12 - 图片协同编辑 - 智能协同云图库项目教程 - 编程导航教程

> 本节重点上一节我们已经完成了团队空间的创建、成员管理和权限控制等功能。

本节重点

上一节我们已经完成了团队空间的创建、成员管理和权限控制等功能。为了提高项目的商业价值，本节来完成本项目的亮点功能 —— 图片协同编辑。

大纲：

- 图片协同编辑需求分析
- 图片协同编辑方案设计
- 图片协同编辑后端开发
- 图片协同编辑前端开发

通过本节，你将学习到多人实时协作功能的设计开发，涉及 WebSocket、事件驱动设计、Disruptor 无锁队列等技术知识。学会后再去开发聊天室之类的业务，都会轻松很多。

一、需求分析

现在很多产品都有多人协作功能，比如协同文档、协同素材设计、协同代码编辑器等等，可以提高协作的效率。

对于我们的项目，所谓的图片协同编辑功能，是在图片编辑的基础上增加了 “协同” 的概念。当用户编辑某张图片时，其他用户可以 实时 看到编辑效果和操作提示。如图：

注意，因为只有团队空间才会有多个用户编辑同一张图片，所以该功能只对团队空间开放，需要成员具有编辑权限。协同的图片编辑操作包括左旋、右旋、放大、缩小。

二、方案设计

虽然需求介绍很简单，但是涉及到多人协作的业务，有很多问题需要考虑，比如：

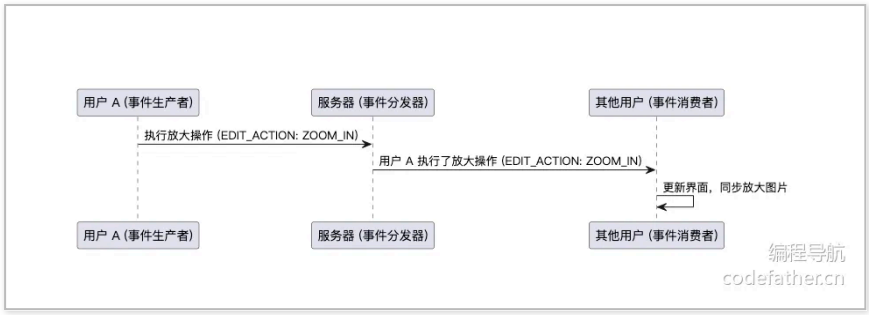
- 多个用户之间如何进行交互？
- 如何防止协作编辑时出现冲突？
- 如何提高协作的实时性？

协作交互流程

多人协作时，每个用户的动作都需要通知到其他用户，收到通知消息的用户需要进行相应的处理。

比如用户 A 放大了图片，就需要给其他正在编辑的用户发送“图片放大”消息，其他用户收到这个消息后，需要同步放大自己界面上的图片。

这其实是一种 事件驱动 的架构设计思想，协作编辑中的每个用户动作本质上是一个 事件，执行动作时会产生事件并提交给服务器；服务器收到事件后，会转发给其他用户；其他用户收到事件后，就要作为事件的消费者来处理事件。流程如图：



相比于生产者直接调用消费者，事件驱动模型的主要优点在于 解耦 和 异步性。在事件驱动模型中，生产者和消费者不需要直接依赖于彼此的实现，生产者只需触发事件并将其发送到事件分发器，消费者则根据事件类型处理逻辑。这样多个消费者可以独立响应同一事件（比如一个用户旋转了图片，其他用户都能同步），系统更加灵活，可扩展性更强。此外，事件驱动还可以提升系统的 并发性 和 实时性，可以理解为多引入了一个中介来帮忙，通过异步消息传递，减少了阻塞和等待，能够更高效地处理多个并发任务。

下面我们按照事件驱动的设计，来详细列举协作编辑的交互流程：

事件触发者（用户 A 的动作）	**事件类型（发送消息）**	**事件消费者（其他用户的处理）**
用户 A 建立连接，加入编辑	INFO	显示 “用户 A 加入编辑” 的通知
用户 A 执行编辑操作	EDIT_ACTION	放大 / 缩小 / 左旋 / 右旋当前图片
用户 A 断开连接，离开编辑	INFO	显示 “用户 A 离开编辑” 的通知

解决协作冲突

1、解决方案

假设这样一种场景：鱼皮和李蛋同时快速点击了十次旋转，最终的结果会是怎样的呢？

如果所有事件都是按顺序处理的，那结果就很清晰了，但事实上，为了提高性能和响应速度，事件通常是 **并发** 的，而不是严格的顺序执行。这种并发操作会引发 **协作冲突**，导致其他用户看到的旋转效果是乱序的。

那么你会怎么解决协作冲突的问题呢？

我们可以通过业务设计来减少开发成本，比如约定 **同一时刻只允许一位用户进入编辑图片的状态**，此时其他用户只能实时浏览到修改效果，但不能参与编辑；进入编辑状态的用户可以退出编辑，其他用户才可以进入编辑状态。类似于给图片编辑这个动作加了一把锁，直接从源头上解决了编辑冲突的问题。

此时，协作编辑的交互流程又要增加 2 个动作 —— 进入编辑状态和退出编辑状态：

事件触发者（用户 A 的动作）	**事件类型（发送消息）**	**事件消费者（其他用户的处理）**
用户 A 建立连接，加入编辑	INFO	显示 “用户 A 加入编辑” 的通知
用户 A 进入编辑状态	ENTER_EDIT	其他用户界面显示 “用户 A 开始编辑图片”，锁定编辑状态
用户 A 执行编辑操作	EDIT_ACTION	放大 / 缩小 / 左旋 / 右旋当前图片
用户 A 退出编辑状态	EXIT_EDIT	解锁编辑状态，提示其他用户可以进入编辑状态

事件触发者（用户 A 的动作）	事件类型（发送消息）	事件消费者（其他用户的处理）
用户 A 断开连接，离开编辑	INFO	显示 “用户 A 离开编辑” 的通知，并释放编辑状态
用户 A 发送了错误的消息	ERROR	显示错误消息的通知

其实核心流程是前 5 行，但是考虑到前端传递了错误参数的情况，我们新增一种 `ERROR` 事件类型，可用于展示错误提示信息。

在本项目中，我们就采用这种方案，不仅实现简单、流程清晰，也尽最大可能减少了编辑冲突的风险。

但这种方案的缺点也很明显，减少了实时协作的便利性，对于协作设计、协作编码、协作文档的场景，同一时间只能有一个用户编辑，提高的效率有限。所以这里再分享另外一种实时协同算法作为扩展知识。

2、扩展知识 - OT 算法

实时协同 OT 算法（Operational Transformation）是一种支持分布式系统中多个用户实时协作编辑的核心算法，广泛应用于在线文档协作等场景。OT 算法的主要功能是解决并发编辑冲突，确保编辑结果在所有用户终端一致。

OT 算法其实很好理解，先看下 3 个核心概念：

- 操作 (Operation)：表示用户对协作内容的修改，比如插入字符、删除字符等。
- 转化 (Transformation)：当多个用户同时编辑内容时，OT 会根据操作的上下文将它们转化，使得这些操作可以按照不同的顺序应用而结果保持一致。

- 因果一致性：OT 算法确保操作按照用户看到的顺序被正确执行，即每个用户的操作基于最新的内容状态。

其中，最重要的就是 **转化** 步骤了，相当于有一个负责人统一收集大家的操作，然后按照设定的规则和信息进行排序与合并，最终给大家一个统一的结果。

举一个简单的例子，假设初始内容是 `"abc"` ，用户 A 和 B 同时进行编辑：

- 用户 A 在位置 `1` 插入 `"x"`
- 用户 B 在位置 `2` 删除 `"b"`

如果不使用 OT 算法，结果是：

1. 用户 A 操作后，内容变为 `"axbc"`
2. 用户 B 操作后，内容变为 `"ac"`

如果直接应用 B 的操作到 A 的结果，得到的是 `"ac"` ，对于 A 来说，相当于删除了 `"b"` ，A 会感到一脸懵逼。

如果使用 OT 算法，结果是：

1. 用户 A 的操作，应用后内容为 `"axbc"`
2. 用户 B 的操作经过 OT 转化为删除 `"b"` 在 `"axbc"` 中的新位置

最终用户 A 和 B 的内容都一致为 `"axc"` ，符合预期。OT 算法确保无论用户编辑的顺序如何，最终内容是一致的。

当然，具体的 OT 算法还是要根据需求来设计了，协作密度越高，算法设计难度越大。

此外，还有一种与 OT 类似的协同算法 CRDT（Conflict-free Replicated Data Type），其通过数学模型实现无需中心化转化的冲突解决，在离线协作场景中更具优势，感兴趣的同学可以自行了解。

提高协作实时性

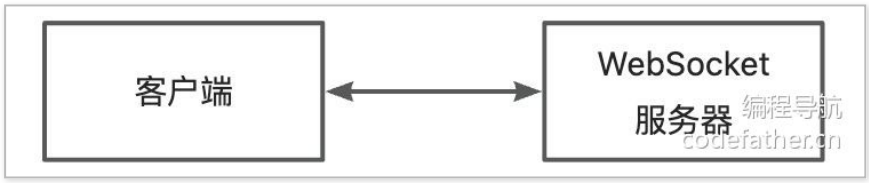
在实时通讯的业务场景中，常用的技术方案包括长轮询、SSE 和 WebSocket。由于我们的业务需求需要实现频繁且高效的双向通信，因此我们选用 WebSocket 来实现即时通讯。

1、什么是 WebSocket?

WebSocket 是一种 **全双工通信协议**，让客户端（比如浏览器）和服务器之间能够保持实时、持续的连接。和传统的 HTTP 请求 - 响应模式不同，WebSocket 是一条 **"常开的隧道"**，连接的双方可以随时发送和接收数据，而不需要不断建立和关闭连接。

打个比方：

- HTTP 就像点外卖： 每次下单（请求）- 到货（响应）都是一次独立的操作，完成后连接关闭。
- WebSocket 像是打电话：你打通了电话（建立连接），可以随时聊天（双向通信），直到挂断（关闭连接）。



2、WebSocket 的应用场景

WebSocket 的主要作用是 **实现实时数据传输**，适用于需要频繁交互或者实时更新数据的场景。比如：

- 即时通讯（聊天软件、实时协作工具）
- 实时数据更新（股票行情、体育比赛比分）
- 在线游戏（多人实时互动）
- 物联网（设备状态实时传输）
- 协同编辑（像语雀这样的多人协作编辑）

通过 WebSocket，客户端与服务器之间能够显著减少消息传输的延迟，提高通信效率，同时降低数据传输的开销。

3、WebSocket 和 HTTP 的关系

WebSocket 和 HTTP 是两种不同的通信协议，但它们是紧密相关的，都是基于 TCP 协议、都可以在同样的端口上工作（比如 80 和 443）。

** 首先要明确，WebSocket 是建立在 HTTP 基础之上的！**WebSocket 的连接需要通过 HTTP 协议发起一个握手（称为 HTTP Upgrade 请求），这个握手请求是 WebSocket 建立连接的前提，表明希望切换协议；服务器如果支持 WebSocket，会返回一个 HTTP 101 状态码，表示协议切换成功。

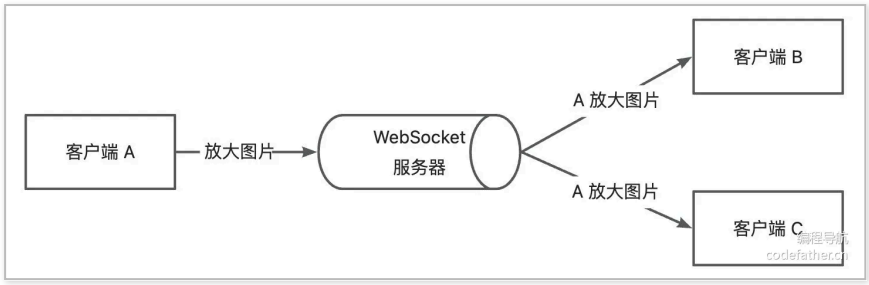
握手完成后，HTTP 协议的作用结束，通信会切换为 WebSocket 协议，双方可以开始全双工通信。

二者的区别如下，大家了解一下就好：

对比项	HTTP	WebSocket
通信模式	请求 - 响应（单向）	全双工通信（双向）
连接状态	每次请求创建新的连接	握手后保持持续连接
数据传输效率	每次通信都需要带完整头部，开销大	数据帧小，传输高效
适用场景	静态网页加载、API 调用等非实时场景	实时交互场景，如聊天、游戏、直播等

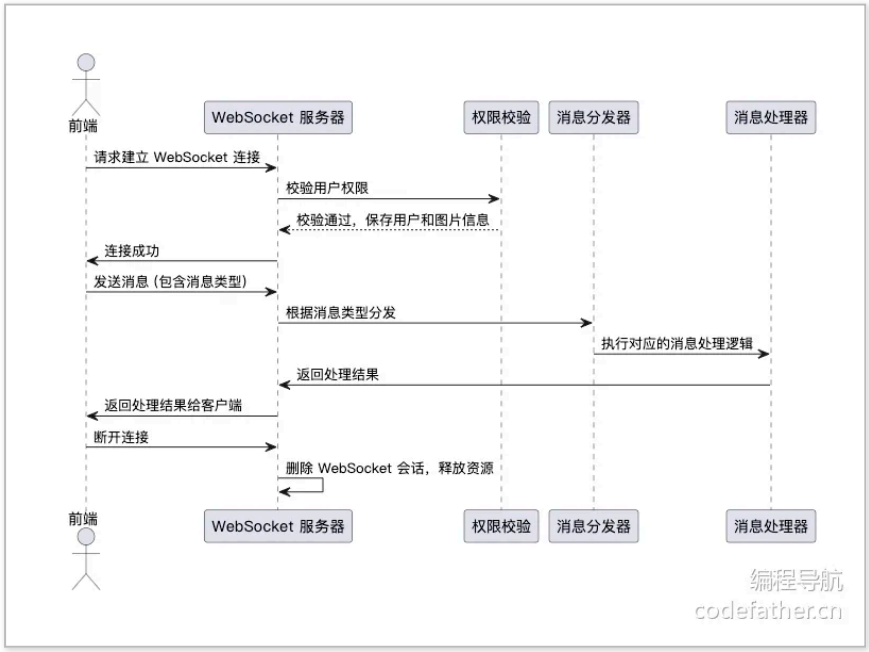
4、WebSocket 协作编辑的流程

通过 WebSocket 实时通信的能力，可以将用户的编辑操作发给 WebSocket 服务器，再由服务器转发给其他连接服务器的用户前端，前端就可以根据操作处理图片。



具体的业务流程：

1. 建立连接之前，先进行用户权限校验；校验通过后，将登录用户信息、要编辑的图片信息保存到要建立的 WebSocket 连接的会话属性中。
2. 建立连接成功后，将 WebSocket 会话保存到该图片对应的会话集合中，便于后续分发消息给其他会话。
3. 前端将消息发送到后端，后端根据消息类型分发到对应的处理器。
4. 处理器处理消息，将处理结果作为消息发送给需要的 WebSocket 客户端。
5. 当前端断开连接时，删除会话集合中的 WebSocket 会话，释放资源。



和 HTTP 请求一样，前端和 WebSocket 服务器之间传输信息时，也可以通过 JSON 格式对数据进行序列化。

5、WebSocket 的实现方式

对于 Java Spring 项目，主要有原生 WebSocket（基于 `WebSocketHandler` 实现）、STOMP、WebFlux 这 3 种实现方式。

它们之间的对比如下：

实现方式	特点	优点	缺点	适用场景
原生 WebSocket	低层 API，手动管理连接与消息	轻量、灵活、适用于简单点对点通信	需要手动管理会话和分发，不支持 STOMP	简单的实时推送，低并发场景
WebSocket + STOMP + SockJS	基于 STOMP，支持发布 / 订阅模式	支持 STOMP、消息代理、适配 SockJS	依赖外部代理，配置较复杂	聊天室、多人协作，高级实时应用
WebFlux + Reactive WebSocket	基于 WebFlux 的响应式实现	高并发、非阻塞、适用于大流量场景	学习曲线高，不支持 STOMP	高并发场景、大数据流推送

鱼皮的选择建议是：对于大多数简单实时推送，选用原生 WebSocket；对于复杂的聊天室和协同系统，选用 WebSocket + STOMP + SockJS；对于高并发、低延迟数据流推送，选用 WebFlux + Reactive WebSocket。

对于我们的项目，并发要求不高，选择 Spring 原生的 WebSocket 来降低开发成本。

明确方案后，我们进入后端开发。

三、后端开发

1、引入 WebSocket 依赖

引入依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

新建 `manager.websocket` 包，所有和 WebSocket 相关的代码都放到该包下。

2、定义数据模型

新建 `websocket.model` 包，存放数据模型，包括请求类、响应类、枚举类。

1）定义图片编辑请求消息，也就是前端要发送给后端的参数：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PictureEditRequestMessage {

    private String type;

    private String editAction;
}
```

2）定义图片编辑响应消息，也就是后端要发送给前端的信息：

```
@Data
@NoArgsConstructor
```

```
@AllArgsConstructor
public class PictureEditResponseMessage {


    private String type;


    private String message;


    private String editAction;


    private UserVO user;
}
```

3）定义图片编辑消息类型枚举，便于后续根据消息类型进行相应的处理：

```
@Getter
public enum PictureEditMessageTypeEnum {

    INFO("发送通知", "INFO"),
    ERROR("发送错误", "ERROR"),
    ENTER_EDIT("进入编辑状态", "ENTER_EDIT"),
    EXIT_EDIT("退出编辑状态", "EXIT_EDIT"),
    EDIT_ACTION("执行编辑操作", "EDIT_ACTION");

    private final String text;
    private final String value;

    PictureEditMessageTypeEnum(String text, String value) {
        this.text = text;
        this.value = value;
    }


    public static PictureEditMessageTypeEnum getEnumByValue(String va
        if (value == null || value.isEmpty()) {
            return null;
        }
        for (PictureEditMessageTypeEnum typeEnum : PictureEditMessage
            if (typeEnum.value.equals(value)) {
                return typeEnum;
            }
        }
        return null;
    }
}
```

4）定义图片编辑操作类型枚举：

```
@Getter
public enum PictureEditActionEnum {
```

```
    ZOOM_IN("放大操作", "ZOOM_IN"),
    ZOOM_OUT("缩小操作", "ZOOM_OUT"),
    ROTATE_LEFT("左旋操作", "ROTATE_LEFT"),
    ROTATE_RIGHT("右旋操作", "ROTATE_RIGHT");

    private final String text;
    private final String value;

    PictureEditActionEnum(String text, String value) {
        this.text = text;
        this.value = value;
    }


    public static PictureEditActionEnum getEnumByValue(String value)
        if (value == null || value.isEmpty()) {
            return null;
        }
        for (PictureEditActionEnum actionEnum : PictureEditActionEnum
            if (actionEnum.value.equals(value)) {
                return actionEnum;
            }
        }
        return null;
    }
}
```

3、WebSocket 拦截器 - 权限校验

在 WebSocket 连接前需要进行权限校验，如果发现用户没有团队空间内编辑图片的权限，则拒绝握手，可以通过定义一个 WebSocket 拦截器实现这个能力。

此外，由于 HTTP 和 WebSocket 的区别，我们不能在后续收到前端消息时直接从 request 对象中获取到登录用户信息，因此也需要通过 WebSocket 拦截器，为即将建立连接的 WebSocket 会话指定一些属性，比如登录用户信息、编辑的图片 id 等。

编写拦截器的代码，需要实现 `HandshakeInterceptor` 接口：

```
@Component
@Slf4j
public class WsHandshakeInterceptor implements HandshakeInterceptor {

    @Resource
    private UserService userService;

    @Resource
    private PictureService pictureService;
```

```
    @Resource
    private SpaceService spaceService;

    @Resource
    private SpaceUserAuthManager spaceUserAuthManager;

    @Override
    public boolean beforeHandshake(@NotNull ServerHttpRequest request
        if (request instanceof ServletServerHttpRequest) {
            HttpServletRequest servletRequest = ((ServletServerHttpRe

            String pictureId = servletRequest.getParameter("pictureId
            if (StrUtil.isBlank(pictureId)) {
                log.error("缺少图片参数，拒绝握手");
                return false;
            }
            User loginUser = userService.getLoginUser(servletRequest)
            if (ObjUtil.isEmpty(loginUser)) {
                log.error("用户未登录，拒绝握手");
                return false;
            }

            Picture picture = pictureService.getById(pictureId);
            if (picture == null) {
                log.error("图片不存在，拒绝握手");
                return false;
            }
            Long spaceId = picture.getSpaceId();
            Space space = null;
            if (spaceId != null) {
                space = spaceService.getById(spaceId);
                if (space == null) {
                    log.error("空间不存在，拒绝握手");
                    return false;
                }
                if (space.getSpaceType() != SpaceTypeEnum.TEAM.getVal
                    log.info("不是团队空间，拒绝握手");
                    return false;
                }
            }
            List<String> permissionList = spaceUserAuthManager.getPer
            if (!permissionList.contains(SpaceUserPermissionConstant.
                log.error("没有图片编辑权限，拒绝握手");
                return false;
            }

            attributes.put("user", loginUser);
            attributes.put("userId", loginUser.getId());
            attributes.put("pictureId", Long.valueOf(pictureId));
        }
        return true;
    }

    @Override
    public void afterHandshake(@NotNull ServerHttpRequest request, @N
    }
}
```

4、WebSocket 处理器

我们需要定义 WebSocket 处理器类，在连接成功、连接关闭、接收到客户端消息时进行相应的处理。

可以实现 TextWebSocketHandler 接口，这样就能以字符串的方式发送和接受消息了：

```
@Component
public class PictureEditHandler extends TextWebSocketHandler {
}
```

1）首先在处理器类中定义 2 个常量，分别为：

- 保存当前正在编辑的用户 id，执行编辑操作、进入或退出编辑时都会校验。
- 保存参与编辑图片的用户 WebSocket 会话的集合。

由于每个图片的协作编辑都是相互独立的，所以需要用 Map 来区分每个图片 id 对应的数据。代码如下：

```
private final Map<Long, Long> pictureEditingUsers = new ConcurrentHas

private final Map<Long, Set<WebSocketSession>> pictureSessions = new
```

注意，由于可能同时有多个 WebSocket 客户端建立连接和发送消息，集合要使用并发包（JUC）中的 `ConcurrentHashMap` ，来保证线程安全。

2）由于接下来很多消息都需要传递给所有协作者，所以先编写一个 **广播消息** 的方法。该方法会根据 pictureId，将响应消息发送给编辑该图片的所有会话。考虑到可能会有消息不需要发送给编辑者本人的情况，该方法还可以接受 excludeSession 参数，支持排除掉向某个会话发送消息。

代码如下：

```
private void broadcastToPicture(Long pictureId, PictureEditResponseMe
    Set<WebSocketSession> sessionSet = pictureSessions.get(pictureId)
    if (CollUtil.isNotEmpty(sessionSet)) {

        ObjectMapper objectMapper = new ObjectMapper();

        SimpleModule module = new SimpleModule();
        module.addSerializer(Long.class, ToStringSerializer.instance)
        module.addSerializer(Long.TYPE, ToStringSerializer.instance);
        objectMapper.registerModule(module);

        String message = objectMapper.writeValueAsString(pictureEditR
        TextMessage textMessage = new TextMessage(message);
        for (WebSocketSession session : sessionSet) {

            if (excludeSession != null && excludeSession.equals(sessi
                continue;
            }
            if (session.isOpen()) {
                session.sendMessage(textMessage);
            }
        }
    }
}
```

上述代码中有个小细节，由于前端 JS 的长整数可能会丢失精度，所以使用 Jackson 自定义序列化器，在将对象转换为 JSON 字符串时，将 Long 类型转换为 String 类型。

再编写一个不排除 Session，给所有会话广播的方法：

```
private void broadcastToPicture(Long pictureId, PictureEditResponseMe
    broadcastToPicture(pictureId, pictureEditResponseMessage, null);
}
```

3）实现连接建立成功后执行的方法，保存会话到集合中，并且给其他会话发送消息：

```
@Override
public void afterConnectionEstablished(WebSocketSession session) thro

    User user = (User) session.getAttributes().get("user");
    Long pictureId = (Long) session.getAttributes().get("pictureId");
    pictureSessions.putIfAbsent(pictureId, ConcurrentHashMap.newKeySe
    pictureSessions.get(pictureId).add(session);


    PictureEditResponseMessage pictureEditResponseMessage = new Pictu
    pictureEditResponseMessage.setType(PictureEditMessageTypeEnum.INF
    String message = String.format("%s加入编辑", user.getUserName());
```

```
        pictureEditResponseMessage.setMessage(message);
        pictureEditResponseMessage.setUser(userService.getUserVO(user));

        broadcastToPicture(pictureId, pictureEditResponseMessage);
    }
```

4）编写接收客户端消息的方法，根据消息类别执行不同的处理：

```
    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessag

        PictureEditRequestMessage pictureEditRequestMessage = JSONUtil.to
        String type = pictureEditRequestMessage.getType();
        PictureEditMessageTypeEnum pictureEditMessageTypeEnum = PictureEd


        Map<String, Object> attributes = session.getAttributes();
        User user = (User) attributes.get("user");
        Long pictureId = (Long) attributes.get("pictureId");


        switch (pictureEditMessageTypeEnum) {
            case ENTER_EDIT:
                handleEnterEditMessage(pictureEditRequestMessage, session
                break;
            case EDIT_ACTION:
                handleEditActionMessage(pictureEditRequestMessage, sessio
                break;
            case EXIT_EDIT:
                handleExitEditMessage(pictureEditRequestMessage, session,
                break;
            default:
                PictureEditResponseMessage pictureEditResponseMessage = n
                pictureEditResponseMessage.setType(PictureEditMessageType
                pictureEditResponseMessage.setMessage("消息类型错误");
                pictureEditResponseMessage.setUser(userService.getUserVO(
                session.sendMessage(new TextMessage(JSONUtil.toJsonStr(pi
        }
    }
```

接下来依次编写每个处理消息的方法。首先是用户进入编辑状态，要设置当前用户为编辑用户，并且向其他客户端发送消息：

```
    public void handleEnterEditMessage(PictureEditRequestMessage pictureE

        if (!pictureEditingUsers.containsKey(pictureId)) {

            pictureEditingUsers.put(pictureId, user.getId());
            PictureEditResponseMessage pictureEditResponseMessage = new P
```

```
            pictureEditResponseMessage.setType(PictureEditMessageTypeEnum
            String message = String.format("%s开始编辑图片", user.getUserN
            pictureEditResponseMessage.setMessage(message);
            pictureEditResponseMessage.setUser(userService.getUserVO(user
            broadcastToPicture(pictureId, pictureEditResponseMessage);
        }
    }
```

用户执行编辑操作时，将该操作同步给 除了当前用户之外 的其他客户端，也就是说编辑操作不用再同步给自己：

```
public void handleEditActionMessage(PictureEditRequestMessage picture
    Long editingUserId = pictureEditingUsers.get(pictureId);
    String editAction = pictureEditRequestMessage.getEditAction();
    PictureEditActionEnum actionEnum = PictureEditActionEnum.getEnumB
    if (actionEnum == null) {
        return;
    }

    if (editingUserId != null && editingUserId.equals(user.getId()))
        PictureEditResponseMessage pictureEditResponseMessage = new P
        pictureEditResponseMessage.setType(PictureEditMessageTypeEnum
        String message = String.format("%s执行%s", user.getUserName()
        pictureEditResponseMessage.setMessage(message);
        pictureEditResponseMessage.setEditAction(editAction);
        pictureEditResponseMessage.setUser(userService.getUserVO(user

        broadcastToPicture(pictureId, pictureEditResponseMessage, ses
    }
}
```

用户退出编辑操作时，移除当前用户的编辑状态，并且向其他客户端发送消息：

```
public void handleExitEditMessage(PictureEditRequestMessage pictureEd
    Long editingUserId = pictureEditingUsers.get(pictureId);
    if (editingUserId != null && editingUserId.equals(user.getId()))

        pictureEditingUsers.remove(pictureId);

        PictureEditResponseMessage pictureEditResponseMessage = new P
        pictureEditResponseMessage.setType(PictureEditMessageTypeEnum
        String message = String.format("%s退出编辑图片", user.getUserN
        pictureEditResponseMessage.setMessage(message);
        pictureEditResponseMessage.setUser(userService.getUserVO(user
        broadcastToPicture(pictureId, pictureEditResponseMessage);
    }
}
```

5）WebSocket 连接关闭时，需要移除当前用户的编辑状态、并且从集合中删除当前会话，还可以给其他客户端发送消息通知：

```
@Override
public void afterConnectionClosed(WebSocketSession session, @NotNull
    Map<String, Object> attributes = session.getAttributes();
    Long pictureId = (Long) attributes.get("pictureId");
    User user = (User) attributes.get("user");

    handleExitEditMessage(null, session, user, pictureId);


    Set<WebSocketSession> sessionSet = pictureSessions.get(pictureId)
    if (sessionSet != null) {
        sessionSet.remove(session);
        if (sessionSet.isEmpty()) {
            pictureSessions.remove(pictureId);
        }
    }


    PictureEditResponseMessage pictureEditResponseMessage = new Pictu
    pictureEditResponseMessage.setType(PictureEditMessageTypeEnum.INF
    String message = String.format("%s离开编辑", user.getUserName());
    pictureEditResponseMessage.setMessage(message);
    pictureEditResponseMessage.setUser(userService.getUserVO(user));
    broadcastToPicture(pictureId, pictureEditResponseMessage);
}
```

💡 由于处理器的代码并不复杂，而且处理逻辑中使用到了当前类的全局变量，所以鱼皮没有选择将每个处理器封装为单独的类。大家也可以将每个处理器封装为单独的类（相当于设计模式中的策略模式），并且根据消息类别调用不同的处理器类。

5、WebSocket 配置

类似于编写 Spring MVC 的 Controller 接口，可以为指定的路径配置处理器和拦截器：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Resource
    private PictureEditHandler pictureEditHandler;

    @Resource
    private WsHandshakeInterceptor wsHandshakeInterceptor;
```

```
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry re

        registry.addHandler(pictureEditHandler, "/ws/picture/edit")
                .addInterceptors(wsHandshakeInterceptor)
                .setAllowedOrigins("*");
    }
}
```

之后，前端就可以通过 WebSocket 连接项目启动端口的 `/ws/picture/edit` 路径了。

扩展知识 - Disruptor 优化

1、现存的系统问题

WebSocket 通常是长连接，每个客户端都需要占用服务器资源。在 Spring WebSocket 中，每个 WebSocket 连接（客户端）对应一个独立的 `WebSocketSession` ，消息的处理是在该 `WebSocketSession` 所属的线程中执行。

如果 **同一个** WebSocket 连接（客户端）连续发送多条消息，服务器会 **按照接收的顺序依次同步处理**，而不是并发执行。这是为了保证每个客户端的消息处理是线程安全的。

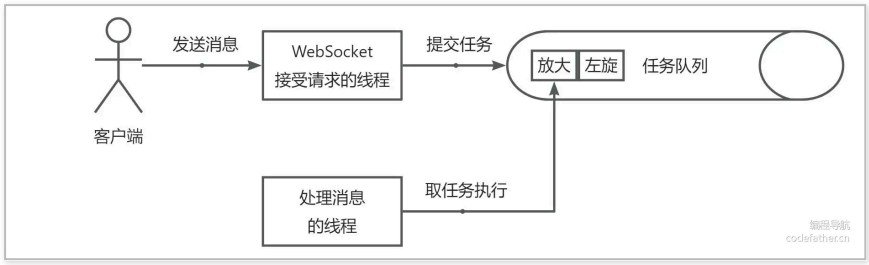
可以在 `handleTextMessage` 方法中增加 `Thread.sleep` 来测试一下。连续点击多次编辑操作，会发现每隔一段时间方法才会执行一次。

虽然多个客户端的消息处理是可以并发执行的，但是接受消息和具体处理某个消息使用的是 **同一个线程**。如果处理消息的耗时比较长，并发量又比较高，可能会导致系统响应时间变长，甚至因为资源耗尽而服务崩溃。

💡 为了便于理解，可以类比一下调用 Spring MVC 的某个接口时，如果该接口内部的耗时较长，请求线程就会一直阻塞，最终导致 Tomcat 请求连接数耗尽。

怎么解决这个问题呢？最简单的方法就是开一个线程专门来异步处理消息。但是我们还要保证操作是按照顺序同步给其

他客户端的，因此还需要引入一个队列，将任务按照顺序放到队列中，交给线程去处理。



其实上述的异步操作 + 从任务队列取任务执行，使用线程池就可以实现了。

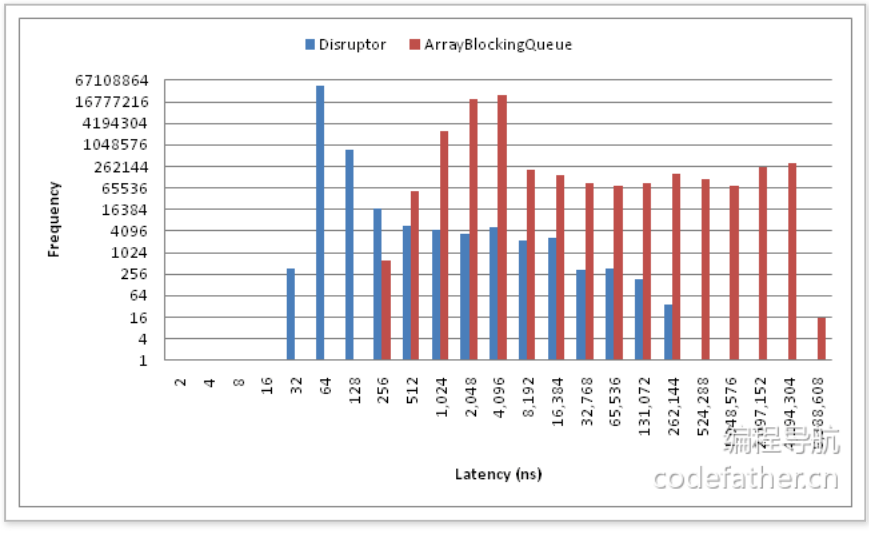
但对于协同编辑场景，需要尽可能地保证低延迟，因此我们选用一种高级技术 Disruptor 无锁队列来减少线程上下文的切换，能够在高并发场景下保持低延迟和高吞吐量。

此外，使用 Disruptor 还有一个优点，可以将任务放到队列中，通过优雅停机机制，在服务停止前执行完所有的任务，再退出服务，防止消息丢失。

2、Disruptor 介绍

Disruptor 是一种高性能的并发框架，由 LMAX（一个金融交易系统公司）开发，它是一种 无锁的环形队列 数据结构，用于解决高吞吐量和低延迟场景中的并发问题。支持生产者 - 消费者模式，可作为消息队列使用，适用于金融交易、实时数据处理、游戏事件等对并发和实时性要求较高的场景。

它最大的特点就是快、延迟低，非常低！



Disruptor 的核心思想是基于固定大小的 **环形缓冲区**（Ring Buffer），并通过序列化控制访问，以避免传统队列中常见的锁竞争问题。

它主要通过以下几点实现高性能的消息传递机制：

1. 环形缓冲区：使用固定大小的数组，可以复用内存，避免了频繁的内存分配和垃圾回收。
2. 无锁设计：依赖 CAS（Compare-And-Swap）和内存屏障，而不是传统的锁，降低了线程切换的开销。
3. 缓存友好：最大化利用 CPU 的缓存局部性，提高访问速度。
4. 序列号机制：通过序列号管理生产者和消费者的访问，保证数据一致性。
5. 多消费者模式：支持多消费者共享同一环形缓冲区，并能配置不同的消费策略（如依赖关系、并行消费等）。

Disruptor 与传统队列对比：

特性	**Disruptor**	**BlockingQueue**
并发控制	无锁（CAS + 内存屏障）	基于锁（ReentrantLock）
内存管理	固定长度的环形数组	动态数组或链表
性能	极高（百万级别消息 / 秒）	较低（数万消息 / 秒）
延迟	纳秒级别	毫秒级别
GC 压力	极低（数据复用）	较高（频繁创建新对象）
适用场景	高频实时消息处理、金融系统	一般生产者消费者模型

3、Disruptor 核心概念与工作流程

先了解 Disruptor 的核心概念：

- RingBuffer（环形缓冲区）：固定大小的循环数组，用于存储数据项，生产者和消费者共享该数据结构。
- Event（事件）：存储在 `RingBuffer` 中的数据对象，用于表示要传递的消息或数据。
- Producer（生产者）：负责向 `RingBuffer` 写入数据的角色。
- Consumer（消费者）：从 `RingBuffer` 中读取并处理数据的角色。
- Sequencer（序列器）：管理生产者与消费者的索引，确保并发安全的序列管理。
- SequenceBarrier（序列屏障）：控制消费者等待数据可用的机制，确保数据完整性。
- WaitStrategy（等待策略）：定义消费者如何等待新的数据（如自旋、自适应等待等）。
- EventProcessor（事件处理器）：集成了 `Consumer` 和 `SequenceBarrier`，用于更高级的消费控制。

而 Disruptor 是封装了 `RingBuffer`、`Producer` 和 `Consumer` 的核心管理类，用于协调所有组件的运行。

下面我举例来说明 Disruptor 的工作流程：

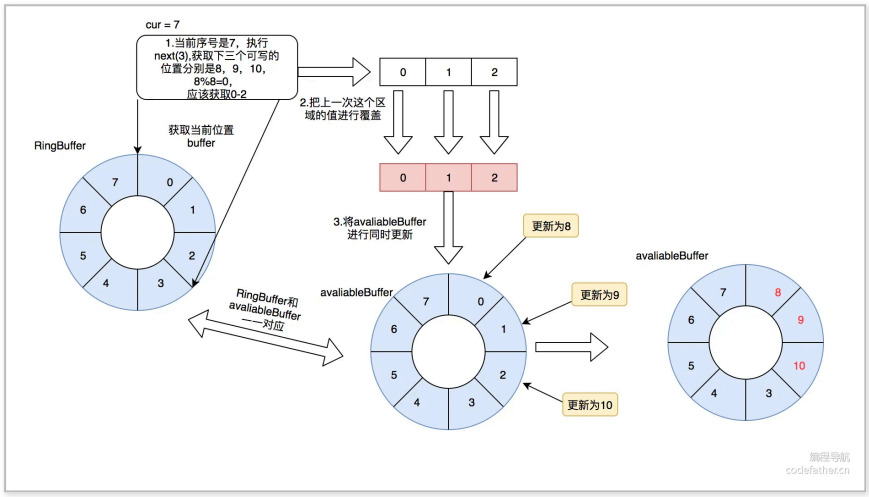
1. 环形队列初始化：创建一个固定大小为 8 的 RingBuffer（索引范围 0-7），每个格子存储一个可复用的事件对象，序号初始为 0。
2. 生产者写入数据：生产者申请索引 0（序号 0），将数据 "A" 写入事件对象，提交后序号递增为 1，下一个写入索引变为 1。
3. 消费者读取数据：消费者检查索引 0（序号 0），读取数据 "A"，处理后提交，序号递增为 1，下一个读取索引变为

1。

4. 环形队列循环使用：当生产者写入到索引 7（序号 7）后，索引回到 0（序号 8），形成循环存储，**但序号会持续自增以区分数据的先后顺序。**

5. 防止数据覆盖：如果生产者追上消费者，消费者尚未处理完数据，生产者会等待，确保数据不被覆盖。

下图是一个 Disruptor 生产者的模型，仅供参考，了解一下即可：



其实对大家来说，先将 Disruptor 当做一个高性能的队列来使用就可以了，可以向队列中添加事件并定义处理方式。感兴趣的同学可以阅读 这篇文章 深入了解 Disruptor 性能高的原因。

下面我们来引入 Disruptor 来优化代码。

4、Disruptor 实战

1）引入 Disruptor 依赖

```
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.4.2</version>
</dependency>
```

2）定义事件

事件是 Disruptor 执行的核心单位，在 `websocket.disruptor` 包中新建 PictureEditEvent 类，充当了上下文容器，所有处理消息所需的数据都被封装在其中。

```
@Data
public class PictureEditEvent {

    private PictureEditRequestMessage pictureEditRequestMessage;

    private WebSocketSession session;

    private User user;

    private Long pictureId;

}
```

3）定义事件处理器（消费者）

这里基本上是把 `PictureEditHandler` 分发消息的逻辑搬了过来，它的作用就是将不同类型的消息分发到对应的处理器中。

```
@Slf4j
@Component
public class PictureEditEventWorkHandler implements WorkHandler<Pictu

    @Resource
    @Lazy
    private PictureEditHandler pictureEditHandler;

    @Resource
    private UserService userService;

    @Override
    public void onEvent(PictureEditEvent event) throws Exception {
        PictureEditRequestMessage pictureEditRequestMessage = event.g
        WebSocketSession session = event.getSession();
        User user = event.getUser();
        Long pictureId = event.getPictureId();

        String type = pictureEditRequestMessage.getType();
        PictureEditMessageTypeEnum pictureEditMessageTypeEnum = Pictu

        switch (pictureEditMessageTypeEnum) {
            case ENTER_EDIT:
                pictureEditHandler.handleEnterEditMessage(pictureEdit
                break;
            case EDIT_ACTION:
                pictureEditHandler.handleEditActionMessage(pictureEdi
```

```
                break;
            case EXIT_EDIT:
                pictureEditHandler.handleExitEditMessage(pictureEditR
                break;
            default:
                PictureEditResponseMessage pictureEditResponseMessage
                pictureEditResponseMessage.setType(PictureEditMessage
                pictureEditResponseMessage.setMessage("消息类型错误");
                pictureEditResponseMessage.setUser(userService.getUse
                session.sendMessage(new TextMessage(JSONUtil.toJsonSt
        }
    }
}
```

4）添加 Disruptor 配置类，将我们刚定义的事件及处理器关联到 Disruptor 实例中：

```
@Configuration
public class PictureEditEventDisruptorConfig {

    @Resource
    private PictureEditEventWorkHandler pictureEditEventWorkHandler;

    @Bean("pictureEditEventDisruptor")
    public Disruptor<PictureEditEvent> messageModelRingBuffer() {

        int bufferSize = 1024 * 256;
        Disruptor<PictureEditEvent> disruptor = new Disruptor<>(
                PictureEditEvent::new,
                bufferSize,
                ThreadFactoryBuilder.create().setNamePrefix("pictureE
        );

        disruptor.handleEventsWithWorkerPool(pictureEditEventWorkHand

        disruptor.start();
        return disruptor;
    }
}
```

5、定义事件生产者

生产者负责将数据（事件）发到 Disruptor 的环形缓冲区中。为了保证在停机时所有的消息都能够被处理，我们通过 `shutdown` 方法完成 Disruptor 的优雅停机。

```
@Component
@Slf4j
public class PictureEditEventProducer {

    @Resource
```

```
    Disruptor<PictureEditEvent> pictureEditEventDisruptor;

    public void publishEvent(PictureEditRequestMessage pictureEditReq
        RingBuffer<PictureEditEvent> ringBuffer = pictureEditEventDis

        long next = ringBuffer.next();
        PictureEditEvent pictureEditEvent = ringBuffer.get(next);
        pictureEditEvent.setSession(session);
        pictureEditEvent.setPictureEditRequestMessage(pictureEditRequ
        pictureEditEvent.setUser(user);
        pictureEditEvent.setPictureId(pictureId);

        ringBuffer.publish(next);
    }


    @PreDestroy
    public void close() {
        pictureEditEventDisruptor.shutdown();
    }
}
```

6、修改 PictureEditHandler 的原有逻辑，改为使用事件生产者：

```
@Resource
private PictureEditEventProducer pictureEditEventProducer;

@Override
protected void handleTextMessage(WebSocketSession session, TextMessag

    PictureEditRequestMessage pictureEditRequestMessage = JSONUtil.to

    Map<String, Object> attributes = session.getAttributes();
    User user = (User) attributes.get("user");
    Long pictureId = (Long) attributes.get("pictureId");

    pictureEditEventProducer.publishEvent(pictureEditRequestMessage,
}
```

这样，我们就实现了基于 Disruptor 的异步消息处理机制，将原有的同步消息分发逻辑改造为高效解耦的异步处理模型，也更有利于代码的扩展。

扩展

1、为防止消息丢失，可以使用 Redis 等高性能存储保存执行的操作记录。

目前如果图片已经被编辑了，新用户加入编辑时没办法查看到已编辑的状态，这一点也可以利用 Redis 保存操作记录来解决，新用户加入编辑时读取 Redis 的操作记录即可。

2、每种类型的消息处理可以封装为独立的 Handler 处理器类，也就是采用策略模式。

3、支持分布式 WebSocket。实现思路很简单，只需要保证要编辑同一图片的用户连接的是相同的服务器即可，和游戏分服务器大区、聊天室分房间是类似的原理。

4、一些小问题的优化：比如 WebSocket 连接建立之后，如果用户退出了登录，这时 WebSocket 的连接是没有断开的。不过影响并不大，大家可以思考下怎么处理。

四、前端开发

前端开发主要集中在基础图片编辑组件 `ImageCropper.vue` 中。

1、基础代码

首先根据后端的枚举类和常量，在 `picture.ts` 中定义图片编辑消息类型、图片编辑动作：

```
export const PICTURE_EDIT_MESSAGE_TYPE_ENUM = {
  INFO: 'INFO',
  ERROR: 'ERROR',
  ENTER_EDIT: 'ENTER_EDIT',
  EXIT_EDIT: 'EXIT_EDIT',
  EDIT_ACTION: 'EDIT_ACTION',
};

export const PICTURE_EDIT_MESSAGE_TYPE_MAP = {
  INFO: '发送通知',
  ERROR: '发送错误',
  ENTER_EDIT: '进入编辑状态',
  EXIT_EDIT: '退出编辑状态',
  EDIT_ACTION: '执行编辑操作',
};

export const PICTURE_EDIT_ACTION_ENUM = {
  ZOOM_IN: 'ZOOM_IN',
  ZOOM_OUT: 'ZOOM_OUT',
  ROTATE_LEFT: 'ROTATE_LEFT',
  ROTATE_RIGHT: 'ROTATE_RIGHT',
};

export const PICTURE_EDIT_ACTION_MAP = {
  ZOOM_IN: '放大操作',
```

```
  ZOOM_OUT: '缩小操作',
  ROTATE_LEFT: '左旋操作',
  ROTATE_RIGHT: '右旋操作',
};
```

2、WebSocket 前端基础代码

为了让页面或组件的代码中能够更方便地使用 WebSocket 连接，我们可以先在 `utils` 目录下编写适用于图片编辑 WebSocket 连接的工具类。定义了：

- 连接 WebSocket 的地址
- WebSocket 各个事件的处理函数，比如连接成功和连接关闭事件，跟后端对应
- 向 WebSocket 服务端发送消息的函数等

代码如下，这段属于样板代码，大家了解一下即可，不必自己敲：

```
export default class PictureEditWebSocket {
  private pictureId: number
  private socket: WebSocket | null
  private eventHandlers: any

  constructor(pictureId: number) {
    this.pictureId = pictureId
    this.socket = null
    this.eventHandlers = {}
  }


  connect() {
    const url = `ws://localhost:8123/api/ws/picture/edit?pictureId=${
    this.socket = new WebSocket(url)


    this.socket.binaryType = 'blob'


    this.socket.onopen = () => {
      console.log('WebSocket 连接已建立')
      this.triggerEvent('open')
    }


    this.socket.onmessage = (event) => {
      const message = JSON.parse(event.data)
      console.log('收到消息:', message)
```

```
      const type = message.type
      this.triggerEvent(type, message)
    }


    this.socket.onclose = (event) => {
      console.log('WebSocket 连接已关闭:', event)
      this.triggerEvent('close', event)
    }


    this.socket.onerror = (error) => {
      console.error('WebSocket 发生错误:', error)
      this.triggerEvent('error', error)
    }
  }


  disconnect() {
    if (this.socket) {
      this.socket.close()
      console.log('WebSocket 连接已手动关闭')
    }
  }


  sendMessage(message: object) {
    if (this.socket && this.socket.readyState === WebSocket.OPEN) {
      this.socket.send(JSON.stringify(message))
      console.log('消息已发送:', message)
    } else {
      console.error('WebSocket 未连接，无法发送消息:', message)
    }
  }


  on(type: string, handler: (data?: any) => void) {
    if (!this.eventHandlers[type]) {
      this.eventHandlers[type] = []
    }
    this.eventHandlers[type].push(handler)
  }


  triggerEvent(type: string, data?: any) {
    const handlers = this.eventHandlers[type]
    if (handlers) {
      handlers.forEach((handler: any) => handler(data))
    }
  }
}
```

上述代码中比较巧妙的是，我们自定义了一套事件监听机制，使用工具类的组件可以通过 `on` 方法注册事件处理函数，然后通过 `triggerEvent` 函数触发事件处理函数。

3、图片编辑组件开发

1）定义响应式变量，包括正在编辑的用户、用户是否可以进入编辑、用户是否可以退出编辑，这些变量会用于控制页面的展示和编辑按钮是否可点击：

```
const loginUserStore = useLoginUserStore()
let loginUser = loginUserStore.loginUser

const editingUser = ref<API.UserVO>()

const canEnterEdit = computed(() => {
  return !editingUser.value
})

const canExitEdit = computed(() => {
  return editingUser.value?.id === loginUser.id
})

const canEdit = computed(() => {
  return editingUser.value?.id === loginUser.id
})
```

2）开发协同编辑操作相关的按钮元素：

```
<!-- 协同编辑操作 -->
<div>
  <a-space>
    <a-button v-if="editingUser" disabled> {{ editingUser.userName }}
    <a-button v-if="canEnterEdit" type="primary" ghost @click="enterE
    <a-button v-if="canExitEdit" danger ghost @click="exitEdit">退出编
  </a-space>
</div>
```

给所有的图片编辑操作按钮补充禁用状态，如果有其他人在编辑，则禁用按钮：

```
<a-space>
  <a-button @click="rotateLeft" :disabled="!canEdit">向左旋转</a-buttc
  <a-button @click="rotateRight" :disabled="!canEdit">向右旋转</a-butt
  <a-button @click="changeScale(1)" :disabled="!canEdit">放大</a-butt
  <a-button @click="changeScale(-1)" :disabled="!canEdit">缩小</a-but
  <a-button type="primary" :loading="loading" :disabled="!canEdit" @c
    确认
  </a-button>
</a-space>
```

效果如图：

3）初始化 WebSocket 连接，绑定事件：

```
let websocket: PictureEditWebSocket | null


const initWebsocket = () => {
  const pictureId = props.picture?.id
  if (!pictureId || !visible.value) {
    return
  }

  if (websocket) {
    websocket.disconnect()
  }

  websocket = new PictureEditWebSocket(pictureId)

  websocket.connect()


  websocket.on(PICTURE_EDIT_MESSAGE_TYPE_ENUM.INFO, (msg) => {
    console.log('收到通知消息：', msg)
    message.info(msg.message)
  })


  websocket.on(PICTURE_EDIT_MESSAGE_TYPE_ENUM.ERROR, (msg) => {
    console.log('收到错误消息：', msg)
    message.error(msg.message)
  })


  websocket.on(PICTURE_EDIT_MESSAGE_TYPE_ENUM.ENTER_EDIT, (msg) => {
    console.log('收到进入编辑状态消息：', msg)
    message.info(msg.message)
    editingUser.value = msg.user
  })
```

```
  websocket.on(PICTURE_EDIT_MESSAGE_TYPE_ENUM.EDIT_ACTION, (msg) => {
    console.log('收到编辑操作消息：', msg)
    message.info(msg.message)
    switch (msg.editAction) {
      case PICTURE_EDIT_ACTION_ENUM.ROTATE_LEFT:
        cropperRef.value.rotateLeft()
        break
      case PICTURE_EDIT_ACTION_ENUM.ROTATE_RIGHT:
        cropperRef.value.rotateRight()
        break
      case PICTURE_EDIT_ACTION_ENUM.ZOOM_IN:
        cropperRef.value.changeScale(1)
        break
      case PICTURE_EDIT_ACTION_ENUM.ZOOM_OUT:
        cropperRef.value.changeScale(-1)
        break
    }
  })


  websocket.on(PICTURE_EDIT_MESSAGE_TYPE_ENUM.EXIT_EDIT, (msg) => {
    console.log('收到退出编辑状态消息：', msg)
    message.info(msg.message)
    editingUser.value = undefined
  })
}

watchEffect(() => {
  initWebsocket()
})

onUnmounted(() => {

  if (websocket) {
    websocket.disconnect()
  }
  editingUser.value = undefined
})


const closeModal = () => {
  visible.value = false

  if (websocket) {
    websocket.disconnect()
  }
  editingUser.value = undefined
}
```

上述代码中的几个注意事项：

1. 定义了收到消息后的事件处理函数，比如收到编辑操作消息时，调用图片编辑器组件的对应操作方法，来同步编辑结果。收到有用户进入编辑状态的消息时，设置 editingUser 的值；收到有用户退出编辑状态的消息时，清空 editingUser 的值。

2. 及时释放 WebSocket 连接和资源：在组件销毁时（onUnmounted 函数）、弹窗关闭时（closeModal 函数）、重新连接时（initWebsocket 函数开头）都要释放连接并重置正在编辑的用户。

4）编辑发送 WebSocket 消息的函数，包括进入编辑状态、退出编辑状态、执行编辑图片操作：

```
const enterEdit = () => {
  if (websocket) {

    websocket.sendMessage({
      type: PICTURE_EDIT_MESSAGE_TYPE_ENUM.ENTER_EDIT,
    })
  }
}


const exitEdit = () => {
  if (websocket) {

    websocket.sendMessage({
      type: PICTURE_EDIT_MESSAGE_TYPE_ENUM.EXIT_EDIT,
    })
  }
}


const editAction = (action: string) => {
  if (websocket) {

    websocket.sendMessage({
      type: PICTURE_EDIT_MESSAGE_TYPE_ENUM.EDIT_ACTION,
      editAction: action,
    })
  }
}
```

所有编辑图片的操作都要补充上发送 WebSocket 消息：

```
const rotateLeft = () => {
  cropperRef.value.rotateLeft()
  editAction(PICTURE_EDIT_ACTION_ENUM.ROTATE_LEFT)
}


const rotateRight = () => {
  cropperRef.value.rotateRight()
  editAction(PICTURE_EDIT_ACTION_ENUM.ROTATE_RIGHT)
}


const changeScale = (num: number) => {
  cropperRef.value.changeScale(num)
  if (num > 0) {
    editAction(PICTURE_EDIT_ACTION_ENUM.ZOOM_IN)
```

```
  } else {
    editAction(PICTURE_EDIT_ACTION_ENUM.ZOOM_OUT)
  }
}
```

4、协同编辑范围控制

只有团队空间才支持协作编辑，否则还是跟之前一样，默认就进入可编辑状态。

为此，我们需要在 `ImageCropper` 组件中获取到空间信息，并判断是否为团队空间。

1）可以由引入该组件的父页面 `AddPicturePage` 获取空间信息：

```
const space = ref<API.SpaceVO>()


const fetchSpace = async () => {

  if (spaceId.value) {
    const res = await getSpaceVoByIdUsingGet({
      id: spaceId.value,
    })
    if (res.data.code === 0 && res.data.data) {
      space.value = res.data.data
    }
  }
}

watchEffect(() => {
  fetchSpace()
})
```

然后传入给组件：

```
<ImageCropper
  ref="imageCropperRef"
  :imageUrl="picture?.url"
  :picture="picture"
  :spaceId="spaceId"
  :space="space"
  :onSuccess="onSuccess"
/>
```

2）在图片编辑组件中新增 space 属性：

```
interface Props {
  imageUrl?: string
  picture?: API.PictureVO
  spaceId?: number
  space?: API.SpaceVO
  onSuccess?: (newPicture: API.PictureVO) => void
}
```

然后就可以根据 space 判断是否为团队空间了，定义一个变量便于复用：

```
const isTeamSpace = computed(() => {
  return props.space?.spaceType === SPACE_TYPE_ENUM.TEAM;
})
```

3）使用该变量来控制协同编辑的范围，包括是否可编辑、是否初始化 WebSocket 连接：

```
const canEdit = computed(() => {

  if (!isTeamSpace.value) {
    return true
  }
  return editingUser.value?.id === loginUser.id
})

watchEffect(() => {

  if (isTeamSpace.value) {
    initWebsocket()
  }
})
```

以及是否展示协同编辑操作的按钮：

```
<!-- 协同编辑操作 -->
<div v-if="isTeamSpace">
</div>
```

5、测试

OK，接下来就可以愉快地测试了。可以依次验证：

1. 编辑功能是否可以正常使用，能否正常进入和退出编辑状态、并保存图片

2. 验证资源能否正常释放

3. 验证用户编辑时，其他用户能否实时查看到效果

4. 验证用户编辑时，其他用户是否可以编辑

5. 验证用户 A 退出编辑或关闭弹窗后，其他用户是否可以进入编辑；用户 A 再次进入时，能否跟其他用户的操作保持同步。

效果如图：

扩展

1、支持 WebSocket 断线重连，应对服务器突然宕机的情况

2、如果没有用户进入编辑状态，打开图片编辑弹窗时自动进入编辑，不需要手动点击按钮进入编辑

3、新增一个 “用户保存” 事件，某用户点击保存后，关闭其他用户的编辑弹窗，并且更新当前展示的图片

4、可能还会有一些细节问题，比如新用户打开编辑弹窗时，无法获取到正在编辑的用户信息、也无法获取到当前已编辑的图片状态，大家可以自行测试和优化。

5、可以通过传递 CSS 样式的方式实现裁切框区域的实时协作。但其实移动编辑框时并没有修改图片，所以作用不是很大。

以上就是本期教程，希望大家通过这些示例图片协同编辑操作，学会实时协同业务的设计和开发方法，以后开发更复杂的实时协作系统都是类似的~

全文完