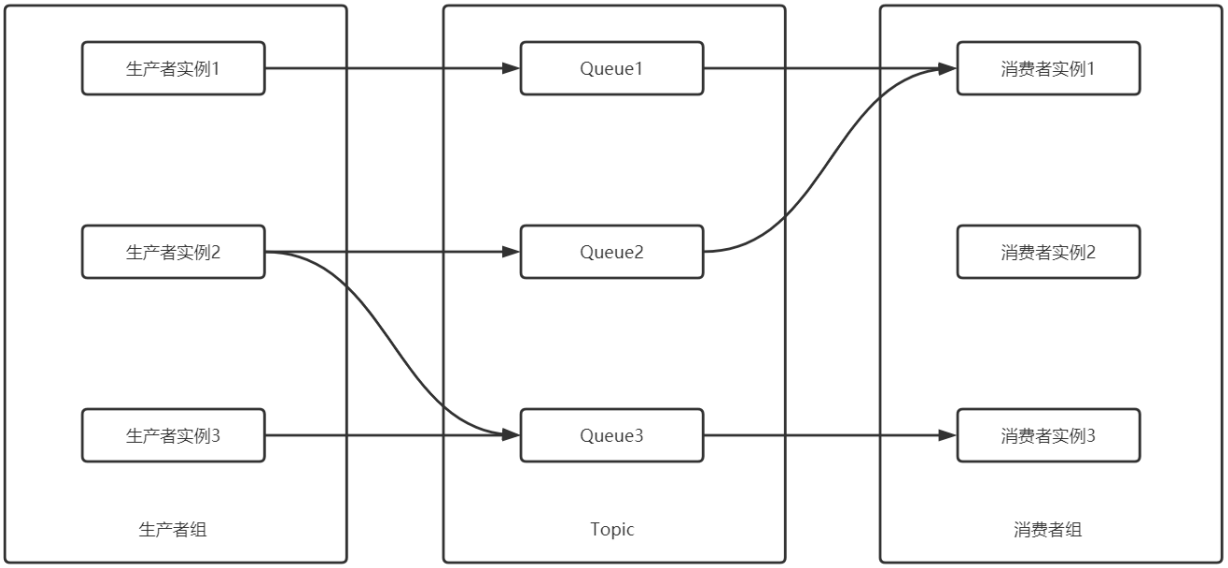
生产者启动流程与消息发送流程

生产者概述

发送消息的一方被称为生产者，它在整个RocketMQ的生产和消费体系中扮演的角色。



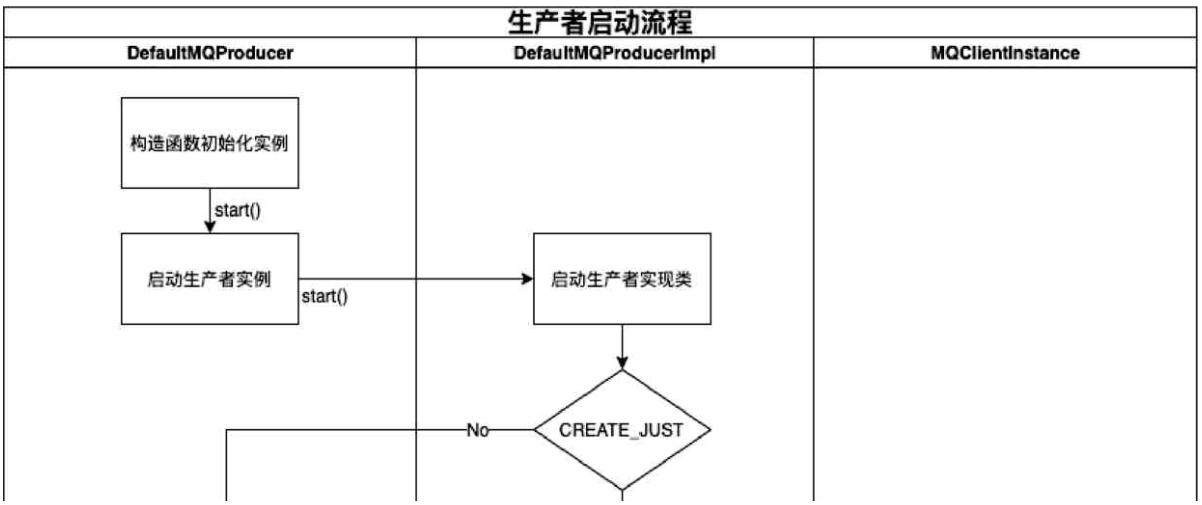
- 生产者组：一个逻辑概念，在使用生产者实例的时候需要指定一个组名。一个生产者组可以生产多个Topic的消息。
- 生产者实例：一个生产者组部署了多个进程，每个进程都可以称为一个生产者实例。
- Topic：主题名字，一个Topic由若干Queue组成。
- RocketMQ 客户端中的生产者有两个独立实现类：
    - org.apache.rocketmq.client.producer.**DefaultMQProducer**
    - org.apache.rocketmq.client.producer.**TransactionMQProducer。**
- 前者用于生产普通消息、顺序消息、单向消息、批量消息、延迟消息，后者主要用于生产事务消息。

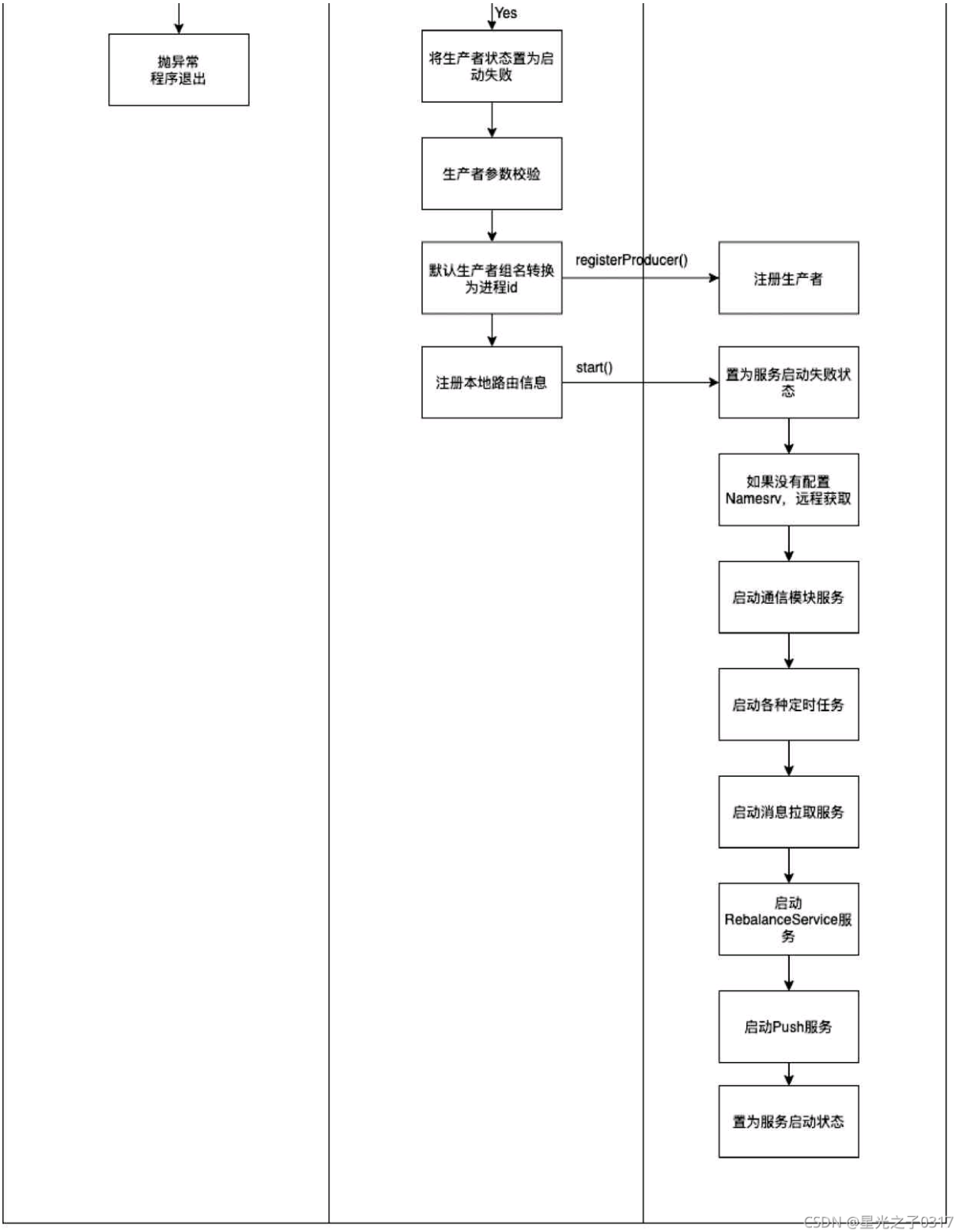
消息的结构（了解一下）

```
public class Message implements Serializable {
    private static final long serialVersionUID = 8445773977080406428L;
    private String topic;
    private int flag;
    private Map<String, String> properties;
    private byte[] body;
    public void setKeys(String keys) { }
    public void setKeys(Collection<String> keys) { }
    public void setTags(String tags) { }
    public void setDelayTimeLevel(int level) { }
    public void setTopic(String topic) { }
    public void putUserProperty(final String name, final String value) {…}
}
```

- Topic：主题名字，可以通过RocketMQ Console创建。
- Flag：目前没用。
- Properties：消息扩展信息，Tag、keys、延迟级别都保存在这里。
- Body：消息体，字节数组。需要注意生产者使用什么编码，消费者也必须使用相同编码解码，否则会产生乱码。
- setKeys（）：设置消息的key，Key用于唯一标识这个消息，相当于消息id，多个key可以用MessageConst.KEY_SEPARATOR（空格）分隔或者直接用另一个重载方法。
- 如果 Broker 中 messageIndexEnable=true 则会根据 key创建消息的Hash索引，帮助用户进行快速查询。
- setTags（）：消息过滤的标记，用户可以订阅某个Topic的某些Tag，这样Broker只会把订阅了topic-tag的消息发送给消费者。
- setDelayTimeLevel（）：设置延迟级别，延迟多久消费者可以消费。
- putUserProperty（）：如果还有其他扩展信息，可以存放在这里。内部是一个Map，重复调用会覆盖旧值。

生产者启动流程





涉及的类

DefaultMQProducer：默认生产者实现类

DefaultMQProducerImpl：默认生产者的具体实现类，被DefaultMQProducer引用

MQClientInstance：MQ客户端实例，MQClientInstance包含了生产者与消费者需要的所有底层功能。

关键启动流程

1. 调用producer.start()开始启动生产者实例，实例状态为CREATE_JUST，生产者可用状态为“失败”

```
//ProducerSample01
DefaultMQProducer producer = new DefaultMQProducer("pg1");
producer.setNamesrvAddr("192.168.31.103:9876");
try {
    producer.start();
    ...
}
...
```

2. 校验生产者实例设置的各种参数。比如生产者组名是否为空、是否满足命名规则、长度是否满足等。

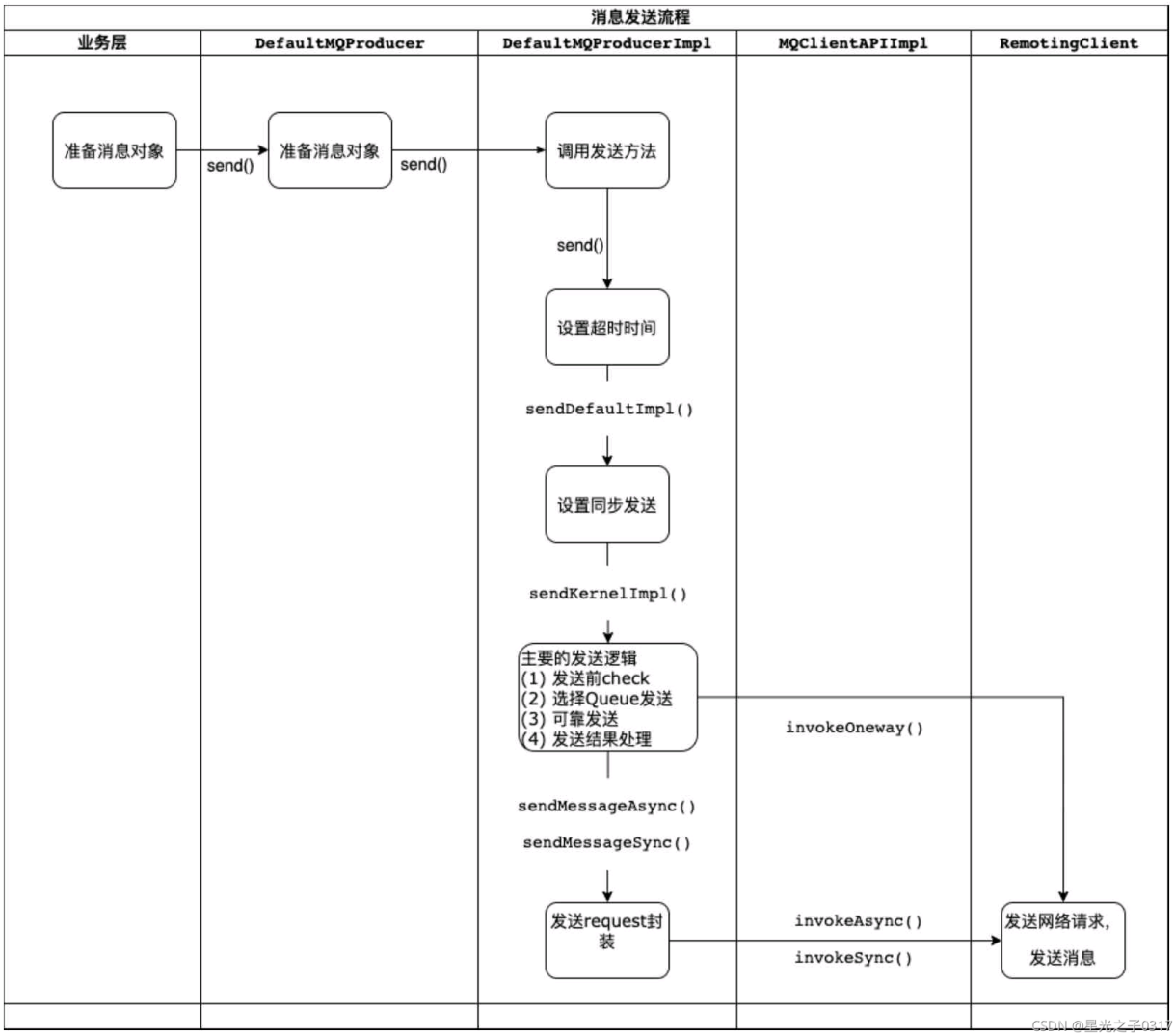
3. 执行changeInstanceNameToPID（）方法。校验instance name，如果是默认名字则将其修改为进程id

```
//DefaultMQProducerImpl
public void changeInstanceNameToPID() {
    if (this.instanceName.equals("DEFAULT")) {
        this.instanceName = UtilAll.getPid() + "#" + System.nanoTime();
    }
}
```

4. 创建MQClientInstance实例并初始化，按MQClientInstance负责NameSrv通信获取Broker配置、启动各种服务模块、开启各种定时任务

5. MQClientInstance初始化完毕，生产者启动完毕

消息发送流程



RocketMQ客户端的消息发送通常分为以下3层：

- 业务层：通常指直接调用RocketMQ Client发送API的业务代码。
- 消息处理层：指RocketMQ Client获取业务发送的消息对象后，一系列的参数检查、消息发送准备、参数包装等操作。
- 通信层：指RocketMQ基于Netty封装的一个RPC通信服务，RocketMQ的各个组件之间的通信全部使用该通信层。

消息发送步骤

1. 调用defaultMQProducer.send()方法准备发送消息。

```
for(int i = 0 ; i< 10000 ; i++) {
   String data = "{\"title\":\"X市2021年度第四季度税务汇总数据\"}";
   Message message = new Message("tax-data", "2021S4", data.getBytes());
   SendResult result = producer.send(message);
   System.out.println("消息已发送：MsgId:" + result.getMsgId() + "，发送状态:" +
result.getSendStatus());
}
```

2. 通过设置的发送超时时间，默认3秒

3. 调用 defaultMQProducerImpl.sendDefaultImpl() 设置发送方式，可选值ASYNC（异步）| ONEWAY（单向）| SYNC（同步）

4. defaultMQProducerImpl.sendKernelImpl()用于控制发送过程

- 前置检查
- 选择Queue进行发送
- 可靠发送
- 发送结果处理

5. 根据前面设置的CommunicationMode（通信模式）,MQClientAPIImpl.sendMessage()调用remotingClient对象不同的方法完成通信。

```
switch (communicationMode) {
    case ONEWAY:
        //单向通道方法
        this.remotingClient.invokeOneway(addr, request, timeoutMillis);
        return null;
    case ASYNC:
        final AtomicInteger times = new AtomicInteger();
        long costTimeAsync = System.currentTimeMillis() - beginStartTime;
        if (timeoutMillis < costTimeAsync) {
            throw new RemotingTooMuchRequestException("sendMessage call
timeout");
        }
        //异步传输
        this.sendMessageAsync(addr, brokerName, msg, timeoutMillis -
costTimeAsync, request, sendCallback, topicPublishInfo, instance,
            retryTimesWhenSendFailed, times, context, producer);
        return null;
    case SYNC:
        long costTimeSync = System.currentTimeMillis() - beginStartTime;
        if (timeoutMillis < costTimeSync) {
            throw new RemotingTooMuchRequestException("sendMessage call
timeout");
        }
        //同步调用
        return this.sendMessageSync(addr, brokerName, msg, timeoutMillis -
costTimeSync, request);
    default:
        assert false;
        break;
}
```

上述三个方法最终都是通过remotingClient提供的invokeXXX方法完成与Broker的通信，底层基于Netty框架实现异步网络传输。

- remotingClient.invokeSync //异步
- remotingClient.invokeAsync //同步
- remotingClient.invokeOneway //单向