

《软件架构与中间件》

作业一：设计模式应用——简易中间件

学号：2022211939

姓名：杨涛

1. 简易的消息中间件实现1的代码分析

实现一主要包括消息（Message）、消息队列（MessageQueue）、消息中心（MessageCenter）的定义和及其相关实现。代码的设计采用了包括**工厂模式**、**单例模式**等在内的设计模式。

1. **类图设计和时序图设计**

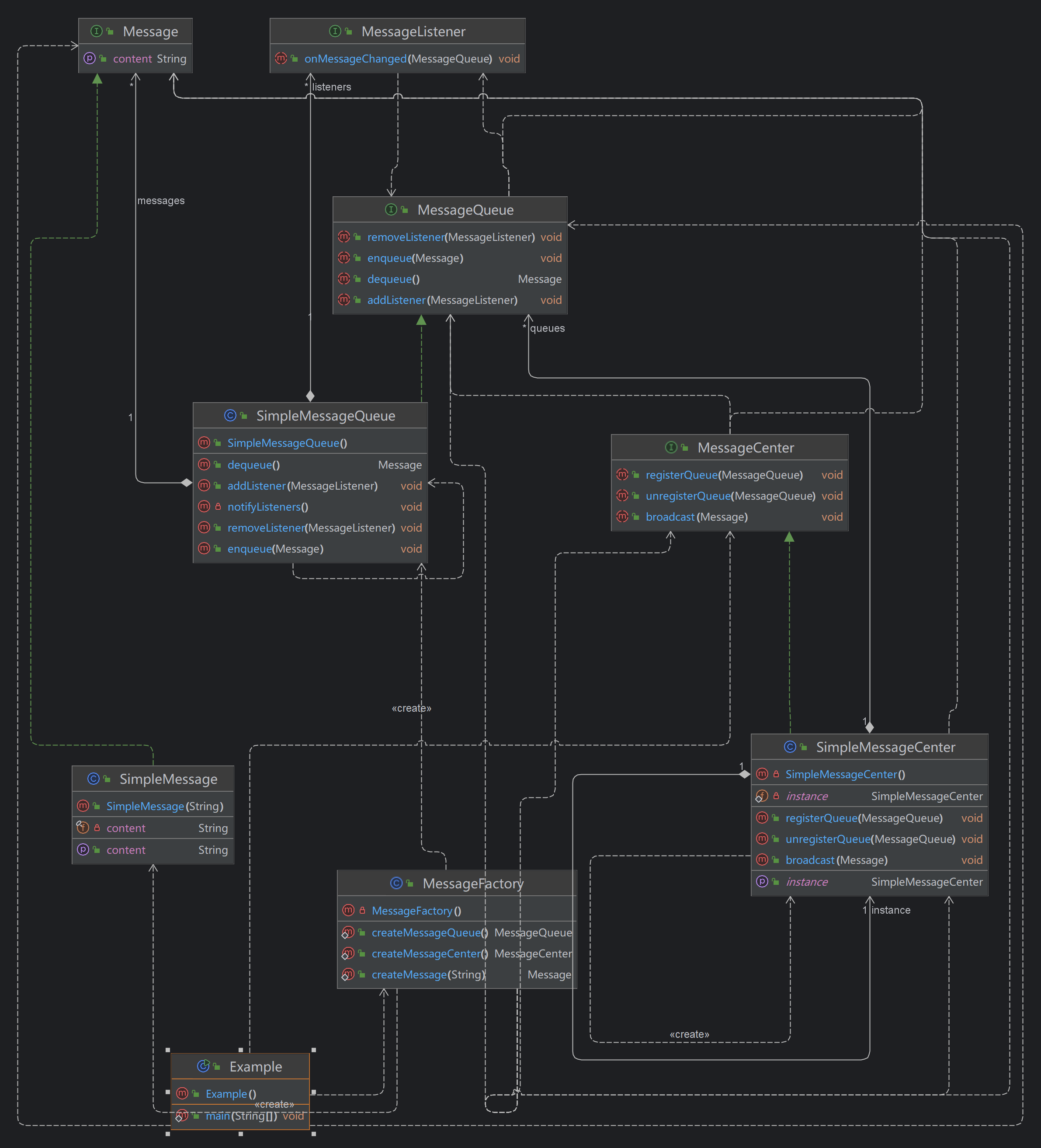


图 1 类图

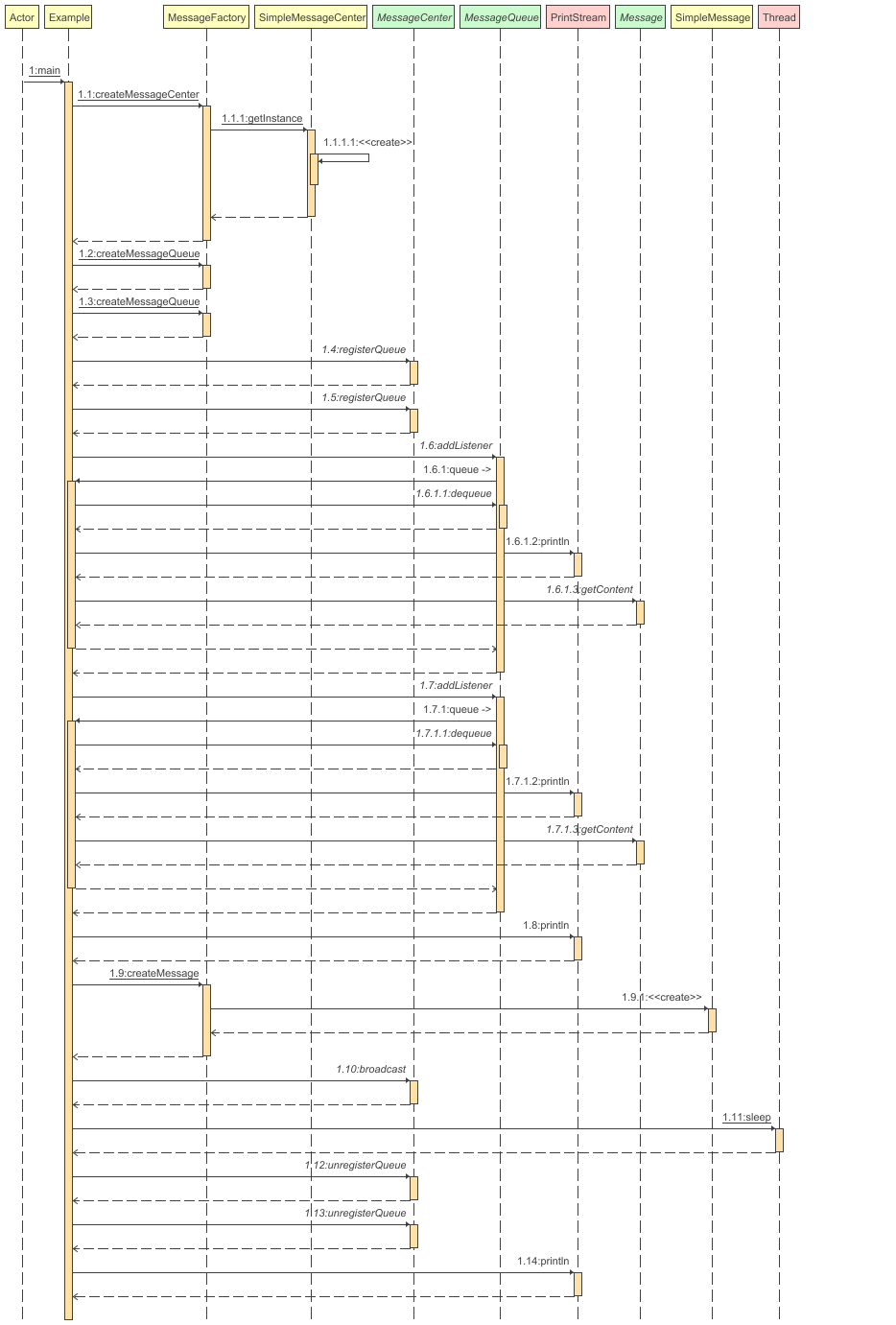


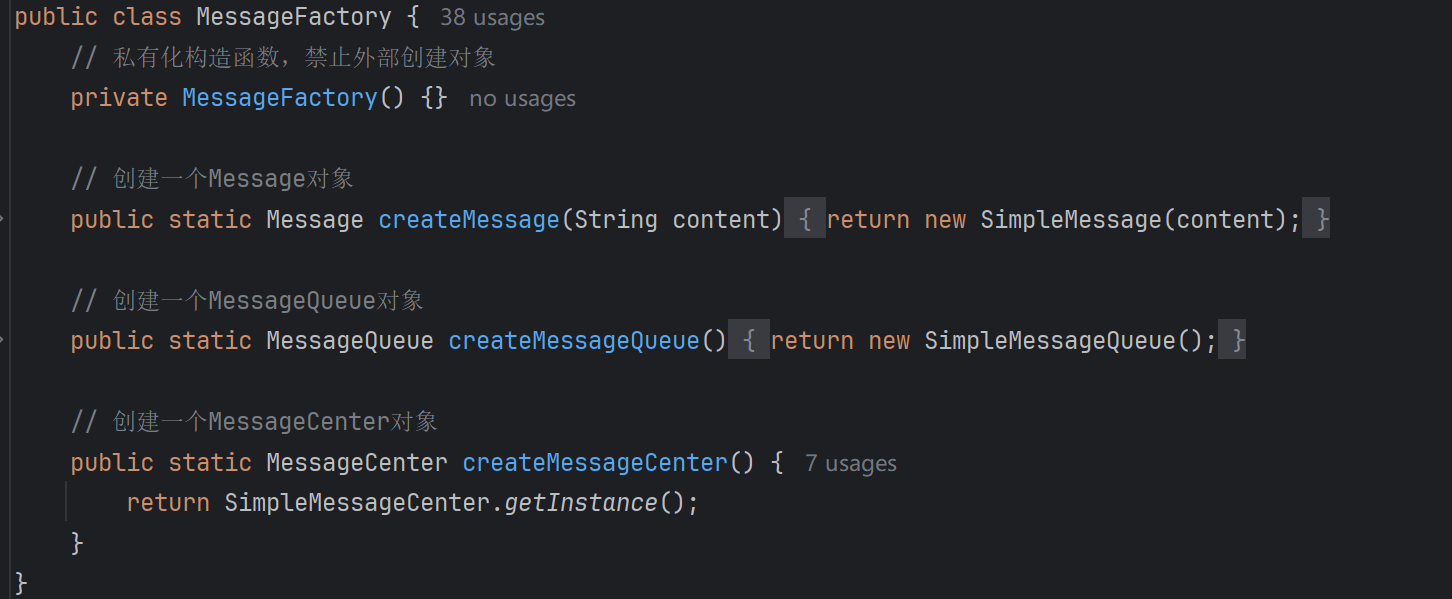
图 2 时序图

### ****1. MessageFactory 类（工厂类）****

#### ****作用：****

MessageFactory 是一个工厂类，负责创建消息对象（Message）、消息队列对象（MessageQueue）和消息中心对象（MessageCenter）。

#### ****功能分析：****



1. **私有构造函数：** MessageFactory 的构造函数被私有化，禁止外部直接实例化 MessageFactory 对象。
2. **静态工厂方法：**
3. createMessage(String content): 创建并返回一个 SimpleMessage 对象，它实现了 Message 接口。
4. createMessageQueue(): 创建并返回一个 SimpleMessageQueue 对象，它实现了 MessageQueue 接口。
5. createMessageCenter(): 创建并返回一个 SimpleMessageCenter 对象，它实现了 MessageCenter 接口，采用单例模式确保只有一个实例。

#### ****总结：****

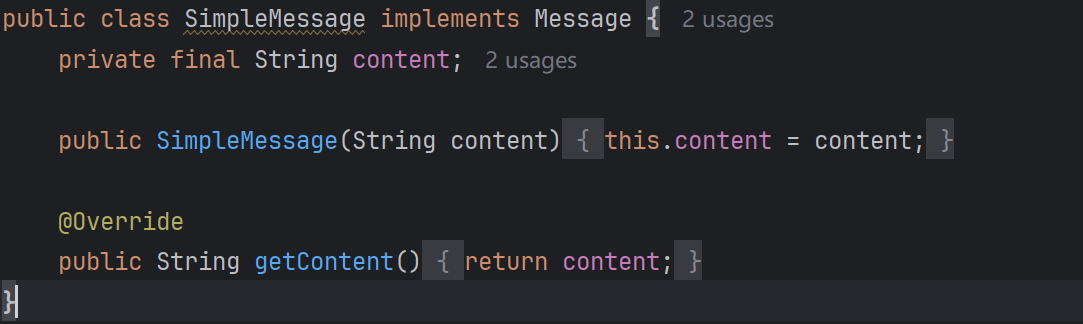
MessageFactory 简化了客户端对消息系统组件的创建过程，提供了清晰的接口来创建不同的对象。

### ****2. SimpleMessage 类（消息类）****

#### ****作用：****

SimpleMessage 是一个简单的消息实现，表示一个消息实例。它实现了 Message 接口，封装了消息的内容。

#### ****功能分析：****



1. **属性：**

content: 存储消息的具体内容。

1. **构造方法：**

SimpleMessage(String content): 构造函数初始化消息内容。

1. **接口实现：**

getContent(): 实现 Message 接口的方法，返回消息的内容。

#### ****总结：****

SimpleMessage 是一个简单的消息对象类，仅保存和提供消息内容。

### ****3. SimpleMessageCenter 类（消息中心类）****

#### ****作用：****

SimpleMessageCenter 是一个简单的消息中心类，实现了 MessageCenter 接口，主要功能是注册和注销消息队列，并广播消息到所有注册的队列。

#### ****功能分析：****



1. **属性：**
2. queues: 存储所有已注册的 MessageQueue 实例。
3. instance: 静态变量用于保存 SimpleMessageCenter 的唯一实例（用于实现单例模式）。
4. **单例模式（双重检查锁定）：**

getInstance(): 使用双重检查锁定（Double-Checked Locking）方式实现了线程安全的单例模式，确保 SimpleMessageCenter 只有一个实例。

1. **接口实现：**
2. registerQueue(MessageQueue queue): 将一个队列注册到消息中心。
3. unregisterQueue(MessageQueue queue): 从消息中心注销一个队列。
4. broadcast(Message message): 将消息广播到所有已注册的消息队列。此方法会遍历 queues 列表，调用每个队列的 enqueue 方法将消息加入队列。

#### ****总结：****

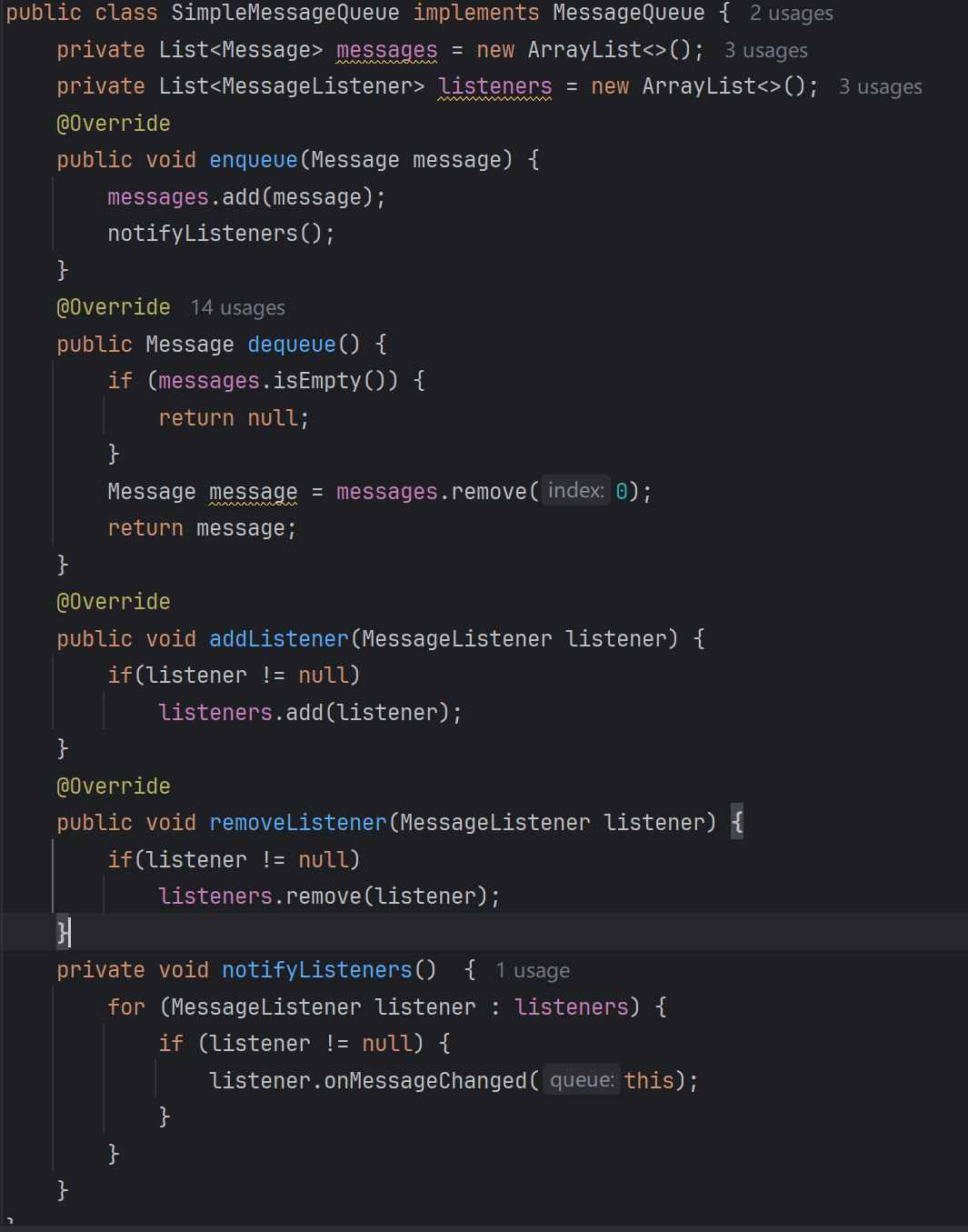
SimpleMessageCenter 类充当了消息的广播中心，通过注册和注销队列来管理消息的分发。

### ****4. SimpleMessageQueue 类（消息队列类）****

#### ****作用：****

SimpleMessageQueue 是一个简单的消息队列实现，支持消息的入队、出队以及监听者的管理。它实现了 MessageQueue 接口。

#### ****功能分析：****



1. **属性：**
2. messages: 存储队列中的消息列表。
3. listeners: 存储所有监听该队列的监听器（MessageListener）。
4. **接口实现：**
5. enqueue(Message message): 将消息加入队列，并调用 notifyListeners 方法通知所有监听者。
6. dequeue(): 从队列中取出一条消息。如果队列为空，返回 null。
7. addListener(MessageListener listener): 添加监听器到队列。
8. removeListener(MessageListener listener): 从队列中移除监听器。
9. **私有方法：**

notifyListeners(): 遍历所有监听器，调用 onMessageChanged 方法通知它们消息状态发生了变化。

#### ****总结：****

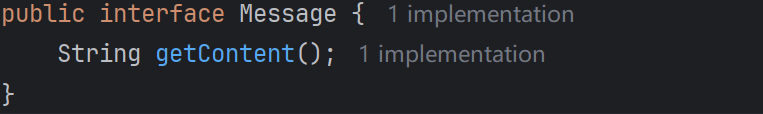
SimpleMessageQueue 类实现了消息队列的基本操作，如入队、出队、添加和移除监听器，并且通过通知机制告知监听者队列中消息的变化。

### ****5. 接口层（Message, MessageQueue, MessageCenter, MessageListener）****

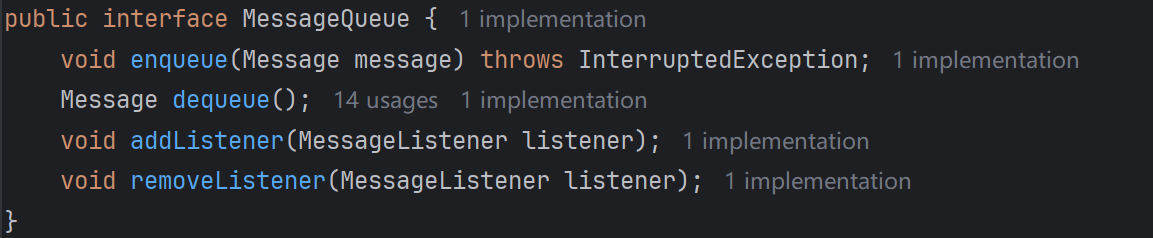
#### ****作用：****

这些接口定义了消息系统的核心行为和约定，使得具体的实现类（如 SimpleMessage、SimpleMessageQueue、SimpleMessageCenter）能够遵循统一的标准。

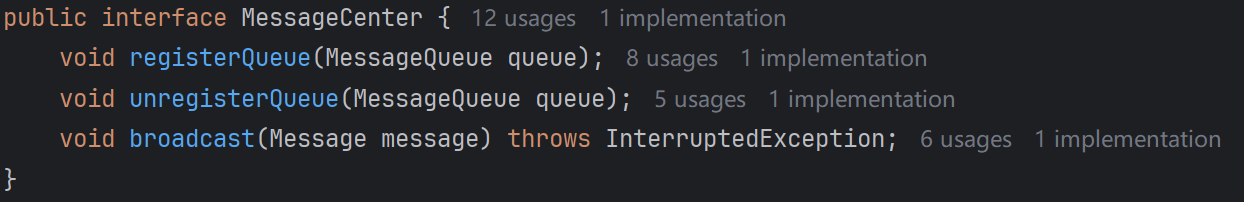
#### ****功能分析：****

1. **Message 接口：**

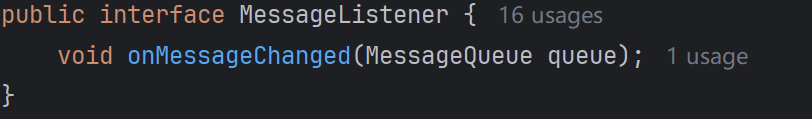
定义了获取消息内容的 getContent 方法，所有消息类都需要实现该方法。

1. **MessageQueue 接口：**

定义了消息队列的基本操作，如 enqueue（入队）、dequeue（出队）、addListener（添加监听器）和 removeListener（移除监听器）。

1. **MessageCenter 接口：**

定义了消息中心的操作，如注册队列（registerQueue）、注销队列（unregisterQueue）和广播消息（broadcast）。

1. **MessageListener 接口：**

定义了队列消息变动的监听机制，onMessageChanged 方法在队列中的消息发生变化时被调用。

#### ****总结：****

这些接口提供了标准化的行为规范，确保不同的实现类能够一致地工作。通过接口，系统具有良好的扩展性和灵活性。

### ****6. 整体设计总结：****

1. **工厂模式**：MessageFactory 类提供了静态方法来创建消息、消息队列和消息中心实例，简化了对象的创建过程。
2. **单例模式**：SimpleMessageCenter 类采用了单例模式，确保消息中心全局只有一个实例。
3. **消息广播与监听机制**：SimpleMessageCenter 和 SimpleMessageQueue 类通过广播和监听机制实现了消息的分发和处理，支持多队列和多监听器的管理。
4. **面向接口编程**：通过接口的设计，系统具有良好的解耦性，易于扩展和维护。

该设计方案具有较好的灵活性，能适应不同的消息处理需求，同时确保了高效的消息分发与队列管理。

1. 简易的消息中间件实现2的代码分析

实现二是一个基于消息队列的生产者-消费者模式实现，包含消息（Message）、消息队列（MessageQueue）、消息中心（MessageCenter）、生产者（Producer）和消费者（Consumer）等组成部分。其设计也使用了**工厂模式**、**单例模式**、**阻塞队列**等技术。

**0.类图设计和时序图设计**

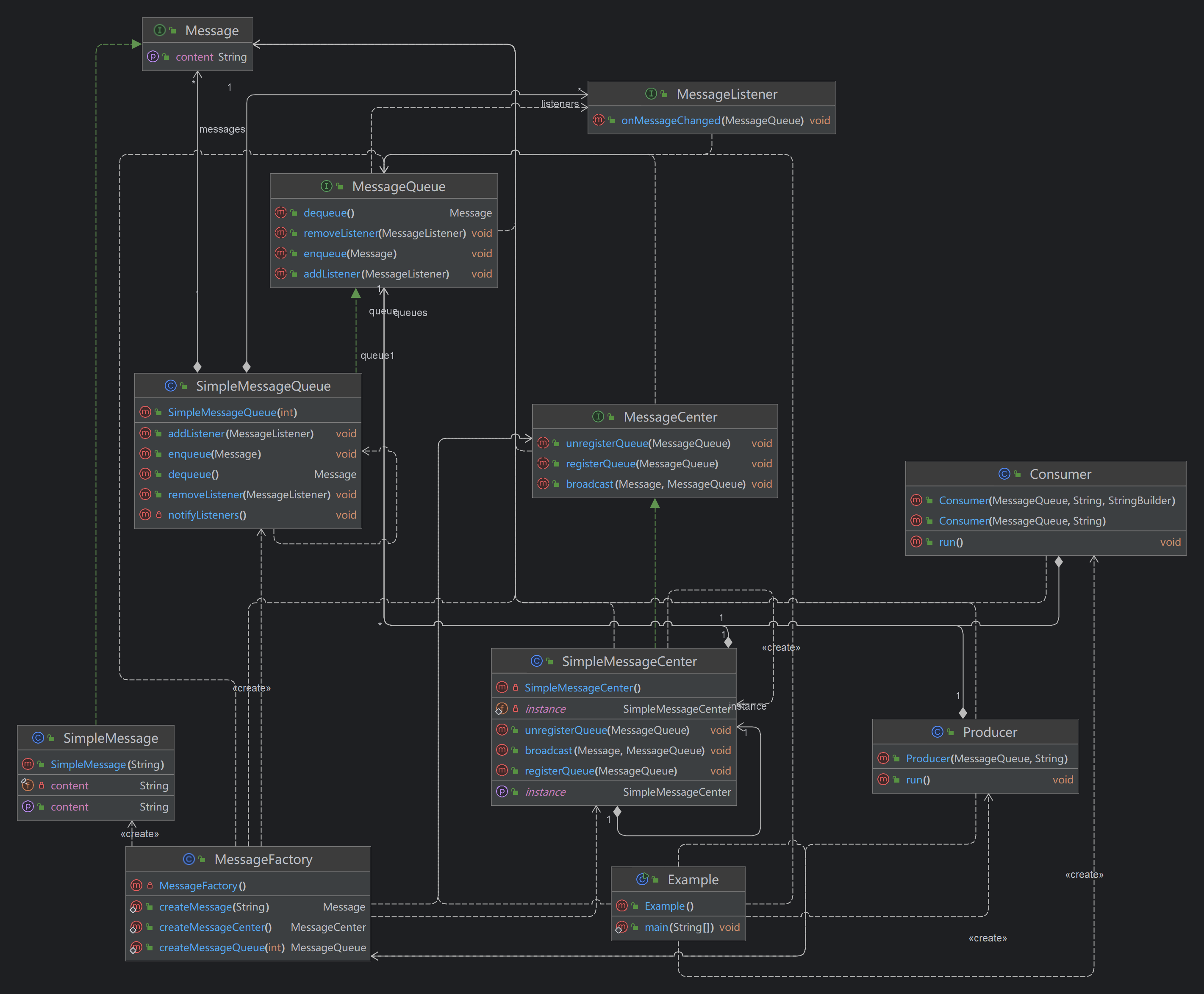


图 3 类图

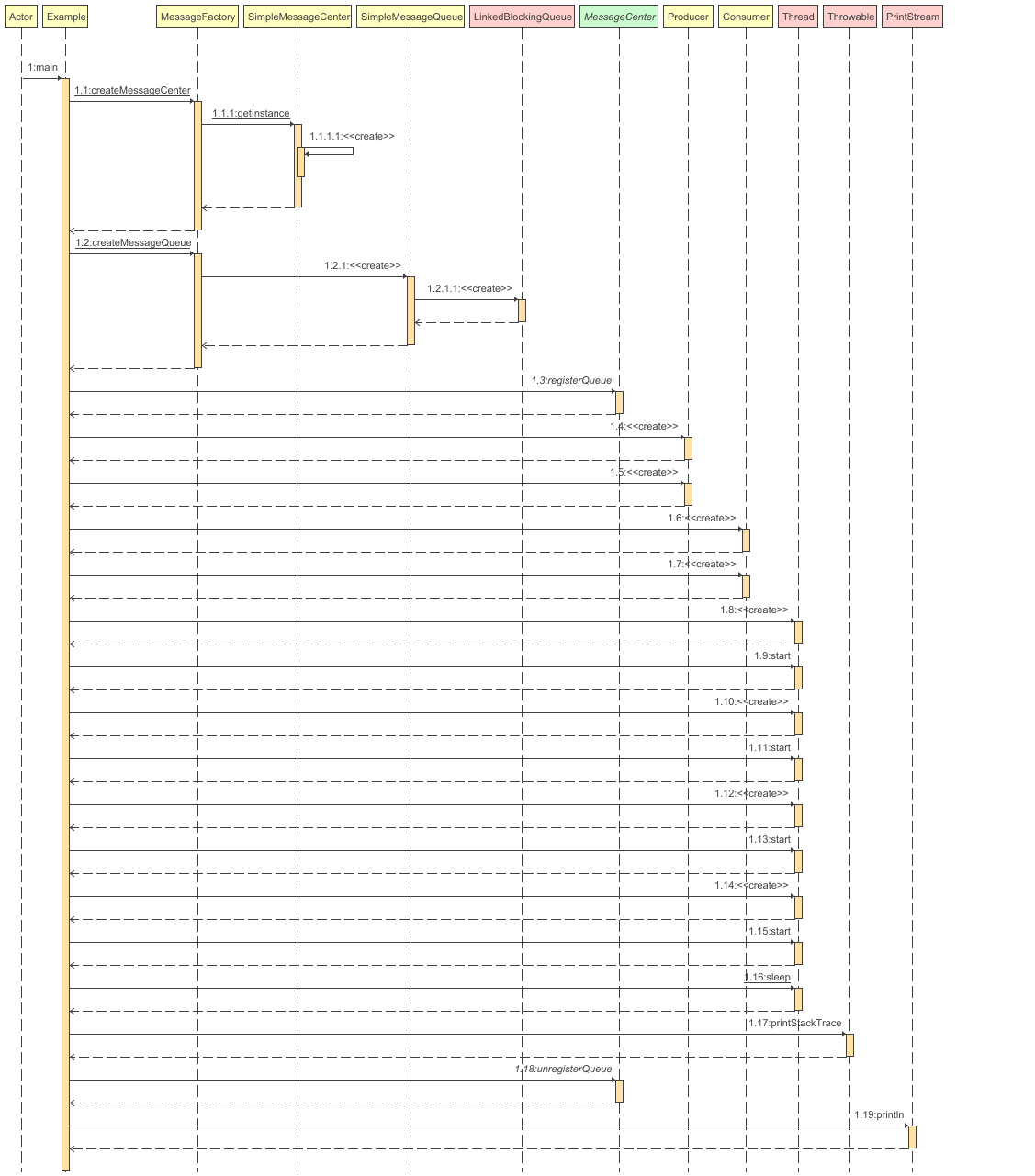


图 4 时序图

### ****1. MessageFactory 类（工厂类）****

#### ****作用：****

MessageFactory 类用于提供静态方法创建不同类型的消息对象、消息队列对象和消息中心对象。它是一个工厂模式的实现，用于简化对象的创建。

#### ****功能分析：****

1. **私有构造函数：** 禁止外部直接实例化 MessageFactory。
2. **静态工厂方法：**
3. createMessage(String content)：创建并返回一个 SimpleMessage 对象，传入消息内容。
4. createMessageQueue(int capacity)：创建并返回一个具有指定容量的 SimpleMessageQueue 对象。此处队列容量的引入是为了限制队列的大小。
5. createMessageCenter()：创建并返回一个 SimpleMessageCenter 对象，采用单例模式保证消息中心只有一个实例。

#### ****总结：****

MessageFactory 类提供了简化对象创建的工厂方法，客户端无需直接调用构造函数即可获得所需的对象。

### ****2. SimpleMessage 类（消息类）****

#### ****作用：****

SimpleMessage 类实现了 Message 接口，表示一个简单的消息对象。

#### ****功能分析：****

1. **属性：**

content: 存储消息的具体内容。

1. **构造方法：**

SimpleMessage(String content)：构造函数用于初始化消息内容。

1. **接口实现：**

getContent()：返回消息的内容，遵循 Message 接口。

#### ****总结：****

SimpleMessage 是一个简单的消息实体类，只包含消息内容的字段和相关的 getter 方法。

### ****3. SimpleMessageCenter 类（消息中心类）****

#### ****作用：****

SimpleMessageCenter 类实现了 MessageCenter 接口，负责注册消息队列、注销队列，并向队列广播消息。

#### ****功能分析：****

1. **属性：**
2. queues: 存储已注册的 MessageQueue 队列列表。
3. instance: 单例模式中用于保存唯一实例的静态变量。
4. **单例模式（双重检查锁定）：**

getInstance()：通过双重检查锁定（Double-Checked Locking）实现线程安全的单例模式，保证 SimpleMessageCenter 只会有一个实例。

1. **接口实现：**
2. registerQueue(MessageQueue queue)：将消息队列注册到消息中心。
3. unregisterQueue(MessageQueue queue)：将消息队列从消息中心注销。
4. broadcast(Message message, MessageQueue queue)：将消息广播到指定的队列，调用队列的 enqueue() 方法将消息加入队列。

#### ****总结：****

SimpleMessageCenter 类充当消息分发中心，允许将消息发送到多个队列，并确保只有一个消息中心实例（使用单例模式）。

### ****4. SimpleMessageQueue 类（消息队列类）****

#### ****作用：****

SimpleMessageQueue 类实现了 MessageQueue 接口，表示一个消息队列，支持消息的入队、出队、以及监听器管理。

#### ****功能分析：****

1. **属性：**
2. capacity: 队列的容量，限制队列能够存储的消息数量。
3. messages: 使用 BlockingQueue（具体是 LinkedBlockingQueue）来存储消息。BlockingQueue 提供线程安全的队列操作，支持阻塞队列的功能。
4. listeners: 监听器列表，使用 CopyOnWriteArrayList 来管理，保证并发访问时线程安全。
5. lock: 锁对象，用于同步操作。
6. **构造方法：**

SimpleMessageQueue(int capacity)：使用指定的容量初始化 LinkedBlockingQueue，并确保队列大小不会超过该容量。

1. **接口实现：**
2. enqueue(Message message)：将消息加入队列，如果队列满了则会阻塞，直到有空间。调用 notifyListeners() 方法通知所有监听器队列已更新。
3. dequeue()：从队列中取出消息，如果队列为空则会阻塞，直到有新消息到来。
4. addListener(MessageListener listener)：添加一个消息监听器。
5. removeListener(MessageListener listener)：移除一个消息监听器。
6. **私有方法：**

notifyListeners()：通知所有的监听器队列的状态发生了变化。

#### ****总结：****

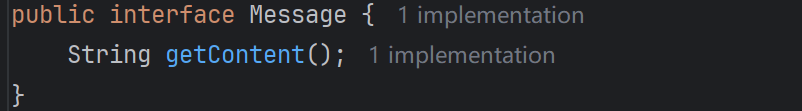
SimpleMessageQueue 类提供了基于阻塞队列的消息存取操作。它支持消息的线程安全入队和出队，并支持消息队列的监听机制。

### ****5. 消息接口层（Message, MessageQueue, MessageCenter, MessageListener）****

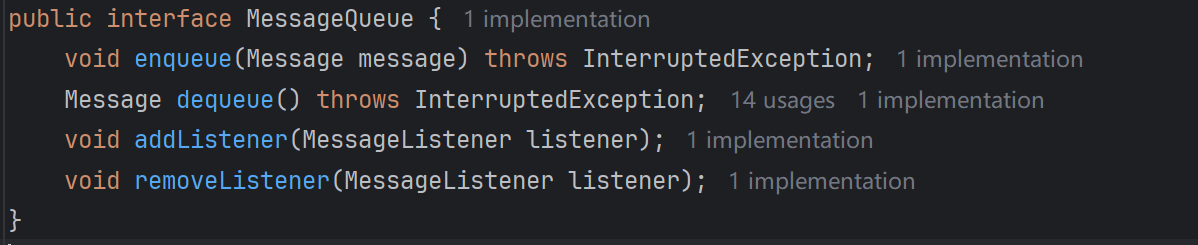
#### ****作用：****

这些接口定义了消息系统中的核心行为，并确保不同实现类遵循统一的规范。

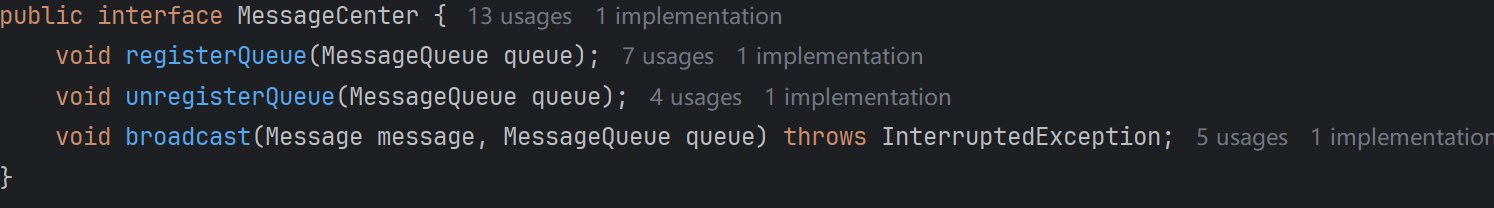
#### ****功能分析：****

1. **Message 接口：**

定义了 getContent() 方法，表示消息的内容。任何消息类必须实现这个方法。

1. **MessageQueue 接口：**

定义了消息队列的基本操作：enqueue(Message message)（入队）、dequeue()（出队）、addListener(MessageListener listener)（添加监听器）和 removeListener(MessageListener listener)（移除监听器）。

1. **MessageCenter 接口：**

定义了消息中心的操作：registerQueue(MessageQueue queue)（注册队列）、unregisterQueue(MessageQueue queue)（注销队列）和 broadcast(Message message, MessageQueue queue)（广播消息）。

1. **MessageListener 接口：**

定义了监听器的操作：onMessageChanged(MessageQueue queue)，用于通知监听器消息队列的状态发生变化。

#### ****总结：****

这些接口提供了系统组件之间的标准化通信方式，确保了不同实现类之间的兼容性和灵活性。

### ****6. 生产者类（Producer）****

#### ****作用：****

Producer 类模拟了消息的生产者，负责向消息队列中添加消息。

#### ****功能分析：****

1. **构造方法：**

Producer(MessageQueue queue, String name)：初始化生产者并指定消息队列和生产者名称。

1. run() **方法：**

启动生产者线程，生产者连续创建10条消息，并通过 queue.enqueue() 方法将其加入队列，每条消息之间有一定的延时（Thread.sleep(100)）。

#### ****总结：****

Producer 类模拟了消息的生产过程，向队列中不断添加消息，符合生产者-消费者模型。

### ****7. 消费者类（Consumer）****

#### ****作用：****

Consumer 类模拟了消息的消费者，负责从消息队列中取出消息并进行处理。

#### ****功能分析：****

1. **构造方法：**

Consumer(MessageQueue queue, String name)：初始化消费者并指定消息队列和消费者名称。

1. run() **方法：**

启动消费者线程，消费者持续从队列中取出消息进行处理。每处理一条消息后，消费者会有一定的延时（Thread.sleep(200)）。

#### ****总结：****

Consumer 类模拟了消息的消费过程，不断从队列中消费消息并进行处理，符合生产者-消费者模型。

### ****8. 总结：****

实现二是一个基于阻塞队列的生产者-消费者模型，包含了以下核心特性：

1. **工厂模式**：MessageFactory 提供了简化的对象创建方法。
2. **单例模式**：SimpleMessageCenter 确保消息中心只有一个实例。
3. **阻塞队列**：SimpleMessageQueue 使用 BlockingQueue 提供线程安全的消息入队和出队操作，避免了多线程竞争。
4. **生产者-消费者模式**：Producer 和 Consumer 类分别模拟消息的生产和消费过程。
5. **监听机制**：SimpleMessageQueue 类实现了消息队列状态变更的通知机制。

整体设计具有较好的线程安全性，灵活的消息传递和处理能力，适合用于高并发环境中的消息队列系统。

1. 我的事件中间件实现

第三个实现构建了一个具有生产者-消费者模式的消息系统，并在设计上加入了许多优化和扩展，涉及到了**工厂模式**、**单例模式**、**观察者模式**（通过消费者和消息队列的监听机制）、以及**多线程管理**等概念。以下是对各个部分的详细分析：

1. **类图设计和时序图设计**

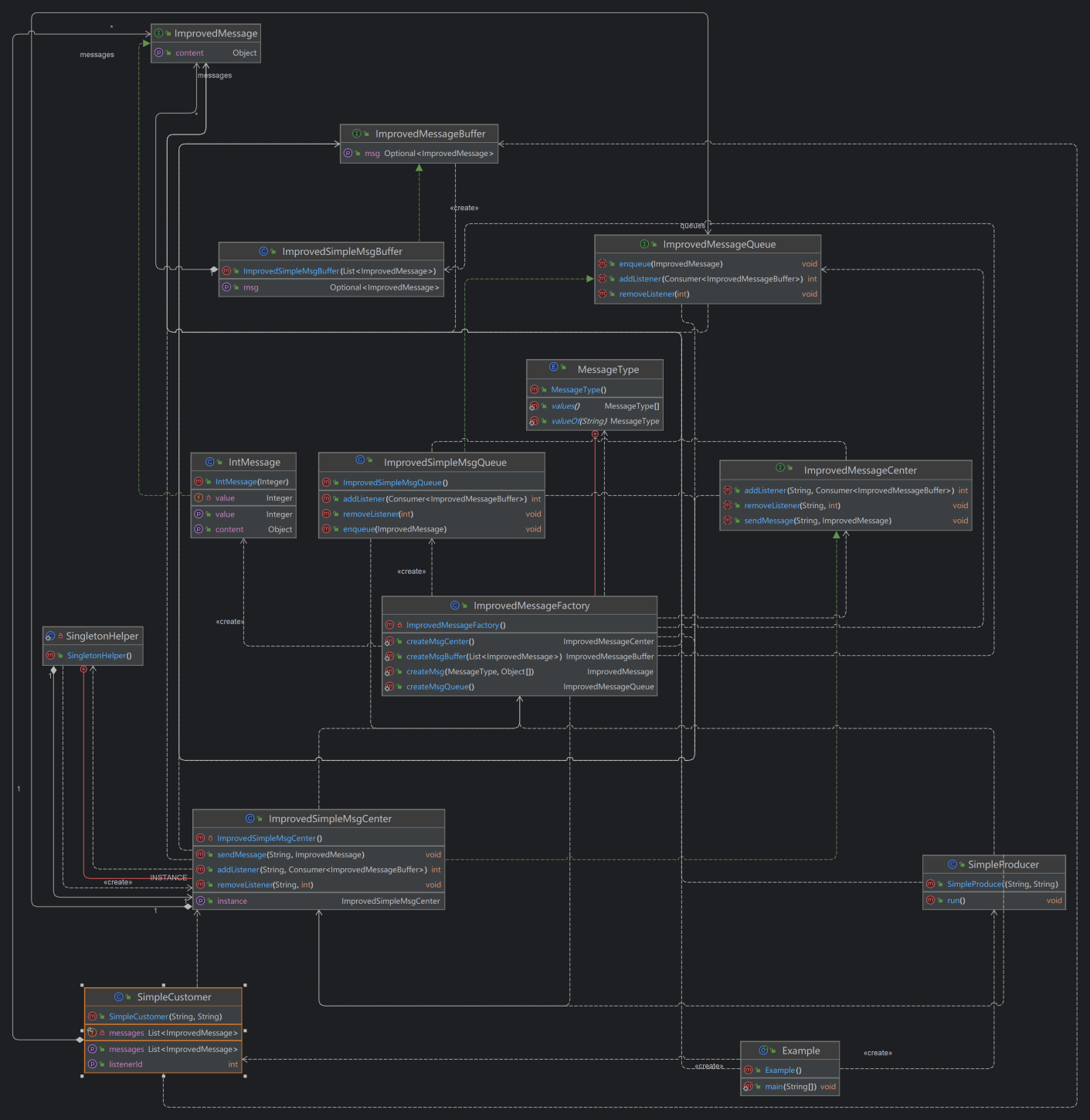


图 5 类图

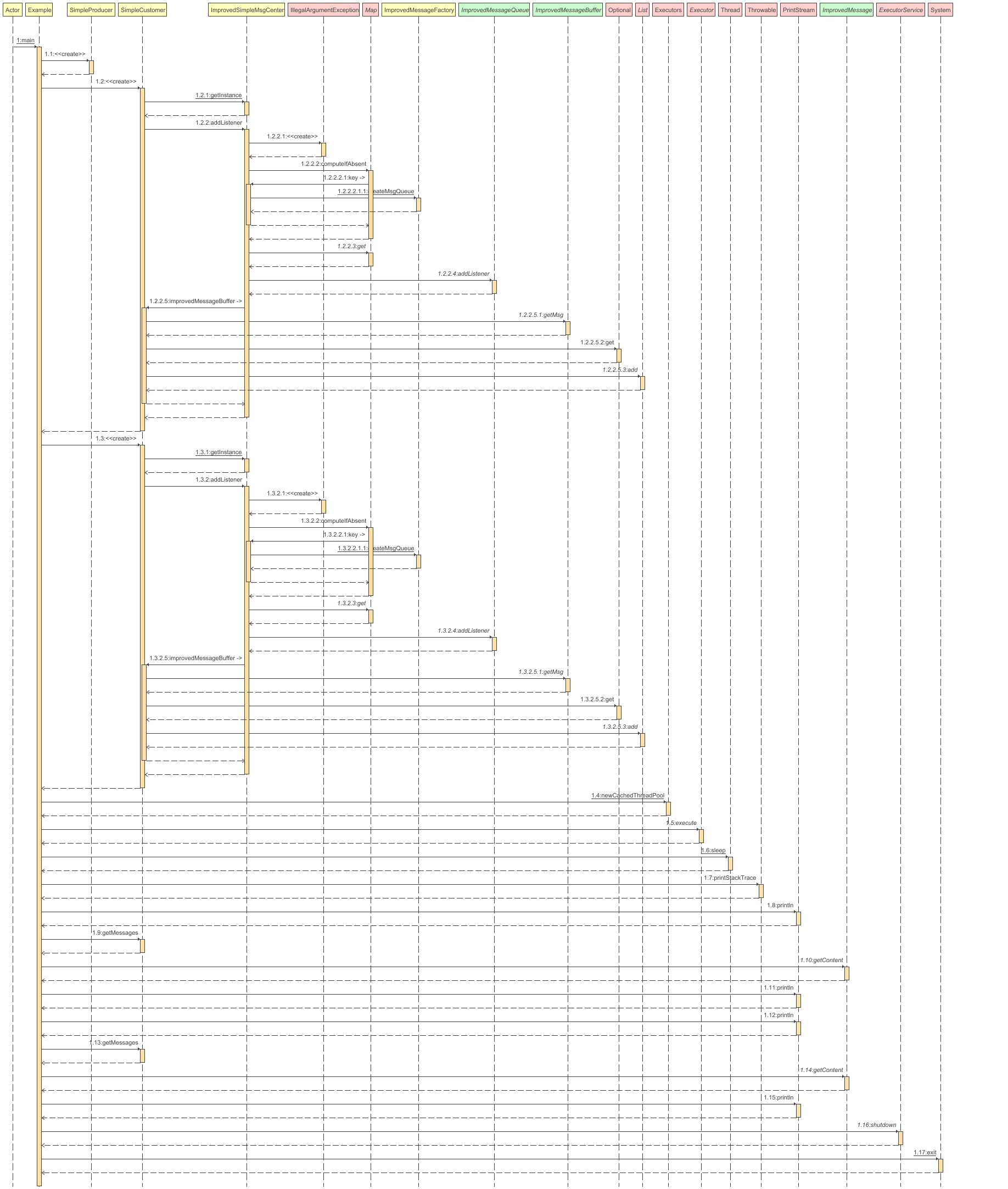


图 6 时序图

### ****1. ImprovedMessageFactory（工厂类）****

#### ****作用：****

ImprovedMessageFactory 提供了静态方法来创建不同类型的消息对象、消息队列、消息中心等，简化了对象创建的复杂度。

#### ****功能分析：****

1. **私有构造函数：**

禁止外部直接实例化 ImprovedMessageFactory。

1. **MessageType 枚举：**

MessageType 枚举定义了支持的消息类型，这里有 intMessage 类型，便于将来扩展新的消息类型。

1. **静态工厂方法：**
2. createMsg(MessageType type, Object... params)：根据传入的类型创建相应的 ImprovedMessage 对象。当前仅支持 intMessage 类型，未来可以扩展支持更多类型的消息。
3. createMsgQueue()：创建并返回一个新的 ImprovedSimpleMsgQueue 对象，代表消息队列。
4. createMsgCenter()：创建并返回 ImprovedSimpleMsgCenter 的单例实例，代表消息中心。
5. createMsgBuffer(List<ImprovedMessage> messageList)：创建并返回一个 ImprovedSimpleMsgBuffer 对象，代表一个消息缓冲区。

#### ****总结：****

ImprovedMessageFactory 类封装了对象的创建逻辑，提供了统一且简便的接口来创建各种消息相关对象。

### ****2. ImprovedSimpleMsgBuffer（消息缓冲区）****

#### ****作用：****

ImprovedSimpleMsgBuffer 实现了 ImprovedMessageBuffer 接口，表示一个简单的消息缓冲区，用于存储和获取消息。

#### ****功能分析：****

1. **属性：**

messages：一个 List<ImprovedMessage> 用于存储消息。

1. **构造方法：**

ImprovedSimpleMsgBuffer(List<ImprovedMessage> messages)：通过传入的消息列表初始化缓冲区。

1. **接口实现：**

getMsg()：从缓冲区中取出一个消息。如果缓冲区为空，返回 Optional.empty()，否则移除并返回第一个消息。

#### ****总结：****

ImprovedSimpleMsgBuffer 提供了一个简单的消息存储与获取方式，确保在处理消息时，如果消息为空，能够安全地返回 Optional，避免 NullPointerException。

### ****3. ImprovedSimpleMsgCenter（消息中心）****

#### ****作用：****

ImprovedSimpleMsgCenter 实现了 ImprovedMessageCenter 接口，是系统的核心部分，负责管理多个消息队列，并处理消息的发送、接收和监听。

#### ****功能分析：****

1. **单例模式（静态内部类）：**

使用静态内部类 SingletonHelper 来实现线程安全的单例模式，确保 ImprovedSimpleMsgCenter 只有一个实例。

1. **属性：**

queues：使用 ConcurrentHashMap 存储队列，支持高并发。

1. **接口实现：**
2. addListener(String msgQueueName, Consumer<ImprovedMessageBuffer> listener)：为指定的消息队列添加监听器，监听队列中的消息变化。当队列中有新消息时，调用 listener.accept() 来处理消息。
3. removeListener(String msgQueueName, int identifier)：从指定消息队列中移除监听器。
4. sendMessage(String msgQueueName, ImprovedMessage msg)：将消息发送到指定的队列。如果队列存在，则调用队列的 enqueue() 方法将消息加入队列。

#### ****总结：****

ImprovedSimpleMsgCenter 作为消息中心，负责协调消息的广播、接收和队列的管理。它使用 ConcurrentHashMap 来存储消息队列，确保在高并发环境下依然能够稳定工作。

### ****4. ImprovedSimpleMsgQueue（消息队列）****

#### ****作用：****

ImprovedSimpleMsgQueue 实现了 ImprovedMessageQueue 接口，表示一个简单的消息队列，支持消息的入队、出队和监听。

#### ****功能分析：****

1. **属性：**
2. consumers：使用 ConcurrentHashMap 存储队列的消费者，消费者是监听器回调（Consumer<ImprovedMessageBuffer>）。
3. executor：使用 ExecutorService（固定大小线程池）来处理消息，避免在主线程中直接执行回调，提升性能。
4. **接口实现：**
5. enqueue(ImprovedMessage message)：将消息放入队列，并遍历所有已注册的消费者，提交到线程池中异步执行回调。
6. addListener(Consumer<ImprovedMessageBuffer> callback)：添加消息队列的监听器。每个监听器都由一个唯一的 listenerId 标识。
7. removeListener(int hash)：根据监听器的 hash（即 listenerId）移除监听器。

#### ****总结：****

ImprovedSimpleMsgQueue 结合了 ExecutorService 和 ConcurrentHashMap，支持异步和高效的消息处理。它能够在消息队列中存储消息，并通过多个消费者（监听器）进行并发消费。

### ****5. IntMessage（整型消息）****

#### ****作用：****

IntMessage 类实现了 ImprovedMessage 接口，表示一个包含整型值的消息。

#### ****功能分析：****

1. **属性：**

value：存储消息的整型值。

1. **构造方法：**

IntMessage(Integer value)：构造函数初始化消息的整型值。

1. **接口实现：**

getContent()：返回消息的内容，即 value。

#### ****总结：****

IntMessage 类封装了一个整型值，实现了 ImprovedMessage 接口，表示一种消息类型。

### ****6. 消息接口层（ImprovedMessage, ImprovedMessageBuffer, ImprovedMessageQueue, ImprovedMessageCenter）****

#### ****作用：****

这些接口定义了消息系统的核心行为，确保系统组件之间能够灵活解耦。

#### ****功能分析：****

1. **ImprovedMessage**：定义了 getContent() 方法，获取消息的内容。
2. **ImprovedMessageBuffer**：定义了 getMsg() 方法，获取缓冲区中的消息。
3. **ImprovedMessageQueue**：定义了消息队列的基本操作，包括 enqueue()（入队）、addListener()（添加监听器）和 removeListener()（移除监听器）。
4. **ImprovedMessageCenter**：定义了消息中心的操作，包括添加监听器、移除监听器和发送消息。

#### ****总结：****

这些接口为系统的扩展性和可维护性提供了保障。通过接口的方式定义了各组件的行为，允许不同的实现类遵循统一的规范。

### ****7. 消费者类（SimpleCustomer）****

#### ****作用：****

SimpleCustomer 类模拟了一个消费者，它从指定的消息队列中接收消息并进行处理。

#### ****功能分析：****

1. **构造方法：**

SimpleCustomer(String customerName, String msgQueueName)：初始化消费者并为指定的队列添加监听器。每当队列中有新消息，消费者会获取并处理该消息。

1. **属性：**

messages：存储消费者接收到的消息。

1. **方法：**

getMessages()：返回消费者收到的所有消息。

#### ****总结：****

SimpleCustomer 类模拟了一个简单的消费者，它注册到消息中心，监听指定的消息队列，并将接收到的消息存储起来。

### ****8. 生产者类（SimpleProducer）****

#### ****作用：****

SimpleProducer 类模拟了一个生产者，它向指定的消息队列中发送消息。

#### ****功能分析：****

1. **构造方法：**

SimpleProducer(String producerName, String msgQueueName)：初始化生产者并指定消息队列。

1. run() **方法：**

每隔一定时间向队列中发送一条整型消息，模拟生产者生成消息的过程。

#### ****总结：****

SimpleProducer 类模拟了一个简单的生产者，它向消息中心发送消息，消费者可以在队列中接收到这些消息。

### ****9. 总结：****

第三个实现，实现了一个消息队列系统，具有生产者-消费者模式，并采用了以下设计模式和技术：

1. **工厂模式**：ImprovedMessageFactory 提供了简化的对象创建方法。
2. **单例模式**：ImprovedSimpleMsgCenter 确保消息中心只有一个实例。
3. **观察者模式**：消费者通过监听消息队列接收消息，提供灵活的消息通知机制。
4. **多线程管理**：通过 ExecutorService 提供异步执行，优化性能。
5. **高并发支持**：使用 ConcurrentHashMap 和其他线程安全数据结构，确保在多线程环境下的稳定性。

整体设计清晰、灵活，并具备良好的扩展性，适用于处理高并发的消息系统。

1. 测试介绍
2. **实现一测试**

### ****MessageSystemTests1 测试类总结****

该测试类对消息系统的主要组件进行了单元测试，涵盖了 **消息（Message）**、**消息队列（MessageQueue）**、**消息中心（MessageCenter）**、**消息监听器（MessageListener）** 以及 **消息工厂（MessageFactory）** 等模块的功能验证。以下是各个部分的测试总结：

### ****1. MessageTest（Message接口测试）****

1. **testGetContent\_ValidMessage\_ReturnsCorrectContent**  
   **目的**: 验证 Message 接口能正确获取消息内容。  
   **关键点**: 创建一个消息对象，确保返回的消息内容是 "Hello"。
2. **testGetContent\_EmptyMessage\_ReturnsEmptyString**  
   **目的**: 验证 Message 接口能正确处理空消息。  
   **关键点**: 创建一个空消息对象，确保返回内容为空字符串。

### ****2. MessageQueueTest（MessageQueue接口测试）****

1. **testEnqueue\_ValidMessage\_MessageIsEnqueued**  
   **目的**: 验证消息是否能正确入队。  
   **关键点**: 向队列添加一条消息并确认队列可以正确出队。
2. **testDequeue\_ValidMessage\_MessageIsDequeued**  
   **目的**: 验证消息是否能够成功出队。  
   **关键点**: 向队列添加一条消息并确保出队后的内容正确。
3. **testDequeue\_EmptyQueue\_ReturnsNull**  
   **目的**: 测试空队列的 dequeue 操作返回 null。  
   **关键点**: 在空队列上调用 dequeue 方法，确保返回 null。
4. **testAddListener\_ValidListener\_ListenerIsAdded**  
   **目的**: 测试向队列添加监听器。  
   **关键点**: 向队列添加监听器并确保消息能被监听器处理。
5. **testRemoveListener\_ValidListener\_ListenerIsRemoved**  
   **目的**: 测试从队列移除监听器。  
   **关键点**: 向队列添加监听器后移除，确保移除后的监听器不再处理队列中的消息。

### ****3. MessageCenterTest（MessageCenter接口测试）****

1. **testRegisterQueue\_ValidQueue\_QueueIsRegistered**  
   **目的**: 测试消息队列能否成功注册到消息中心。  
   **关键点**: 注册队列到消息中心并广播消息，确保消息能够被成功广播。
2. **testUnregisterQueue\_RegisteredQueue\_QueueIsUnregistered**  
   **目的**: 测试已注册的队列是否能够从消息中心注销。  
   **关键点**: 注册并注销队列，确保广播操作在注销后仍能正确执行。
3. **testBroadcast\_ValidMessage\_AllQueuesReceiveMessage**  
   **目的**: 测试消息是否能成功广播到所有注册的队列。  
   **关键点**: 向消息中心广播一条消息，确保所有注册的队列都能接收到该消息。
4. **testBroadcast\_EmptyQueue\_NoMessagesReceived**  
   **目的**: 测试当没有队列注册时广播消息的情况。  
   **关键点**: 当没有队列注册时，广播操作不应抛出异常。

### ****4. MessageListenerTest（MessageListener接口测试）****

1. **testOnMessageChanged\_ValidMessage\_ListenerReceivesMessage**  
   **目的**: 测试消息监听器是否能正确接收消息。  
   **关键点**: 向队列添加消息并确保监听器能够正确接收到消息内容。
2. **testOnMessageChanged\_EmptyQueue\_NoMessageReceived**  
   **目的**: 测试队列为空时，监听器不会接收到任何消息。  
   **关键点**: 在空队列上注册监听器并确认没有接收到消息。

### ****5. MessageFactoryTest（MessageFactory测试）****

1. **testCreateMessage\_ValidContent\_MessageIsCreated**  
   **目的**: 测试 MessageFactory 是否能够正确创建消息对象。  
   **关键点**: 使用工厂方法创建消息，确保消息内容符合预期。
2. **testCreateMessageQueue\_ValidQueue\_MessageQueueIsCreated**  
   **目的**: 测试 MessageFactory 是否能够正确创建消息队列。  
   **关键点**: 使用工厂方法创建消息队列并确认队列有效。
3. **testCreateMessageCenter\_ValidCenter\_MessageCenterIsCreated**  
   **目的**: 测试 MessageFactory 是否能够正确创建消息中心。  
   **关键点**: 使用工厂方法创建消息中心并确认中心有效。

### ****6. ExampleTest（示例类测试）****

**testExample\_CompleteMessageFlow\_MessageIsSentAndReceived**  
**目的**: 测试完整的消息流转过程，包括消息的发送、接收、监听等。  
**关键点**: 注册多个队列和监听器，广播消息，验证消息是否被正确处理和转发。

### ****7. 总结****

这组测试全面覆盖了消息系统中的关键功能，确保以下几个方面的正确性：

1. **Message接口**：验证消息内容的获取和处理。
2. **MessageQueue接口**：验证队列的入队、出队操作，监听器的添加和移除。
3. **MessageCenter接口**：验证消息中心的队列注册、注销和消息广播功能。
4. **MessageListener接口**：验证消息监听器的工作机制，确保其正确处理队列中的消息。
5. **MessageFactory**：确保消息、队列和消息中心的创建过程没有问题。
6. **示例类**：验证消息流的完整性，确保消息能够在系统中正确传递。

这些测试确保了系统的各个模块的功能正常，并验证了消息在系统中的流转过程，保证了系统的可靠性和稳定性。

### 8 . 测试结果

1. **实现二测试**

### ****MessageSystemTests2 测试类总结****

此测试类主要验证了消息系统中的各个核心组件（消息、消息队列、消息中心、消息监听器等）的功能。

### ****1. MessageTest（Message接口测试）****

1. **testGetContent\_ValidMessage\_ReturnsCorrectContent**  
   **目的**: 验证 Message 对象能够正确返回消息内容。  
   **关键点**: 创建一个消息并验证其内容是否为 "Hello"。
2. **testGetContent\_EmptyMessage\_ReturnsEmptyString**  
   **目的**: 验证 Message 对象能够正确处理空字符串内容。  
   **关键点**: 创建一个空内容的消息并验证其内容是否为空。

### ****2. MessageQueueTest（MessageQueue接口测试）****

1. **testEnqueue\_ValidMessage\_MessageIsEnqueued**  
   **目的**: 测试消息是否可以成功入队。  
   **关键点**: 向队列添加消息并验证消息是否成功出队。
2. **testDequeue\_ValidMessage\_MessageIsDequeued**  
   **目的**: 测试从队列中成功出队消息。  
   **关键点**: 向队列添加消息并验证出队消息的内容。
3. **testDequeue\_EmptyQueue\_BlockAndTimeout**  
   **目的**: 测试在队列为空时，dequeue 是否会在超时后返回 null。  
   **关键点**: 启动一个线程执行 dequeue 操作并模拟超时场景。
4. **testAddListener\_ValidListener\_ListenerIsAdded**  
   **目的**: 测试监听器是否能正确添加并处理队列中的消息。  
   **关键点**: 向队列添加消息，验证监听器是否能够正确处理该消息。
5. **testRemoveListener\_ValidListener\_ListenerIsRemoved**  
   **目的**: 测试移除监听器后，监听器是否仍能接收消息。  
   **关键点**: 向队列添加消息，验证已移除的监听器是否处理该消息。

### ****3. MessageCenterTest（MessageCenter接口测试）****

1. **testRegisterQueue\_ValidQueue\_QueueIsRegistered**  
   **目的**: 测试消息队列是否能成功注册到消息中心。  
   **关键点**: 向消息中心注册队列并广播消息，确保消息正确广播。
2. **testUnregisterQueue\_RegisteredQueue\_QueueIsUnregistered**  
   **目的**: 测试已注册的队列是否能够从消息中心注销。  
   **关键点**: 注销队列并广播消息，确保广播不再发送给注销的队列。
3. **testBroadcast\_ValidMessage\_OnlyOneQueueReceivesMessage**  
   **目的**: 测试广播消息时，只有注册的队列会接收到消息。  
   **关键点**: 广播消息到某一个队列，确保只有该队列的监听器能收到消息。
4. **testBroadcast\_EmptyQueue\_NoMessagesReceived**  
   **目的**: 测试广播消息到空队列时的行为。  
   **关键点**: 广播消息到未注册队列，确保不会抛出异常。

### ****4. MessageListenerTest（MessageListener接口测试）****

1. **testOnMessageChanged\_ValidMessage\_ListenerReceivesMessage**  
   **目的**: 测试消息监听器是否能正确接收消息。  
   **关键点**: 向队列添加消息，验证监听器是否能正确接收到消息内容。
2. **testOnMessageChanged\_EmptyQueue\_NoMessageReceived**  
   **目的**: 测试当队列为空时，监听器是否接收到任何消息。  
   **关键点**: 检查队列为空时监听器是否没有接收到消息。

### ****5. MessageFactoryTest（MessageFactory测试）****

1. **testCreateMessage\_ValidContent\_MessageIsCreated**  
   **目的**: 测试 MessageFactory 是否能正确创建消息。  
   **关键点**: 使用工厂方法创建消息并验证其内容是否正确。
2. **testCreateMessageQueue\_ValidQueue\_MessageQueueIsCreated**  
   **目的**: 测试 MessageFactory 是否能正确创建消息队列。  
   **关键点**: 创建消息队列并确保其有效。
3. **testCreateMessageCenter\_ValidCenter\_MessageCenterIsCreated**  
   **目的**: 测试 MessageFactory 是否能正确创建消息中心。  
   **关键点**: 创建消息中心并确保其有效。

### ****6. ExampleTest（示例类测试）****

**testExample\_CompleteMessageFlow\_MessageIsSentAndReceived**  
**目的**: 测试完整的消息传递流程，包括消息的发送、接收和监听。  
**关键点**: 注册多个队列和监听器，广播消息到指定队列，并验证消息的流向。确保只有指定的队列接收到消息。

### ****7. 总结****

这组测试涵盖了消息系统的核心功能，确保了以下几个方面的正确性：

1. **Message接口**：验证消息内容的正确性。
2. **MessageQueue接口**：验证消息队列的入队、出队操作以及监听器的处理机制。
3. **MessageCenter接口**：验证消息中心的队列注册、注销和消息广播功能。
4. **MessageListener接口**：验证监听器能否在消息变化时收到相应的消息。
5. **MessageFactory**：验证消息、队列和消息中心的创建功能。
6. **示例类**：验证消息的完整传递流程，确保消息从生产者到消费者的正确传递。

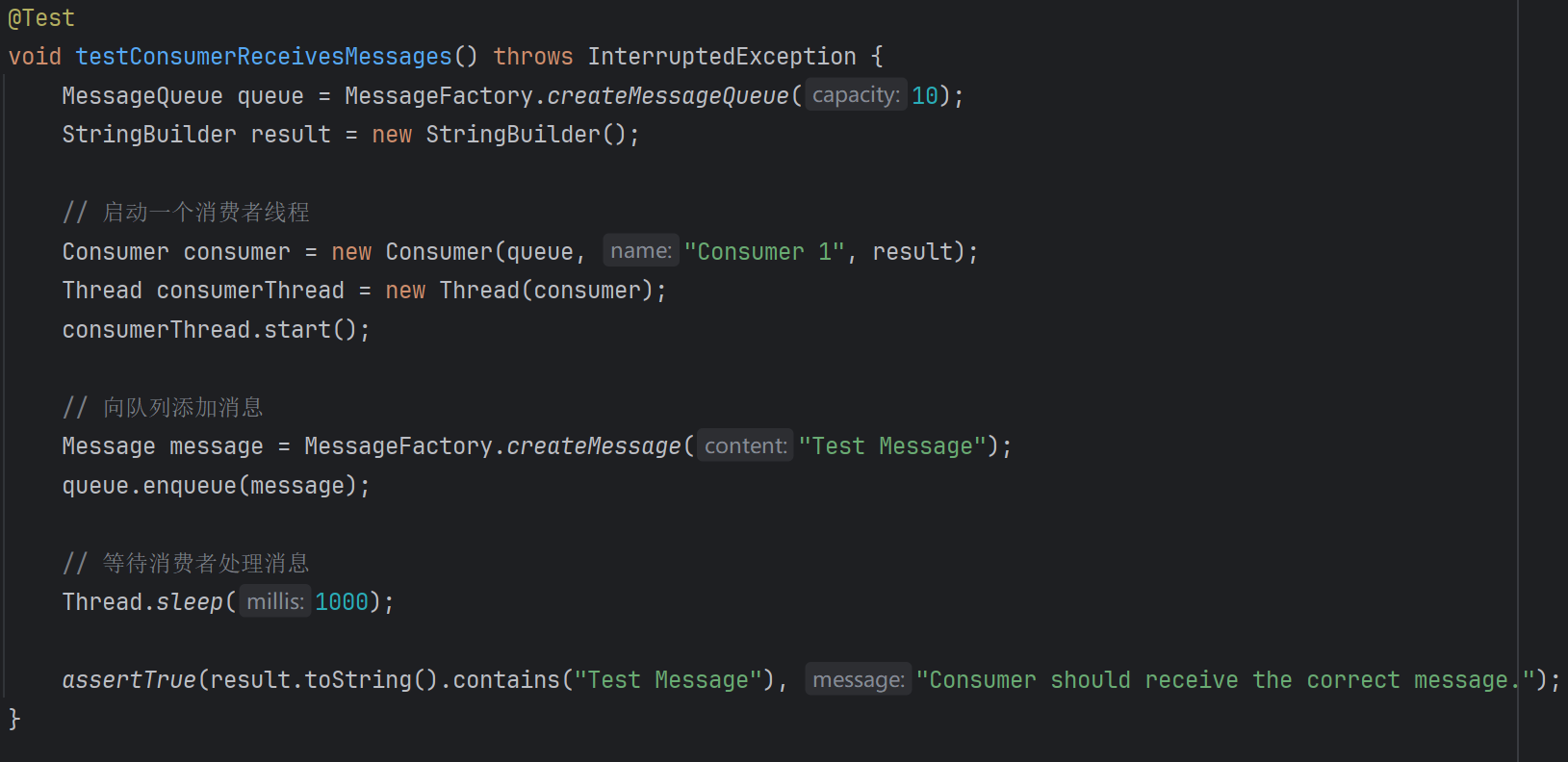
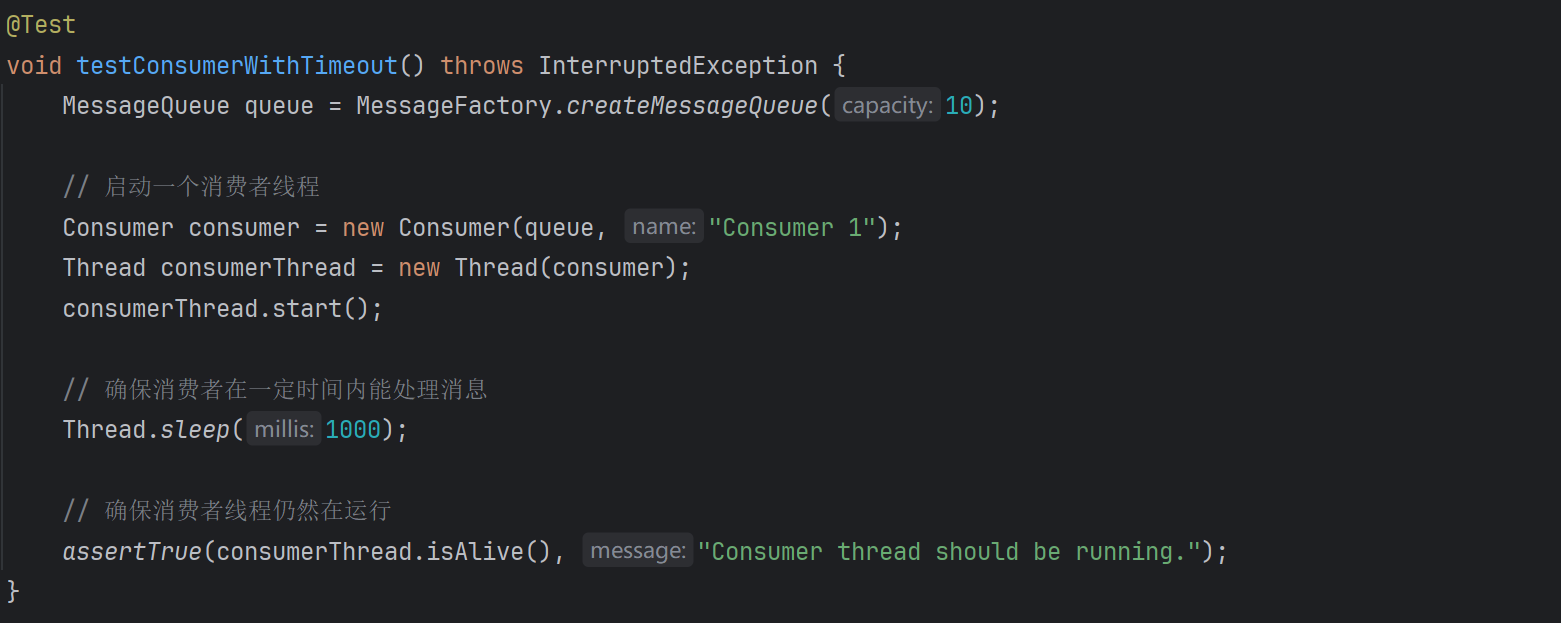
这些测试确保了系统的基本功能，如消息的正确创建、队列的管理、消息的发送与接收、以及监听器和消息中心的协作等，确保了消息系统的可靠性和稳定性。

### 8 . 测试结果

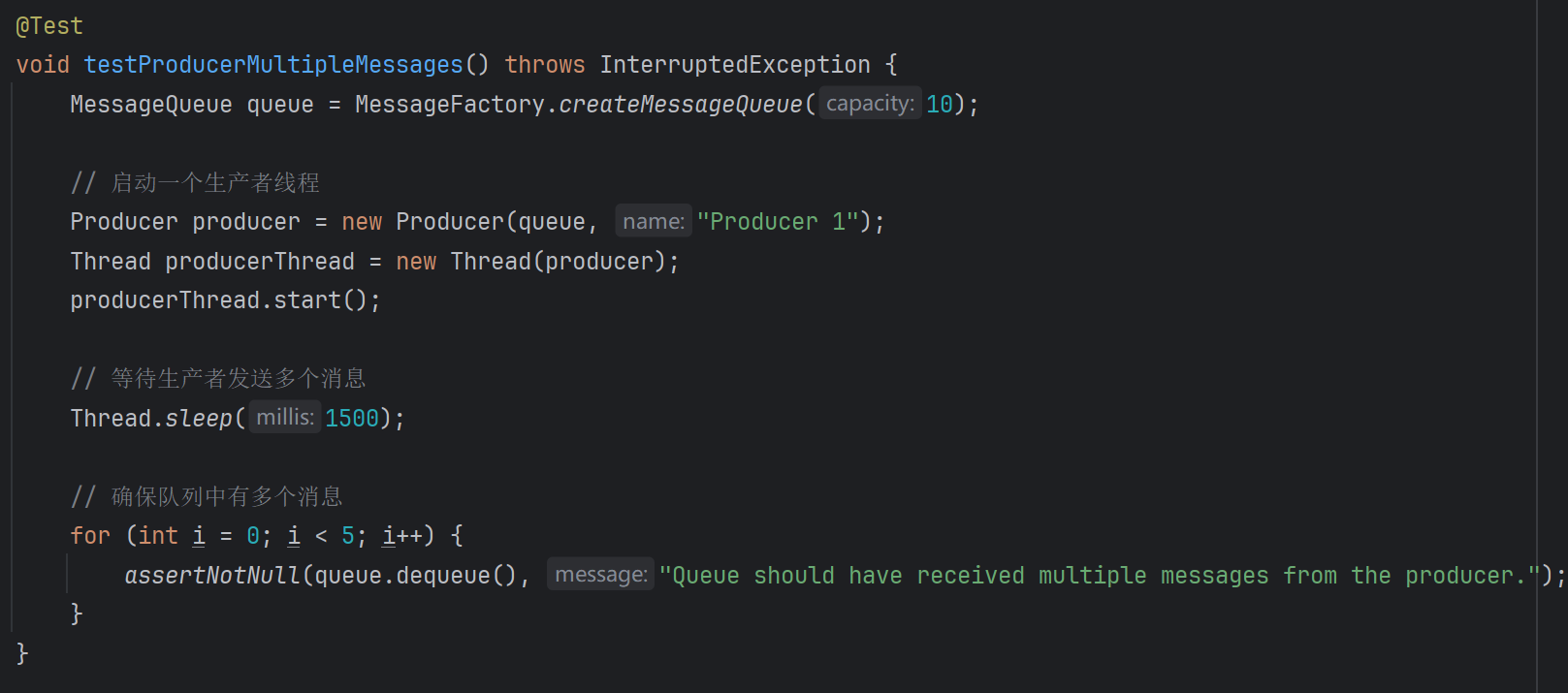
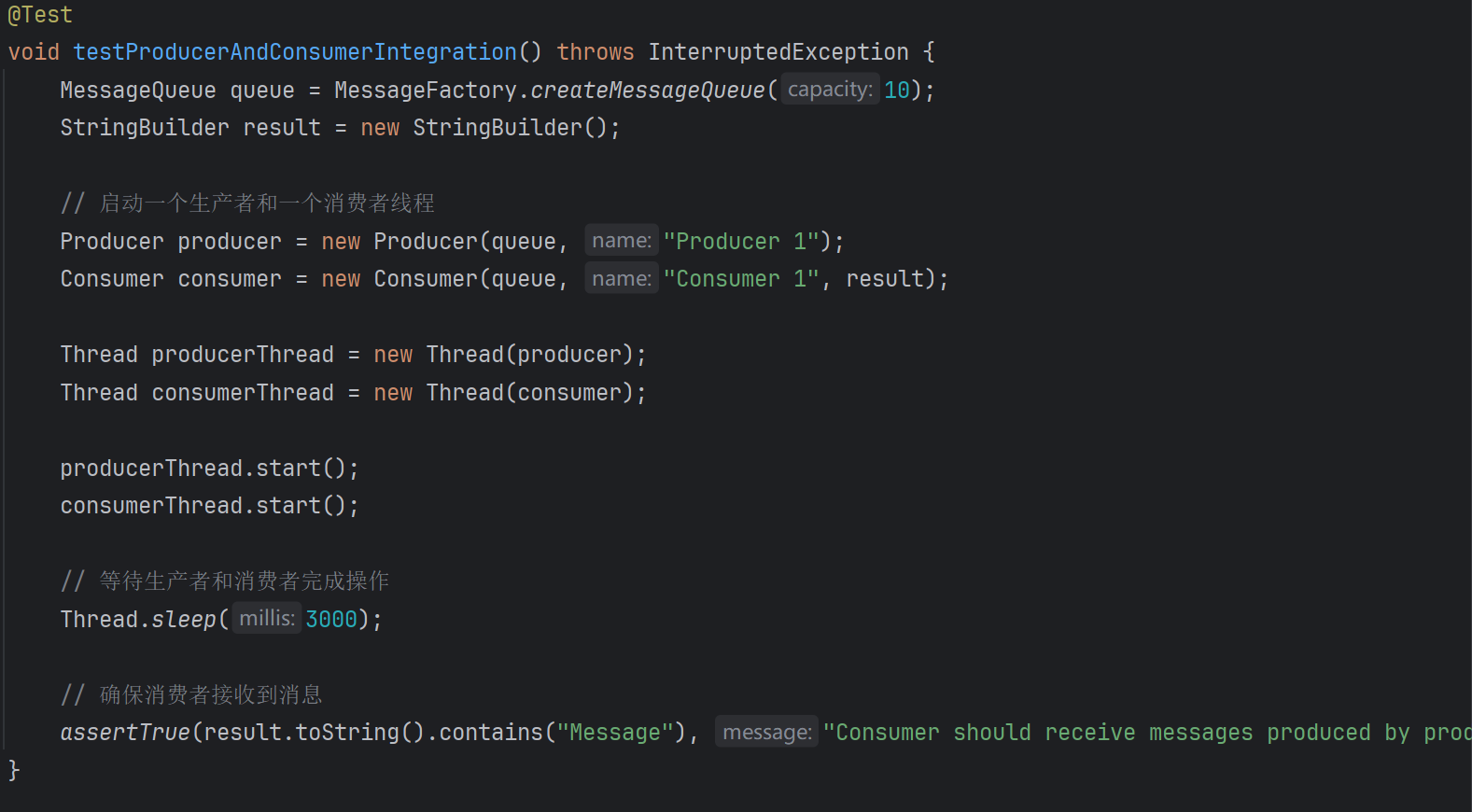


1. **实现二消费者生产者测试**

### 1. ****ConsumerTest（消费者测试）****

1. **testConsumerReceivesMessages**  
   **目的**: 测试消费者是否能够正确接收生产者发送的消息。  
   **关键点**: 启动一个消费者线程，向队列添加单条消息，验证消费者是否能接收到该消息。
2. **testConsumerMultipleMessages**  
   **目的**: 测试消费者是否能够正确接收多个消息。  
   **关键点**: 启动消费者线程，向队列添加多个消息，验证消费者是否成功处理所有消息。
3. **testConsumerWithTimeout**  
   **目的**: 测试消费者能否在一定时间内处理消息，并且线程是否仍在运行。  
   **关键点**: 启动消费者线程并验证线程是否仍然在处理消息，确保消费者能在指定时间内持续工作。

### 2. ****ProducerTest（生产者测试）****

1. **testProducerSendsMessages**  
   **目的**: 测试生产者是否能够成功地将消息发送到消息队列。  
   **关键点**: 启动生产者线程，等待一段时间后验证消息是否被放入队列中。
2. **testProducerMultipleMessages**  
   **目的**: 测试生产者是否能连续发送多条消息。  
   **关键点**: 启动生产者线程，等待一段时间后检查队列中是否有多条消息。
3. **testProducerAndConsumerIntegration**  
   **目的**: 测试生产者和消费者的集成是否正常。确保生产者发送消息，消费者能够接收到并处理这些消息。  
   **关键点**: 同时启动生产者和消费者线程，等待他们完成操作后检查消费者是否接收到消息。

### 3. ****Producer-Consumer 并发测试****

**testMultipleProducersAndConsumers**  
**目的**: 测试多个生产者和多个消费者并发工作时是否能够正确处理消息。  
**关键点**: 启动多个生产者和消费者线程，模拟并发场景，确保消费者能够正确接收来自多个生产者的消息。

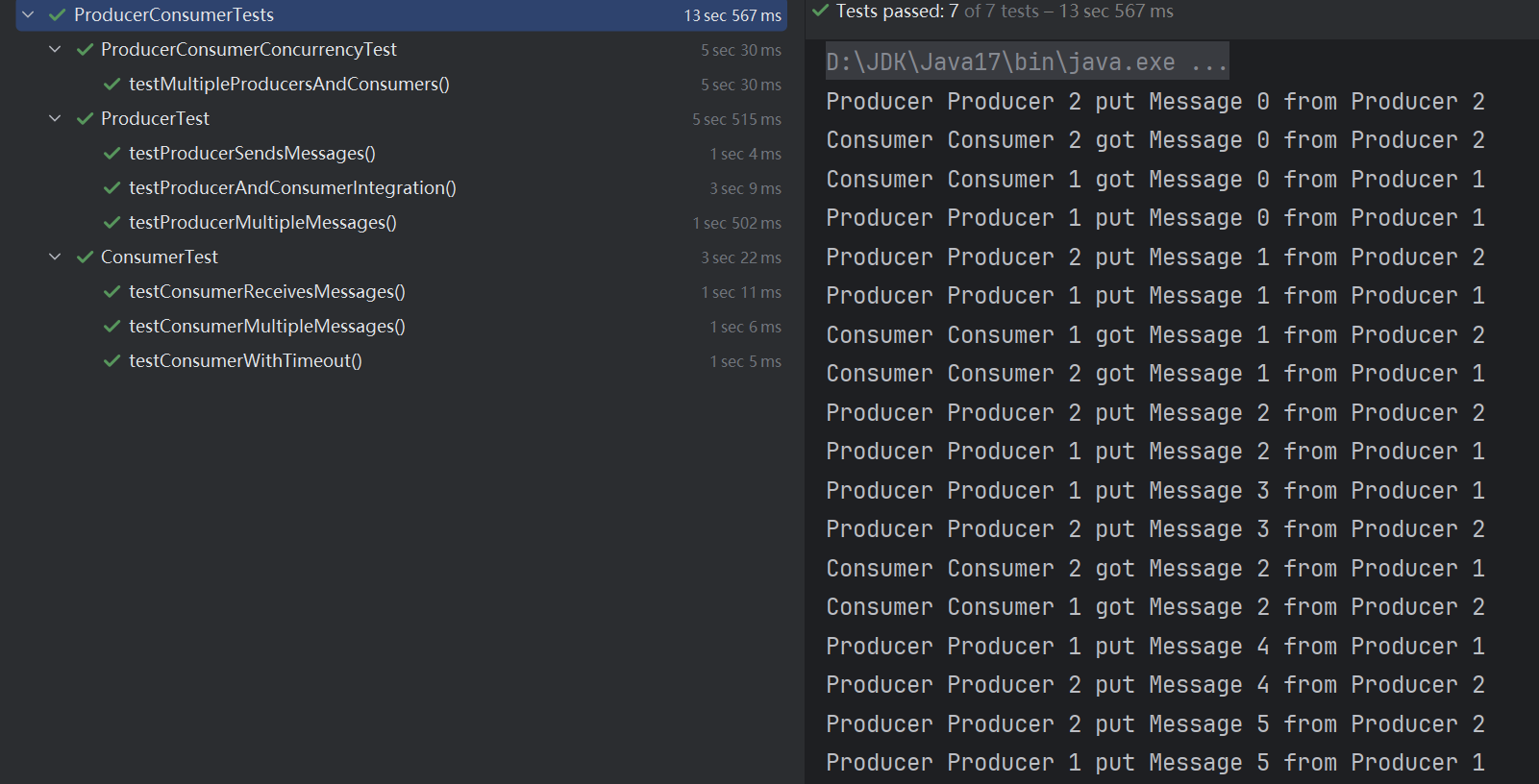
### 4. 总结：

这组测试的核心目的是验证 **生产者** 和 **消费者** 在消息队列中的正确交互，包括单线程和多线程的情况。测试涵盖了以下几个方面：

1. **消费者**是否能正确接收单个和多个消息，并且能够在特定时间内处理消息。
2. **生产者**是否能成功发送消息到队列，并且能够发送多个消息。
3. **生产者与消费者的集成**，确保消息能够正确从生产者流向消费者。
4. **并发情况下的消息传递**，确保多个生产者和消费者能正确地并发工作，并且每个消费者都能接收到它们应该接收的消息。

这些测试验证了消息的发送、接收和处理的顺序以及并发情况下的稳定性。

### 5. 测试结果



1. **实现三消费者生产者测试**

以下是该 ProducerConsumerTest 测试文件中每个测试方法的目的总结：

### 1. ****testProducerConsumerIntegration（生产者消费者集成测试）****

1. **目的**: 测试生产者和消费者的基本集成。验证生产者通过消息中心发送消息后，消费者是否能正确接收到消息，并且消息的内容是否按预期顺序（从0到9）到达。
2. **关键点**: 使用单独的线程运行生产者，验证消费者接收到的消息数量和内容。

### 2. ****testProducerConsumerConcurrency（生产者消费者并发测试）****

1. **目的**: 测试多个生产者并发工作时，消费者是否能够正确接收所有消息。模拟并发场景，使用线程池并行运行多个生产者，并确保消费者接收到所有消息。
2. **关键点**: 使用 ExecutorService 来运行多个生产者线程，验证消费者能接收到所有20条消息。

### 3. ****testMessageOrder（消息顺序测试）****

1. **目的**: 测试消费者是否按正确的顺序接收到生产者发送的消息。确保消息的顺序与生产者发送的顺序一致。
2. **关键点**: 通过检查消息的内容，确保消费者接收到的每条消息的内容与预期的顺序相匹配（即从0到9）。

### 4. ****testMessageListenerRemoval（消息监听器移除测试）****

1. **目的**: 测试移除消息监听器后的行为。确保在移除监听器后，生产者发送消息时，已经移除的监听器不再接收到任何消息。
2. **关键点**: 创建一个临时消费者并移除其监听器，发送消息后确保该消费者未接收到任何消息。

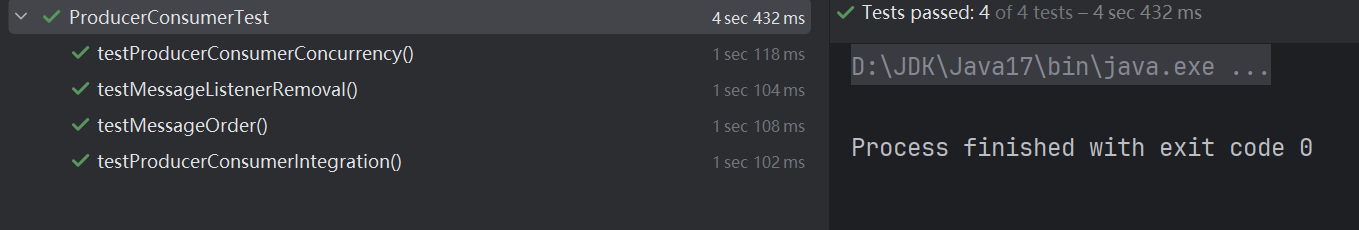
### 5. 总结：

这些测试方法主要关注生产者和消费者之间的消息传递及其相关操作的正确性，验证了以下几点：

1. 生产者和消费者的集成能否正常工作。
2. 并发情况下生产者和消费者是否能保持正确的消息处理。
3. 消息的顺序是否得以保持。
4. 消息监听器的移除是否能正确生效。

这些测试确保了在多线程环境下，消息系统的基本功能（如消息发送、接收、顺序、并发处理和监听器管理）能按照预期正确运行。

### 6. 测试结果



1. 基础优化介绍

优化部分（第三个实现）主要与同样是多线程的第二个实现进行对比。

### 接口优化

**原实现问题：**

原有实现中，使用Listrner来作为监听者基类，所有监听者必须实现onMessageChanged方法来实现事件触发。

**新实现解决办法：**

使用了lambda表达式的方式，实现了函数式编程，监听者直接传入一个函数供事件触发时调用

#### ****改进和好处：****

1. **动态增删**：新实现在注册事件后会返回一个唯一的注册hash值，监听者可以使用这个hash值传入事件中心从而动态删除之前的的监听事件，并且传入新的监听策略，避免了耦合。
2. **扩展适用范围**：原实现中，每一个监听者只可以实现一次onMessageChanged，从而导致不可以同时对多个事件做监听。新的实现可以传入不同的string用于区分不同的事件信道。

### ****消息队列和监听机制的优化****

**原实现问题：**

1. SimpleMessageQueue 使用 BlockingQueue<Message>（具体是 LinkedBlockingQueue）来存储消息，并通过 addListener() 和 removeListener() 方法添加和移除监听器。监听器会在每次队列有消息变化时被触发。
2. 监听器的执行是同步的，即每次有消息变化时，监听器会在主线程中被调用。

**新实现解决办法：**

1. ImprovedSimpleMsgQueue 使用 ExecutorService（线程池）来异步处理消息队列的监听器回调。通过 submit() 方法将消费者任务提交给线程池，避免了在主线程中执行回调，减少了阻塞和性能开销。
2. 通过 ConcurrentHashMap 和 AtomicInteger 来管理监听器和消费者，支持高并发并且避免了资源冲突。

#### ****改进和好处：****

1. **异步执行**：通过线程池 ExecutorService，ImprovedSimpleMsgQueue 在处理监听器回调时不会阻塞主线程，能更好地支持高并发环境下的消息处理。
2. **性能提升**：减少了线程的等待时间和阻塞，提高了系统的响应速度和吞吐量，尤其是在高并发情况下，能够充分利用多核 CPU。
3. **高并发支持**：通过 ConcurrentHashMap 和 AtomicInteger 的线程安全设计，保证了在并发环境中对消费者的管理是安全的。

### ****3. 消息中心的改进****

**原实现问题：**

1. SimpleMessageCenter 是一个单例类，管理多个 MessageQueue 实例，并提供消息的广播功能。
2. 它使用 synchronized 锁来保证线程安全，使用 queues.add(queue) 等方式注册队列。

**新实现解决办法：**

1. ImprovedSimpleMsgCenter 通过 **静态内部类单例模式** 实现了线程安全的单例，避免了 synchronized 的使用，从而提高了性能。
2. 使用了 ConcurrentHashMap 来存储多个消息队列，支持高并发下对队列的操作（比如 computeIfAbsent() 方法避免了重复创建队列）。

#### ****改进和好处：****

1. **更高效的单例模式**：采用 **静态内部类单例模式**，这是一种更加高效且线程安全的单例实现方式，避免了不必要的同步操作，提升了性能。
2. **高并发支持**：通过 ConcurrentHashMap 来存储队列，实现了在高并发场景下对队列的安全访问。相比 ArrayList 等普通集合，ConcurrentHashMap 能更好地应对并发写入和读取的场景。

### ****4. 消息类型的改进****

**原实现问题：**

使用了 SimpleMessage 类来表示消息，其中只有一个字符串类型的内容。Message 接口定义了 getContent() 方法，返回消息内容。

**新实现解决办法：**

使用了更灵活的 ImprovedMessage 接口，能够支持多种类型的消息（如 IntMessage）。IntMessage 存储一个整数，且 ImprovedMessage 的 getContent() 方法返回的是一个通用的 Object 类型，能支持多种数据类型。

#### ****改进和好处：****

1. **灵活性增强**：通过 ImprovedMessage 接口，可以支持更多类型的消息（不仅限于字符串）。比如，IntMessage 可以表示一个整数消息，未来可以轻松地扩展为支持其他类型（如浮点数、JSON 等）。
2. **类型安全**：ImprovedMessage 通过泛型或多态可以更好地支持不同类型的消息，并保证消息类型的一致性和类型安全。

### ****5. 消息缓冲区的设计改进****

1. **原实现问题：**

SimpleMessageQueue 类没有明确的缓冲区实现，消息直接通过队列存储并等待被消费者处理。

1. **新实现解决办法：**

ImprovedSimpleMsgBuffer 类作为缓冲区存在，它提供了 Optional<ImprovedMessage> 作为消息的返回类型。如果没有消息可供获取，返回 Optional.empty()，避免了 null 值的出现。

#### ****改进和好处：****

1. **更安全的消息获取**：通过 Optional<ImprovedMessage> 的使用，避免了 null 值的问题，减少了 NullPointerException 的风险，使得代码更加健壮。
2. **更清晰的设计**：通过引入 ImprovedMessageBuffer，代码结构更加清晰，分工明确，消息的存储和获取被清晰地分开，利于代码的扩展和维护。

### ****6. 生产者和消费者的改进****

**原实现问题：**

Producer 类直接在主线程中生成消息并放入队列，消费者则通过阻塞的方式从队列中取出消息进行处理。

**新实现解决办法：**

SimpleProducer 和 SimpleCustomer 类中的生产者和消费者逻辑并未做太大改变，但可以看出，消息发送和接收在 ImprovedSimpleMsgCenter 和 ImprovedSimpleMsgQueue 的配合下，支持了更高效的消息处理机制。

#### ****改进和好处：****

1. **线程池的使用**：虽然 Producer 和 Consumer 的逻辑基本相同，但消息的异步处理和多线程支持使得消息处理更加高效。
2. **更好的扩展性**：通过使用 ImprovedMessageCenter 和 ImprovedMessageQueue，系统更容易扩展，如增加更多的消息类型和队列处理机制。

### ****7. 总结对比****

#### ****原实现问题的优点：****

1. **简单易懂**：设计较为直观，适合小规模的消息系统。
2. **适用场景**：适合低并发或者不需要高性能的简单应用。

#### ****新实现解决办法的改进和优势：****

1. **高并发支持**：通过 ExecutorService 线程池和 ConcurrentHashMap，改进后的实现能在高并发场景下更高效地处理消息和监听器。
2. **异步消息处理**：异步消息队列和消费者回调机制，减少了主线程的阻塞，提高了系统的吞吐量和响应速度。
3. **消息类型和缓冲区的灵活设计**：通过 ImprovedMessage 接口和 ImprovedMessageBuffer，可以轻松扩展和处理不同类型的消息，并避免了 null 值带来的问题。
4. **更优的单例实现**：使用 **静态内部类单例模式**，更高效且线程安全。
5. **更清晰的代码结构**：通过清晰的接口和类划分，系统更加模块化，便于维护和扩展。

总体来说，新实现解决办法相较于原实现问题，具备更好的性能和可扩展性，尤其适合于高并发场景中的消息处理。

1. 我的进阶作业介绍（基于RabbitMQ微服务）

这段代码实现了一个基于 **Spring Boot** 和 **RabbitMQ** 的物流系统，其中包括订单管理、仓库管理、配送管理等模块。整个系统通过消息队列实现模块间的通信，利用 RabbitMQ 作为中间件。

1. **类图设计和时序图设计**

图 7 类图

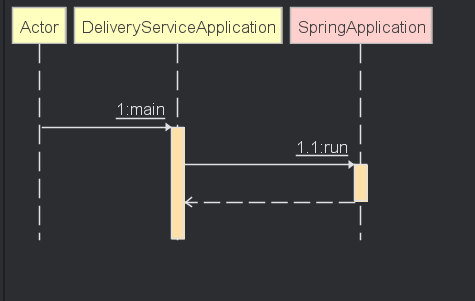


图 8 时序图1



图 9 时序图2

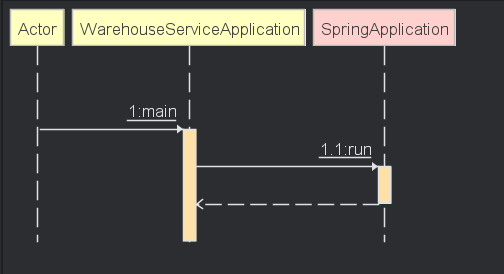


图 10 时序图3

### 1. ****RabbitConfig.java**** - 配置消息队列

该类负责定义和配置 RabbitMQ 相关的消息队列、交换机和绑定关系。

1. **logisticsExchange()**: 创建一个名为 logistics\_exchange 的 TopicExchange。TopicExchange 支持根据路由键将消息路由到队列。
2. **orderManagementQueue()**, **warehouseQueue()**, **deliveryQueue()**: 定义了三个消息队列，分别用于订单管理、仓库管理和配送管理。每个队列都被标记为持久化（true），确保队列在 RabbitMQ 重启后不丢失。
3. **orderManagementBinding()**, **warehouseBinding()**, **deliveryBinding()**: 这些方法将队列与交换机绑定，并设置路由键。当消息的路由键与绑定的路由键匹配时，消息会被发送到相应的队列。例如：
   * order.placed 路由键绑定到 order\_management\_queue。
   * warehouse.shipped 路由键绑定到 warehouse\_queue。
   * delivery.delivered 路由键绑定到 delivery\_queue。

### 2. ****PackageStatusUpdateMessage.java**** - 消息实体类

该类定义了包裹状态更新的消息格式。

* **字段**:
  + packageId: 包裹的唯一标识符。
  + status: 包裹当前的状态（如 order\_placed, warehouse\_shipped 等）。
  + timestamp: 消息生成的时间戳。

该类实现了 Serializable 接口，允许消息在不同的组件之间进行传输。

### 3. ****JsonMessageConverter.java**** - 消息转换器

该类负责将 Java 对象和 RabbitMQ 消息之间进行转换。

1. **toMessage()**: 将 PackageStatusUpdateMessage 对象转换为 Message 对象，并将其序列化为 JSON 格式。
2. **fromMessage()**: 将接收到的 Message 对象反序列化为 PackageStatusUpdateMessage 对象。

### 4. ****DeliveryService.java**** - 配送服务

该服务负责处理来自 warehouse\_queue 的消息，并在包裹配送完成后发送相应的通知。

1. **handleWarehouseShipped()**: 使用 @RabbitListener 注解监听 warehouse\_queue 队列。该方法接收来自仓库的消息，打印日志并处理配送相关的逻辑。
2. **sendDeliveryCompleted()**: 在包裹配送完成后，打印配送完成的日志，模拟配送完成的操作。

### 5. ****OrderCenter.java**** - 订单中心

该控制器处理外部请求，接收订单创建请求并将其发送到 warehouse\_queue，然后让仓库进行处理。

* **createOrder()**: 处理 POST 请求，接收一个 packageId，创建一个 PackageStatusUpdateMessage 对象，设置状态为 order\_placed，并将消息发送到 RabbitMQ 的 logistics\_exchange 交换机，路由键为 order.placed。

### 6. ****OrderCenterApplication.java**** - 启动类（订单中心）

这是一个 Spring Boot 启动类，负责启动订单中心服务。

### 7. ****WarehouseService.java**** - 仓库服务

该服务负责接收来自 order\_management\_queue 队列的订单创建请求，处理订单并将其转发到配送队列。

1. **handleOrderPlaced()**: 使用 @RabbitListener 注解监听 order\_management\_queue 队列。收到订单创建消息后，仓库处理包裹并将包裹发送到配送模块。
2. **sendToDelivery()**: 处理完订单后，发送消息到 logistics\_exchange 交换机，路由键为 warehouse.shipped，告知配送模块包裹已出库。

### 8. ****WarehouseServiceApplication.java**** - 启动类（仓库服务）

这是一个 Spring Boot 启动类，负责启动仓库服务。

### 9. 整体流程

1. **订单创建**: 用户通过 API 调用 /order/create 创建订单，OrderCenter 接收到请求后，将订单信息发送到 RabbitMQ 的 order\_management\_queue 队列。
2. **仓库处理**: WarehouseService 监听 order\_management\_queue，一旦收到订单创建消息，仓库开始处理包裹，并将包裹的状态更新为 warehouse\_shipped，然后将消息发送到 warehouse\_queue。
3. **配送处理**: DeliveryService 监听 warehouse\_queue，一旦收到包裹出库的消息，处理配送过程并发送配送完成的通知。

### 10. 总结

这个系统使用消息驱动架构，通过 RabbitMQ 作为消息中间件实现模块之间的解耦。每个服务只关心自身的功能（如订单处理、仓库管理或配送管理），通过消息队列进行通信。系统的扩展性和可靠性较强，适合处理高并发和分布式任务。

1. 运行截图

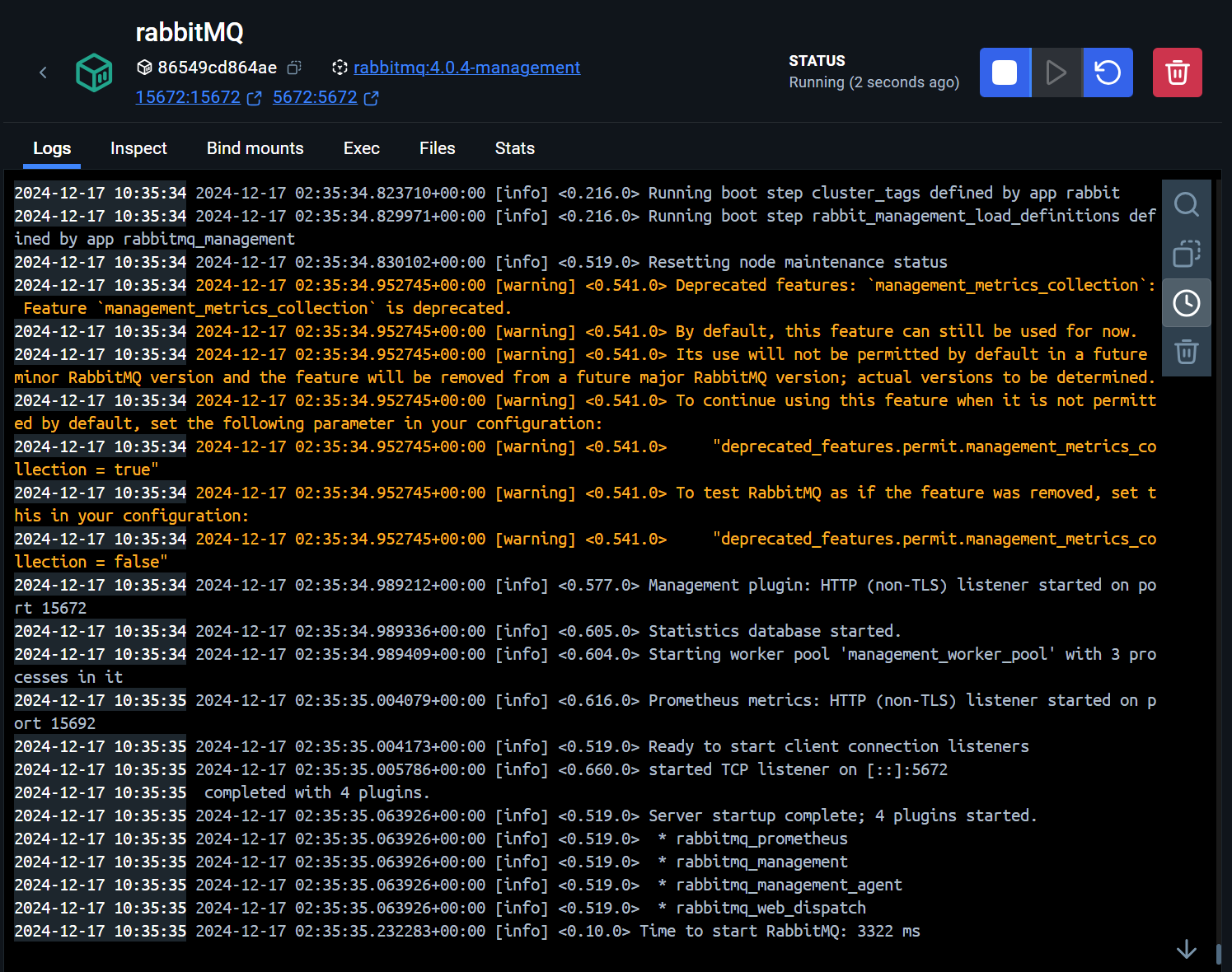


图 11 rabbitMQ

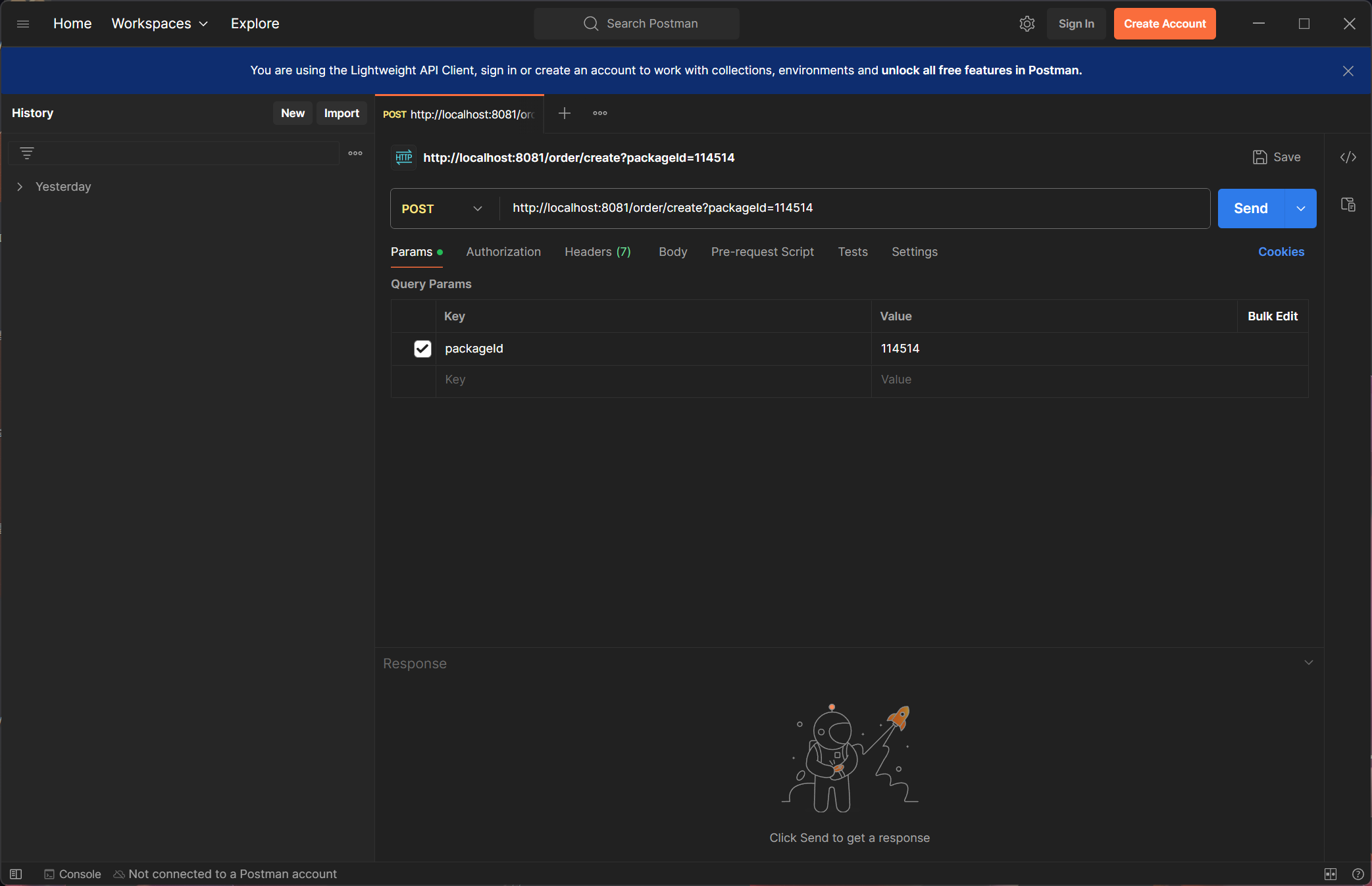


图 12 postman 测试

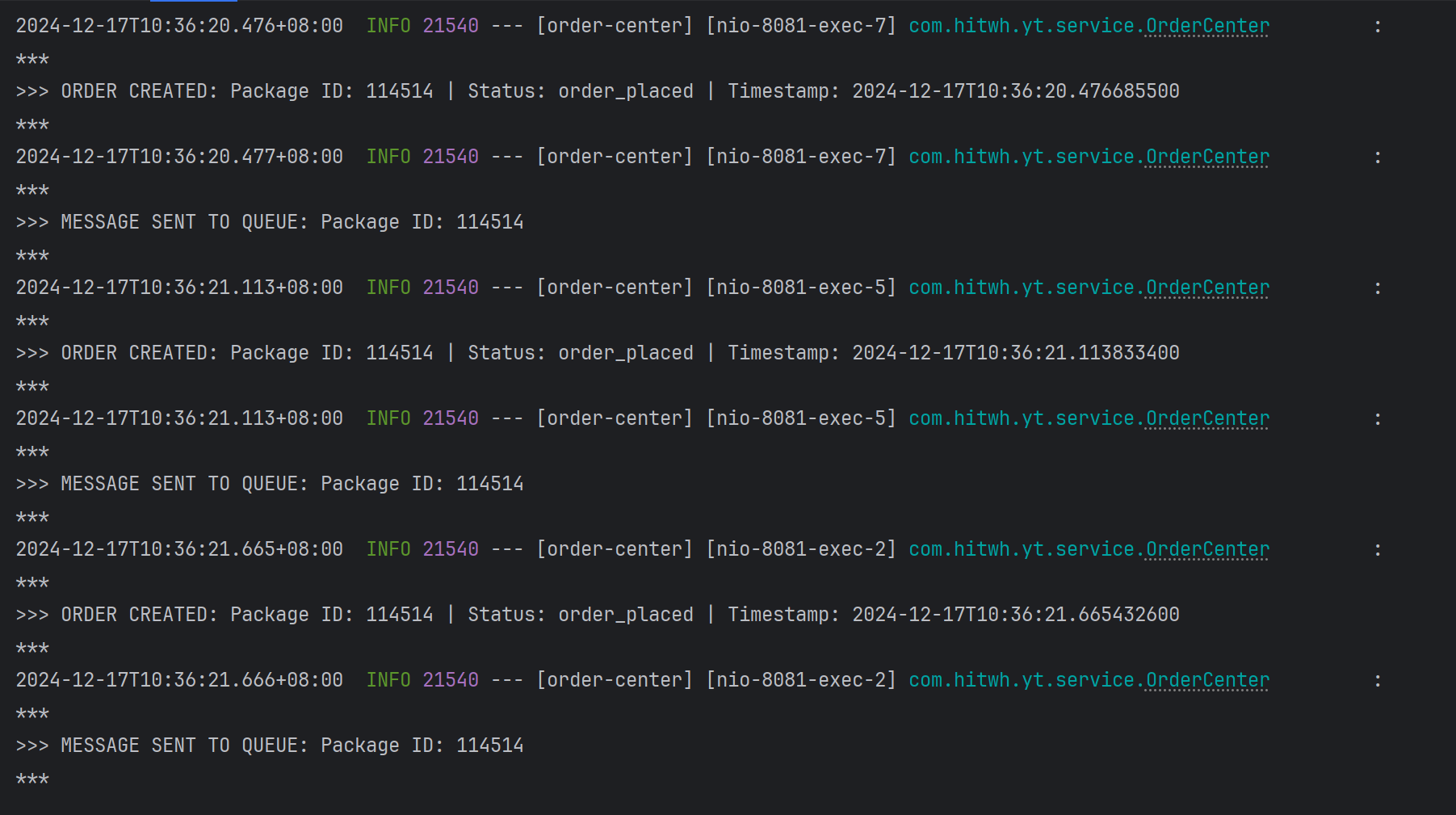


图 13 Order服务

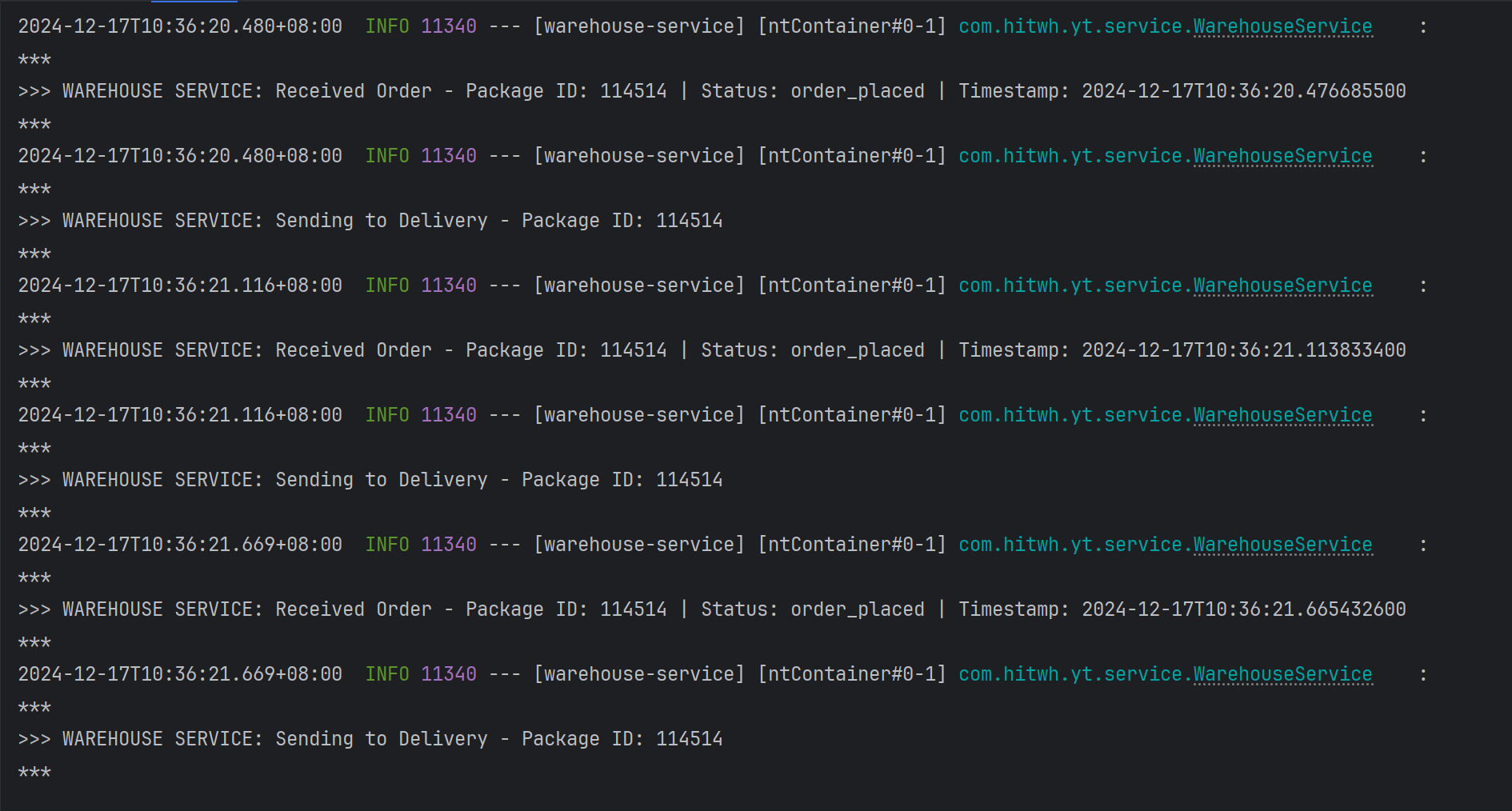


图 14 Warehouse服务

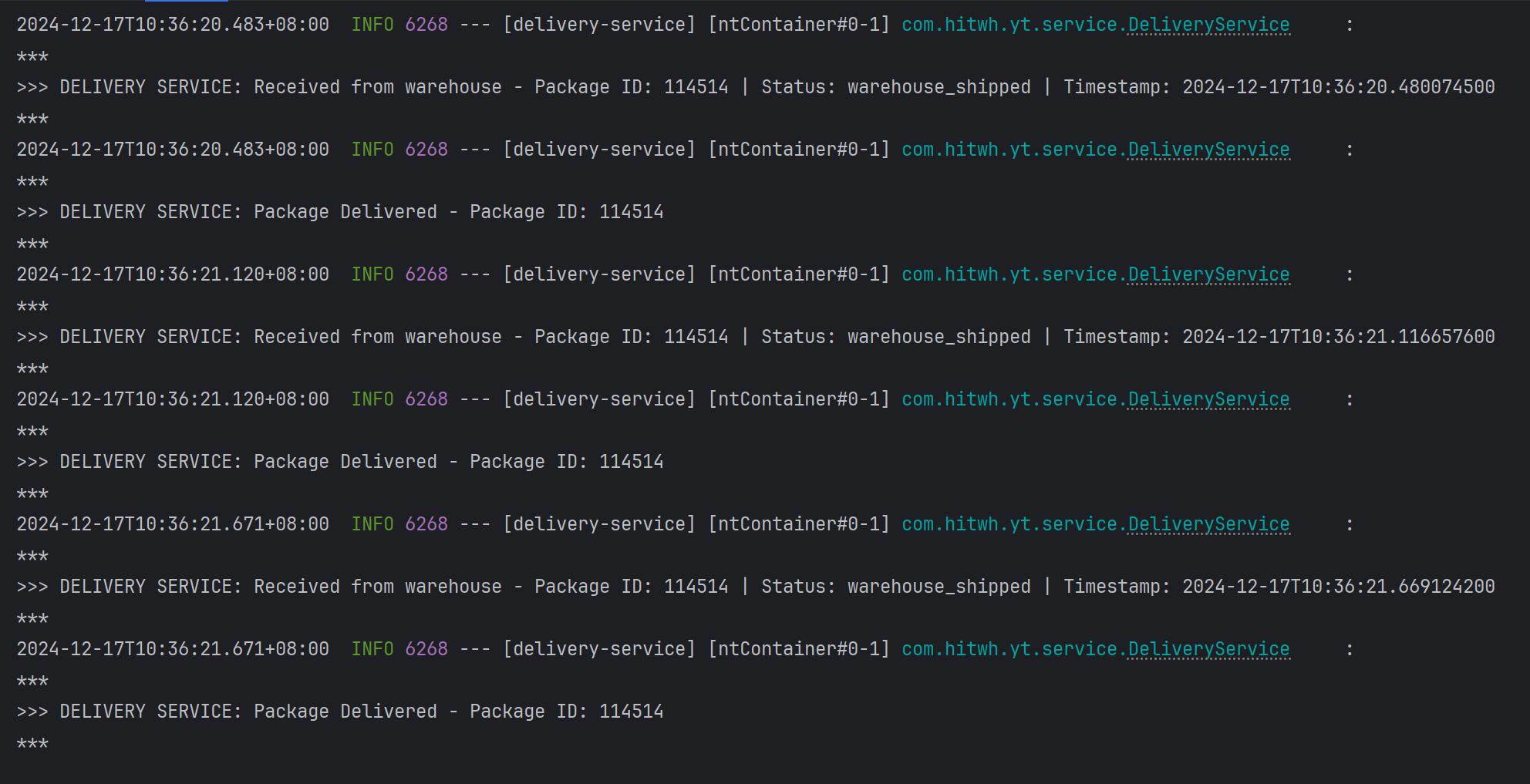


图 15 Delivery服务

1. 给出的进阶作业代码问题

### 1. ****线程死循环问题（死锁）****

1. **问题描述**：在Consumer类中的run方法里，消费者线程不断地从队列中获取消息并打印，但没有适当的退出条件。尽管在Main类中有注销消息队列的代码，但消费者线程依然会持续运行，导致死循环。
2. **改进建议**：消费者线程应当在消息队列被注销或程序退出时适当地停止，避免无谓的死循环。可以通过添加一个标志来控制消费者线程的退出，或者通过InterruptedException来处理中断信号。

### 2. ****线程同步问题****

1. **问题描述**：SimpleMessageQueue类中使用了BlockingQueue来管理消息队列，这本身是线程安全的。但在该类中依然存在一个lock对象（没有使用），这导致了代码的冗余，并可能引入同步问题。
2. **改进建议**：lock对象可以移除，因为BlockingQueue已经保证了线程安全。这样可以简化代码。

### 3. ****不必要的引用传递问题****

1. **问题描述**：在Producer类中的PackageState方法中，相同的LogisticsInfo对象被多次修改并加入队列。如果多个线程共享这个对象，可能会发生并发问题，导致传递的消息状态被异常更新。
2. **改进建议**：每次修改消息状态时，可以创建一个新的LogisticsInfo对象，以避免共享引用的问题，确保每个消息的独立性。

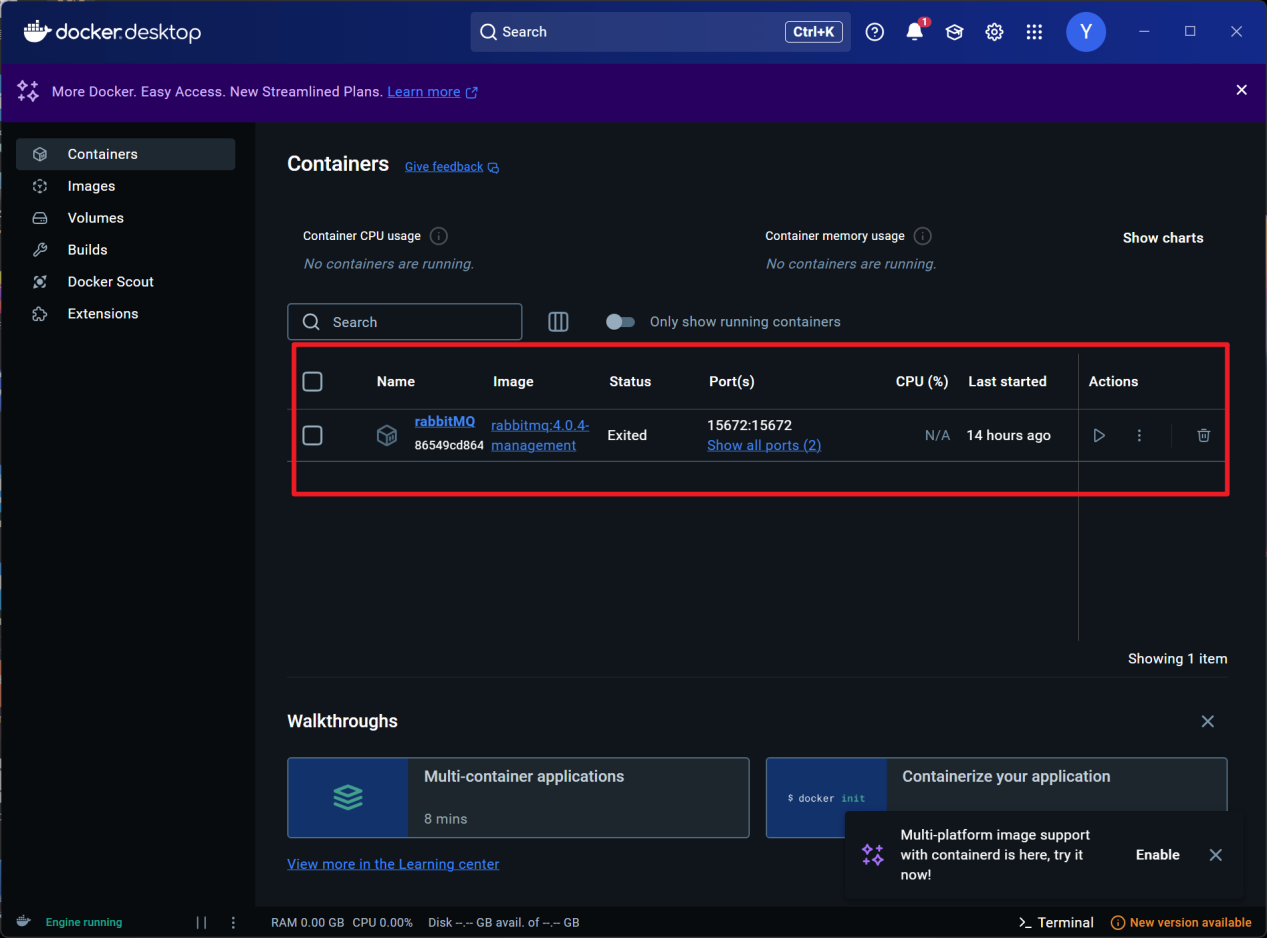
### 4. ****代码冗余****

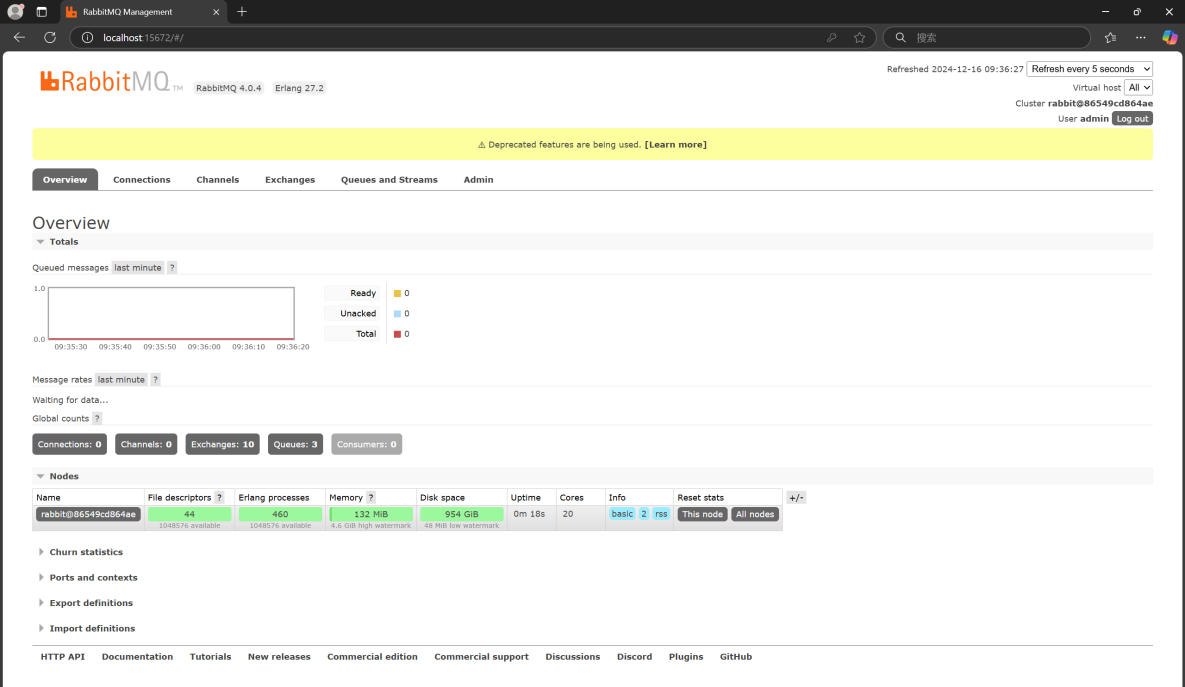
1. **问题描述**：SimpleMessageCenter类中的queues集合和registerQueue、unregisterQueue方法似乎多余。如果系统只使用一个队列来处理消息，可以简化设计，直接管理这个队列。
2. **改进建议**：简化SimpleMessageCenter类，改为只保存一个队列对象，而不需要维护一个队列集合。

### 5. ****线程池管理****

1. **问题描述**：在Main类中使用了ExecutorService创建一个固定大小的线程池，但线程池未在使用后关闭，可能会导致资源泄漏。
2. **改进建议**：在使用完线程池后，应该调用shutdown()来优雅地关闭线程池，避免资源泄漏。
3. 写基于RabbitMQ实现时遇到的问题和解决办法
4. RabbitMQ安装问题

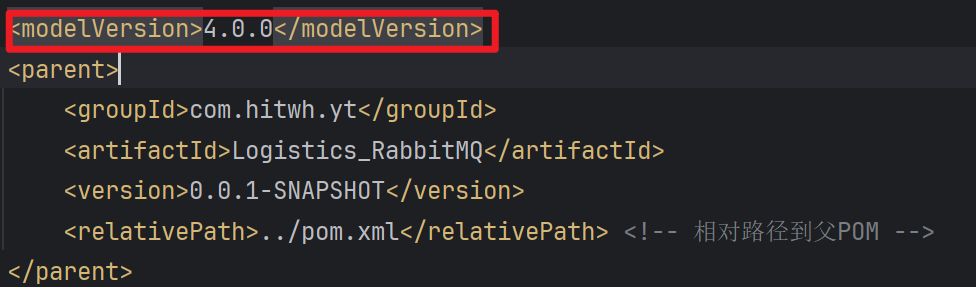
首先是安装位置，RabbitMQ基本适配了各个系统，包括win，linux，docker，mac等，原本打算直接使用官方的windows打包二进制文件，但是这个桌面版依赖与E语言编译器，而且版本不够新，于是放弃使用windows直接安装。又因为linux直接安装容易出一些环境问题，最终选择使用docker，在学了一上午docker后，成功通过安装dockerdesktop并在此基础上安装了RabbitMQ，成功打开网页管理。





1. Spring子项目的maven配置问题

接下来，因为需要让各个模块可以独立运行，也就是需要采用微服务架构，那就必须创建多个spring工程，为了简化和方便管理，使用idea的子模块功能，将每个子项目使用子模块来组织管理。由于我认为给每个项目都加一遍重复maven依赖过于冗余，由此想要创建一个父工程依赖，子工程都继承这些共同的依赖，并且如消息类的公共类也可也方便的组织在一起。但是遇到了非常多奇奇怪怪的报错，比如必须要加入一个modelVersion，而且必须是4.0才可以正确运行，最后依靠上网查询和询问，花了一些时间得以解决。



1. Spring项目启动注解扫描默认位置问题

这个问题相对比较简单，主要就是因为spring在启动的时候会扫描同一包和子包下的注解类，并且会通过反射等实例化出来。我遇到主要是因为把application类埋的过深，导致扫描不到我的service，最后将application类往外移动一层就可以扫描到了。

1. 自定义消息序列化问题

这个问题是由于Spring-AMQP需要对除了原生类型如String以外的类型就是我的自定义消息类进行序列化，序列化后传给运行的rabbitMQ程序，所以必须让自定义消息类继承Serializable。

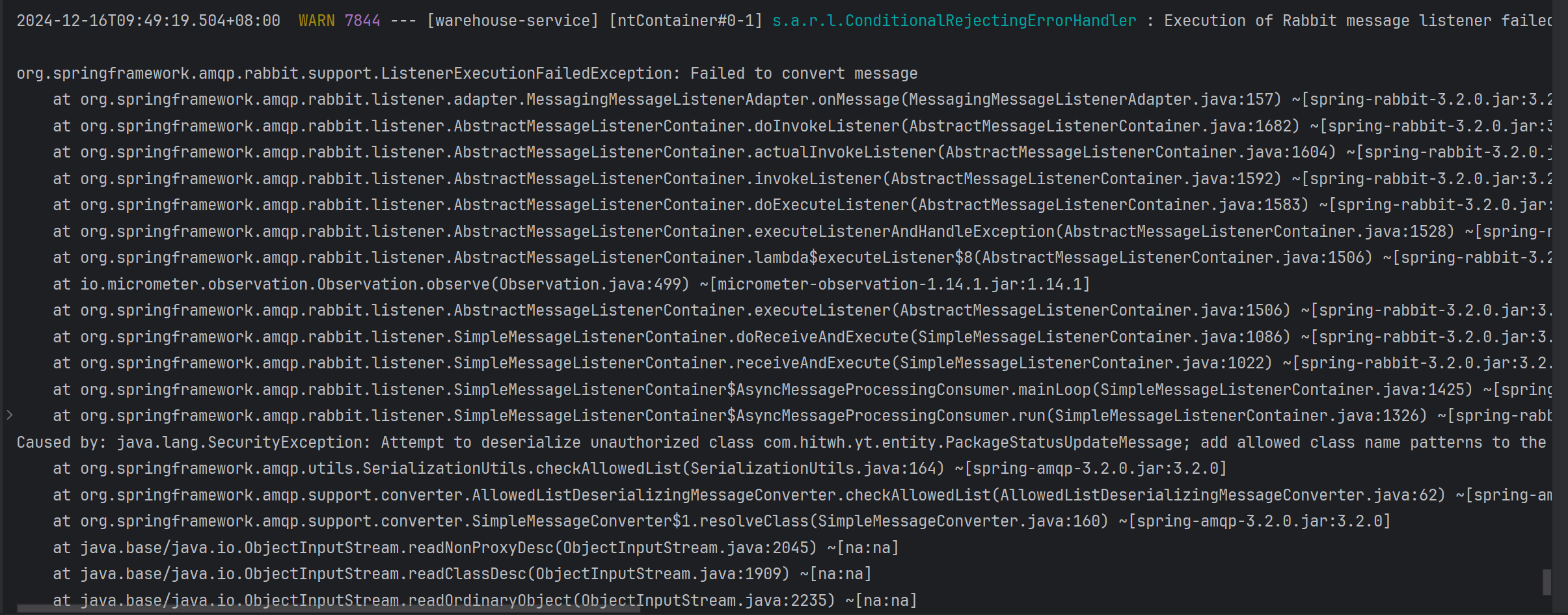
第二就是序列化方式，默认是二进制序列化，会有安全和浪费空间的问题，可以手动改为使用json序列化方式。

1. 自定义消息反序列化问题

最后这个问题也是困扰我最久的问题，到现在都没有解决。

这个问题表现形式是无法对消息进行发序列化，从而导致消费者彻底卡死，amqp会不断尝试并且打印报错，非常恐怖。报错的信息里面提到了几种解决办法：1.定义自己的convert转换器，专门对自己的消息进行序列化和反序列化，2.把自定义的消息类添加到受信任的消息名单中，3.添加一个全局环境变量SPRING\_AMQP\_DESERIALIZATION\_TRUST\_ALL = true，彻底信任所有的类。

我本来是有一点技术追求的，但是其他的两个办法都没有任何作用，花了一晚上，无可奈何只能污染了自己的全局变量。所以这个问题到现在仍没有完全解决。



会不断尝试并且打印报错，非常恐怖

