熟悉python常用的各个模块，并以此为基础，实现一个简单聊天室功能。

- 异常处理
- 基于TCP协议的socket编程通用流程
- 多线程
- I/O模型

2 常用模块介绍

- 异常处理
- 基于TCP协议的socket编程通用流程
- 多线程
- I/O模型

2.1 异常处理

异常(Exception)是一种为了处理错误或其他异常条件而打破代码块的正常控制流的方法

- 当python解释器检测到一个运行时错误时，会抛出异常（如除0错误）
- 程序也可以使用raise语句显式抛出一个异常

Python使用“终止”(termination)模式来进行异常处理

1. 异常处理可以帮助定位发生错误的原因，并在外层继续执行，但是不能修复错误并进行重试
2. 如果异常没有进行处理，则会中断程序的运行

2.1 异常处理

异常处理使用 try ... except语句。

```
try:
        code blocks
except ExceptionClass1 as name:
        # handle exception

except ExceptionClass2 as name:
        # handle exception

finally:
        cleanup
```

Demo: exception.py

with语句：

- 当执行with语句时，我们可以定义一个上下文管理器(context manage)
- 上下文管理器在运行代码块时，对进入、离开运行环境(runtime)进行了处理
- 常用于关闭文件，释放锁等情况

```
def try_style():
    try:
        f = open('test.txt', 'r')
        print('openfile ok.')

    finally:
        f.close()
```

```
def with_style():
    with open('test.txt', 'r') as f:
        print('openfile ok.')
```

二者等价

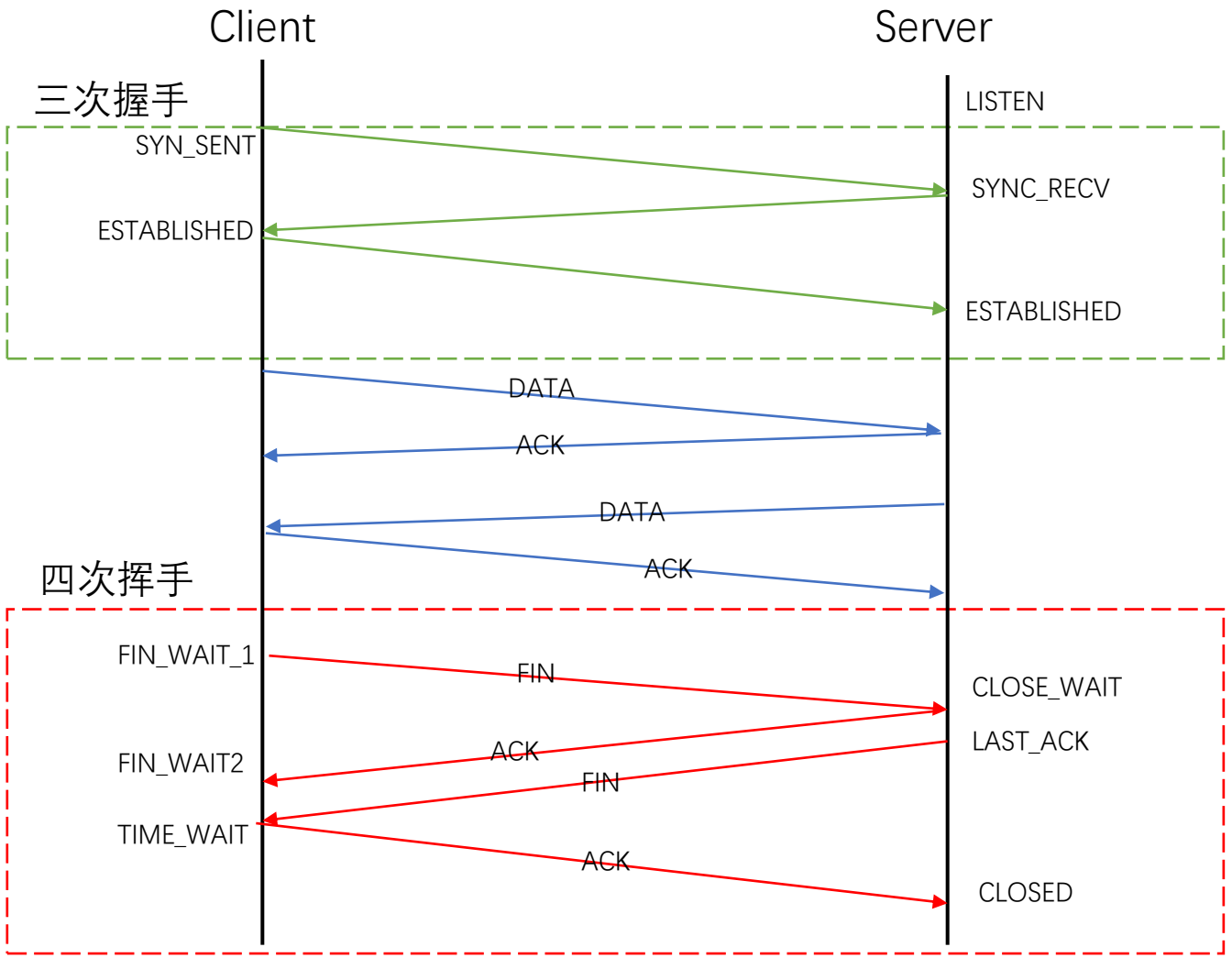
sys.exceptionhook: 自定义默认异常输出

Demo: exception_hook.py

2常用模块介绍

- 异常处理
- 基于TCP协议的socket编程通用流程
- 多线程
- I/O模型

2.2 基于TCP协议的socket编程通用流程



2.2 基于TCP协议的socket编程通用流程

- 相关接口实现在module socket中

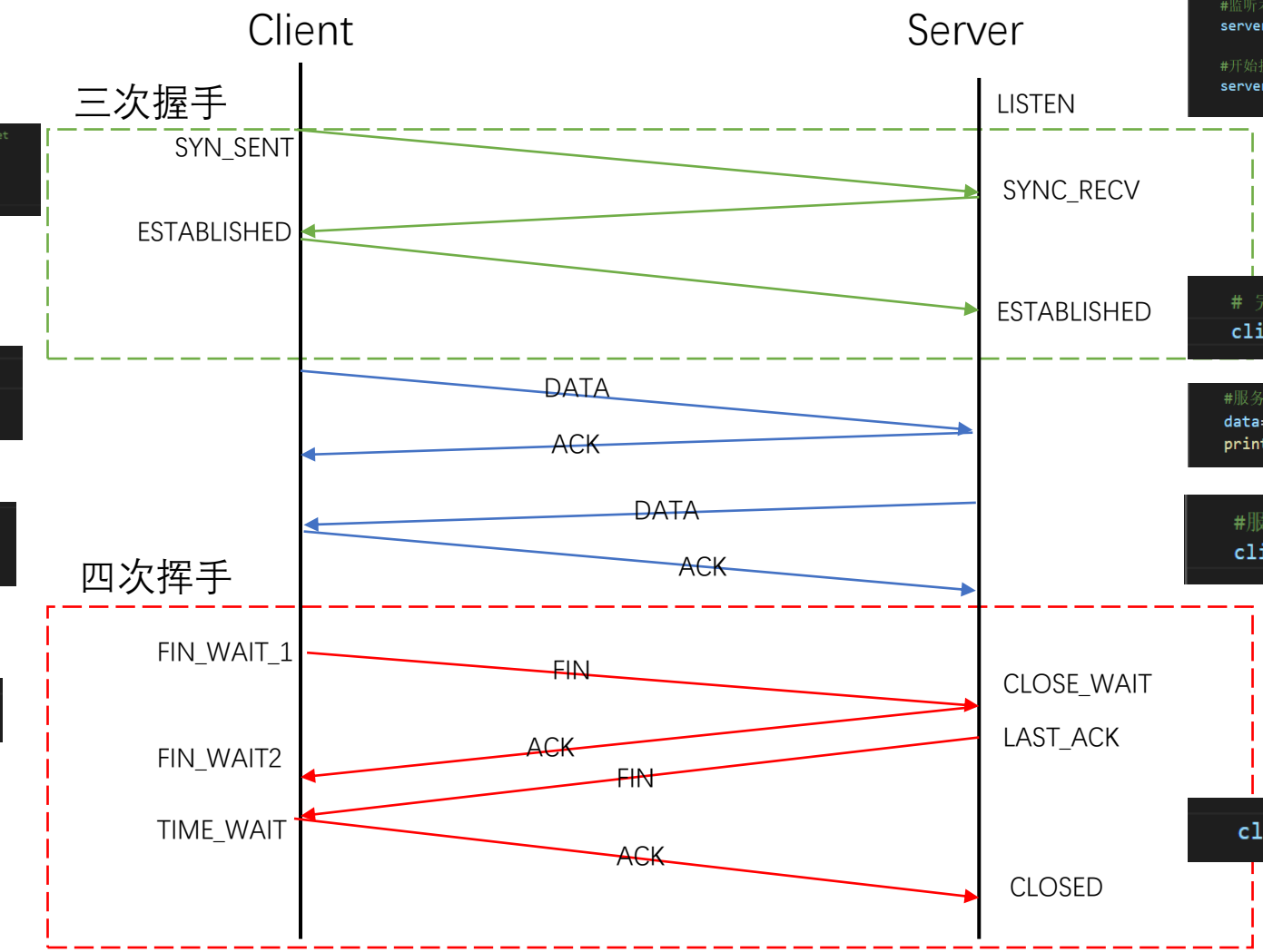
```
# 创建使用IPV4(socket.AF_INET)、TCP(socket.SOCK_STREAM)协议的socket
cli = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#连接127.0.0.1:8080的服务器
cli.connect(('127.0.0.1', 8080))
```

```
# 向服务器发送消息
cli.send('hello'.encode())
```

```
# 接收服务器的消息
data = cli.recv(1024)
print(f'client recv server data {data}')
```

```
client.close()
```




```
# 创建使用IPV4(socket.AF_INET)、TCP(socket.SOCK_STREAM)协议的socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#监听本地的8080端口
serversocket.bind(('0.0.0.0', 8080))

#开始接受连接
serversocket.listen(5)
```

```
# 完成三次握手，创建与客户端绑定，并进行通信的套接字
client, addr = serversocket.accept()
```

```
#服务器接收客户端发送的消息
data=client.recv(1024)
print(f'server recv data is : {data}')
```

```
#服务器发送消息给客户端
client.send('world'.encode())
```

```
client.close()
```

2.2 基于TCP协议的socket编程通用流程

- 完整代码: echo_server_1.py && echo_client_1.py

问题：

1. 怎么处理断线问题，调用close的时机
2. 服务器是否能够同时处理多个客户端

问题1: 怎么处理断线问题，调用close的时机

- 调用recv收到长度为0的数据
- 调用send时抛出异常ConnectionResetError

- 完整代码: echo_server_2.py && echo_client_2.py

问题2: 服务器是否能够同时处理多个客户端

- accept、read会互相阻塞后续操作

- 完整代码: echo_server_3.py && echo_client_3.py

2常用模块介绍

- 异常处理
- 基于TCP协议的socket编程通用流程
- 多线程
- I/O模型

threading.thread

在python中提供了一种并发执行的方法。但是由于全局解释器锁(Global Interpreter Lock, GIL)的存在，并不能提供类似c/c++线程中的性能，因此一般用于I/O密集型程序中。

全局解释器锁(Global Interpreter Lock, GIL)

在python虚拟机的实现中，解释器为了简化数据模型的实现，对整个解释器进行加锁，保证在同一个时刻，只有一个线程能够执行字节码。但是代价就是降低了多处理器提供的并发能力。

例外情况：

1. 在进行I/O操作时会释放GIL
2. 在实现的C扩展程序中可以根据情况释放GIL

适用python线程的情况：

- 为了提高程序的实时性
- I/O密集型的程序，需要提高并发性能

创建线程对象的两种方法：

1. **threading.Threading**(*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

   - *group:* 预留参数，无作用
   - *target:* threading启动后执行的callable对象（如函数等）
   - *name:* 线程名 “Thread-<name>”
   - *args:* 传递给target的参数列表
   - *kwargs:* 传递给target的参数字典
   - *daemon:*是否为守护线程

创建线程对象的两种方法：

2. 实现自定义线程类，继承自threading.Threading

- 不能重载除__init__以及run以外的其他函数
- 将线程逻辑实现在run函数中
- 如果重载了__init__函数，必须在其中调用基类的__init__函数

关键方法：

start()*:* 激活线程,调用线程对象的run()函数。默认run()函数会调用target对象

join(*timeout=None*)*:* 等待线程结束或者超时

demo: thread_1.py

线程间同步：

- Lock
- RLock
- 条件变量(Condition object)
- 信号量(Semaphore object)
- Barrier Object
- queue

线程间同步： Lock

与互斥锁mutex类似，拥有“lock” 与 “unlock”状态。会自动根据平台选择性能最高的版本。

- 使用**acquire**(*blocking=True, timeout=-1*)去占有锁
- 使用**release**()释放锁

	Lock	Unlock
blocking==True timeout==-1	阻塞直至锁被释放	立即返回True
blocking==True timeout != -1	阻塞直至锁被释放 或者超时	立即返回True
blocking== False timeout==-1	立即返回False	立即返回True
blocking = False timeout !=-1	立即返回False	立即返回True

线程间同步： RLock

与Lock相似，但是可以被持有锁的线程多次调用acquire以及release。**R**代表可以被相同线程重入(reentrant)。

demo: thread_2.py

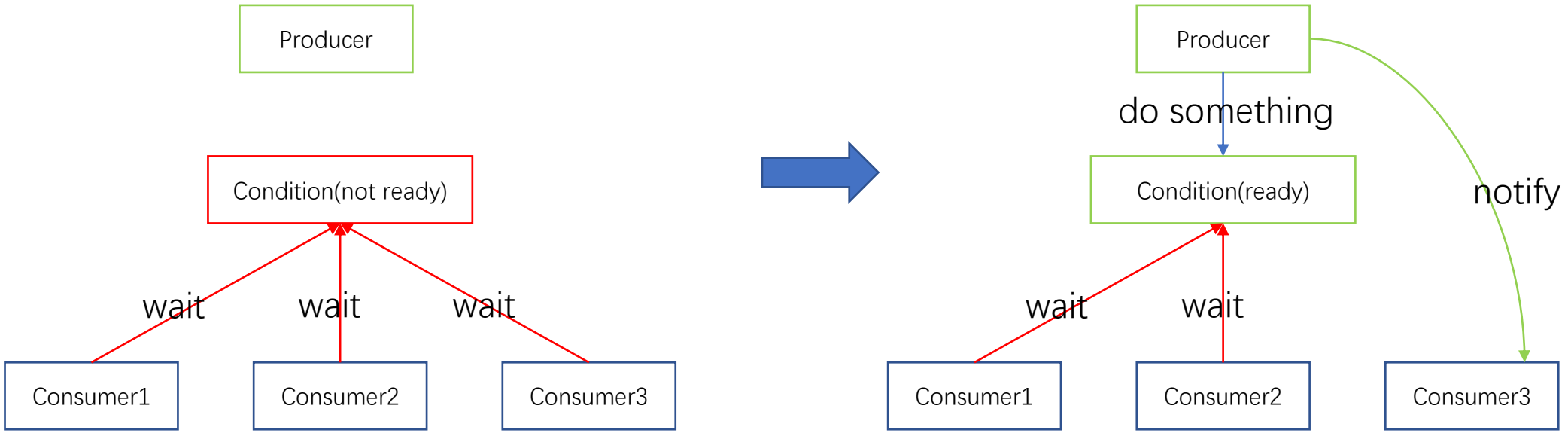
线程间同步： Lock

与互斥锁mutex类似，拥有“lock” 与 “unlock”状态。会自动根据平台选择性能最高的版本。

- 使用**acquire**(*blocking=True, timeout=-1*)去占有锁
- 使用**release**()释放锁

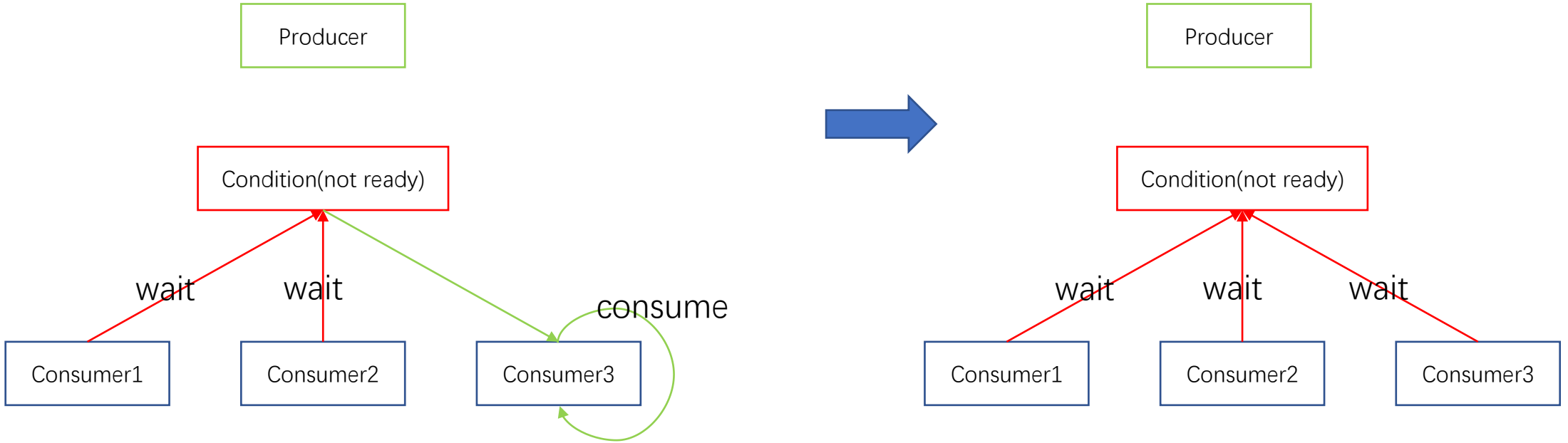
线程间同步： 条件变量

常用于生产者-消费者模式。当生产者使某些状态发生变化时，通知(notify)消费者进行处理。



线程间同步： 条件变量

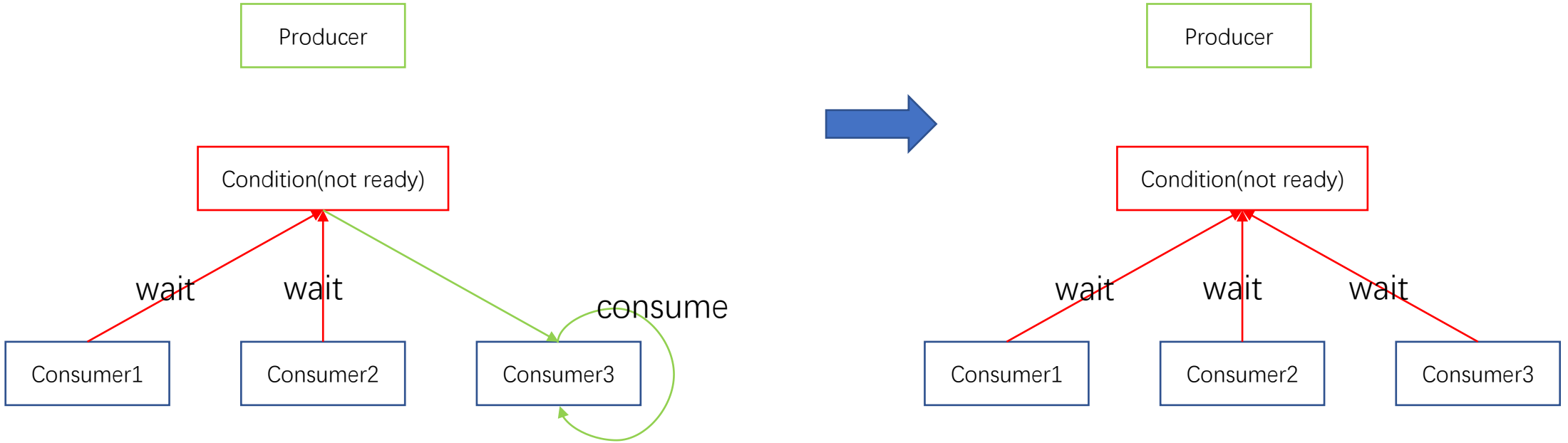
常用于生产者-消费者模式。当生产者使某些状态发生变化时，通知(notify)消费者进行处理。



demo: thread_3.py

线程间同步： 条件变量

常用于生产者-消费者模式。当生产者使某些状态发生变化时，通知(notify)消费者进行处理。



demo: thread_3.py

线程间同步： 信号量

信号量管理了一个内部计数器。

- 调用acquire()时，计数器减一；当调用release()时，计数器加一。
- 当在信号量的值为0时调用acquire,当前线程会block，直至某个线程调用release
- 一般用于资源有限的场景（例如数据库连接等）

线程间同步： Barrier Object

当参与的线程全部调用wait后，取消阻塞状态。

- **threading.Barrier**(*parties, action=None, timeout=None*)　　# 创建对象
  - *parties:*参与线程数
  - *action:*当取消阻塞状态时，被其中一个线程调用
  - *timeout:*超时时间

- **wait**(*timeout=None*)
  - 当所有参与线程调用过此函数后，它们同时取消阻塞状态

demo: thread_4.py

线程间同步： 以with的方式使用lock, RLock, conditions, semaphores

- 简化写法，避免人为原因忘记release后导致死锁
- 防止因为异常等情况未执行到release相关代码，导致死锁

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

```
with some_lock:
    # do something...
```

二者等价

线程间同步： queue

queue模块实现了在多线程情况下能够安全使用多生产者-多消费者队列。它提供了以下三类队列:

1. queue.Queue : FIFO队列
2. queue.LifoQueue： LIFO队列（stack）
3. queue.PriorityQueue: 优先级队列

思考：如何使用多线程技术，使服务器能够同时响应多个客户端

demo : echo_server_4.py && echo_client_4.py

2常用模块介绍

- 异常处理
- 基于TCP协议的socket编程通用流程
- 多线程
- I/O模型

思考：为什么我们在echo_server_4中需要使用多线程模型才能同时响应多个客户端

思考：为什么我们在echo_server_4中需要使用多线程模型才能同时响应多个客户端

- I/O操作不能立即完成，需要时间甚至无法完成
- 调用I/O操作的接口只能等待完成或者超时后才能执行后面的流程

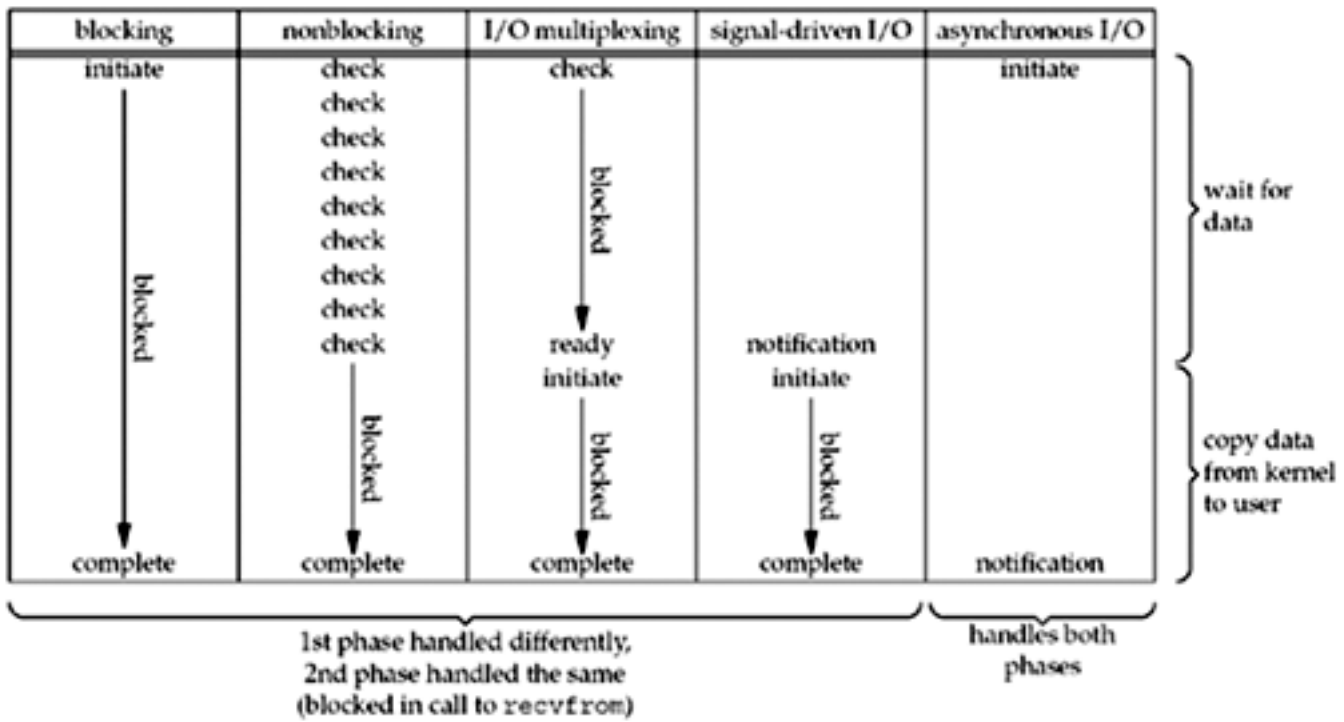
我们将这类I/O操作模型成为阻塞式I/O (blocking I/O)

I/O操作的流程

1. 等待数据就位
2. 将数据从内核拷贝至用户空间

I/O模型

1. 阻塞式同步I/O （blocking I/O）
2. 非阻塞式I/O （nonblocking I/O）
3. I/O多路复用（I/O multiplexing）
4. 异步I/O (asynchronous I/O)
5. 信号驱动I/O (signal-driven I/O)



图自《unix网络编程》

使用I/O多路复用改造服务器：selectors

- **selectors.DefaultSelector**() 创建一个I/O多路复用
- **register**(*object, event_type, data*) 注册一个监听事件
  - *object:* 文件或者网络套接字
  - *event_type:* EVENT_READ 读就绪； EVENT_WRITE(写就绪)
  - *data:* 附带参数
- **unregister**(*object*) 取消一个监听事件

- **select**(*timeout=None*) 返回已经就绪的对象或者超时
  - *timeout <= 0* 立即返回所有就绪的对象
  - *timeout == None* 阻塞直至有对象就绪
  - *timeout > 0* 阻塞直至有对象就绪或者超时

demo echo_server_5.py && echo_client_5.py

思考：多线程与I/O多路复用的优劣？

思考：多线程与I/O多路复用的优劣？

	优势	劣势
多线程	代码简单直观	线程之间交互需要加锁，带来性能的影响 线程数过大时，线程切换可能带来额外消耗
I/O多路复用	所有逻辑运行在单线程中，不同链接之间交互简单高效	单线程仍有性能上限（?）

课后练习：在echo_server_5.py以及 echo_client_5.py的基础上，实现某个用户发送消息给服务器后，服务器广播给其他所有用户，并显示在客户端控制台的功能。