1、问题描述

1.1 概述

本次生产者-消费者模型以上一次实验实现的模型为基础（已实现进程间通信、多线程、阻塞队列），对队列继续进行改进，实现一个消息队列中心作为服务器，对生产者消费者客户端采用订阅推送的模式。实现消息的永久化存储，即使服务器意外断连，也能保证生产的数据一定会被消费。

1.2 本次实验思路

（1）生产者：

① 生产的消息分类：

使用一个一维字符串数组存储所有的tag，可以设置每一个tag具体是什么。

② 多线程处理：

将要求生产的1000000个随机数使用数据分解的方式，平均分给线程池中 的线程生产（此处由于生产者的速度快于消费者，仅分解为了2线程）。生产的消息字符串带上了此消息的tag和该消息在此tag中的编号。

（2）消费者：

① 订阅：

使用一个长度为4的一维字符串数组存储该消费者对应订阅的tag,（长度为4表示最多订阅4个tag）。第一次发送的消息是订阅消息，使用{login:userID,[tag...]}的格式，指定了消费者id和订阅的tag.相当于在服务器注册一个账户。

② 获取信息：

订阅后回调信息采用不断发送get命令，如果服务器没有消息发过来就等待，直到有消息便接收处理。

③ 多线程处理：

处理数据时采用数据分解的方式，将收到的数据平均分给线程池中线程进行数据处理。

（3）队列：

① ServerSocket监听请求：

使用ServerSocket监听生产者或消费者的Socket请求，一旦接收到一个socket，则分配一个线程池中的线程对请求处理，将接收的socket作为参数传递给线程执行的方法，在方法内部对接收到的消息进行识别和处理。

② 多线程处理：

使用容量为100的线程池，接收到一个socket，则分配一个线程池中的线程对请求处理。

③ 队列存储：

使用线程安全的ConcurrentHashMap和阻塞队列BlockingQueue，在hashmap中每个tag对应一个阻塞队列。

④ 处理生产者消息：

如果收到的是生产者的消息，分为生产produce消息和生产结束end消息。如果收到produce消息：使用消息管理类对消息字符串进行处理，包装成Message类；然后使用队列管理类，将Message的tag和该Message对象作为参数调用方法，如果没有该tag就在哈希表创建新的<tag,queue>，如果有则直接将该Message对象放入哈希表中对应tag的阻塞队列中。如果收到end消息：则将end返回给生产者，表示此次操作结束。

⑤ 处理消费者消息：

如果收到的是消费者的消息，分为login注册消息和get获取消息。如果收到get消息，则先查看文件是否有上一次没有处理完的消息，如果有，则将文件中的消息按行读取，包装成Message类，对应每一个tag加入对应的队列中。然后根据消费者消息中所带的注册时消费者id，在消费者哈希表中找到对应所订阅的tag数组，将此tag数组和作为参数传递给队列处理类中的方法，将对应的Message对象拆成字符串传递给消费者；如果文件中没有消息，则直接从队列中回调给消费者。

⑥ 数据持久化：

使用一个关机钩子，在JVM结束前，执行将队列中的数据写入文件的操作，保证数据持久化且不易丢失。

1.3 解决的问题

(1) 数据持久化：

消息持久化保证数据不会丢失，即使在程序异常中断也能保持数据的完整性。

(2) 线程安全：

使用了BlockingQueue阻塞队列和ConcurrentHashmap保证线程安全，高效率实现线程阻塞唤醒，保证线程安全。

(3) 模块解耦：

生产者，消费者，队列是各自独立的模块，生产者和消费者之间借助队列消息中心服务器数据传输，使得两者之间解耦，可以独立修改和优化每个模块，而不影响整个系统的运行。

(4) 提高系统灵活性和可扩展性：

由于生产者和消费者之间的解耦，可以根据需求动态调整生产者和消费者的数量或者速率，从而更好地适应系统负载的变化。消费者订阅-推送模式使得系统更具弹性和灵活性，消费者可以选择订阅自己感兴趣的消息类型，而不需要每个消费者都处理所有消息。

(5) 平衡系统负载：

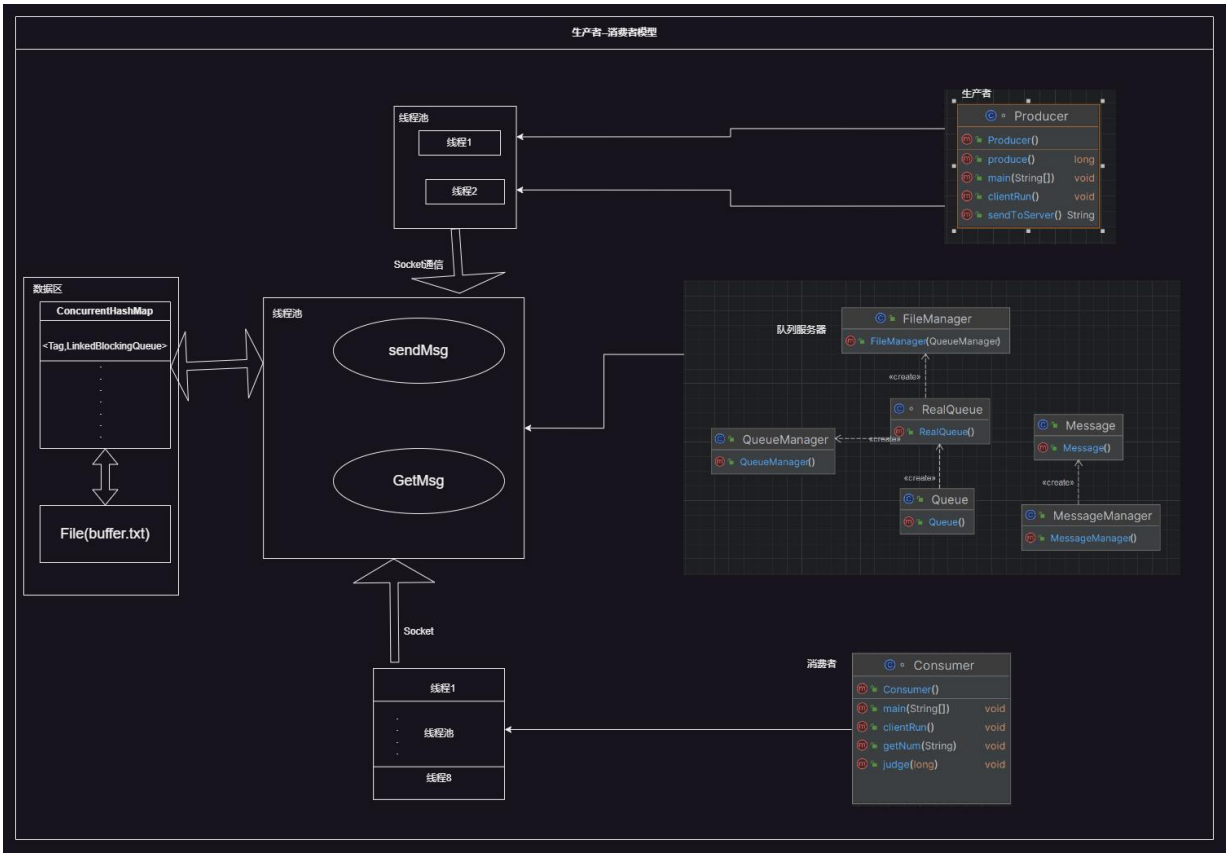
生产者和消费者模型可以用于缓解不同速率下生产和消费的不匹配。通过队列作为中间存储，生产者和消费者可以按照各自的速率进行操作，不至于因为速度不匹配导致系统性能下降或资源浪费。

(6) 提高系统的响应速度：

通过异步操作和并发处理，提高了系统的响应速度。生产者可以不断产生数据，而消费者可以在数据可用时立即处理，提高系统整体的效率。

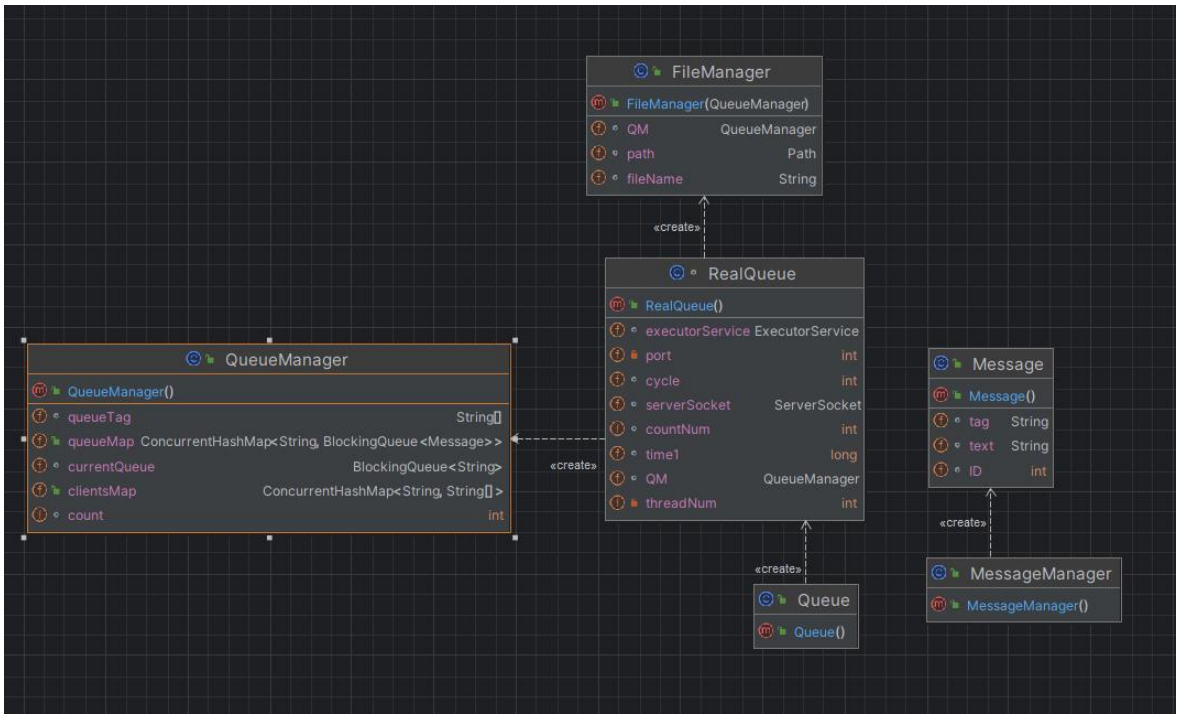
2、系统架构

2.1 整体系统架构图



该生产者消费者模型由生产者、消费者作为客户端，队列作为服务器组成，其中生产者与消费者可以根据需要多开。在系统中，生产者使用线程池多线程产生数据并通过socket与服务器通信；消费者使用线程池多线程消费数据并通过socket与服务器通信。在服务器中采用线程池异步接收并处理来自客户端的socket请求，其中对生产者的请求使用sendMsg处理，消费者的请求使用getMsg处理。在线程池中处理的数据来自一个并发安全的哈希表，其中键为tag，值为阻塞队列。此表会根据需要在进程启动接收到第一个消费者请求时，或者JVM关闭时根据情况与文件File进行数据交换。

2.2 队列服务器进程的UML类图：


FileManager
FileManager(QueueManager)
QM QueueManager
path Path
fileName String
«create»
RealQueue
RealQueue()
executorService ExecutorService
port int
cycle int
serverSocket ServerSocket
countNum int
time1 long
QM QueueManager
threadNum int
QueueManager
QueueManager()
queueTag String[]
queueMap ConcurrentHashMap<String, BlockingQueue<Message>>
currentQueue BlockingQueue<String>
clientsMap ConcurrentHashMap<String, String[]>
count int
«create»
Message
Message()
tag String
text String
ID int
«create»
MessageManager
MessageManager()
«create»
Queue
Queue()

3、关键技术实现

3.1 缓存队列的实现

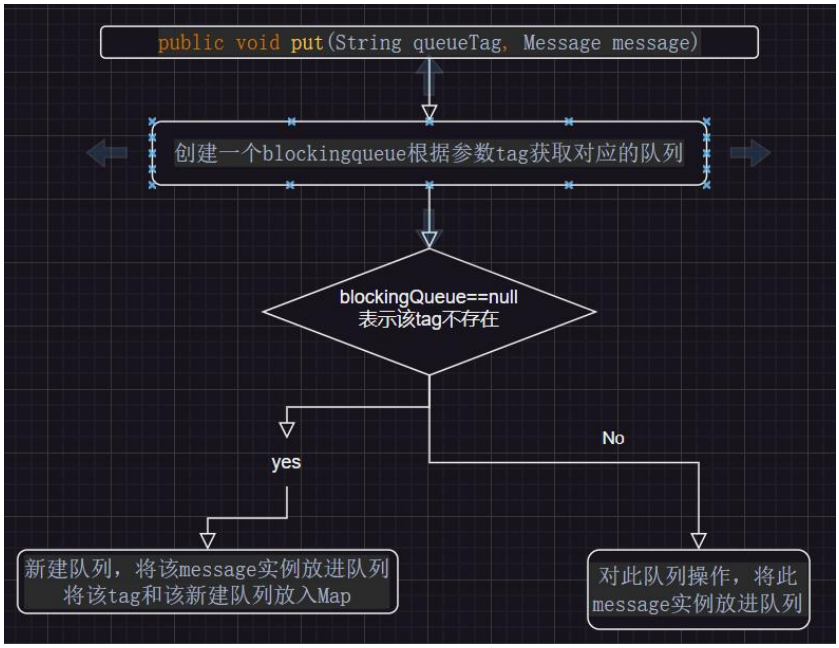
(1) 队列映射管理：

使用 ConcurrentHashMap<String, BlockingQueue<Message>> 存储队列。键是标签 queueTag，代表消息的分类，值是具有不同标签的消息队列 BlockingQueue<Message>。

```
String[] queueTag =new String[20];//消息分类，上限20个
public ConcurrentHashMap<String, BlockingQueue<Message>> queueMap;
BlockingQueue<String> currentQueue = new LinkedBlockingQueue<>( capacity: 1);
```

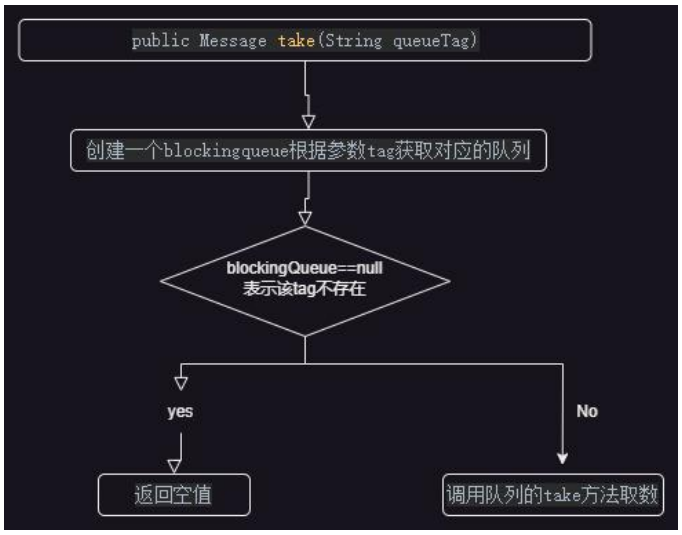
(2) 消息入队 put 方法：

如果队列中不存在特定标签的队列，则创建一个新的 LinkedBlockingQueue 并将消息放入其中，然后将该队列放入 queueMap 中。如果队列已经存在，直接将消息放入对应的队列中。



(3) 消息出队 take 方法

如果队列中不存在特定标签的队列，则根据需求进行处理，可以返回 null 或者抛出异常。如果队列存在，从对应的队列中取出消息。



3.2　异步操作与多线程并发

(1)　线程池的创建和管理

在队列、生产者、消费者均使用了线程池在构造函数中使Executors.newFixed ThreadPool (threadCount) 创建固定大小的线程池，管理着该线程池，通过通过 executorService.execute() 执行任务。主程序通过 executorService.execute() 方法执行异步任务，将客户端的处理放入线程池中处理，实现多线程并发。

```
private int threadNum =100;//线程数
ExecutorService executorService = Executors.newFixedThreadPool(threadNum);
```

(2)　异步消息处理

通过线程池处理接收到的客户端连接，使得每个连接请求在独立的线程中执行，从而实现异步处理多个客户端的请求。serverRun(Socket client) 方法处理客户端发送的消息，根据消息内容执行不同的逻辑操作。

```
try {
    client = serverSocket.accept();//接受请求
} catch (IOException e) {
    e.printStackTrace();
}
System.out.println("连接成功！");
Socket currentClient = client;
executorService.execute(() -> {
        try {
            System.out.println("连接成功！");
            serverRun(currentClient);
            currentClient.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
 });
```

3.3 线程阻塞与唤醒

(1) BlockingQueue<String> currentQueue 对象：

从map中获取对应tag的queue保存在此对象中，而 非直接对map操作。控制对队列的访问，以保证线程的同步和阻塞唤醒操作。

(2) LinkedBlockingQueue：

主要用到了该类阻塞添加元素（put 操作）和阻塞获取元素（take操作）的特性：当队列满时，执行 put 方法会导致调用线程被阻塞，直到队列有空间可用或者线程被中断。当队列为空时，执行 take 方法会导致调用线程被阻塞，直到队列中有元素可用或者线程被中断。

(3) Queue类中的客户端处理方法queueRun( )：

在 Queue 类的 queueRun 方法中，使用了 reader.readLine() 操作来读取客户端发送的消息。这个方法本身也是一个阻塞方法，当没有新的消息输入时，线程会被阻塞在这里等待消息的到来。这些阻塞操作都是基于 BlockingQueue 的特性实现的。put 方法会阻塞直到队列有空间可用，而 take 方法会阻塞直到队列中有消息可以取出。这种机制保证了线程在适当的时机等待并且在特定条件满足时被唤醒，从而实现了线程之间的同步。

- 伪代码如下：

```
public void queueRun(Socket client) throws IOException {
    // ...其他逻辑
   do {
      // 从输入流中读取消息
      mess_client = reader.readLine();
   } while (mess_client == null);
   // ...其他逻辑
  }
```

3.4 socket通信

(1) ServerSocket 和 Socket 对象

使用 ServerSocket 在指定的端口上监听生产者和消费者的连接请求，通过 serverSocket.accept() 接受来自生产者和消费者的连接请求，一旦有连接请求到达，就会创建一个新的 Socket 对象并获取对应的 Socket 实例，并作为参数传递给线程异步处理与客户端进行通信。

- 伪代码如下：

```
RealQueue() {
    // 初始化
    serverSocket = new ServerSocket(port); //设置服务器serversocket开始监听
    while (循环) {
        Serversocket.accept( ) // 等待连接
        Socket client = 接受客户端连接()
        // 执行连接的处理
         连接成功 {
            executorService.execute(() -> {
            线程池处理客户端连接(client)
            }}}
    // 关闭资源
```

}

(2) 客户端发起连接：

使用new Socket的方式，绑定服务器ip和监听的端口。

Socket socket = new Socket(ip,port);

3.5 消息的持久化

(1) 文件操作

① 读取文件readFile()方法：

当消费者获取数据时，服务器首先读取文件，文件为空则直接服务器直接队列获取数据，文件不为空则先把文件读到队列里，再从队列中获取数据。readFile() 方法通过 BufferedReader 读取文件中的内容，并将每行内容转换为 Message 对象放入队列管理器中。

➢ 伪代码如下：

```
readFile() {
    BufferReader reader =  新建File(fileName) //  创建文件读取reader
    //  读取文件内容
    while (存在下一行内容) {
        currentLine =  读取下一行内容
            mess = MessageManager.ToMsg(currentLine)//  转换字符串为消息对象
        Try {
            QM.put(mess.getTag(), mess)//  将消息放入队列管理器
        } catch (InterruptedException  异常) {
            输出异常信息
        }
    }
    关闭BufferReader
}
```

② 写入文件writeFile() 方法：

如果队列中还有数据，则写入文件，如果没有则不操作。writeFile() 方法遍历队列管理器中的消息队列，将每个 Message 对象转换为字符串形式，并使用 BufferedWriter 将其追加写入文件中。

➢ 伪代码如下：

```
writeFile() {
    对于队列管理器中的每个键值对  (entry : QM.queueMap.entrySet()) {
        String str
        BlockingQueue<Message> queueString = entry.getValue()
         对于  队列中的每条消息  (mess : queueString) { //  遍历队列中的每条消息
            //  将消息处理
            调用  inWrite(str)//  写入文件
        }
    }
}
```

(2) 主程序中的持久化调用

在 QueueBuffer 类的构造函数中使用了 Runtime.getRuntime().addShutdownHook 方法，在程序即将退出时，调用 writeFile() 方法，将内存中的消息队列持久化到文件中。

- 代码如下：

```
Runtime.getRuntime().addShutdownHook(new Thread(()->{
    fileMan.writeFile();
}));
```

(3) 消费者处理数据的结果记录

消费者对数据的处理后的结果（此处是判断了1000000个数是否是素数），记录在了文件中。

3.6 消息的传递

(1) 消息格式定义与转换

消息以字符串进行传递，类 MessageManager 中有 MsgTo 和 ToMsg 方法负责将构建消息和解析消息。生产者消息的格式按照[send : tag, 具体消息]定义，消费者订阅的格式按照[login:userID,tag[]]定义,推送的格式按照[get:tag[]]的格式定义。

(2) 流的传输

通过每一次socket连接的socket.getInputStream()和socket.getOutPutStream()获取输入流和输出流，并通过BufferedReader 读取消息。

3.7 订阅-推送模式

(1) 订阅过程：

消费者在连接到服务器时，可以发送包含订阅信息的消息。通过发送包含特定格式的数据，指定用户ID和感兴趣的标签，以此表示消费者希望订阅这些标签下的消息。

(2) 处理订阅请求：

服务端接收到消费者的订阅请求消息后，将订阅信息进行解析和处理。这些信息可以存储在服务器端的数据结构中，例如在 QueueManager 中以某种形式维护消费者的订阅信息，比如一个映射表，将用户ID与其订阅的标签列表关联起来。

(3) 实现消息推送：

① 生成和发送消息：

当有新消息到达时，服务端根据消息的标签确定需要将该消息推送给哪些订阅了相关标签的消费者。

② 推送消息给订阅者：

通过在 QueueManager 中查找特定标签的订阅者列表，服务端可以将新消息发送给这些订阅者。这个过程涉及将消息从服务器发送到客户端，这可以通过已建立的 Socket 连接并利用输出流实现。

4、实验结果及讨论

4.1 实验结果：

1 生产者-1 消费者，系统运行时间:27s

1 生产者-2 消费者，系统运行时间:24s

2 生产者-2 消费者，系统运行时间:32s

2 生产者-4 消费者，系统运行时间:28s

4.2 结果分析与讨论

(1) 实验结果增加生产者消费者数量运行时间不降反增，此结果可能与系统的设计仍有缺陷有关，也可能受多个因素的影响，例如：

资源竞争： 使用多个生产者和消费者时，会增加资源竞争的可能性。如果共享的资源（比如队列）需要频繁地进行锁定和释放，会增加系统运行的开销，导致整体运行时间增加。

同步与通信开销： 在多个生产者和消费者的情况下，需要更多的同步和通信。这可能会增加上下文切换、锁争用和通信开销，导致系统性能下降。

任务分配： 系统的性能可能受到任务分配不均匀的影响。如果生产者的产出速度高于消费者的处理速度，队列可能会在生产者生产新数据时变得很快，这会增加消费者的等待时间，导致整体运行时间增加。

硬件和系统限制： 系统的硬件能力和资源分配也可能对这些结果有影响。如果硬件资源（如 CPU、内存、网络带宽等）有限，可能会导致系统的性能下降，特别是在处理更多任务时。

5、本课程收获

本次分布并行计算的课程收获颇多，实验内容很丰富，课堂充实，可谓是收获颇多。以下是我在本次课程中主要的收获。

1、通过多线程计算 π，初步了解了程序并行执行的原理以及在 java 中的实现方法。

2、通过生命游戏的实验，进一步学习了在多线程中需要注意的点，比如数据分解和任务分解两种分解方法，以及这两种方法对应的应用场景；学到了线程与线程之间如何实现同步或异步，如何实现线程间在保持线程安全和数据一致性的情况下进行数据交换通信；

3、通过 socket 编程的学习，结合了计算机网络的知识，学到了局域网进程间不同主机如何通信；学到了 C-S 服务器客户端模型：服务器和客户端之间的协议是如何在代码上实现的；服务器与客户端之间的 session 传递等等。

4、通过生产者-消费者模型的学习，进一步了解到分布式的系统结构，强化了 socket 编程和多线程编程的能力，可以将理论知识运用在多种运用场景中，可以说基本有了简单分布式系统实现的能力。

5、通过 Web 部分的学习，了解了 Web 服务器的实现原理，掌握了如何在网页调用 webApi 接口。

6、通过本次生产者-消息中心-消费者模型，更是锻炼了我的编程能力，之前的课程学习都是在一个进程就可以完成，在这次实验实现了一个局域网内的简单系统。在实现的过程中，通过不断上网搜索资料，也尝试实践了很多之前不了解的知识，例如 java 程序如何连接 MySQL 数据库，类的序列化与传输，利用 Runtime 类实现操作系统的进程。