实验十三 设计模块（二）

实验目的：

面向对象设计原则

实验内容：

论述利斯科夫替换原则（里氏代换原则）（李雨桐）

里氏代换原则（Liskov Substitution Principle, LSP）是面向对象设计的SOLID原则之一，提出了子类对象应该能够替换父类对象，并且程序行为保持不变。简言之，如果S是T的子类型，那么类型T的对象可以用类型S的对象替换，而不会改变程序的正确性。

在苍穹外卖项目中，我们可以通过一个订单处理的例子来说明里氏代换原则的使用。

场景：订单处理

假设我们有一个基本的订单类，以及用于处理普通订单和高级订单的不同处理类。

初始设计

我们首先设计一个基础订单类 `Order`，以及一个处理普通订单的类 OrderProcessor：

class Order:

    def \_\_init\_\_(self, order\_id: int):

        self.order\_id = order\_id

class OrderProcessor:

    def process(self, order: Order):

        print(f"Processing order {order.order\_id}")

扩展设计

现在，我们引入一个新的订单类型 PremiumOrder，它继承自 Order 类，并增加了一些额外的属性或方法：

class PremiumOrder(Order):

    def \_\_init\_\_(self, order\_id: int, membership\_level: str):

        super().\_\_init\_\_(order\_id)

        self.membership\_level = membership\_level

    def apply\_premium\_discount(self):

        print(f"Applying premium discount for order {self.order\_id} with membership level {self.membership\_level}")

为了处理PremiumOrder类型的订单，我们可以创建一个新的处理类 PremiumOrderProcessor：

class PremiumOrderProcessor(OrderProcessor):

    def process(self, order: PremiumOrder):

        order.apply\_premium\_discount()

        print(f"Processing premium order {order.order\_id} with membership level {order.membership\_level}")

使用里氏代换原则

根据里氏代换原则，我们应该能够用 PremiumOrderProcessor来替换OrderProcessor，并且处理Order类型的订单，而不会改变程序的正确性。我们修改OrderProcessor类，使其可以处理普通订单和高级订单。

首先，确保 `PremiumOrderProcessor` 可以处理所有 `OrderProcessor` 可以处理的订单：

class PremiumOrderProcessor(OrderProcessor):

    def process(self, order: Order):

        if isinstance(order, PremiumOrder):

            order.apply\_premium\_discount()

            print(f"Processing premium order {order.order\_id} with membership level {order.membership\_level}")

        else:

            super().process(order)

这样，我们可以无缝地替换处理器，而不会破坏代码的正确性：

def handle\_order(processor: OrderProcessor, order: Order):

    processor.process(order)

# 创建订单和处理器

order = Order(order\_id=1)

premium\_order = PremiumOrder(order\_id=2, membership\_level="Gold")

order\_processor = OrderProcessor()

premium\_order\_processor = PremiumOrderProcessor()

# 使用普通订单处理器

handle\_order(order\_processor, order)  # Output: Processing order 1

# 使用高级订单处理器处理普通订单

handle\_order(premium\_order\_processor, order)  # Output: Processing order 1

# 使用高级订单处理器处理高级订单

handle\_order(premium\_order\_processor, premium\_order)

# Output:

# Applying premium discount for order 2 with membership level Gold

# Processing premium order 2 with membership level Gold

总结

通过这个例子，我们展示了如何应用里氏代换原则，使得子类 `PremiumOrderProcessor` 能够替换父类 `OrderProcessor`，而不改变程序的行为。具体实现中：

1.扩展父类功能：子类 `PremiumOrderProcessor` 扩展了父类 `OrderProcessor` 的功能，使其能够处理高级订单的特有需求。

2.确保行为一致：通过 `super().process(order)` 确保 `PremiumOrderProcessor` 能处理所有父类可以处理的订单，保持程序行为的一致性。

这种设计方式确保了代码的灵活性和可维护性，使得系统能够随着业务需求的变化而扩展，同时不影响现有的功能。

单一职责原则（张勇）

单一职责原则（Single Responsibility Principle, SRP）是面向对象设计的五大基本原则之一，该原则强调一个类或者模块应当只负责一个功能领域中的相关职责，也就是说，一个类应该只有一个引起它变化的原因。这样做的目的是为了提高代码的可维护性、可读性和可重用性。

单一职责原则的核心思想

高内聚低耦合：确保类的内部功能紧密相关，减少类与类之间的不必要依赖。

易于修改和扩展：当需求变化时，只影响到系统中的一个小部分，而不是波及整个系统。

提高可测试性：每个类负责单一职责，测试时可以更加聚焦，容易验证其正确性。

在外卖系统中的应用实例

假设我们正在设计一个外卖系统的后端服务，可以将系统分解为几个关键组件，每个组件遵循单一职责原则设计：

用户管理模块：

职责：负责用户注册、登录、信息修改等操作。

实践：设计一个UserManager类，仅包含与用户管理相关的逻辑和数据访问。例如，处理密码加密、验证用户身份等功能，而不涉及订单处理逻辑。

商品管理模块：

职责：管理餐厅菜单、菜品上下架、价格调整等。

实践：创建一个MenuItemManager类，专注于菜品的增删改查操作，与用户管理和订单处理逻辑分离。

订单处理模块：

职责：处理订单的创建、修改、取消以及支付状态的更新。

实践：设计一个OrderProcessor类，它负责接收用户提交的订单请求，验证库存、计算总价、更新订单状态等，但不直接处理用户认证或菜品信息管理。

配送管理模块：

职责：调度骑手、跟踪订单配送状态、通知用户等。

实践：实现一个DeliveryManager类，专门处理与配送相关的逻辑，如根据订单位置分配最合适的骑手，更新配送进度等，独立于其他模块。

单一职责原则促使我们将外卖系统分解为多个具有明确职责的模块或类。这样设计不仅使得代码结构清晰，便于团队成员分工合作，而且在面对需求变更时，能够快速定位到需要修改的部分，降低修改带来的风险，同时提高系统的可测试性和可扩展性。遵循这一原则是构建高质量、可维护软件系统的关键。

开闭原则（王攀）

1.实现不同的支付方式。

初始设计

最初，我们可能只实现了一种支付方式，比如通过信用卡支付。我们的支付处理类可能看起来像这样：

class PaymentProcessor:

    def process\_payment(self, amount: float, payment\_type: str):

        if payment\_type == "credit\_card":

            self.process\_credit\_card\_payment(amount)

        else:

            raise ValueError("Unsupported payment type")

    def process\_credit\_card\_payment(self, amount: float):

        print(f"Processing credit card payment of {amount}")

在这个初始设计中，PaymentProcessor 类直接处理信用卡支付，如果我们要增加新的支付方式（例如，PayPal或支付宝），我们需要修改 process\_payment 方法。

使用开闭原则进行重构

为了符合开闭原则，我们可以重构代码，使得 PaymentProcessor类对扩展开放，对修改关闭。我们可以使用策略模式来实现这一点。首先，定义一个支付策略接口：

from abc import ABC, abstractmethod

class PaymentStrategy(ABC):

    @abstractmethod

    def pay(self, amount: float):

        pass

然后，为每种支付方式实现具体的支付策略：

class CreditCardPayment(PaymentStrategy):

    def pay(self, amount: float):

        print(f"Processing credit card payment of {amount}")

class PayPalPayment(PaymentStrategy):

    def pay(self, amount: float):

        print(f"Processing PayPal payment of {amount}")

接着，修改 PaymentProcessor类以使用这些策略：

class PaymentProcessor:

    def \_\_init\_\_(self, strategy: PaymentStrategy):

        self.\_strategy = strategy

    def process\_payment(self, amount: float):

        self.\_strategy.pay(amount)

现在，PaymentProcessor类的 process\_payment 方法不需要再修改，只需传入不同的支付策略实例即可实现不同的支付方式：

# 使用信用卡支付

credit\_card\_payment = CreditCardPayment()

processor = PaymentProcessor(credit\_card\_payment)

processor.process\_payment(100.0)

# 使用PayPal支付

paypal\_payment = PayPalPayment()

processor = PaymentProcessor(paypal\_payment)

processor.process\_payment(150.0)

### 添加新支付方式

如果要添加一种新的支付方式，比如支付宝，只需实现一个新的策略类，而无需修改现有的 PaymentProcessor类：

class AlipayPayment(PaymentStrategy):

    def pay(self, amount: float):

        print(f"Processing Alipay payment of {amount}")

# 使用支付宝支付

alipay\_payment = AlipayPayment()

processor = PaymentProcessor(alipay\_payment)

processor.process\_payment(200.0)

通过这种方式，我们实现了对支付方式的扩展，而无需修改已有的代码，符合开闭原则。这使得系统更具灵活性和可维护性，同时减少了因修改已有代码而引入错误的风险。

2.订单处理流程的扩展。

初始设计

假设我们的订单处理类最初只处理简单的订单确认和取消流程：

class OrderProcessor:

    def process\_order(self, order, action):

        if action == "confirm":

            self.confirm\_order(order)

        elif action == "cancel":

            self.cancel\_order(order)

        else:

            raise ValueError("Unsupported action")

    def confirm\_order(self, order):

        print(f"Order {order.id} confirmed.")

    def cancel\_order(self, order):

        print(f"Order {order.id} cancelled.")

在这个设计中，如果我们要添加新的订单操作（例如，退单或修改订单），我们需要修改 process\_order 方法。

使用开闭原则进行重构

为了符合开闭原则，我们可以使用命令模式来重构代码，使得OrderProcessor类对扩展开放，对修改关闭。首先，定义一个订单命令接口：

from abc import ABC, abstractmethod

class OrderCommand(ABC):

    @abstractmethod

    def execute(self, order):

        pass

然后，为每种订单操作实现具体的命令类：

class ConfirmOrderCommand(OrderCommand):

    def execute(self, order):

        print(f"Order {order.id} confirmed.")

class CancelOrderCommand(OrderCommand):

    def execute(self, order):

        print(f"Order {order.id} cancelled.")

接着，修改OrderProcessor类以使用这些命令：

class OrderProcessor:

    def \_\_init\_\_(self):

        self.commands = {}

    def register\_command(self, action: str, command: OrderCommand):

        self.commands[action] = command

    def process\_order(self, order, action: str):

        if action in self.commands:

            self.commands[action].execute(order)

        else:

            raise ValueError("Unsupported action")

现在OrderProcessor类的process\_order方法不需要再修改，只需注册不同的命令实例即可实现不同的订单操作：

# 创建订单处理器

processor = OrderProcessor()

# 注册确认和取消命令

processor.register\_command("confirm", ConfirmOrderCommand())

processor.register\_command("cancel", CancelOrderCommand())

# 处理订单

order = Order(id=123)

processor.process\_order(order, "confirm")

processor.process\_order(order, "cancel")

### 添加新订单操作

如果要添加一种新的订单操作，比如退单，只需实现一个新的命令类，并注册到 OrderProcessor中，而无需修改现有的代码：

class RefundOrderCommand(OrderCommand):

    def execute(self, order):

        print(f"Order {order.id} refunded.")

# 注册退单命令

processor.register\_command("refund", RefundOrderCommand())

# 处理退单操作

processor.process\_order(order, "refund")

3.另一个例子：促销活动

在苍穹外卖项目中，促销活动是常见的需求。假设我们最初的设计只支持一种简单的折扣活动：

class Promotion:

    def apply\_promotion(self, order, promotion\_type):

        if promotion\_type == "discount":

            self.apply\_discount(order)

        else:

            raise ValueError("Unsupported promotion type")

    def apply\_discount(self, order):

        print(f"Applying discount to order {order.id}")

为了符合开闭原则，我们可以使用策略模式来重构代码。首先，定义一个促销策略接口：

from abc import ABC, abstractmethod

class PromotionStrategy(ABC):

    @abstractmethod

    def apply(self, order):

        pass

然后，为每种促销活动实现具体的促销策略：

class DiscountPromotion(PromotionStrategy):

    def apply(self, order):

        print(f"Applying discount to order {order.id}")

class FreeDeliveryPromotion(PromotionStrategy):

    def apply(self, order):

        print(f"Applying free delivery to order {order.id}")

接着，修改 Promotion类以使用这些策略：

class Promotion:

    def \_\_init\_\_(self, strategy: PromotionStrategy):

        self.\_strategy = strategy

    def apply\_promotion(self, order):

        self.\_strategy.apply(order)

现在，Promotion类的apply\_promotion方法不需要再修改，只需传入不同的促销策略实例即可实现不同的促销活动：

# 使用折扣促销

discount\_promotion = DiscountPromotion()

promotion = Promotion(discount\_promotion)

promotion.apply\_promotion(order)

# 使用免费配送促销

free\_delivery\_promotion = FreeDeliveryPromotion()

promotion = Promotion(free\_delivery\_promotion)

promotion.apply\_promotion(order)

通过这种方式，我们实现了对促销活动的扩展，而无需修改已有的代码，符合开闭原则。这同样使得系统更具灵活性和可维护性，同时减少了因修改已有代码而引入错误的风险。

4.总结

在这三个例子中，通过引入策略模式和命令模式，我们实现了以下目标：

（1）对扩展开放：可以轻松添加新的支付方式、订单操作和促销活动，只需实现新的策略类或命令类。

（2）对修改关闭：无需修改已有的处理逻辑，减少了引入错误的风险，提升了系统的稳定性和可维护性。

（3）提高灵活性：业务逻辑的变化可以通过添加新类来实现，而不是修改现有类，符合面向对象设计的基本原则。

这些设计模式和开闭原则的结合应用，使得系统能够更好地应对变化和扩展需求，提高了代码的可维护性和可扩展性。

德（迪）米特法则（刘易）

德米特法则（Law of Demeter）强调模块间的低耦合和高内聚。结合《苍穹外卖》项目，可以通过以下具体方式运用德米特法则：

1. 封装内部数据

用户模块（User）：

封装用户数据：将用户的详细信息（如用户名、密码、联系方式）封装起来，只通过方法提供必要的信息访问。

订单管理：通过方法来获取用户的订单列表，而不是直接暴露订单数据。例如，通过 getUserOrders() 返回订单概览，而不是返回订单对象。

订单模块（Order）：

封装订单详情：订单类应封装订单详细信息，通过方法获取订单状态和总金额等信息，而不是直接暴露订单项列表。例如，提供 getOrderStatus() 和 calculateTotalAmount() 方法。

2. 使用中介者模式

订单管理模块（OrderManager）：

订单处理：用户创建订单、餐厅接收订单和外卖员接单配送的操作都通过订单管理模块进行，避免用户、餐厅和外卖员模块的直接通信。

示例：用户下单时，调用订单管理模块的 createOrder(userId, restaurantId, items) 方法，由订单管理模块处理订单创建、通知餐厅和分配外卖员等操作。

3. 避免链式调用

简化调用链：

用户模块：提供简化方法，如 getOrderDetails(orderId)，内部处理获取订单详细信息，避免用户模块调用 user.getOrder().getItems()。

订单模块：提供 getOrderSummary() 方法，返回订单的简要信息，避免外部模块访问内部复杂结构。

4. 接口隔离

模块化设计：

用户模块接口：只提供与用户相关的操作方法，如注册、登录、查看订单等，而不会暴露订单或配送的细节。

订单模块接口：专注于订单创建、支付、状态更新等操作，不直接与用户模块或外卖员模块交互。

具体实例应用

用户注册和登录

封装用户数据：用户类封装用户名、密码、联系方式等，通过 getUserDetails() 提供访问，内部处理登录逻辑，确保用户数据安全。

订单处理流程

订单管理：用户在客户端下单时，通过调用订单管理模块的 createOrder() 方法，由该模块处理订单创建、支付和状态更新。餐厅和外卖员模块通过订单管理模块获取订单信息和状态更新。

外卖员接单和配送

中介者模式：外卖员模块通过调用订单管理模块的 acceptOrder(orderId) 方法，处理接单和配送任务，订单管理模块更新订单状态并通知相关方。

评价反馈系统

接口隔离：用户对订单的评价和反馈通过订单管理模块记录和处理，评价数据由订单管理模块统一管理，不直接暴露给用户或餐厅模块。

具体设计方案

用户模块设计

用户类（User）：封装用户信息和订单列表，提供 register(), login(), getUserOrders() 方法。

示例：getUserOrders() 方法返回用户订单概览，避免直接暴露订单对象。

订单模块设计

订单类（Order）：封装订单详情，提供 getOrderStatus(), calculateTotalAmount() 方法。

示例：getOrderStatus() 返回订单当前状态，避免外部直接访问订单详情。

订单管理模块设计

订单管理类（OrderManager）：处理订单创建、通知餐厅和分配外卖员等操作，提供 createOrder(), assignDelivery() 方法。

示例：createOrder(userId, restaurantId, items) 方法创建新订单，并通知相关方处理。

依赖倒转原则（丁弘扬）

依赖倒转原则的核心思想是：要依赖于抽象，而不是依赖于具体的实现。这意味着高层模块不应该依赖于低层模块的具体实现，它们都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

基本介绍：

1. 高层模块与低层模块的依赖：高层模块（通常是封装了复杂逻辑的类）和低层模块（实现基本或初级操作的类）之间，传统上高层模块会依赖于低层模块的具体实现。但依赖倒转原则要求这两者都应该依赖于抽象，而不是具体的实现。

2. 抽象与细节的关系：抽象（如接口或抽象类）不应该依赖于细节（具体的类），而是细节应该依赖于抽象。这意味着设计的中心思想应当是面向接口编程，而不是面向具体的实现。

3. 设计理念：依赖倒转原则基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定得多。因此，以抽象为基础搭建的架构比以细节为基础的架构要稳定得多。

依赖倒转原则的重要性：

1. 提高代码的可维护性：当高层模块不直接依赖于具体的实现时，低层模块的实现细节可以更容易地改变，而不需要修改高层模块的代码。这降低了模块之间的耦合度，使得系统更加易于维护。

2.增强代码的扩展性：通过依赖抽象，可以轻松地为系统添加新的功能或行为，只需要实现相应的抽象接口或继承抽象类即可。这降低了代码扩展的复杂性。

3.促进测试：当高层模块依赖于抽象而不是具体实现时，可以使用模拟（Mock）或桩（Stub）对象来替代实际的低层模块，从而更容易地进行单元测试。

在实际应用中，依赖倒转原则通常通过接口和抽象类来实现。例如，在一个系统中，如果需要从不同的数据源（如文件、数据库、网络等）读取数据，可以定义一个抽象的数据读取接口。高层模块（如数据处理类）可以依赖于这个接口，而不是具体的数据读取实现。这样，当需要添加新的数据读取方式时，只需实现一个新的数据读取类，而无需修改高层模块的代码。

依赖倒转原则是一种重要的面向对象设计原则，它强调在软件系统中应该通过抽象来解耦高层模块与低层模块之间的依赖关系。这种设计思想有助于提高系统的稳定性和可维护性，降低模块之间的耦合度。通过面向接口编程，可以实现更好的代码复用和扩展性。

在外卖系统应用实例：

定义抽象接口

1.用户服务接口：定义用户注册、登录、信息获取等操作的接口，使得订单管理、餐厅管理等模块不直接依赖于具体的用户管理实现。

2.餐厅服务接口：包含餐厅信息查询、菜单获取、订单处理等功能的接口，供点餐业务模块等高层模块使用。

3.订单服务接口：定义订单创建、修改、查询、支付等操作的接口，确保高层模块不直接依赖于订单数据库访问层。

高层模块不依赖低层模块

1.点餐业务模块：该模块依赖于用户服务接口和餐厅服务接口，而不是直接依赖于用户管理或餐厅管理的具体实现。这使得点餐业务模块更加灵活，可以适应不同的用户管理和餐厅管理方式。

2.支付模块：支付模块依赖于订单服务接口，而不是直接依赖于订单数据库访问层。这允许支付模块与订单处理流程保持松散的耦合关系，便于后续的维护和扩展。

合成复用原则（李雨桐）

合成复用原则（Composition over Inheritance）建议在设计类的时候，优先考虑使用组合（composition）而不是继承（inheritance）来实现复用。组合意味着一个类中包含另一个类的实例，从而可以在运行时灵活地组合行为，而不是通过继承来直接获得行为。

在苍穹外卖项目中，合成复用原则可以帮助我们更好地设计系统，使其更加灵活和易于维护。以下是一个具体的例子，涉及到订单折扣策略的实现。

初始设计：使用继承

假设我们有一个基类 `Discount`，用于表示各种折扣策略，并通过继承实现不同类型的折扣：

class Discount:

    def apply(self, order\_total):

        raise NotImplementedError("Subclasses should implement this method")

class PercentageDiscount(Discount):

    def \_\_init\_\_(self, percentage):

        self.percentage = percentage

    def apply(self, order\_total):

        return order\_total \* (1 - self.percentage / 100)

class FixedAmountDiscount(Discount):

    def \_\_init\_\_(self, amount):

        self.amount = amount

    def apply(self, order\_total):

        return order\_total - self.amount

这种设计的问题在于，如果我们需要添加更多复杂的折扣策略，例如先应用百分比折扣再应用固定金额折扣，我们可能需要创建新的类并使用多重继承，这会使代码变得复杂且难以维护。

改进设计：使用组合

为了遵循合成复用原则，我们可以通过组合不同的折扣策略类来实现更灵活的设计。首先，定义基本的折扣接口和具体的折扣类：

class DiscountStrategy:

    def apply(self, order\_total):

        raise NotImplementedError("Subclasses should implement this method")

class PercentageDiscount(DiscountStrategy):

    def \_\_init\_\_(self, percentage):

        self.percentage = percentage

    def apply(self, order\_total):

        return order\_total \* (1 - self.percentage / 100)

class FixedAmountDiscount(DiscountStrategy):

    def \_\_init\_\_(self, amount):

        self.amount = amount

    def apply(self, order\_total):

        return order\_total - self.amount

然后，创建一个 `CompositeDiscount` 类，用于组合多个折扣策略：

class CompositeDiscount(DiscountStrategy):

    def \_\_init\_\_(self):

        self.discounts = []

    def add\_discount(self, discount):

        self.discounts.append(discount)

    def apply(self, order\_total):

        for discount in self.discounts:

            order\_total = discount.apply(order\_total)

        return order\_total

现在，我们可以通过组合不同的折扣策略来创建更复杂的折扣逻辑，而无需通过继承来实现：

# 创建订单

order\_total = 100

# 创建折扣策略

percentage\_discount = PercentageDiscount(10)

fixed\_discount = FixedAmountDiscount(5)

# 组合折扣策略

composite\_discount = CompositeDiscount()

composite\_discount.add\_discount(percentage\_discount)

composite\_discount.add\_discount(fixed\_discount)

# 应用组合折扣

final\_total = composite\_discount.apply(order\_total)

print("Total after composite discount:", final\_total)

通过这种方式，我们可以灵活地组合不同的折扣策略，而不需要创建复杂的继承结构。这使得系统更易于扩展和维护。

另一个例子：订单处理中的通知系统

假设在苍穹外卖项目中，订单处理完成后需要发送通知（如短信、邮件等）。初始设计可能使用继承来实现不同类型的通知：

class Notifier:

    def send(self, message):

        raise NotImplementedError("Subclasses should implement this method")

class SMSNotifier(Notifier):

    def send(self, message):

        print(f"Sending SMS: {message}")

class EmailNotifier(Notifier):

    def send(self, message):

        print(f"Sending Email: {message}")

如果需要同时发送多种通知，继承会变得复杂。

改进设计：使用组合

我们可以通过组合来实现通知系统，使得每个通知类型独立，并且可以灵活组合。

class Notifier:

    def send(self, message):

        raise NotImplementedError("Subclasses should implement this method")

class SMSNotifier(Notifier):

    def send(self, message):

        print(f"Sending SMS: {message}")

class EmailNotifier(Notifier):

    def send(self, message):

        print(f"Sending Email: {message}")

class CompositeNotifier(Notifier):

    def \_\_init\_\_(self):

        self.notifiers = []

    def add\_notifier(self, notifier):

        self.notifiers.append(notifier)

    def send(self, message):

        for notifier in self.notifiers:

            notifier.send(message)

通过组合不同的通知策略，我们可以创建一个灵活的通知系统：

# 创建通知策略

sms\_notifier = SMSNotifier()

email\_notifier = EmailNotifier()

# 组合通知策略

composite\_notifier = CompositeNotifier()

composite\_notifier.add\_notifier(sms\_notifier)

composite\_notifier.add\_notifier(email\_notifier)

# 发送通知

composite\_notifier.send("Order has been processed.")

通过使用组合，我们创建了一个更灵活且易于扩展的通知系统。这使得我们可以轻松地添加新的通知方式，而不需要修改现有的代码结构，符合合成复用原则。

结合自己的实践项目举例说明如何应用 （保存到每个小组选定的协作开发平台上，以组为单位）。

项目跟踪，建立能反映项目及小组每个人工作的进度、里程碑、工作量的跟踪图或表，将其保存到每个小组选定的协作开发平台上，每周更新。