消息队列设计

一. 消息队列背景知识

曾经我们学习过 **阻塞队列 (BlockingQueue)** , 我们说, 阻塞队列最大的用途, 就是用来实现 **生产者消费者模型.**

生产者消费者模型, 存在诸多好处, 是后端开发的常用编程方式.

- 解耦合
- 削峰填谷

在实际的后端开发中, 尤其是分布式系统里, 跨主机之间使用生产者消费者模型, 也是非常普遍的需求.

因此, 我们通常会把阻塞队列, 封装成一个独立的服务器程序, 并且赋予其更丰富的功能.

这样的程序我们就称为 **消息队列 (Message Queue, MQ)**

市面上成熟的消息队列非常多.

- RabbitMQ
- Kafka
- RocketMQ
- ActiveMQ
- ……

其中, RabbitMQ 是一个非常知名, 功能强大, 广泛使用的消息队列.

咱们就仿照 RabbitMQ, 模拟实现一个简单的消息队列.

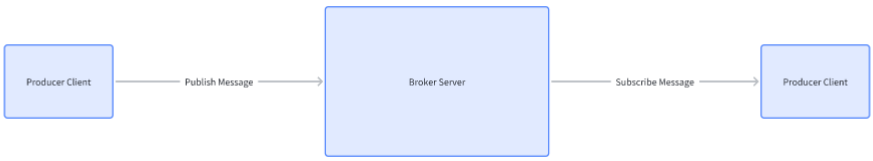
二. 需求分析

核心概念

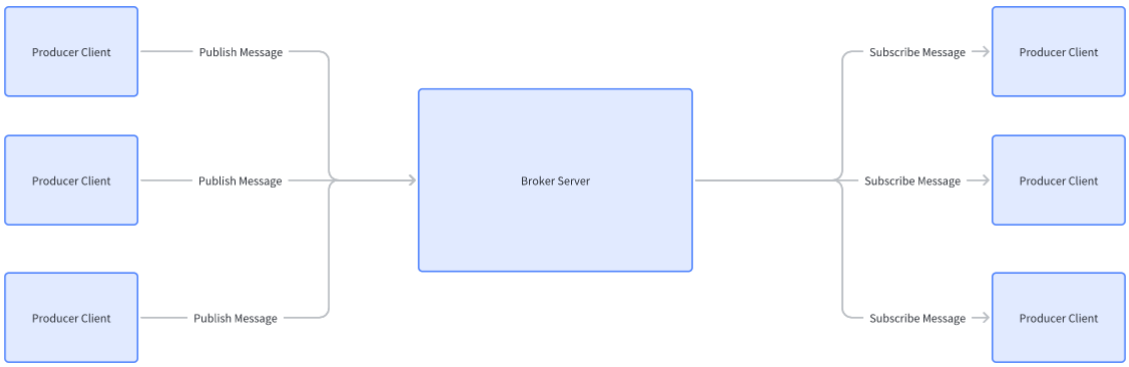
- 生产者 (Producer)
- 消费者 (Consumer)
- 中间人 (Broker)
- 发布 (Publish)

- 订阅 (Subscribe)

一个生产者, 一个消费者



N 个生产者, N 个消费者

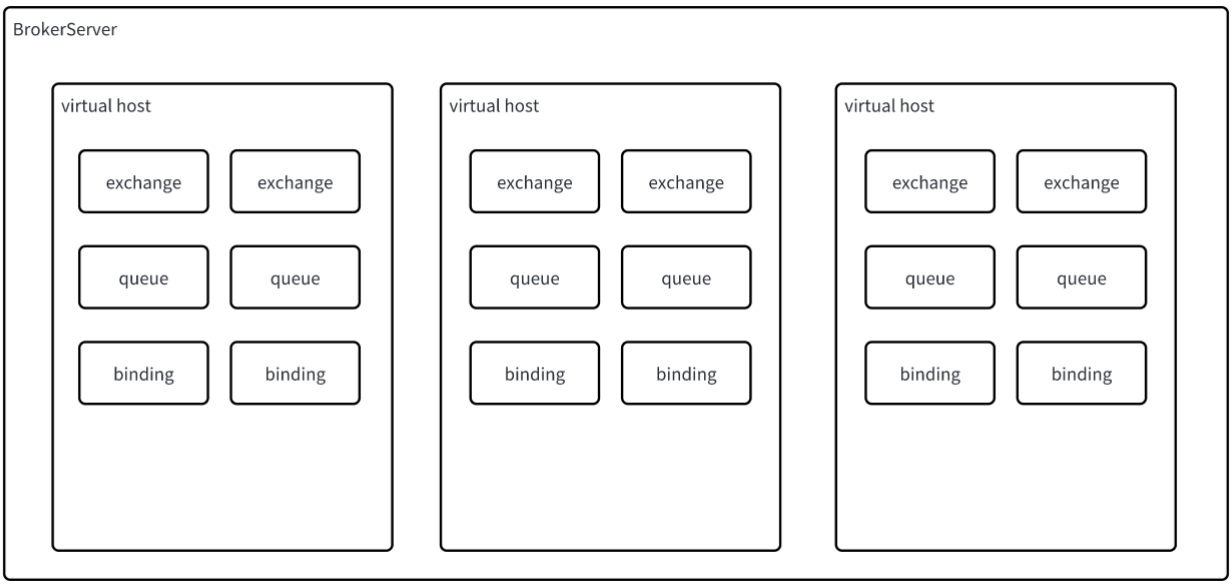


其中, Broker 是最核心的部分. 负责消息的存储和转发.

在 Broker 中, 又存在以下概念.

- 虚拟机 (VirtualHost): 类似于 MySQL 的 "database", 是一个逻辑上的集合. 一个 BrokerServer 上可以存在多个 VirtualHost.
- 交换机 (Exchange): 生产者把消息先发送到 Broker 的 Exchange 上. 再根据不同的规则, 把消息转发给不同的 Queue.
- 队列 (Queue): 真正用来存储消息的部分. 每个消费者决定自己从哪个 Queue 上读取消息.
- 绑定 (Binding): Exchange 和 Queue 之间的关联关系. Exchange 和 Queue 可以理解成 "多对多" 关系. 使用一个关联表就可以把这两个概念联系起来.
- 消息 (Message): 传递的内容.

> 所谓的 Exchange 和 Queue 可以理解成 "多对多" 关系, 和数据库中的 "多对多" 一样. 意思是:
>
> 一个 Exchange 可以绑定多个 Queue (可以向多个 Queue 中转发消息).
>
> 一个 Queue 也可以被多个 Exchange 绑定 (一个 Queue 中的消息可以来自于多个 Exchange).



这些概念, 既需要在内存中存储, 也需要在硬盘上存储.

- 内存存储: 方便使用.
- 硬盘存储: 重启数据不丢失.

核心 API

对于 Broker 来说, 要实现以下核心 API. 通过这些 API 来实现消息队列的基本功能.

1. 创建队列 (queueDeclare)
2. 销毁队列 (queueDelete)
3. 创建交换机 (exchangeDeclare)
4. 销毁交换机 (exchangeDelete)
5. 创建绑定 (queueBind)
6. 解除绑定 (queueUnbind)
7. 发布消息 (basicPublish)
8. 订阅消息 (basicConsume)
9. 确认消息 (basicAck)

另一方面, Producer 和 Consumer 则通过网络的方式, 远程调用这些 API, 实现 **生产者消费者模型**.

> 关于 VirtualHost
>
> 对于 RabbitMQ 来说, VirtualHost 也是可以随意创建删除的.

> 此处咱们暂时不做这部分功能(实现起来也比较简单, 咱们的代码中会完成部分和虚拟主机相关的结构设计. 大家可以自行完成管理逻辑).

交换机类型 (Exchange Type)

对于 RabbitMQ 来说, 主要支持四种交换机类型.

- Direct
- Fanout
- Topic
- Header

其中 Header 这种方式比较复杂, 比较少见. 常用的是前三种交换机类型. 咱们此处也主要实现这三种.

- Direct: 生产者发送消息时, 直接指定被该交换机绑定的队列名.
- Fanout: 生产者发送的消息会被复制到该交换机的所有队列中.
- Topic: 绑定队列到交换机上时, 指定一个字符串为 bindingKey. 发送消息指定一个字符串为 routingKey. 当 routingKey 和 bindingKey 满足一定的匹配条件的时候, 则把消息投递到指定队列.

> 这三种操作就像给 qq 群发红包.
>
> - Direct 是发一个专属红包, 只有指定的人能领.
> - Fanout 是使用了魔法, 发一个 10 块钱红包, 群里的每个人都能领 10 块钱.
> - Topic 是发一个画图红包, 发 10 块钱红包, 同时出个题, 得画的像的人, 才能领. 也是每个领到的人都能领 10 块钱.

持久化

Exchange, Queue, Binding, Message 都有持久化需求.

当程序重启 / 主机重启, 保证上述内容不丢失.

网络通信

生产者和消费者都是客户端程序, broker 则是作为服务器. 通过网络进行通信.

在网络通信的过程中, 客户端部分要提供对应的 api, 来实现对服务器的操作.

1. 创建 Connection

2. 关闭 Connection
3. 创建 Channel
4. 关闭 Channel
5. 创建队列 (queueDeclare)
6. 销毁队列 (queueDelete)
7. 创建交换机 (exchangeDeclare)
8. 销毁交换机 (exchangeDelete)
9. 创建绑定 (queueBind)
10. 解除绑定 (queueUnbind)
11. 发布消息 (basicPublish)
12. 订阅消息 (basicConsume)
13. 确认消息 (basicAck)

可以看到, 在 broker 的基础上, 客户端还要增加 Connection 操作和 Channel 操作.

Connection 对应一个 TCP 连接.

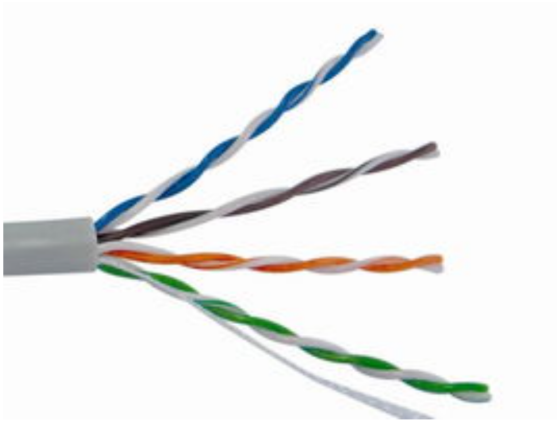
Channel 则是 Connection 中的逻辑通道.

一个 Connection 中可以包含多个 Channel.

Channel 和 Channel 之间的数据是独立的. 不会相互干扰.

这样的设定主要是为了能够更好的复用 TCP 连接, 达到长连接的效果, 避免频繁的创建关闭 TCP 连接.

> Connection 可以理解成一根网线. Channel 则是网线里具体的线缆.



消息应答

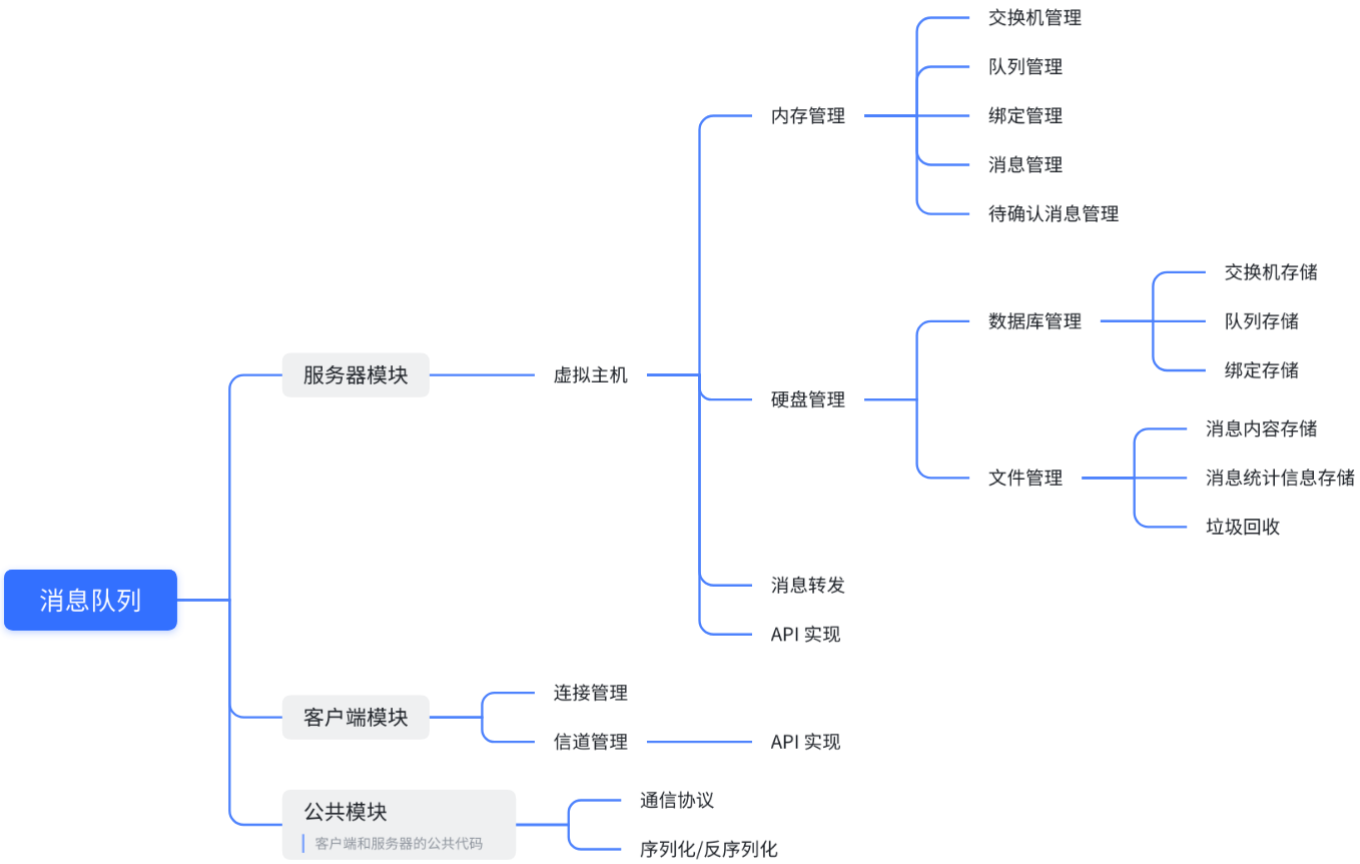
被消费的消息, 需要进行应答.

应答模式分成两种.

- 自动应答: 消费者只要消费了消息, 就算应答完毕了. Broker 直接删除这个消息.
- 手动应答: 消费者手动调用应答接口, Broker 收到应答请求之后, 才真正删除这个消息.

> 手动应答的目的, 是为了保证消息确实被消费者处理成功了. 在一些对于数据可靠性要求高的场景, 比较常见.

三. 模块划分



可以看到, 像 交换机, 队列, 绑定, 消息, 这几个核心概念在内存和硬盘中都是存储了的.

其中内存为主, 是用来实现消息转发的关键; 硬盘为辅, 主要是保证服务器重启之后, 之前的信息都可以正常保持.

四. 项目创建

创建 SpringBoot 项目.

使用 SpringBoot 2 系列版本, Java 8.

依赖引入 Spring Web 和 MyBatis.

五. 创建核心类

创建包 `mqserver.core`

创建 Exchange

```java
public class Exchange {
    private String name;
    private ExchangeType type = ExchangeType.DIRECT;
    private boolean durable = false;
    private boolean autoDelete = false;
    private Map<String, Object> arguments = new HashMap<>();

    // 省略 getter setter
}
```

```java
public enum ExchangeType {
    DIRECT(0),
    FANOUT(1),
    TOPIC(2);

    private final int type;

    private ExchangeType(int type) {
        this.type = type;
    }

    public int getType() {
        return this.type;
    }
}
```

- `name` : 交换机的名字. 相当于交换机的身份标识.
- `type` : 交换机的类型. 三种取值, DIRECT, FANOUT, TOPIC.
- `durable` : 交换机是否要持久化存储. true 为持久化, false 不持久化.
- `autoDelete` : 使用完毕后是否自动删除. 预留字段, 暂时未使用.
- `arguments` : 交换机的其他参数属性. 预留字段, 暂时未使用.

> RabbitMQ 中的交换机, 支持 `autoDelete` 和 `arguments`, 咱们此处为了简单, 暂时没有实现对应功能, 只是预留了字段, 同学们可以尝试自己完成.

创建 MSGQueue

```java
public class MSGQueue {
    private String name;
    private boolean durable;
    private boolean exclusive;
    private boolean autoDelete;
    private Map<String, Object> arguments = new HashMap<>();

    // 省略 getter setter
}
```

> 类名叫做 MSGQueue, 而不是 Queue, 是为了防止和标准库中的 Queue 混淆.

- `name` : 队列的名字. 相当于队列的身份标识.
- `durable` : 交换机是否要持久化存储. true 为持久化, false 不持久化.
- `exclusive` : 独占(排他), 队列只能被一个消费者使用.
- `autoDelete` : 使用完毕后是否自动删除. 预留字段, 暂时未使用.
- `arguments` : 交换机的其他参数属性. 预留字段, 暂时未使用.

创建 Binding

```java
public class Binding {
    private String exchangeName;
    private String queueName;
    private String bindingKey;

    // 省略 getter setter
}
```

- `exchangeName` 交换机名字
- `queueName` 队列名字
- `bindingKey` 只在交换机类型为 `TOPIC` 时才有效. 用于和消息中的 `routingKey` 进行匹配.

创建 Message

```java
public class Message implements Serializable {
    private BasicProperties basicProperties = new BasicProperties();
    private byte[] body;

    // 消息在文件中对应的 offset 的范围, [offsetBeg, offsetEnd)
    // 从这个范围取出的 byte[] 正好可以反序列化成一个 Message 对象.
    // offsetBeg 前面的 4 个字节是消息的长度
    private transient long offsetBeg = 0;
    private transient long offsetEnd = 0;

    // 消息在文件中是否有效. 0x0 表示无效, 0x1 表示有效
    private byte isValid = 0x1;

    // 创建新的消息, 同时给该消息分配一个新的 messageId
    // routingKey 以参数的为准. 会覆盖掉 basicProperties 中的 routingKey
    public static Message createMessageWithId(String routingKey, BasicProperties
        Message message = new Message();
        if (basicProperties != null) {
            message.basicProperties = basicProperties;
        }
        message.basicProperties.setMessageId("M-" + UUID.randomUUID().toString()
        message.basicProperties.setRoutingKey(routingKey);
        message.body = body;
        return message;
    }

    // 省略 getter setter
}
```

```java
public class BasicProperties implements Serializable {
    // 消息的唯一 id. 使用 uuid 表示.
    private String messageId;
    private String routingKey;
    // 1 表示消息非持久化. 2 表示消息持久化
    private int deliveryMode = 1;

    // 省略 getter setter
}
```

- `Message` 需要实现 `Serializable` 接口. 后续需要把 Message 写入文件以及进行网络传输.
- `basicProperties` 是消息的属性信息. `body` 是消息体.
- `offsetBeg` 和 `offsetEnd` 表示消息在消息文件中所在的起始位置和结束位置. 这一块具体的设计后面再详细介绍. 使用 `transient` 关键字避免属性被序列化.
- `isValid` 用来表示消息在文件中是否有效. 这一块具体的设计后面再详细介绍.
- `createMessageWithId` 相当于一个工厂方法, 用来创建一个 Message 实例. messageId 通过 UUID 的方式生成.

六. 数据库设计

对于 Exchange, MSGQueue, Binding, 我们使用数据库进行持久化保存.

此处我们使用的数据库是 SQLite, 是一个更轻量的数据库.

SQLite 只是一个动态库(当然, 官方也提供了可执行程序 exe), 我们在 Java 中直接引入 SQLite 依赖, 即可直接使用, 不必安装其他的软件.

配置 sqlite

引入 pom.xml 依赖

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.41.0.1</version>
</dependency>
```

配置数据源 application.yml

```
spring:
  datasource:
    url: jdbc:sqlite:./data/meta.db
    username:
    password:
    driver-class-name: org.sqlite.JDBC

mybatis:
  mapper-locations: classpath:mapper/**Mapper.xml
```

> Username 和 password 空着即可.

此处我们约定, 把数据库文件放到 `./data/meta.db` 中.

SQLite 只是把数据单纯的存储到一个文件中. 非常简单方便.

实现创建表

```java
@Mapper
public interface MetaMapper {
    void createUserTable();
    void createExchangeTable();
    void createQueueTable();
    void createBindingTable();
}
```

> 本身 MyBatis 针对 MySQL / Oracle 支持执行多个 SQL 语句的, 但是针对 SQLite 是不支持的, 只能写成多个方法.

```xml
<update id="createExchangeTable">
    create table if not exists exchange (
        name varchar(50) primary key,
        type int,                       -- 0 表示 direct, 1 表示 fanout, 2 表示 t
        durable boolean,                -- false 表示不持久化, true 表示持久化.
        autoDelete boolean,             -- false 表示不自动删除, true 表示自动删除.
        arguments varchar(1024)         -- 创建交换机指定的参数
    );
</update>

<update id="createQueueTable">
    create table if not exists queue (
        name varchar(50) primary key,
        durable boolean,                -- false 表示不持久化, true 表示持久化.
        autoDelete boolean,             -- false 表示不自动删除, true 表示自动删除.
        arguments varchar(1024)         -- 创建交换机指定的参数
    );
</update>

<update id="createBindingTable">
    create table if not exists binding (
        exchangeName varchar(50),
        queueName varchar(50),
        bindingKey varchar(256)
```

```
    );
</update>
```

实现数据库基本操作

给 `mapper.MetaMapper` 中添加

```
void insertExchange(Exchange exchange);
void deleteExchange(String exchangeName);
void insertQueue(MSGQueue msgQueue);
void deleteQueue(String queueName);
void insertBinding(Binding binding);
void deleteBinding(Binding binding);
```

给 `MetaMapper` 中添加

```
<insert id="insertExchange" parameterType="com.example.java_message_queue.mqserv
    insert into exchange values(#{name}, #{type}, #{durable}, #{autoDelete}, #{a
</insert>

<delete id="deleteExchange" parameterType="java.lang.String">
    delete from exchange where name = #{exchangeName};
</delete>

<insert id="insertQueue" parameterType="com.example.java_message_queue.mqserver.
    insert into queue values(#{name}, #{durable}, #{autoDelete}, #{arguments});
</insert>

<delete id="deleteQueue" parameterType="java.lang.String">
    delete from queue where name = #{queueName};
</delete>

<insert id="insertBinding" parameterType="com.example.java_message_queue.mqserve
    insert into binding values(#{exchangeName}, #{queueName}, #{bindingKey});
</insert>

<delete id="deleteBinding" parameterType="com.example.java_message_queue.mqserve
    delete from binding where exchangeName = #{exchangeName} and queueName = #{q
</delete>
```

实现 DataBaseManager

`mqserver.datacenter.DataBaseManager`

1) 创建 DataBaseManager 类

通过这个类来封装针对数据库的操作.

```
public class DataBaseManager {
    // 由于 DataBaseManager 不是一个 Bean
    // 需要手动来获取实例
    private MetaMapper metaMapper;

    public void init() {
        this.metaMapper = JavaMessageQueueApplication.ac.getBean(MetaMapper.clas

        // 构造数据库
        if (!checkDBExists()) {
            // 1. 读取 sql 文件中的内容, 并创建表
            createTable();
            // 2. 插入默认数据
            createDefaultData();
            System.out.println("[DataBaseManager] 数据库初始化完成!");
        } else {
            System.out.println("[DataBaseManager] 数据库已经存在!");
        }
    }
}
```

> 如果数据库文件存在, 则不必建库建表了.

针对 JavaMessageQueueApplication, 需要新增一个 ac 属性. 并初始化

```
@SpringBootApplication
public class JavaMessageQueueApplication {
   public static ConfigurableApplicationContext ac;

   public static void main(String[] args) throws IOException {
      ac = SpringApplication.run(JavaMessageQueueApplication.class);
   }
```

```
}
```

2) 实现 checkDBExists

```
private boolean checkDBExists() {
    File file = new File("./meta.db");
    if (file.exists()) {
        return true;
    }
    return false;
}
```

3) 实现 createTable

```
// 创建数据表
private void createTable() {
    metaMapper.createExchangeTable();
    metaMapper.createQueueTable();
    metaMapper.createBindingTable();
    System.out.println("[DataBaseManager] 创建表完成!");
}
```

4) 实现 createDefaultData

```
// 创建表中的默认数据
private void createDefaultData() {
    // 构造默认交换机
    Exchange exchange = new Exchange();
    exchange.setName("");
    exchange.setType(ExchangeType.DIRECT);
    exchange.setDurable(true);
    exchange.setAutoDelete(false);
    metaMapper.insertExchange(exchange);
    System.out.println("[DataBaseManager] 创建初始数据完成!");
}
```

> 默认数据主要是创建一个默认的交换机. 这个默认交换机没有名字, 并且是直接交换机.

5) 封装其他数据库操作

```
public void insertExchange(Exchange exchange) {
    metaMapper.insertExchange(exchange);
}

public void deleteExchange(String exchangeName) {
    metaMapper.deleteExchange(exchangeName);
}

public List<Exchange> selectAllExchanges() {
    return metaMapper.selectAllExchanges();
}

public void insertQueue(MSGQueue queue) {
    metaMapper.insertQueue(queue);
}

public void deleteQueue(String queueName) {
    metaMapper.deleteQueue(queueName);
}

public List<MSGQueue> selectAllQueues() {
    return metaMapper.selectAllQueues();
}

public void insertBinding(Binding binding) {
    metaMapper.insertBinding(binding);
}

public void deleteBinding(Binding binding) {
    metaMapper.deleteBinding(binding);
}

public List<Binding> selectAllBindings() {
    return metaMapper.selectAllBindings();
}
```

测试 DataBaseManager

使用 Spring 自带的单元测试, 针对上述代码进行测试验证.

在 test 目录中, 创建 DataBaseManagerTests

1) 准备工作

```
@SpringBootTest
public class DataBaseManagerTests {
    private static DataBaseManager dataBaseManager = new DataBaseManager();

    @BeforeAll
    public static void setupAll() throws IOException {
        // 初始情况下, 先统一清除数据库
        dataBaseManager.deleteDB();
    }

    @BeforeEach
    public void setUp() throws IOException {
        // 每次运行一个用例, 都重置数据库. 防止用例之间的数据相互干扰.
        // 需要初始化 ac 对象
        JavaMessageQueueApplication.ac = SpringApplication.run(JavaMessageQueueA
        // 再初始化数据库
        dataBaseManager.init();
    }

    @AfterEach
    public void tearDown() throws IOException {
        // 需要关闭 ac 对象
        JavaMessageQueueApplication.ac.close();
        // 然后再删除数据库
        dataBaseManager.deleteDB();
    }
}
```

- `@SpringBootTest` 注解表示该类是一个测试类.
- `@BeforeAll` 在所有测试执行之前执行. 此处先删除之前的数据库, 避免干扰.
- `@BeforeEach` 每个测试用例之前执行. 一般用来做准备工作. 此处进行数据库初始化, 以及针对 Spring 服务的初始化.

- `@AfterEach` 每个测试用例之后执行. 一般用来做收尾工作. 此处需要先关闭 Spring 服务, 再删除数据库.

> 由于 Spring 服务启动的时候, 会和数据库建立连接(通过 MyBatis). 因此需要先关闭服务, 才能删除数据库, 否则会删除失败(Spring 服务会持有数据库文件的访问权限).

2) 编写测试用例

- `@Test` 注解表示一个测试用例.
- `Assertions` 是断言, 用来断定执行结果.
- 每个用例执行之前, 都会自动调用到 setUp, 每次用例执行结束之后, 都会自动调用 tearDown
- 要确保每个用例的执行都是 "clean" 的, 也就是该用例不会被上个用例干扰到.

```
@Test
public void testInitTable() throws IOException {
    List<Exchange> exchangeList = dataBaseManager.selectAllExchanges();
    List<MSGQueue> queueList = dataBaseManager.selectAllQueues();
    List<Binding> bindingList = dataBaseManager.selectAllBindings();

    Assertions.assertEquals(1, exchangeList.size());
    Assertions.assertEquals("", exchangeList.get(0).getName());
    Assertions.assertEquals(ExchangeType.DIRECT, exchangeList.get(0).getType());
    Assertions.assertEquals(0, queueList.size());
    Assertions.assertEquals(0, bindingList.size());
}

private Exchange createTestExchange(String exchangeName) {
    Exchange exchange = new Exchange();
    exchange.setName(exchangeName);
    exchange.setType(ExchangeType.FANOUT);
    exchange.setAutoDelete(true);
    exchange.setDurable(true);
    HashMap<String, Object> arguments = new HashMap<>();
    arguments.put("aaa", "111");
    arguments.put("bbb", "222");
    exchange.setArguments(arguments);
    return exchange;
}

@Test
public void testInsertExchange() {
    Exchange exchange = createTestExchange("test");
    dataBaseManager.insertExchange(exchange);
```

```
31      List<Exchange> exchangeList = dataBaseManager.selectAllExchanges();
32      Assertions.assertEquals(2, exchangeList.size());
33      Assertions.assertEquals("test", exchangeList.get(1).getName());
34      Assertions.assertEquals(ExchangeType.FANOUT, exchangeList.get(1).getType());
35      Assertions.assertEquals(true, exchangeList.get(1).isAutoDelete());
36      Assertions.assertEquals(true, exchangeList.get(1).isDurable());
37      Assertions.assertEquals("111", exchangeList.get(1).getArgument("aaa"));
38      Assertions.assertEquals("222", exchangeList.get(1).getArgument("bbb"));
39  }
40
41  @Test
42  public void testDeleteExchange() {
43      Exchange exchange = createTestExchange("test");
44      dataBaseManager.insertExchange(exchange);
45      List<Exchange> exchangeList = dataBaseManager.selectAllExchanges();
46      Assertions.assertEquals(2, exchangeList.size());
47      Assertions.assertEquals("test", exchangeList.get(1).getName());
48
49      dataBaseManager.deleteExchange("test");
50      exchangeList = dataBaseManager.selectAllExchanges();
51      Assertions.assertEquals(1, exchangeList.size());
52      Assertions.assertEquals("", exchangeList.get(0).getName());
53  }
54
55  private MSGQueue createTestQueue(String queueName) {
56      MSGQueue queue = new MSGQueue();
57      queue.setName(queueName);
58      queue.setDurable(true);
59      queue.setAutoDelete(true);
60      queue.setExclusive(true);
61      HashMap<String, Object> hashMap = new HashMap<>();
62      hashMap.put("aaa", "111");
63      hashMap.put("bbb", "222");
64      queue.setArguments(hashMap);
65      return queue;
66  }
67
68  @Test
69  public void testInsertQueue() {
70      MSGQueue queue = createTestQueue("test");
71      dataBaseManager.insertQueue(queue);
72      List<MSGQueue> queueList = dataBaseManager.selectAllQueues();
73      Assertions.assertEquals(1, queueList.size());
74      Assertions.assertEquals("test", queueList.get(0).getName());
75      Assertions.assertEquals(true, queueList.get(0).isDurable());
76      Assertions.assertEquals(true, queueList.get(0).isAutoDelete());
77      Assertions.assertEquals(true, queueList.get(0).isExclusive());
```

```
78      Assertions.assertEquals("111", queueList.get(0).getArgument("aaa"));
79      Assertions.assertEquals("222", queueList.get(0).getArgument("bbb"));
80  }
81
82  @Test
83  public void testDeleteQueue() {
84      MSGQueue queue = createTestQueue("test");
85      dataBaseManager.insertQueue(queue);
86      List<MSGQueue> queueList = dataBaseManager.selectAllQueues();
87      Assertions.assertEquals(1, queueList.size());
88      Assertions.assertEquals("test", queueList.get(0).getName());
89
90      dataBaseManager.deleteQueue("test");
91      queueList = dataBaseManager.selectAllQueues();
92      Assertions.assertEquals(0, queueList.size());
93  }
94
95  @Test
96  public void testInsertBinding() {
97      Binding binding = new Binding();
98      binding.setQueueName("testQueue");
99      binding.setExchangeName("testExchange");
100     binding.setBindingKey("testBindingKey");
101     dataBaseManager.insertBinding(binding);
102
103     List<Binding> bindingList = dataBaseManager.selectAllBindings();
104     Assertions.assertEquals(1, bindingList.size());
105     Assertions.assertEquals("testQueue", bindingList.get(0).getQueueName());
106     Assertions.assertEquals("testExchange", bindingList.get(0).getExchangeName()
107     Assertions.assertEquals("testBindingKey", bindingList.get(0).getBindingKey()
108 }
109
110 @Test
111 public void testDeleteBinding() {
112     Binding binding = new Binding();
113     binding.setQueueName("testQueue");
114     binding.setExchangeName("testExchange");
115     binding.setBindingKey("testBindingKey");
116     dataBaseManager.insertBinding(binding);
117
118     List<Binding> bindingList = dataBaseManager.selectAllBindings();
119     Assertions.assertEquals(1, bindingList.size());
120
121     dataBaseManager.deleteBinding(binding);
122
123     bindingList = dataBaseManager.selectAllBindings();
124     Assertions.assertEquals(0, bindingList.size());
```

```
125 }
```

七. 消息存储设计

设计思路

消息需要在硬盘上存储. 但是并不直接放到数据库中, 而是直接使用文件存储.

原因如下:

1. 对于消息的操作并不需要复杂的 增删改查 .
2. 对于文件的操作效率比数据库会高很多.

> 主流 MQ 的实现(包括 RabbitMQ), 都是把消息存储在文件中, 而不是数据库中.

我们给每个队列分配一个目录. 目录的名字为 data + 队列名. 形如 `./data/testQueue`

该目录中包含两个固定名字的文件.

- `queue_data.txt` 消息数据文件, 用来保存消息内容.
- `queue_stat.txt` 消息统计文件, 用来保存消息统计信息.

`queue_data.txt` 文件格式:

使用二进制方式存储.

每个消息分成两个部分:

- 前四个字节, 表示 Message 对象的长度(字节数)
- 后面若干字节, 表示 Message 内容.
- 消息和消息之间首尾相连.

每个 Message 基于 Java 标准库的 ObjectInputStream / ObjectOutputStream 序列化.

> Message 对象中的 offsetBeg 和 offsetEnd 正是用来描述每个消息体所在的位置.

`queue_stat.txt` 文件格式:

使用文本方式存储.

文件中只包含一行, 里面包含两列(都是整数), 使用 \t 分割.

第一列表示当前总的消息数目. 第二列表示有效消息数目.

形如:

```
2000\t1500
```

创建 MessageFileManager 类

创建 `mqserver.database.MessageFileManager`

```
public class MessageFileManager {
    // 表示消息的统计信息
    static public class Stat {
        public int totalCount;
        public int validCount;
    }

    public void init() {
        // 当前这里不需要做任何工作.
    }

    // 队列目录
    private String getQueueDir(String queueName) {
        return "./data/" + queueName;
    }

    // 队列数据文件
    // 这个文件来存储队列的真实数据
    private String getQueueDataPath(String queueName) {
        return getQueueDir(queueName) + "/queue_data.txt";
    }

    // 队列统计文件
    // 这个文件用来存储队列中的统计信息.
    // 包含一行, 两个列使用 \t 分割, 分别是总数据, 和无效数据.
    private String getQueueStatPath(String queueName) {
        return getQueueDir(queueName) + "/queue_stat.txt";
    }
}
```

- 内部包含一个 Stat 类, 用来表示消息统计文件的内容.
- getQueueDir, getQueueDataPath, getQueueStatPath 用来表示这几个文件所在位置.

实现统计文件读写

> 这是后续操作的一项准备工作.

```
// 从统计文件中读取结果
private Stat readStat(String queueName) {
    Stat stat = new Stat();
    try (InputStream inputStream = new FileInputStream(getQueueStatPath(queueNam
        Scanner scanner = new Scanner(inputStream);
        stat.totalCount = scanner.nextInt();
        stat.validCount = scanner.nextInt();
        return stat;
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

// 向统计文件中写入结果
private void writeStat(String queueName, Stat stat) {
    try (OutputStream outputStream = new FileOutputStream(getQueueStatPath(queue
        PrintWriter printWriter = new PrintWriter(outputStream);
        printWriter.write(stat.totalCount + "\t" + stat.validCount);
        printWriter.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

直接使用 Scanner 和 PrintWriter 进行读写即可.

实现创建队列目录

每个队列都有自己的目录和配套的文件. 通过下列方法把目录和文件先准备好.

```
public void createQueueFiles(String queueName) throws IOException {
    // 1. 创建目录指定队列的目录
```

```
    File baseDir = new File(getQueueDir(queueName));
    if (!baseDir.exists()) {
        boolean ok = baseDir.mkdirs();
        if (!ok) {
            throw new IOException("创建目录失败! baseDir=" + baseDir.getAbsoluteP
        }
    }
    // 2. 创建队列数据文件
    File queueDataFile = new File(getQueueDataPath(queueName));
    if (!queueDataFile.exists()) {
        boolean ok = queueDataFile.createNewFile();
        if (!ok) {
            throw new IOException("创建文件失败! queueDataFile=" + queueDataFile.
        }
    }
    // 3. 创建队列统计文件
    File queueStatFile = new File(getQueueStatPath(queueName));
    if (!queueStatFile.exists()) {
        boolean ok = queueStatFile.createNewFile();
        if (!ok) {
            throw new IOException("创建文件失败! queueStatFile=" + queueStatFile.
        }
    }
    // 4. 给队列统计文件写入初始数据
    Stat stat = new Stat();
    stat.totalCount = 0;
    stat.validCount = 0;
    writeStat(queueName, stat);
}
```

把上述约定的文件都创建出来, 并对消息统计文件进行初始化.

> 初始化 `0\t0` 这样的初始值.

实现删除队列目录

如果队列需要删除, 则队列对应的目录/文件也需要删除.

```
public void destroyQueueFiles(String queueName) throws IOException {
    // 1. 先删除目录中的文件
    File queueDataFile = new File(getQueueDataPath(queueName));
    boolean ok1 = queueDataFile.delete();
    File queueStatFile = new File(getQueueStatPath(queueName));
    boolean ok2 = queueStatFile.delete();
```

```
    // 2. 再删除目录. delete 要求必须是空目录才能删除.
    File baseDir = new File(getQueueDir(queueName));
    boolean ok3 = baseDir.delete();
    if (!ok1 || !ok2 || !ok3) {
        throw new IOException("删除队列目录失败! baseDir=" + baseDir.getAbsolutePa
    }
}
```

注意: File 类的 delete 方法只能删除空目录. 因此需要先把内部的文件先删除掉.

检查队列文件是否存在

判定该队列的消息文件和统计文件是否存在. 一旦出现缺失, 则不能进行后续工作.

```
private boolean checkFilesExists(String queueName) {
    File queueData = new File(getQueueDataPath(queueName));
    if (!queueData.exists()) {
        return false;
    }
    File queueStat = new File(getQueueStatPath(queueName));
    if (!queueStat.exists()) {
        return false;
    }
    return true;
}
```

实现消息对象序列化/反序列化

Message 对象需要转成二进制写入文件. 并且也需要把文件中的二进制读出来解析成 Message 对象. 此处针对这里的逻辑进行封装.

创建 `common.BinaryTool`

```
public class BinaryTool {
    public static Object fromBytes(byte[] data) throws IOException, ClassNotFoun
        Object object = null;
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(dat
        try (ObjectInputStream objectInputStream = new ObjectInputStream(byteArr
            object = objectInputStream.readObject();
        }
```

```
        return object;
    }

    public static byte[] toBytes(Object object) throws IOException {
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(
        try (ObjectOutputStream objectOutputStream = new ObjectOutputStream(byte
            objectOutputStream.writeObject(object);
        }
        return byteArrayOutputStream.toByteArray();
    }
}
```

- 使用 ByteArrayInputStream / ByteArrayOutputStream 针对 byte[] 进行封装, 方便后续操作. (这两个流对象是纯内存的, 不需要进行 close).
- 使用 ObjectInputStream / ObjectOutputStream 进行序列化 / 反序列化操作. 通过内部的 readObject / writeObject 即可完成对应操作.
- 此处涉及到的序列化对象, 需要实现 Serializable 接口. 这一点咱们的 Message 对象已经实现过了.

> 对于 serialVersionUID , 此处咱们暂时不需要. 大家可以自行了解 serialVersionUID 的用途

实现写入消息文件

```
public void sendMessage(MSGQueue queue, Message message) throws MqException, IOE
    if (!checkFilesExists(queue.getName())) {
        throw new MqException("[MessageFileManager] 队列匹配的文件不存在! queueNam
    }
    // 1. 先把 message 转成二进制
    byte[] messageBinary = BinaryTool.toBytes(message);
    // 此处的锁对象以队列为维度. 不同队列之间不涉及锁冲突.
    synchronized (queue) {
        // 2. 先获取到文件总长度
        File queueDataFile = new File(getQueueDataPath(queue.getName()));
        message.setOffsetBeg(queueDataFile.length() + 4);
        message.setOffsetEnd(queueDataFile.length() + 4 + messageBinary.length);
        // 3. 写入消息数据文件
        try (OutputStream outputStream = new FileOutputStream(queueDataFile, tru
            DataOutputStream dataOutputStream = new DataOutputStream(outputStrea
            // 先写入消息长度
            dataOutputStream.writeInt(messageBinary.length);
            // 再写入消息本体
            dataOutputStream.write(messageBinary);
        }
```

```
21          // 4. 写入消息统计文件
22          Stat stat = readStat(queue.getName());
23          stat.totalCount += 1;
24          stat.validCount += 1;
25          writeStat(queue.getName(), stat);
26      }
27  }
```

- 考虑线程安全, 按照队列维度进行加锁.
- 使用 DataOutputStream 进行二进制写操作. 比原生 OutputStream 要方便.
- 需要记录 Message 对象在文件中的偏移量. 后续的删除操作依赖这个偏移量定位到消息. offsetBeg 是原有文件大小的基础上, 再 + 4. 4 个字节是存放消息大小的空间. (参考上面的图).
- 写完消息, 要同时更新统计信息.

创建 `common.MqException` , 作为自定义异常类. 后续业务上出现问题, 都统一抛出这个异常.

> 实践中创建多个异常类, 分别表示不同异常种类是更好的做法. 此处我们只是偷懒了.

```
1  public class MqException extends Exception {
2      public MqException(String message) {
3          super(message);
4      }
5  }
```

实现删除消息

此处的删除只是 "逻辑删除", 即把 Message 类中的 isValid 字段设置为 0.

> 这样删除速度比较快. 实际的彻底删除, 则通过我们自己实现的 GC 来解决.

```
1  // 把文件上的对应消息给删除掉. (标记成无效)
2  public void deleteMessage(MSGQueue queue, Message message) throws IOException, C
3      synchronized (queue) {
4          try (RandomAccessFile randomAccessFile = new RandomAccessFile(getQueueDa
5              // 1. 先从文件中读取出 Message 的数据
6              byte[] bufferSrc = new byte[(int) (message.getOffsetEnd() - message.
7              randomAccessFile.seek(message.getOffsetBeg());
8              randomAccessFile.read(bufferSrc);
9              // 2. 转成 Message 对象
```

```
            Message diskMessage = (Message) BinaryTool.fromBytes(bufferSrc);
            // 3. 设置成无效.
            diskMessage.setIsValid((byte)0x0);
            // 4. 重新写入文件
            byte[] bufferDest = BinaryTool.toBytes(diskMessage);
            randomAccessFile.seek(message.getOffsetBeg());
            randomAccessFile.write(bufferDest);
        }

        // 更新统计文件
        Stat stat = readStat(queue.getName());
        if (stat.validCount > 0) {
            stat.validCount -= 1;
        }
        writeStat(queue.getName(), stat);
    }
}
```

- 使用 RandomAccessFile 来随机访问到文件的内容.
- 根据 Message 中的 offsetBeg 和 offsetEnd 定位到消息在文件中的位置. 通过 randomAccessFile.seek 操作文件指针偏移过去. 再读取.
- 读出的结果解析成 Message 对象, 修改 isValid 字段, 再重新写回文件. 注意写的时候要重新设定文件指针的位置. 文件指针会随着上述的读操作产生改变.
- 最后, 要记得更新统计文件, 把合法消息 - 1.

实现消息加载

把消息内容从文件加载到内存中. 这个功能在服务器重启, 和垃圾回收的时候都很关键.

```
// 从消息数据文件中读取出所有消息
public LinkedList<Message> loadAllMessageFromQueue(String queueName) throws MqEx
    // 记录当前读到的数据在文件的 offset
    long currentOffset = 0;
    LinkedList<Message> messages = new LinkedList<>();
    try (InputStream inputStream = new FileInputStream(getQueueDataPath(queueNam
        DataInputStream dataInputStream = new DataInputStream(inputStream);
        while (true) {
            // 读到文件末尾, 会触发 EOFException
            int messageSize = dataInputStream.readInt();
            byte[] buffer = new byte[messageSize];
            int actualSize = dataInputStream.read(buffer);
            if (messageSize != actualSize) {
```

```
                throw new MqException("[MessageFileManager] 文件格式错误! queueNar
            }
            Message message = (Message) BinaryTool.fromBytes(buffer);
            if (message.getIsValid() != 0x1) {
                // 被删除的无效数据, 直接跳过. 不要忘记更新 currentOffset
                currentOffset += 4 + messageSize;
                continue;
            }
            // 计算该 message 的 offset
            message.setOffsetBeg(currentOffset + 4);
            message.setOffsetEnd(currentOffset + 4 + messageSize);
            // 每个消息, 开头 4 个字节保存的是消息的长度. 接下来 [offsetBeg, offsetEn
            currentOffset += 4 + messageSize;
            messages.add(message);
        }
    } catch (EOFException e) {
        // 数据读取完毕, 循环正常退出!
        System.out.println("[MessageFileManager] 恢复 Message 数据完成!");
    }
    return messages;
}
```

- 使用 DataInputStream 读取数据. 先读 4 个字节为消息的长度, 然后再按照这个长度来读取实际消息内容.
- 读取完毕之后, 转换成 Message 对象.
- 同时计算出该对象的 offsetBeg 和 offsetEnd.
- 最终把结果整理成链表, 返回出去.
- 注意, 对于 DataInputStream 来说, 如果读取到 EOF, 会抛出一个 EOFException , 而不是返回特定值. 因此需要注意上述循环的结束条件.

实现垃圾回收(GC)

上述删除操作, 只是把消息在文件上标记成了无效. 并没有腾出硬盘空间. 最终文件大小可能会越积越多. 因此需要定期的进行批量清除.

此处使用类似于复制算法. 当总消息数超过 2000, 并且有效消息数目少于 50% 的时候, 就触发 GC.

GC 的时候会把所有有效消息加载出来, 写入到一个新的消息文件中, 使用新文件, 代替旧文件即可.

```
// 检查是否要针对文件进行 GC 操作
public boolean checkGC(String queueName) {
    Stat stat = readStat(queueName);
```

```
    if (stat.totalCount >= 2000 && (double)stat.validCount / (double) stat.total
        return true;
    }
    return false;
}

private String getQueueDataNewPath(String queueName) {
    return getQueueDir(queueName) + "/queue_data_new.txt";
}

// 真正执行 GC 操作
// 使用复制算法.
// 先创建一个新的文件, 名字为 "queue_data_new.txt"
// 然后加载出旧的文件的所有有效消息内容
// 把这些内容写入到新的文件中.
// 删除旧文件, 对新文件重命名.
public void gc(MSGQueue queue) throws MqException, IOException, ClassNotFoundExc
    synchronized (queue) {
        long gcBeg = System.currentTimeMillis();
        // 1. 创建一个新的文件, 名字为 "queue_data_new.txt"
        File queueDataNew = new File(getQueueDataNewPath(queue.getName()));
        if (queueDataNew.exists()) {
            throw new MqException("[MessageFileManager] gc 时发现队列新数据文件已经
        }
        boolean ok = queueDataNew.createNewFile();
        if (!ok) {
            throw new IOException("创建文件失败! queueDataNew=" + queueDataNew.ge
        }
        // 2. 遍历旧文件, 读取出每个对象 (只保留有效消息)
        List<Message> messageList = loadAllMessageFromQueue(queue.getName());
        // 3. 把有效消息写入到新的文件中.
        try (OutputStream outputStream = new FileOutputStream(queueDataNew)) {
            DataOutputStream dataOutputStream = new DataOutputStream(outputStrea
            for (Message message : messageList) {
                byte[] buffer = BinaryTool.toBytes(message);
                dataOutputStream.writeInt(buffer.length);
                dataOutputStream.write(buffer);
            }
        }
        // 4. 删除 queue_data.txt, 把 queue_data_new.txt 重命名为 queue_data
        File queueDataOld = new File(getQueueDataPath(queue.getName()));
        ok = queueDataOld.delete();
        if (!ok) {
            throw new IOException("删除文件失败! queueDataOld=" + queueDataOld.ge
        }
        ok = queueDataNew.renameTo(queueDataOld);
        if (!ok) {
```

```
            throw new IOException("文件重命名失败! queueDataOld=" + queueDataOld.g
                    ", queueDataNew=" + queueDataNew.getAbsolutePath());
        }
        // 5. 更新统计文件
        Stat stat = readStat(queue.getName());
        stat.validCount = messageList.size();
        stat.totalCount = messageList.size();
        writeStat(queue.getName(), stat);

        long gcEnd = System.currentTimeMillis();
        System.out.println("[MessageFileManager] gc 执行完毕! queueName=" + queue
    }
}
```

> 如果文件很大, 消息非常多, 可能比较低效, 这种就需要把文件做拆分和合并了.
>
> Rabbitmq 本体是这样实现的. 但是咱们此处为了实现简单, 就不做这个了.

测试 MessageFileManager

创建 `MessageFileManagerTests` 编写测试用例代码.

- 创建两个队列, 用来辅助测试.
- 使用 `ReflectionTestUtils.invokeMethod` 来调用私有方法.

```
@SpringBootTest
public class MessageFileManagerTests {
    private String queueName1 = "testQueue1";
    private String queueName2 = "testQueue2";
    private MessageFileManager messageFileManager = new MessageFileManager();


    @BeforeEach
    public void setUp() throws IOException {
        messageFileManager.createQueueFiles(queueName1);
        messageFileManager.createQueueFiles(queueName2);
    }

    @AfterEach
    public void tearDown() throws IOException {
        messageFileManager.destroyQueueFiles(queueName1);
        messageFileManager.destroyQueueFiles(queueName2);
    }
}
```

```java
@Test
public void testCreateFile() {
    File queueDataFile1 = new File("./data/" + queueName1 + "/queue_data.txt");
    Assertions.assertEquals(true, queueDataFile1.isFile());
    File queueStatFile1 = new File("./data/" + queueName1 + "/queue_stat.txt");
    Assertions.assertEquals(true, queueStatFile1.isFile());
    Assertions.assertTrue(queueStatFile1.length() > 0);
    File queueDataFile2 = new File("./data/" + queueName2 + "/queue_data.txt");
    Assertions.assertEquals(true, queueDataFile2.isFile());
    File queueStatFile2 = new File("./data/" + queueName2 + "/queue_stat.txt");
    Assertions.assertEquals(true, queueStatFile2.isFile());
    Assertions.assertTrue(queueStatFile2.length() > 0);
}

@Test
public void testReadWriteStat() {
    MessageFileManager.Stat stat = new MessageFileManager.Stat();
    stat.totalCount = 100;
    stat.validCount = 50;
    // 通过 Spring 提供的反射工具类, 调用私有方法.
    ReflectionTestUtils.invokeMethod(messageFileManager, "writeStat", queueName1
    MessageFileManager.Stat newStat = ReflectionTestUtils.invokeMethod(messageFi
    Assertions.assertEquals(100, newStat.totalCount);
    Assertions.assertEquals(50, newStat.validCount);
}

private MSGQueue createTestQueue(String queueName) {
    MSGQueue queue = new MSGQueue();
    queue.setName(queueName);
    queue.setDurable(true);
    queue.setAutoDelete(true);
    queue.setExclusive(true);
    HashMap<String, Object> hashMap = new HashMap<>();
    hashMap.put("aaa", "111");
    hashMap.put("bbb", "222");
    queue.setArguments(hashMap);
    return queue;
}

private Message createTestMessage(String content) {
    Message message = new Message();
    message.setMessageId("M-" + UUID.randomUUID().toString());
    message.setRoutingKey("testRoutingKey");
    message.setDeliveryMode(2);
    message.setBody(content.getBytes());
```

```
46      return message;
47  }
48
49  @Test
50  public void testSendMessage() throws IOException, MqException, ClassNotFoundExce
51      Message message = createTestMessage("testMessage");
52      MSGQueue queue = createTestQueue(queueName1);
53      messageFileManager.sendMessage(queue, message);
54
55      // 检查 stat 文件
56      MessageFileManager.Stat newStat = ReflectionTestUtils.invokeMethod(messageFi
57      Assertions.assertEquals(1, newStat.totalCount);
58      Assertions.assertEquals(1, newStat.validCount);
59
60      // 读文件内容
61      List<Message> messageList = messageFileManager.loadAllMessageFromQueue(queue
62      Assertions.assertEquals(1, messageList.size());
63      Message curMessage = messageList.get(0);
64      Assertions.assertEquals(message.getMessageId(), curMessage.getMessageId());
65      Assertions.assertEquals(message.getRoutingKey(), curMessage.getRoutingKey())
66      Assertions.assertEquals(message.getDeliveryMode(), curMessage.getDeliveryMod
67      Assertions.assertArrayEquals(message.getBody(), curMessage.getBody());
68  }
69
70  @Test
71  public void testLoadAllMessageFromQueue() throws IOException, MqException, Class
72      MSGQueue queue = createTestQueue(queueName1);
73      List<Message> expectedMessages = new ArrayList<>();
74      for (int i = 0; i < 100; i++) {
75          Message message = createTestMessage("testMessage");
76          messageFileManager.sendMessage(queue, message);
77          expectedMessages.add(message);
78      }
79      List<Message> actualMessages = messageFileManager.loadAllMessageFromQueue(qu
80      Assertions.assertEquals(100, actualMessages.size());
81      for (int i = 0; i < 100; i++) {
82          Message expectedMessage = actualMessages.get(i);
83          Message actualMessage = actualMessages.get(i);
84          System.out.println("[" + i + "] " + actualMessage);
85          Assertions.assertEquals(expectedMessage.getMessageId(), actualMessage.ge
86          Assertions.assertEquals(expectedMessage.getRoutingKey(), actualMessage.g
87          Assertions.assertEquals(expectedMessage.getDeliveryMode(), actualMessage
88          Assertions.assertArrayEquals(expectedMessage.getBody(), actualMessage.ge
89          Assertions.assertEquals(0x1, actualMessage.getIsValid());
90      }
91  }
92
```

```java
@Test
public void testDeleteMessage() throws IOException, MqException, ClassNotFoundEx
    MSGQueue queue = createTestQueue(queueName1);
    List<Message> expectedMessages = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        Message message = createTestMessage("testMessage");
        messageFileManager.sendMessage(queue, message);
        expectedMessages.add(message);
    }
    System.out.println("expected:" + expectedMessages);

    messageFileManager.deleteMessage(queue, expectedMessages.get(0));
    messageFileManager.deleteMessage(queue, expectedMessages.get(1));
    messageFileManager.deleteMessage(queue, expectedMessages.get(2));

    // 读出来，这个方法只能加载有效数据.
    List<Message> actualMessages = messageFileManager.loadAllMessageFromQueue(qu
    System.out.println("actual: " + actualMessages);
    Assertions.assertEquals(7, actualMessages.size());
    for (int i = 0; i < actualMessages.size(); i++) {
        Assertions.assertEquals(expectedMessages.get(i + 3).getMessageId(), actu
    }
}

@Test
public void testGc() throws IOException, MqException, ClassNotFoundException {
    MSGQueue queue = createTestQueue(queueName1);
    List<Message> expectedMessages = new ArrayList<>();
    // 创建 100 个元素
    for (int i = 0; i < 100; i++) {
        Message message = createTestMessage("testMessage");
        messageFileManager.sendMessage(queue, message);
        expectedMessages.add(message);
    }
    // 删除 偶数 下标的元素
    for (int i = 0; i < 100; i += 2) {
        messageFileManager.deleteMessage(queue, expectedMessages.get(i));
    }
    // 获取旧文件大小
    File oldFile = new File("./data/" + queueName1 + "/queue_data.txt");
    long oldLength = oldFile.length();

    // 调用 gc
    messageFileManager.gc(queue);
    // 重新读文件
    List<Message> actualMessages = messageFileManager.loadAllMessageFromQueue(qu
    Assertions.assertEquals(50, actualMessages.size());
```

```
    for (int i = 0; i < 50; i++) {
        // 注意这里的下标换算
        Assertions.assertEquals(expectedMessages.get(2 * i + 1).getMessageId(),
    }
    // 获取新文件大小
    File newFile = new File("./data/" + queueName1 + "/queue_data.txt");
    long newLength = newFile.length();
    System.out.println("oldLength=" + oldLength);
    System.out.println("newLength=" + newLength);
    Assertions.assertTrue(oldLength > newLength);
}
```

八. 整合数据库和文件

上述代码中, 使用数据库存储了 Exchange, Queue, Binding, 使用文本文件存储了 Message.

接下来我们把两个部分整合起来, 统一进行管理.

创建 DiskDataCenter

使用 DiskDataCenter 来综合管理数据库和文本文件的内容.

DiskDataCenter 会持有 DataBaseManager 和 MessageFileManager 对象.

```
// 管理硬盘上的数据.
// 分成两个部分:
// 1. 数据库管理元信息
// 2. 文件管理消息内容
public class DiskDataCenter {
    private String virtualHostName;

    // 管理数据库中的元数据
    private DataBaseManager dataBaseManager = new DataBaseManager();
    // 管理文件中的消息数据
    private MessageFileManager messageFileManager = new MessageFileManager();

    public void init(String virtualHostName) {
        this.virtualHostName = virtualHostName;

        initDir();
        dataBaseManager.init();
        messageFileManager.init();
    }
```

```
}
```

实现 initDir

```
// 初始化目录结构
// virtualHostName 为 default-VirtualHost
// 则存放数据的目录名为: ./data/default-VirtualHost/
private void initDir() {
    File baseDir = new File("./data/" + virtualHostName);
    if (!baseDir.exists()) {
        boolean ok = baseDir.mkdirs();
        if (ok) {
            System.out.println("[DiskDataCenter] 初始化数据目录完成!");
        } else {
            System.out.println("[DiskDataCenter] 初始化数据目录失败!");
        }
    } else {
        System.out.println("[DiskDataCenter] 数据目录已经存在!");
    }
}
```

封装 Exchange 方法

```
public void insertExchange(Exchange exchange) {
    dataBaseManager.insertExchange(exchange);
}

public void deleteExchange(String exchangeName) {
    dataBaseManager.deleteExchange(exchangeName);
}

public List<Exchange> selectAllExchanges() {
    return dataBaseManager.selectAllExchanges();
}
```

封装 Queue 方法

```
public void insertQueue(MSGQueue queue) throws IOException {
    dataBaseManager.insertQueue(queue);
    messageFileManager.createQueueFiles(queue.getName());
}

public void deleteQueue(String queueName) throws IOException {
    dataBaseManager.deleteQueue(queueName);
    messageFileManager.destroyQueueFiles(queueName);
}

public List<MSGQueue> selectAllQueues() {
    return dataBaseManager.selectAllQueues();
}
```

- 创建/删除队列的时候同时创建/删除队列目录.

封装 Binding 方法

```
public void insertBinding(Binding binding) {
    dataBaseManager.insertBinding(binding);
}

public void deleteBinding(Binding binding) {
    dataBaseManager.deleteBinding(binding);
}

public List<Binding> selectAllBindings() {
    return dataBaseManager.selectAllBindings();
}
```

封装 Message 方法

```
public void sendMessage(MSGQueue queue, Message message) throws MqException, IOE
    messageFileManager.sendMessage(queue, message);
}

public void deleteMessage(MSGQueue queue, Message message) throws MqException, I
    messageFileManager.deleteMessage(queue, message);

```

```
    // 判定是否要 GC
    if (messageFileManager.checkGC(queue.getName())) {
        messageFileManager.gc(queue);
    }
}

public LinkedList<Message> loadAllMessageFromQueue(String queueName) throws MqEx
    return messageFileManager.loadAllMessageFromQueue(queueName);
}
```

- 在 deleteMessage 的时候判定是否进行 GC.

小结

通过上述封装, 把数据库和硬盘文件两部分合并成一个整体. 上层代码在调用的时候则不再关心该数据是存储在哪个部分的.

这个类的整体实现并不复杂, 关键逻辑在之前都已经准备好了.

该类我们就不单独进行单元测试了. 同学们可以自行完成.

九. 内存数据结构设计

硬盘上存储数据, 只是为了实现 "持久化" 这样的效果. 但是实际的消息存储/转发, 还是主要靠内存的结构.

对于 MQ 来说, 内存部分是更关键的, 内存速度更快, 可以达成更高的并发.

创建 MemoryDataCenter

创建 `mqserver.datacenter.MemoryDataCenter`

```
// 管理所有的内存数据.
public class MemoryDataCenter {
    // key 是 exchangeName
    private ConcurrentHashMap<String, Exchange> exchangeMap = new ConcurrentHash
    // key 是 queueName
    private ConcurrentHashMap<String, MSGQueue> queueMap = new ConcurrentHashMap
    // 第一个 key 是 exchangeName, 第二个 key 是 queueName
    private ConcurrentHashMap<String, HashMap<String, Binding>> bindingsMap = ne
    // 保存所有消息, key 是 messageId
    private ConcurrentHashMap<String, Message> messageMap = new ConcurrentHashMa
```

```
    // key 是 queueName
    private ConcurrentHashMap<String, LinkedList<Message>> queueMessageMap = new
    // 用来存放待确认的消息
    // key1 是 queueName, key2 是 messageId.
    // 这个结构不需要有对应的硬盘数据. 换句话说, 如果某个消息消费了, 但是没有 ack, 这个时
    // 就把刚才的消息当做从来没消费过.
    private ConcurrentHashMap<String, HashMap<String, Message>> queueMessageWait

    public void init() {
    }
}
```

- 使用四个哈希表, 管理 Exchange, Queue, Binding, Message.
- 使用一个哈希表 + 链表管理 队列 -> 消息 之间的关系.
- 使用一个哈希表 + 哈希表管理所有的未被确认的消息.

> 为了保证消息被正确消费了, 会使用两种方式进行确认. 自动 ACK 和 手动 ACK.
>
> 其中自动 ACK 是指当消息被消费之后, 就会立即被销毁释放.
>
> 其中手动 ACK 是指当消息被消费之后, 由消费者主动调用一个 basicAck 方法, 进行主动确认. 服务器收到这个确认之后, 才能真正销毁消息.
>
> 此处的 "未确认消息" 就是指在手动 ACK 模式下, 该消息还没有被调用 basicAck. 此时消息不能删除, 但是要和其他未消费的消息区分开. 于是另搞了个结构.
>
> 当后续 basicAck 到了, 就可以删除消息了.

封装 Exchange 方法

```
public void insertExchange(Exchange exchange) {
    exchangeMap.put(exchange.getName(), exchange);
}

public Exchange getExchange(String exchangeName) {
    return exchangeMap.get(exchangeName);
}

public void deleteExchange(String exchangeName) {
    exchangeMap.remove(exchangeName);
}
```

封装 Queue 方法

```
public void insertQueue(MSGQueue queue) {
    queueMap.put(queue.getName(), queue);
}

public MSGQueue getQueue(String queueName) {
    return queueMap.get(queueName);
}

public void deleteQueue(String queueName) {
    queueMap.remove(queueName);
}
```

封装 Binding 方法

```
public void insertBinding(Binding binding) throws MqException {
    HashMap<String, Binding> bindingMap = bindingsMap.computeIfAbsent(binding.ge
    synchronized (bindingMap) {
        // 不存在就创建一份
        if (bindingMap.get(binding.getQueueName()) != null) {
            throw new MqException("[MemoryDataCenter] 绑定已经存在! exchangeName='
                    + ", queueName=" + binding.getQueueName());
        }
        bindingMap.put(binding.getQueueName(), binding);
    }
}

public Binding getBinding(String queueName, String exchangeName) {
    HashMap<String, Binding> bindingMap = bindingsMap.get(exchangeName);
    if (bindingMap == null) {
        return null;
    }
    synchronized (bindingMap) {
        return bindingMap.get(queueName);
    }
}

public void deleteBinding(Binding binding) throws MqException {
    HashMap<String, Binding> bindingMap = bindingsMap.get(binding.getExchangeNam
    if (bindingMap == null) {
        throw new MqException("[MemoryDataCenter] 绑定不存在! exchangeName=" + bi
```

```
                + ", queueName=" + binding.getQueueName());
    }
    synchronized (bindingMap) {
        Binding toDelete = bindingMap.get(binding.getQueueName());
        if (toDelete == null) {
            throw new MqException("[MemoryDataCenter] 绑定不存在! exchangeName="
                    + ", queueName=" + binding.getQueueName());
        }
        bindingMap.remove(binding.getQueueName());
    }
}

public Map<String, Binding> getBindingsByExchange(String exchangeName) {
    return bindingsMap.get(exchangeName);
}
```

封装 Message 方法

```
// 查询指定的消息
public Message getMessage(String messageId) {
    return messageMap.get(messageId);
}

// 向消息中心中添加消息
public void addMessage(Message message) {
    messageMap.put(message.getMessageId(), message);
    System.out.println("[MemoryCenter] 新消息被添加! messageId=" + message.getMes
}

// 从消息中心删除消息
public void removeMessage(String messageId) {
    messageMap.remove(messageId);
    System.out.println("[MemoryCenter] 消息被彻底删除! messageId=" + messageId);
}

// 发送消息到指定队列中
public void sendMessage(MSGQueue queue, Message message) {
    List<Message> messageList = queueMessageMap.computeIfAbsent(queue.getName(),
    synchronized (messageList) {
        messageList.add(message);
    }
    // 如果消息已经存在, 重复调用也没啥大不了的.
    addMessage(message);
```

```
    System.out.println("[MemoryCenter] 消息被投递到队列中! messageId=" + message.g
}

// 从指定队列中取消息.
public Message pollMessage(String queueName) throws MqException {
    List<Message> messageList = queueMessageMap.get(queueName);
    if (messageList == null) {
        throw new MqException("[MemoryDataCenter] 队列不存在! queueName=" + queue
    }
    synchronized (messageList) {
        if (messageList.size() == 0) {
            return null;
        }
        // 出队列头元素
        Message currentMessage = messageList.remove(0);
        System.out.println("[MemoryCenter] 消息从队列中取出! messageId=" + current
        return currentMessage;
    }
}

public int getMessageCount(String queueName) throws MqException {
    List<Message> messageList = queueMessageMap.get(queueName);
    if (messageList == null) {
        // 如果队列不存在, 则直接返回长度 0, 说明该 queueName 下还没有消息.
        return 0;
    }
    synchronized (messageList) {
        return messageList.size();
    }
}
```

针对未确认的消息的处理

```
// 未被确认的消息, 先临时存放一下
public void addMessageWaitAck(String queueName, Message message) {
    HashMap<String, Message> messageHashMap = queueMessageWaitAck.computeIfAbsen
    synchronized (messageHashMap) {
        messageHashMap.put(message.getMessageId(), message);
    }
    System.out.println("[MemoryCenter] 消息进入待确认队列! messageId=" + message.g
}

// 消息被确认之后, 就可以真正删除了.
```

```
public void removeMessageWaitAck(String queueName, String messageId) {
    HashMap<String, Message> messageHashMap = queueMessageWaitAck.get(queueName)
    if (messageHashMap == null) {
        return;
    }
    synchronized (messageHashMap) {
        messageHashMap.remove(messageId);
    }
    System.out.println("[MemoryCenter] 消息从待确认队列删除! messageId=" + message
}

public Message getMessageWaitAck(String queueName, String messageId) {
    HashMap<String, Message> messageHashMap = queueMessageWaitAck.get(queueName)
    if (messageHashMap == null) {
        return null;
    }
    synchronized (messageHashMap) {
        return messageHashMap.get(messageId);
    }
}
```

实现重启后恢复内存

```
// 从硬盘上恢复数据
public void recovery(DiskDataCenter diskDataCenter) throws MqException, IOExcept
    // 1. 恢复交换机数据
    List<Exchange> exchanges = diskDataCenter.selectAllExchanges();
    for (Exchange exchange : exchanges) {
        exchangeMap.put(exchange.getName(), exchange);
    }
    // 2. 恢复队列数据
    List<MSGQueue> queues = diskDataCenter.selectAllQueues();
    for (MSGQueue queue : queues) {
        queueMap.put(queue.getName(), queue);
    }
    // 3. 恢复绑定数据
    List<Binding> bindings = diskDataCenter.selectAllBindings();
    for (Binding binding : bindings) {
        HashMap<String, Binding> bindingMap = bindingsMap.computeIfAbsent(bindin
        bindingMap.put(binding.getQueueName(), binding);
    }
    // 4. 恢复消息数据
    //    只需要恢复 queueMessageMap 和 messageMap
```

```
21      //    queueMessageWaitAck 则不必恢复. 未被确认的消息只是在内存存储. 如果这个时候 b
22      for (MSGQueue queue : queues) {
23          LinkedList<Message> messages = diskDataCenter.loadAllMessageFromQueue(qu
24          queueMessageMap.put(queue.getName(), messages);
25          for (Message message : messages) {
26              messageMap.put(message.getMessageId(), message);
27          }
28      }
29  }
```

测试 MemoryDataCenter

创建 `MemoryDataCenterTests`

```
1  @SpringBootTest
2  public class MemoryDataCenterTests {
3      private MemoryDataCenter memoryDataCenter = null;
4
5      @BeforeEach
6      public void setUp() {
7          memoryDataCenter = new MemoryDataCenter();
8          memoryDataCenter.init();
9      }
10
11     @AfterEach
12     public void tearDown() {
13         memoryDataCenter = null;
14     }
```

```
1  private Exchange createTestExchange(String exchangeName) {
2      Exchange exchange = new Exchange();
3      exchange.setName(exchangeName);
4      exchange.setType(ExchangeType.FANOUT);
5      exchange.setAutoDelete(true);
6      exchange.setDurable(true);
7      HashMap<String, Object> arguments = new HashMap<>();
8      arguments.put("aaa", "111");
9      arguments.put("bbb", "222");
10     exchange.setArguments(arguments);
11     return exchange;
12 }
```

```
13
14 private MSGQueue createTestQueue(String queueName) {
15     MSGQueue queue = new MSGQueue();
16     queue.setName(queueName);
17     queue.setDurable(true);
18     queue.setAutoDelete(true);
19     queue.setExclusive(true);
20     HashMap<String, Object> hashMap = new HashMap<>();
21     hashMap.put("aaa", "111");
22     hashMap.put("bbb", "222");
23     queue.setArguments(hashMap);
24     return queue;
25 }
26
27 @Test
28 public void testExchange() {
29     Exchange expectedExchange = createTestExchange("testExchange");
30     memoryDataCenter.insertExchange(expectedExchange);
31
32     Exchange actualExchange = memoryDataCenter.getExchange("testExchange");
33     Assertions.assertEquals(expectedExchange, actualExchange);
34
35     memoryDataCenter.deleteExchange("testExchange");
36     actualExchange = memoryDataCenter.getExchange("testExchange");
37     Assertions.assertNull(actualExchange);
38 }
39
40 @Test
41 public void testQueue() {
42     MSGQueue expectedQueue = createTestQueue("testQueue");
43     memoryDataCenter.insertQueue(expectedQueue);
44
45     MSGQueue actualQueue = memoryDataCenter.getQueue("testQueue");
46     Assertions.assertEquals(expectedQueue, actualQueue);
47
48     memoryDataCenter.deleteQueue("testQueue");
49     actualQueue = memoryDataCenter.getQueue("testQueue");
50     Assertions.assertNull(actualQueue);
51 }
52
53 @Test
54 public void testBinding() throws MqException {
55     Binding expectedBinding = new Binding();
56     expectedBinding.setQueueName("testQueue");
57     expectedBinding.setExchangeName("testExchange");
58     expectedBinding.setBindingKey("testBindingKey");
59     memoryDataCenter.insertBinding(expectedBinding);
```

```
60
61      Binding actualBinding = memoryDataCenter.getBinding("testQueue", "testExchan
62      Assertions.assertEquals(expectedBinding, actualBinding);
63
64      Map<String, Binding> bindingMap = memoryDataCenter.getBindingsByExchange("te
65      actualBinding = bindingMap.get("testQueue");
66      Assertions.assertEquals(expectedBinding, actualBinding);
67
68      memoryDataCenter.deleteBinding(expectedBinding);
69      actualBinding = memoryDataCenter.getBinding("testQueue", "testExchange");
70      Assertions.assertNull(actualBinding);
71  }
72
73  private Message createTestMessage(String content) {
74      Message message = new Message();
75      message.setMessageId("M-" + UUID.randomUUID().toString());
76      message.setRoutingKey("testRoutingKey");
77      message.setDeliveryMode(2);
78      message.setBody(content.getBytes());
79      return message;
80  }
81
82  @Test
83  public void testMessage() {
84      Message expectedMessage = createTestMessage("testMessage");
85      memoryDataCenter.addMessage(expectedMessage);
86
87      Message actualMessage = memoryDataCenter.getMessage(expectedMessage.getMessa
88      Assertions.assertEquals(expectedMessage, actualMessage);
89
90      memoryDataCenter.removeMessage(expectedMessage.getMessageId());
91      actualMessage = memoryDataCenter.getMessage(expectedMessage.getMessageId());
92      Assertions.assertNull(actualMessage);
93  }
94
95  @Test
96  public void testSendMessage() throws MqException {
97      MSGQueue queue = createTestQueue("testQueue");
98      List<Message> expectedMessages = new ArrayList<>();
99      for (int i = 0; i < 10; i++) {
100         Message message = createTestMessage("testMessage");
101         memoryDataCenter.sendMessage(queue, message);
102         expectedMessages.add(message);
103     }
104
105     List<Message> actualMessages = new ArrayList<>();
106     while (true) {
```

```
107         Message message = memoryDataCenter.pollMessage("testQueue");
108         if (message == null) {
109             break;
110         }
111         actualMessages.add(message);
112     }
113 
114     Assertions.assertEquals(expectedMessages.size(), actualMessages.size());
115     for (int i = 0; i < expectedMessages.size(); i++) {
116         Assertions.assertEquals(expectedMessages.get(i), actualMessages.get(i));
117     }
118 }
119 
120 @Test
121 public void testMessageWaitAck() {
122     Message expectedMessage = createTestMessage("testMessage");
123     memoryDataCenter.addMessageWaitAck("testQueue", expectedMessage);
124     Message actualMessage = memoryDataCenter.getMessageWaitAck("testQueue", expe
125     Assertions.assertEquals(expectedMessage, actualMessage);
126 
127     memoryDataCenter.removeMessageWaitAck("testQueue", expectedMessage.getMessag
128     actualMessage = memoryDataCenter.getMessageWaitAck("testQueue", expectedMess
129     Assertions.assertNull(actualMessage);
130 }
131 
132 @Test
133 public void testRecovery() throws IOException, MqException, ClassNotFoundExcepti
134     JavaMessageQueueApplication.ac = SpringApplication.run(JavaMessageQueueAppli
135 
136     // 构造初始数据
137     DiskDataCenter diskDataCenter = new DiskDataCenter();
138     diskDataCenter.init("");
139 
140     Exchange expectedExchange = createTestExchange("testExchange");
141     diskDataCenter.insertExchange(expectedExchange);
142 
143     MSGQueue expectedQueue = createTestQueue("testQueue");
144     diskDataCenter.insertQueue(expectedQueue);
145 
146     Binding expectedBinding = new Binding();
147     expectedBinding.setExchangeName("testExchange");
148     expectedBinding.setQueueName("testQueue");
149     expectedBinding.setBindingKey("testBindingKey");
150     diskDataCenter.insertBinding(expectedBinding);
151 
152     Message expectedMessage = createTestMessage("testMessage");
153     diskDataCenter.sendMessage(expectedQueue, expectedMessage);
```

```
154
155     // 恢复数据
156     memoryDataCenter.recovery(diskDataCenter);
157
158     // 对比结果
159     Exchange actualExchange = memoryDataCenter.getExchange("testExchange");
160     Assertions.assertEquals(expectedExchange.getType(), actualExchange.getType()
161     Assertions.assertEquals(expectedExchange.isDurable(), actualExchange.isDurab
162     Assertions.assertEquals(expectedExchange.isAutoDelete(), actualExchange.isAu
163     Assertions.assertEquals(expectedExchange.getArguments(), actualExchange.getA
164
165     MSGQueue actualQueue = memoryDataCenter.getQueue("testQueue");
166     Assertions.assertEquals(expectedQueue.isDurable(), actualQueue.isDurable());
167     Assertions.assertEquals(expectedQueue.isAutoDelete(), actualQueue.isAutoDele
168     Assertions.assertEquals(expectedQueue.isExclusive(), actualQueue.isExclusive
169     Assertions.assertEquals(expectedQueue.getArguments(), actualQueue.getArgumen
170
171     Binding actualBinding = memoryDataCenter.getBinding("testQueue", "testExchan
172     Assertions.assertEquals(expectedBinding.getBindingKey(), actualBinding.getBi
173
174     // 清理
175     JavaMessageQueueApplication.ac.close();
176     File dbFile = new File("meta.db");
177     dbFile.delete();
178     File dataFile = new File("./data");
179     FileUtils.deleteDirectory(dataFile);
180 }
```

十. 虚拟主机设计

至此, 内存和硬盘的数据都已经组织完成. 接下来使用 "虚拟主机" 这个概念, 把这两部分的数据也串起来.

并且实现一些 MQ 的关键 API.

注意: 在 RabbitMQ 中, 虚拟主机是可以随意创建/删除的. 咱们此处为了实现简单, 并没有实现虚拟主机的管理. 因此我们默认就只有一个虚拟主机的存在. 但是在数据结构的设计上我们预留了对于多虚拟主机的管理.

保证不同虚拟主机中的 Exchange, Queue, Binding, Message 都是相互隔离的.

创建 VirtualHost

创建 `mqserver.VirtualHost`

```
public class VirtualHost {
    private String virtualhostName;
    private DiskDataCenter diskDataCenter = new DiskDataCenter();
    private MemoryDataCenter memoryDataCenter = new MemoryDataCenter();
    private Router router = new Router();
    private ConsumerManager consumerManager = new ConsumerManager(this);
}
```

其中 Router 用来定义转发规则, ConsumerManager 用来实现消息消费. 这两个内容后续再介绍

实现构造方法和 getter

构造方法中会针对 DiskDataCenter 和 MemoryDataCenter 进行初始化.

同时会把硬盘的数据恢复到内存中.

```
public VirtualHost(String virtualhostName) {
    this.virtualhostName = virtualhostName;

    // 先初始化硬盘数据
    diskDataCenter.init(virtualhostName);
    // 后初始化内存数据
    memoryDataCenter.init();
    try {
        // 进行恢复操作
        memoryDataCenter.recovery(diskDataCenter);
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("[VirtualHost] 恢复内存数据失败!");
    }
}

public String getVirtualhostName() {
    return virtualhostName;
}

public DiskDataCenter getDiskDataCenter() {
    return diskDataCenter;
}

public MemoryDataCenter getMemoryDataCenter() {
```

```
26      return memoryDataCenter;
27  }
```

创建交换机

- 此处的 autoDelete, arguments 其实并没有使用. 只是先预留出来. (RabbitMQ 是支持的) .
- 约定, 交换机/队列的名字, 都加上 VirtualHostName 作为前缀. 这样不同 VirtualHost 中就可以存在同名的交换机或者队列了.
- exchangeDeclare 的语义是, 不存在就创建, 存在则直接返回. 因此不叫做 "exchangeCreate".
- 先写硬盘, 后写内存. 因为写硬盘失败概率更大. 如果硬盘写失败了, 也就不必写内存了.

```
1  // 创建交换机
2  // 先写硬盘，后写内存. 写硬盘失败概率更大，如果异常了，也就不写内存了.
3  public boolean exchangeDeclare(String exchangeName, ExchangeType exchangeType, b
4                                 Map<String, Object> arguments) {
5      // 真实的 exchangeName 需要拼接上 virtualhostName
6      exchangeName = virtualhostName + exchangeName;
7      try {
8          // 1. 判定该交换机是否存在
9          Exchange existsExchange = memoryDataCenter.getExchange(exchangeName);
10         if (existsExchange != null) {
11             System.out.println("[VirtualHost] 交换机已经存在! exchangeName=" + exc
12             return true;
13         }
14         // 2. 构造 Exchange 对象
15         Exchange exchange = new Exchange();
16         exchange.setName(exchangeName);
17         exchange.setType(exchangeType);
18         exchange.setDurable(durable);
19         exchange.setAutoDelete(autoDelete);
20         exchange.setArguments(arguments);
21         // 3. 把数据写入硬盘
22         if (durable) {
23             diskDataCenter.insertExchange(exchange);
24         }
25         // 4. 把数据写入内存
26         memoryDataCenter.insertExchange(exchange);
27         System.out.println("[VirtualHost] 交换机创建完成! exchangeName=" + exchang
28         return true;
29     } catch (Exception e) {
30         System.out.println("[VirtualHost] 交换机创建失败! exchangeName=" + exchang
31         e.printStackTrace();
```

```
        return false;
    }
}
```

删除交换机

```
// 删除交换机
// 先写硬盘，后写内存. 写硬盘失败概率更大，如果异常了，也就不写内存了.
public boolean exchangeDelete(String exchangeName) {
    // 真实的 exchangeName 需要拼接上 virtualhostName
    exchangeName = virtualhostName + exchangeName;
    try {
        // 1. 先找到对应的交换机.
        Exchange toDelete = memoryDataCenter.getExchange(exchangeName);
        if (toDelete == null) {
            throw new MqException("[VirtualHost] 交换机不存在，无法删除!");
        }
        // 2. 删除硬盘上的交换机数据
        if (toDelete.isDurable()) {
            diskDataCenter.deleteExchange(exchangeName);
        }
        // 3. 删除内存中的交换机数据
        memoryDataCenter.deleteExchange(exchangeName);
        System.out.println("[VirtualHost] 交换机删除成功! exchangeName=" + exchang
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] 交换机删除失败! exchangeName=" + exchang
        e.printStackTrace();
        return false;
    }
}
```

创建队列

```
// 创建队列
public boolean queueDeclare(String queueName, boolean durable, boolean exclusive
                            Map<String, Object> arguments) {
    // 真实的 queueName 需要拼接上 virtualhostName
    queueName = virtualhostName + queueName;
```

```
    try {
        // 1. 判定队列是否存在
        MSGQueue existsQueue = memoryDataCenter.getQueue(queueName);
        if (existsQueue != null) {
            System.out.println("[VirtualHost] 队列已经存在! queueName=" + queueNar
            return true;
        }
        // 2. 创建队列对象
        MSGQueue queue = new MSGQueue();
        queue.setName(queueName);
        queue.setDurable(durable);
        queue.setAutoDelete(autoDelete);
        queue.setArguments(arguments);
        // 3. 写硬盘
        if (durable) {
            diskDataCenter.insertQueue(queue);
        }
        // 4. 写内存
        memoryDataCenter.insertQueue(queue);
        System.out.println("[VirtualHost] 队列创建成功! queueName=" + queueName);
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] 队列创建失败! queueName=" + queueName);
        e.printStackTrace();
        return false;
    }
}
```

删除队列

```
// 删除队列
public boolean queueDelete(String queueName) {
    // 真实的 queueName 需要拼接上 virtualhostName
    queueName = virtualhostName + queueName;
    try {
        // 1. 根据 queueName 查询对应的队列对象
        MSGQueue queue = memoryDataCenter.getQueue(queueName);
        if (queue == null) {
            throw new MqException("[VirtualHost] 队列不存在, 无法删除!");
        }
        // 2. 删除硬盘数据
        if (queue.isDurable()) {
            diskDataCenter.deleteQueue(queueName);
```

```
        }
        // 3. 删除内存数据
        memoryDataCenter.deleteQueue(queueName);
        System.out.println("[VirtualHost] 队列删除成功! queueName=" + queueName);
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] 队列删除失败! queueName=" + queueName);
        e.printStackTrace();
        return false;
    }
}
```

创建绑定

- bindingKey 是进行 topic 转发时的一个关键概念. 使用 router 类来检测是否是合法的 bindingKey.
- 后续再介绍 `router.checkBindingKeyValid` 的实现. 此处先留空.

```
// 创建绑定
public boolean queueBind(String queueName, String exchangeName, String bindingKe
    // 真实的 queueName 需要拼接上 virtualhostName
    queueName = virtualhostName + queueName;
    exchangeName = virtualhostName + exchangeName;
    try {
        // 1. 判定 binding 是否存在
        Binding existsBinding = memoryDataCenter.getBinding(queueName, exchangeN
        if (existsBinding != null) {
            throw new MqException("[VirtualHost] binding 已经存在! queueName=" +
        }
        // 2. 校验 bindingKey 是否合法
        if (!router.checkBindingKeyValid(bindingKey)) {
            throw new MqException("[VirtualHost] bindingKey 非法! bindingKey=" +
        }
        // 3. 创建 binding 对象
        Binding binding = new Binding();
        binding.setQueueName(queueName);
        binding.setExchangeName(exchangeName);
        binding.setBindingKey(bindingKey);
        // 4. 获取到对应的 exchange 和 queue 对象
        MSGQueue queue = memoryDataCenter.getQueue(queueName);
        if (queue == null) {
            throw new MqException("[VirtualHost] 对应的队列不存在! queueName=" + q
        }
        Exchange exchange = memoryDataCenter.getExchange(exchangeName);
```

```
        if (exchange == null) {
            throw new MqException("[VirtualHost] 对应的交换机不存在! exchangeName=
        }
        // 5. 如果 exchange 和 queue 都是持久化的, 则 binding 也持久化.
        if (queue.isDurable() && exchange.isDurable()) {
            diskDataCenter.insertBinding(binding);
        }
        // 6. 写入内存
        memoryDataCenter.insertBinding(binding);
        System.out.println("[VirtualHost] 创建绑定成功! exchangeName=" + exchangeN
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] 创建绑定失败! exchangeName=" + exchangeN
        e.printStackTrace();
        return false;
    }
}
```

删除绑定

```
// 解除绑定
public boolean queueUnbind(String queueName, String exchangeName) {
    // 真实的 queueName 需要拼接上 virtualhostName
    queueName = virtualhostName + queueName;
    exchangeName = virtualhostName + exchangeName;
    try {
        // 1. 获取到 binding
        Binding binding = memoryDataCenter.getBinding(queueName, exchangeName);
        if (binding == null) {
            throw new Exception("[VirtualHost] 绑定不存在!");
        }
        // 2. 获取到对应的 exchange 和 queue 对象
        MSGQueue queue = memoryDataCenter.getQueue(queueName);
        if (queue == null) {
            throw new Exception("[VirtualHost] 对应的队列不存在! queueName=" + que
        }
        Exchange exchange = memoryDataCenter.getExchange(exchangeName);
        if (exchange == null) {
            throw new Exception("[VirtualHost] 对应的交换机不存在! exchangeName="
        }
        // 3. 如果 exchange 和 queue 都是持久化的, 则 binding 从硬盘删除
        if (queue.isDurable() && exchange.isDurable()) {
            diskDataCenter.deleteBinding(binding);
```

```
        }
        // 4. 从内存删除 binding
        memoryDataCenter.deleteBinding(binding);
        System.out.println("[VirtualHost] 绑定删除成功! exchangeName=" + exchangeN
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] 绑定删除失败! exchangeName=" + exchangeN
        e.printStackTrace();
        return false;
    }
}
```

发布消息

- 发布消息其实是把消息发送给指定的 Exchange, 再根据 Exchange 和 Queue 的 Binding 关系, 转发到对应队列中.
- 发送消息需要指定 routingKey, 这个值的作用和 ExchangeType 是相关的.
  - Direct: routingKey 就是对应队列的名字. 此时不需要 binding 关系, 也不需要 bindingKey, 就可以直接转发消息.
  - Fanout: routingKey 不起作用, bindingKey 也不起作用. 此时消息会转发给绑定到该交换机上的所有队列中.
  - Topic: routingKey 是一个特定的字符串, 会和 bindingKey 进行匹配. 如果匹配成功, 则发到对应的队列中. 具体规则后续介绍.
- BasicProperties 是消息的元信息. body 是消息本体.

```
// 发送消息
public boolean basicPublish(String exchangeName, String routingKey,
                            BasicProperties basicProperties, byte[] body) {
    try {
        // 1. 转换交换机名字. 如果是 null, 则使用默认交换机
        if (exchangeName == null) {
            exchangeName = "";
        }
        exchangeName = virtualhostName + exchangeName;

        // 2. 检查参数合法性
        if (!router.checkRoutingKeyValid(routingKey)) {
            throw new MqException("[VirtualHost] routingKey 非法! routingKey=" +
        }

```

```
16          // 3. 查找到交换机对象
17          Exchange exchange = memoryDataCenter.getExchange(exchangeName);
18          if (exchange == null) {
19              throw new MqException("[VirtualHost] 交换机不存在! exchangeName=" + e
20          }
21
22          if (exchange.getType() == ExchangeType.DIRECT) {
23              String queueName = virtualhostName + routingKey;
24              // 4. 构造消息对象
25              Message message = Message.createMessageWithId(routingKey, basicPrope
26              // 5. 直接转发, 不需要 binding, 直接根据 routingKey 找到队列名, 进行转发.
27              MSGQueue queue = memoryDataCenter.getQueue(queueName);
28              if (queue == null) {
29                  throw new MqException("[VirtualHost] 队列不存在! queueName=" + qu
30              }
31              // 6. 直接转发消息
32              sendMessage(queue, message);
33          } else {
34              // 4. 找到交换机对应的绑定对象
35              Map<String, Binding> bindings = memoryDataCenter.getBindingsByExchan
36
37              // 5. 遍历所有绑定, 进入消息转发逻辑.
38              for (Map.Entry<String, Binding> entry : bindings.entrySet()) {
39                  // 1) 判定队列是否存在
40                  Binding binding = entry.getValue();
41                  MSGQueue queue = memoryDataCenter.getQueue(binding.getQueueName(
42                  if (queue == null) {
43                      throw new MqException("[VirtualHost] 队列不存在! queueName="
44                  }
45                  // 2) 构造消息对象. 针对每次写入队列, 都构造一个唯一的消息对象 id. 使同
46                  //    如果两个队列中的消息 id 一样, 此时就可能在 messageMap 中只存在一
47                  //    此时针对消息进行消费操作, 就可能出现一个队列消费了之后, 把消息从
48                  //    的时候, 就无法从 messageMap 中获取到消息了.
49                  Message message = Message.createMessageWithId(routingKey, basicP
50                  // 3) 判定能否转发
51                  if (!router.route(exchange.getType(), binding, message)) {
52                      continue;
53                  }
54                  // 4) 真正转发消息
55                  sendMessage(queue, message);
56              }
57          }
58          return true;
59      } catch (Exception e) {
60          System.out.println("[VirtualHost] 消息发布失败!");
61          e.printStackTrace();
62          return false;
```

```
    }
}
```

```
private void sendMessage(MSGQueue queue, Message message) throws Exception {
    // 1. 先写硬盘
    //    deliverMode 为 1, 表示不持久化; 为 2 表示持久化. AMQP 协议规定的.
    int deliveryMode = message.getBasicProperties().getDeliveryMode();
    if (deliveryMode == 2) {
        diskDataCenter.sendMessage(queue, message);
    }
    // 2. 再写内存
    memoryDataCenter.sendMessage(queue, message);
    // 3. 通知消费者去取消息
    consumerManager.notifyConsume(queue.getName());
}
```

路由规则

实现 `mqserver.core.Router`

1) 实现 route 方法

```
public class Router {
    public boolean route(ExchangeType exchangeType, Binding binding, Message mes
        // 根据不同的 exchangeType 进行不同的转发逻辑
        // DIRECT 的转发逻辑已经在外部判定过.
        if (exchangeType == ExchangeType.FANOUT) {
            return routeFanout(binding, message);
        } else if (exchangeType == ExchangeType.TOPIC) {
            return routeTopic(binding, message);
        } else {
            throw new MqException("[VirtualHost] 未知的 exchangeType! exchangeTyp
        }
    }

    private boolean routeFanout(Binding binding, Message message) {
        // 对于 fanout 类型, 直接转发, 不需要进行任何匹配.
        return true;
    }
}
```

2) 实现 checkRoutingKeyValid

一个 RoutingKey 是由数字字母下划线构成的, 并且可以使用 . 分成若干部分.

形如 `aaa.bbb.ccc`

```
// 不包含通配符, 规则更简单.
public boolean checkRoutingKeyValid(String routingKey) {
    if (routingKey.length() == 0) {
        return true;
    }
    // 数字字母下划线构成
    for (int i = 0; i < routingKey.length(); i++) {
        char ch = routingKey.charAt(i);
        if (ch >= 'A' && ch <= 'Z') {
            continue;
        }
        if (ch >= 'a' && ch <= 'z') {
            continue;
        }
        if (ch >= '0' && ch <= '9') {
            continue;
        }
        if (ch == '_' || ch == '.') {
            continue;
        }
        return false;
    }
    return true;
}
```

3) 实现 checkBindingKeyValid

一个 BindingKey 是由数字字母下划线构成的, 并且使用 . 分成若干部分.

另外, 支持 * 和 # 两种通配符. (* # 只能作为 . 切分出来的独立部分, 不能和其他数字字母混用, 比如 a.*.b 是合法的, a.*a.b 是不合法的).

其中 * 可以匹配任意一个单词.

其中 # 可以匹配任意零个或者多个单词.

例如:

bindingKey 为 a.*.b, 可以匹配 routingKey 为 a.a.b 和 a.b.b 和 a.aaa.b

bindingKey 为 a.#.b, 可以匹配 routingKey 为 a.a.b 和 a.b.b 和 a.aaa.b 和 a.aa.bb.b 和 a.b

```
    // 需要考虑通配符, 复杂一些
    public boolean checkBindingKeyValid(String bindingKey) {
        // 1. 允许是空字符串
        // 2. 数字字母下划线构成
        // 3. 可以包含通配符
        // 4. # 不能连续出现.
        // 5. # 和 * 不能相邻
        if (bindingKey.length() == 0) {
            return true;
        }
        // 先判定基础构成
        for (int i = 0; i < bindingKey.length(); i++) {
            char ch = bindingKey.charAt(i);
            if (ch >= 'A' && ch <= 'Z') {
                continue;
            }
            if (ch >= 'a' && ch <= 'z') {
                continue;
            }
            if (ch >= '0' && ch <= '9') {
                continue;
            }
            if (ch == '.' || ch == '_' || ch == '*' || ch == '#') {
                continue;
            }
            return false;
        }
        // 再判定每个词的情况
        // 比如 aaa.a*a 这种应该视为非法.
        String[] words = bindingKey.split("\\.");
        for (String word : words) {
            if (word.length() > 1 && (word.contains("*") || word.contains("#")))
                return false;
            }
        }
        // 再判定相邻词的情况
        for (int i = 0; i < words.length - 1; i++) {
            // 连续两个 ##
            if (words[i].equals("#") && words[i + 1].equals("#")) {
                return false;
            }
```

```
            // # 连着 *
            if (words[i].equals("#") && words[i + 1].equals("*")) {
                return false;
            }
            // * 连着 #
            if (words[i].equals("*") && words[i + 1].equals("#")) {
                return false;
            }
        }
        return true;
    }
```

4) 实现 routeTopic

```
// 需要按照通配符匹配
// binding key 包含通配符
// 1. * 表示任意一个 token 都可以匹配
// 2. # 表示任意 0 个或 N 个 token 都可以匹配
// 3. 其他内容则要求严格匹配.
// 4. # 不会连续出现. # 和 * 不会相邻.
// routing key 不包含通配符. 这个在发消息的时候校验
private boolean routeTopic(Binding binding, Message message) {
    // 按照 . 来切分 binding key 和 routing key
    String[] bindingTokens = binding.getBindingKey().split("\\.");
    String[] routingTokens = message.getRoutingKey().split("\\.");
    // 使用双指针的方式来实现匹配
    // 1. 如果是普通字符, 直接匹配内容是否相等, 不相等则返回 false, 相等直接进入下一轮
    // 2. 如果是 * , 直接进入下一轮
    // 3. 如果 # 没有下一个位置, 则直接返回 true
    // 4. 如果遇到 # , 则找到 # 下一个位置的 token 在 routingKey 中的位置.
    // 5. 如果能找到对应的位置了, 就可以继续匹配. 如果找不到, 就返回 false
    // 6. 循环结束后, 检查看两个下标是否同时到达末尾. 是则匹配成功, 否则匹配失败.
    int bindingIndex = 0;
    int routingIndex = 0;
    while (bindingIndex < bindingTokens.length && routingIndex < routingTokens.l
        if (bindingTokens[bindingIndex].equals("*")) {
            // 2. 如果是 * , 直接进入下一轮
            // 直接进入下一轮比较
            bindingIndex++;
            routingIndex++;
        } else if (bindingTokens[bindingIndex].equals("#")) {
            bindingIndex++;
            if (bindingIndex == bindingTokens.length) {
```

```
30                  // 3. 如果 # 没有下一个位置，则直接返回 true
31                  return true;
32              }
33              // 4. 如果遇到 # ，则找到 # 下一个位置的 token 在 routingKey 中的位置.
34              routingIndex = findNextMatch(routingTokens, routingIndex, bindingTok
35              // 5. 如果能找到对应的位置了，就可以继续下一轮匹配. 如果找不到，就返回 fals
36              if (routingIndex == -1) {
37                  return false;
38              }
39              bindingIndex++;
40              routingIndex++;
41          } else {
42              // 1. 如果是普通字符，直接匹配内容是否相等，不相等则返回 false，相等直接进入
43              if (!bindingTokens[bindingIndex].equals(routingTokens[routingIndex])
44                  return false;
45              }
46              bindingIndex++;
47              routingIndex++;
48          }
49      }
50
51      // 如果两方不能同时结束，则也视为匹配失败.
52      // 比如 aaa.*.bbb 和 aaa.bbb
53      if (bindingIndex == bindingTokens.length && routingIndex == routingTokens.le
54          return true;
55      }
56      return false;
57  }
58
59  private int findNextMatch(String[] routingTokens, int routingIndex, String bindi
60      for (int i = routingIndex; i < routingTokens.length; i++) {
61          if (routingTokens[i].equals(bindingToken)) {
62              return i;
63          }
64      }
65      return -1;
66  }
```

5) 匹配规则测试用例

```
1  // [测试用例]
2  // binding key          routing key          result
3  // aaa                  aaa                  true
```

```
// aaa.bbb              aaa.bbb             true
// aaa.bbb              aaa.bbb.ccc         false
// aaa.bbb              aaa.ccc             false
// aaa.bbb.ccc          aaa.bbb.ccc         true
// aaa.*                aaa.bbb             true
// aaa.*.bbb            aaa.bbb.ccc         false
// *.aaa.bbb            aaa.bbb             false
// #                    aaa.bbb.ccc         true
// aaa.#                aaa.bbb             true
// aaa.#                aaa.bbb.ccc         true
// aaa.#.ccc            aaa.ccc             true
// aaa.#.ccc            aaa.bbb.ccc         true
// aaa.#.ccc            aaa.aaa.bbb.ccc     true
// #.ccc                ccc                 true
// #.ccc                aaa.bbb.ccc         true
```

6) 测试 Router

创建 `RouterTests`

```java
@SpringBootTest
public class RouterTests {
    private Router router = new Router();
    private Message message = null;
    private Binding binding = null;

    @BeforeEach
    public void setUp() {
        message = new Message();
        binding = new Binding();
    }

    @AfterEach
    public void tearDown() {
        message = null;
        binding = null;
    }
}
```

```java
@Test
public void test() throws MqException {
    binding.setBindingKey("aaa");
```

```
    message.setRoutingKey("aaa");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.bbb");
    message.setRoutingKey("aaa.bbb");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.bbb");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertFalse(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.bbb");
    message.setRoutingKey("aaa.ccc");
    Assertions.assertFalse(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.bbb.ccc");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.*");
    message.setRoutingKey("aaa.bbb");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.*.bbb");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertFalse(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("*.aaa.bbb");
    message.setRoutingKey("aaa.bbb");
    Assertions.assertFalse(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("#");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.#");
    message.setRoutingKey("aaa.bbb");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.#");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.#.ccc");
    message.setRoutingKey("aaa.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

```

```
    binding.setBindingKey("aaa.#.ccc");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("aaa.#.ccc");
    message.setRoutingKey("aaa.aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("#.ccc");
    message.setRoutingKey("ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));

    binding.setBindingKey("#.ccc");
    message.setRoutingKey("aaa.bbb.ccc");
    Assertions.assertTrue(router.route(ExchangeType.TOPIC, binding, message));
}
```

订阅消息

1) 添加一个订阅者

```
// 订阅消息
// 如果是多个消费者消费一个队列, 将使用轮询的方式进行消费.
// 参数的 consumerTag 应该在网络通信部分设定.
public boolean basicConsume(String consumerTag, String queueName, boolean autoAc
    queueName = virtualhostName + queueName;
    try {
        // 把 consumer 加到监听线程管理的消费者数组中
        consumerManager.addConsumer(consumerTag, queueName, autoAck, consumer);
        System.out.println("[VirtualHost] basicConsume 成功! queueName=" + queueI
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] basicConsume 失败! queueName=" + queueI
        e.printStackTrace();
        return false;
    }
}
```

Consumer 相当于一个回调函数. 放到 `common.Consumer` 中.

```
@FunctionalInterface
```

```
public interface Consumer {
    // consumerTag 消费者标识, 后面使用 channelId 填充.
    void handleDelivery(String consumerTag, BasicProperties properties, byte[] b
}
```

2) 创建订阅者管理管理类

创建 `mqserver.core.ConsumerManager`

```
public class ConsumerManager {
    private VirtualHost parent;
    // 存放令牌的队列. 通过令牌来触发消费线程的消费操作.
    private BlockingQueue<String> tokenQueue = new LinkedBlockingQueue<>();
    private ExecutorService workerPool = Executors.newFixedThreadPool(4);
}
```

- parent 用来记录虚拟主机.
- 使用一个阻塞队列用来触发消息消费. 称为令牌队列. 每次有消息过来了, 都往队列中放一个令牌(也就是队列名), 然后消费者再去消费对应队列的消息.
- 使用一个线程池用来执行消息回调.

> 这样令牌队列的设定避免搞出来太多线程. 否则就需要给每个队列都安排一个单独的线程了, 如果队列很多则开销就比较大了.

3) 添加令牌接口

```
// 通知消费者去消费消息
public void notifyConsume(String queueName) throws InterruptedException {
    tokenQueue.put(queueName);
}
```

4) 实现添加订阅者

- 新来订阅者的时候, 需要先消费掉之前积压的消息.
- consumeMessage 真正的消息消费操作, 一会再实现.

```
public void addConsumer(String consumerTag, String queueName, boolean autoAck, C
    // 消费已经积压的消息消息
    MSGQueue msgQueue = parent.getMemoryDataCenter().getQueue(queueName);
    if (msgQueue == null) {
        throw new MqException("[ConsumerManager] 队列不存在! queueName=" + queueN
    }
    ConsumerEnv consumerEnv = new ConsumerEnv(consumerTag, queueName, autoAck, c
    synchronized (msgQueue) {
        msgQueue.addConsumerEnv(consumerEnv);

        // 把已经积压的 n 个数据都先消费掉
        int n = parent.getMemoryDataCenter().getMessageCount(queueName);
        for (int i = 0; i < n; i++) {
            consumeMessage(msgQueue);
        }
    }
}
```

创建 `ConsumerEnv`,这个类表示一个订阅者的执行环境.

```
// 表示一个消费者的上下文环境
public class ConsumerEnv {
    private String consumerTag;
    private String queueName;
    private boolean autoAck;
    private Consumer consumer;

    public ConsumerEnv(String consumerTag, String queueName, boolean autoAck, Co
        this.consumerTag = consumerTag;
        this.queueName = queueName;
        this.autoAck = autoAck;
        this.consumer = consumer;
    }

    // 省略 getter setter
}
```

给 `MsgQueue` 添加一个订阅者列表.

```
// 该队列被哪些消费者订阅
```

```
private List<ConsumerEnv> consumerEnvList = new ArrayList<>();
// 轮询序号
private AtomicInteger consumerSeq = new AtomicInteger(0);

public void addConsumerEnv(ConsumerEnv consumerEnv) {
    consumerEnvList.add(consumerEnv);
}

public ConsumerEnv chooseConsumer() {
    if (consumerEnvList.size() == 0) {
        return null;
    }
    int index = consumerSeq.get() % consumerEnvList.size();
    consumerSeq.getAndIncrement();
    return consumerEnvList.get(index);
}
```

> 此处的 chooseConsumer 是实现一个轮询效果. 如果一个队列有多个订阅者, 将会按照轮询的方式轮流拿到消息.

5) 实现扫描线程

在 ConsumerManager 中创建一个线程, 不停的尝试扫描令牌队列. 如果拿到了令牌, 就真正触发消费消息操作.

```
public ConsumerManager(VirtualHost parent) {
    this.parent = parent;

    // 启动扫描线程
    Thread scanThread = new Thread(() -> {
        while (true) {
            try {
                // 1. 拿到令牌
                String queueName = tokenQueue.take();
                // 2. 找到队列
                MSGQueue msgQueue = parent.getMemoryDataCenter().getQueue(queueN
                if (msgQueue == null) {
                    throw new MqException("[ConsumerManager] 队列不存在! queueNam
                }
                // 3. 消费一个数据
                synchronized (msgQueue) {
                    consumeMessage(msgQueue);
                }
            } catch (MqException | InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
    }, "scanThread");
    scanThread.start();
}
```

6) 实现消费消息

所谓的消费消息, 其实就是调用消息的回调. 并把消息删除掉.

```
private void consumeMessage(MSGQueue msgQueue) throws MqException {
    // 1. 按照轮询方式, 先找个消费者出来
    ConsumerEnv luckyDog = msgQueue.chooseConsumer();
    if (luckyDog == null) {
        // 如果当前还没有订阅者, 就先暂时不消费.
        return;
    }
    // 2. 从指定队列中取一个元素
    Message message = parent.getMemoryDataCenter().pollMessage(msgQueue.getName(
    if (message == null) {
        return;
    }
    System.out.println("[ConsumerManager] 消息被成功消费! queueName=" + msgQueue.g
    // 3. 丢到线程池中干活. 回调执行时间可能比较长. 不适合让扫描线程去调用.
    workerPool.submit(() -> {
        try {
            // 1. 先把消息放到待确认队列中
            //    (这个逻辑必须放到执行回调前面. 如果是 autoAck false, 在回调内部会调
            parent.getMemoryDataCenter().addMessageWaitAck(msgQueue.getName(), m
            // 2. 调用消费者的回调. 如果回调抛出异常了, 则不会对消息进行任何 ack 操作.
            //    相当于消息仍然处在待消费的状态.
            luckyDog.getConsumer().handleDelivery(luckyDog.getConsumerTag(), mes
            // 3. 如果消息是自动确认, 则可以直接把消息彻底删除了.
            //    (这个逻辑必须放到执行回调后面. 万一执行回调一半服务器崩溃, 这个消息仍
            if (luckyDog.isAutoAck()) {
                //  则修改硬盘上的消息为 "无效". 同时删除内存中的消息
                if (message.getDeliveryMode() == 2) {
                    parent.getDiskDataCenter().deleteMessage(msgQueue, message);
                }
                parent.getMemoryDataCenter().removeMessageWaitAck(msgQueue.getNa
                parent.getMemoryDataCenter().removeMessage(message.getMessageId(
            }
        } catch (MqException | IOException | ClassNotFoundException e) {
```

```
34              e.printStackTrace();
35          }
36      });
37  }
```

注意: 一个队列可能有 N 个消费者, 此处应该按照轮询的方式挑一个消费者进行消费.

小结

一. 消费消息的两种典型情况

1) 订阅者已经存在了, 才发送消息

这种直接获取队列的订阅者, 从中按照轮询的方式挑一个消费者来调用回调即可.

2) 消息先发送到队列了, 订阅者还没到.

此时当订阅者到达, 就快速把指定队列中的消息全都消费掉.

二. 关于消息不丢失的论证

每个消息在从内存队列中出队列时, 都会先进入 待确认 中.

- 如果 autoAck 为 true

消息被消费完毕后(执行完消息回调之后), 再执行清除工作.

分别清除硬盘数据, 待确认队列, 消息中心.

- 如果 autoAck 为 false

在回调内部, 进行清除工作.

分别清除硬盘数据, 待确认队列, 消息中心.

1) 执行消息回调的时候抛出异常

此时消息仍然处在待确认队列中.

此时可以用一个线程扫描待确认队列, 如果发现队列中的消息超时未确认, 则放入死信队列.

> 死信队列咱们此处暂不实现.

2) 执行消息回调的时候服务器宕机

内存所有数据都没了, 但是消息在硬盘上仍然存在. 会在服务下次启动的时候, 加载回内存. 重新被消费到.

消息确认

下列方法只是手动应答的时候才会使用.

应答成功, 则把消息删除掉.

```
public boolean basicAck(String queueName, String messageId) {
    queueName = virtualhostName + queueName;
    try {
        // 删除待 ack 队列中的数据
        memoryDataCenter.removeMessageWaitAck(queueName, messageId);
        // 删除硬盘上的数据
        MSGQueue queue = memoryDataCenter.getQueue(queueName);
        Message message = memoryDataCenter.getMessage(messageId);
        if (message.getDeliveryMode() == 2) {
            diskDataCenter.deleteMessage(queue, message);
        }
        // 删除内存中的数据
        memoryDataCenter.removeMessage(messageId);
        System.out.println("[VirtualHost] basicAck 成功! queueName=" + queueName
        return true;
    } catch (Exception e) {
        System.out.println("[VirtualHost] basicAck 失败! queueName=" + queueName
        e.printStackTrace();
    }
    return false;
}
```

> 对于 RabbitMQ 来说, 还支持否定应答的情况. 此处没有支持. 同学们可以自行尝试实现.

测试 VirtualHost

编写 `VirtualHostTests`

- 操作数据库, 需要先启动 Spring 服务.
- 同时, 需要先关闭 Spring 服务, 才能删除数据库文件
- 使用 `FileUtils.deleteDirector` 递归的删除目录中的内容. 这个是 Spring 自带的类 `org.apache.tomcat.util.http.fileupload.FileUtils`

```
@SpringBootTest
public class VirtualHostTests {
```

```
    private VirtualHost virtualHost = null;

    @BeforeEach
    public void setUp() {
        JavaMessageQueueApplication.ac = SpringApplication.run(JavaMessageQueueA
        virtualHost = new VirtualHost("");
    }

    @AfterEach
    public void tearDown() throws IOException {
        JavaMessageQueueApplication.ac.close();
        File dbFile = new File("meta.db");
        dbFile.delete();
        File dataFile = new File("./data");
        FileUtils.deleteDirectory(dataFile);
    }
}
```

编写测试用例

```
@Test
public void testExchangeDeclare() {
    boolean ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT
    Assertions.assertTrue(ok);
}

@Test
public void testExchangeDelete() {
    boolean ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT
    Assertions.assertTrue(ok);
    ok = virtualHost.exchangeDelete("testExchange");
    Assertions.assertTrue(ok);
}

@Test
public void testQueueDeclare() {
    boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
    Assertions.assertTrue(ok);
}

@Test
public void testQueueDelete() {
    boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
    Assertions.assertTrue(ok);
    ok = virtualHost.queueDelete("testQueue");
```

```
26      Assertions.assertTrue(ok);
27  }
28
29  @Test
30  public void testQueueBind() {
31      boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
32      Assertions.assertTrue(ok);
33      ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
34      Assertions.assertTrue(ok);
35      ok = virtualHost.queueBind("testQueue", "testExchange", "testBindingKey");
36      Assertions.assertTrue(ok);
37  }
38
39  @Test
40  public void testQueueUnbind() {
41      boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
42      Assertions.assertTrue(ok);
43      ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
44      Assertions.assertTrue(ok);
45      ok = virtualHost.queueBind("testQueue", "testExchange", "testBindingKey");
46      Assertions.assertTrue(ok);
47      ok = virtualHost.queueUnbind("testQueue", "testExchange");
48      Assertions.assertTrue(ok);
49  }
50
51  @Test
52  public void testBasicPublic() {
53      boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
54      Assertions.assertTrue(ok);
55      ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
56      Assertions.assertTrue(ok);
57
58      ok = virtualHost.basicPublish("testExchange", "testQueue", null, "hello".get
59      Assertions.assertTrue(ok);
60  }
61
62  // 先订阅消息，后发送消息
63  @Test
64  public void testBasicConsumeDirect1() throws InterruptedException {
65      boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
66      Assertions.assertTrue(ok);
67      ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
68      Assertions.assertTrue(ok);
69
70      ok = virtualHost.basicConsume("testConsumerTag", "testQueue", true, new Cons
71          @Override
72          public void handleDelivery(String consumerTag, BasicProperties propertie
```

```
73              System.out.println("messageId=" + properties.getMessageId());
74              Assertions.assertEquals("testQueue", properties.getRoutingKey());
75              Assertions.assertEquals(1, properties.getDeliveryMode());
76              Assertions.assertArrayEquals("hello".getBytes(), body);
77          }
78      });
79      Assertions.assertTrue(ok);
80
81      ok = virtualHost.basicPublish("testExchange", "testQueue", null, "hello".get
82      Assertions.assertTrue(ok);
83  }
84
85  // 先发送消息，后订阅
86  @Test
87  public void testBasicConsumeDirect2() throws InterruptedException {
88      boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
89      Assertions.assertTrue(ok);
90      ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
91      Assertions.assertTrue(ok);
92
93      ok = virtualHost.basicPublish("testExchange", "testQueue", null, "hello".get
94      Assertions.assertTrue(ok);
95
96      ok = virtualHost.basicConsume("testConsumerTag", "testQueue", true, new Cons
97          @Override
98          public void handleDelivery(String consumerTag, BasicProperties propertie
99              System.out.println("messageId=" + properties.getMessageId());
100             Assertions.assertEquals("testQueue", properties.getRoutingKey());
101             Assertions.assertEquals(1, properties.getDeliveryMode());
102             Assertions.assertArrayEquals("hello".getBytes(), body);
103         }
104     });
105     Assertions.assertTrue(ok);
106     // 保证消费者有足够的时间完成消费
107     Thread.sleep(500);
108 }
109
110 @Test
111 public void testBasicConsumeFanout() throws InterruptedException {
112     boolean ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.FANOUT
113     Assertions.assertTrue(ok);
114
115     ok = virtualHost.queueDeclare("testQueue1", true, false, false, null);
116     Assertions.assertTrue(ok);
117     ok = virtualHost.queueBind("testQueue1", "testExchange", "");
118     Assertions.assertTrue(ok);
119     ok = virtualHost.queueDeclare("testQueue2", true, false, false, null);
```

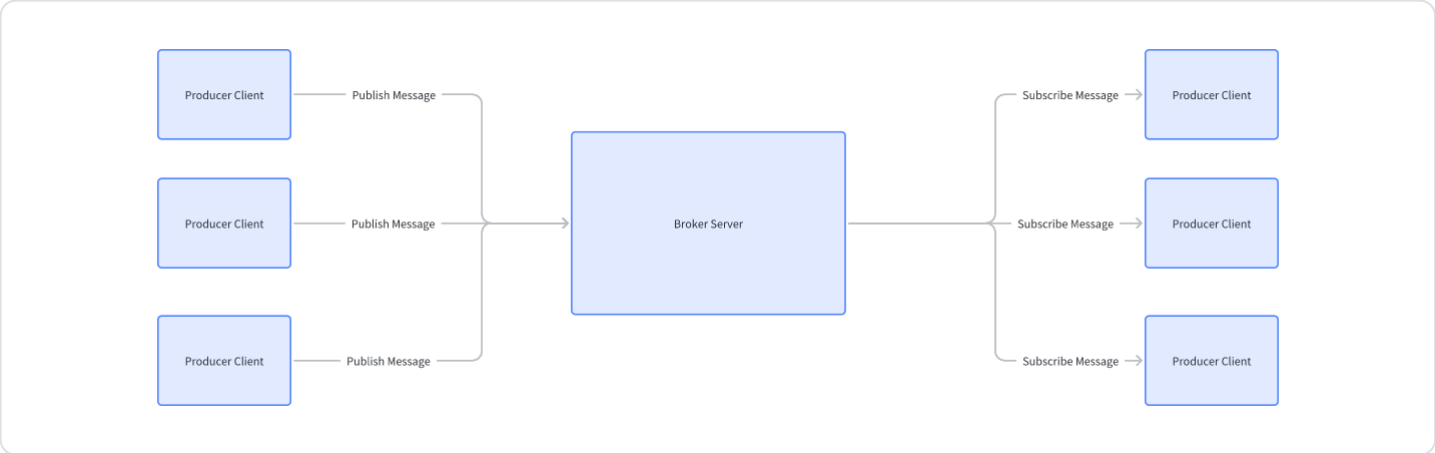
```
120     Assertions.assertTrue(ok);
121     ok = virtualHost.queueBind("testQueue2", "testExchange", "");
122     Assertions.assertTrue(ok);
123 
124     ok = virtualHost.basicPublish("testExchange", "", null, "hello".getBytes());
125     Assertions.assertTrue(ok);
126 
127     ok = virtualHost.basicConsume("testConsumerTag", "testQueue1", false, new Co
128         @Override
129         public void handleDelivery(String consumerTag, BasicProperties propertie
130             System.out.println("messageId=" + properties.getMessageId());
131             Assertions.assertEquals("testQueue1", properties.getRoutingKey());
132             Assertions.assertEquals(1, properties.getDeliveryMode());
133             Assertions.assertArrayEquals("hello".getBytes(), body);
134         }
135     });
136     Assertions.assertTrue(ok);
137 
138     ok = virtualHost.basicConsume("testConsumerTag", "testQueue2", true, new Con
139         @Override
140         public void handleDelivery(String consumerTag, BasicProperties propertie
141             System.out.println("messageId=" + properties.getMessageId());
142             Assertions.assertEquals("testQueue2", properties.getRoutingKey());
143             Assertions.assertEquals(1, properties.getDeliveryMode());
144             Assertions.assertArrayEquals("hello".getBytes(), body);
145         }
146     });
147     Assertions.assertTrue(ok);
148 
149     Thread.sleep(500);
150 }
151 
152 @Test
153 public void testBasicConsumeTopic() throws InterruptedException {
154     boolean ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.TOPIC,
155     Assertions.assertTrue(ok);
156     ok = virtualHost.queueDeclare("testQueue", true, false, false, null);
157     Assertions.assertTrue(ok);
158     ok = virtualHost.queueBind("testQueue", "testExchange", "aaa.*");
159     Assertions.assertTrue(ok);
160 
161     ok = virtualHost.basicPublish("testExchange", "aaa.bbb", null, "hello".getBy
162     Assertions.assertTrue(ok);
163 
164     ok = virtualHost.basicConsume("testConsumerTag", "testQueue", true, new Cons
165         @Override
166         public void handleDelivery(String consumerTag, BasicProperties propertie
```

```
167             System.out.println("messageId=" + properties.getMessageId());
168             Assertions.assertEquals("testQueue", properties.getRoutingKey());
169             Assertions.assertEquals(1, properties.getDeliveryMode());
170             Assertions.assertArrayEquals("hello".getBytes(), body);
171         }
172     });
173     Assertions.assertTrue(ok);
174 
175     Thread.sleep(500);
176 }
177 
178 @Test
179 public void testBasicAck() throws InterruptedException {
180     boolean ok = virtualHost.queueDeclare("testQueue", true, false, false, null)
181     Assertions.assertTrue(ok);
182     ok = virtualHost.exchangeDeclare("testExchange", ExchangeType.DIRECT, true,
183     Assertions.assertTrue(ok);
184 
185     ok = virtualHost.basicPublish("testExchange", "testQueue", null, "hello".get
186     Assertions.assertTrue(ok);
187 
188     Thread.sleep(500);
189 
190     ok = virtualHost.basicConsume("testConsumerTag", "testQueue", false, new Con
191         @Override
192         public void handleDelivery(String consumerTag, BasicProperties propertie
193             System.out.println("messageId=" + properties.getMessageId());
194             Assertions.assertEquals("testQueue", properties.getRoutingKey());
195             Assertions.assertEquals(1, properties.getDeliveryMode());
196             Assertions.assertArrayEquals("hello".getBytes(), body);
197             System.out.println("===================================");
198             // 手动调用 ack
199             Assertions.assertTrue(virtualHost.basicAck("testQueue", properties.g
200         }
201     });
202     Assertions.assertTrue(ok);
203 
204     Thread.sleep(500);
205 }
```

十一. 网络通信协议设计

明确需求

接下来需要考虑客户端和服务器之间的通信. 回顾交互模型.



生产者和消费者都是客户端, 都需要通过网络和 Broker Server 进行通信.

此处我们使用 TCP 协议, 来作为通信的底层协议. 同时在这个基础上自定义应用层协议, 完成客户端对服务器这边功能的远程调用.

要调用的功能有:

- 创建 channel
- 关闭 channel
- 创建 exchange
- 删除 exchange
- 创建 queue
- 删除 queue
- 创建 binding
- 删除 binding
- 发送 message
- 订阅 message
- 发送 ack
- 返回 message (服务器 -> 客户端)

设计应用层协议

使用二进制的方式设定协议.

> 因为 Message 的消息体本身就是二进制的. 因此不太方便使用 json 等文本格式的协议.

请求:

响应:

其中 type 表示请求响应不同的功能. 取值如下:

- `0x1` 创建 channel
- `0x2` 关闭 channel
- `0x3` 创建 exchange
- `0x4` 销毁 exchange
- `0x5` 创建 queue
- `0x6` 销毁 queue
- `0x7` 创建 binding
- `0x8` 销毁 binding
- `0x9` 发送 message
- `0xa` 订阅 message
- `0xb` 返回 ack
- `0xc` 服务器给客户端推送的消息. (被订阅的消息) 响应独有的.

其中 payload 部分, 会根据不同的 type, 存在不同的格式.

对于请求来说, payload 表示这次方法调用的各种参数信息.

对于响应来说, payload 表示这次方法调用的返回值.

定义 Request / Response

创建 `common.Request`

```
1 public class Request {
2     private int type;
3     private int length;
4     private byte[] payload;
5     // 省略 getter setter
6 }
```

创建 `common.Response`

```
1 public class Response {
2     private int type;
3     private int length;
4     private byte[] payload;
5     // 省略 getter setter
6 }
```

定义参数父类

构造一个类表示方法的参数, 作为 Request 的 payload.

不同的方法中, 参数形态各异, 但是有些信息是通用的, 使用一个父类表示出来. 具体每个方法的参数再通过继承的方式体现.

`common.BasicArguments`

```
1 public class BaseArguments implements Serializable {
2     // 表示一次请求/响应的唯一 id. 用来把响应和请求对上.
3     protected String rid;
4     protected String channelId;
5
6     // 省略 getter setter
7 }
```

- 此处的 rid 和 channelId 都是基于 UUID 来生成的. rid 用来标识一个请求-响应. 这一点在请求响应比较多的时候非常重要.

定义返回值父类

和参数同理, 也需要构造一个类表示返回值, 作为 Response 的 payload.

`common.BasicReturns`

```
1 public class BaseReturns implements Serializable {
2     // 表示一次请求/响应的唯一 id. 用来把响应和请求对上.
3     protected String rid;
4     protected String channelId;
5     protected boolean ok;
6
7     // 省略 getter setter
8 }
```

定义其他参数类

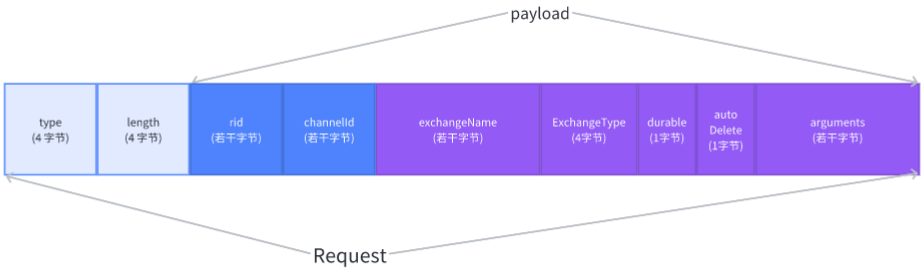
针对每个 VirtualHost 提供的方法, 都需要有一个类表示对应的参数.

1) ExchangeDeclareArguments

```
1 public class ExchangeDeclareArguments extends BaseArguments implements Serializa
2     private String exchangeName;
3     private ExchangeType exchangeType;
4     private boolean durable;
5     private boolean autoDelete;
6     private Map<String, Object> arguments;
7 }
```

一个创建交换机的请求, 形如:

- 可以把 ExchangeDeclareArguments 转成 byte[], 就得到了下列图片的结构.
- 按照 length 长度读取出 payload, 就可以把读到的二进制数据转换成 ExchangeDeclareArguments 对象.



> 后续请求报文格式同理, 就不再重复画了.

2) ExchangeDeleteArguments

```
1 public class ExchangeDeleteArguments extends BaseArguments implements Serializab
2     private String exchangeName;
3 }
```

3) QueueDeclareArguments

```
1 public class QueueDeclareArguments extends BaseArguments implements Serializable
2     private String queueName;
3     private boolean durable;
4     private boolean exclusive;
5     private boolean autoDelete;
6     private Map<String, Object> arguments;
7 }
```

4) QueueDeleteArguments

```
1 public class QueueDeleteArguments extends BaseArguments implements Serializable
2     private String queueName;
3 }
```

5) QueueBindArguments

```
public class QueueBindArguments extends BaseArguments implements Serializable {
    private String queueName;
    private String exchangeName;
    private String bindingKey;
}
```

6) QueueUnbindArguments

```
public class QueueUnbindArguments extends BaseArguments implements Serializable
    private String queueName;
    private String exchangeName;
}
```

7) BasicPublishArguments

```
public class BasicPublishArguments extends BaseArguments implements Serializable
    private String exchangeName;
    private String routingKey;
    private BasicProperties basicProperties;
    private byte[] body;
}
```

8) BasicConsumeArguments

```
public class BasicConsumeArguments extends BaseArguments implements Serializable
    private String consumeTag;
    private String queueName;
    private boolean autoAck;
}
```

9) SubScribeReturns

- 这个不是参数, 是返回值. 是服务器给消费者推送的订阅消息.
- consumerTag 其实是 channelId.

- basicProperties 和 body 共同构成了 Message.

```
public class SubScribeReturns extends BaseReturns implements Serializable {
    private String consumerTag;
    private BasicProperties basicProperties;
    private byte[] body;
}
```

十二. 实现 BrokerServer

创建 BrokerServer 类

```
public class BrokerServer {
    // 当前程序只考虑一个虚拟主机的情况.
    private VirtualHost virtualHost = new VirtualHost("default-VirtualHost");
    // key 为 channelId, value 为 channel 对应的 socket 对象.
    private ConcurrentHashMap<String, Socket> sessions = new ConcurrentHashMap<>

    private ServerSocket serverSocket;
    private ExecutorService executorService;
    private volatile boolean runnable = true;
}
```

- virtualHost 表示服务器持有的虚拟主机. 队列, 交换机, 绑定, 消息都是通过虚拟主机管理.
- sessions 用来管理所有的客户端的连接. 记录每个客户端的 socket.
- serverSocket 是服务器自身的 socket
- executorService 这个线程池用来处理响应.
- runnable 这个标志位用来控制服务器的运行停止.

启动/停止服务器

- 这里就是一个单纯的 TCP 服务器, 没啥特别的.
- 实现停止操作, 主要是为了方便后续开展单元测试.

```
public BrokerServer(int port) throws IOException {
    serverSocket = new ServerSocket(port);
```

```
}

public void start() throws IOException {
    System.out.println("[BrokerServer] 启动完成!");
    executorService = Executors.newCachedThreadPool();
    try {
        while (runnable) {
            Socket clientSocket = serverSocket.accept();
            executorService.submit(() -> processConnection(clientSocket));
        }
    } catch (SocketException e) {
        System.out.println("[BrokerServer] 服务器关闭完成!");
    }
}

public void stop() throws IOException {
    runnable = false;
    // 立即结束所有的线程池的任务
    executorService.shutdownNow();
    serverSocket.close();
}
```

实现处理连接

- 对于 EOFException 和 SocketException , 我们视为客户端正常断开连接.
  - 如果是客户端先 close, 后调用 DataInputStream 的 read, 则抛出 EOFException
  - 如果是先调用 DataInputStream 的 read, 后客户端调用 close, 则抛出 SocketException

```
private void processConnection(Socket clientSocket) {
    try (InputStream inputStream = clientSocket.getInputStream();
         OutputStream outputStream = clientSocket.getOutputStream()) {
        DataInputStream dataInputStream = new DataInputStream(inputStream);
        DataOutputStream dataOutputStream = new DataOutputStream(outputStream);
        while (true) {
            Request request = readRequest(dataInputStream);
            Response response = process(request, clientSocket);
            writeResponse(dataOutputStream, response);
        }
    } catch (EOFException | SocketException e) {
        System.out.println("[BrokerServer] connection 关闭! serverIP=" + clientS
                + ", port=" + clientSocket.getPort());
    } catch (MqException | IOException | ClassNotFoundException e) {
```

```
        System.out.println("[BrokerServer] connection 出现异常!");
        e.printStackTrace();
    } finally {
        try {
            clientSocket.close();
            // 对 sessions 进行清理
            clearClosedSession(clientSocket);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

实现 readRequest

```
private Request readRequest(DataInputStream dataInputStream) throws IOException
    Request request = new Request();
    request.setType(dataInputStream.readInt());
    request.setLength(dataInputStream.readInt());
    byte[] payload = new byte[request.getLength()];
    int n = dataInputStream.read(payload);
    if (n != request.getLength()) {
        throw new IOException("读取请求数据出错!");
    }
    request.setPayload(payload);
    return request;
}
```

实现 writeResponse

- 注意这里的 flush 操作很关键, 否则响应不一定能及时返回给客户端.

```
private void writeResponse(DataOutputStream dataOutputStream, Response response)
    dataOutputStream.writeInt(response.getType());
    dataOutputStream.writeInt(response.getLength());
    dataOutputStream.write(response.getPayload());
    dataOutputStream.flush();
}
```

实现处理请求

- 先把请求转换成 BaseArguments , 获取到其中的 channelId 和 rid
- 再根据不同的 type, 分别处理不同的逻辑. (主要是调用 virtualHost 中不同的方法).
- 针对消息订阅操作, 则需要在存在消息的时候通过回调, 把响应结果写回给对应的客户端.
- 最后构造成统一的响应.

```
private Response process(Request request, Socket clientSocket) throws MqExceptio
    // 1. 从 request 中解析出业务请求
    BaseArguments baseArguments = (BaseArguments) BinaryTool.fromBytes(request.g
    System.out.println("[Request] rid=" + baseArguments.getRid() + ", channelId=
            + ", type=" + request.getType() + ", length=" + request.getLength())
    // 2. 根据 type 来区分业务分支.
    boolean ok = true;
    if (request.getType() == 0x1) {
        // 创建 channel
        sessions.put(baseArguments.getChannelId(), clientSocket);
        System.out.println("[BrokerServer] 创建 channel 完成! channelId=" + baseA
    } else if (request.getType() == 0x2) {
        // 销毁 channel
        sessions.remove(baseArguments.getChannelId());
        System.out.println("[BrokerServer] 销毁 channel 完成! channelId=" + baseA
    } else if (request.getType() == 0x3) {
        // 创建交换机
        ExchangeDeclareArguments exchangeDeclareArguments = (ExchangeDeclareArgu
        ok = virtualHost.exchangeDeclare(exchangeDeclareArguments.getExchangeNam
                exchangeDeclareArguments.isDurable(), exchangeDeclareArguments.i
    } else if (request.getType() == 0x4) {
        // 删除交换机
        ExchangeDeleteArguments exchangeDeleteArguments = (ExchangeDeleteArgumen
        ok = virtualHost.exchangeDelete(exchangeDeleteArguments.getExchangeName(
    } else if (request.getType() == 0x5) {
        // 创建队列
        QueueDeclareArguments queueDeclareArguments = (QueueDeclareArguments) ba
        ok = virtualHost.queueDeclare(queueDeclareArguments.getQueueName(), queu
                queueDeclareArguments.isAutoDelete(), queueDeclareArguments.getA
    } else if (request.getType() == 0x6) {
        // 删除队列
        QueueDeleteArguments queueDeleteArguments = (QueueDeleteArguments) baseA
        ok = virtualHost.queueDelete(queueDeleteArguments.getQueueName());
    } else if (request.getType() == 0x7) {
        // 创建绑定
        QueueBindArguments queueBindArguments = (QueueBindArguments) baseArgumen
```

```
37          ok = virtualHost.queueBind(queueBindArguments.getQueueName(), queueBindA
38      } else if (request.getType() == 0x8) {
39          // 解除绑定
40          QueueUnbindArguments queueUnbindArguments = (QueueUnbindArguments) baseA
41          ok = virtualHost.queueUnbind(queueUnbindArguments.getQueueName(), queueU
42      } else if (request.getType() == 0x9) {
43          // 发送消息
44          BasicPublishArguments basicPublishArguments = (BasicPublishArguments) ba
45          ok = virtualHost.basicPublish(basicPublishArguments.getExchangeName(), b
46                  basicPublishArguments.getBasicProperties(), basicPublishArgument
47      } else if (request.getType() == 0xa) {
48          // 订阅消息
49          BasicConsumeArguments basicConsumeArguments = (BasicConsumeArguments) ba
50          // 创建个回调，用来把消费的数据转发回客户端.
51          ok = virtualHost.basicConsume(basicConsumeArguments.getConsumeTag(), bas
52                  basicConsumeArguments.isAutoAck(), new Consumer() {
53                      @Override
54                      public void handleDelivery(String consumerTag, BasicProperti
55                          // 1. 根据 channelId 找到对应的 socket
56                          Socket clientSocket = sessions.get(basicConsumeArguments
57                          if (clientSocket == null || clientSocket.isClosed()) {
58                              throw new MqException("[BrokerServer] 订阅消息的客户端
59                          }
60                          // 2. 构造响应数据
61                          SubScribeReturns subScribeReturns = new SubScribeReturns
62                          subScribeReturns.setChannelId(basicConsumeArguments.getC
63                          subScribeReturns.setConsumerTag(consumerTag);
64                          subScribeReturns.setBasicProperties(properties);
65                          subScribeReturns.setBody(body);
66                          byte[] payload = BinaryTool.toBytes(subScribeReturns);
67                          // 3. 写入到对应 socket 中
68                          Response response = new Response();
69                          response.setType(0xc);
70                          response.setLength(payload.length);
71                          response.setPayload(payload);
72                          // 此处不应该关闭 DataOutputStream, 关闭这个会导致内部持有的
73                          DataOutputStream dataOutputStream = new DataOutputStream
74                          writeResponse(dataOutputStream, response);
75                      }
76                  });
77      } else if (request.getType() == 0xb) {
78          // 确认 ack
79          BasicAckArguments basicAckArguments = (BasicAckArguments) baseArguments;
80          ok = virtualHost.basicAck(basicAckArguments.getQueueName(), basicAckArgu
81      } else {
82          throw new MqException("[BrokerServer] 未知的请求 type ! type=" + request.
83      }
```

```
    // 3. 构造响应.
    BaseReturns baseReturns = new BaseReturns();
    baseReturns.setRid(baseArguments.getRid());
    baseReturns.setChannelId(baseArguments.getChannelId());
    baseReturns.setOk(ok);

    byte[] payload = BinaryTool.toBytes(baseReturns);
    Response response = new Response();
    response.setType(request.getType());
    response.setLength(payload.length);
    response.setPayload(payload);
    System.out.println("[Response] rid=" + baseReturns.getRid() + ", channelId="
            + ", type=" + response.getType() + ", length=" + response.getLength(
    return response;
}
```

实现 clearClosedSession

- 如果客户端只关闭了 Connection, 没关闭 Connection 中包含的 Channel, 也没关系, 在这里统一进行清理.
- 注意迭代器失效问题.

```
private void clearClosedSession(Socket clientSocket) {
    // 这里不要一个循环, 同时遍历 + 删除. 否则可能有迭代器失效问题.
    List<String> toDeleteChannelId = new ArrayList<>();
    for (Map.Entry<String, Socket> entry : sessions.entrySet()) {
        if (entry.getValue() == clientSocket) {
            toDeleteChannelId.add(entry.getKey());
        }
    }
    for (String channelId : toDeleteChannelId) {
        sessions.remove(channelId);
    }
    System.out.println("[BrokerServer] 清理 session 完成! 被清理的 channelId=" + t
}
```

十三. 实现客户端

创建包 `mqclient`

创建 ConnectionFactory

用来创建连接的工厂类.

- 当前没有实现用户认证和多虚拟主机, 用户名密码可以暂时先不要.

```
1  public class ConnectionFactory {
2      // BrokerServer 的 ip 和 port
3      private String host;
4      private int port;
5      // 这几个部分暂时不加.
6  //    private String virtualHost;
7  //    private String username;
8  //    private String password;
9
10     // 建立一个 tcp 连接
11     public Connection newConnection() throws IOException {
12         Connection connection = new Connection(host, port);
13         return connection;
14     }
15 }
```

Connection 和 Channel 的定义

一个客户端可以创建多个 Connection.

一个 Connection 对应一个 socket, 一个 TCP 连接.

一个 Connection 可以包含多个 Channel

1) Connection 的定义

```
1  public class Connection {
2      private Socket socket;
3      private InputStream inputStream;
4      private OutputStream outputStream;
5      private DataInputStream dataInputStream;
6      private DataOutputStream dataOutputStream;
7      // 记录当前 Connection 包含的 Channel
8      private ConcurrentHashMap<String, Channel> channelMap = new ConcurrentHashMa
9      // 执行消费消息回调的线程池
```

```
    private ExecutorService callbackPool = Executors.newFixedThreadPool(4);
}
```

- Socket 是客户端持有的套接字. InputStream OutputStream DataInputStream DataOutputStream 均为 socket 通信的接口.
- channelMap 用来管理该连接中所有的 Channel.
- callbackPool 是用来在客户端这边执行用户回调的线程池.

2) Channel 的定义

```
public class Channel {
    private String channelId;
    private Connection connection;
    // key 为 rid, 即 requestId / responseId.
    private ConcurrentHashMap<String, BaseReturns> baseReturnsMap = new Concurre
    // 订阅消息的回调
    private Consumer consumer = null;

    public Channel(String channelId, Connection connection) {
        this.channelId = channelId;
        this.connection = connection;
    }
}
```

- channelId 为 channel 的身份标识, 使用 UUID 标识.
- Connection 为 channel 对应的连接.
- baseReturnsMap 用来保存响应的返回值. 放到这个哈希表中方便和请求匹配.
- consumer 为消费者的回调(用户注册的). 对于消息响应, 应该调用这个回调处理消息.

封装请求响应读写操作

在 Connection 中, 实现下列方法

```
// 读取响应应该在另外一个单独的线程中完成.
public void writeRequest(Request request) throws IOException {
    dataOutputStream.writeInt(request.getType());
    dataOutputStream.writeInt(request.getLength());
```

```
    dataOutputStream.write(request.getPayload());
    dataOutputStream.flush();
    System.out.println("[Connection] 发送请求! type=" + request.getType() + ", le
}
```

```
public Response readResponse() throws IOException {
    Response response = new Response();
    response.setType(dataInputStream.readInt());
    response.setLength(dataInputStream.readInt());
    byte[] payload = new byte[response.getLength()];
    int n = dataInputStream.read(payload);
    if (n != response.getLength()) {
        throw new IOException("读取到的响应数据不完整!");
    }
    response.setPayload(payload);
    System.out.println("[Connection] 收到响应! type=" + response.getType() + ", l
    return response;
}
```

创建 channel

在 Connection 中, 定义下列方法来创建一个 channel

```
public Channel createChannel() throws IOException {
    // 使用 UUID 生产 channelId, 以 C- 开头
    String channelId = "C-" + UUID.randomUUID().toString();
    Channel channel = new Channel(channelId, this);
    // 这里需要先把 channel 键值对放到 Map 中. 否则后续 createChannel 的阻塞等待就等不
    channelMap.put(channelId, channel);
    boolean ok = channel.createChannel();
    if (!ok) {
        // 服务器返回创建 channel 失败!
        // 把 channelId 删除掉即可
        channelMap.remove(channelId);
        return null;
    }
    return channel;
}
```

发送请求

通过 Channel 提供请求的发送操作.

1) 创建 channel

```
public boolean createChannel() throws IOException {
    BaseArguments baseArguments = new BaseArguments();
    baseArguments.setRid(generateRid());
    baseArguments.setChannelId(channelId);
    byte[] payload = BinaryTool.toBytes(baseArguments);
    Request request = new Request();
    request.setType(0x1);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);
    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(baseArguments.getRid());
    return baseReturns.isOk();
}
```

generateRid 的实现

```
private String generateRid() {
    return "R-" + UUID.randomUUID().toString();
}
```

waitResult 的实现

- 由于服务器的响应是异步的. 此处通过 waitResult 实现同步等待的效果.

```
private BaseReturns waitResult(String rid){
    BaseReturns baseReturns = null;
    while ((baseReturns = baseReturnsMap.get(rid)) == null) {
        synchronized (this) {
            try {
                wait();
            } catch (InterruptedException e) {
```

```
                // 如果 wait 被提前唤醒, 也应该继续循环.
                // 所以这里啥都不干, 但是 try 需要放到 while 内部.
                e.printStackTrace();
            }
        }
    }
    return baseReturns;
}
```

2) 关闭 channel

```
public boolean close() throws IOException {
    // 删除服务器上的 channel. 如果不显式调用, 也没关系. 服务器会在 Connection 断开的时
    BaseArguments baseArguments = new BaseArguments();
    baseArguments.setRid(generateRid());
    baseArguments.setChannelId(channelId);
    byte[] payload = BinaryTool.toBytes(baseArguments);
    Request request = new Request();
    request.setType(0x2);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);
    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(baseArguments.getRid());
    return baseReturns.isOk();
}
```

3) 创建交换机

```
public boolean exchangeDeclare(String exchangeName, ExchangeType exchangeType, b
                               Map<String, Object> arguments) throws IOException {
    ExchangeDeclareArguments exchangeDeclareArguments = new ExchangeDeclareArgum
    exchangeDeclareArguments.setRid(generateRid());
    exchangeDeclareArguments.setChannelId(channelId);
    exchangeDeclareArguments.setExchangeName(exchangeName);
    exchangeDeclareArguments.setExchangeType(exchangeType);
    exchangeDeclareArguments.setDurable(durable);
    exchangeDeclareArguments.setAutoDelete(autoDelete);
    exchangeDeclareArguments.setArguments(arguments);
```

```
    byte[] payload = BinaryTool.toBytes(exchangeDeclareArguments);

    Request request = new Request();
    request.setType(0x3);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(exchangeDeclareArguments.getRid());
    return baseReturns.isOk();
}
```

4) 删除交换机

```
public boolean exchangeDelete(String exchangeName) throws IOException {
    ExchangeDeleteArguments exchangeDeleteArguments = new ExchangeDeleteArgument
    exchangeDeleteArguments.setRid(generateRid());
    exchangeDeleteArguments.setChannelId(channelId);
    exchangeDeleteArguments.setExchangeName(exchangeName);
    byte[] payload = BinaryTool.toBytes(exchangeDeleteArguments);

    Request request = new Request();
    request.setType(0x4);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(exchangeDeleteArguments.getRid());
    return baseReturns.isOk();
}
```

5) 创建队列

```
public boolean queueDeclare(String queueName, boolean durable, boolean exclusive
                            Map<String, Object> arguments) throws IOException {
    QueueDeclareArguments queueDeclareArguments = new QueueDeclareArguments();
    queueDeclareArguments.setRid(generateRid());
```

```
    queueDeclareArguments.setChannelId(channelId);
    queueDeclareArguments.setQueueName(queueName);
    queueDeclareArguments.setDurable(durable);
    queueDeclareArguments.setExclusive(exclusive);
    queueDeclareArguments.setAutoDelete(autoDelete);
    queueDeclareArguments.setArguments(arguments);
    byte[] payload = BinaryTool.toBytes(queueDeclareArguments);

    Request request = new Request();
    request.setType(0x5);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(queueDeclareArguments.getRid());
    return baseReturns.isOk();
}
```

6) 删除队列

```
public boolean queueDelete(String queueName) throws IOException {
    QueueDeleteArguments queueDeleteArguments = new QueueDeleteArguments();
    queueDeleteArguments.setRid(generateRid());
    queueDeleteArguments.setChannelId(channelId);
    queueDeleteArguments.setQueueName(queueName);
    byte[] payload = BinaryTool.toBytes(queueDeleteArguments);

    Request request = new Request();
    request.setType(0x6);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(queueDeleteArguments.getRid());
    return baseReturns.isOk();
}
```

7) 创建绑定

```java
// 对于直接交换机和 fanout 交换机, bindingKey 不生效. 直接设为 "" 即可
public boolean queueBind(String queueName, String exchangeName) throws IOExcepti
    return queueBind(queueName, exchangeName, "");
}

public boolean queueBind(String queueName, String exchangeName, String bindingKe
    QueueBindArguments queueBindArguments = new QueueBindArguments();
    queueBindArguments.setRid(generateRid());
    queueBindArguments.setChannelId(channelId);
    queueBindArguments.setQueueName(queueName);
    queueBindArguments.setExchangeName(exchangeName);
    queueBindArguments.setBindingKey(bindingKey);
    byte[] payload = BinaryTool.toBytes(queueBindArguments);

    Request request = new Request();
    request.setType(0x7);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(queueBindArguments.getRid());
    return baseReturns.isOk();
}
```

8) 删除绑定

```java
public boolean queueUnbind(String queueName, String exchangeName) throws IOExcep
    QueueUnbindArguments queueUnbindArguments = new QueueUnbindArguments();
    queueUnbindArguments.setRid(generateRid());
    queueUnbindArguments.setChannelId(channelId);
    queueUnbindArguments.setQueueName(queueName);
    queueUnbindArguments.setExchangeName(exchangeName);
    byte[] payload = BinaryTool.toBytes(queueUnbindArguments);

    Request request = new Request();
    request.setType(0x8);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
```

```
    BaseReturns baseReturns = waitResult(queueUnbindArguments.getRid());
    return baseReturns.isOk();
}
```

9) 发送消息

```
public boolean basicPublish(String exchangeName, String routingKey, BasicPropert
    BasicPublishArguments basicPublishArguments = new BasicPublishArguments();
    basicPublishArguments.setRid(generateRid());
    basicPublishArguments.setChannelId(channelId);
    basicPublishArguments.setExchangeName(exchangeName);
    basicPublishArguments.setRoutingKey(routingKey);
    basicPublishArguments.setBasicProperties(basicProperties);
    basicPublishArguments.setBody(body);
    byte[] payload = BinaryTool.toBytes(basicPublishArguments);

    Request request = new Request();
    request.setType(0x9);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(basicPublishArguments.getRid());
    return baseReturns.isOk();
}
```

10) 订阅消息

```
public boolean basicConsume(String queueName, boolean autoAck, Consumer consumer
    BasicConsumeArguments basicConsumeArguments = new BasicConsumeArguments();
    basicConsumeArguments.setRid(generateRid());
    basicConsumeArguments.setChannelId(channelId);
    basicConsumeArguments.setQueueName(queueName);
    basicConsumeArguments.setAutoAck(autoAck);
    basicConsumeArguments.setConsumeTag(channelId);
    byte[] payload = BinaryTool.toBytes(basicConsumeArguments);

    Request request = new Request();
```

```
    request.setType(0xa);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);

    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(basicConsumeArguments.getRid());
    if (baseReturns.isOk()) {
        // 设置回调
        if (this.consumer != null) {
            throw new MqException("该 channel 已经设置过消费回调, 不能重复设置!");
        }
        this.consumer = consumer;
    }
    return baseReturns.isOk();
}
```

11) 确认消息

```
public boolean basicAck(String queueName, String messageId) throws IOException {
    BasicAckArguments basicAckArguments = new BasicAckArguments();
    basicAckArguments.setRid(generateRid());
    basicAckArguments.setChannelId(channelId);
    basicAckArguments.setQueueName(queueName);
    basicAckArguments.setMessageId(messageId);
    byte[] payload = BinaryTool.toBytes(basicAckArguments);

    Request request = new Request();
    request.setType(0xb);
    request.setLength(payload.length);
    request.setPayload(payload);
    connection.writeRequest(request);
    // 阻塞等待服务器的响应
    BaseReturns baseReturns = waitResult(basicAckArguments.getRid());
    return baseReturns.isOk();
}
```

小结

上述发送请求的操作, 逻辑基本一致. 构造参数 + 构造请求 + 发送 + 等待结果.

处理响应

1) 创建扫描线程

创建一个扫描线程, 用来不停的读取 socket 中的响应数据.

注意: 一个 Connection 中可能包含多个 channel, 需要把响应分别放到对应的 channel 中.

```
public Connection(String host, int port) throws IOException {
    socket = new Socket(host, port);
    inputStream = socket.getInputStream();
    outputStream = socket.getOutputStream();
    dataInputStream = new DataInputStream(inputStream);
    dataOutputStream = new DataOutputStream(outputStream);

    // 创建一个读响应的线程
    Thread t = new Thread(() -> {
        try {
            while (!socket.isClosed()) {
                Response response = readResponse();
                dispatchResponse(response);
            }
        } catch (SocketException e) {
            // 连接断开, 忽略该异常.
            // System.out.println("[Connection] 连接断开!");
        } catch (IOException | ClassNotFoundException | MqException e) {
            System.out.println("[Connection] 连接出现异常!");
            e.printStackTrace();
        }
    });
    t.start();
}
```

2) 实现响应的分发

给 Connection 创建 dispatchResponse 方法.

- 针对服务器返回的控制响应和消息响应, 分别处理.
  - 如果是订阅数据, 则调用 channel 中的回调.
  - 如果是控制消息, 直接放到结果集合中.

```java
private void dispatchResponse(Response response) throws IOException, ClassNotFou
    if (response.getType() == 0xc) {
        // 1. 解析到服务器返回的订阅数据
        SubScribeReturns subScribeReturns = (SubScribeReturns) BinaryTool.fromBy
        // 2. 获取到 channel
        Channel channel = channelMap.get(subScribeReturns.getChannelId());
        if (channel == null) {
            throw new MqException("该消息对应的 channel 不存在! channelId=" + subS
        }
        // 3. 执行 channel 中对应的回调.
        callbackPool.submit(() -> {
            try {
                channel.getConsumer().handleDelivery(subScribeReturns.getConsume
                        subScribeReturns.getBasicProperties(), subScribeReturns.
            } catch (MqException | IOException e) {
                e.printStackTrace();
            }
        });
    } else {
        // 1. 拿到服务器返回的控制消息
        BaseReturns baseReturns = (BaseReturns) BinaryTool.fromBytes(response.ge
        // System.out.printf("[Connection] 收到响应: type=0x%x, channelId=%s, ok=
        //         baseReturns.getChannelId(), baseReturns.isOk());
        // 2. 找到对应 Channel
        Channel channel = channelMap.get(baseReturns.getChannelId());
        if (channel == null) {
            // 这个是小问题, 不要抛异常
            System.out.println("[Connection] channel 不存在! channelId=" + baseRe
            return;
        }
        // 3. 把响应放到对应的 Channel 的 map 中.
        channel.putReturns(baseReturns);
    }
}
```

3) 实现 channel.putReturns

把响应放到响应的 hash 表中, 同时唤醒等待响应的线程去消费.

```java
public void putReturns(BaseReturns baseReturns) {
    baseReturnsMap.put(baseReturns.getRid(), baseReturns);
    synchronized (this) {
        // 这里要唤醒所有等待的线程, 不能只唤醒一个.
```

```
        notifyAll();
    }
}
```

关闭 Connection

给 Connection 实现 close 方法

```
public void close() {
    try {
        callbackPool.shutdown();
        channelMap = null;
        inputStream.close();
        outputStream.close();
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

测试客户端-服务器

创建 `MqClientTests`

```
public class MqClientTests {
    private BrokerServer brokerServer = null;
    private Thread t = null;

    private ConnectionFactory factory = null;

    @BeforeEach
    public void setUp() throws IOException {
        JavaMessageQueueApplication.ac = SpringApplication.run(JavaMessageQueueA
        t = new Thread(() -> {
            try {
                brokerServer = new BrokerServer(9090);
                brokerServer.start();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        });
        t.start();

        factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(9090);
    }

    @AfterEach
    public void tearDown() throws IOException, InterruptedException {
        // 结束服务器
        brokerServer.stop();
        // 等待线程结束
        t.join();
        // 关闭 Spring 服务器
        JavaMessageQueueApplication.ac.close();
        // 删除服务器的数据文件
        File dbFile = new File("meta.db");
        dbFile.delete();
        // 删除数据文件
        File dataFile = new File("./data");
        FileUtils.deleteDirectory(dataFile);

        factory = null;
    }
}
```

编写测试用例

```
@Test
public void testConnection() throws IOException {
    Connection connection = factory.newConnection();
    Assertions.assertNotNull(connection);
}

@Test
public void testChannel() throws IOException {
    Connection connection = factory.newConnection();
    Assertions.assertNotNull(connection);
    Channel channel = connection.createChannel();
    Assertions.assertNotNull(channel);
}

@Test
public void testExchange() throws IOException, InterruptedException {
```

```
17      Connection connection = factory.newConnection();
18      Assertions.assertNotNull(connection);
19      Channel channel = connection.createChannel();
20      Assertions.assertNotNull(channel);
21
22      boolean ok = channel.exchangeDeclare("testExchange", ExchangeType.DIRECT, tr
23      Assertions.assertTrue(ok);
24      ok = channel.exchangeDelete("testExchange");
25      Assertions.assertTrue(ok);
26
27      channel.close();
28      connection.close();
29  }
30
31  @Test
32  public void testQueue() throws IOException {
33      Connection connection = factory.newConnection();
34      Assertions.assertNotNull(connection);
35      Channel channel = connection.createChannel();
36      Assertions.assertNotNull(channel);
37
38      boolean ok = channel.queueDeclare("testQueue", true, false, false, null);
39      Assertions.assertTrue(ok);
40      ok = channel.queueDelete("testQueue");
41      Assertions.assertTrue(ok);
42
43      channel.close();
44      connection.close();
45  }
46
47  @Test
48  public void testBind() throws IOException {
49      Connection connection = factory.newConnection();
50      Assertions.assertNotNull(connection);
51      Channel channel = connection.createChannel();
52      Assertions.assertNotNull(channel);
53
54      boolean ok = channel.exchangeDeclare("testExchange", ExchangeType.DIRECT, tr
55      Assertions.assertTrue(ok);
56      ok = channel.queueDeclare("testQueue", true, false, false, null);
57      Assertions.assertTrue(ok);
58
59      ok = channel.queueBind("testQueue", "testExchange");
60      Assertions.assertTrue(ok);
61
62      ok = channel.queueUnbind("testQueue", "testExchange");
63      Assertions.assertTrue(ok);
```

```
64
65      channel.close();
66      connection.close();
67  }
68
69  @Test
70  public void testMessageDirect() throws IOException, MqException, InterruptedExce
71      Connection connection = factory.newConnection();
72      Assertions.assertNotNull(connection);
73      Channel channel = connection.createChannel();
74      Assertions.assertNotNull(channel);
75
76      boolean ok = channel.exchangeDeclare("testExchange", ExchangeType.DIRECT, tr
77      Assertions.assertTrue(ok);
78      ok = channel.queueDeclare("testQueue", true, false, false, null);
79      Assertions.assertTrue(ok);
80
81      byte[] requestBody = "hello".getBytes();
82      // DIRECT 模式, routingKey 就是队列名字
83      // 发送的时候 basicProperties 可以是空着的. 服务器会进行构造. 订阅者收到的消息则是有
84      ok = channel.basicPublish("testExchange", "testQueue", null, requestBody);
85      Assertions.assertTrue(ok);
86
87      ok = channel.basicConsume("testQueue", true, new Consumer() {
88          @Override
89          public void handleDelivery(String consumerTag, BasicProperties propertie
90              System.out.println("[消费数据] 开始!");
91              System.out.println("consumerTag=" + consumerTag);
92              System.out.println("properties=" + properties);
93              String bodyString = new String(responseBody, 0, responseBody.length)
94              System.out.println("body=" + bodyString);
95
96              Assertions.assertEquals(requestBody, responseBody);
97          }
98      });
99      Assertions.assertTrue(ok);
100     // 等待数据消费完.
101     Thread.sleep(500);
102
103     channel.close();
104     connection.close();
105 }
106
107 @Test
108 public void testMessageFanout() throws IOException, MqException, InterruptedExce
109     Connection connection = factory.newConnection();
110     Assertions.assertNotNull(connection);
```

```
111     Channel channel1 = connection.createChannel();
112     Assertions.assertNotNull(channel1);
113 
114     boolean ok = channel1.exchangeDeclare("testExchange", ExchangeType.FANOUT, t
115     Assertions.assertTrue(ok);
116     ok = channel1.queueDeclare("testQueue1", true, false, false, null);
117     Assertions.assertTrue(ok);
118     ok = channel1.queueDeclare("testQueue2", true, false, false, null);
119     Assertions.assertTrue(ok);
120     ok = channel1.queueBind("testQueue1", "testExchange");
121     Assertions.assertTrue(ok);
122     ok = channel1.queueBind("testQueue2", "testExchange");
123     Assertions.assertTrue(ok);
124 
125     byte[] requestBody = "hello".getBytes();
126     // FANOUT 模式, routingKey 不需要
127     ok = channel1.basicPublish("testExchange", "", null, requestBody);
128     Assertions.assertTrue(ok);
129 
130     ok = channel1.basicConsume("testQueue1", true, new Consumer() {
131         @Override
132         public void handleDelivery(String consumerTag, BasicProperties propertie
133             System.out.println("consumerTag=" + consumerTag);
134             System.out.println("properties=" + properties);
135             String bodyString = new String(responseBody, 0, responseBody.length)
136             System.out.println("body=" + bodyString);
137 
138             Assertions.assertEquals(requestBody, responseBody);
139         }
140     });
141     Assertions.assertTrue(ok);
142 
143     Channel channel2 = connection.createChannel();
144     Assertions.assertNotNull(channel1);
145 
146     ok = channel2.basicConsume("testQueue2", true, new Consumer() {
147         @Override
148         public void handleDelivery(String consumerTag, BasicProperties propertie
149             System.out.println("consumerTag=" + consumerTag);
150             System.out.println("properties=" + properties);
151             String bodyString = new String(responseBody, 0, responseBody.length)
152             System.out.println("body=" + bodyString);
153 
154             Assertions.assertEquals(requestBody, responseBody);
155         }
156     });
157     Assertions.assertTrue(ok);
```

```
158
159     Thread.sleep(1000);
160     channel1.close();
161     channel2.close();
162     connection.close();
163 }
164
165 @Test
166 public void testMessageTopic() throws IOException, MqException, InterruptedExcep
167     Connection connection = factory.newConnection();
168     Assertions.assertNotNull(connection);
169     Channel channel = connection.createChannel();
170     Assertions.assertNotNull(channel);
171
172     boolean ok = channel.exchangeDeclare("testExchange", ExchangeType.TOPIC, tru
173     Assertions.assertTrue(ok);
174     ok = channel.queueDeclare("testQueue", true, false, false, null);
175     Assertions.assertTrue(ok);
176     ok = channel.queueBind("testQueue", "testExchange", "aaa.#");
177
178     byte[] requestBody = "hello".getBytes();
179     ok = channel.basicPublish("testExchange", "aaa.bbb.ccc", null, requestBody);
180     Assertions.assertTrue(ok);
181
182     ok = channel.basicConsume("testQueue", true, new Consumer() {
183         @Override
184         public void handleDelivery(String consumerTag, BasicProperties propertie
185             System.out.println("[消费数据] 开始!");
186             System.out.println("consumerTag=" + consumerTag);
187             System.out.println("properties=" + properties);
188             String bodyString = new String(responseBody, 0, responseBody.length)
189             System.out.println("body=" + bodyString);
190
191             Assertions.assertEquals(requestBody, responseBody);
192         }
193     });
194     Assertions.assertTrue(ok);
195     // 等待数据消费完.
196     Thread.sleep(500);
197
198     channel.close();
199     connection.close();
200 }
```

十四. 案例: 基于 MQ 的生产者消费者模型

生产者:

```
public class DemoProducer {
    public static void main(String[] args) throws IOException, InterruptedExcept
        System.out.println("启动生产者!");
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(9090);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare("testExchange", ExchangeType.DIRECT, true, false
        channel.queueDeclare("testQueue", true, false, false, null);

        byte[] body = "hello".getBytes();
        boolean ok = channel.basicPublish("testExchange", "testQueue", null, bod
        System.out.println("投递消息完成! ok=" + ok);

        Thread.sleep(500);
        channel.close();
        connection.close();
    }
}
```

消费者:

```
public class DemoConsumer {
    public static void main(String[] args) throws IOException, MqException, Inte
        System.out.println("启动消费者!");
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(9090);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare("testExchange", ExchangeType.DIRECT, true, false
        channel.queueDeclare("testQueue", true, false, false, null);

        channel.basicConsume("testQueue", true, new Consumer() {
            @Override
```

```
            public void handleDelivery(String consumerTag, BasicProperties prope
                System.out.println("[消费数据] 开始!");
                System.out.println("consumerTag=" + consumerTag);
                System.out.println("properties=" + properties);
                String bodyString = new String(body, 0, body.length);
                System.out.println("body=" + bodyString);
                System.out.println("[消费数据] 完毕!");
            }
        });

        while (true) {
            Thread.sleep(500);
        }
    }
}
```

扩展功能

- 虚拟主机管理
- 用户管理/用户认证
- 交换机/队列 的独占模式和自动删除.
- 发送方确认(broker 给生产者的确认应答)
- 拒绝应答 (nack)
- 死信队列
- 管理接口
- 管理页面