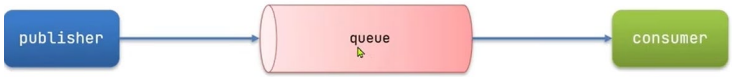
1、同步调用问题
【优点】时效性较强，可以立即得到结果；
【缺点】耦合度高，性能和吞吐能力下降，有级联失败问题。
2、异步调用问题
【优点】耦合度低，吞吐量提升，故障隔离，流量削峰；
【缺点】依赖于 Broker 的可靠性、安全性、吞吐能力。
3、SpringAMQP 是什么？
SpringAMQP 是基于 AMQP 协议定义的一套规范，包含两个部分，其中 spring-amqo 是基础抽象，spring-rabbit 是底层的默认实现；
1）提供**监听器容器**（用于**异步处理**入站的消息）
2）提供 **RabbitTemplate**（用于**发送**和**接受**消息）
3）提供 **RabbitAdmin**（用于**自动声明**队列，交换机和绑定）
4、常见的消息模型
1）简单队列模型
消息发布者将消息发送到消息队列；
消息队列负责接收并缓存消息；
消费者订阅队列，处理队列中的消息。



1>发布者和消费者都需要，故在父工程中**引入 AMQP 依赖**

```
<!--AMQP依赖，包含RabbitMQ-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2>在发布者和消费者中的.yml 文件中**配置 mq 地址等信息**

```
spring:
  rabbitmq:
    host: 192.168.150.101 # 主机名
    port: 5672 # 端口
    virtual-host: / # 虚拟主机
    username: itcast # 用户名
    password: 123321 # 密码
```

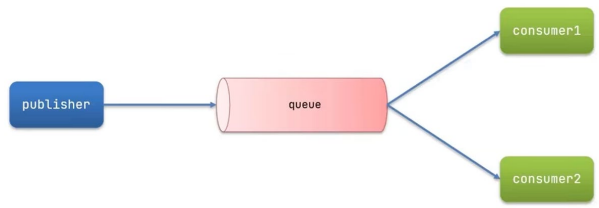
3>在**发布者服务**中，**注入 RabbitTemplate 的实例** rabbitTemplate，使用 **rabbitTemplate.convertAndSend("队列名"，"信息")发送消息到队列。**

```
@Autowired
private RabbitTemplate rabbitTemplate;
@Test
public void testSimpleQueue() {
    String queueName = "simple.queue";
    String message = "hello, spring amqp!";
    rabbitTemplate.convertAndSend(queueName, message);
}
```

4>在**消费者服务**中，用于监听的方法上加**@RabbitListener 注解（queues="队列名"）**，**方法参数**就是**接收的消息**

```
@RabbitListener(queues = "simple.queue")
public void listenSimpleQueueMessage(String msg) throws InterruptedException {
    System.out.println("spring 消费者接收到消息 : [" + msg + "] ");
}
```

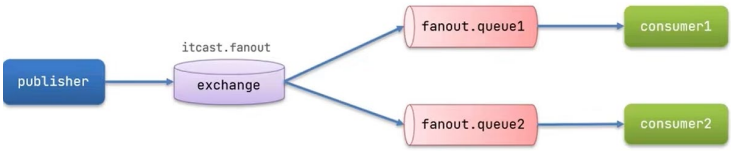
2）工作队列模型（同一条信息只会被一个消费者处理）
一个队列绑定多个消费者，可以**提高消息处理的速度**，**避免消息堆积**。
消费预取限制：.yml 文件中设置 listener.simple.**prefetch**，控制预取消息的上限，默认是无限（即先全部拿过来，后面再慢慢处理）




```
spring:
  rabbitmq:
    host: 192.168.150.101 # 主机名
    port: 5672 # 端口
    virtual-host: / # 虚拟主机
    username: itcast # 用户名
    password: 123321 # 密码
    listener:
      simple:
        prefetch: 1 # 每次只能获取一条消息，处理完成才能获取下一个消息
```

【发布订阅模型（加入了交换机，允许同一消息发送给多个消费者）】
交换机的作用：
1>接受发布者发送的消息；2>将消息按照规则路由到与之绑定的队列中；3>不能缓存消息，路由失败，消息丢失。

3）Fanout Exchange：广播

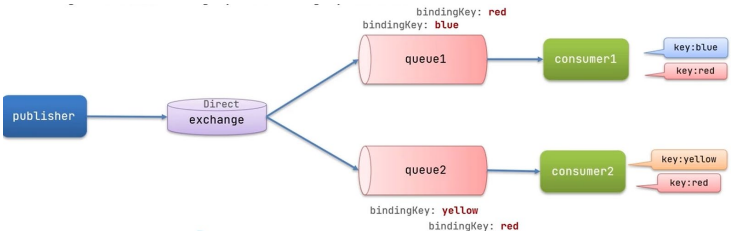


方式 1（不推荐）：创建一个配置类，声明 FanoutExchange、Queue 和绑定关系绑定关系对象 Binding：

```
@Configuration
public class FanoutConfig {
    // 声明FanoutExchange交换机
    @Bean
    public FanoutExchange fanoutExchange(){
        return new FanoutExchange("itcast.fanout");
    }
    // 声明第1个队列
    @Bean
    public Queue fanoutQueue1(){
        return new Queue("fanout.queue1");
    }
    //绑 定队列1和交换机
    @Bean
    public Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange){
        return BindingBuilder.bind(fanoutQueue1).to(fanoutExchange);
    }
    //  ... 略， 以相同方式声明第2个队列，并完成绑定
}
```

```
// 发送消息
rabbitTemplate.convertAndSend(exchangeName, routingKey: "", message);
```

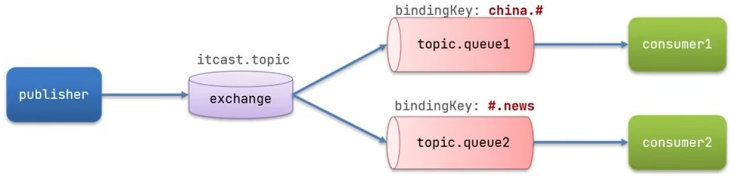
4）Direct Exchange：路由
每个 Queue 都与 Exchange 设置一个 BindingKey，发布者发送消息时，指定消息的 RoutingKey，Exchange 将消息路由到 BindingKey 与消息 RoutingKey 一致的队列。



方式 2：利用**@RabbitListener 注解**声明 Exchange、Queue、RoutingKey
@RabbitListener(**bindings** = **@QueueBinding**(
value = **@Queue**(name = "queue"),
exchange=**@Exchange**(name="exchange",type=ExchangeTypes.DIRECT),
key = { "red" , "blue" }
))放在消费者服务中的监听方法上

```
// 发送消息
rabbitTemplate.convertAndSend(exchangeName, routingKey: "blue", message);
```

5）Topic Exchange：主题
RoutingKey 必须是**多个单词的列表**，并且以**"." 分割**；
Queue 和 Exchange 指定 BindingKey 时可以使**用通配符**：
#：代指 **0 或多**个单词；*：代指**一个**单词。




```
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(name = "topic.queue1"),
        exchange = @Exchange(name = "itcast.topic", type = ExchangeTypes.TOPIC),
        key = "china.#"
))
```

```
// 发送消息
rabbitTemplate.convertAndSend(exchangeName, routingKey: "china.news", message);
```

5、消息转换器
在 SpringAMQP 的发送方法中，接收消息的类型是 Object，即可发送任意对象类型的消息，SpringAMQP 会基于 JDK 的 ObjectOutputStream 自动序列化为字节后发送。
【推荐使用 json 方式序列化】
1）在发布者服务中引入 jackson 依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.9.10</version>
</dependency>
```

2）发布者服务中声明 MessageConverter，Jackson2jsonMessageConverter()

```
@Bean
public MessageConverter jsonMessageConverter(){
    return new Jackson2JsonMessageConverter();
}
```

6、消息丢失

1）生产者发送消息时连接 MQ 失败；
2）消息到达 MQ 后处理消息的进程发生异常。
3）到达 MQ 后未找到交换机；
4）到达 MQ 的交换机后，未找到合适的队列；
5）消息到达 MQ 队列后，未消费突然宕机或不能及时保存。
6）消息接收后尚未处理突然宕机；
7）消息接收后处理过程中抛出异常。

7、发送者的可靠性

1）生产者重试机制

当 **RabbitTemplate 与 MQ 连接超时**后，多次重试，以解决生产者发送消息时，出现了网络故障，导致与 MQ 的连接中断。
修改生产者服务中.yml 文件：

```yaml
spring:
  rabbitmq:
    connection-timeout: 1s # 设置MQ的连接超时时间
    template:
      retry:
        enabled: true # 开启超时重试机制
        initial-interval: 1000ms # 失败后的初始等待时间
        multiplier: 1 # 失败后下次的等待时长倍数，下次等待时长 = initial-interval * multiplier
        max-attempts: 3 # 最大重试次数
```

【注意】多次重试等待的过程中，当前线程是被阻塞的，不建议使用。

2）生产者确认机制（Publisher Confirm（返回 ack 和 nack）和 Publisher Return（返回 return））
当生产者发送消息给 MQ 后，MQ 会根据消息处理的情况**返回不同的回执**。
1>当消息投递到 MQ，但是**路由失败时**，通过 Publisher **Return 返回异常信息**，同时**返回 ack 的确认信息**，代表投递成功；
2>**临时消息**投递到了 MQ，并且**入队成功，返回 ACK**，告知投递成功；
3>**持久消息**投递到了 MQ，并且**入队完成持久化，返回 ACK** ，告知投递成功；
4>其它情况都会返回 NACK，告知投递失败。
【注意】开启生产者确认比较**消耗 MQ 性能**，一般**不建议开启**，路由失败，交换机名称错误等都是**编程过程错误**。

8、MQ 的可靠性

在默认情况下，RabbitMQ 会将接收到的信息**保存在内存中**以降低消息收发的延迟。导致：
>MQ 一旦**宕机**，内存中消息会丢失；
>**内存有限**，处理慢会导致**消息堆积 MQ 阻塞**；

1）数据持久化

1>交换机持久化；2>队列持久化；3>消息持久化（这三在代码中声明的时候默认为持久的）

2）LazyQueue（惰性队列）

1>接收到消息后**直接存入磁盘**而非内存；
2>消费者要消费消息时才会**从磁盘中读取并加载到内存**；（也就是懒加载）
3>支持数百万条的消息存储。
代码配置 Lazy 模式

```
@RabbitListener(bindings = @QueueBinding(
value = @Queue(name = "lazy.queue",
                durable = "true",
                arguments =@Argument(
                name = "x-queue-mode",
                value = "lazy")),
exchange=@Exchange(name="exchange",type=
ExchangeTypes.DIRECT),
key = {"red","blue"}))
```

9、消费者的可靠性

1）消费者确认机制

当消费者处理消息结束后，应该**向 RabbitMQ 发送一个回执**，告知 RabbitMQ 自己消息处理状态。三种回执：
1>**ack：成功**处理消息，RabbitMQ 从队列中**删除该消息**；
2>**nack**：消息处理**失败**，RabbitMQ 需要**再次投递消息**；
3>**reject**：消息处理**失败并拒绝该消息**，RabbitMQ 从队列中**删除该消息**。（例如消息本身格式就有问题，消费者处理不了）

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: none # 不做处理
```

SpringAMQP 实现了消息确认，通过配置文件：
1>**none：不处理**。即消息投递给消费者后**立刻 ack，并立刻从 MQ 删除**。不安全，不建议用；
2>**manual：手动模式**。需要在业务代码中**调用 api，发送 ack 或 reject**，存在业务入侵。
3>**auto：自动模式**。SpringAMQP 利用 **AOP 对消息处理逻辑做环绕增强**，当业务**正常执行时**则自动**返回 ack**；当业务**出现异常时**，根据**异常判断返回不同结果**：
>**业务异常**，会自动**返回 nack**；
>**消息处理或校验异常**，自动**返回 reject**；

2）失败重试机制

当**消费者出现异常**后，消息会**不断重新入队到队列**，再**重新发送**给消费者。无线循环会导致 mq 的消息处理飙升。
Spring 提供了**消费者失败重试机制**：在消费者出现异常时，**在消费者本地重试**，而不是无限制的重新入队到 mq 队列；并且**重试达到最大次数**后，Spring 会**返回 reject**，**消息会被丢弃**。

3）失败处理策略

失败重试达到最大重试次数后，丢弃是不适合的，需要采用合适的处理策略：
1>RejectAndDontRequeueRecoverer（默认）
重试耗尽后，直接 **reject**，**丢弃消息**；
2>ImmediateRequeueMessageRecoverer
重试耗尽后，返回 **nack**，消息**重新入队**；
3>RepublishMessageRecoverer
重试耗尽后，将失败消息**投递到指定交换机**；后续再人工处理。

10、业务幂等性

指同一个业务，执行一次或多次对业务状态的影响是一致的。
1）对于非幂等业务，我们要避免被重复执行：
1>页面**卡顿频繁刷新**会导致表单重复提交；
2>**服务调用的重试**；
3>MQ 消息的**重复投递**。

2）唯一消息 ID（要改数据库，不推荐）
1>每一条**消息**都**生成一个唯一的 id**，**与消息一起投递**给消费者；
2>消费者接收到消息后处理自己的业务，业务处理成功后**将消息 ID 保存到数据库**；
3>如果**下次又收到相同消息**，去数据库**查询判断是否存在该消息 ID**，存在则为重复消息放弃处理。
SpringAMQP MessageConverter 有 MessageID 的功能，开启即可。
jjmc.**setCreateMessageIds(true);**

```
@Bean
public MessageConverter messageConverter(){
    // 1.定义消息转换器
    Jackson2JsonMessageConverter jjmc = new Jackson2JsonMessageConverter();
    // 2.配置自动创建消息id，用于识别不同消息，也可以在业务中基于ID判断是否是重复消息
    jjmc.setCreateMessageIds(true);
    return jjmc;
}
```

3）业务判断

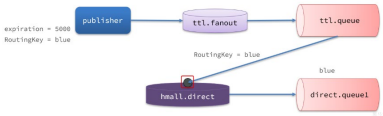
例如将订单状态从未支付变为已支付：我们只需要先查询订单状态是否为未支付，是则改变状态，反之不用再次操作。也可以设置定时任务去查询订单支付状态来兜底。

11、延迟消息

延迟消息：生产者发送消息时指定一个时间，消费者不会立刻收到消息，而是**在指定时间之后才收到消息**。
延时任务：设置在一定时间后才执行的任务。如果直接设置定时，非常消耗 cpu 资源，并且长时间的堆积消息会导致 MQ 压力很大。

12、死信交换机

死信（dead letter）：
1>消费者**返回 reject 或 nack 回执**，并且该消息**不再重新入队列**；
2>消息是一个**过期消息，超时无人消费**；
3>要投递的**队列消息满了，无法投递**。
这些死信所属队列会将死信投递到**由 dead-letter-exchange 属性指定的交换机**中，该交换机就称为**死信交换机**。
死信交换机作用：
1>收集那些因**处理失败而被拒绝的**消息
2>收集那些因**队列满了而被拒绝的**消息
3>收集因 **TTL（有效期）到期的**消息
13、死信交换机实现延时消息
有**一组绑定的交换机和队列**，但是**队列没有消费者监听**，当该队列中的**消息超时**后成为**死信**，会**投递给死信交换机**，然后死信交换机**沿用之前的 RoutingKey**，将死信路由到**消费者绑定的队列**中，供消费者使用。
【缺点】太麻烦，太多交换机和队列。



13、延迟消息插件（DelayExchange 插件）

原理：涉及了一种支持延迟消息功能的交换机，当消息投递到交换机后可以暂存一定时间，到期后再投递到队列。
1）消费者监听的消息队列参数 delay=“true”

```
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(name = "delay.queue", durable = "true"),
        exchange = @Exchange(name = "delay.direct", delayed = "true"),
        key = "delay"
))
public void listenDelayMessage(String msg){
    log.info("接收到delay.queue的延迟消息：{}", msg);
}
```

2）发消息用 MessagePostProcessor **消息后置处理器**，msg.getMessageProperties().**setDlay()** 设置**延迟消息属性**

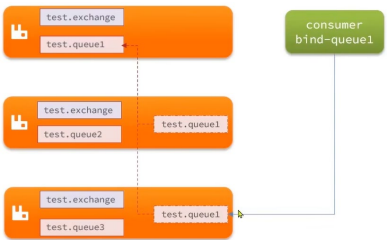
```
// 1.创建消息
String message = "hello, delayed message";
// 2.发送消息，利用消息后置处理器添加消息头
rabbitTemplate.convertAndSend("delay.direct", "delay", message, new MessagePostProcessor() {
    @Override
    public Message postProcessMessage(Message message) throws AmqpException {
        // 添加延迟消息属性
        message.getMessageProperties().setDelay(5000);
        return message;
    }
});
```

14、RabbitMQ 消息怎么传输？

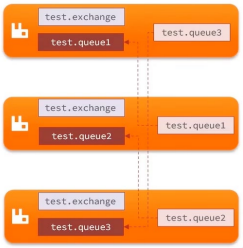
RabbitMQ 使用**信道**的方式来**传输数据**。**信道**（Channel）是生产者、消费者与 RabbitMQ **通信的渠道**，信道是**建立在 TCP 链接上的虚拟链接**，且每条 TCP 链接上的**信道数量没有限制**。就是说 RabbitMQ 在**一条 TCP 链接**上建立**成百上千个信道**来达到**多个线程处理**，**每个信道**在 RabbitMQ **都有唯一的 ID**，保证了**信道私有性**，每个信道**对应一个线程使用**。

15、如何保证 RabbitMQ 高可用的？

1）普通集群：是一种**分布式集群**，将**队列**分散到**集群的各个节点**，从而提高整个集群的**并发能力**。
特征：
1>会在集群的**各个节点共享部分数据**，包括：交换机、队列**元信息**，**不包含队列信息**；
2>当访问集群**某节点**时，如果队列不在该节点，会**从数据所在节点**传到**当前结点并返回**；
3>队列**所在节点宕机**，队列中的**消息会丢失**。



2）镜像集群：是一种**主从集群**，普通集群的基础上，添加**主从备份功能**，提高集群的数据**可用性**。（主从同步并**不是强一致**，数据丢失）
特征：
1>交换机、队列、队列中的消息会在各个 MQ 的镜像节点之间同步备份；
2>**创建队列的节点**被称为该队列的**主节点**，**备份到的其他节点**叫做该队列的**镜像节点**；
3>一个队列的主节点可能是另一个队列的镜像节点；
4>**所有操作都是主节点完成**，然后**同步给**镜像节点；
5>**主宕机后，镜像节点**会**替代**称新的主。



3）仲裁队列：与镜像队列一样都是主从模式；底层采用 **Raft 协议**确保**主从的数据一致性**。