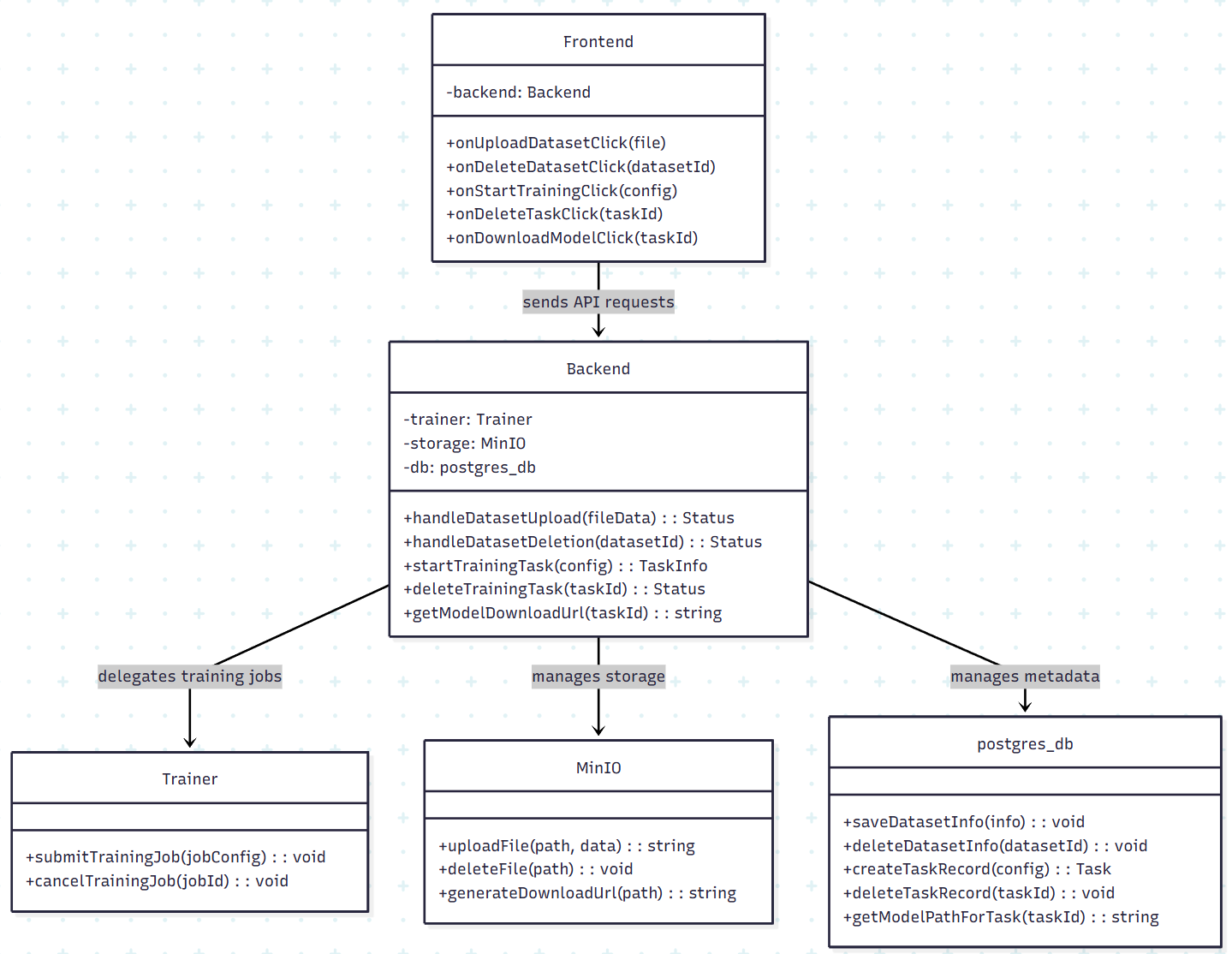
在当前项目中使用了外观（Facade）、代理（Proxy）和状态（State）设计模式。下面将介绍这三种模式在项目中的具体应用。

1. 外观模式 (Facade)
   1. UML 类图

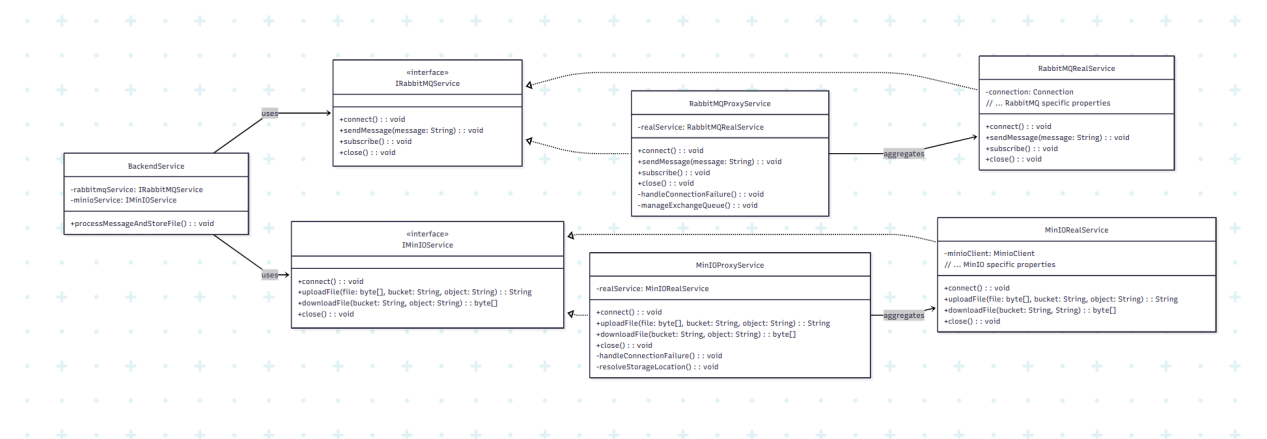


* 1. 设计阐述

在当前项目中，Backend服务是外观模式最核心的体现。我们后端的Trainer、MinIO和postgres\_db都是独立服务，如果让Frontend直接与它们沟通，会导致前端逻辑变得异常复杂和混乱。因此，我们设立Backend服务作为整个后台系统的“总指挥”，它封装了所有复杂的内部交互，只对外提供一组清晰、简洁的API接口。

举个例子，当用户发起一次训练任务时，Frontend只需要调用Backend提供的一个简单接口即可。Backend内部则会负责完成一系列的协调工作：它先与postgres\_db交互来创建任务记录，再调用Trainer服务来提交实际的计算作业。这种方式极大地简化了前端的开发，并使得后端各个组件可以独立地修改和演进，彼此互不影响，保证了整个系统的灵活性和可维护性。

1. 代理模式 (Proxy)
   1. UML 类图

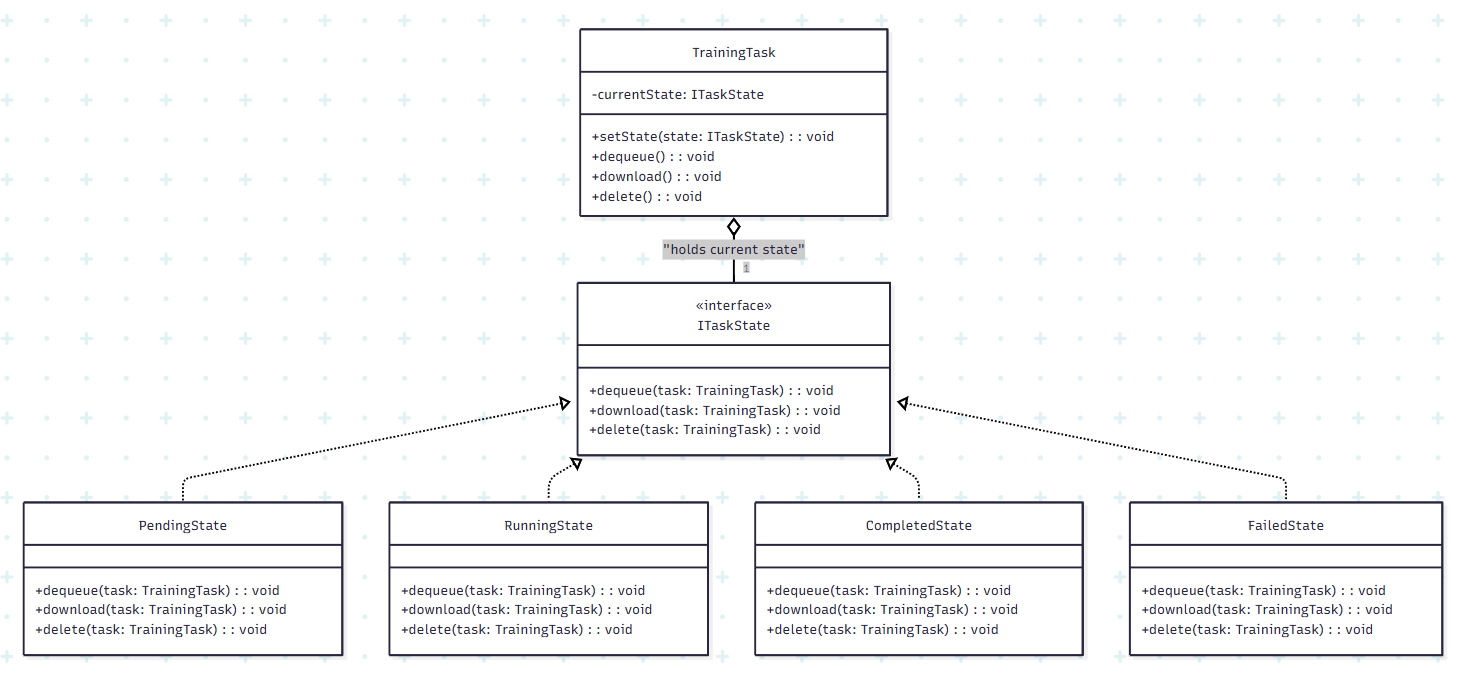


* 1. 设计阐述

我们系统依赖的RabbitMQ和MinIO这类基础网络服务，其直接连接往往是脆弱的，容易受到网络波动的影响。为了在不修改这些核心服务代码的前提下增强其稳定性，我们引入了代理模式。我们为RabbitMQRealService和MinIORealService分别创建了对应的ProxyService作为它们的“智能代理”。Backend服务依赖的是IRabbitMQService和IMinIOService这两个接口，因此可以在运行时透明地将代理注入进去。

这些代理层（Proxy）是我们提升系统健壮性的关键。例如，当Backend要发送一条RabbitMQ消息时，请求会先被RabbitMQProxyService拦截。代理会先检查连接状态，如果发现连接中断，它可以悄无声息地执行重连逻辑（handleConnectionFailure），然后再将请求安全地委托给真正的RabbitMQRealService。通过这种方式，所有复杂的故障处理和连接管理逻辑都被封装在了代理层，使得我们的主业务逻辑（Backend）和核心服务（RealService）都能保持代码的纯净和简洁。

1. 状态模式 (State)
   1. UML 类图



* 1. 设计阐述

一个训练任务（TrainingTask）的生命周期相当复杂，它在pending、running、completed和failed等不同状态下的行为和允许的操作都截然不同。如果使用大量的if/else或switch语句来管理这些状态，TrainingTask类很快会变得臃肿不堪。状态模式优雅地解决了这个问题，它将每一种状态下的行为逻辑都封装到各自独立的类中（如PendingState, CompletedState等），这些类都实现了统一的ITaskState接口。

TrainingTask对象本身不再包含任何状态判断逻辑，它只是持有一个指向当前状态对象的引用，并将所有操作（如dequeue、download）都委托给当前状态对象去处理。更精妙的是，状态的转换也由状态对象自身负责。比如，当PendingState对象成功执行了dequeue操作后，它会主动调用task.setState(new FailedState())，将TrainingTask的上下文切换到下一个状态。这个设计不仅让TrainingTask类的代码变得极其稳定和简洁，也使得未来增加新状态（比如PausedState）变得轻而易举，我们只需添加一个新的状态类即可，完全符合开闭原则。