

RocketMQ研究

1 RocketMQ介绍

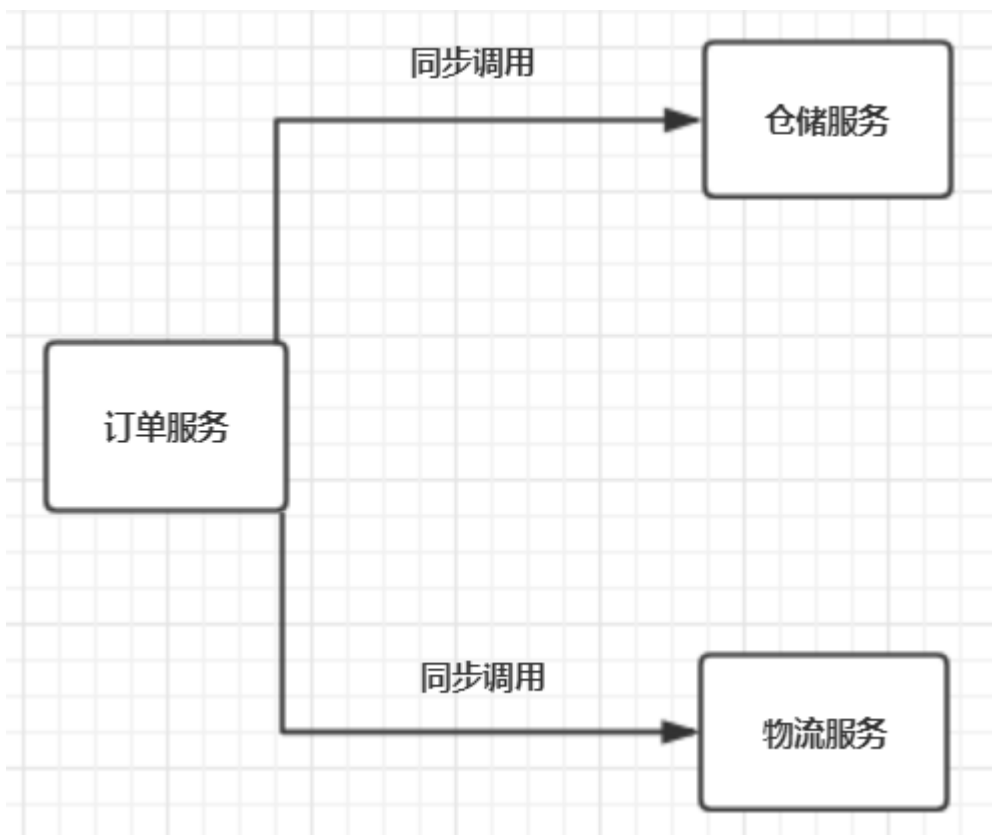
1.1 MQ的应用场景

MQ全称为Message Queue，即消息队列，开发中消息队列通常有如下应用场景：

1、任务异步处理。

将不需要同步处理的并且耗时长操作由消息队列通知消息接收方进行异步处理。提高了应用程序的响应时间。

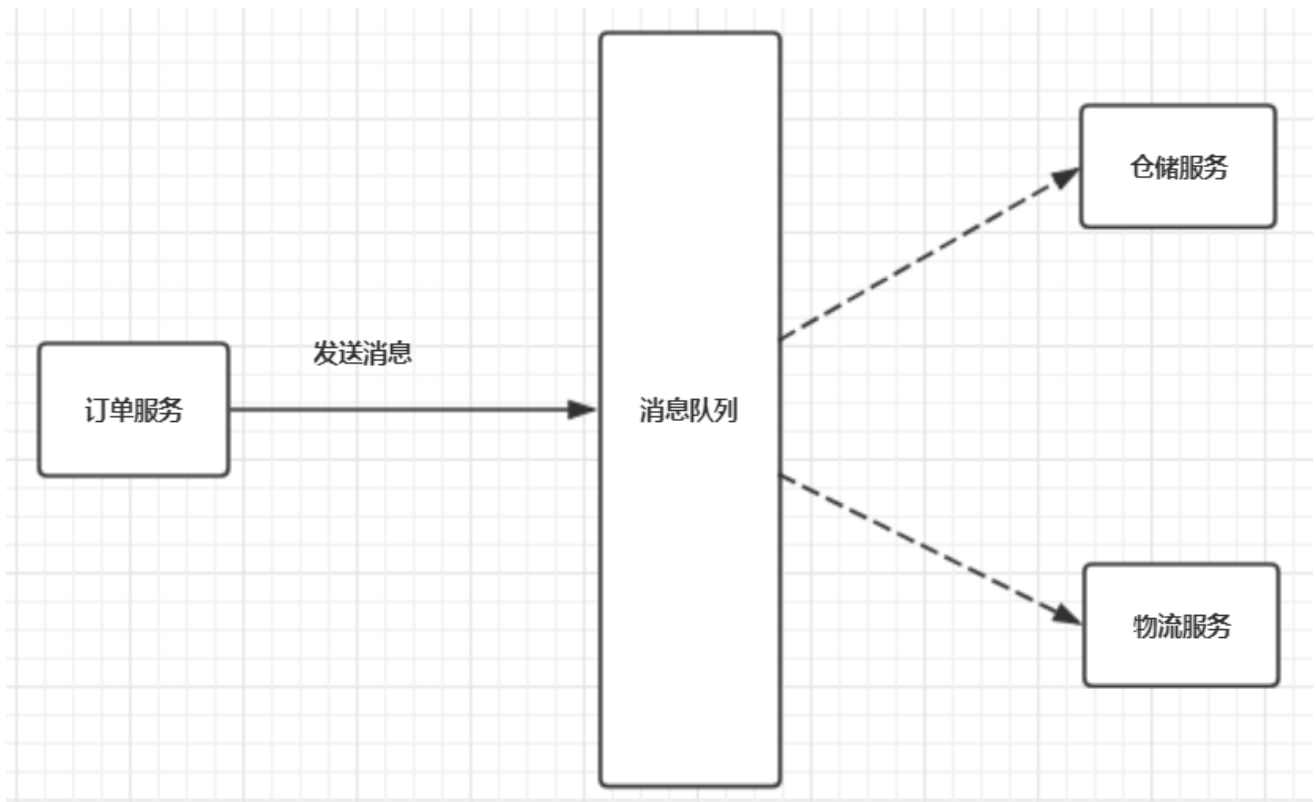
传统方式：



改造后方式:

交互流程如下：

- 1、订单服务发消息到消息队列。
- 2、消息队列将消息发给仓储服务和物流服务。
- 3、仓储服务和物流服务接收到消息进行业务处理。



2、应用程序解耦合

MQ相当于一个中介，生产方通过MQ与消费方交互，它将应用程序进行解耦合。

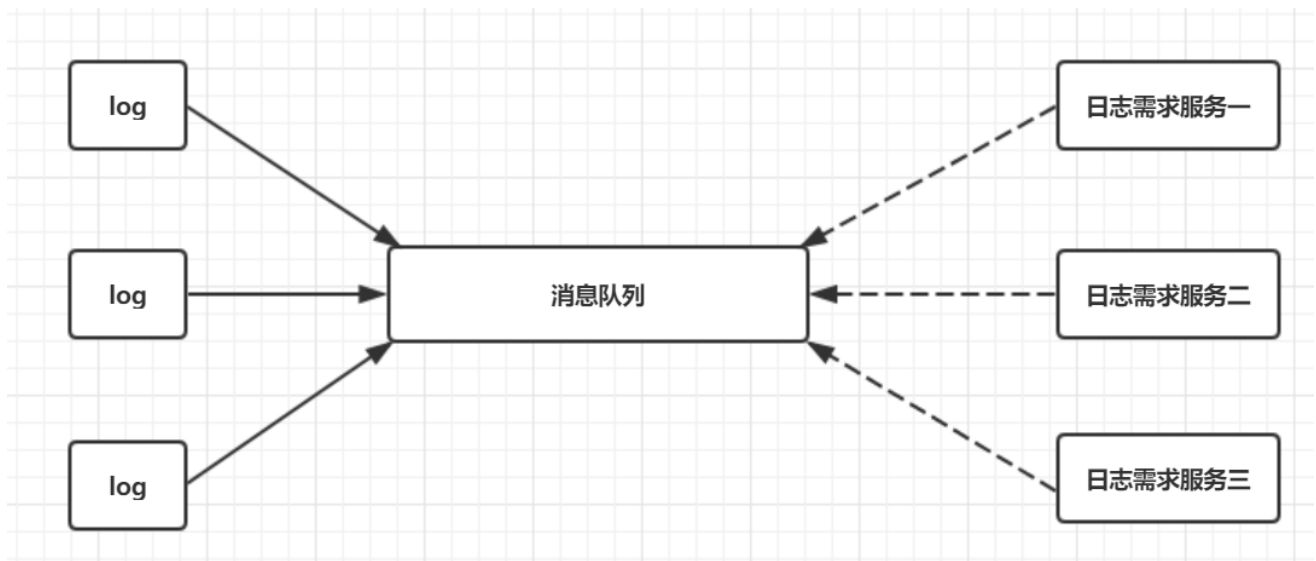
举例：上图中，消息队列将订单服务和仓储服务进行解耦合，将订单服务和物流服务进行解耦合。

3) 日志收集：

进行统一业务日志收集，供分析系统进行数据分析，消息队列作为日志数据的中转站。

交互流程如下：

- 1、采集系统从log日志文件采集数据，发送至消息队列。
- 2、各日志需求服务从消息队列 接收消息进行日志处理。



1.2 RocketMQ介绍

RocketMQ是阿里开源的一款非常优秀中间件产品，脱胎于阿里的另一款队列技术MetaQ，后捐赠给Apache基金会作为一款孵化技术，仅仅经历了一年多的时间就成为Apache基金会的顶级项目。并且它现在已经在阿里内部被广泛的应用，并且经受住了多次双十一的这种极致场景的压力。（**2017年的双十一，RocketMQ流转的消息量达到了万亿级，峰值TPS达到5600万**）。并且其内部通过Java语言开发，便于阅读与修改。

<http://rocketmq.apache.org/>

1.3 消息队列技术选型对比

市场上还有哪些消息队列？

ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，Redis。

本项目选用RocketMQ的一个主要原因如下：

- 1、支持事务消息
- 2、支持延迟消息
- 3、天然支持集群、负载均衡
- 4、支持指定次数和时间间隔的失败消息重发

详细的技术选型对比如下：

RabbitMQ:

优点:

- 1.支持AMQP协议
- 2.基于erlang语言开发，高并发性能较好
- 3.工作模式较为灵活
- 4.支持延迟消息
- 5.提供较为友好的后台管理页面



6. 单机部署，1~2WTPS

缺点：

1. 不支持水平扩容
2. 不支持事务
3. 消息吞吐量三者最差
4. 当产生消息堆积，性能下降明显
5. 消息重发机制需要手动设置
6. 不支持消息重复消费

RocketMQ:

优点：

1. 高可用，高吞吐量，海量消息堆积，低延迟性能上，都表现出色
2. api与架构设计更加贴切业务场景
3. 支持顺序消息
4. 支持事务消息
5. 支持消息过滤
6. 支持重复消费
7. 支持延迟消息
8. 支持消息跟踪
9. 天然支持集群、负载均衡
10. 支持指定次数和时间间隔的失败消息重发
11. 单机部署，5~10WTPS

缺点：

1. 生态圈相较Kafka有所不如
2. 消息吞吐量与消息堆积能力也不如Kafka
3. 不支持主从自动切换
4. 只支持Java

Kafka:

优点：

1. 高可用，高吞吐量，低延迟性能上，都表现出色
2. 使用人数多，技术生态圈完善
3. 支持顺序消息
4. 支持多种客户端
5. 支持重复消费

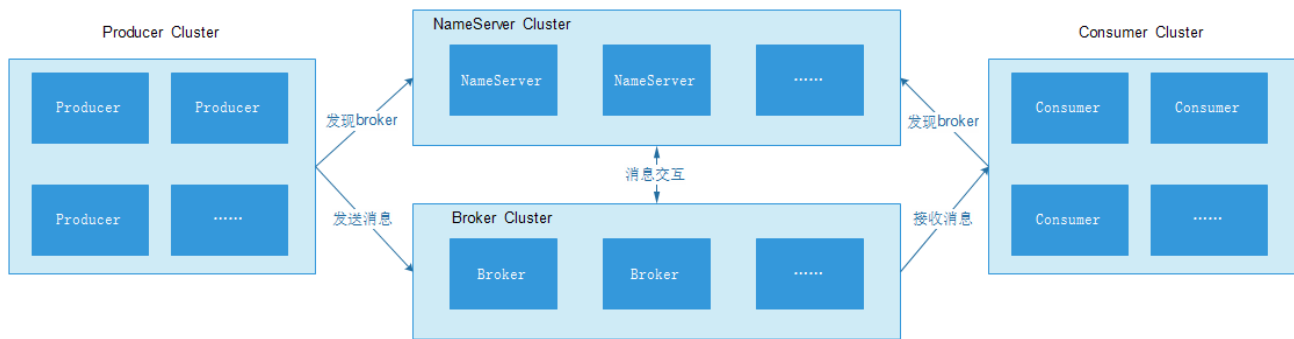
缺点：

1. 依赖分区，消费者数量受限于分区数
2. 单机消息过多时，性能下降明显
3. 不支持事务消息
4. 不支持指定次数和时间间隔的失败消息重发

2 RocketMQ基础

2.1 组成结构

RocketMQ组成结构图如下：



交互过程如下：

- 1) Broker定时发送自身状态 到NameServer。
- 2) Producer 请求NameServer获取Broker的地址。
- 3) Producer 将消息发送到Broker中的消息队列。
- 4) Consumer订阅Broker中的消息队列，通过拉取消息，或由Broker将消息推送至Consumer。

1) Producer Cluster 消息生产者群

- 负责发送消息，一般由业务系统负责产生消息。

2) Consumer Cluster 消息费群

- 负责消费消息，一般是后台系统负责异步消费。
- 两种消费模式：
 - Push Consumer，服务端向消费者端推送消息
 - Pull Consumer，消费者端向服务定时拉取消息

3) NameServer 名称服务器

- 集群架构中的组织协调员，相当于注册中心，收集broker的工作情况，不负责消息的处理

4) Broker 消息服务器

- 是RocketMQ的核心，负责消息的接受，存储，发送等。
- 需要定时发送自身状态 到NameServer，默认10秒发送一次，超时2分钟会认为该broker失效。

2.2 安装RocketMQ

2.2.1 环境要求

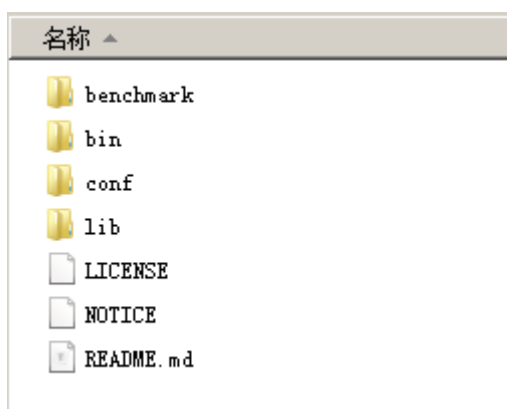
1. 64位JDK 1.8+;
2. Maven 3.2.x;
3. 64位操作系统系统，本文档在Windows上安装。

2.2.2 下载

下载地址：<http://archive.apache.org/dist/rocketmq/4.5.0/rocketmq-all-4.5.0-bin-release.zip>

版本使用：4.5.0

下载后解压到一个没有空格和中文的目录。



2.2.3 启动

1) 配置

开发环境不需要太多的内存,这里调小一点.

Broker默认磁盘空间利用率达到85%就不再接收，这里在开发环境可以提高磁盘空间利用率报警阈值为98%。

调整默认的内存大小参数

```
cd bin/
```

```
vim runserver.cmd
```

```
set "JAVA_OPT=%JAVA_OPT% -server -Xms512m -Xmx512m -Xmn512m -XX:MetaspaceSize=128m -  
XX:MaxMetaspaceSize=320m"
```

```
cd bin/
```

```
vim runbroker.cmd
```

```
set "JAVA_OPT=%JAVA_OPT% -server -Drocketmq.broker.diskSpaceWarningLevelRatio=0.98 -Xms512m -  
Xmx512m -Xmn512m"
```

2) 启动NameServer

进入cmd命令窗口，执行bin/mqnamesrv.cmd

```
f:  
cd F:\devenv\rocketmq-all-4.5.0-bin-release\bin  
mqnamesrv.cmd
```

3) 启动broker

进入cmd命令窗口，执行bin/mqbroker.cmd -n 127.0.0.1:9876

```
f:
cd F:\devenv\rocketmq-all-4.5.0-bin-release\bin
mqbroker.cmd -n 127.0.0.1:9876
```

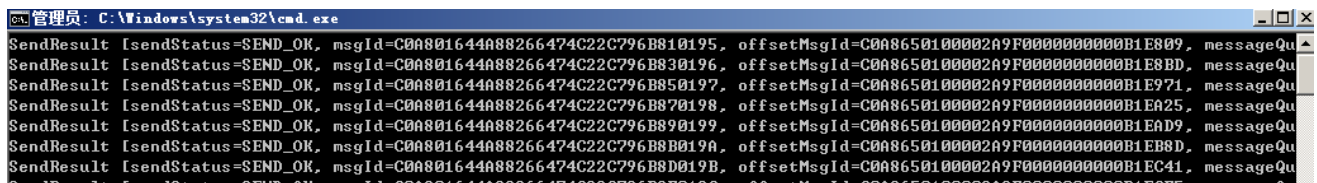
-n：指定NameServer的地址

2.2.4 测试

1、发送消息

进入cmd命令窗口，执行：

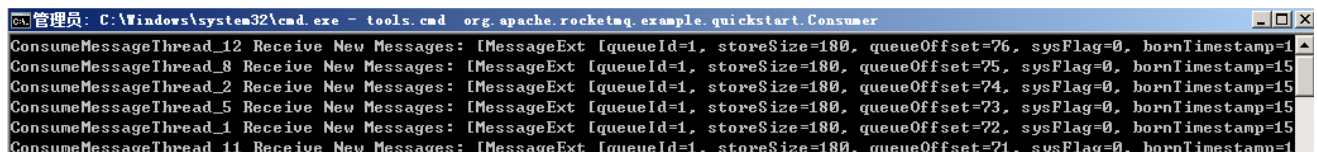
```
set NAMESRV_ADDR=127.0.0.1:9876
cd F:\devenv\rocketmq-all-4.5.0-bin-release\bin
tools.cmd org.apache.rocketmq.example.quickstart.Producer
```



2、接送消息

进入cmd命令窗口，执行：

```
set NAMESRV_ADDR=127.0.0.1:9876
tools.cmd org.apache.rocketmq.example.quickstart.Consumer
```



2.2.5 安装管理端

RocketMQ提供了UI管理工具，名为rocketmq-console，项目地址：<https://github.com/apache/rocketmq-externals/tree/master/rocketmq-console>

1) 下载源代码

见资料文件夹rocketmq-console.zip

解压rocketmq-console.zip

2) 修改配置

修改rocketmq-console\src\main\resources\application.properties

server.port=9877 // 服务端口号 rocketmq.config.namesrvAddr=127.0.0.1:9876 // 名称服务地址

rocketmq.config.dataPath=/tmp/rocketmq-console/data // mq数据路径

```
1 server.contextPath=
2 server.port=8080
3 #spring.application.index=true
4 spring.application.name=rocketmq-console
5 spring.http.encoding.charset=UTF-8
6 spring.http.encoding.enabled=true
7 spring.http.encoding.force=true
8 logging.config=classpath:logback.xml
9 #if this value is empty, use env value rocketmq.config.namesrvAddr NAMESRV_ADDR | now, you can set it in ops page.default localhost:9876
10 rocketmq.config.namesrvAddr=127.0.0.1:9876
11 #if you use rocketmq version < 3.5.8, rocketmq.config.isVIPChannel should be false.default true
12 rocketmq.config.isVIPChannel=
13 #rocketmq-console's data path:dashboard/monitor
14 rocketmq.config.dataPath=/tmp/rocketmq-console/data
15 #set it false if you don't want use dashboard.default true
16 rocketmq.config.enableDashBoardCollect=true
```

3) 打包

进入rocketmq-console目录下

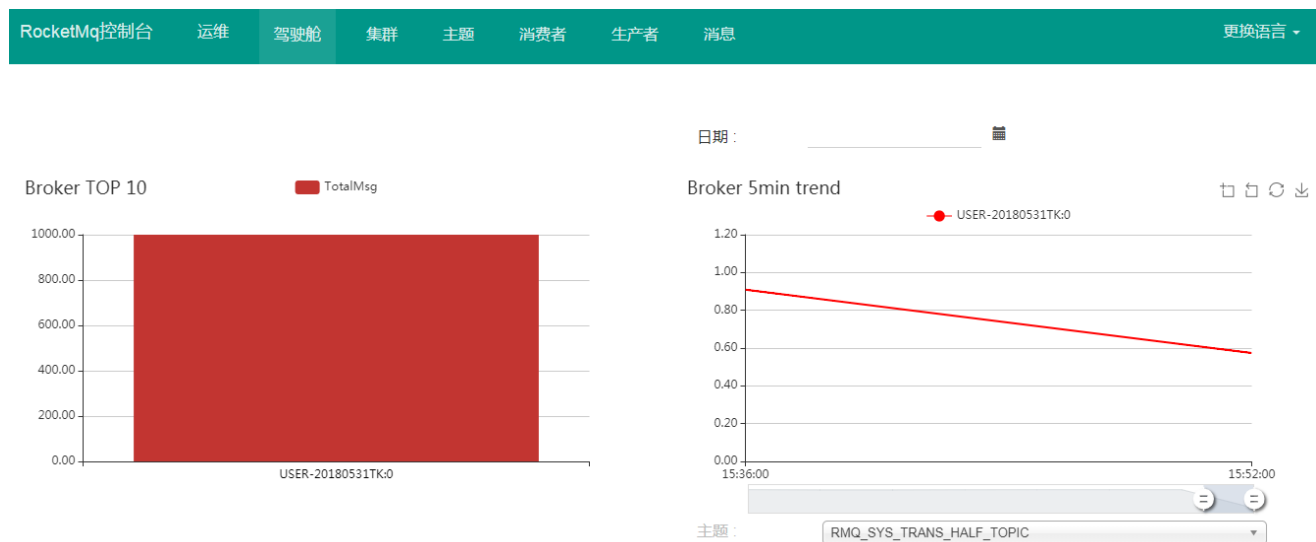
cmd下执行：

```
mvn clean package -Dmaven.test.skip=true
```

4) 启动 进入/rocketmq-console/target

java -jar rocketmq-console-ng-1.0.1.jar --server.port=9877 --rocketmq.config.namesrvAddr=127.0.0.1:9876

5)访问 <http://127.0.0.1:9877>



2.3 快速入门

2.3.1 三种消息发送方式

RocketMQ 支持 3 种消息发送方式：

1、同步消息 (sync message)

producer向 broker 发送消息，执行 API 时同步等待，直到broker 服务器返回发送结果。

2、异步消息 (async message)

producer向 broker 发送消息时指定消息发送成功及发送异常的回调方法，调用 API 后立即返回，producer发送消息线程不阻塞，消息发送成功或失败的回调任务在一个新的线程中执行。

3、单向消息 (oneway message)

producer向 broker 发送消息，执行 API 时直接返回，不等待broker 服务器的结果。

2.3.2 消息结构

RocketMQ的消息包括基础属性和扩展属性两部分：

1、基础属性

1) topic：主题相当于消息的一级分类，具有相同topic的消息将发送至该topic下的消息队列中，比方说一个电商系统可以分为商品消息、订单消息、物流消息等，就可以在broker中创建商品主题、订单主题等，所有商品的消息发送至该主题下的消息队列中。

2) 消息体：即消息的内容，可以是字符串、对象等类型（可序列化）。消息的最大长度是4M。

3) 消息 Flag：消息的一个标记，RocketMQ不处理，留给业务系统使用。

2、扩展属性

1) tag：相当于消息的二级分类，用于消费消息时进行过滤，可为空。

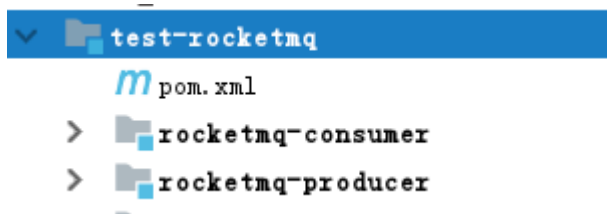
2) keys: Message 索引键，在运维中可以根据这些 key 快速检索到消息，可为空。3) waitStoreMsgOK：消息发送时是否等消息存储完成后再返回。

Message 的基础属性主要包括消息所属主题 topic，消息 Flag(RocketMQ 不做处理)、扩展属性、消息体。

2.3.3 生产者工程

2.3.3.1 创建生产者工程

工程结构如下：



1) 创建test-rocketmq

创建一个test-rocketmq的测试工程专门用于rocketmq的功能测试。

test-rocketmq父工程的pom.xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```



```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <parent>
            <artifactId>shanjupay</artifactId>
            <groupId>com.shanjupay</groupId>
            <version>1.0-SNAPSHOT</version>
        </parent>
        <modelVersion>4.0.0</modelVersion>

        <artifactId>test-rocketmq</artifactId>
        <packaging>pom</packaging>
        <dependencies>
            <dependency>
                <groupId>org.apache.rocketmq</groupId>
                <artifactId>rocketmq-spring-boot-starter</artifactId>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
        </dependencies>

    </project>
```

2) 创建rocketmq-producer生产者工程

rocketmq-producer的pom.xml如下

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>test-rocketmq</artifactId>
        <groupId>com.shanjupay</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>rocketmq-producer</artifactId>

</project>
```

3) 新建rocketmq-producer工程的application.yml文件



```
server:
  port: 8181 #服务端口
  servlet:
    context-path: /rocketmq-producer

spring:
  application:
    name: rocketmq-producer #指定服务名
rocketmq:
  nameServer: 127.0.0.1:9876
  producer:
    group: demo-producer-group
```

4) 新建启动类

```
@SpringBootApplication
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class, args);
    }

}
```

2.3.3.2 发送同步消息

```
package com.shanjupay.test.rocketmq.message;

import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 * rocketmq发送消息类
 * @author Administrator
 * @version 1.0
 */
@Component
public class ProducerSimple {

    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    /**
     * 发送同步消息
     * @param topic
     * @param msg
     */
}
```

```
public void sendSyncMsg(String topic, String msg){  
    rocketMQTemplate.syncSend(topic,msg);  
}  
  
}
```

2.3.3.3 测试

1、在test下编写测试类，发送同步消息。

```
package com.shanjupay.test.rocketmq.message;  
  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.junit4.SpringRunner;  
  
/**  
 * @author Administrator  
 * @version 1.0  
 **/  
@RunWith(SpringRunner.class)  
@SpringBootTest  
public class ProducerSimpleTest {  
  
    @Autowired  
    private ProducerSimple producerSimple;  
  
    //测试发送同步消息  
    @Test  
    public void testSendSyncMsg(){  
        this.producerSimple.sendSyncMsg("my-topic", "第一条同步消息");  
        System.out.println("end...");  
    }  
  
}
```

2、启动NameServer、Broker、管理端

3、执行testSendSyncMsg方法

4、观察控制台和管理端

控制台出现end... 表示消息发送成功。

进入管理端，查询消息。

RocketMQ控制台 运维 驾驶舱 集群 主题 消费者 生产者 消息 消息轨迹

TOPIC MESSAGE KEY MESSAGE ID

Only Return 2000 Messages

主题: my-topic 开始: 2020-01-10 10:06 结束: 2020-01-10 12:06 搜索

Message ID	Tag	Key	StoreTime	Operation
C0A8016447D018B4AAC230BB98340000			2020-01-10 11:06:40	MESSAGE DETAIL

Message ID: C0A8016447D018B4AAC230BB98340000

Topic: my-topic

Tag:

Key:

Storetime: 2020-01-10

Message body: 第一条同步消息

2.3.4 消费者工程

2.3.4.1 创建消费者工程

创建消息消费者工程，pom.xml如下

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>test-rocketmq</artifactId>
    <groupId>com.shanjupay</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>rocketmq-consumer</artifactId>

</project>
```

2、启动类

```
@SpringBootApplication
public class ConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

}
```

3、配置文件application.yml

```
server:
  port: 8182 #服务端口
  servlet:
    context-path: /rocketmq-consumer

spring:
  application:
    name: rocketmq-consumer #指定服务名
rocketmq:
  nameServer: 127.0.0.1:9876
```

2.3.4.2 消费消息

编写消费消息监听类：

```
package com.shanjupay.test.rocketmq.message;

import org.apache.rocketmq.spring.annotation.ConsumeMode;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * 消费消息监听类
 * @author Administrator
 * @version 1.0
 */
@Component
@RocketMQMessageListener(topic = "my-topic", consumerGroup = "demo-consumer-group")
public class ConsumerSimple implements RocketMQListener<String> {

    //接手到消息调用此方法
    @Override
    public void onMessage(String s) {
        System.out.println(s);
    }
}
```

```
}  
}
```

监听消息队列 需要指定：

topic：监听的主题

consumerGroup：消费组，相同消费组的消费者共同消费该主题的消息，它们组成一个集群。

2.3.4.3 测试

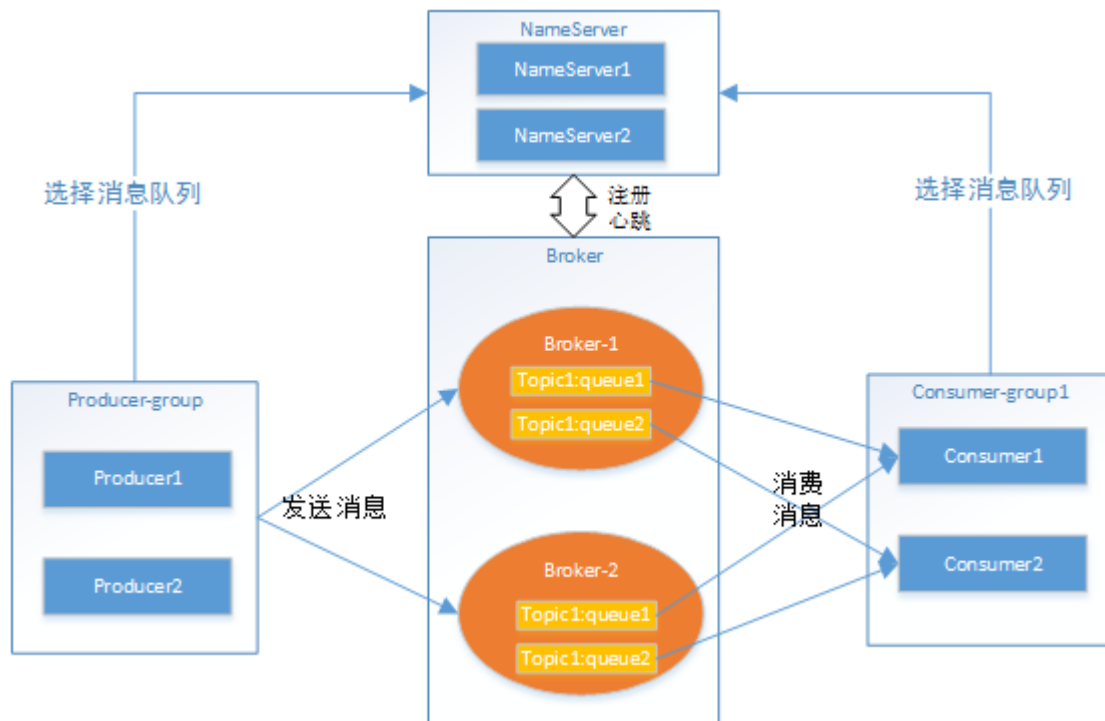
1、启动消费者工程

启动消费者工程，观察控制台输出“第一条同步消息”消息内容，这说明从消息队列已经读取到消息。

2、保证消费者工程已启动，再次发送消息，观察控制台是否输出“第一条同步消息”消息内容，输出则说明接收消息成功。

2.4 消息发送过程

通过快速入门对消息的发送和接收有一个粗略的认识，下边分析具体的消息发送过程，如下图：



消息发送流程如下：

1、Producer从NameServer中获取主题路由信息

Broker将自己的状态上报给NameServer，NameServer中存储了每个Broker及主题、消息队列的信息。

Producer根据 topic从NameServer查询所有消息队列，查询到的结果例如：

```
[
  {"brokerName": "Broker-1", "queueId": 0},
  {"brokerName": "Broker-1", "queueId": 1},
  {"brokerName": "Broker-2", "queueId": 0},
  {"brokerName": "Broker-2", "queueId": 1}
]
```

Producer按选择算法从以上队列中选择一个进行消息发送，如果发送消息失败则在下次选择的时候会规避掉失败的broker。

2、构建消息，发送消息

发送消息前进行校验，比如消息的内容长度不能为0、消息最大长度、消息必要的属性是否具备等（topic、消息体，生产组等）。

如果该topic下还没有队列则自动创建，默认一个topic下自动创建4个写队列，4个读队列。

为什么要多个队列？

1) 高可用

当某个队列不可用时其它队列顶上。

2) 提高并发

发送消息是选择队列进行发送，提高发送消息的并发能力。

消息消费时每个消费者可以监听多个队列，提高消费消息的并发能力。

生产组有什么用？

在事务消息中broker需要回查producer，同一个生产组的producer组成一个集群，提高并发能力。

3、监听队列，消费消息

一个消费组可以包括多个消费者，一个消费组可以订阅多个主题。

一个队列同时只允许一个消费者消费，一个消费者可以消费多个队列中的消息。

消费组有两种消费模式：

1) 集群模式

一个消费组内的消费者组成一个集群，主题下的一条消息只能被一个消费者消费。

2) 广播模式

主题下的一条消息能被消费组下的所有消费者消费。

消费者和broker之间通过推模式和拉模式接收消息，推模式即broker推送给消费者，拉模式是消费者主动从broker查询消息。

2.5 三种消息发送方式

RocketMQ 支持 3 种消息发送方式，即同步消息（sync message）、异步消息（async message）、单向消息（oneway message）。

2.5.1 同步消息

参见快速入门的测试程序。

2.5.2 异步消息

producer向 broker 发送消息时指定消息发送成功及发送异常的回调方法，调用 API 后立即返回，producer发送消息线程不阻塞，消息发送成功或失败的回调任务在一个新的线程中执行。

在ProducerSimple中编写发送异步消息的方法

```
/**
 * 发送异步消息
 * @param topic
 * @param msg
 */
public void sendAsyncMsg(String topic, String msg){
    rocketMQTemplate.asyncSend(topic,msg,new SendCallback() {
        @Override
        public void onSuccess(SendResult sendResult) {
            //成功回调
            System.out.println(sendResult.getSendStatus());
        }

        @Override
        public void onException(Throwable e) {
            //异常回调
            System.out.println(e.getMessage());
        }
    });
}
```

测试：

```
@Test
public void testSendAsyncMsg() throws InterruptedException {
    this.producerSimple.sendAsyncMsg("my-topic", "第一条异步消息");
    System.out.println("end...");
    //异步消息，为跟踪回调线程这里加入延迟
    Thread.sleep(3000);
}
```

2.5.3 单向消息

producer向 broker 发送消息，执行 API 时直接返回，不等待broker 服务器的结果。

```
/**
 * 发送单向消息
 * @param topic
 * @param msg
 */
public void sendOneWayMsg(String topic, String msg){
    this.rocketMQTemplate.sendOneWay(topic,msg);
}
```

测试：

略。

2.6 自定义消息格式

前边我们发送的消息内容格式都是字符串，在生产开发中消息内容格式是复杂的，本小节介绍如何对消息格式进行自定义。

JSON是互联网开发中非常常用的数据格式，它具有格式标准，扩展方便的特点，将消息的格式使用JSON进行定义可以提高消息内容的扩展性，RocketMQ支持传递JSON数据格式。

在生产端和消费端定义模型类：

```
package com.shanjupay.test.rocketmq.model;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

import java.io.Serializable;
import java.util.Date;

@Data
@NoArgsConstructor
@ToString
public class OrderExt implements Serializable {

    private String id;

    private Date createTime;

    private Long money;

    private String title;

}
```

生产端：

```
/**
 * 消息内容为json格式
 */
public void sendMsgByJson(String topic, OrderExt orderExt){
    //发送同步消息，消息内容将orderExt转为json
    this.rocketMQTemplate.convertAndSend(topic,orderExt);
    System.out.printf("send msg : %s",orderExt);
}
```

消费端：

```
@Component
@RocketMQMessageListener(topic = "my-topic",consumerGroup = "demo-consumer-group")
public class ConsumerSimple implements RocketMQListener<String> {

    //接手到消息调用此方法
    @Override
    public void onMessage(String s) {
        //如果是json数据，可以将json转为对象
        // OrderExt orderExt = JSON.parseObject(s, OrderExt.class);
        System.out.println(s);
    }
}
```

上例实现了RocketMQ传输JSON消息的过程，消费端在接收到JSON手动将JSON转成对象，也可以自动转换成对象，代码如下：

定义新的监听类，RocketMQListener泛型指定要转换的对象类型。

```
@Component
@RocketMQMessageListener(topic = "my-topic-obj",consumerGroup = "demo-consumer-group-obj")
public class ConsumerSimpleObj implements RocketMQListener<OrderExt> {

    //接手到消息调用此方法
    @Override
    public void onMessage(OrderExt orderExt) {
        System.out.println(orderExt);
    }
}
```

2.7 延迟消息

2.7.1 延迟消息介绍

延迟消息也叫做定时消息，比如在电商项目的交易系统中，当用户下单之后超过一段时间之后仍然没有支付，此时就需要将该订单关闭。要实现该功能的话，可以在用户创建订单时就发送一条包含订单内容的延迟消息，该消息在一段时间之后投递给消息消费者，当消息消费者接收到该消息后，判断该订单的支付状态，如果处于未支付状态，则将该订单关闭。

RocketMQ的延迟消息实现非常简单，只需要发送消息前设置延迟的时间，延迟时间存在十八个等级（1s/5s/10s/30s/1m/2m/3m/4m/5m/6m/7m/8m/9m/10m/20m/30m/1h/2h），调用setDelayTimeLevel()设置与时间相对应的延迟级别即可。

2.7.2 同步消息延迟

生产端：

```
/**
 * 发送延迟消息
 * 消息内容为json格式
 */
public void sendMsgByJsonDelay(String topic, OrderExt orderExt) throws JsonProcessingException,
    InterruptedException, RemotingException, MQClientException, MQBrokerException {
    //发送同步消息，消息内容将orderExt转为json
    Message<OrderExt> message = MessageBuilder.withPayload(orderExt).build();
    //指定发送超时时间（毫秒）和延迟等级
    this.rocketMQTemplate.syncSend(topic, message, 1000, 3);

    System.out.printf("send msg : %s", orderExt);
}
```

消费端：

同自定义消息格式章节。

测试：

```
//测试发送同步消息
@Test
public void testSendMsgByJsonDelay() throws JsonProcessingException, InterruptedException,
    RemotingException, MQClientException, MQBrokerException {
    OrderExt orderExt = new OrderExt();
    orderExt.setId(UUID.randomUUID().toString());
    orderExt.setCreateTime(new Date());
    orderExt.setMoney(168L);
    orderExt.setTitle("测试订单");
    this.producerSimple.sendMsgByJsonDelay("my-topic-obj", orderExt);
    System.out.println("end...");
}
```

2.7.3 异步消息延迟（自学）

生产端：

```
/**
 * 发送异步延迟消息
```



```
* 消息内容为json格式
*/
public void sendAsyncMsgByJsonDelay(String topic, OrderExt orderExt) throws
JsonProcessingException, InterruptedException, RemotingException, MQClientException,
MQBrokerException {
    //消息内容将orderExt转为json
    String json = this.rocketMQTemplate.getObjectMapper().writeValueAsString(orderExt);
    org.apache.rocketmq.common.message.Message message = new
    org.apache.rocketmq.common.message.Message(topic, json.getBytes(Charset.forName("utf-8")));
    //设置延迟等级
    message.setDelayTimeLevel(3);
    //发送异步消息
    this.rocketMQTemplate.getProducer().send(message, new SendCallback() {
        @Override
        public void onSuccess(SendResult sendResult) {
            System.out.println(sendResult);
        }

        @Override
        public void onException(Throwable throwable) {
            System.out.println(throwable.getMessage());
        }
    });

    System.out.printf("send msg : %s", orderExt);
}
```

消费端：

同自定义消息格式章节。

测试

```
//测试发送异步消息
@Test
public void testSendAsyncMsgByJsonDelay() throws JsonProcessingException, InterruptedException,
RemotingException, MQClientException, MQBrokerException {
    OrderExt orderExt = new OrderExt();
    orderExt.setId(UUID.randomUUID().toString());
    orderExt.setCreateTime(new Date());
    orderExt.setMoney(168L);
    orderExt.setTitle("测试订单");
    this.producerSimple.sendAsyncMsgByJsonDelay("my-topic-obj", orderExt);
    System.out.println("end...");
    Thread.sleep(20000);
}
```

2.8 消费重试

2.8.1 什么是消费重试

当消息发送到Broker成功，在被消费者消费时如果消费者没有正常消费，此时消息会重试消费。消费重试存在两种场景：

- 1) 消息没有被消费者接收，比如消费者与broker存在网络异常。此种情况消息会一直被消费重试。
- 2) 当消息已经被消费者成功接收，但是在进行消息处理时出现异常，消费端无法向Broker返回成功，这种情况下RocketMQ会不断重试。本小节重点讨论第二个场景。

针对第二种消费重试的场景，broker是怎么知道重试呢？

消费者在消费消息成功会向broker返回成功状态，否则会不断进行消费重试。

2.8.2 处理策略

当消息在消费时出现异常，此时消息被不断重试消费。RocketMQ会一直重试消费吗？

答案是不会！

消息会按照延迟消息的延迟时间等级（1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h）从第3级开始重试，每试一次如果还不成功则延迟等级加1。

比如：一条消息消费失败，等待10s(第3级)进行重试，如果还没有被成功消费则延迟等级加1，即按第4级别延迟等待，等30s继续进行重试，如此进行下去，直到重试16次。

当重试了16次还未被成功消费将会投递到死信队列，到达死信队列的消息将不再被消费。

实际生产中的处理策略是什么呢？

实际生产中不会让消息重试这么多次，通常在重试一定的次数后将消息写入数据库，由另外单独的程序或人工去处理。

项目使用的Spring整合RocketMQ的方式，消费者实现RocketMQListener的onMessage方法，在此方法中实现处理策略的示例代码如下：

```
public class ConsumerSimple implements RocketMQListener<MessageExt> {

    @Override
    public void onMessage(MessageExt messageExt) {
        //取出当前重试次数
        int reconsumeTimes = messageExt.getReconsumeTimes();
        //当大于一定的次数后将消息写入数据库，由单独的程序或人工去处理
        if(reconsumeTimes >=2){
            //将消息写入数据库，之后正常返回
            return ;
        }
        throw new RuntimeException(String.format("第%s次处理失败..",reconsumeTimes));
    }
}
```

