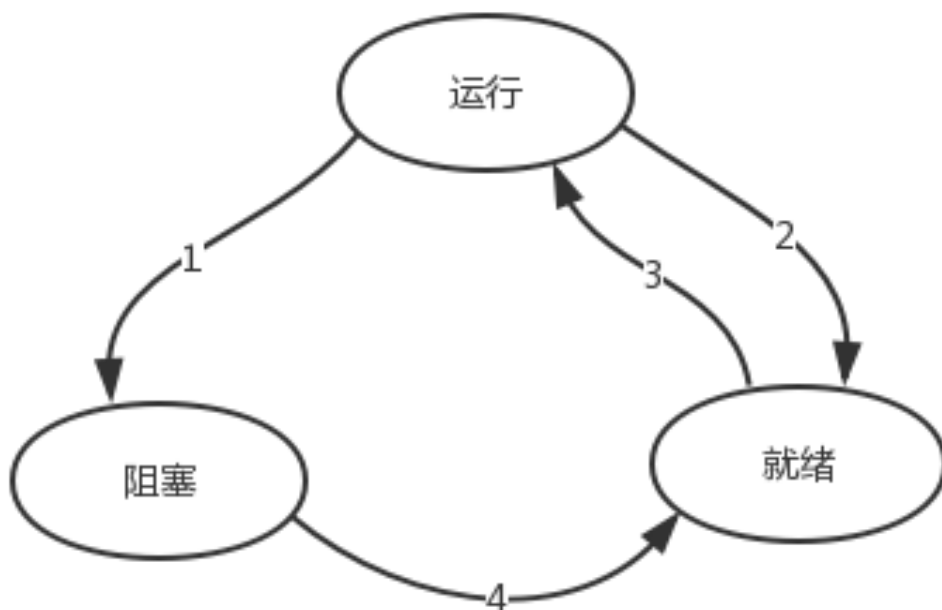
进程创建后，父进程和子进程有各自不同的地址空间（多道技术要求物理层面实现进程之间内存的隔离），任何一个进程的在其地址空间中的修改都不会影响到另外一个进程。

# 并发编程-进程

## 进程的中止:

- 正常退出（如用户点击交互式页面的叉号，或程序执行完毕后退出，linux: `exit`, windows: `ExitProcess`）
- 出错退出（自愿，`python a.py`中`a.py`不存在）
- 严重错误（非自愿，执行非法指令，如引用不存在的内存，1/0等，可以捕捉异常，`try...except...`）
- 被其他进程杀死（非自愿，如`kill -9`）

## 进程的状态:



- 1、IO阻塞或调用`sleep()`
- 2、进程用时过长而被操作系统切换
- 3、操作系统切就绪态进程到CPU
- 4、阻塞结束，转入就绪

# 并发编程- multiprocessing模块-Process方法

multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数）。

需要再次强调的一点是：进程没有任何共享状态，进程修改的数据，改动仅限于该进程内。

```
import time
import random
from multiprocessing import Process

def Sleep(name):
    print('%s %s 睡觉中。。。。' % (__name__, name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床' % name)

if __name__ == '__main__':
    # 实例化得到四个对象
    p1 = Process(target=Sleep, args=('p1',)) # 必须加, 号
    p2 = Process(target=Sleep, args=('p2',))
    p3 = Process(target=Sleep, args=('p3',))
    p4 = Process(target=Sleep, args=('p4',))

    # 调用对象下的方法，开启四个进程
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    print('主')
```

```
import time
import random
from multiprocessing import Process

val = 0

def Sleep(name):
    global val
    val = val + 1
    print('%s %s 睡觉中。。。。' % (__name__, name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床 %s' % (name, val))

if __name__ == '__main__':
    # 实例化得到四个对象
    p1 = Process(target=Sleep, args=('p1',))
    p2 = Process(target=Sleep, args=('p2',))
    p3 = Process(target=Sleep, args=('p3',))
    li = [p1, p2, p3]

    for p in li:
        p.start()
    print('主')

    for p in li:
        p.join()
    print('end 主')
```

## 并发编程- multiprocessing模块

- 1、思考开启进程的方式一和方式二各开启了几个进程？
- 2、进程之间的内存空间是共享的还是隔离的？下述代码的执行结果是什么？

```
from multiprocessing import Process
```

```
n=100 #在windows系统中应该把全局变量定义在if __name__ == '__main__'之上就可以了
```

```
def work():
```

```
    global n
```

```
    n=0
```

```
    print('子进程内:',n)
```

```
if __name__ == '__main__':
```

```
    p=Process(target=work)
```

```
    p.start()
```

```
    print('主进程内:',n)
```



## 课堂练习（2） 用多进程去封装模拟FTP

```
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        try:
            while True:
                head_recv = conn.recv(4)
                head_len = struct.unpack("i", head_recv)[0]
                head_recv = conn.recv(head_len)
                head_data = json.loads(head_recv)
                print("recv", head_data)
                file_cmd = head_data['cmd']
                func = globals().get(file_cmd, None)
                if func is None:
                    raise ValueError(f"cmd error: {head_recv}")
                func(conn, head_data)
        except Exception as e:
            print("Error:", e)

if __name__ == "__main__":
    main()
```

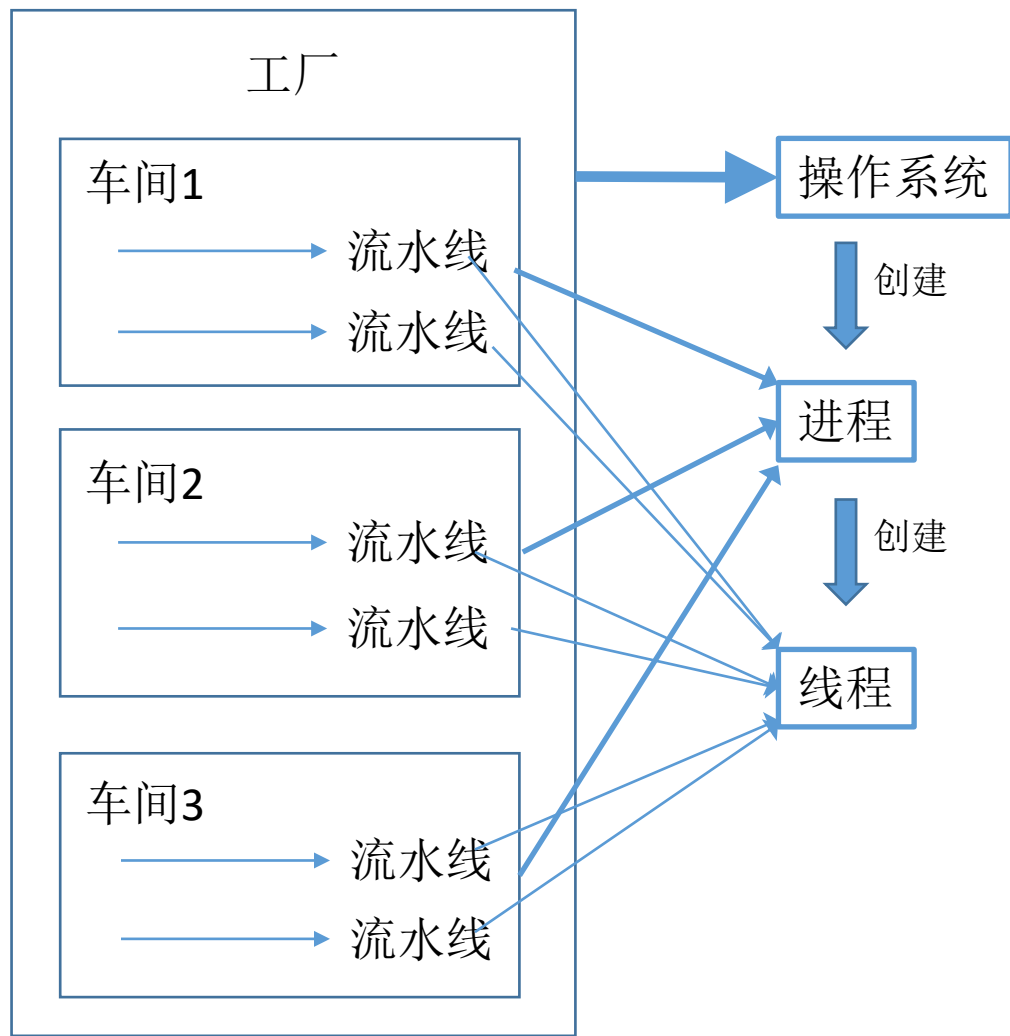
```
from multiprocessing import Process

def conn_thread(conn):
    print("new thread", conn)
    try:
        while True:
            head_recv = conn.recv(4)
            head_len = struct.unpack("i", head_recv)[0]
            head_recv = conn.recv(head_len)
            head_data = json.loads(head_recv)
            print("recv", head_data)
            file_cmd = head_data['cmd']
            func = globals().get(file_cmd, None)
            if func is None:
                raise ValueError(f"cmd error: {head_recv}")
            func(conn, head_data)
    except Exception as e:
        print("Error:", e)
```

```
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        process = Process(target=conn_thread, args=(conn,))
        process.start()
```

```
if __name__ == "__main__":
    main()
```

# 并发编程-线程



线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

每个进程都有自己的地址空间，在高并发场景下，为每一个请求都创建一个进程显然行不通（系统开销大响应用户请求效率低），为此才引入线程：

- 线程是执行单位，不是资源单位，申请线程不需要向操作系统额外申请地址空间（启动快）。
- 每一个进程至少有一个主线程，单线程进程宏观来看也是线性执行过程，微观上只有单一的执行过程。多线程进程宏观是线性的，微观上多个执行操作。

并发编程-线程与进程的区别：

	进程	线程
地址空间	独立	共享进程内地址空间
资源拥有	独立	共享本进程资源，如内存、I/O、cpu等
健壮性	独立	线程崩溃将整个进程故障
生成与切换速度	消耗资源大，效率低	消耗资源小，效率高
执行过程	操作系统调用进程	进程调用线程
处理器调度	资源单位	执行单位
并发执行	可以	可以

# 并发编程-threading模块

用法与multiprocessing模块相同

```
import time
import random
from multiprocessing import Process
from threading import Thread
def Sleep(name):
    print('%s %s 睡觉中。。。。' % (__name__, name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床' % name)

if __name__ == '__main__':
    # 实例化得到四个对象

    p1 = Thread(target=Sleep, args=('p1',))
    p2 = Thread(target=Sleep, args=('p2',))
    p3 = Thread(target=Sleep, args=('p3',))
    p4 = Thread(target=Sleep, args=('p4',))
    # 调用对象下的方法, 开启四个进程

    p1.start()
    p2.start()
    p3.start()
    p4.start()
    print('主')
```

```
import time
import random
from multiprocessing import Process
from threading import Thread
class Sleep(Thread):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print('%s %s 睡觉中。。。。' % (__name__, self.name))
        time.sleep(random.randrange(1, 5))
        print('%s 起床' % self.name)

# if __name__ == '__main__':
# 实例化得到四个对象
p1 = Sleep('p1')
p2 = Sleep('p2')
p3 = Sleep('p3')
p4 = Sleep('p4')
p_list = [p1, p2, p3, p4]
for p in p_list:
    p.start()
print('主')
```

# 并发编程-threading模块-其他参数

## ■ Thread实例对象的方法

# isAlive(): 返回线程是否活动的。

# getName(): 返回线程名。

# setName(): 设置线程名。

## ■ threading模块提供的一些方法:

# threading.currentThread(): 返回当前的线程变量。

# threading.enumerate(): 返回一个包含正在运行的线程的list, 不包括启动前和终止后的线程。

# threading.activeCount(): 返回正在运行的线程数量, 与len(threading.enumerate())有相同的结果。

```
import threading
import time
def work():
    time.sleep(3)
    print(threading.current_thread().getName())
```

```
t = threading.Thread(target=work)
t.start()

print(threading.current_thread().getName())
print(threading.current_thread()) # 主线程
print(threading.enumerate()) # 连同主线程在内有两个运行的线程
print(threading.active_count())
print('主线程/主进程')
```

# 并发编程-threading模块-其他参数

主系统等待子线程结束

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('egon',))
    t.start()
    t.join()
    print('主线程')
    print(t.is_alive())
```

主系统不等待子线程结束

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('egon',))
    t.daemon = True
    t.start()
    # t.join()
    print('主线程')
    print(t.is_alive())
```

思考执行结果？

```
from threading import Thread
import time
def foo():
    print(123)
    time.sleep(3)
    print("end123")
def bar():
    print(456)
    time.sleep(1)
    print("end456")

if __name__ == '__main__':
    t1=Thread(target=foo)
    t2=Thread(target=bar)

    t1.daemon=True
    t1.start()
    t2.start()

    print("main-----")
```

## 课堂练习（3） 用多线程去封装模拟FTP

```
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        try:
            while True:
                head_recv = conn.recv(4)
                head_len = struct.unpack("i", head_recv)[0]
                head_recv = conn.recv(head_len)
                head_data = json.loads(head_recv)
                print("recv", head_data)
                file_cmd = head_data['cmd']
                func = globals().get(file_cmd, None)
                if func is None:
                    raise ValueError(f"cmd error: {head_recv}")
                func(conn, head_data)
        except Exception as e:
            print("Error:", e)

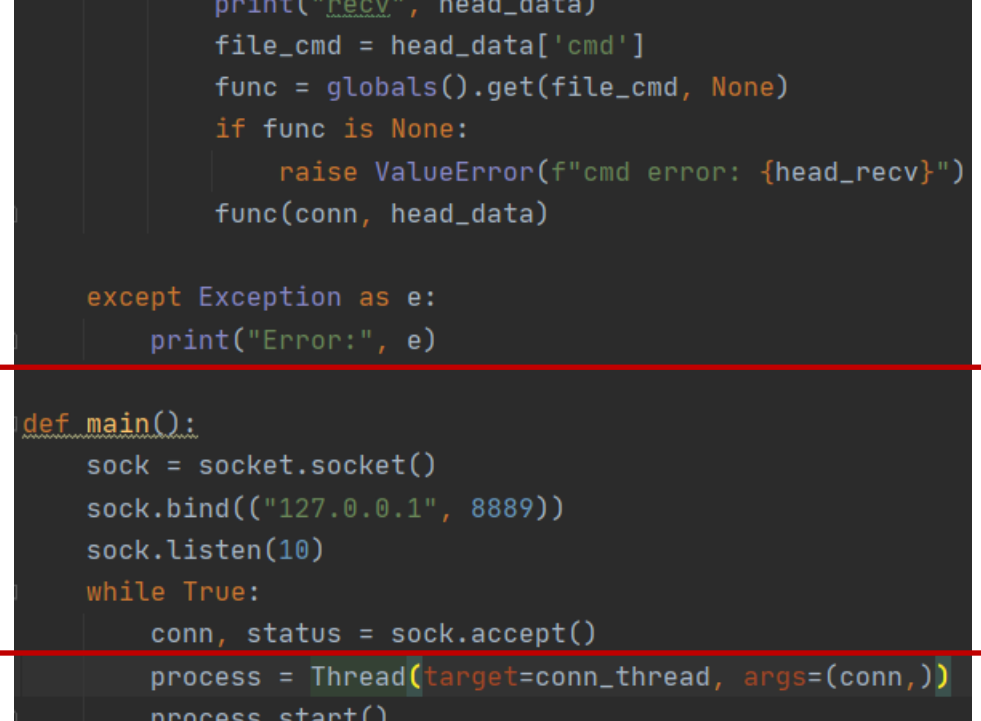
if __name__ == "__main__":
    main()
```

```
from threading import Thread

def conn_thread(conn):
    print("new thread", conn)
    try:
        while True:
            head_recv = conn.recv(4)
            head_len = struct.unpack("i", head_recv)[0]
            head_recv = conn.recv(head_len)
            head_data = json.loads(head_recv)
            print("recv", head_data)
            file_cmd = head_data['cmd']
            func = globals().get(file_cmd, None)
            if func is None:
                raise ValueError(f"cmd error: {head_recv}")
            func(conn, head_data)
    except Exception as e:
        print("Error:", e)

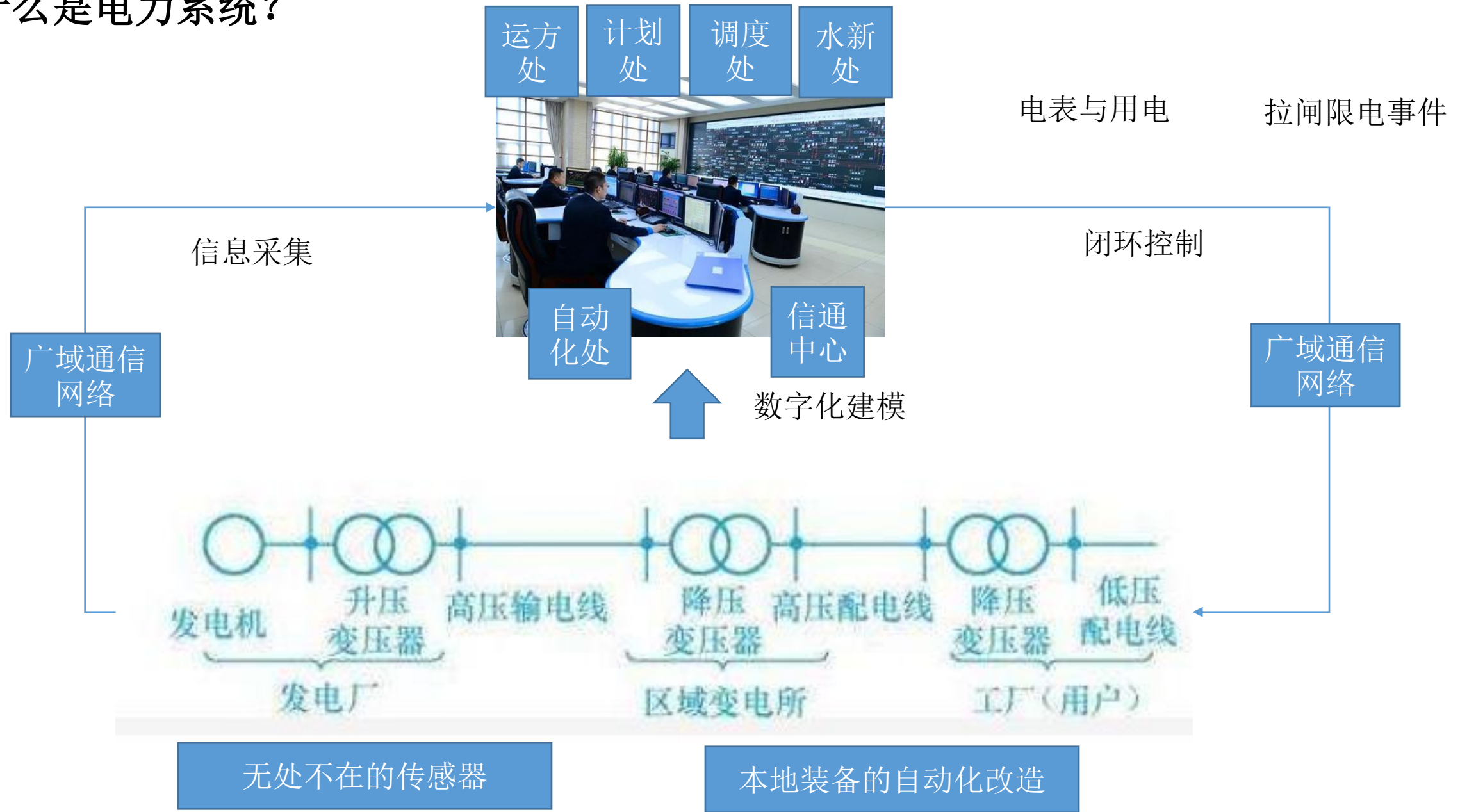
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        process = Thread(target=conn_thread, args=(conn,))
        process.start()

if __name__ == "__main__":
    main()
```



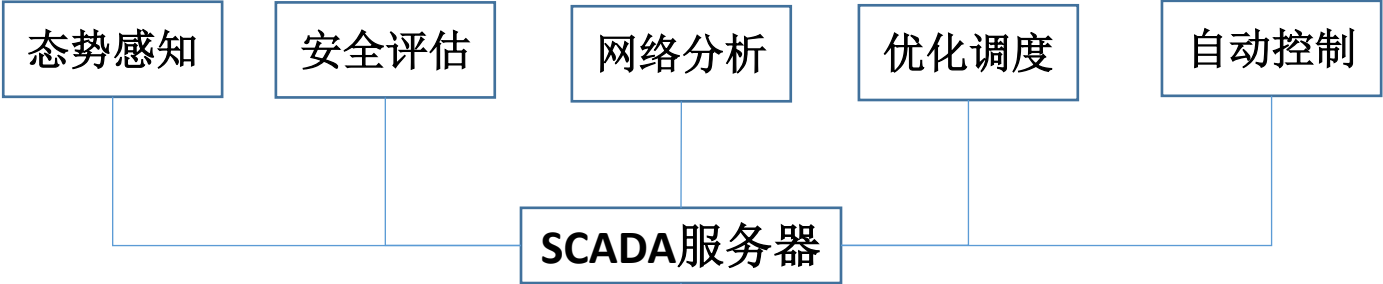
# 实际上：信息物理系统

## 什么是电力系统？

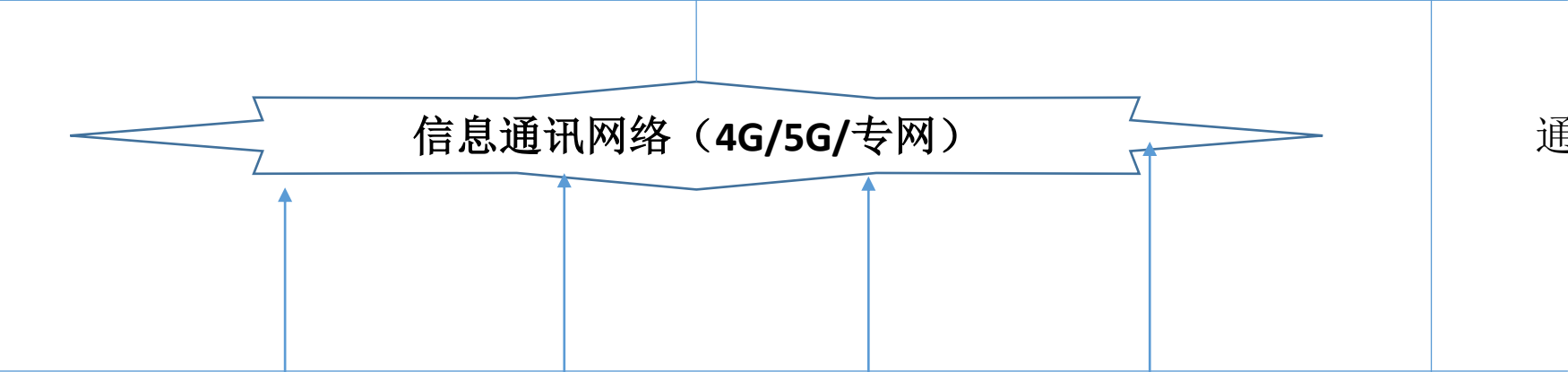




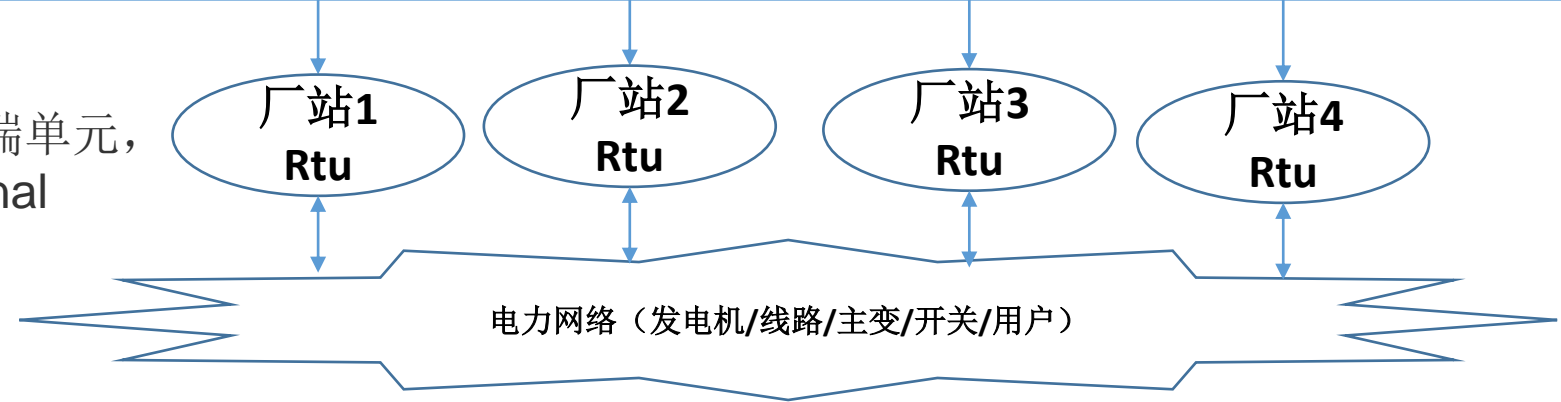
# 人机交互界面



电网能量管理系统



通信网络

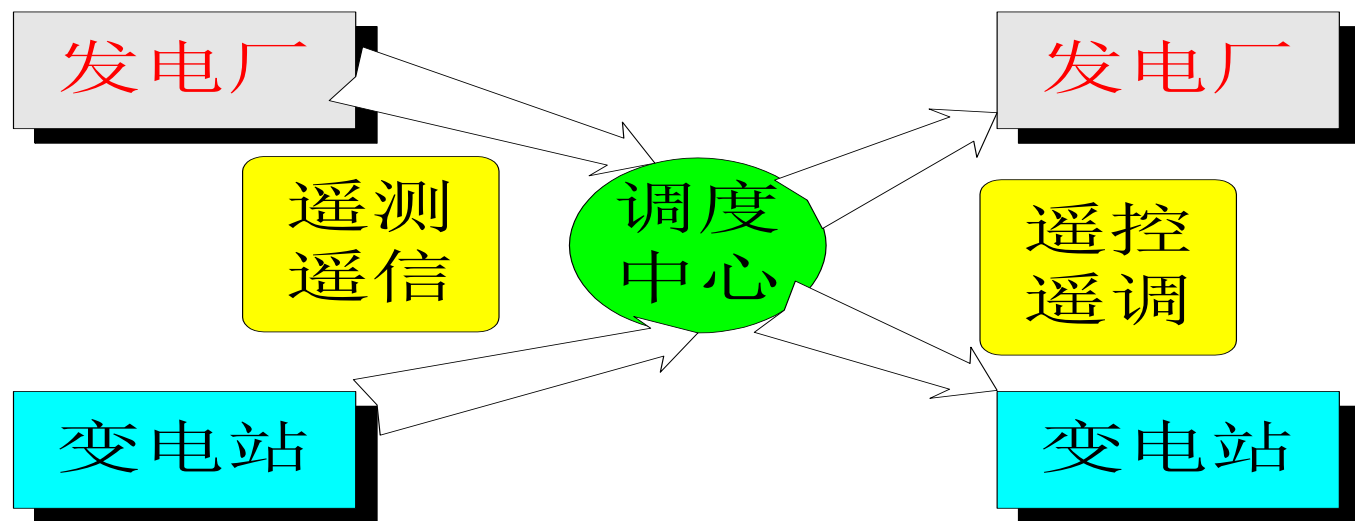


物理系统

RTU, 远程终端单元,  
Remote Terminal  
Unit

## ■ 远动的概念

- 远动(Telecontrol)
- 遥测(Telemetering)
- 遥信(Telesignal)
- 遥控(Teleswitching)
- 遥调(Teleadjusting)



# 课程目的

- 定义电网公司EMS和变电站RTU侧的通信相关数据表结构
- 开发EMS侧通信程序、开发RTU侧通信程序。
- EMS侧具备功能：
  - 根据RTU表里的内容， 动态链接RTU服务端程序
  - 接收RTU传送过来的变化遥测、变化遥信，更新本地数据库。
  - 检测本地数据库里的遥调和遥控变化，向RTU发送遥调、遥控。
- RTU侧具备功能：
  - 启动数据服务
  - 检测本地数据库里的遥测和遥信变化，向EMS上传变化遥测、变化遥信。
  - 接收RTU传送过来的变化遥测、变化遥信，回写本地数据库。

# 数据库操作

## 本节重点:

- 数据库基本概念
- Python 接口用法
- 电网前置表定义

数据表列  
(数据对象的一个属性)

The screenshot shows a WPS spreadsheet with a data table. The table has 4 columns: A (姓名), B (年龄), C (性别), and D (学校). The data rows are numbered 1 to 9. A red box highlights the entire table area from row 1 to row 9 and column A to column D. Another red box highlights a single row (row 4) from column A to column D. A third red box highlights a single column (column B) from row 1 to row 9. The spreadsheet interface includes a menu bar at the top with options like '文件' (File), '开始' (Start), '插入' (Insert), '页面布局' (Page Layout), '公式' (Formulas), '数据' (Data), '审阅' (Review), and '视图' (View). The status bar at the bottom shows '学生信息' (Student Information), '教师信息' (Teacher Information), and '院系信息' (Department Information).

	A	B	C	D	E
1	姓名	年龄	性别	学校	
2	张三	18	男	清华	
3	李四	20	男	清华	
4	李5	20	男	清华	
5	李6	20	男	清华	
6	李7	20	男	清华	
7	李8	20	男	清华	
8	李9	20	男	清华	
9					
10					
11					
12					
13					
14					
15					
16					
17					

数据表  
(多个同类数据的集合)

数据表行  
(一个数据对象)

数据库  
(多张表的集合)

# 关系数据库基本概念

- 关系：是一个二维表，其中包含以下性质
  - (1) 表中的每个项是单值的
  - (2) 每一列都有唯一的一个名称（属性名）
  - (3) 一列中所有的值必须是相同的属性的值
  - (4) 各列的顺序没有限制
  - (5) 每一行不能相同
  - (6) 各行的顺序没有限制
- 表中的行称为记录（元组,record, tuple），列称为域（属性，字段,field）
- 关系数据库：一组关系的组合
- 主关键字（primary key）：能唯一识别出表中某一行的列或列的集合

# 数据库基本概念

开关表

标识	RTU	遥信点号	电压等级	...
开关1	1	1	220	...
				...
				...

遥信表

标识	RTU	点号	遥控点号	遥信类型	...
开关1.遥信	1	1	1	1	...
					...
					...

遥控表

标识	RTU	点号	关键字	...
开关1.遥控	1	1	开关1	...
				...
				...

数据库是一种存放多种实体、实体属性和关系的信息的结构



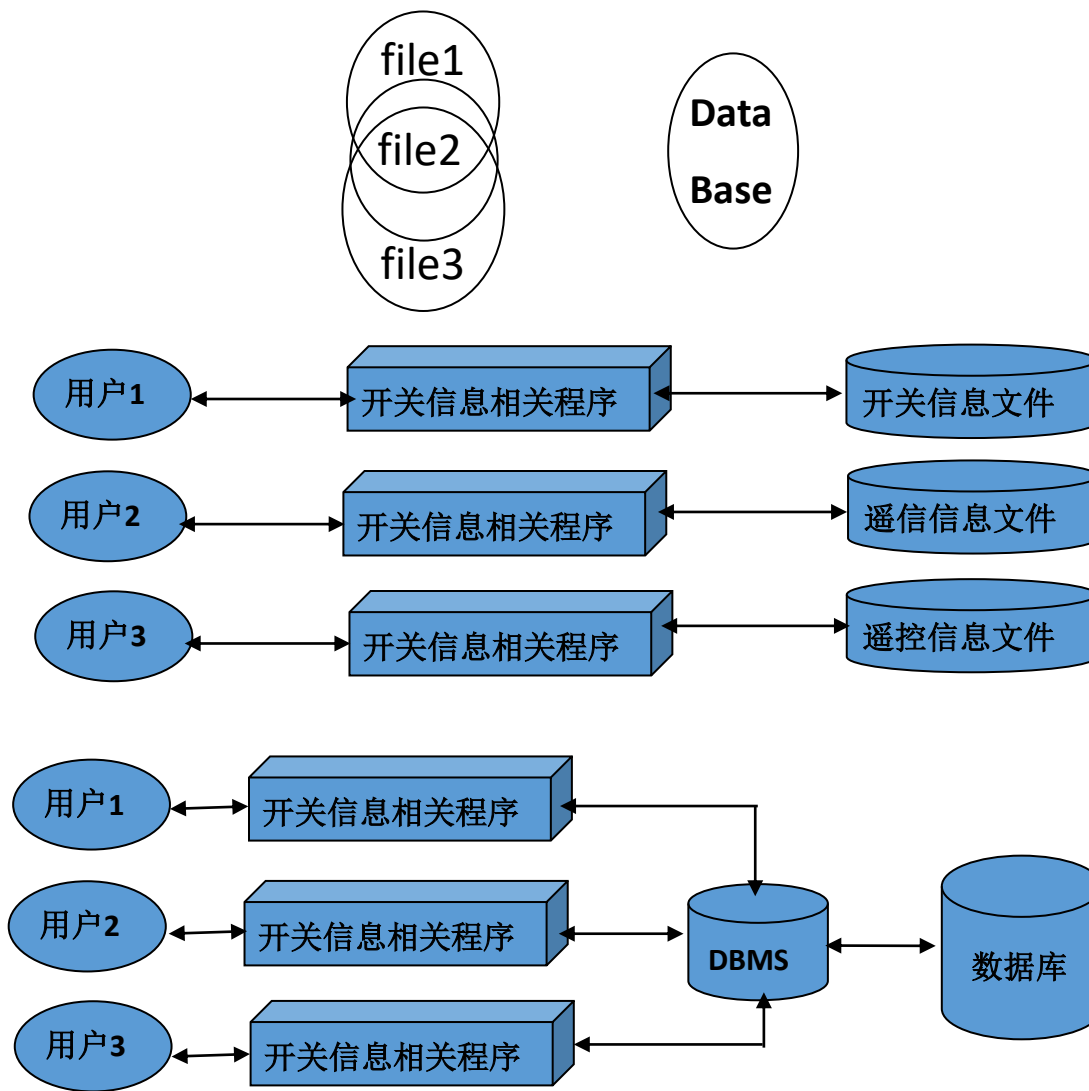
# 数据库基本概念

- 数据库系统提供：数据的增加、插入、删除、修改和检索功能。

	FILE NAME	NEW_HOST	MODIFY_TIME	CAN_SAVE	IS_FILEEXIST	TYPE	PARENT	FILE_PATH
▶	长春地区	student2	2002-8-3 15:57:25	1	1	2	吉林省	长春地区.ems
	镇费一次变	dtsserver	2002-9-17 11:22:36	1	1	3	白城（松原）地区	镇费一次变.BMS
	test	dtsserver	2002-12-28 9:26:12	1	1	4	图	test.ems
	水洞一次变	student2	2002-8-3 9:28:53	1	1	3	通化地区	水洞一次变.BMS
	通过一次变	dtsserver	2002-10-16 14:10:48	1	1	3	外网	通过一次变.ems
	吉林省1	teacher1	2000-3-5 11:10:41	0	1	4	<NULL>	吉林省1.ems
	东丰66KV	student2	2002-9-6 21:26:47	1	1	3	辽源地区	东丰66KV.ems
	吉林省	student2	2002-9-19 12:03:57	1	1	1	<NULL>	吉林省.ems
	潮流图8.5	student1	2002-8-5 8:43:06	1	0	4	系统工况	潮流图8.5.ems
	龙江等值	dtsserver	2002-10-17 15:05:33	1	1	3	外网	龙江等值.ems
	系统工况	student2	2002-11-14 14:57:48	1	1	4	图	系统工况.ems
	吉林省发电机组出	telecom	2003-1-11 17:26:29	1	1	4	图	吉林省发电机组出
	虎石台一次变	student1	2002-9-11 15:31:47	1	1	3	外网	虎石台一次变.em

# 数据库比文件系统的优势

- 减少冗余
- 数据一致性
- 共享
- 标准化
- 数据安全
- 完整性
- 平衡需求冲突
- 数据独立性



# 数据库简介

- 数据库是指以一定方式存储在一起，能为多个用户共享，具有尽可能小的冗余特性，是与应用程序彼此独立的数据集合
- 相关术语
  - DB 数据库(database)
  - DBMS 数据库管理系统
  - DBA 数据库管理员
  - RDB 关系式数据库




- 常见的商业数据库
  - 甲骨文Oracle
  - 微软的Sqlserver
  - IBM的DB2
- 常见的开源数据库
  - Sun公司的Mysql，2009年sun公司被Oracle收购，所以mysql现在属于甲骨文旗下的产品
  - 开源的SQLite，轻量级嵌入式关系型数据库

- 数据库操作语言（SQL）
  - SQL（Structured Query Language）是一种针对数据库的结构化查询语言，用于实现数据库查询和程序设计，常用于关系数据库系统，实现数据存取、查询、更新等操作。
- 常用的SQL语句
  - 数据操作语言（DML）：SELECT、INSERT、DELETE、UPDATE
  - 数据定义语言（DDL）：CREATE TABLE

# Sqlite数据库

- SQLite是一个轻量级的开源数据库，源代码完全公开不受版权限制，实现了自给自足的、无服务器、零配置的SQL 数据库引擎，也是最广泛使用关系式数据库
- SQLite ([www.sqlite.org](http://www.sqlite.org))提供SQLite 的已编译版本和源代码程序，目前支持如下平台：
  - Linux
  - Mac OS X
  - Windows
  - Windows Phone 8
  - Windows Runtime



- Sqlite主要特性
  - SQLite 不需要一个单独的服务器进程或操作的系统（无服务器的）
  - SQLite 可以不用配置，安装或管理十分简单 
  - SQLite 数据库是存储在一个单一的跨平台的磁盘文件
  - SQLite 是非常小的，是轻量级的，完全配置时小于400KiB，省略可选功能配置时小于250KiB
  - SQLite 是自给自足的，开发者不需要外部的依赖
  - SQLite 支持 SQL92标准的大多数查询语言的功能
  - SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问

注：ACID即原子性、一致性、隔离性、持久性

- Sqlite的操作指令

- 进入SQLite 的命令行界面后，可以输入两种指令，一种是自身配置和格式控制相关指令，这些指令都以 “.” 开头；另外一种指令是SQL语言(Sql语句)，实现对数据库的增删改查等操作，这些指令以 “;” 结束
- 输入.help 或 .h 可以获取以 “.” 开头指令的帮助信息
- 输入.exit 或.quit 退出SQLite的命令界面，回到系统的控制终端
- 如果需要清屏，可以使用 “ctrl+L”



- 以 “.” 开头的常用命令

- .database       //查看数据库的名字和对应的文件名
- .table           //查看数据表的名字
- .schema          //查看数据表创建时的详细信息
- .mode            //设置显示模式，如tab/list/column/html/...
- .nullvalue        //设置空白字段显示的字符串
- .header on        //显示数据表的表头

注：可以将格式显示配置命令写入配置文件(.sqliterc)，下次启动时会自动从该配置文件中加载配置。

## 1. 常用的sqlite自身配置和格式显示相关指令

命令	含义
<code>.help</code>	//查看这些指令的帮助信息
<code>.exit</code> 或 <code>.quit</code>	//退出数据库回到控制终端
<code>.open testDB.db</code>	//打开testDB.db数据库文件
<code>.database</code>	//查看数据库名字和对应的文件名
<code>.table</code>	//查看数据库中数据表的名字
<code>.mode MODE</code>	//设置数据表显示模式, MODE:list(默认)/column/tab/html...
<code>.header on</code>	//显示数据表的表头(列名)
<code>.schema</code>	//查看数据表创建时的详细信息
<code>.nullvalue "NULL"</code>	//设置数据表空白位置显示"NULL"

- 创建数据表

- 语法格式

CREATE TABLE 表名(列名1 类型 [约束], 列名2 类型 [约束],...);

- 常用的类型

INT(整型)、TEXT(字符串)、REAL(浮点数)

- 常用的约束：

PRIMARY KEY(主键):表示该列数据唯一,可以加快数据访问.

NOT NULL(非空):该列数据不能为空

CSDN @weixin\_101

```
1      sqlite> CREATE TABLE student (  
2          ...> id INT NOT NULL PRIMARY KEY,  
3          ...> name TEXT NOT NULL,  
4          ...> score REAL NOT NULL );  
5      sqlite> .table //查看已存在数据表名字  
6      company  student
```

create table student(id int, name text, score float, primary key(id));

## 四、删除数据表

语法:

DROP TABLE 表名;

注:

慎用，数据表一旦删除，里面所包含的数据也将随之消失!

eg:

```
1      sqlite> .table
2      company  student
3      sqlite> DROP TABLE student;
4      sqlite> .table
5      company
```

**drop table if exists student**

## 五、向数据表插入数据

语法:

INSERT INTO 表名 (列名1,列名2,...) VALUES(数值1,数值2,...);

eg:

```
1  sqlite> INSERT INTO company
2      ...> (id,name,age,address,salary)
3      ...> VALUES(10018,'貂蝉',22,'山东',13000.5);
4  sqlite> SELECT * FROM company;
5  sqlite> INSERT INTO company
6      ...> (name,id,age,salary)
7      ...> VALUES('小乔',10019,21,12000);
8
9  sqlite> INSERT INTO company
10     ...> VALUES(10021,'孙尚香',26,'江南',9000);
```

insert into student (id, name, score) values(1, 'amao', 100.0);

- 从数据表删除数据

- 语法格式

DELETE FROM 表名 WHERE 条件表达式;

注：可以没有WHERE子句，但数据表的所有数据都将被删除

注：如果有多个条件可以使用逻辑与(and)或逻辑或(or)连接

- 删除数据实例

DELETE FROM company WHERE id=10029;

DELETE FROM company WHERE address='成都' and salary<1000;

```
1  sqlite> DELETE FROM company
2    ...> WHERE id = 10029;
3  sqlite> DELETE FROM company
4    ...> WHERE age>25 and address='江南';
```

CSDN @weixin\_101

delete from student;

delete from student where id >= 2 and name = 'jess';

delete from student where id >= 2 or name = 'jess';



- 修改数据表中的数据

- 语法格式

UPDATE 表名 SET 列名1=新数值,列名2=新数值,... WHERE 条件表达式;

注：可以没有WHERE子句，但数据表的所有数据都将被修改

注：“新数值”可以是一个常数，也可以是一个表达式

- 修改数据实例

UPDATE company SET age = 45 WHERE id = 10011;

UPDATE company SET salary=salary+2000 WHERE age>=30 and age<=35;

CSDN @weixin\_101

```
sqlite> update student set score = 70;
```

```
sqlite> update student set score = 60 where id = 1;
```

```
sqlite> update student set score = 60, name = 'cuole' where id = 1;
```

- 查询数据表中的数据

- 语法格式

SELECT 列名1,列名2,... FROM 表名;

SELECT 列名1,列名2,... FROM 表名 WHERE 条件表达式;

SELECT 列名1,列名2,... FROM 表名 ORDER BY 列名 排序方式;

注：ORDER BY子句可以和WHERE子句配合使用，也可以独立使用

- 查询操作实例

SELECT \* FROM company WHERE salary>10000 or salary<3500;

SELECT \* FROM company ORDER BY id ASC;

```
sqlite> select * from student;
```

```
sqlite> select * from student where id = 1;
```

```
sqlite> select * from student where id > 1;
```

```
sqlite> select * from student where id = 1;
```

```
sqlite> select id,name from student where id = 1;
```

```
sqlite> select name from student where id = 1;
```



- 模糊查询

- 语法格式

SELECT 列名1,列名2,... FROM 表名 WHERE 列名 LIKE 模糊匹配条件;

- 模糊匹配通配符

百分号 ( % ) : 代表零个、一个或多个数字或字符

下划线 ( \_ ) : 代表一个单一的数字或字符

注 : 它们可以被组合使用

- 模糊查询实例

SELECT \* FROM company WHERE salary LIKE '%8%';

SELECT \* FROM company WHERE name LIKE '关\_';

SELECT \* FROM company WHERE salary WHERE SALARY LIKE ' \_2%3 ';

CSDN @weixin\_101

## 练习使用SQL语句在命令行操作Sqlite数据库

- 在命令行启动Sqlite数据库(student.db)，创建名为student数据表，用于保存学生成绩，列的名字依次为学号(id)，姓名(name)，成绩(score)，并完成以下操作：

- 1) 插入以下数据

ID	Name	Score
10001	张三	88
10002	李四	99
10005	闵卫	100
10004	游成伟	59.5
10006	Jerry	100.5

- 2) 删除：Jerry一条数据
- 3) 修改：将游成伟成绩修改为:89.5
- 4) 查询：升序、降序、模糊查找...

## 课堂练习（4） 安装并操作sqlite数据库

### 登录sqlite3数据库

```
D:\selib>sqlite3
SQLite version 3.28.0 2019-04-16 19:49:53
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open test.db
sqlite>
```

### 插入表数据: insert into ... (...) values (...)

```
sqlite> insert into student(id,name, score) values(10001, '张三', 88);
sqlite> insert into student(id,name, score) values(10002, '李四', 99);
sqlite> insert into student(id,name, score) values(10002, '王五', 100);
Error: UNIQUE constraint failed: student.id
sqlite> insert into student(id,name, score) values(10003, '王五', 100);
```

### 删除表数据: delete from ... where ...

```
sqlite> delete from student where id = 10003;
sqlite> select * from student;
10001|张三|88.0
10002|李四|99.0
sqlite>
```

### 删除表 drop table if exists ...

```
sqlite>
sqlite> drop table if exists student;
sqlite>
sqlite> .tables
sqlite>
sqlite>
```

### 创建表 create table ... (...)

```
sqlite> create table student(id int, name text, score float, primary key(id));
sqlite>
sqlite>
sqlite> .table
student
```

### 修改表数据: update ... set .... where ...

```
sqlite> update student set score = 60 where name = '张三';
sqlite>
sqlite> select * from student;
10001|张三|60.0
10002|李四|99.0
sqlite>
```

# SQLAlchemy包

SQLAlchemy是Python SQL工具包和对象关系映射器，是python中最著名的ORM(Object Relationship Mapping)框架，它简化了应用程序开发人员在原生SQL上的操作，使开发人员将主要精力都放在程序逻辑上，从而提高开发效率。它提供了一整套著名的企业级持久性模式，设计用于高效和高性能的数据库访问。

使用ORM操作数据库：

优势：代码易读，隐藏底层原生SQL语句，提高了开发效率。

劣势：执行效率低，将方法转换为原生SQL后 原生SQL不一定是最优的。

# 建立引擎：

任何SQLAlchemy应用程序的开始都是一个名为 [Engine](#) . 此对象充当连接到特定数据库的中心源，提供工厂和称为 [connection pool](#) 对于这些数据库连接。引擎通常是一个只为特定数据库服务器创建一次的全局对象，并使用一个URL字符串进行配置，该字符串将描述如何连接到数据库主机或后端。

```
from sqlalchemy import create_engine
```

```
engine = create_engine("sqlite:///center.db", echo=True, future=True)
```

这个 [Engine](#) ，第一次返回时 [create\\_engine\(\)](#) ，尚未实际尝试连接到数据库；只有在第一次要求它对数据库执行任务时才会发生这种情况。这是一种称为 [lazy initialization](#) .

## 获取链接：

从面向用户的角度来看，Engine对象将提供到数据库的连接单元，称为 Connection 。直接使用核心时 Connection 对象是如何完成与数据库的所有交互。作为 Connection 表示针对数据库的开放资源，我们希望始终将此对象的使用范围限制在特定上下文中，最好的方法是使用Python上下文管理器表单，也称为 the with statement：

```
>>> with engine.connect() as conn:  
...     result = conn.execute("select 'hello world'")  
...     print(result.all())
```

# 添加数据：

两种数据添加方法（插入一行，插入多行）

```
insertTable = engine.execute("insert into userDemo(id,name) values(1, 'test1')")
print(insertTable)
insertTable = engine.execute("insert into userDemo(id,name) values(2, 'test2')")
print(insertTable)

insertTable = engine.execute("insert into userDemo(id,name) values(:x, :y)",
                              [{"x": 3, "y": "test3"}, {"x": 4, "y": "test4"}, {"x": 5, "y": "test5"}])
print(insertTable)
#
```

## 获取数据：

- **元组赋值** -这是Python最惯用的风格，即在接收到变量时将变量按位置分配给每一行：

```
result = conn.execute(text("select x, y from some_table"))

for x, y in result:
    # ...
```

- **整数索引** -元组是Python序列，因此也可以进行常规整数访问：

```
result = conn.execute(text("select x, y from some_table"))

for row in result:
    x = row[0]
```



# 获取数据：

- **属性名称** - 由于这些是Python命名的元组，这些元组具有与每个列的名称相匹配的动态属性名。这些名称通常是SQL语句为每行中的列指定的名称。虽然它们通常是相当可预测的，也可以由标签控制，但在定义较少的情况下，它们可能会受到特定于数据库的行为的影响：

```
result = conn.execute(text("select x, y from some_table"))

for row in result:
    y = row.y

    # illustrate use with Python f-strings
    print(f"Row: {row.x} {row.y}")
```

- **映射访问** - 以Python形式接收行 **映射** 对象，它本质上是Python的公共接口的只读版本 dict 对象 `Result` 可能是 **转化** 变成一个 `MappingResult` 对象使用 `Result.mappings()` 修饰符；这是一个结果对象，它生成类似dictionary的 `RowMapping` 对象而不是 `Row` 物体：

```
result = conn.execute(text("select x, y from some_table"))

for dict_row in result.mappings():
    x = dict_row['x']
    y = dict_row['y']
```

## 带参数获取数据：

```
selectUser =
```

```
    engine.execute("select * from userDemo")
```

```
selectUser =
```

```
    engine.execute("select * from userDemo where name = 'test2' and id > 1")
```

```
selectUser =
```

```
    engine.execute("select * from userDemo where name = :x and id > :y", {"x": "test2", "y": 1})
```

# 操作示例

```
from sqlalchemy import create_engine

# 建立数据库链接
engin = create_engine("sqlite:///db/test.db")
conn = engin.connect()

# 删除数据库表结构
conn.execute("drop table if exists person")

# 添加数据库表结构
conn.execute("create table person(id int, name varchar2(64), age int)")

# 向数据库中插入数据
conn.execute("insert into person(id,name,age) values (1, 'a', 18)")
conn.execute("insert into person(id,name,age) values (2, 'b', 18)")
```

## 操作示例

```
# 4种对 select 结果进行遍历的方法:  
  
for result in results:  
    print(result)  
  
for id_name_age in results:  
    print(id_name_age)  
  
for result in results:  
    print(result[0], result[1], result[2], result[3])  
  
for result in results:  
    print(result.id, result.name, result.age)
```

# 课堂练习: sqlalchemy

```
from sqlalchemy import create_engine, text

engine = create_engine("sqlite:///db/student.db", echo=True)

with engine.connect() as conn:
    conn.execute(text("drop table if exists student"))
    info = "create table student(id int, name text, score float, primary key(id))"
    conn.execute(text(info))

    for p in range(1, 11):
        info = f"insert into student(id, name, score) values ({p}, 'person_{p}', {100-p})"
        conn.execute(text(info))

    conn.execute(text("update student set name = 'amao' where id = 1"))
    conn.execute(text("update student set score = 60.0 where name = 'person_5'"))

    conn.execute(text("delete from student where score > 98.0"))

    results = conn.execute(text("select * from student where score > 95.0"))
    for result in results:
        print(result.id, result.score, result.name)

    conn.commit()
```

# 课堂练习（1）

1. 新建文件夹，建议取名为scada，存储 SCADA 原型系统程序和数据。
2. 在scada里，新建两个文件，建议取名为 `ems.py` 和 `rtu.py`，分别运行 `ems`主站和 `rtu`子站侧的程序，运行方式：
  - `python ems.py`
  - `python rtu.py 1`，`n`分别代表第`n`个`rtu`，取值：1-10
3. 在scada里，新建子文件夹`db`，用于存放`sqlite`数据库文件。
  - `ems`主站的数据库文件名称：`db/ems.db`
  - 第`n`个`RTU`子站的数据库文件名称：`db/rtu{n}.db`，比如 `db/rtu1.db`、`db/rtu2.db`，以此类推。
4. 在`ems.py`和 `rtu.py` 里，增加函数 `auto_sql_create()`，在每次程序启动时，自动重新数据库结构，并插入相关通信数据。
  - `ems`主站：每个`RTU`子站的子站定义、遥信、遥测、遥控、遥调；
  - `Rtu`子站：该`RTU`子站的子站定义、遥信、遥测、遥控、遥调。

RTU 子站信息的测试数据(rtu\_info):

RTU号 (id)	RTU名称 (name)	ip地址 (ip_addr)	端口 (port)	通信状态 (status)	数据刷新时间 (refresh_time)
1	rtu1	127.0.0.1	6666	0	0
2	rtu2	127.0.0.1	6667	0	0
3	rtu3	127.0.0.1	6668	0	0
4	rtu4	127.0.0.1	6669	0	0
5	rtu5	127.0.0.1	6670	0	0

第n个RTU子站的遥测数据(rtu\_yc\_info):

点号 id	名称 name	遥测值 value	数据质量 status	数据刷新时间 refresh_time
1	yc.rtu{n}.1	0.0	0	0
2	yc.rtu{n}.2	0.0	0	0
3	yc.rtu{n}.3	0.0	0	0
4	yc.rtu{n}.4	0.0	0	0
5	yc.rtu{n}.5	0.0	0	0
6	yc.rtu{n}.6	0.0	0	0

第n个RTU子站的遥信数据(rtu\_yx\_info) :

点号 id	名称 name	遥信值 value	数据质量 status	数据刷新时间 refresh_time
1	yx.rtu{n}.1	1	0	0
2	yx.rtu{n}.2	1	0	0
3	yx.rtu{n}.3	1	0	0
4	yx.rtu{n}.4	1	0	0
5	yx.rtu{n}.5	1	0	0
6	yx.rtu{n}.6	1	0	0



第n个RTU子站的遥调数据(rtu\_yt\_info)：

点号 id	名称 name	遥调值 value	数据质量 status	数据刷新时间 refresh_time
1	yt.rtu{n}.1	0.0	0	0
2	yt.rtu{n}.2	0.0	0	0
3	yt.rtu{n}.3	0.0	0	0
4	yt.rtu{n}.4	0.0	0	0
5	yt.rtu{n}.5	0.0	0	0

第n个RTU子站的遥控数据(rtu\_yk\_info)：

点号 id	名称 name	遥控值 value	数据质量 status	数据刷新时间 refresh_time
1	yk.rtu{n}.1	1	0	0
2	yk.rtu{n}.2	1	0	0
3	yk.rtu{n}.3	1	0	0
4	yk.rtu{n}.4	1	0	0
5	yk.rtu{n}.5	1	0	0

电网公司：RTU信息表(ems\_rtu\_info)

列名	描述	类型	是否主键
id	序号	int	是
name	名称	varchar2(64)	否
ip_addr	地址	varchar2(64)	否
port	端口	int	否
status	连接状态	int	否
refresh_time	刷新时间	int	否

电网公司：遥测信息表(ems\_yc\_info)

列名	描述	类型	是否主键
rtu_id	RTU号	int	是
pnt_no	点号	int	是
name	名称	varchar2(64)	否
value	值	float	否
status	数据质量	int	否
refresh_time	刷新时间	int	否

遥测信息表(ems\_yx\_info)

列名	描述	类型	是否主键
rtu_id	RTU号	int	是
pnt_no	点号	int	是
name	名称	varchar2(64)	否
value	值	int	否
status	数据质量	int	否
refresh_time	刷新时间	int	否

电网公司：遥调信息表(ems\_yt\_info)

列名	描述	类型	是否主键
rtu_id	RTU号	int	是
pnt_no	点号	int	是
name	名称	varchar2(64)	否
value	值	float	否
ctrl_time	控制时间	int	否
ret_code	返回码	int	否

电网公司：遥控信息表(ems\_yk\_info)

列名	描述	类型	是否主键
rtu_id	RTU号	int	是
pnt_no	点号	int	是
name	名称	varchar2(64)	否
value	值	int	否
ctrl_time	控制时间	int	否
ret_code	返回码	int	否

站端：RTU信息表(rtu\_info)

列名	描述	类型	是否主键
id	序号	int	是
name	名称	varchar2(64)	否
status	连接状态	int	否
ip_addr	IP地址	varchar2(64)	否
port	端口号	int	否
refresh_time	刷新时间	int	否

站端：遥测信息表(rtu\_yc\_info)

列名	描述	类型	是否主键
id	点号	int	是
name	名称	varchar2(64)	否
value	值	float	否
status	数据质量	Int	否
refresh_time	刷新时间	int	否

站端：遥信信息表(rtu\_yx\_info)

列名	描述	类型	是否主键
id	点号	int	是
name	名称	varchar2(64)	否
value	值	int	否
status	数据质量	Int	否
refresh_time	刷新时间	int	否

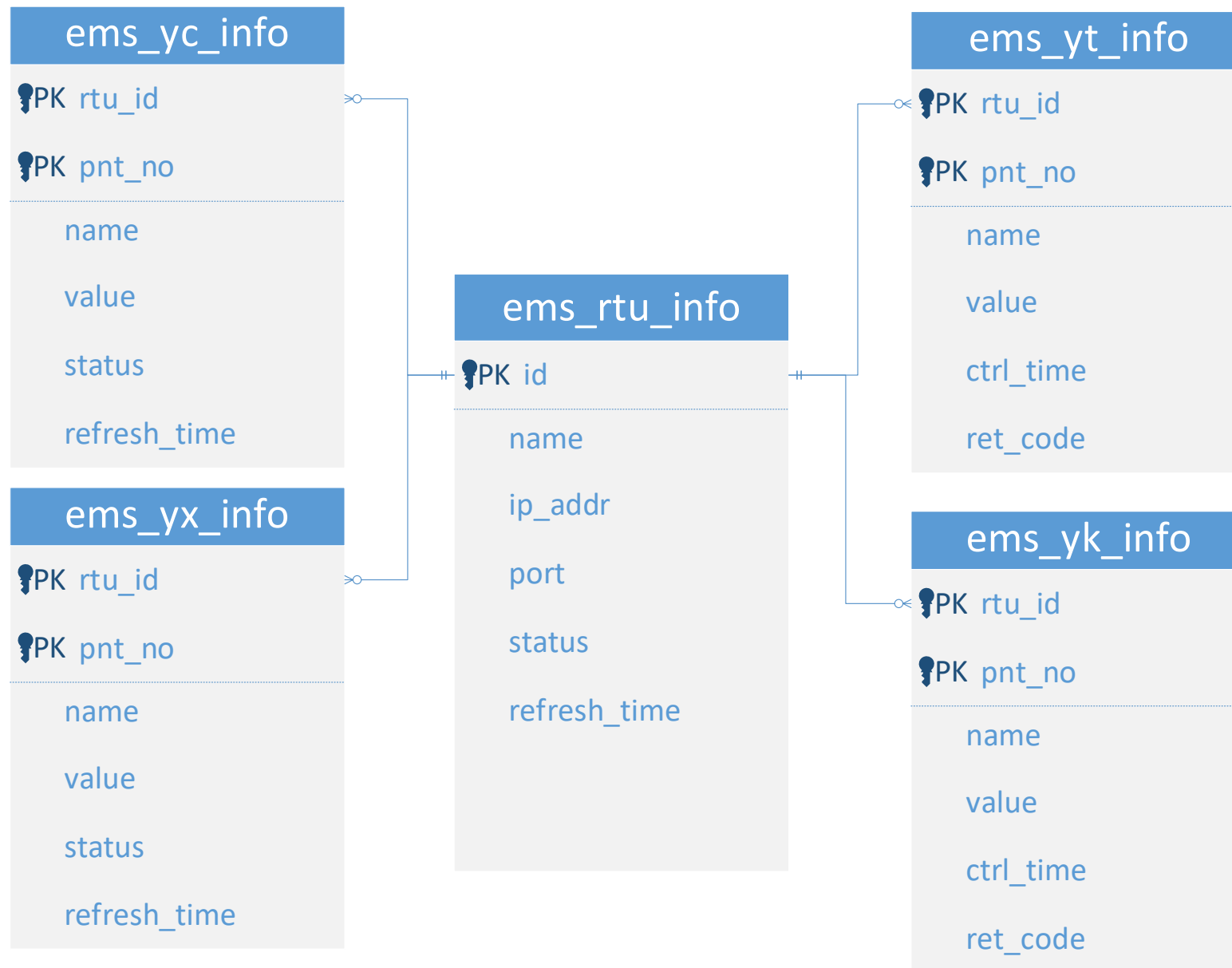
站端：遥调信息表(rtu\_yt\_info)

列名	描述	类型	是否主键
id	点号	int	是
name	名称	varchar2(64)	否
value	值	float	否
refresh_time	指令时间	int	否
ctrl_code	执行结果	int	否

站端：遥控信息表(rtu\_yk\_info)

列名	描述	类型	是否主键
id	点号	int	是
name	名称	varchar2(64)	否
value	值	int	否
refresh_time	指令时间	int	否
ctrl_code	执行结果	int	否

## 主站数据库(ems.db)中的表关系






子站数据库(rtu{n}.db)中的表关系

rtu_yc_info
 PK id
name
value
status
refresh_time

rtu_yx_info
 PK id
name
value
status
refresh_time

rtu_info
 PK id
name
ip_addr
port
status
refresh_time

rtu_yt_info
 PK id
name
value
ctrl_time
ret_code

rtu_yk_info
 PK id
name
value
ctrl_time
ret_code

# 第1步: init 数据库

- **python rtu.py 1 init:** 调用函数auto\_gen\_tables(), 自动创建 rtu\_info 表, 插入数据。
- **python ems.py init:** 调用函数auto\_gen\_tables(), 自动创建 ems\_rtu\_info 表, 插入数据。

```
import sys
from sqlalchemy import create_engine, text

def auto_gen_tables(engine, rtu_id):
    with engine.connect() as sqldb:
        sqldb.execute(text("drop table if exists rtu_info"))
        sqldb.execute(text(
            "create table rtu_info(id int, name text, ip_addr varchar2(64), "
            "port int, status int, refresh_time int)"))

        sqldb.execute(text(
            f"insert into rtu_info (id, name, ip_addr, port, status, refresh_time) "
            f"values({rtu_id}, 'rtu_{rtu_id}', '127.0.0.1', {8800 + rtu_id}, 0, 0)"))
        sqldb.commit()

if __name__ == "__main__":
    rtu_id = 1
    if len(sys.argv) > 1:
        rtu_id = int(sys.argv[1])
    engine = create_engine(f"sqlite:///../db/rtu_{rtu_id}.db")

    if len(sys.argv) > 2 and sys.argv[2] == 'init':
        auto_gen_tables(engine, rtu_id)
        exit(0)
```

rtu.py

```
from sqlalchemy import create_engine, text
import sys

def auto_gen_tables(engine):
    with engine.connect() as sqldb:
        sqldb.execute(text("drop table if exists ems_rtu_info"))
        sqldb.execute(text(
            "create table ems_rtu_info(id int, name text, ip_addr varchar2(64), "
            "port int, status int, refresh_time int)"))

        for rtu_id in range(1, 6):
            sqldb.execute(text(
                f"insert into ems_rtu_info (id, name, ip_addr, port, status, refresh_time) "
                f"values({rtu_id}, 'rtu_{rtu_id}', '127.0.0.1', {8800 + rtu_id}, 0, 0)"))

        results = sqldb.execute(text("select * from ems_rtu_info"))
        for result in results:
            print(result.id, result.name, result.ip_addr,
                  result.port, result.status, result.refresh_time)
        sqldb.commit()

if __name__ == "__main__":
    engine = create_engine("sqlite:///../db/ems.db")

    print(sys.argv)
    if len(sys.argv) > 1 and sys.argv[1] == 'init':
        auto_gen_tables(engine)
        exit(0)
```

ems.py

## 第2步: 启动服务

- **python rtu.py 1:** 自动读取 rtu\_info表, 获取 ip地址和端口号, 创新socket 服务端
- **python ems.py:** 自动读取 ems\_rtu\_info表, 遍历rtu, 获取 ip地址和端口号, 并主动连接rtu服务

```
def get_ip_addr(engine, rtu_id):
    with engine.connect() as sqldb:
        results = sqldb.execute(text(f"select * from rtu_info "
                                     f"where id = {rtu_id}"))
        for result in results:
            return (result.ip_addr, result.port)

def rtu_thread_send_ycyx(engine, conn):
    while True:
        time.sleep(1.0)

if __name__ == "__main__":
    rtu_id = 1
    if len(sys.argv) > 1:
        rtu_id = int(sys.argv[1])
    engine = create_engine(f"sqlite:///../db/rtu_{rtu_id}.db")

    if len(sys.argv) > 2 and sys.argv[2] == 'init':...

    rtu_addr = get_ip_addr(engine, rtu_id)
    print("rtu_addr:", rtu_addr)
    sock = socket.socket()
    sock.bind(rtu_addr)
    sock.listen(10)

    while True:
        conn, status = sock.accept()
        thread = Thread(target=rtu_thread_send_ycyx, args=(engine, conn))
        thread.start()
```

rtu.py

```
def rtu_thread_get_ycyx(engine, rtu):
    try:
        sock = socket.socket()
        print("conn", (rtu.ip_addr, rtu.port))
        sock.connect((rtu.ip_addr, rtu.port))
        while True:
            time.sleep(1.0)
    except Exception as e:
        print("conn error !!", (rtu.ip_addr, rtu.port), e)

if __name__ == "__main__":
    engine = create_engine("sqlite:///../db/ems.db")

    if len(sys.argv) > 1 and sys.argv[1] == 'init':...

    with engine.connect() as sqldb:
        rtus = sqldb.execute(text("select * from ems_rtu_info"))
        thread_list = []
        for rtu in rtus:
            print(rtu)
            thread = Thread(target=rtu_thread_get_ycyx, args=(engine, rtu))
            thread.start()
            break
```

ems.py

## 第3步，修改auto\_gen\_tables()，添加 yc 表

```
with engine.connect() as sqlldb:
    sqlldb.execute(text("drop table if exists ems_yc_info"))
    sqlldb.execute(text(
        f"create table ems_yc_info(rtu_id int, pnt_no int, name text, "
        f"value float, status int, refresh_time int)"))

    for rtu_id in range(1, 6):
        for pnt_no in range(1, 6):
            sqlldb.execute(text(
                f"insert into ems_yc_info (rtu_id, pnt_no, name, value, status, refresh_time) "
                f"values({rtu_id}, {pnt_no}, 'rtu_{rtu_id}_yc_{pnt_no}', 0, 0, 0)"))

    results = sqlldb.execute(text("select * from ems_yc_info"))
    for result in results:
        print(result.rtu_id, result.pnt_no, result.name, result.value,
              result.status, result.refresh_time)
```

**ems.py**

**rtu.py**

```
with engine.connect() as sqlldb:
    sqlldb.execute(text("drop table if exists rtu_yc_info"))
    sqlldb.execute(text(
        f"create table rtu_yc_info(id int, name text, value float, status int, refresh_time int)"))

    for i in range(1, 6):
        sqlldb.execute(text(
            f"insert into rtu_yc_info (id, name, value, status, refresh_time) "
            f"values({i}, 'rtu_{rtu_id}_yc_{i}', 0, 0, 0)"))

    results = sqlldb.execute(text("select * from rtu_yc_info"))
    for result in results:
        print(result.id, result.name, result.value, result.status, result.refresh_time)
```

## 第4步，以1秒为周期，rtu子站的yc传送到ems主站

```
import json
import struct

def send_data(conn, head, data):
    print("send_data", head, data)
    json_data = json.dumps(data).encode("utf8")
    head['size'] = len(json_data)
    json_head = json.dumps(head).encode("utf8")
    head_len = len(json_head)
    data_head = struct.pack('i', head_len)
    conn.send(data_head)
    conn.send(json_head)
    conn.send(json_data)

def update_yc_data(engine, conn):
    with engine.connect() as sqldb:
        results = sqldb.execute(text("select * from rtu_yc_info"))
        data = []
        for result in results:
            data.append([result.id, result.value,
                        result.status, result.refresh_time])
        return send_data(conn, {"type": "update_yc"}, data)

def rtu_thread_send_ycyx(engine, conn):
    while True:
        update_yc_data(engine, conn)
        time.sleep(1.0)
```

rtu.py

```
def recv_data(sock, sqldb, rtu_id):
    head = sock.recv(4)
    if head:
        head_len = struct.unpack("i", head)[0]
        head = sock.recv(head_len)
        head_data = json.loads(head)
        body_len, body_type = head_data['size'], head_data['type']
        body = sock.recv(body_len)
        body_data = json.loads(body)
        if body_type == 'update_yc':
            return recv_yc_data(sqldb, rtu_id, body_data)

def recv_yc_data(sqldb, rtu_id, data):
    for result in data:
        info = f"update ems_yc_info set value = {result[2]} " \
              f"where rtu_id = {rtu_id} and pnt_no= {result[0]}"
        print(info)
        sqldb.execute(text(info))

def rtu_thread_get_ycyx(engine, rtu):
    try:
        sock = socket.socket()
        sock.connect((rtu.ip_addr, rtu.port))
        while True:
            with engine.connect() as sqldb:
                recv_data(sock, sqldb, rtu.id)
    except Exception as e:
        print("conn error !!", (rtu.ip_addr, rtu.port), e)
```

ems.py

## 第5步，ems主站的处理结果返回给rtu子站

```
def recv_yc_result(data):
    print("recv_yc_result", data)

def recv_data(conn, sqldb):
    head = conn.recv(4)
    if head:
        print("recv_data", head)
        head = conn.recv(struct.unpack("i", head)[0])
        head_data = json.loads(head)
        body = conn.recv(head_data['size'])
        body_data = json.loads(body)
        if head_data['type'] == 'return_yx':
            return recv_yc_result(body_data)

def rtu_thread_recv_data(engine, conn):
    with engine.connect() as sqldb:
        while True:
            recv_data(conn, sqldb)
```

```
while True:
    conn, status = sock.accept()
    thread = Thread(target=rtu_thread_send_ycyx, args=(engine, conn))
    thread.start()

    thread = Thread(target=rtu_thread_recv_data, args=(engine, conn))
    thread.start()
```

rtu.py

```
import json
import struct

def send_data(sock, head, data):
    print("send_data", head, data)
    json_data = json.dumps(data).encode("utf8")
    head['size'] = len(json_data)
    json_head = json.dumps(head).encode("utf8")
    head_len = len(json_head)
    data_head = struct.pack('i', head_len)
    sock.send(data_head)
    sock.send(json_head)
    sock.send(json_data)

def recv_yc_data(sqldb, rtu_id, data, sock):
    for result in data:
        info = f"update ems_yc_info set value = {result[2]} " \
            f"where rtu_id = {rtu_id} and pnt_no= {result[0]}"
        print(info)
        sqldb.execute(text(info))

    result = {"result": "ok"}
    send_data(sock, {"type": "return_yx"}, result)
```

ems.py

## 第6步，以1秒为周期，ems主站的yt传送到rtu子站

```
def recv_yt_data(sqlldb, data):
    for result in data:
        info = f"update rtu_yt_info set value = {result[2]} " \
            f"where id = {result[0]}"
        print(info)
        sqlldb.execute(text(info))
    result = {"result": "ok"}
    send_data(conn, {"type": "return_yt"}, result)

def recv_data(conn, sqlldb):
    head = conn.recv(4)
    if head:
        print("recv_data", head)
        head = conn.recv(struct.unpack("i", head)[0])
        head_data = json.loads(head)
        body = conn.recv(head_data['size'])
        body_data = json.loads(body)
        if head_data['type'] == 'return_yc':
            return recv_yc_result(body_data)
        if head_data['type'] == 'update_yt':
            return recv_yt_data(sqlldb, body_data)
```

rtu.py

```
if body_type == 'recv_yt':
    return recv_yt_result(rtu_id, body_data)

def recv_yt_result(rtu_id, data):
    print("recv_yt_result", rtu_id, data)
```

```
def update_yt_data(sqlldb, sock, rtu_id):
    results = sqlldb.execute(text(f"select * from ems_yt_info "
        f"where rtu_id = {rtu_id}"))
    data = []
    for result in results:
        data.append([result.pnt_no, result.name, result.value,
            result.ret_code, result.ctrl_time])
    send_data(sock, {"type": "update_yt"}, data)

def rtu_thread_update_ykyl(engine, sock, rtu):
    print("rtu_thread_update_ykyl...")
    with engine.connect() as sqlldb:
        while True:
            print("update_yt_data...")
            update_yt_data(sqlldb, sock, rtu.id)
            time.sleep(1.0)
```

```
thread = Thread(target=rtu_thread_update_ykyl, args=(engine, sock, rtu))
thread.start()
```

ems.py

第7步，类似操作，处理 $yx$



第8步，类似操作，处理 $y_k$