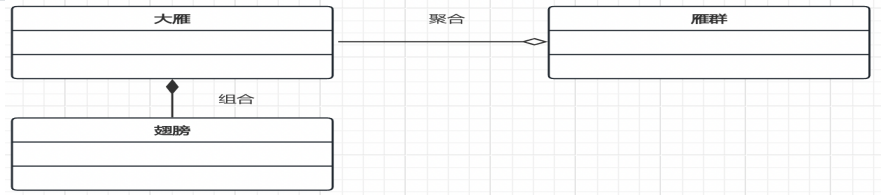
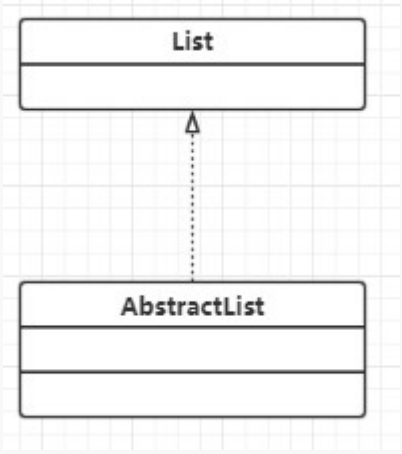
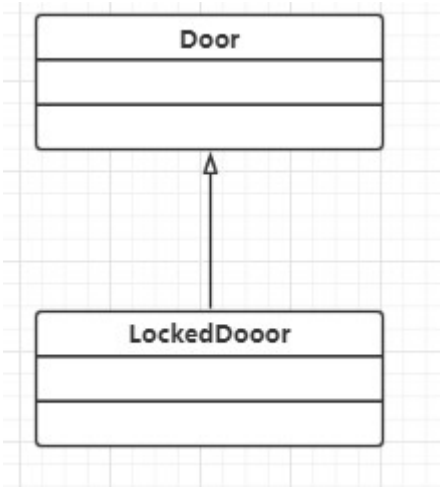
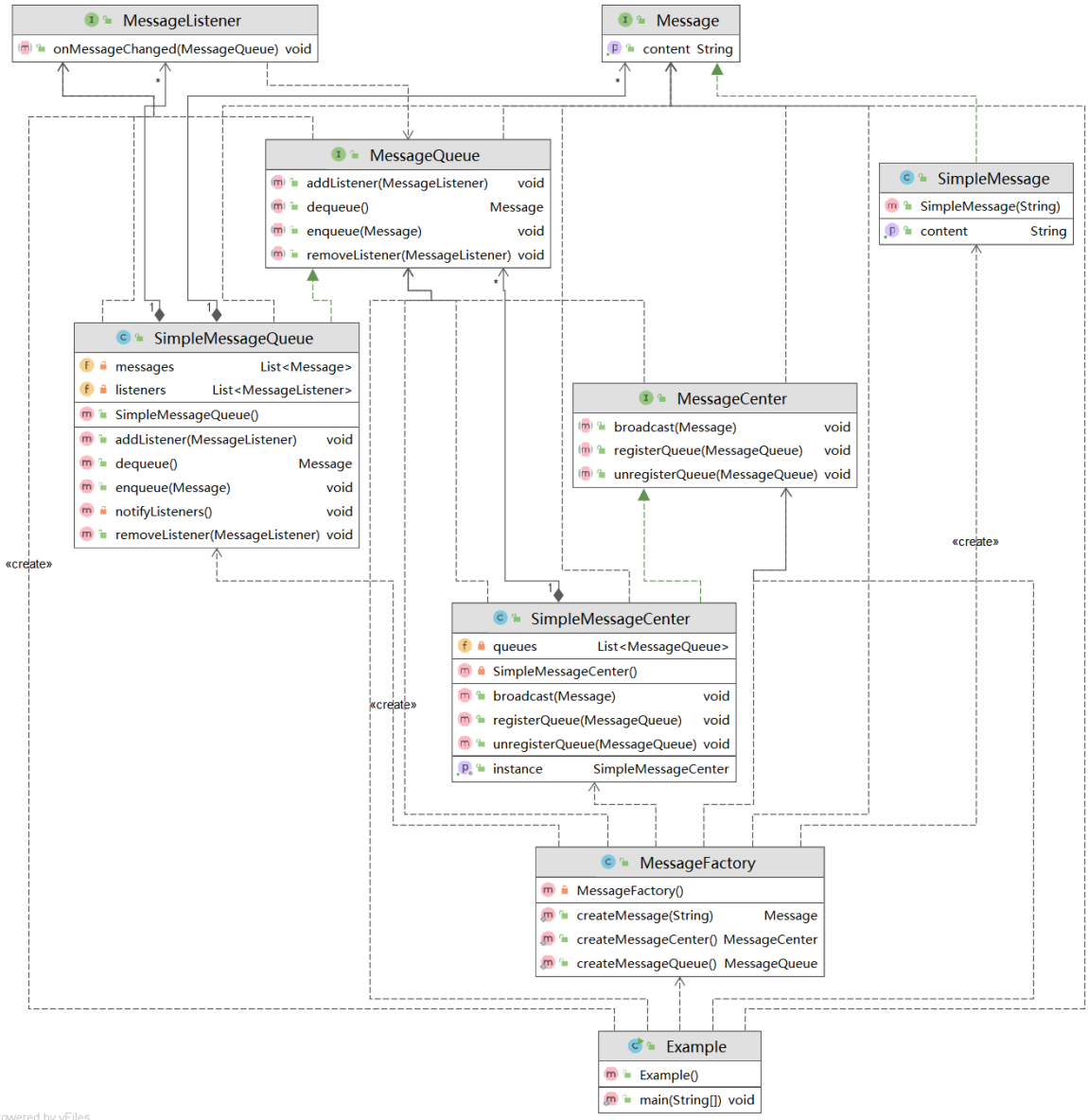
画类图时弄清各种符号的含义

依赖	
聚合	
实现	
继承	

双重校验Volatile关键字：保证可见性 ；防止创建对象过程的指令重排

	线程1	线程2
t1	分配内存	
t2	变量赋值	
t3		判断对象是否为 null
t4		由于对象不为 null，访问改对象
t5	初始化对象	

simple1类图



simple1改进

1. 出队后进行message的非空判断

```
public void onMessageChanged(MessageQueue queue) {
                Message message = queue.dequeue();
                if(message != null)
                    System.out.println("Listener 1: " + message.getContent());
            }
        };
```

2.添加和删除监听器时进行非空判断

```
    @Override
    public void addListener(MessageListener listener) {
        if(listener != null)
            listeners.add(listener);
    }

    @Override
    public void removeListener(MessageListener listener) {
        if(listener != null)
            listeners.remove(listener);
    }
```

simple2改进

3.SimpleMessageQueue的入队和出队操作加锁步骤过于繁琐，可以使用保证线程安全的队列LinkedBlockingQueue，使用两个ReentrantLock保证

方法名	做法
put()	如果队列满了，就阻塞，当队列不满的时候，会再执行入队操作
offer()	如果队列满了，返回false
take()	如果队列为空，就阻塞，当队列不空的时候，会再执行出队操作
poll()	如果队列空了，返回null
peek()	返回队列首元素，不会出队

```
import org.example.interfaces.Message;
import org.example.interfaces.MessageListener;
import org.example.interfaces.MessageQueue;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

/**
 * @description: 简单的消息队列类，实现了 MessageQueue 接口，使用 LinkedBlockingQueue
实现队列操作，提高可靠性和性能
 */
public class SimpleMessageQueue implements MessageQueue

    private int capacity; // 队列容量
    private BlockingQueue<Message> messages; // 使用 LinkedBlockingQueue 代替 List
    private List<MessageListener> listeners = new CopyOnWriteArrayList<>();

    public SimpleMessageQueue(int capacity) {
        this.capacity = capacity;
        this.messages = new LinkedBlockingQueue<>(capacity);
    }

    @Override
```

```
    public void enqueue(Message message) throws InterruptedException {
        messages.put(message); // 使用阻塞操作插入元素
    }

    @Override
    public Message dequeue() throws InterruptedException {
        Message message = messages.take(); // 使用阻塞操作移除元素
        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);
            }
        }
    }
}
```

4.监听器ArrayList存在线程安全问题，若多线程同时同时添加listener，可能会导致数据丢失问题的出现，结果只有一个监听器添加上，所以使用线程安全的CopyOnWriteArrayList代替ArrayList，CopyOnWriteArrayList读取时不加锁，写时加锁，适合读多写少的场景，MessageListener仅创建时添加一次，之后都是遍历读取，符合当前的应用场景。

5.添加出队和入队的异常处理

```
@Override
public Message dequeue() throws InterruptedException {
    synchronized (lock) {
        try {
            while (messages.isEmpty()) {
                lock.wait();
            }
            Message message = messages.remove(0);
            notifyListeners();
            lock.notifyAll();
            return message;
        } catch (InterruptedException e) {
            // 记录日志
            logger.error("Dequeue operation was interrupted: ", e);
            // 释放资源
            cleanup();
            // 重新抛出中断异常
            Thread.currentThread().interrupt();
            throw e;
```

```
        }
    }
}

// 释放资源的方法
private void cleanup() {
    // 在这里进行资源释放操作，例如关闭文件、数据库连接等
    // 注意要在 finally 块中调用 cleanup() 方法，以保证资源能够得到释放
}
```

6.手动创建线程池复杂而繁琐，可以使用 `java.util.concurrent.Executors` 类来创建线程池

```
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 = MessageFactory.createMessageQueue(10);
    center.registerQueue(queue);

    // 创建线程池
    ExecutorService executor = Executors.newFixedThreadPool(4);

    // 提交生产者和消费者任务到线程池
    executor.submit(new Producer(queue, "Producer 1"));
    executor.submit(new Producer(queue, "Producer 2"));
    executor.submit(new Consumer(queue, "Consumer 1"));
    executor.submit(new Consumer(queue, "Consumer 2"));

    // 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);

    // 关闭线程池
    executor.shutdown();
}
```

在观察者模式中，观察者对象通常不会直接操作被观察对象，而是通过被观察对象提供的接口来获取信息或改变状态。观察者模式的核心思想是，当被观察对象的状态发生改变时，所有观察者都会接收到通知并进行相应的处理。

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

在上述代码中，`onMessageChanged()` 方法的参数是被观察对象 `this`，那么观察者对象就可以通过该参数访问被观察对象的状态或方法，并进行相应的操作，这样就会破坏观察者模式中观察者和被观察者之间的松耦合设计。

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。

物流基础类

```
public class LogisticsInfo {
    private String packageId; // 包裹编号
    private String status; // 包裹状态

    public LogisticsInfo(String packageId, String status) {
        this.packageId = packageId;
        this.status = status;
    }

    public String getPackageId() {
        return packageId;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

```
// 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);

}
```

```
public interface MessageListener {
    void onMessageChanged(LogisticsInfo logisticsInfo) throws
InterruptedException;
}
```

```
public interface MessageQueue {
    void enqueue(LogisticsInfo logisticsInfo) throws InterruptedException;
    LogisticsInfo dequeue() throws InterruptedException;
    void addListener(MessageListener listener);
    void removeListener(MessageListener listener);
}
```

```
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);
    }


}
```

```
public class SimpleMessageQueue implements MessageQueue {
    private  int capacity; // 队列容量
    private BlockingQueue<LogisticsInfo> messages;
    private List<MessageListener> listeners = new ArrayList<>();
    private  Object lock = new Object();


    public SimpleMessageQueue(int capacity) {
        this.capacity = capacity;
        this.messages = new LinkedBlockingQueue<>(capacity);
    }



    @Override
    public void enqueue(LogisticsInfo logisticsInfo) throws InterruptedException
{
```

```
        messages.put(logisticsInfo); // 使用阻塞操作插入元素

    }

    @Override
    public LogisticsInfo dequeue() throws InterruptedException {
        LogisticsInfo logisticsInfo = messages.take(); // 使用阻塞操作移除元素
        return logisticsInfo;
    }

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

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


}
```

```
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 LogisticsInfo createMessage(String id, String status) {
        return new LogisticsInfo(id, status);
    }

    // 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();
    }
}
```

```
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() {
```

```
        System.out.println("thread id" + Thread.currentThread().getId());
        int temp = new Random().nextInt(100);
        for (int i = temp; i < temp + 3; i++) {
            LogisticsInfo message =
 MessageFactory.createMessage(Integer.toString(i),"未出库");;
            try {
                PackageState(message);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void PackageState(LogisticsInfo message) throws InterruptedException
{
        String[] statuses = {"未出库","已出库","已发货","已签收"};
        Random random = new Random();
        for (String status: statuses) {
            message.setStatus(status);
            queue.enqueue(message);
            System.out.println("生产者 " + name + " 将包裹信息放入消息队列中: "
+"Package "+ message.getPackageId() + " Status "+message.getStatus());

        }
    }
}
```

```
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 {
                LogisticsInfo message = queue.dequeue();
                System.out.println("消费者获" + name + "取到信息: Package ID: " +
message.getPackageId() + " Status: " + message.getStatus());
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class Main {
    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 = MessageFactory.createMessageQueue(10);
        center.registerQueue(queue);
        //创建一个固定大小的线程池
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // 提交生产者和消费者任务到线程池
        executor.submit(new Producer(queue, "Producer 1"));
        executor.submit(new Producer(queue, "Producer 2"));
        executor.submit(new Producer(queue, "Producer 3"));
        executor.submit(new Consumer(queue, "Consumer 1"));


        // 等待一段时间后，注销消息队列
        try {
            Thread.sleep(8000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        center.unregisterQueue(queue);
        // 关闭线程池

    }
}
```