rabbitmq、activemq、kafka之间的比较

RabbitMq：

它比kafka成熟，支持AMQP事务处理，在可靠性上，RabbitMq超过kafka，在性能方面超过ActiveMQ。

ActiveMQ:

历史悠久的开源项目，已经在很多产品中得到应用，实现了JMS1.1规范，可以和spring-jms轻松融合，实现了多种协议，不够轻巧（源代码比RocketMQ多），支持持久化到数据库，对队列数较多的情况支持不好。

Kafka：

Kafka设计的初衷就是处理日志的，不支持AMQP事务处理，可以看做是一个日志系统，针对性很强，所以它并没有具备一个成熟MQ应该具备的特性
Kafka的性能（吞吐量、tps）比RabbitMq要强，如果用来做大数据量的快速处理是比RabbitMq有优势的。

AMQP协议

即高级消息队列协议，规范客户端与消息中间件服务器之间的通信，并能相互操作。

一、 channel 信道：

概念：信道是生产消费者与rabbit通信的渠道，生产者publish或是消费者subscribe一个队列都是通过信道来通信的。信道是建立在TCP连接上的虚拟连接，什么意思呢？就是说rabbitmq在一条TCP上建立成百上千个信道来达到多个线程处理，这个TCP被多个线程共享，每个线程对应一个信道，信道在rabbit都有唯一的ID ,保证了信道私有性，对应上唯一的线程使用。

疑问：为什么不建立多个TCP连接呢？原因是rabbit保证性能，系统为每个线程开辟一个TCP是非常消耗性能，每秒成百上千的建立销毁TCP会严重消耗系
统。所以rabbitmq选择建立多个信道（建立在tcp的虚拟连接）连接到rabbit上。

类似概念：TCP是电缆，信道就是里面的光纤，每个光纤都是独立的，互不影响。

二、exchange 交换机和绑定routing key

exchange的作用就是类似路由器，routing key 就是路由键，服务器会根据路由键将消息从交换器路由到队列上去。

exchange有多个种类：direct，fanout，topic

（direct）1:1,（fanout）1：N,（topic）N:1

direct： 1:1类似完全匹配

fanout：1：N 可以把一个消息并行发布到多个队列上去，简单的说就是，当多个队列绑定到fanout的交换器,那么交换器一次性拷贝多个消息分别发送到绑定的队列上，每个队列有这个消息的副本。

ps：这个可以在业务上实现并行处理多个任务，比如，用户上传图片功能，当消息到达交换器上，它可以同时路由到积分增加队列和其它队列上，达到并行处理的目的，并且易扩展，以后有什么并行任务的时候，直接绑定到fanout交换器不需求改动之前的代码。

topic N:1 ，多个交换器可以路由消息到同一个队列。根据模糊匹配，比如一个队列的routing key 为*.test ，那么凡是到达交换器的消息中的routing key 后缀.test都被路由到这个队列上。

三、Queue和Bing

Queue（队列）是RabbitMQ的内部对象，用于存储消息。

消息的生命周期：生产者生产消息A 交由信道，信道通过消息（消息由载体和标签）的标签（路由键）放到交换器发送到队列上（其实就是查询匹配，一旦匹配到了规则，信道就直接和队列产生连接，然后将消息发送过去）

绑定器（Bind）：将交换器和队列连接起来，并且封装消息的路由信息

四、主要方法

声明队列（创建队列）：可以生产者和消费者都声明，也可以消费者声明生产者不声明，也可以生产者声明而消费者不声明。最好是都声明。（生产者未声明，消费者声明这种情况如果生产者先启动，会出现消息丢失的情况，因为队列未创建）

```
channel.queueDeclare(String queue, //队列的名字
                     boolean durable, //该队列是否持久化（即是否保存到磁盘中）
                     boolean exclusive,//该队列是否为该通道独占的，即其他通道是否可以消费该队列
                     boolean autoDelete,//该队列不再使用的时候，是否让RabbitMQ服务器自动删除掉
                     Map<String, Object> arguments)//其他参数
```

四、主要方法

声明路由器（创建路由器）：生产者、消费者都要声明路由器---如果声明了队列，可以不声明路由器。

```
channel.exchangeDeclare(String exchange,//路由器的名字
                        String type,//路由器的类型：topic、direct、fanout
                        boolean durable,//是否持久化该路由器
                        boolean autoDelete,//是否自动删除该路由器
                        boolean internal,//是否是内部使用的，true的话客户端不能使用该路由器
                        Map<String, Object> arguments) //其他参数
```

四、主要方法

绑定队列和路由器：只用在消费者

```
channel.queueBind(String queue, //队列
                  String exchange, //路由器
                  String routingKey, //路由键，即绑定键
                  Map<String, Object> arguments) //其他绑定参数
```

四、主要方法

发布消息：只用在生产者

```
channel.basicPublish(String exchange, //路由器的名字，即将消息发到哪个路由器
                     String routingKey, //路由键，即发布消息时，该消息的路由键是什么
                     BasicProperties props, //指定消息的基本属性
                     byte[] body)//消息体，也就是消息的内容，是字节数组
```

四、主要方法

接收消息：只用在消费者

```
channel.basicConsume(String queue, //队列名字，即要从哪个队列中接收消息
                     boolean autoAck, //是否自动确认，默认true
                     Consumer callback)//消费者，即谁接收消息
```

```
rabbitmq:
        simple          简单模式
        workqueue       工作队列模式
        routing         路由模式
        topic           主题模式
        fanout          广播模式
```

五、案例一simple

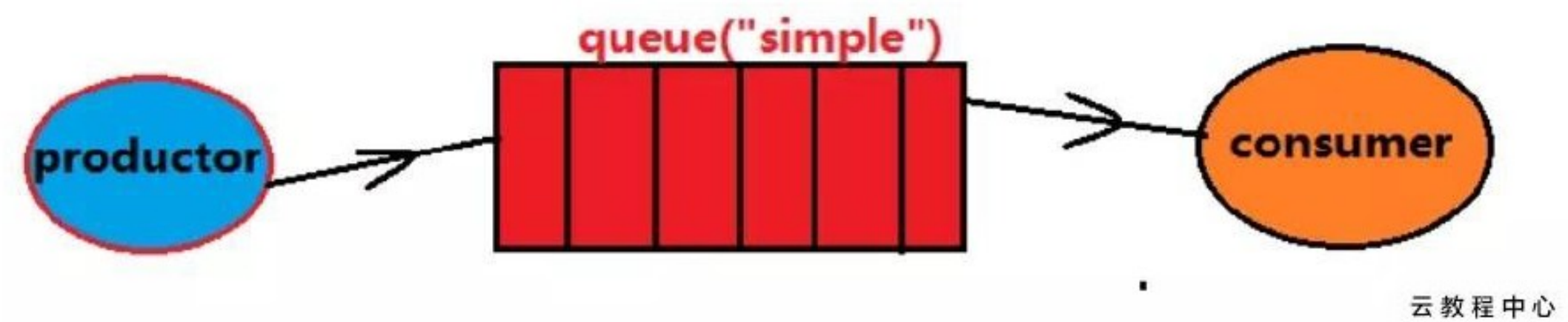
消息生产者p将消息放入队列

消费者监听队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列删除

应用场景:聊天室

1>.首先准备依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```



五、案例一simple

```
public void send() throws Exception{
    /*1 创建连接工厂
     * 2 配置共创config
     * 3 获取连接
     * 4获取信道
     * 5 从信道声明queue
     * 6 发送消息
     * 7 释放资源
     */
    ConnectionFactory factory=new ConnectionFactory();
    factory.setHost("106.23.34.56");
    factory.setPort(5672);
    factory.setVirtualHost("/tb");
    factory.setUsername("admin");
    factory.setPassword("123456");
    //从工厂获取连接
    Connection conn=factory.newConnection();
    //从连接获取信道
    Channel chan=conn.createChannel();
    //利用channel声明第一个队列
    chan.queueDeclare("simple", false, false, false, null);
    //发送消息
    String msg="helloworld,nihaoa";
    chan.basicPublish("", "simple", null, msg.getBytes());
}
```

五、案例一simple

```
    //模拟消费端
  public void receive() throws Exception{


ConnectionFactory factory=new ConnectionFactory();
factory.setHost("106.23.34.56");
factory.setPort(5672);
factory.setVirtualHost("/tb");
factory.setUsername("admin");
factory.setPassword("123456");
//从工厂获取连接

Connection conn=factory.newConnection();/
/从连接获取信道
Channel chan=conn.createChannel();
chan.queueDeclare("simple", false, false, false, null);/
/创建一个消费者
QueueingConsumer consumer= new QueueingConsumer(chan);
chan.basicConsume("simple", consumer);
//监听队列while(true){
//获取下一个delivery,delivery从队列获取消息
Delivery delivery = consumer.nextDelivery();
String msg=new String(delivery.getBody());
System.out.println(msg);}}
}
```

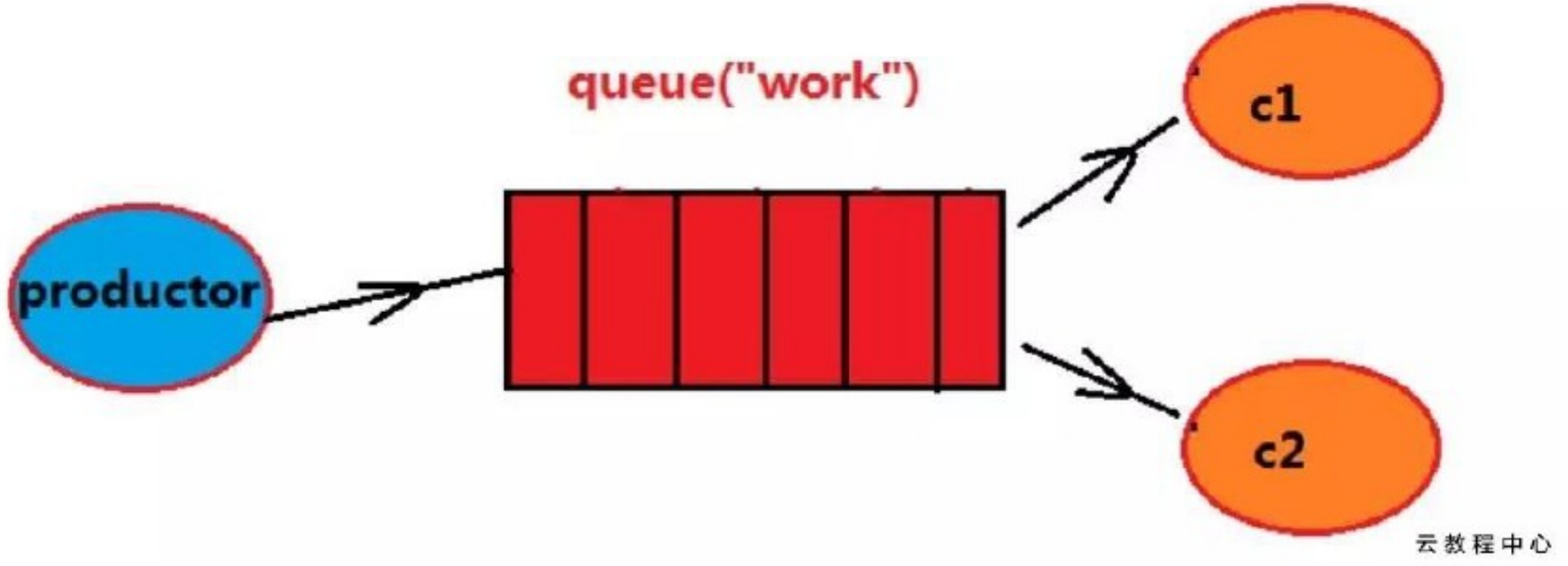
五、案例二work

生产者将消息放入队列多个消费者同时监听同一个队列,消息如何被消费?C1,C2共同争抢当前消息队列的内容,谁先拿到消息,谁来负责消费

应用场景:

红包;

大型项目中的资源调度过程(直接由最空闲的系统争抢到资源处理任务)



五、案例二work

```
    public class WorkTest {
@Test
public void send() throws Exception{
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明队列
  chan.queueDeclare("work", false, false, false, null);
  for(int i=0;i<100;i++){
    String msg="1712,hello:"+i+"message";
    chan.basicPublish("", "work", null, msg.getBytes());
    System.out.println("第"+i+"条信息已经发送");
  }
  chan.close();
  conn.close();
}
```

五、案例二work

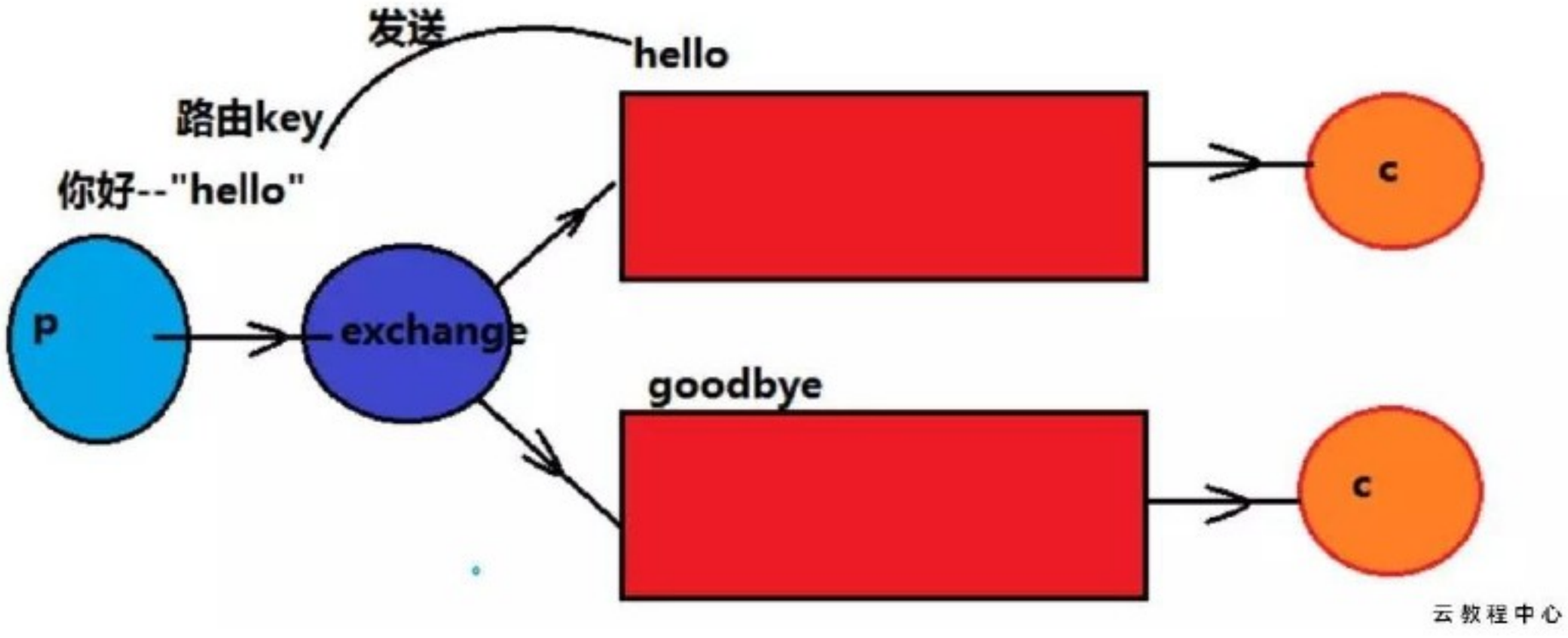
```
public void receive1() throws Exception{
  //获取连接,获取信道
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  chan.queueDeclare("work", false, false, false, null);
  //同一时刻服务器只发送一条消息给同一消费者,消费者空闲,才发送一条
  chan.basicQos(1);
  //定义消费者
  QueueingConsumer consumer=new QueueingConsumer(chan);
  //绑定队列和消费者的关系
  //queue
  //autoAck:消息被消费后,是否自动确认回执,如果false,不自动需要手动在
  //完成消息消费后进行回执确认,channel.ack,channel.nack
  //callback
  //chan.basicConsume(queue, autoAck, callback)
  chan.basicConsume("work", false, consumer);
  //监听
  while(true){
    Delivery delivery=consumer.nextDelivery();
    byte[] result = delivery.getBody();
    String msg=new String(result);
    System.out.println("接受到:"+msg);
    Thread.sleep(50);
    //返回服务器,回执
    chan.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
  }
}
```

五、案例三routing模式

生产者发送消息到交换机,同时绑定一个路由Key,交换机根据路由key对下游绑定的队列进行路

由key的判断,满足路由key的队列才会接收到消息,消费者消费消息

应用场景: 项目中的error报错



五、案例三routing模式

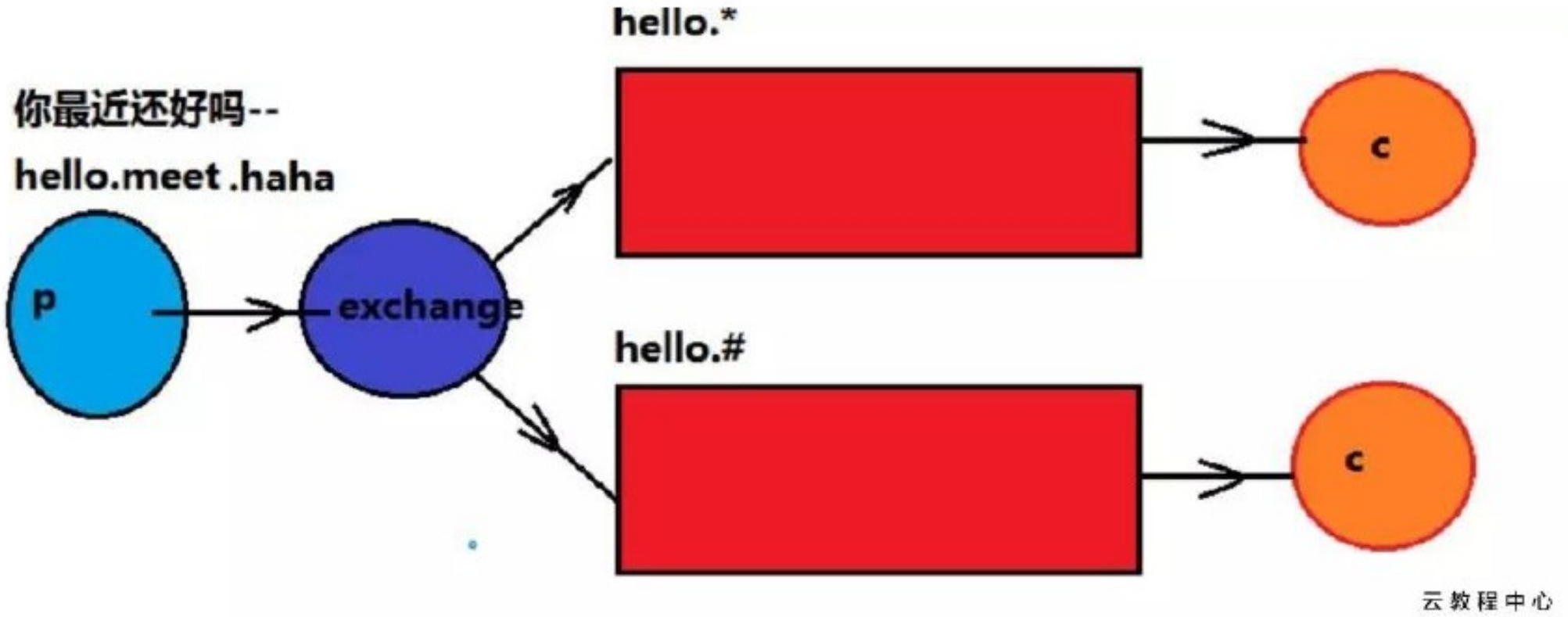
```
    public void routingSend() throws Exception{
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明交换机
  //参数意义,1 交换机名称,2 类型:fanout,direct,topic
  chan.exchangeDeclare("directEx", "direct");
  //发送消息
  String msg="路由模式的消息";
  chan.basicPublish("directEx", "jt1713",
      null, msg.getBytes());
}
```

五、案例三routing模式

```
    public void routingRec01() throws Exception{
  System.out.println("一号消费者等待接收消息");
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明队列
  chan.queueDeclare("direct01", false, false, false, null);
  //声明交换机
  chan.exchangeDeclare("directEx", "direct");
  //绑定队列到交换机
  //参数 1 队列名称,2 交换机名称 3 路由key
  chan.queueBind("direct01", "directEx", "jt1712");
  chan.basicQos(1);
  //定义消费者
  QueueingConsumer consumer=new QueueingConsumer(chan);
  //消费者与队列绑定
  chan.basicConsume("direct01",false, consumer);
  while(true){
    Delivery delivery= consumer.nextDelivery();
    System.out.println("一号消费者接收到"+
    new String(delivery.getBody()));
    chan.basicAck(delivery.getEnvelope().
        getDeliveryTag(), false);
  }
}
```

五、案例四topic模式

*号代表单个词语
#代表多个词语



五、案例四topic模式

```
    @Test
public void topicSend() throws Exception{
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明交换机
  //参数意义,1 交换机名称,2 类型:fanout,direct,topic
  chan.exchangeDeclare("topicEx", "topic");
  //发送消息
  String msg="主题模式的消息";
  chan.basicPublish("topicEx", "jt1712.add.update",
      null, msg.getBytes());
}
```

五、案例四topic模式

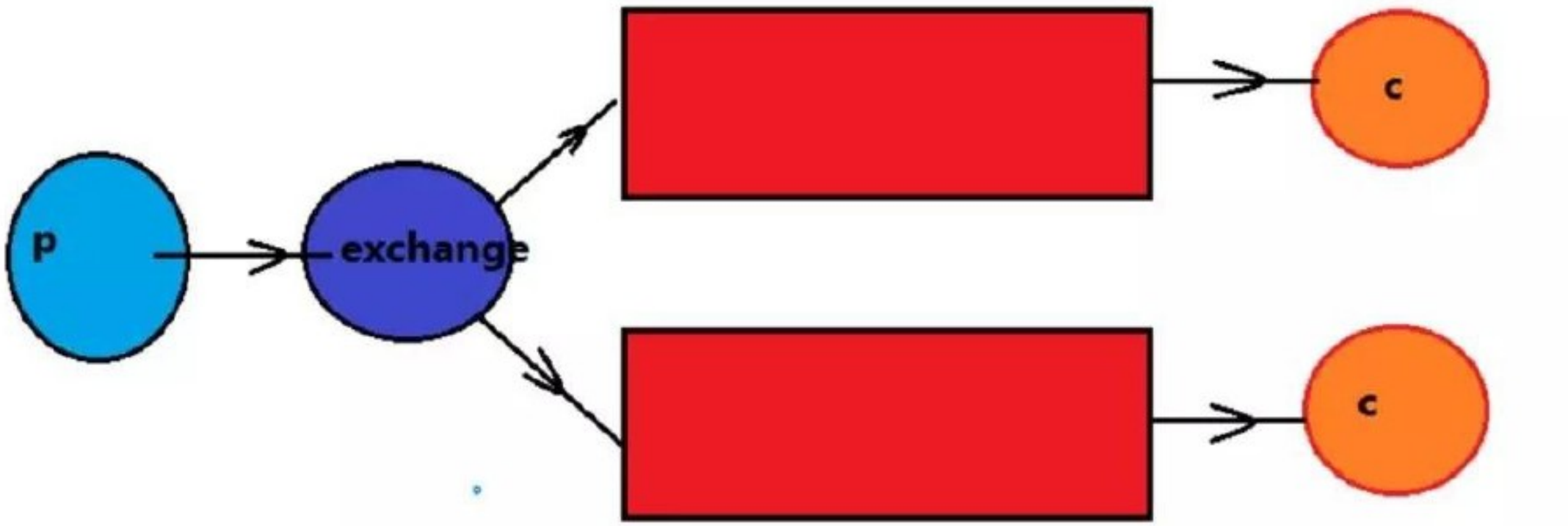
```
    @Test
public void topicRec01() throws Exception{
  System.out.println("一号消费者等待接收消息");
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明队列
  chan.queueDeclare("topic01", false, false, false, null);
  //声明交换机
  chan.exchangeDeclare("topicEx", "topic");
  //绑定队列到交换机
  //参数 1 队列名称,2 交换机名称 3 路由key
  chan.queueBind("topic01", "topicEx", "jt1712.#");
  chan.basicQos(1);
  //定义消费者
  QueueingConsumer consumer=new QueueingConsumer(chan);
  //消费者与队列绑定
  chan.basicConsume("topic01",false, consumer);
  while(true){
    Delivery delivery= consumer.nextDelivery();
    System.out.println("一号消费者接收到"+
    new String(delivery.getBody()));
    chan.basicAck(delivery.getEnvelope().
        getDeliveryTag(), false);
  }
}
```

五、案例五fanout模式

生产者将消息交给交换机

有交换机根据发布订阅的模式设定将消息同步到所有的绑定队列中;后端的消费者都能拿到消息

应用场景:邮件群发,群聊天,广告



五、案例五fanout模式

```
    @Test
public void send() throws Exception {
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //声明交换机
  //参数意义,1 交换机名称,2 类型:fanout,direct,topic
  chan.exchangeDeclare("fanoutEx", "fanout");
  //发送消息
  for(int i=0;i<100;i++){
    String msg="1712 hello:"+i+"msg";
    chan.basicPublish("fanoutEx", "", null, msg.getBytes());
    System.out.println("第"+i+"条信息已经发送");
  }
}
```

五、案例五fanout模式

```
    @Test
public void receiv01() throws Exception{
  //获取连接
  Connection conn = ConnectionUtil.getConn();
  Channel chan = conn.createChannel();
  //生命队列
  chan.queueDeclare("fanout01", false, false, false, null);
  //声明交换机
  chan.exchangeDeclare("fanoutEx", "fanout");
  //绑定队列到交换机
  //参数 1 队列名称,2 交换机名称 3 路由key
  chan.queueBind("fanout01", "fanoutEx", "");
  chan.basicQos(1);
  //定义消费者
  QueueingConsumer consumer=new QueueingConsumer(chan);
  //消费者与队列绑定
  chan.basicConsume("fanout01",false, consumer);
  while(true){
    Delivery delivery= consumer.nextDelivery();
    System.out.println("一号消费者接收到"+
    new String(delivery.getBody()));
    chan.basicAck(delivery.getEnvelope().
        getDeliveryTag(), false);
  }
}
```

六、springboot下的fanout模式

Fanout 就是我们熟悉的广播模式或者订阅模式，给Fanout交换机发送消息，绑定了这个交换机的所有队列都收到这个消息。

```
@Configuration
public class FanoutRabbitConfig {

    @Bean
    public Queue AMessage() {
        return new Queue("fanout.A");
    }
    @Bean
    public Queue BMessage() {
        return new Queue("fanout.B");
    }
    @Bean
    public Queue CMessage() {
        return new Queue("fanout.C");
    }
    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    @Bean
    Binding bindingExchangeA(Queue AMessage,FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(AMessage).to(fanoutExchange);
    }
    @Bean
    Binding bindingExchangeB(Queue BMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(BMessage).to(fanoutExchange);
    }
    @Bean
    Binding bindingExchangeC(Queue CMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(CMessage).to(fanoutExchange);
    }

}
```

六、springboot下的fanout模式

发送者

```
        public void send() {
         String context = "hi, fanout msg ";
         System.out.println("Sender : " + context);
         this.rabbitTemplate.convertAndSend("fanoutExchange","", context);
}
```

其中A的消费者

```
@RabbitListener(queues = "fanout.A")
public class FanoutReceiverA {
    @RabbitHandler
    public void process(String message) {
        System.out.println("fanout Receiver A  : " + message);
    }
}
```

结果如下：

```
Sender : hi, fanout msg
...
fanout Receiver B: hi, fanout msg
fanout Receiver A  : hi, fanout msg
fanout Receiver C: hi, fanout msg
```

七、springboot下的topic模式

```
@Configuration
public class TopicRabbitConfig {

    final static String message = "topic.message";
    final static String messages = "topic.messages";

    @Bean
    public Queue queueMessage() {
        return new Queue(TopicRabbitConfig.message);
    }

    @Bean
    public Queue queueMessages() {
        return new Queue(TopicRabbitConfig.messages);
    }

    @Bean
    TopicExchange exchange() {
        return new TopicExchange("topicExchange");
    }

    @Bean
    Binding bindingExchangeMessage(Queue queueMessage, TopicExchange exchange) {
        return BindingBuilder.bind(queueMessage).to(exchange).with("topic.message");
    }

    @Bean
    Binding bindingExchangeMessages(Queue queueMessages, TopicExchange exchange) {
        return BindingBuilder.bind(queueMessages).to(exchange).with("topic.#");
    }
}
```

七、springboot下的topic模式

发送者

```
    public void send() {
        String context = "hi, i am message all";
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("topicExchange", "topic.1", context);
        }
```

消费者

```
@Component
@RabbitListener(queues = "topic.messages")
public class TopicReceiver {
    @RabbitHandler
    public void process(String message) {
        System.out.println("Topic Receiver1  : " + message);
    }
}
```

结果如下：

```
Sender : hi, i am message all
...
Topic Receiver1  : hi, i am message all
```

八、springboot下的direct模式

```
@Configuration
public class DirectRabbitConfig {

                public static final String DIRECT_QUEUE_A = "direct_queue_a";
                public static final String DIRECT_EXCHANGE = "direct_exchange";
                public static final String DIRECT_ROUTING_KEY_A = "direct.a";

    @Bean
    public Queue directQueueA() {
        return new Queue(DIRECT_QUEUE_A);
    }

    @Bean
    public DirectExchange directExchange() {
        return new DirectExchange(DIRECT_EXCHANGE);
    }

    @Bean
    public Binding directBinding001(Queue directQueueA, DirectExchange directExchange) {
                return BindingBuilder.bind(directQueueA).to(directExchange).with(DIRECT_ROUTING_KEY_A);
    }

}
```

八、springboot下的direct模式

发送者

```
public void send() {
    String context = "hi, i am direct";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("direct_exchange", "direct.a", context);
}
```

消费者

```
@Component
@RabbitListener(queues = "direct_queue_a")
public class DirectReceiver {
    @RabbitHandler
    public void process(String text) {
        System.out.println("Receiver direct : " + text);
    }
}
```

结果如下：

```
Sender : hi, i am direct
...
Receiver direct : hi, i am direct
```