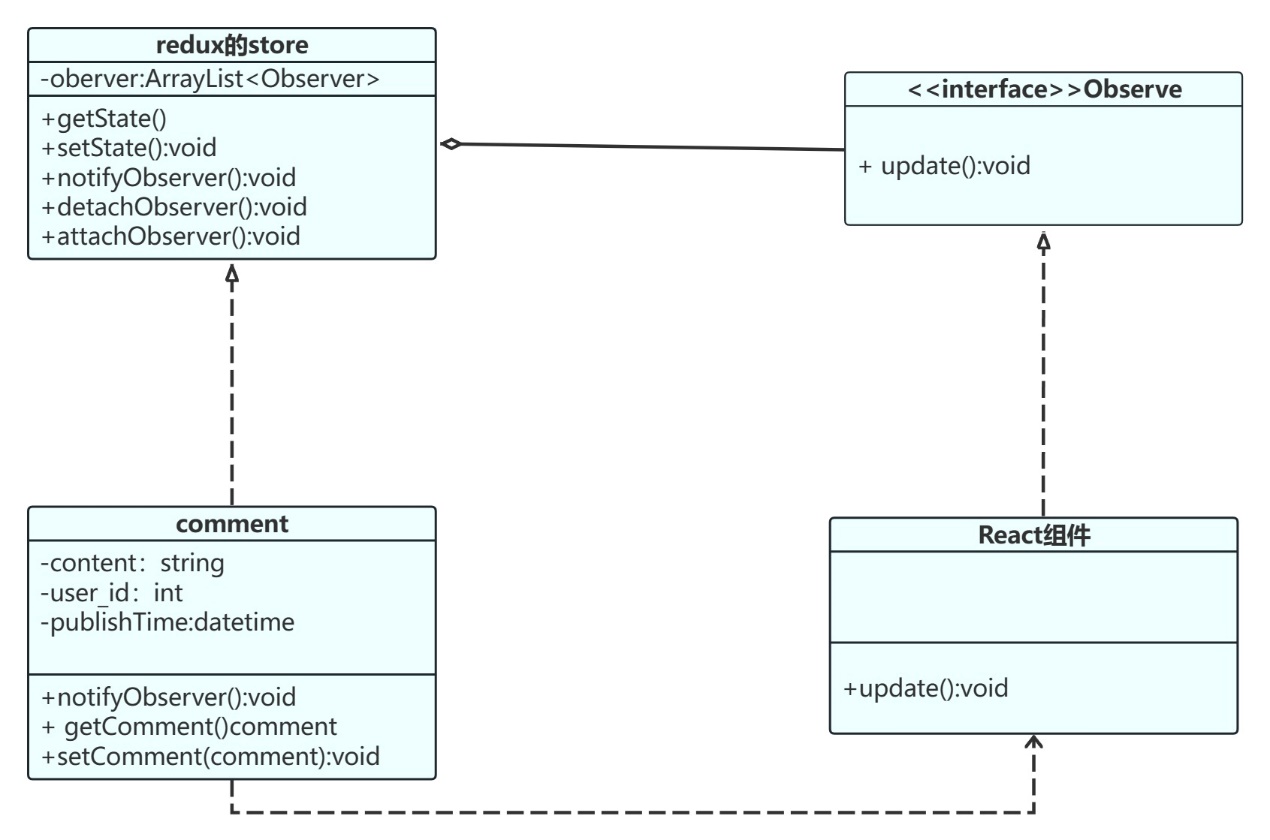
观察者模式

在我们的评价交流平台项目中，如果用户发布新评论之后，其他的用户页面应该需要重新渲染。而我们使用Redux状态管理库管理从后端读取评论。Redux中的store（存储库）可以被视为“被观察者”（Subject），而所有订阅了store的组件或中间件都可以被视为“观察者”（Observer）。当store中的状态（state）发生变化时，Redux会通知所有订阅了该状态的“观察者”，并触发它们的重新渲染或执行相应的操作。



优点：

通过观察者模式，组件之间的通信变得解耦合，组件不需要直接相互引用，而是通过状态的变化来进行通信，降低了组件之间的耦合度。观察者模式使得组件的逻辑更加清晰，状态的变化和更新逻辑被分离到不同的组件中，使得代码更易于理解和维护。通过观察者模式，可以轻松地添加新的观察者或主题，扩展系统的功能和逻辑。

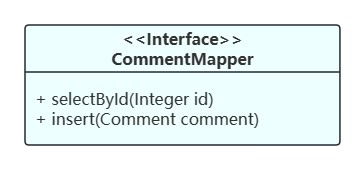
缺点：

如果观察者过多或主题变化频繁，可能会导致性能问题，因为每次状态变化都会触发所有观察者的更新。观察者模式引入了额外的复杂性，需要维护观察者和主题之间的关系，可能会增加代码的复杂度。

单例模式

在我们小组的Spring Boot后端项目中，Mapper类、Service类都是声明的Bean，Bean是一个由Spring IoC容器管理的对象。在Spring Boot中，Bean默认情况下是单例模式的。这是因为Spring框架遵循控制反转（IoC）和依赖注入（DI）的原则，其中Bean的生命周期和依赖关系都由Spring容器（如ApplicationContext）来管理。Spring容器负责创建Bean的实例，并在需要时将其注入到其他Bean中。

默认情况下，当Spring容器启动时，它会为每个Bean类型创建一个实例，并在整个应用程序的生命周期内维护这个实例。这就是所谓的单例模式，即一个类只有一个实例，并且这个实例是全局可访问的。



优点：

单例模式确保了类的唯一实例，在整个应用中可以方便地通过单例实例来访问共享资源，简化了代码的调用和管理。由于单例模式只创建一个实例，可以节省系统资源，减少了内存消耗和性能开销。单例模式可以避免多线程环境下可能出现的竞态条件，保证了线程安全性。

缺点：

单例模式没有抽象层，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。单例类的职责过重，在一定程度上违背了“单一职责原则”。