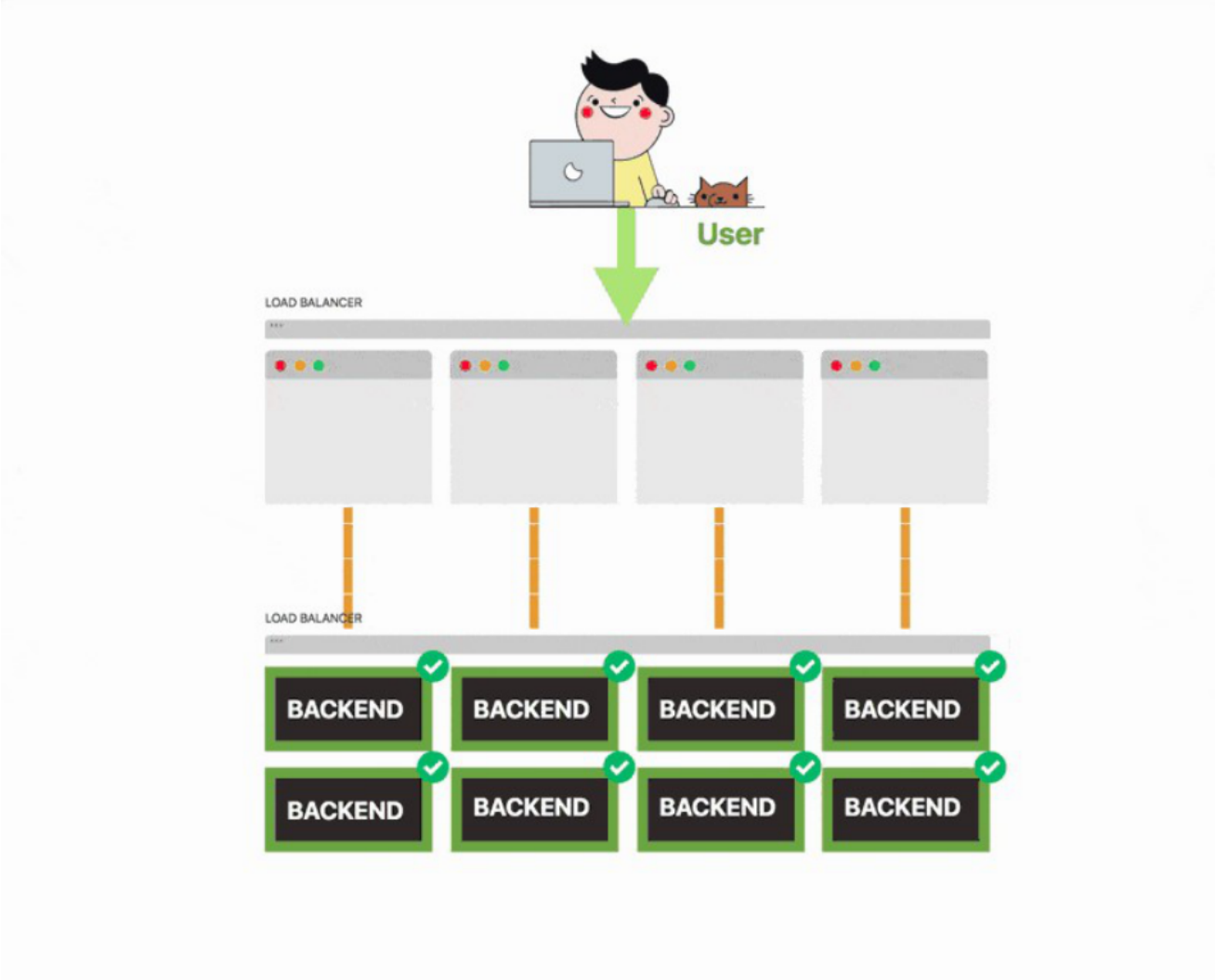
消息队列

考虑一个场景：**在不使用消息队列情况下，若将用户的请求数据直接写入数据库，在高并发的情况下数 据库压力将会剧增，处理速度，响应速度都将变慢。**



队列概念:

一种先进先出的数据结构。

消息队列概念：

是一种进程间通信或同一进程的不同线程间的通信方式，软件的贮列用来处理一系列的输入，通常是来自用户。消息队列提供了异步的通信协议，每一个贮列中的纪录包含详细说明的数据，包含发生的时间，输入设备的种类，以及特定的输入参数，也就是说：消息的发送者和接收者不需要同时与消息队列 互交。消息会保存在队列中，直到接收者取回它

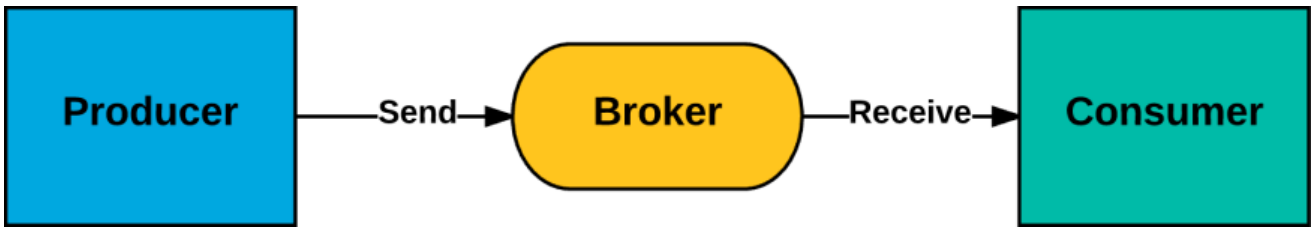
##

消息的生产者、消费者

●**Producer**：发送消息到消息队列。

●**Broker**：消息处理中心。负责消息存储、确认、重试等，一般其中会包含多个 queue；

●**Consumer**：从消息队列接收消息， 并做相应处理。



较常用的消息队列：ActiveMQ、RabbitMQ、Kafka、RocketMQ等。

在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库 有更好的伸缩性），因此响应速度得到大幅改善。

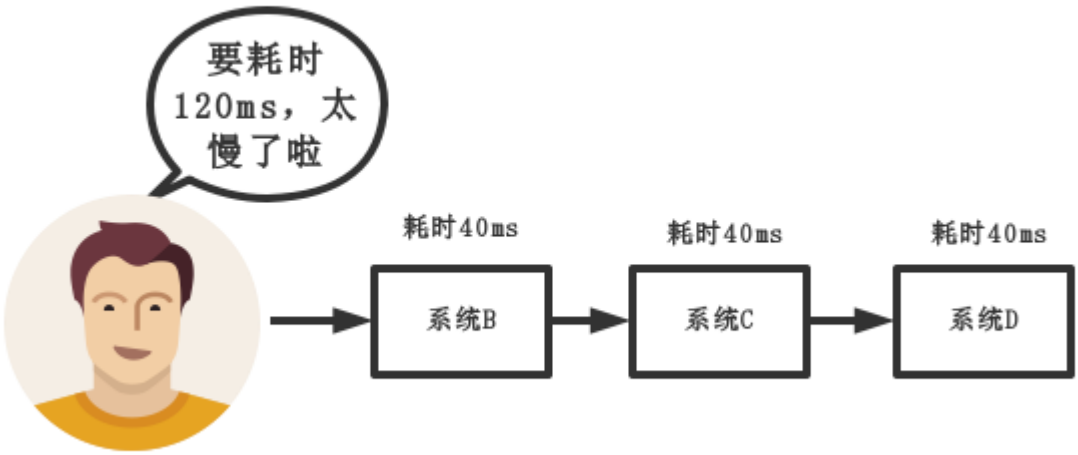
常见应用

异步处理

将一个业务操作分成多个阶段， 每个阶段间通过共享数据的方式异步执行实现：

- 单一服务器内部通过多线程共享消息队列实现
- 分布式系统中通过分布式消息队列实现。 即将消息队列服务部署到独立的服务器上，应用程序通过远程访问接口使用分布式消息队列特点：
- 提高系统可用性。若消费者服务器故障，在故障前用户所传输的数据仍在消息队列服务器中。
- 加快响应速度：处理前端的请求后，写入消息队列即可响应，无需同步处理好了数据，再做出响应
- 减缓并发高峰压力：实际处理数据的服务不会直接受到高峰时段的业务压力，因为处理业务 的服务可以依此读取消息队列中的大量数据

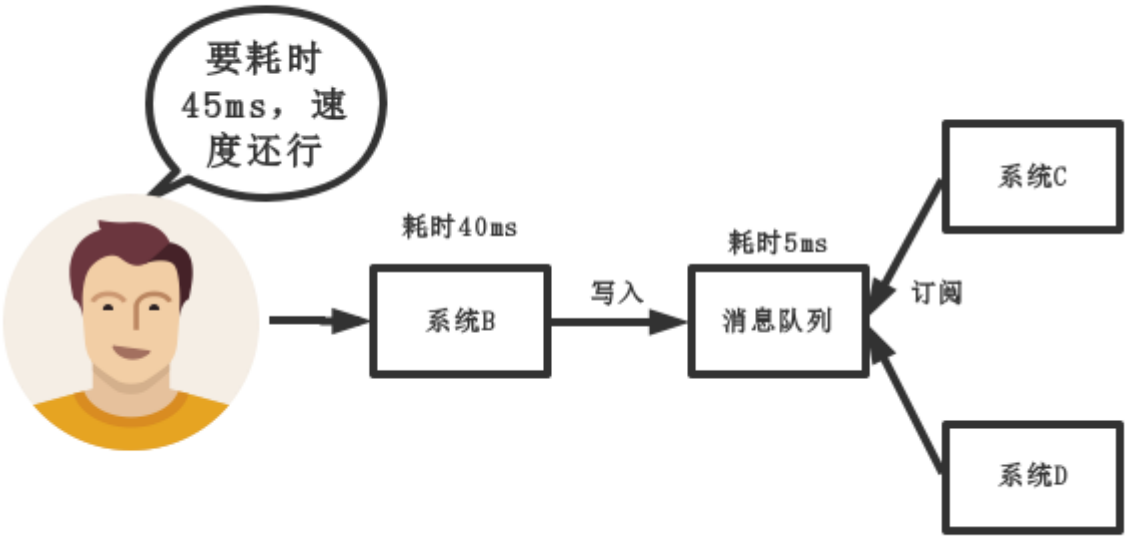
传统模式:



传统模式的缺点：

- 一些非必要的业务逻辑以同步的方式运行，太耗费时间。

中间件模式:

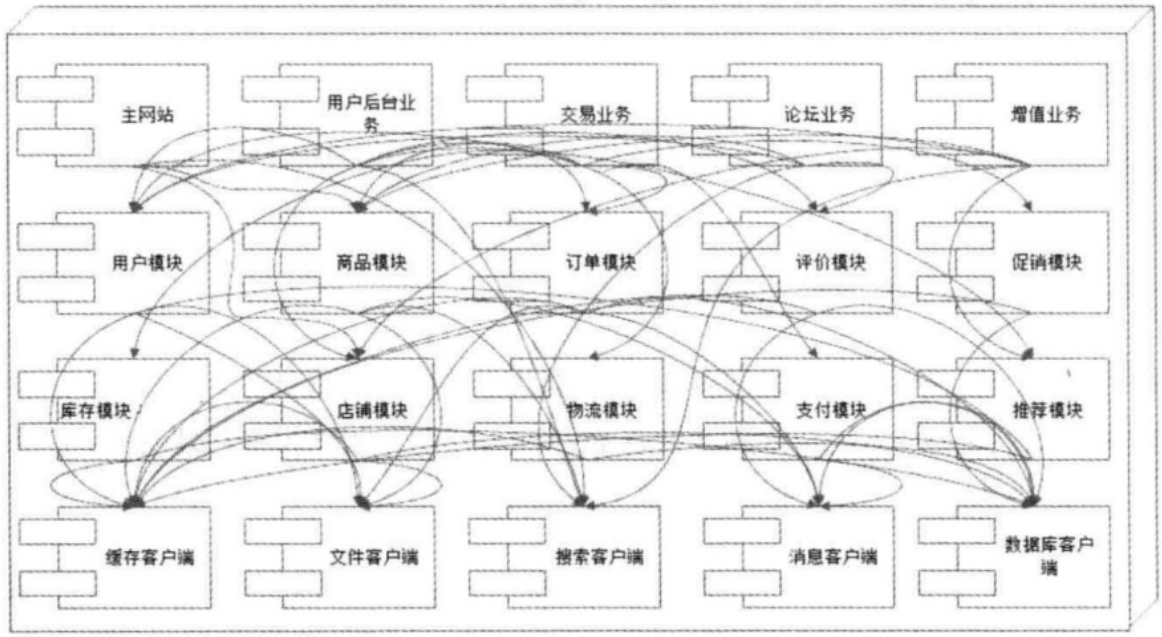


中间件模式的的优点：

- 将消息写入消息队列，非必要的业务逻辑以异步的方式运行，加快响应速度

服务解耦

●考虑一个电商系统

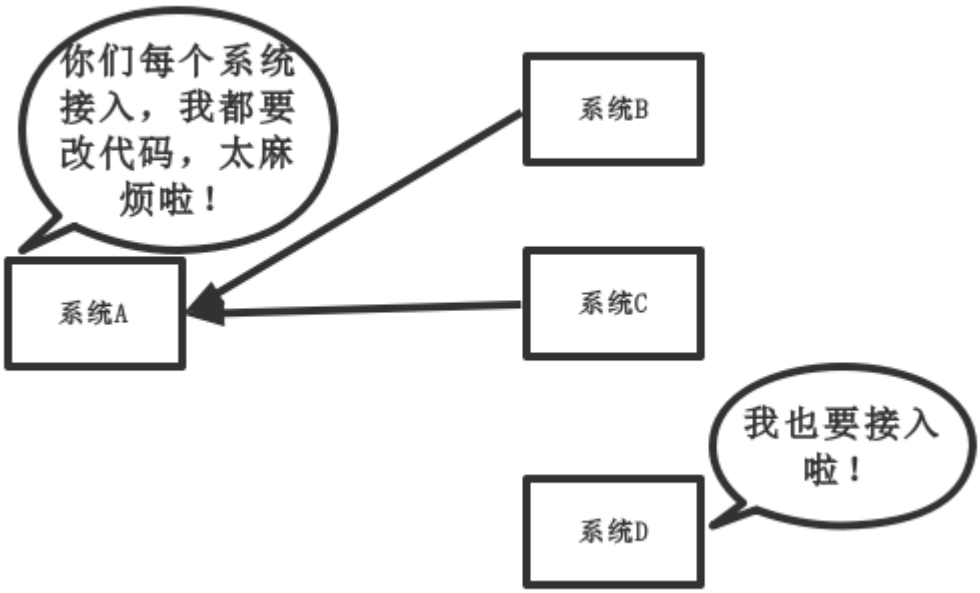


该系统可能面临的困难

- 编译部署困难
- 代码版本管理困难
- 数据库连接耗尽
- 业务需求变更困难

需要解耦

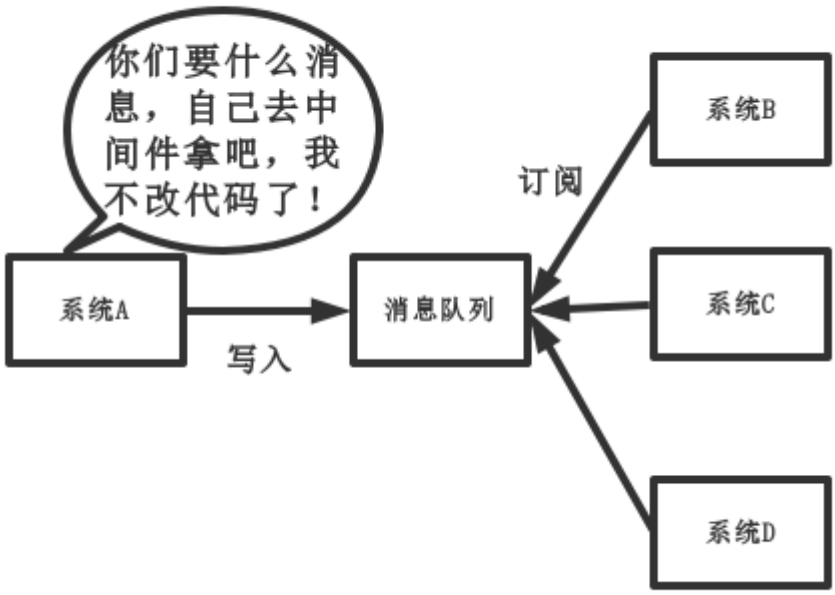
传统模式:



传统模式的缺点：

- 系统间耦合性太强，如上图所示，系统A在代码中直接调用系统B和系统C的代码，如果将来D系统接入，系统A还需要修改代码，过于麻烦

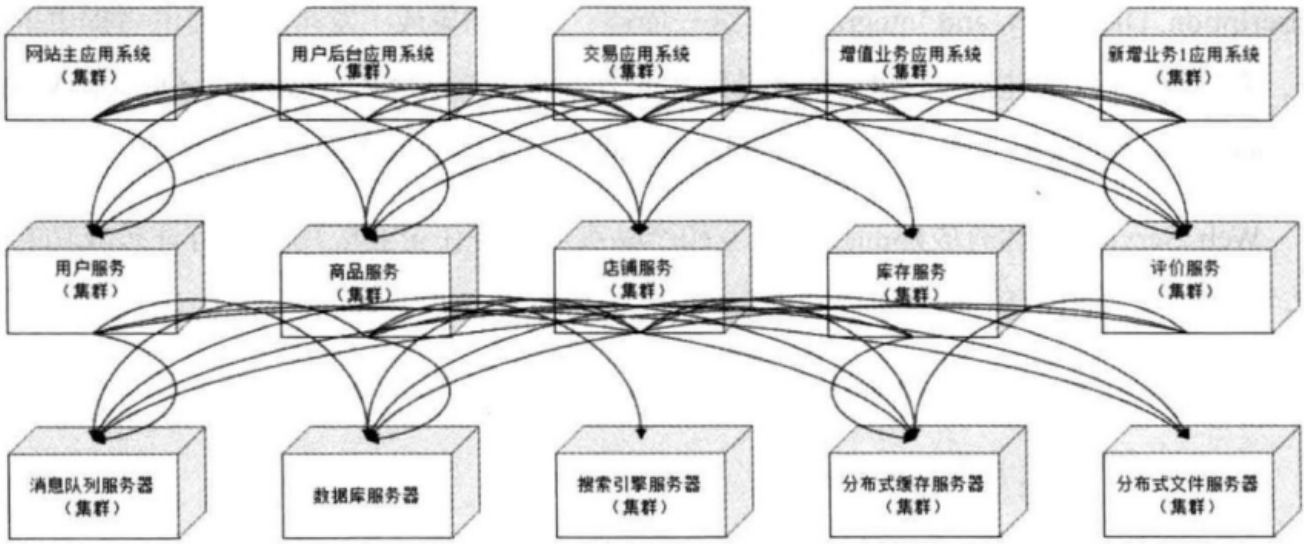
中间件模式:



中间件模式的的优点：

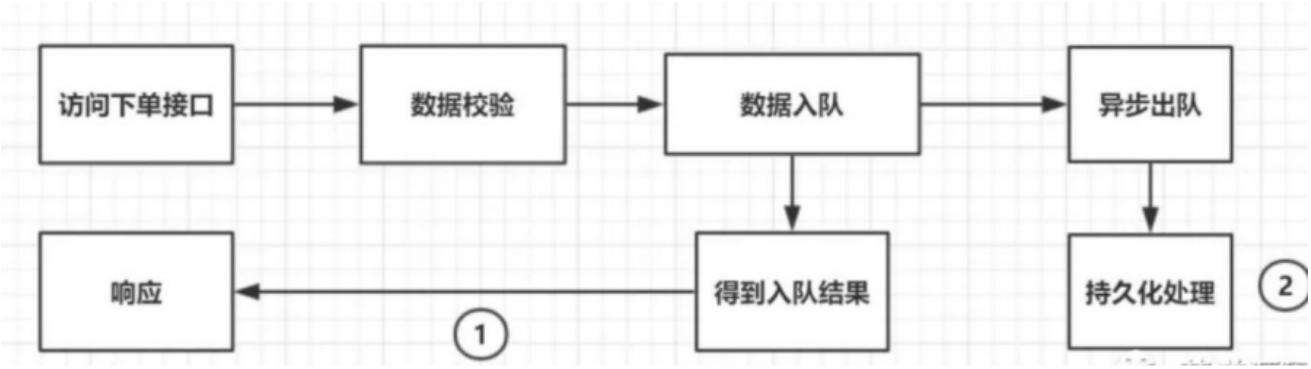
- 将消息写入消息队列，需要消息的系统自己从消息队列中订阅，从而系统A不需要做任何修改。

拆分搭建为分布式服务



此时不同子系统通过分布式消息队列处理同一个消息：

例如：下单



在数据入队进入消息队列集群后，其它子系统，例如库存服务，物流服务，订单服务从消息 队列集群中取得该数据，再做相应的处理。

流量消峰

●购物网站开展秒杀活动

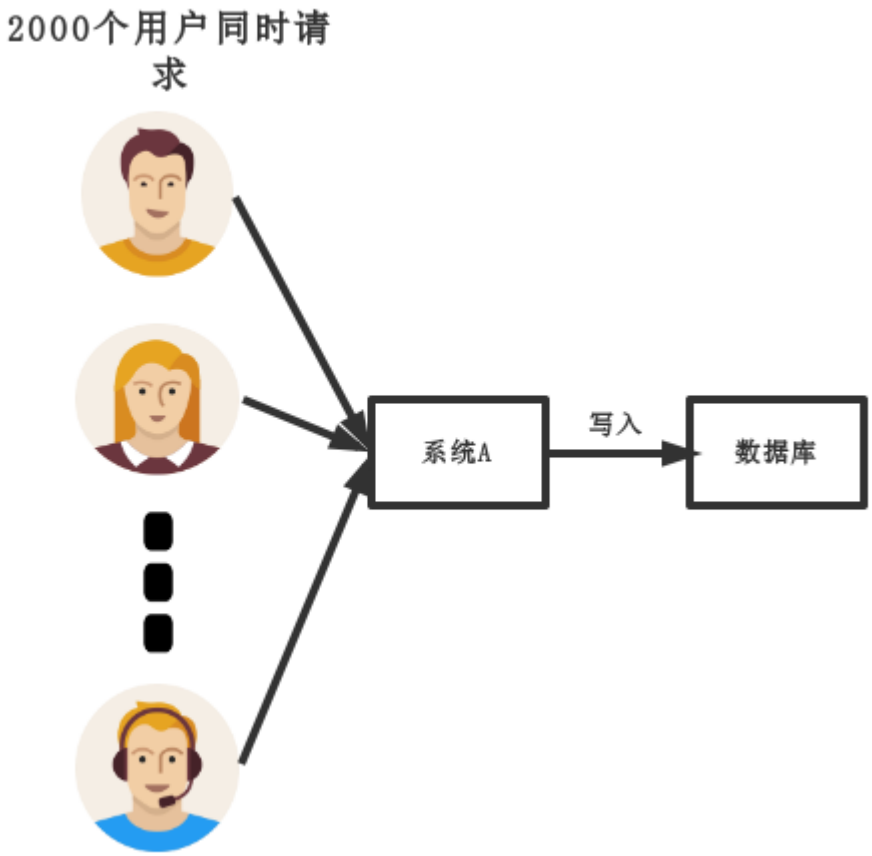
该系统可能面临的困难●○：**一般由于瞬时访问量过大，服务器接收过大，会导致流量暴增，相关系统无法处理请求甚至崩溃。而加入消息队列后，系统可以从消息队列中取数据，相当于消息队列做了一次缓冲。**

该方法有如下优点：

请求先入消息队列，而不是由业务处理系统直接处理，做了一次缓冲,极大地减少了业务处理系统的压力；

队列长度可以做限制，事实上，秒杀时，后入队列的用户无法秒杀到商品，这些请求可以直接被抛弃，返回活动已结束或商品已售完信息；

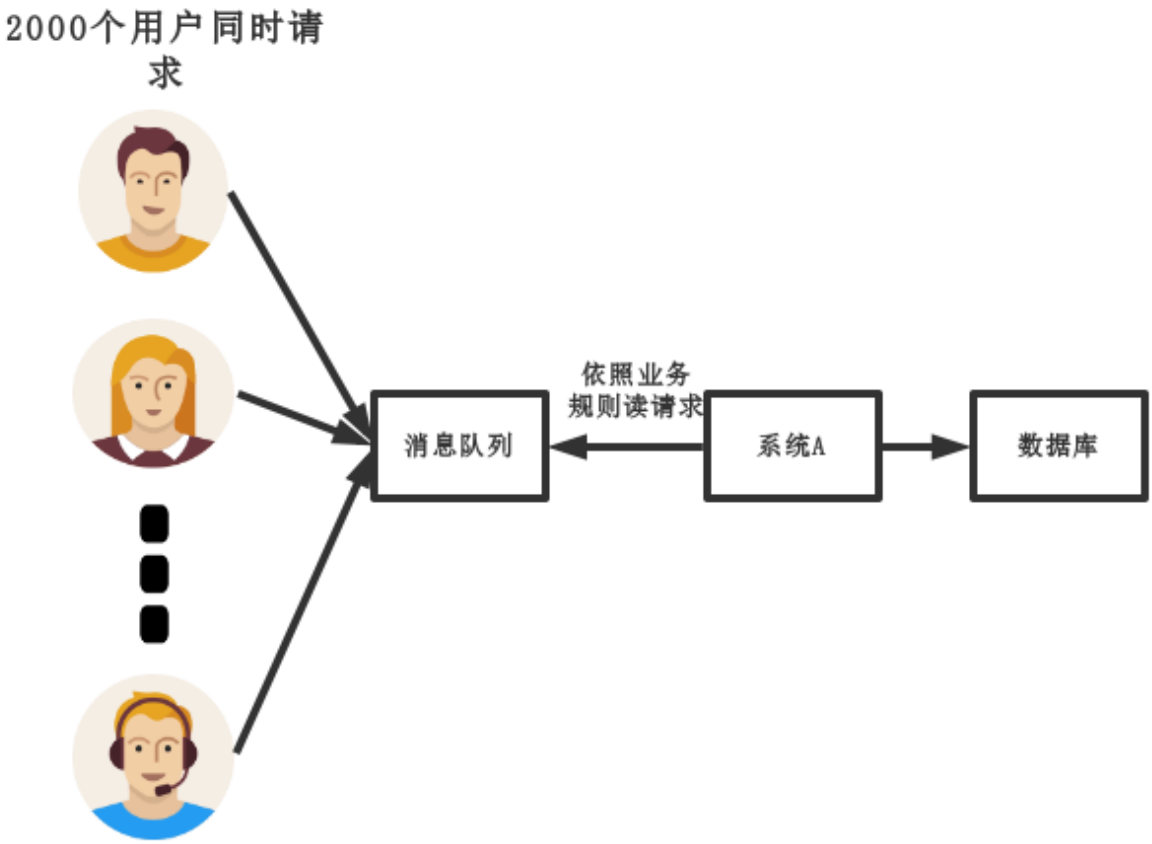
传统模式



传统模式的缺点：

- 并发量大的时候，所有的请求直接怼到数据库，造成数据库连接异常

中间件模式:



中间件模式的的优点：

- 系统A慢慢的按照数据库能处理的并发量，从消息队列中慢慢拉取消息。在生产中，这个短暂的高峰期积压是允许的

其他应用

消息通讯

处理日志

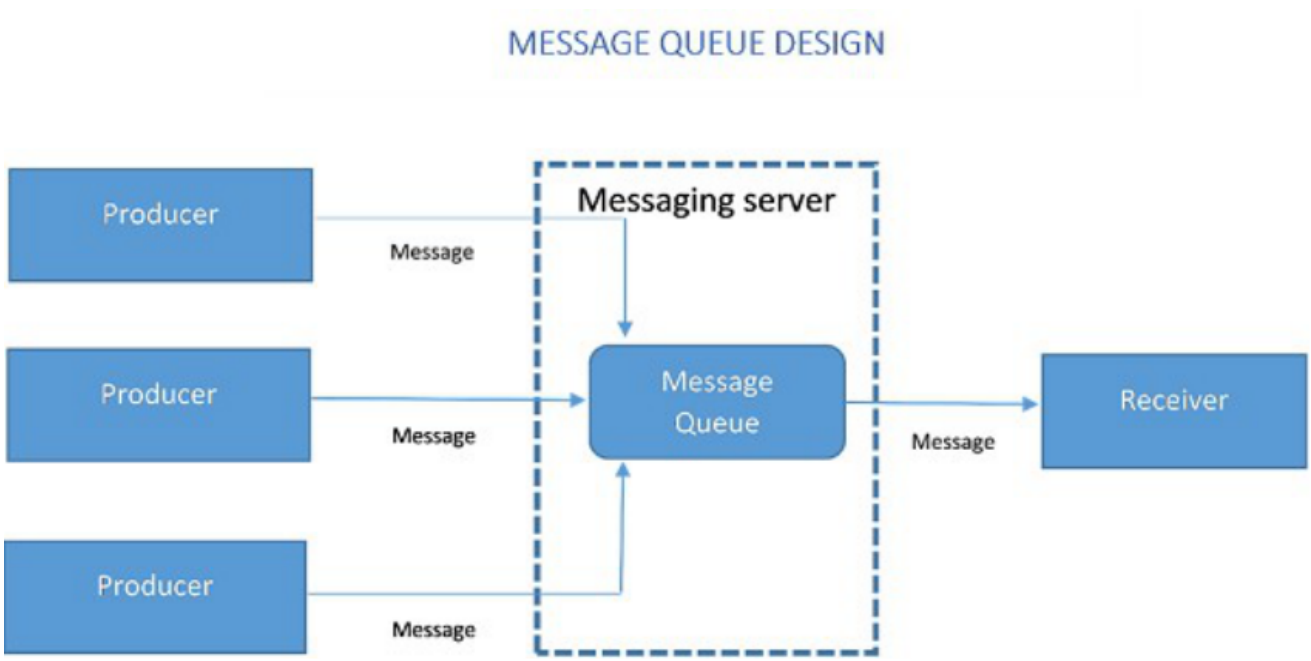
引入mq可能出现的问题：

- 系统可用性降低:本来其他系统只要运行正常，系统就是正常的。现在加个消息队列进去，消息队列挂了，你的系统就挂了。因此，系统可用性降低
- 系统复杂性增加:要多考虑很多方面的问题，比如一致性问题、如何保证消息不被重复消费，如何保证保证消息可靠传输。因此，需要考虑的东西更多，系统复杂性增大。

常用消息队列模型

●**点对点消息队列模型** （一个消息只有一个消费者）

消息生产者向一个特定的队列发送消息，消息消费者从该队列中接收消息； 消息的生产者和消费者可以不同时处于运行状态。

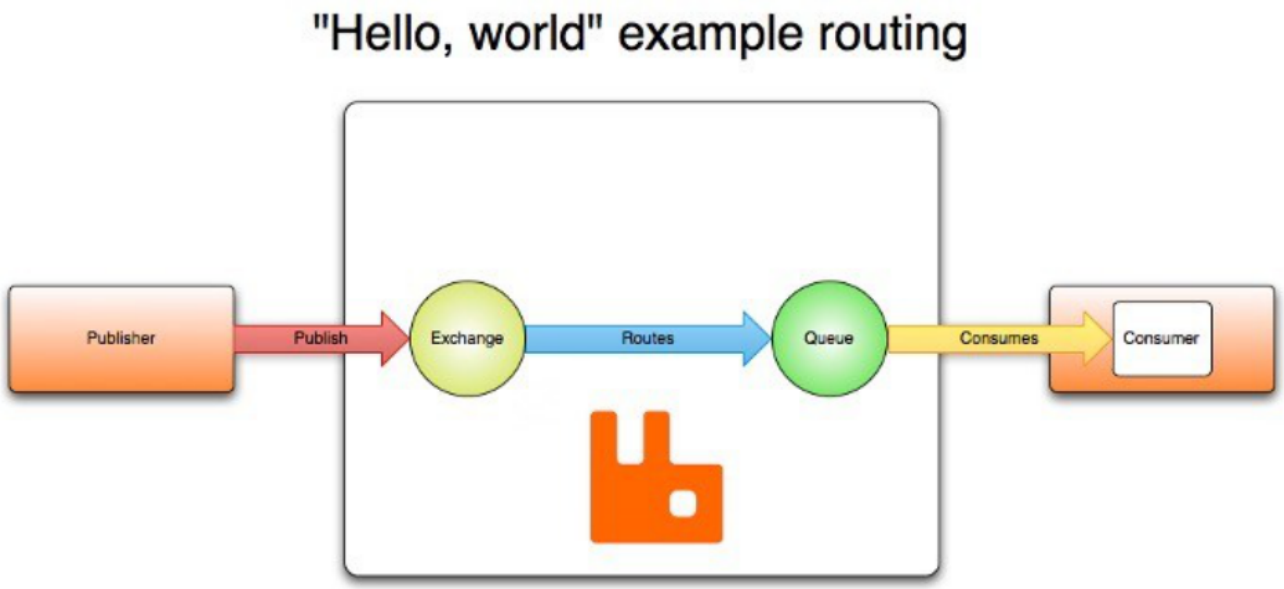


●发布订阅消息模型 （消息发送者发布消息，一个或多个消息接受者订阅消息）

发布订阅消息模型中，支持向一个特定的主题Topic发布消息，0个或多个订阅者接收来自这 个消息主题的消息。在这种模型下，发布者和订阅者彼此不知道对方。实际操作过程中，

必须保证 先订阅，再发送消息，而后接收订阅的消息

RabbitMQ简单实践



Rabbit中除Producer和Consumer的其他概念：

●**Exchange :** 从生产者接收消息并将其传递到队列。

1. 安装RabbitMQ
2. 安装相应的依赖包

`go get github.com/streadway/amqp`

3 .建立消费者

```
import (
   "encoding/json"
   "github.com/streadway/amqp"
   "log"
   "os"
   "rabbitmqdemoProject/model"
)

func handleError(err error, msg string) {
   if err != nil {
      log.Fatalf("%s: %s", msg, err)
   }
}
func OpenConsumer() {
   conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
   handleError(err, "Can't connect to MQ")
   defer conn.Close()
   amqpChannel, err := conn.Channel()
   handleError(err, "Can't create a amqpChannel")
   defer amqpChannel.Close()
   queue, err := amqpChannel.QueueDeclare("goodList", true, false, false,
      false, nil)
   handleError(err, "Could not declare `add` queue")
   err = amqpChannel.Qos(1, 0, false)
   handleError(err, "Could not configure QoS")
   messageChannel, err := amqpChannel.Consume(
      queue.Name,
      "",
      false,
      false,
      false,
      false,
      nil,
   )
// 4. 建立发布者
   handleError(err, "Could not register consumer")
   stopChan := make(chan bool)
   go func() {
      log.Printf("Consumer ready, PID: %d", os.Getpid())
      for d := range messageChannel {
         log.Printf("Received a message: %s", string(d.Body))
         good := &model.Order{}
         err := json.Unmarshal(d.Body, good)
         if err != nil {
            log.Printf("Error decoding JSON: %s", err)
```

```
            }
            log.Printf("Good: %s", string(d.Body))
            if err := d.Ack(false); err != nil {
               log.Printf("Error acknowledging message : %s", err)
            } else {
               log.Printf("Acknowledged message")
            }
         }
      }()
      //终止当前进程
      <-stopChan
   }
```

4. 建立发布者

```
import (
   "encoding/json"
   _ "github.com/jinzhu/gorm/dialects/mysql"
   "github.com/streadway/amqp"
   "log"
   "math/rand"
   "rabbitmqdemoProject/model"
   "time"
)
func failError(err error, msg string) {
   if err != nil {

      log.Fatalf("%s: %s", msg, err)
   }
}

func Order(userid string,shopid string,goodid string,number string){
   conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    failError(err, "Can't connect to MQ")
   defer conn.Close()
   amqpChannel, err := conn.Channel()
   failError(err, "Can't create a Channel")
   defer amqpChannel.Close()
   queue, err := amqpChannel.QueueDeclare("goodList", true, false, false,
      false, nil)
   failError(err, "Could not declare queue")
   rand.Seed(time.Now().UnixNano())
   good := model.Order{Userid:userid , Shopid:shopid,
      Number:number, Goodid:goodid}
   body, err := json.Marshal(good)
   if err != nil {
      failError(err, "Error encoding JSON")
   }
   err = amqpChannel.Publish("", queue.Name, false, false, amqp.Publishing{
      DeliveryMode: amqp.Persistent,
      ContentType: "text/plain",
      Body: body,
   })
```

```
    if err != nil {
       log.Fatalf("Error publishing message: %s", err)
    }
    log.Printf("AddGood: %s", string(body))
}
```

作业

电影票秒杀服务

有两张表

电影票表

- 电影id
- 电影名
- 观影时间
- 观影地点
- 电影票数量

订单表

- 订单id
- 订单发生时间
- 订单人id
- 电影id

要求

1. 写一个抢票的接口，要求将请求的信息写入消息队列并响应
2. 订单服务，读消息队列将信息写入订单表
3. 电影票库存服务，读消息队列修改电影票数量
4. 高并发测试

邮箱：**wangmengfei@redrock.team**