1. 单例模式（Singleton）

应用场景：

- **数据库连接池**：确保只有一个数据库连接池实例，以管理数据库连接的共享和复用。
- **配置管理器**：管理系统配置参数，保证配置的全局访问性和唯一性。
- **日志记录器**：统一管理日志记录，确保日志记录器的实例唯一。

具体应用： 在Node.js应用中，可以通过单例模式管理MySQL数据库连接池，确保系统高效地管理数据库连接资源。

2. 工厂方法模式（Factory Method）

应用场景：

- **商品对象的创建**：根据不同商品类型（如电子产品、书籍、服装等），使用工厂方法模式创建具体商品对象。
- **用户对象的创建**：创建不同类型的用户对象（如普通用户、管理员）。

具体应用： 在Node.js后端，可以使用工厂方法模式根据商品类型创建具体的商品对象，避免大量的if-else或switch-case语句，从而提高代码的可读性和可维护性。

3. 观察者模式（Observer）

应用场景：

- **库存管理**：当商品库存发生变化时，通知相关模块（如销售模块、订单模块）。
- **订单状态更新**：当订单状态改变时，通知用户和相关系统（如发货系统、支付系统）。

具体应用： 在商品交易系统中，当商品库存变化时，可以使用观察者模式通知相关模块进行相应处理，确保系统各部分的数据一致性和及时性。

4. 策略模式（Strategy）

应用场景：

- **商品价格计算**：根据不同的定价策略（如折扣策略、促销策略、会员价格策略）计算商品价格。
- **支付方式选择**：根据用户选择或系统推荐，动态选择支付方式。

具体应用： 在商品交易系统中，可以使用策略模式来实现不同的价格计算方式和支付方式选择，提高系统的灵活性和可扩展性。

5. 责任链模式（Chain of Responsibility）

应用场景：

- **订单处理流程**：订单创建、验证、支付、发货等多个处理步骤，通过责任链模式依次处理。
- **请求处理**：用户请求通过一系列处理对象（如认证、授权、日志记录等）进行处理。

具体应用： 在订单处理过程中，可以通过责任链模式将各个处理步骤解耦，提高系统的灵活性和可维护性。例如，订单验证、支付处理、发货处理可以依次通过责任链进行处理。

6. 外观模式（Facade）

应用场景：

- **提供统一接口**：为复杂的子系统（如订单处理系统、支付系统、库存管理系统）提供一个简单的接口，简化客户端的使用。

具体应用： 在React前端，可以通过外观模式为各种API请求提供统一的接口，简化前端的调用逻辑。例如，可以为用户管理、订单处理、商品管理等子系统提供统一的接口，使前端代码更清晰和易于维护。

7. MVC模式（Model-View-Controller）

应用场景：

- **前后端分离**：React负责视图层（View），Node.js负责控制器（Controller）和模型（Model），通过MVC模式清晰分离职责。

具体应用： 在整个系统中，React前端处理用户界面和用户交互，Node.js后端处理业务逻辑和数据操作，MySQL数据库存储数据。这种分层设计提高了代码的可维护性和可扩展性。

8. 数据访问对象模式（DAO，Data Access Object）

应用场景：

- **数据库操作封装**：将数据库操作封装在独立的DAO对象中，提供统一的数据访问接口。

具体应用： 在Node.js应用中，可以通过DAO模式将数据库操作封装在独立的对象中，为数据的增删改查提供统一接口，从而降低业务逻辑与数据访问代码的耦合度。

总结

在开发商品网上交易系统过程中，结合实际需求，灵活应用单例模式、工厂方法模式、观察者模式、策略模式、责任链模式、外观模式、MVC模式和DAO模式等设计模式。这些模式的应用不仅提高了系统的灵活性和可维护性，还增强了系统的可扩展性和性能，使系统能够更好地适应业务需求的变化和用户体验的提升。

1. 单例模式（Singleton）

特点：

- 确保一个类只有一个实例。
- 提供一个全局访问点。
- 控制实例化过程，防止多个实例的创建。

示例（JavaScript）：

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            Singleton.instance = this;
        }
        return Singleton.instance;
    }

    showMessage() {
        console.log("Hello, I am a singleton instance!");
    }
}

// Usage
const instance1 = new Singleton();
const instance2 = new Singleton();

console.log(instance1 === instance2);  // true
instance1.showMessage();  // Hello, I am a singleton instance!
```

2. 工厂方法模式（Factory Method）

特点：

- 定义一个创建对象的接口，但由子类决定实例化的类是哪一个。
- 延迟到子类以便创建对象。

示例（JavaScript）：

```
class Product {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }
}

class ProductFactory {
```

```
    createProduct(type) {
        if (type === 'A') {
            return new Product('Product A');
        } else if (type === 'B') {
            return new Product('Product B');
        }
    }
}

// Usage
const factory = new ProductFactory();
const productA = factory.createProduct('A');
const productB = factory.createProduct('B');

console.log(productA.getName());  // Product A
console.log(productB.getName());  // Product B
```

3. 观察者模式（Observer）

特点：

- 定义对象间的一对多依赖，当一个对象状态改变时，所有依赖的对象都会自动收到通知并更新。
- 松耦合，被观察者和观察者可以独立变化。

示例（JavaScript）：

```
class Subject {
    constructor() {
        this.observers = [];
    }

    addObserver(observer) {
        this.observers.push(observer);
    }

    notifyObservers(message) {
        this.observers.forEach(observer => observer.update(message));
    }
}

class Observer {
    constructor(name) {
        this.name = name;
    }

    update(message) {
        console.log(`${this.name} received message: ${message}`);
    }
}

// Usage
```

```
const subject = new Subject();
const observer1 = new Observer('Observer 1');
const observer2 = new Observer('Observer 2');

subject.addObserver(observer1);
subject.addObserver(observer2);

subject.notifyObservers('Hello Observers!');
// Observer 1 received message: Hello Observers!
// Observer 2 received message: Hello Observers!
```

4. 策略模式（Strategy）

特点：

- 定义一系列算法，把它们一个个封装起来，并且使它们可以互换。
- 使得算法可以独立于使用它的客户而变化。

示例（JavaScript）：

```
class Strategy {
    execute() {}
}

class ConcreteStrategyA extends Strategy {
    execute() {
        console.log("Executing Strategy A");
    }
}

class ConcreteStrategyB extends Strategy {
    execute() {
        console.log("Executing Strategy B");
    }
}

class Context {
    setStrategy(strategy) {
        this.strategy = strategy;
    }

    executeStrategy() {
        this.strategy.execute();
    }
}

// Usage
const context = new Context();
const strategyA = new ConcreteStrategyA();
const strategyB = new ConcreteStrategyB();
```

```
context.setStrategy(strategyA);
context.executeStrategy();  // Executing Strategy A

context.setStrategy(strategyB);
context.executeStrategy();  // Executing Strategy B
```

总结

1. **单例模式（Singleton）**：
   - **优点**：控制实例数量、提供全局访问点。
   - **缺点**：不易扩展、可能造成资源竞争。
2. **工厂方法模式（Factory Method）**：
   - **优点**：解耦对象创建过程、增强代码灵活性。
   - **缺点**：增加类数量、代码复杂度上升。
3. **观察者模式（Observer）**：
   - **优点**：实现对象间的动态联动、增强系统的灵活性和可扩展性。
   - **缺点**：可能导致过多的通知、增加系统复杂性。
4. **策略模式（Strategy）**：
   - **优点**：算法可以自由切换、避免使用多重条件判断。
   - **缺点**：客户端必须了解所有策略以便选择适当的策略、增加对象数量。