6 Расширенные возможности применения функциональных типов данных

В этой главе

- Использование упрощенного паттерна проектирования «Декоратор».
- Реализация возобновляемого счетчика.
- Обработка длительных операций.
- Написание понятного асинхронного кода с помощью промисов и конструкции `async/await`.

В главе 5 мы рассмотрели основы функциональных типов данных и сценарии, ставшие возможными благодаря работе с функциями подобно любым другим значениям, то есть передаче их в качестве аргументов и возврате в виде результатов. Мы также рассмотрели несколько весьма многообещающих абстракций, реализующих распространенные паттерны обработки данных: `map()`, `filter()` и `reduce()`.

В этой главе мы продолжим обсуждение функциональных типов данных и их более продвинутых приложений. Начнем с паттерна проектирования «Декоратор» и его реализаций — традиционной и альтернативной. (Повторю: не волнуйтесь, если подзабыли его; я напомню, что к чему.) Вы познакомитесь с понятием *«замыкание»* (closure) и узнаете, как с его помощью реализовать простой счетчик. Затем рассмотрим другой способ реализации счетчика, на этот раз задействовав генератор — функцию, выдающую несколько результатов.

Далее мы поговорим об асинхронных операциях. Рассмотрим две основные модели асинхронного выполнения кода: потоки выполнения и циклы ожидания событий — и узнаем, как распланировать выполнение нескольких длительных операций. Мы начнем с функций обратного вызова, затем рассмотрим промисы и, наконец, поговорим о синтаксисе `async/await`, доступном сегодня в большинстве основных языков программирования.

Как мы увидим на последующих страницах, все обсуждаемые в этой главе темы возможны лишь благодаря использованию функций в качестве значений.

6.1. Простой паттерн проектирования «Декоратор»

«Декоратор» — это поведенческий паттерн проектирования программного обеспечения, который расширяет поведение объекта, не прибегая к модификации соответствующего класса. Декорированный объект способен на выполнение задач, выходящих за рамки возможностей его исходной реализации. Схема этого паттерна приведена на рис. 6.1.

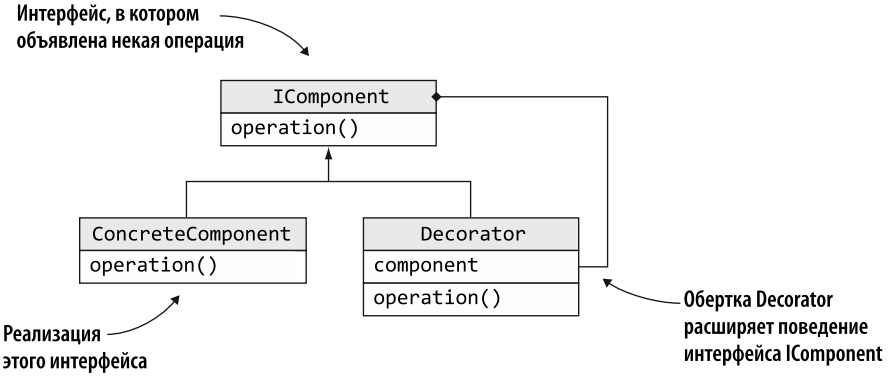

Рис. 6.1. Паттерн «Декоратор»: интерфейс IComponent, его конкретная реализация ConcreteComponent и Decorator, расширяющий IComponent дополнительным поведением

Для примера представим, что у нас есть интерфейс `IWidgetFactory`, в котором объявлен метод `Widget()`, возвращающий объект `Widget`. А в конкретной реализации `WidgetFactory` реализован метод для создания новых объектов `Widget`.

Допустим, что мы хотим повторно использовать `Widget` и вместо того, чтобы создавать каждый раз новый объект, хотели бы создать только один объект класса и всегда возвращать его (то есть реализовать одиночку). Не внося изменений в класс `WidgetFactory`, мы можем создать декоратор `SingletonDecorator` — обертку для `IWidgetFactory`, как показано в листинге 6.1, и расширить его поведение так, чтобы создавался лишь один объект `Widget` (рис. 6.2).

Листинг 6.1. Декоратор для IWidgetFactory

```
class Widget {}

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    public makeWidget(): Widget {
        return new Widget();
    }
}

class SingletonDecorator implements IWidgetFactory {
    private factory: IWidgetFactory;
    private instance: Widget | undefined = undefined;

    constructor(factory: IWidgetFactory) {
        this.factory = factory;
    }

    public makeWidget(): Widget {
        if (this.instance == undefined) {
            this.instance = this.factory.makeWidget();
        }

        return this.instance;
    }
}
```

WidgetFactory просто создает новый объект Widget

SingletonDecorator обертывает IWidgetFactory

Метод makeWidget реализует логику одиночки и гарантирует, что может быть создан только один экземпляр Widget

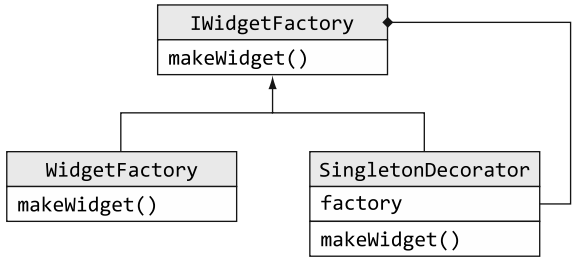

Рис. 6.2. Паттерн «Декоратор» для фабрики виджетов. IWidgetFactory — интерфейс, WidgetFactory — конкретная реализация, а класс SingletonDecorator добавляет в IWidgetFactory поведение одиночки

Преимущество этого паттерна заключается в поддержке *принципа единственной обязанности* (single-responsibility principle), который гласит: класс должен отвечать только за что-то одно. В данном случае класс `WidgetFactory` отвечает за создание виджетов, а `SingletonDecorator` — за поведение, соответствующее одиночке. Если нам потребуется несколько экземпляров класса, то можно воспользоваться непосредственно классом `WidgetFactory`. Если же один — классом `SingletonDecorator`.

6.1.1. Функциональный декоратор

Попробуем упростить эту реализацию опять-таки с помощью типизированных функций. Для начала избавимся от интерфейса `IWidgetFactory`, заменив его функциональным типом данных, описывающим функцию без аргументов, которая возвращает объект `Widget`: `() => Widget`.

Теперь мы можем заменить класс `WidgetFactory` простой функцией `makeWidget()`. Там, где раньше использовался интерфейс `IWidgetFactory` и передавался экземпляр `WidgetFactory`, теперь мы потребуем функции типа `() => Widget` и будем передавать туда `makeWidget()`, как показано в листинге 6.2.

Листинг 6.2. Функциональная фабрика виджетов

```
class Widget {}

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(makeWidget);
```

Функциональный тип данных для фабрики виджетов

Тип функции makeWidget() соответствует типу WidgetFactory

Функция use10Widgets требует наличия параметра типа WidgetFactory и использует его для создания десяти экземпляров Widget

Пример вызова: передаем функцию makeWidget в качестве аргумента

Для создания функциональной фабрики виджетов мы используем методику, очень близкую к паттерну проектирования «Стратегия» из главы 5: передаем функцию в качестве аргумента и вызываем ее при необходимости. Теперь посмотрим, как добавить сюда поведение одиночки.

Создаем новую функцию, `singletonDecorator()`, принимающую в качестве аргумента функцию типа `WidgetFactory` и возвращающую другую функцию типа `WidgetFactory`. Как вы помните из главы 5, лямбда-выражение — это функция без названия, которую можно возвращать из другой функции. В листинге 6.3 наш декоратор получает фабрику и с ее помощью создает новую функцию, отвечающую за поведение одиночки (рис. 6.3).

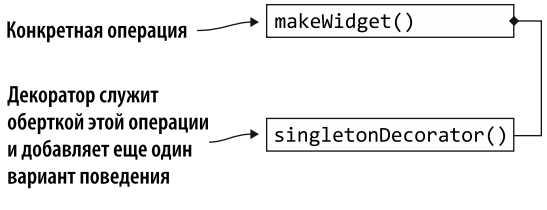

Рис. 6.3. Функциональный декоратор: теперь достаточно функций makeWidget() и singletonDecorator()

Листинг 6.3. Декоратор для функциональной фабрики виджетов

```
class Widget {}

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }
        return instance;
    };
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(singletonDecorator(makeWidget));
```

Функция singletonDecorator() возвращает лямбда-выражение, реализующее поведение одиночки, используя заданную фабрику для создания объекта Widget

А поскольку функция singletonDecorator() возвращает WidgetFactory, ее можно передать в качестве аргумента функции use10Widgets()

Теперь вместо создания десяти объектов `Widget` функция `use10Widgets()` вызывает лямбда-выражение, повторно использующее один и тот же объект `Widget` для всех вызовов.

В этом коде количество компонентов уменьшается с интерфейса и двух классов — по одному методу каждый (конкретная операция и декоратор) — до двух функций.

6.1.2. Реализации декоратора

Как и в случае нашего паттерна «Стратегия», объектно-ориентированный и функциональный подход реализуют один и тот же паттерн проектирования «Декоратор». Объектно-ориентированная версия требует объявления интерфейса (`IWidgetFactory`), по крайней мере одной реализации этого интерфейса (`WidgetFactory`) и класса-декоратора, отвечающего за дополнительный вариант поведения (`SingletonDecorator`). При функциональной реализации же, напротив, просто объявляется тип для фабричной функции (`() => Widget`) и используются две функции: функция-фабрика (`makeWidget()`) и функция-декоратор (`singletonDecorator()`).

Стоит отметить, что в функциональном случае тип декоратора отличается от типа `makeWidget()`. У фабрики аргументов нет, она возвращает `Widget`, а декоратор принимает на входе фабрику виджетов и возвращает другую. Говоря иначе,

`singletonDecorator()` принимает в качестве аргумента функцию и возвращает ее в качестве результата. Это возможно только благодаря полноправности функций, то есть возможности работать с функциями точно так же, как и с прочими переменными, и использовать их в качестве аргументов и возвращаемых значений.

Эта более компактная реализация, ставшая доступной благодаря современным системам типов, вполне подходит для многих случаев. Более «многословное» объектно-ориентированное решение подходит для работы с несколькими функциями. Если в нашем интерфейсе объявлено несколько методов, то заменить их одним функциональным типом данных не получится.

6.1.3. Замыкания

Посмотрим более внимательно на реализацию `singletonDecorator()` в листинге 6.4. Возможно, вы обратили внимание на интересный нюанс: хоть функция возвращает лямбда-выражение, оно ссылается как на аргумент `factory`, так и на, казалось бы, локальную (по отношению к функции `singletonDecorator()`) переменную `instance`.

Листинг 6.4. Функция-декоратор

```
function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }

        return instance;
    };
}
```

И даже после возврата из функции `singletonDecorator()` переменная `instance` все равно существует, поскольку была «захвачена» лямбда-выражением. Это явление называется *лямбда-захватом* (lambda capture).

ЗАМЫКАНИЯ И ЛЯМБДА-ЗАХВАТЫ

> Лямбда-захват представляет собой захват внешней переменной внутри лямбда-выражения. Такие захваты реализуются в языках программирования с помощью замыканий. Замыкание — это не просто функция: оно также фиксирует среду, в которой функция была создана, так что может сохранять состояние от вызова до вызова.

В нашем случае переменная `instance` в функции `singletonDecorator()` является частью такой среды, поэтому возвращенное лямбда-выражение по-прежнему сможет ссылаться на `instance` (рис. 6.4).

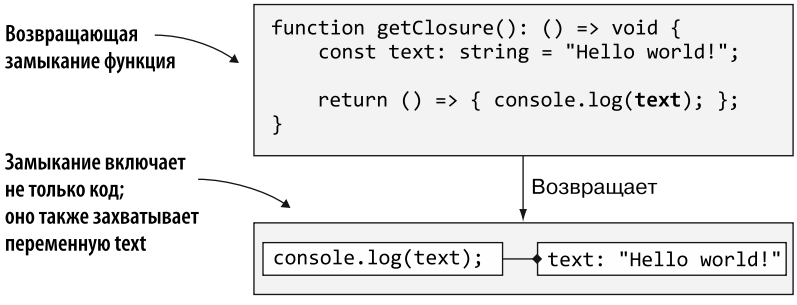

Рис. 6.4. Простая функция, возвращающая замыкание: лямбда-выражение, которое ссылается на локальную (по отношению к этой функции) переменную. Даже после возврата из функции getClosure() замыкание все равно ссылается на переменную, так что она существует дольше, чем функция, в которой появляется

Замыкания имеют смысл только при наличии функций высшего порядка. Если нельзя вернуть из одной функции другую, то нет и среды, которую можно было бы захватить. В этом случае все функции находятся в глобальной области видимости, которая и играет роль их среды. Они могут ссылаться на глобальные переменные.

Можно также сравнить замыкания с объектами. Объект — некое состояние с набором методов; *замыкание* — функция с неким захваченным состоянием. Рассмотрим еще один пример, в котором нам пригодятся замыкания, — реализацию счетчика.

6.1.4. Упражнение

Реализуйте функцию `loggingDecorator()`, принимающую в качестве аргумента другую функцию, `factory()`, которая не принимает аргументов и возвращает объект `Widget`. Декоратор должен вывести в консоль `"Widget created"`, прежде чем с помощью вызова заданной (переданной ему) функции вернуть объект `Widget`.

6.2. Реализация счетчика

Рассмотрим очень простой сценарий: создание счетчика, возвращающего последовательные числа, начиная с 1. Этот пример может показаться тривиальным, однако охватывает несколько возможных реализаций, которые можно применять любому сценарию генерации значений. Один из этих вариантов реализации — воспользоваться глобальной переменной и функцией, которая возвращает ее, после чего увеличивает ее значение на 1, как показано в листинге 6.5.

Данная реализация работает, но она не оптимальна. Во-первых, `n` — глобальная переменная, так что доступ к ней есть у кого угодно. Другой код может изменить ее значение незаметно для нас. Во-вторых, это реализация одного счетчика. А что, если нам понадобятся два счетчика, начинающихся с 1?