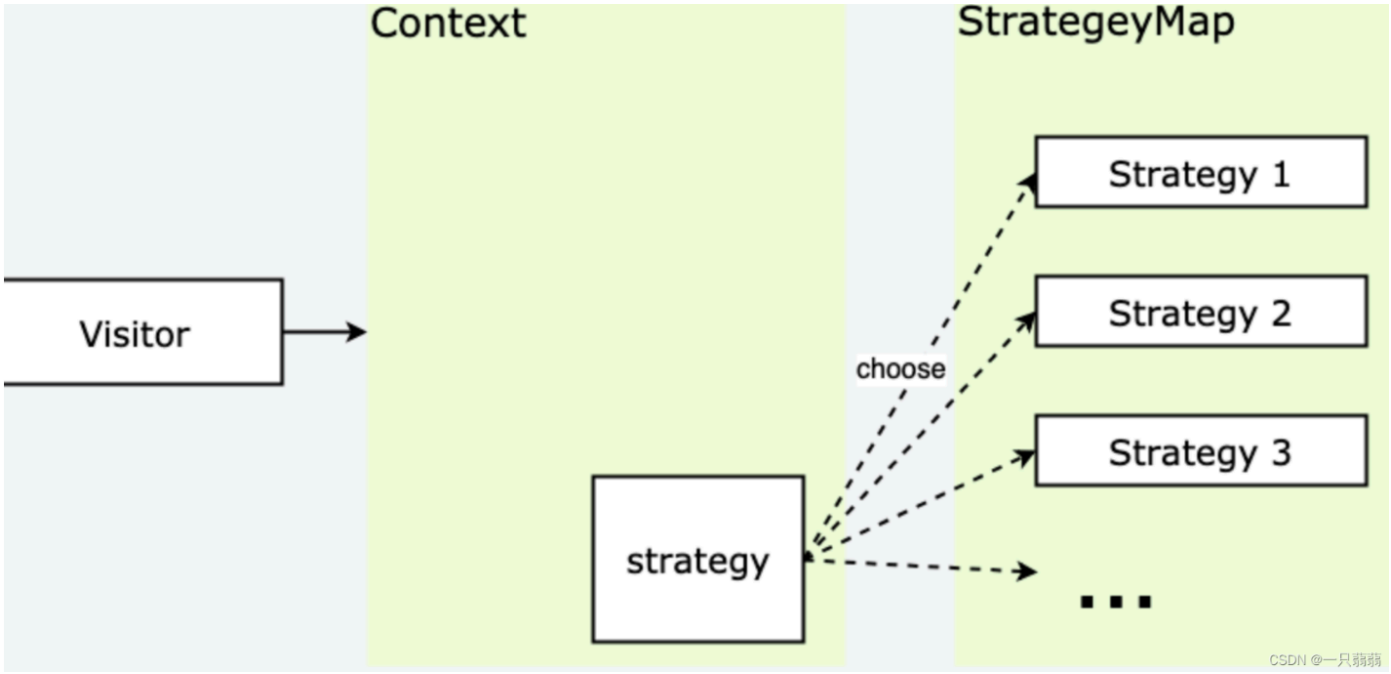
行為型模式 Behavioral Pattern

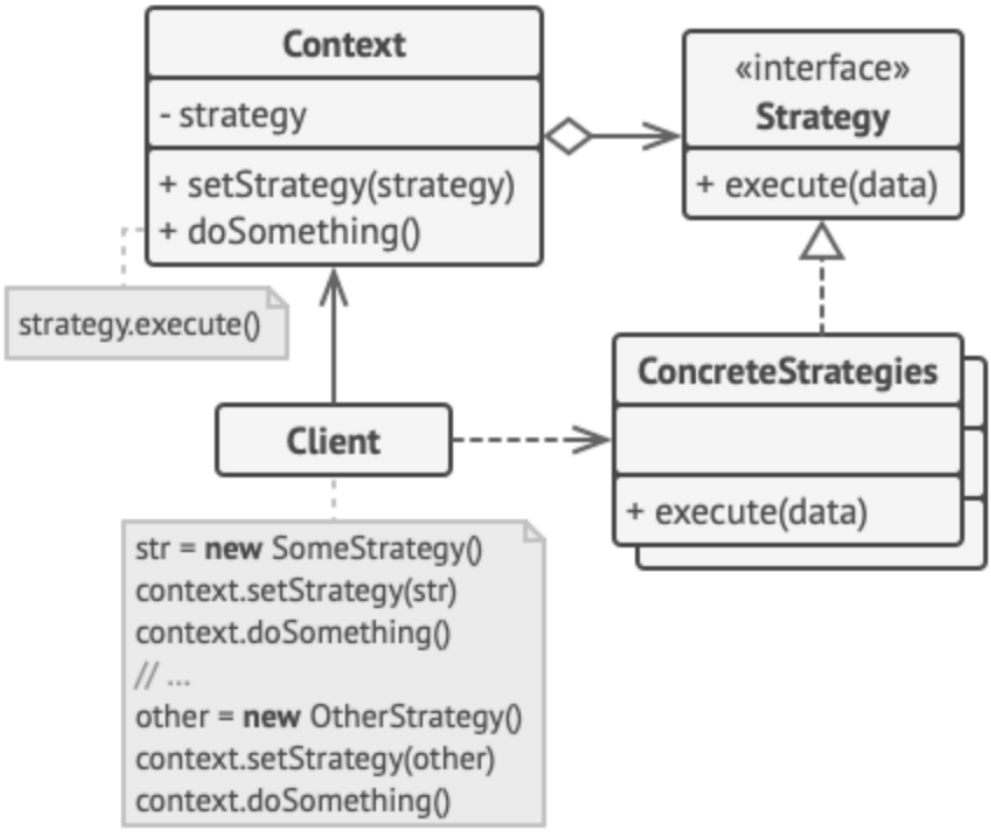
> 封裝變化。

行為模式負則對象之間的溝通與職責畫分，其除了關注結構之外，更關注他門之間的溝通機制。透過行為型模式可以更清楚的劃分類別、物件之間的職責，展現實例物件之間的作用互動。

Strategy （策略模式）

Strategy（策略模式）屬於行為型模式。





- Context：封裝上下文，根據需要調用所需的策略，屏蔽外界對策略的直接調用，只對外提供一個接口，根據需要調用對應的策略。
- Strategy：策略，含有具體的演算法，其方法的外觀相同，因此可以互相替換。
- StrategyMap：所有策略的集合，供封裝上下文呼叫。

意圖

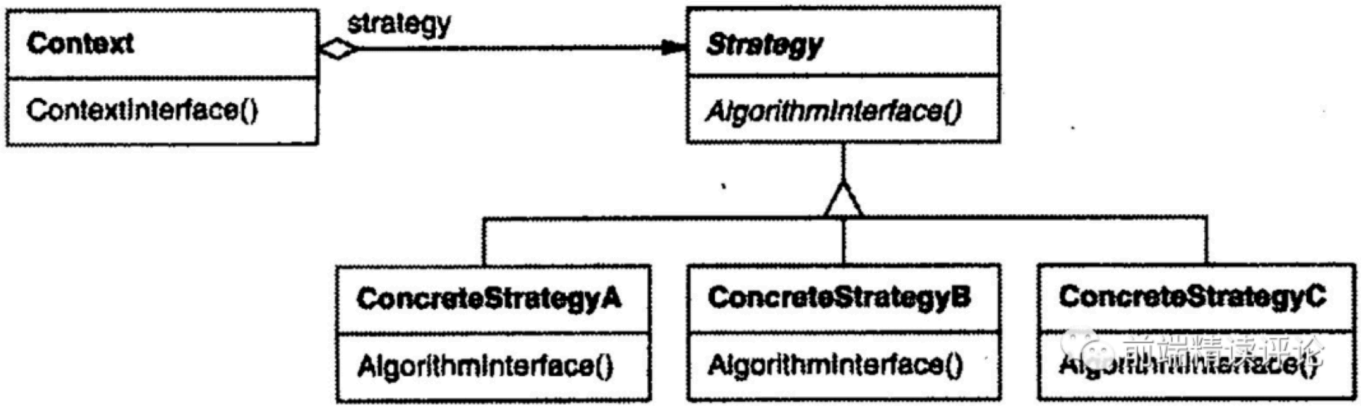
> 定義一系列的演算法，把它們一個個封裝起來，並且使它們可以互相替換。本模式使的演算法可以獨立於使用它的客戶而變化。

策略是形象的表述，所謂策略就是方案，我們都知道任何事情都有多種方案，而且不同方案都能解決問題，所以這些方案可以互相替換。我們將方案從問題中抽象化出來，這樣就可以拋開問題，單獨優化方案了，這就是策略模式的核心思想。

意圖解釋

演算法可以理解為策略，我們制定許多解決某個場景的策略，這些策略都可以獨立的解決這個場景的問題，這樣下次遇到這個場景時，我們就可以選擇任何策略來解決，而且我們還可以脫離場景，單獨優化策略，只要介面不變即可。

這個意圖本質上就是解耦，解耦之後才可以分工。想想一個複雜的系統，如果所有策略都耦合在業務邏輯裡，那麼只有懂業務的人才能小心翼翼的維護，但如果將策略與業務解耦，我們可以獨立維護這些策略，為業務帶來更靈活的變化。



- Strategy：策略公共介面。
- ConcreteStrategy：具體策略，實作了上面這個介面。

範例：

```
// 例1：
interface Strategy {
  doSomething: () => void;
}

class Strategy1 implements Strategy {
  doSomething: () => {
    console.log('實現策略1');
  }
}

class Strategy2 implements Strategt {
  doSomethingL () => {
    console.log('實現策略2')
  }
}

new System(new Strategt1());
new System(new Strategy2());
```

```
// 例2：不同角色使用不同攻擊方式
// interfact & enum
export enum Role {
  Swordsman = 'Swordsman', // 劍士
  Warlock = 'Warlock', // 術士
  Highwayman = 'Highwayman', // 攔路強盜
  BountyHunter = 'BountyHunter', // 賞金獵人
}

export interface ICharacterBase {
  name: string;
```

```
  role: Role;
}

export interface IAttack {
  attack(self: ICharacter, target: ICharacter): void;
  // attack: (self: ICharacter, target: ICharacter) => void;
}

export type ICharacter = ICharacterBase & IAttack;

// class Character
import { IAttack, ICharacter, Role } from '../interface/character.interface';

export class Character implements ICharacter {
  private _name: string;
  private _role: Role;
  private _attackRef: IAttack;

  get name(): string {
    return this._name;
  }

  get role(): Role {
    return this._role;
  }

  get attackRef(): IAttack {
    return this._attackRef;
  }

  constructor(name: string, role: Role, attack: IAttack) {
    this._name = name;
    this._role = role;
    this._attackRef = attack;
  }

  attack(self: ICharacter, target: ICharacter): void {
    this._attackRef.attack(self, target);
  }
}

// class MagicAttack
import { IAttack, ICharacter } from '../interface/character.interface';

export class MagicAttack implements IAttack {
  constructor() {}

  attack(self: ICharacter, target: ICharacter): void {
    console.log(
```

```
      `${self.role}-${self.name} attacking the ${target.role}-${target.name}`
    );
  }
}

// class MeleeAttack
import { IAttack, ICharacter } from '../interface/character.interface';

export class MeleeAttack implements IAttack {
  constructor() {}

  attack(self: ICharacter, target: ICharacter): void {
    console.log(
      `${self.role}-${self.name} attacking the ${target.role}-${target.name}`
    );
  }
}

// app.component
export class AppComponent {
  constructor() {}

  ngOnInit(): void {
    const swordsman: ICharacter = new Character(
      '我是劍士',
      Role.Swordsman,
      new MeleeAttack()
    );
    const warlcok: ICharacter = new Character(
      '我是術士',
      Role.Warlock,
      new MagicAttack()
    );

    swordsman.attack(swordsman, warlcok);
    // Swordsman-我是劍士 attacking the Warlock-我是術士
    warlcok.attack(warlcok, swordsman);
    // Warlock-我是術士 attacking the Swordsman-我是劍士
  }
}
```

```
// 例3：不同支付方式
// interface & enum
export interface IPaymentStrategy {
  pay(amount: number): void;
}

/**
```

```
 * @type CREDITCARD = 1
 * @type LINEPAY = 2
 * @type CASH = 3
 */
export enum PaymentType {
  CREDITCARD = 1,
  LINEPAY = 2,
  CASH = 3,
}

// class
export class CreditCardStrategy implements IPaymentStrategy {
  constructor() {}

  pay(amount: number): void {
    console.log(`use CreditCardStrategy to pay ${amount}`);
  }
}

export class LinePayStrategy implements IPaymentStrategy {
  constructor() {}

  pay(amount: number): void {
    console.log(`use LinePayStrategy to pay ${amount}`);
  }
}

export class CashStrategy implements IPaymentStrategy {
  constructor() {}

  pay(amount: number): void {
    console.log(`use CashStrategy to pay ${amount}`);
  }
}

// service
@Injectable({
  providedIn: 'root',
})
export class PaymentService {
  private _paymentStrategy: IPaymentStrategy | null = null;

  setPaymentStrategy(paymentStrategy: IPaymentStrategy): void {
    this._paymentStrategy = paymentStrategy;
  }

  constructor() {}

  processPayment(amount: number): void {
```

```
    if (!this._paymentStrategy) {
      console.error('Payment strategy is not set.');
      return;
    }
    this._paymentStrategy.pay(amount);
  }
}

// app.component.html
<h1>Payment Strategy</h1>

<section>
  <div class="amount-wrap">
    <label for="amount">Amount: </label>
    <input type="number" [(ngModel)]="amount" />
  </div>

  <div class="btn-wrap">
    <button (click)="payment(1)">Pay with Credit Card</button>
    <button (click)="payment(2)">Pay with Line</button>
    <button (click)="payment(3)">Pay with Cash</button>
  </div>
</section>

// app.component.ts
export class AppComponent {
  amount: number = 0;

  constructor(private paymentService: PaymentService) {}

  ngOnInit(): void {}

  payment(paymentType: PaymentType): void {
    let paymentStrategy: IPaymentStrategy;
    switch (paymentType) {
      case 2:
        paymentStrategy = new LinePayStrategy();

        break;
      case 3:
        paymentStrategy = new CashStrategy();
        break;
      default:
        paymentStrategy = new CreditCardStrategy();
        break;
    }
    this.paymentService.setpaymentStrategy(paymentStrategy);
    this.paymentService.processPayment(this.amount);
  }
```

```
}
```

```
// 例4：不同支付加上不同幣別
// interface & enum
export interface IPaymentStrategy {
  pay(currencyType: CurrencyType, amount: number): void;
}

export interface ICurrencyStrategy {
  convert(amount: number): number;
}

/**
 * @type CREDITCARD = 1
 * @type LINEPAY = 2
 * @type CASH = 3
 */
export enum PaymentType {
  CREDITCARD = 1,
  LINEPAY = 2,
  CASH = 3,
}

export enum CurrencyType {
  TWD = 'TWD',
  USD = 'USD',
  JPY = 'JPY',
  CNY = 'CNY',
}

export enum ExchangeRate {
  TWD = 1,
  USD = 31.4,
  JPY = 0.21,
  CNY = 4.6,
}

// class
export class USDStrategy implements ICurrencyStrategy {
  constructor() {}

  convert(amount: number): number {
    return +(amount * ExchangeRate.USD).toFixed(1);
  }
}

export class TWDStrategy implements ICurrencyStrategy {
  constructor() {}
```

```
  convert(amount: number): number {
    return amount * ExchangeRate.TWD;
  }
}

export class JPYStrategy implements ICurrencyStrategy {
  constructor() {}

  convert(amount: number): number {
    return Math.ceil(amount * ExchangeRate.JPY);
  }
}

export class CNYStrategy implements ICurrencyStrategy {
  constructor() {}

  convert(amount: number): number {
    return +(amount * ExchangeRate.CNY).toFixed(1);
  }
}

export class CreditCardStrategy implements IPaymentStrategy {
  constructor() {}

  pay(currencyType: CurrencyType, amount: number): void {
    console.log(`use CreditCardStrategy pay ${currencyType} ${amount};`);
  }
}

export class LinePayStrategy implements IPaymentStrategy {
  constructor() {}

  pay(currencyType: CurrencyType, amount: number): void {
    console.log(`use LinePayStrategy pay ${currencyType} ${amount};`);
  }
}

export class CashStrategy implements IPaymentStrategy {
  constructor() {}

  pay(currencyType: CurrencyType, amount: number): void {
    console.log(`use CashStrategy pay ${currencyType} ${amount};`);
  }
}

// service
import { Injectable } from '@angular/core';
import {
  CurrencyType,
```

```
  ICurrencyStrategy,
  IPaymentStrategy,
} from '../interface/payment-strategy.interface';

@Injectable({
  providedIn: 'root',
})
export class PaymentService {
  private _paymentStrategy: IPaymentStrategy | null = null;
  private _currencyStrategy: ICurrencyStrategy | null = null;

  setPaymentStrategy(paymentStrategy: IPaymentStrategy): void {
    this._paymentStrategy = paymentStrategy;
  }

  setCurrencyStrategy(currencyStrategy: ICurrencyStrategy): void {
    this._currencyStrategy = currencyStrategy;
  }

  constructor() {}

  processPayment(currencyType: CurrencyType, amount: number): void {
    if (!this._paymentStrategy || !this._currencyStrategy) {
      console.log('Payment strategy or currency strategy is not set.');
      return;
    }
    this._paymentStrategy.pay(
      currencyType,
      this._currencyStrategy.convert(amount)
    );
  }
}

@Injectable({
  providedIn: 'root',
})
export class CurrencyStrategyFactory {
  createCurrencyStrategy(
    currencyType: CurrencyType = CurrencyType.TWD
  ): ICurrencyStrategy {
    const currencyStrategyFactoryMap: {
      [key in CurrencyType]: ICurrencyStrategy;
    } = {
      [CurrencyType.TWD]: new TWDStrategy(),
      [CurrencyType.USD]: new USDStrategy(),
      [CurrencyType.JPY]: new JPYStrategy(),
      [CurrencyType.CNY]: new CNYStrategy(),
    };
    return currencyStrategyFactoryMap[currencyType];
```

```
  }
}

@Injectable({
  providedIn: 'root'
})
export class PaymentStrategyFactory {
  createPaymentStrategy(paymentType: PaymentType = PaymentType.CREDITCARD):
IPaymentStrategy {
    const createPaymentStrategyFactoryMap: { [key in PaymentType]: IPaymentStrategy } =
{
      [PaymentType.CREDITCARD]: new CreditCardStrategy(),
      [PaymentType.LINEPAY]: new LinePayStrategy(),
      [PaymentType.CASH]: new CashStrategy()
    }
    return createPaymentStrategyFactoryMap[paymentType];
  }
}

// app.component.html
<section>
  <div>
    <div>
      <label for="currency">Currency: </label>
      <select name="currency" id="currency" [(ngModel)]="selectedCurrencyType">
        <option *ngFor="let currency of currencyList" [value]="currency">
          {{ currency }}
        </option>
      </select>
    </div>
    <div>
      <label for="currency">Amount: </label>
      <input type="number" [(ngModel)]="amount" />
    </div>
  </div>
  <div>
    <button type="button" (click)="payment(1)">CreditCard</button>
    <button type="button" (click)="payment(2)">Line Pay</button>
    <button type="button" (click)="payment(3)">Cash</button>
  </div>
</section>

// app.component.ts
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
```

```
  currencyList: CurrencyType[] = [
    CurrencyType.TWD,
    CurrencyType.USD,
    CurrencyType.JPY,
    CurrencyType.CNY,
  ];
  selectedCurrencyType: CurrencyType = this.currencyList[0];
  amount: number = 0;

  constructor(
    private paymentService: PaymentService,
    private currencyStrategyFactory: CurrencyStrategyFactory,
    private paymentStrategyFactory: PaymentStrategyFactory
  ) {}

  ngOnInit(): void {}

  payment(paymentType: PaymentType): void {
    this.paymentService.setCurrencyStrategy(
      this.currencyStrategyFactory.createCurrencyStrategy(
        this.selectedCurrencyType
      )
    );
    this.paymentService.setPaymentStrategy(
      this.paymentStrategyFactory.createPaymentStrategy(paymentType)
    );
    this.paymentService.processPayment(this.selectedCurrencyType, this.amount);
  }
}
```

弊端

不要走極端，不要每個分支有一個策略模式，這樣會導致策略類過多，當分支邏輯簡單且清晰好維護時，不需要使用策略模式抽象化。

總結

策略模式是很重要的抽象思維，我們首先要意識到問題有許多種解法，才能意識到策略模式的存在。當一個問題需要採取不同策略，且策略相對較複雜，且未來可能要拓展新策略時，可以考慮使用策略模式。