多线程IT黑马

2020年9月18日 16:53

第一课：线程的介绍

2020年9月17日 11:32

1.2 为什么使用多线程?

进程是分配资源的最小单位，一旦创建一个进程就会分配一定的资源，就像跟两个人聊QQ就需要打开两个QQ软件一样是比较浪费资源的.

线程是**程序执行的最小单位**，实际上进程只负责分配资源，而利用这些资源执行程序的是线程，也就说进程是线程的容器，**一个进程中最少有一个线程**来负责执行程序. 同时线程自己不拥有系统资源，只需要一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程**共享进程所拥有的全部资源**. 这就像通过一个QQ软件(一个进程)打开两个窗口(两个线程)跟两个人聊天一样，实现多任务的同时也节省了资源.

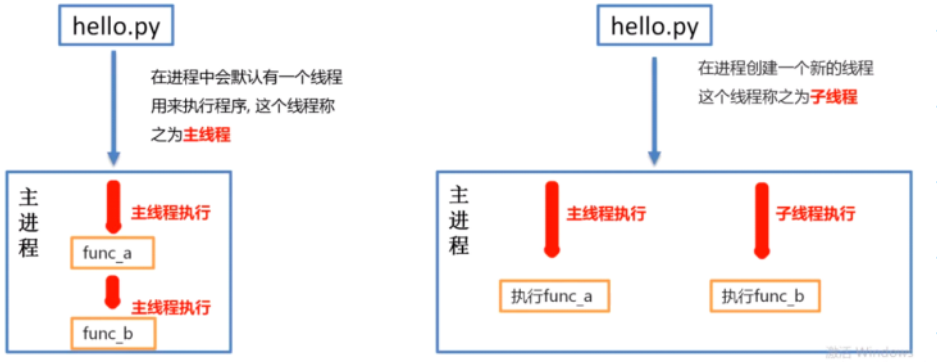
1.3 多线程的作用

hello.py

```python
def func_a():
    print("任务A")

def func_b():
    print("任务B")

func_a()
func_b()
```

同样需求使用多线程完成

1.4 多线程的作用



第二课：使用多线程执行任务

2020年9月19日 9:36

2.1 线程的创建步骤

1. 导入线程模块

   import threading

2. 通过线程类创建线程对象

   线程对象 = threading.Thread(target=任务名)

3. 启动线程执行任务

   线程对象.start()

2.2 通过线程类创建线程对象

线程对象 = threading.Thread(target=任务名)

参数名	说明
target	执行的目标任务名,这里指的是函数名(方法名)
name	线程名，一般不用设置
group	线程组，目前只能使用None

2.3 线程创建与启动的代码

```
# 创建子线程
sing_thread = threading.Thread(target=sing)
# 创建子线程
dance_thread = threading.Thread(target=dance)
# 启动线程
sing_thread.start()
dance_thread.start()
```

第三课：线程执行带有参数的任务

2020年9月19日 9:43

3.1 线程执行带有参数的任务

参数名	说明
args	以元组的方式给执行任务传参
kwargs	以字典方式给执行任务传参

3.2 args参数的使用

```
# target: 线程执行的函数名
# args: 表示以元组的方式给函数传参
sing_thread = threading.Thread(target=sing, args=(3,))
sing_thread.start()
```

3.3 kwargs参数的使用

```
# target: 线程执行的函数名
# kwargs: 表示以字典的方式给函数传参
dance_thread = threading.Thread(target=dance, kwargs={"count": 3})
# 开启线程
dance_thread.start()
```

```
import time
import threading


def sing(num):
    for i in range(num):
        print("唱歌...")
        time.sleep(1)


def dance(count):
    for i in range(count):
        print("跳舞...")
        time.sleep(1)


if __name__ == '__main__':
    # args: 以元组的方式给执行任务传递参数
    sing_thread = threading.Thread(target=sing, args=(3,))
    # kwargs : 以字典方式给执行任务传递参数
    dance_thread = threading.Thread(target=dance, kwargs={"count": 2})
```

```
sing_thread.start()
dance_thread.start()
```

第四课：主线程和子线程的结束顺序

2020年9月19日 9:49

```
import time
import threading


def work():
    for i in range(10):
        print("工作...")
        time.sleep(0.2)


if __name__ == '__main__':
    sub_thread = threading.Thread(target=work)
    sub_thread.start()

    # 主线程等待1s,后结束
    time.sleep(1)
    print("主线程结束了...")


# 结论：主线程会等待所有的子线程执行结束再结束
```

4.1 设置守护主线程

要想主线程不等待子线程执行完成可以设置守护主线程。

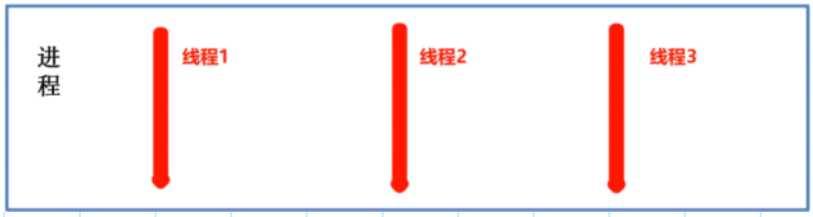
```
# 设置守护主线程方式1. daemon=True 守护主线程
work_thread = threading.Thread(target=work, daemon=True)
# 设置守护主线程方式2
# work_thread.setDaemon(True)
work_thread.start()
# 主线程延时1秒
time.sleep(1)
print("over")
```

```python
if __name__ == '__main__':
    # sub_thread = threading.Thread(target=work)
    # sub_thread.start()

    # 主线程结束不想等待子线程结束再结束，可以设置子线程守护主线程
    #   1. threading.Thread(target=work, daemon=True)
    sub_thread = threading.Thread(target=work, daemon=True)
    sub_thread.start()

    # 主线程等待1s,后结束
    time.sleep(1)
    print("主线程结束了...")

# 主线程结束不想等待子线程结束再结束，可以设置子线程守护主线程
#   1. threading.Thread(target=work, daemon=True)
# sub_thread = threading.Thread(target=work, daemon=True)
sub_thread = threading.Thread(target=work)
sub_thread.setDaemon(True)
```

第五课：线程间的执行顺序

2020年9月19日 9:55

线程间执行顺序



5.1 线程之间执行是无序的

```
for i in range(5):
    sub_thread = threading.Thread(target=task)
    sub_thread.start()
```

5.2 获取当前的线程信息

```
# 通过current_thread方法获取线程对象
current_thread = threading.current_thread()
# 通过current_thread对象可以知道线程的相关信息，例如被创建的顺序
print(current_thread)
```

```python
import threading
import time

def mask():
    time.sleep(1)
    # current_thread: 获取当前线程的线程对象
    thread = threading.current_thread()
    print(thread)


if __name__ == '__main__':
    for i in range(5):
        sub_thread = threading.Thread(target=mask)
        sub_thread.start()
# 结论：多线程之间执行是无序 由cpu调度
```

```
<Thread(Thread-1, started 290248)>
<Thread(Thread-5, started 279740)>
<Thread(Thread-2, started 279528)>
<Thread(Thread-4, started 287912)>
<Thread(Thread-3, started 279948)>
```

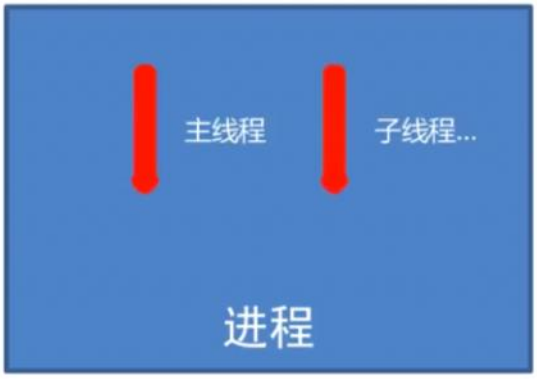
第六课：进程和线程的对比

2020年9月19日 10:03

6.1 关系对比

1. 线程是依附在进程里面的，没有进程就没有线程。

2. 一个进程默认提供一条线程，进程可以创建多个线程。



6.2 区别对比

1. 创建进程的资源开销要比创建线程的资源开销要大

2. 进程是操作系统资源分配的基本单位，线程是CPU调度的基本单位

3. 线程不能够独立执行，必须依存在进程中

6.3 优缺点对比

1. 进程优缺点:

   优点：可以用多核

   缺点：资源开销大

2. 线程优缺点:

   优点：资源开销小

   缺点：不能使用多核

第七课：教学视频文件夹高并发copy器

2020年9月19日 10:06

案例：需求分析

① 目标文件夹是否存在，如果不存在就创建，如果存在则不创建
② 遍历源文件夹中所有文件，并拷贝到目标文件夹
③ 采用多线程实现多任务，完成高并发拷贝

1、定义源文件夹所在的路径、目标文件夹所在路径

```
# 1、定义源文件目录和目标文件夹的目录
source_dir = "python教学视频"
dest_dir = "/home/python/桌面/test"
```

2、创建目标文件夹

```
try:
    # 2、创建目标文件夹目录
    os.mkdir(dest_dir)

except:
    print("目标文件夹已经存在，未创建~")
```

3、通过os.listdir 获取源目录中的文件列表

```
# 3、列表得到所有的源文件中的文件
file_list = os.listdir(source_dir)
print(file_list)
```

4、遍历每个文件，定义一个函数，专门实现文件拷贝

```
# 4、for 循环，依次拷贝每个文件
for file_name in file_list:
    copy_work(file_name, source_dir, dest_dir)
```

5、采用进程实现多任务，完成高并发拷贝

```
    # 4、for 循环，依次拷贝每个文件
    for file_name in file_list:
        # copy_work(file_name, source_dir, dest_dir)
        sub_thread = threading.Thread(target=copy_work, args=(file_name,
source_dir, dest_dir))
        sub_thread.start()
```

1、拼接源文件和目标文件所在的路径

```
def copy_work(file_name, source_dir, dest_dir):

    # 拼接路径
    source_path = source_dir+"/"+file_name

    dest_path = dest_dir+"/"+file_name
```

2、打开源文件、创建目标文件

```
def copy_work(file_name, source_dir, dest_dir):
    # 拼接路径
    source_path = source_dir+"/"+file_name
    dest_path = dest_dir+"/"+file_name

    print(source_path, "----->", dest_path)
    # 打开源文件、创建目标文件
    with open(source_path,"rb") as source_file:
        with open(dest_path,"wb") as dest_file:
```

3、读取源文件的内容并且写入到目标文件中（循环）

```
def copy_work(file_name, source_dir, dest_dir):
            .....
            while True:
                # 循环读取数据
                file_data = source_file.read(1024)
                if file_data:
                    # 循环写入到目标文件
                    dest_file.write(file_data)
                else:
                    break
```