实验一：设计模式应用——简易中间件

一、实验要求

1、基础要求

按照实验指导书和源代码，完成简易消息中间件的代码调试和运行。做好代码分析工作，分别针对两个简易消息中间件的代码，绘制类图和时序图。做好代码测试分析工作，设计多个测试用例，给出测试结果，分析代码中存在的问题，并做好相应的优化调整。最终需要提交代码分析报告。

2、进阶要求

按照下述的业务场景描述，基于简易消息中间件的代码，完成简单应用的实现。最终需要提交应用开发报告和源代码。

业务场景：

假设快递公司需要实时追踪每个包裹的物流信息，并及时更新包裹的状态。在这种情况下，我们可以使用上述消息中间件来实现消息的传递和事件通知。

具体来说，我们可以将物流系统中的各个模块（例如订单管理、仓库管理、快递配送等）注册到消息中间件中，并为它们创建一个共享的消息队列。当一个包裹的物流状态发生变化时，例如包裹已出库、包裹已发货、包裹已签收等，相应的模块（生产者）可以向消息队列中发送一个消息，包含相应的物流信息和状态更新。消费者来监听消息队列中的消息变化，并在收到新消息时触发相应的处理逻辑。

3、实验评分标准

实验分数包含以下两部分：

- 完成基础要求，按照代码分析报告的质量进行评分，满分 80 分。
- 完成进阶要求，按照应用开发报告和源代码的质量进行评分，满分 20 分。

二、实验指导

简易的消息中间件实现 1

用到的设计模式：工厂模式、单例模式和观察者模式

```java
// 消息接口，表示一个消息
public interface Message {
    String getContent();
}

// 消息监听接口，标识一个消息观察者
public interface MessageListener {
    void onMessageChanged(MessageQueue queue);
}

// 消息队列接口，表示一个消息队列
public interface MessageQueue {
    void enqueue(Message message);
    Message dequeue();
    void addListener(MessageListener listener);
    void removeListener(MessageListener listener);
}


// 消息中心接口，表示一个消息中心
public interface MessageCenter {
    void registerQueue(MessageQueue queue);
    void unregisterQueue(MessageQueue queue);
    void broadcast(Message message);
}

// 简单的消息类，实现了 Message 接口
public class SimpleMessage implements Message {
    private final String content;

    public SimpleMessage(String content) {
        this.content = content;
    }

    @Override
    public String getContent() {
        return content;
    }
}

// 简单的消息队列类，实现了 MessageQueue 接口
public class SimpleMessageQueue implements MessageQueue {
    private List<Message> messages = new ArrayList<>();
    private List<MessageListener> listeners = new ArrayList<>();
```

```
    @Override
    public void enqueue(Message message) {
        messages.add(message);
        notifyListeners();
    }

    @Override
    public Message dequeue() {
        if (messages.isEmpty()) {
            return null;
        }
        Message message = messages.remove(0);
        return message;
    }

    @Override
    public void addListener(MessageListener listener) {
        listeners.add(listener);
    }

    @Override
    public void removeListener(MessageListener listener) {
        listeners.remove(listener);
    }

    private void notifyListeners() {
        for (MessageListener listener : listeners) {
            if (listener != null) {
                listener.onMessageChanged(this);
            }
        }
    }
}

// 简单的消息中心类，实现了 MessageCenter 接口
public class SimpleMessageCenter implements MessageCenter {
    private List<MessageQueue> queues = new ArrayList<>();
    private volatile static SimpleMessageCenter instance;
    private SimpleMessageCenter() {}
    //使用单例模式（双重检查锁定）创建唯一实例
    public static SimpleMessageCenter getInstance() {
        if (instance == null) {
            synchronized (SimpleMessageCenter.class) {
```

```
                if (instance == null) {
                    instance = new SimpleMessageCenter();
                }
            }
        }
        return instance;
    }

    @Override
    public void registerQueue(MessageQueue queue) {
        queues.add(queue);
    }

    @Override
    public void unregisterQueue(MessageQueue queue) {
        queues.remove(queue);
    }

    @Override
    public void broadcast(Message message) {
        for (MessageQueue queue : queues) {
            queue.enqueue(message);
        }
    }
}
```

接下来，我们可以使用工厂模式来创建这些组件。我们定义一个工厂类 `MessageFactory`，它负责创建消息、消息队列和消息中心的实例。

```
public class MessageFactory {
    // 私有化构造函数，禁止外部创建对象
    private MessageFactory() {}

    // 创建一个 Message 对象
    public static Message createMessage(String content) {
        return new SimpleMessage(content);
    }

    // 创建一个 MessageQueue 对象
    public static MessageQueue createMessageQueue() {
        return new SimpleMessageQueue();
    }
```

```
    // 创建一个 MessageCenter 对象
    public static MessageCenter createMessageCenter() {
        return SimpleMessageCenter.getInstance();
    }
}
```

最后使用上述组件来发送和接收消息。使用示例如下：

```
public class Example {
    public static void main(String[] args) {
        // 创建消息中心
        MessageCenter center = MessageFactory.createMessageCenter();

        // 创建两个消息队列并注册到消息中心
        MessageQueue queue1 = MessageFactory.createMessageQueue();
        MessageQueue queue2 = MessageFactory.createMessageQueue();
        center.registerQueue(queue1);
        center.registerQueue(queue2);

        // 创建两个消息监听器（观察者）并注册到消息队列
        MessageListener listener1 = new MessageListener() {
            @Override
            public void onMessageChanged(MessageQueue queue) {
                Message message = queue.dequeue();
                System.out.println("Listener 1: " + message.getContent());
            }
        };
        queue1.addListener(listener1);

        MessageListener listener2 = new MessageListener() {
            @Override
            public void onMessageChanged(MessageQueue queue) {
                Message message = queue.dequeue();
                System.out.println("Listener 2: " + message.getContent());
            }
        };
        queue2.addListener(listener2);

        // 发送一条消息到消息中心，所有注册的消息队列都会收到该消息
        Message message = MessageFactory.createMessage("Hello, world!");
        center.broadcast(message);
```

```
        // 注销消息队列和监听器
        center.unregisterQueue(queue1);
        center.unregisterQueue(queue2);
        queue1.removeListener(listener1);
        queue2.removeListener(listener2);
    }
}
```

执行结果：

```
Listener 1: Hello, world!
Listener 2: Hello, world!
```

简易的消息中间件实现 2

用到的设计模式：生产者消费者模式、工厂模式、单例模式和观察者模式

相较于实现 1，增加了生产者消费者模式的使用。

```
// 消息接口，表示一个消息
public interface Message {
    String getContent();
}

// 消息中心接口，表示一个消息中心
public interface MessageCenter {
    void registerQueue(MessageQueue queue);
    void unregisterQueue(MessageQueue queue);
    void broadcast(Message message) throws InterruptedException;
}

// 消息监听器接口，表示一个消息监听器
public interface MessageListener {
    void onMessageChanged(MessageQueue queue);
}

// 消息队列接口，表示一个消息队列
public interface MessageQueue {
    void enqueue(Message message) throws InterruptedException;
    Message dequeue() throws InterruptedException;
    void addListener(MessageListener listener);
```

```
    void removeListener(MessageListener listener);
}

// 简单的消息类，实现了 Message 接口
public class SimpleMessage implements Message {
    private final String content;

    public SimpleMessage(String content) {
        this.content = content;
    }

    @Override
    public String getContent() {
        return content;
    }
}

// 简单的消息中心类，实现了 MessageCenter 接口
public class SimpleMessageCenter implements MessageCenter {
    private List<MessageQueue> queues = new ArrayList<>();
    private volatile static SimpleMessageCenter instance;
    private SimpleMessageCenter() {}

    public static SimpleMessageCenter getInstance() {
        if (instance == null) {
            synchronized (SimpleMessageCenter.class) {
                if (instance == null) {
                    instance = new SimpleMessageCenter();
                }
            }
        }
        return instance;
    }

    @Override
    public void registerQueue(MessageQueue queue) {
        queues.add(queue);
    }

    @Override
    public void unregisterQueue(MessageQueue queue) {
        queues.remove(queue);
    }
```

```
    @Override
    public void broadcast(Message message) throws InterruptedException {
        for (MessageQueue queue : queues) {
            queue.enqueue(message);
        }
    }
}

// 简单的消息队列类，实现了 MessageQueue 接口
public class SimpleMessageQueue implements MessageQueue {
    private   int capacity; // 队列容量
    private List<Message> messages = new ArrayList<>();
    private List<MessageListener> listeners = new ArrayList<>();
    private   Object lock = new Object();


    public SimpleMessageQueue(int capacity) {
        this.capacity = capacity;
    }


    @Override
    public void enqueue(Message message) throws InterruptedException {
        synchronized (lock) {
            while (messages.size() == capacity) {
                lock.wait();
            }
            messages.add(message);
            notifyListeners();
            lock.notifyAll();
        }
    }

    @Override
    public Message dequeue() throws InterruptedException {
        synchronized (lock) {
            while (messages.isEmpty()) {
                lock.wait();
            }
            Message message = messages.remove(0);
            notifyListeners();
            lock.notifyAll();
            return message;
        }
```

```
    }

    @Override
    public void addListener(MessageListener listener) {
        listeners.add(listener);
    }

    @Override
    public void removeListener(MessageListener listener) {
        listeners.remove(listener);
    }

    private void notifyListeners() {
        for (MessageListener listener : listeners) {
            if (listener != null) {
                listener.onMessageChanged(this);
            }
        }
    }

}

// 消息工厂，用来创建各个组件
public class MessageFactory {
    // 私有化构造函数，禁止外部创建对象
    private MessageFactory() {}

    // 创建一个 Message 对象
    public static Message createMessage(String content) {
        return new SimpleMessage(content);
    }

    // 创建一个 MessageQueue 对象
    public static MessageQueue createMessageQueue(int capacity) {
        return new SimpleMessageQueue(capacity);
    }

    // 创建一个 MessageCenter 对象
    public static MessageCenter createMessageCenter() {
        return SimpleMessageCenter.getInstance();
    }
}
```

```
// 生产者类，用于向消息队列中添加消息
public class Producer implements Runnable {
    private final MessageQueue queue;
    private final String name;

    public Producer(MessageQueue queue, String name) {
        this.queue = queue;
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            Message message = MessageFactory.createMessage("Message " + i + "
from " + name);
            try {
                queue.enqueue(message);
                System.out.println("Producer " + name + " put " +
message.getContent());
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// 消费者类，用于从消息队列中取出消息并处理
public class Consumer implements Runnable {
    private final MessageQueue queue;
    private final String name;

    public Consumer(MessageQueue queue, String name) {
        this.queue = queue;
        this.name = name;
    }

    @Override
    public void run() {
        while (true) {
            try {
                Message message = queue.dequeue();
                System.out.println("Consumer " + name + " got " +
message.getContent());
```

```
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Example {
    public static void main(String[] args) {
        // 创建消息中心
        MessageCenter center = MessageFactory.createMessageCenter();

        // 创建一个消息队列并注册到消息中心
        MessageQueue queue = new SimpleMessageQueue(5);
        center.registerQueue(queue);

        // 创建两个生产者和两个消费者
        Producer producer1 = new Producer(queue, "Producer 1");
        Producer producer2 = new Producer(queue, "Producer 2");
        Consumer consumer1 = new Consumer(queue, "Consumer 1");
        Consumer consumer2 = new Consumer(queue, "Consumer 2");
       //启动两个生产者线程，用于向消息队列发送消息
        new Thread(producer1).start();
        new Thread(producer2).start();
      //启动两个消费者线程，用于从消息队列接收消息
        new Thread(consumer1).start();
        new Thread(consumer2).start();

        // 稍微等待一段时间后，注销消息队列
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        center.unregisterQueue(queue);
    }
}
```

执行结果：

```
Consumer Consumer 2 got Message 0 from Producer 1
Producer Producer 2 put Message 0 from Producer 2
```

```
Consumer Consumer 1 got Message 0 from Producer 2
Producer Producer 1 put Message 0 from Producer 1
Producer Producer 2 put Message 1 from Producer 2
Producer Producer 1 put Message 1 from Producer 1
Consumer Consumer 2 got Message 1 from Producer 2
Consumer Consumer 1 got Message 1 from Producer 1
Producer Producer 1 put Message 2 from Producer 1
Producer Producer 2 put Message 2 from Producer 2
Producer Producer 2 put Message 3 from Producer 2
Producer Producer 1 put Message 3 from Producer 1
Consumer Consumer 1 got Message 2 from Producer 2
Consumer Consumer 2 got Message 2 from Producer 1
Producer Producer 2 put Message 4 from Producer 2
Producer Producer 1 put Message 4 from Producer 1
Producer Producer 2 put Message 5 from Producer 2
Consumer Consumer 1 got Message 3 from Producer 2
Producer Producer 1 put Message 5 from Producer 1
Consumer Consumer 2 got Message 3 from Producer 1
Producer Producer 2 put Message 6 from Producer 2
Consumer Consumer 2 got Message 4 from Producer 2
Producer Producer 2 put Message 7 from Producer 2
Consumer Consumer 1 got Message 4 from Producer 1
Producer Producer 1 put Message 6 from Producer 1
Consumer Consumer 2 got Message 5 from Producer 2
Producer Producer 1 put Message 7 from Producer 1
Producer Producer 2 put Message 8 from Producer 2
Consumer Consumer 1 got Message 5 from Producer 1
Consumer Consumer 2 got Message 6 from Producer 2
Producer Producer 1 put Message 8 from Producer 1
Consumer Consumer 1 got Message 7 from Producer 2
Producer Producer 2 put Message 9 from Producer 2
Consumer Consumer 1 got Message 6 from Producer 1
Producer Producer 1 put Message 9 from Producer 1
Consumer Consumer 2 got Message 7 from Producer 1
Consumer Consumer 1 got Message 8 from Producer 2
Consumer Consumer 2 got Message 8 from Producer 1
Consumer Consumer 1 got Message 9 from Producer 2
Consumer Consumer 2 got Message 9 from Producer 1
```