

# 设计模式优化与重构: Diffusers

袁晨圃

2025-12-19

目录

一、 缺陷诊断

1.1 利用大模型发现问题

二、 重构采用的设计模式介绍

2.1 建造者模式 (Builder Pattern)

2.2 策略模式 (Strategy Pattern)

三、 重构过程以及效果

3.1 重构过程

3.2 效果展示

1

2

6

7

10

15

16

19

# 1.1 利用大模型发现问题

与 Deep-Wiki 多轮  
对话，  
整理成重构文档

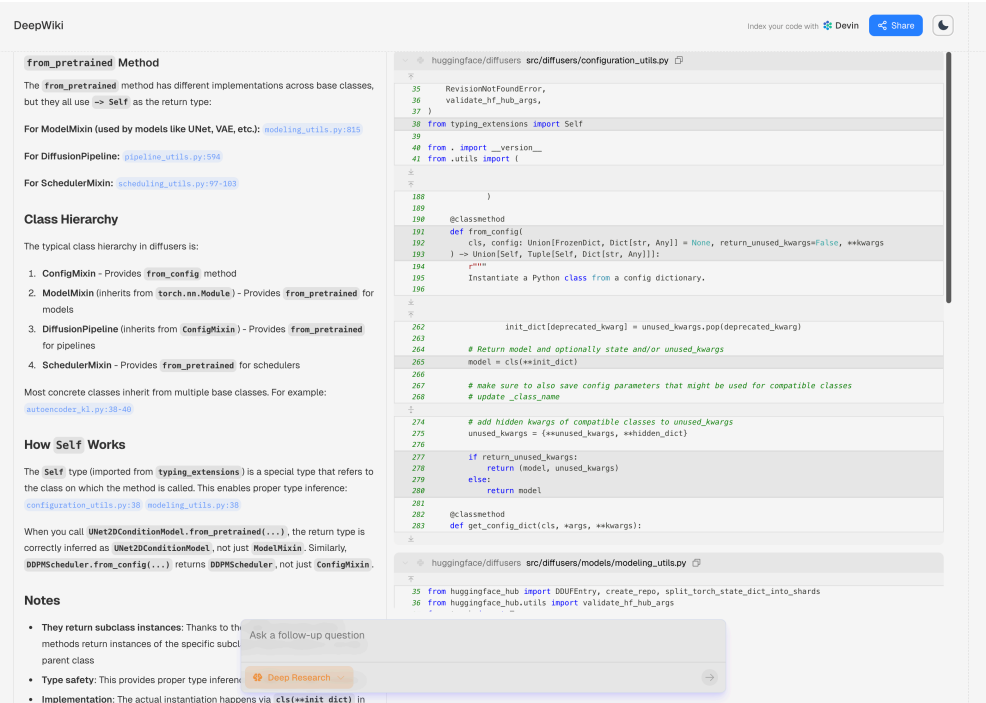


图 1 对话截图

## 1.1 利用大模型发现问题

文档节选：

## 现有设计模式识别

### 1. 注册表模式 (Registry Pattern)

`\_AttentionBackendRegistry`已经实现了注册表模式 diffusers:206-248 :

```
```python
class _AttentionBackendRegistry:
    _backends = {}
    _constraints = {}

    @classmethod
    def register(cls, backend, constraints=None, supports_context_parallel=False):
        ...
```
```

### 2. 策略模式 (Strategy Pattern)

不同的attention后端实现 (FlashAttention、xFormers、Native等) 是不同的算法策略 diffusers:1323-1369 。

### 3. 分发器模式 (Dispatcher Pattern)

`dispatch\_attention\_fn()`作为分发器，根据backend选择具体实现 diffusers:305-351 。

## ## 重构计划

### ### 1. 抽象策略接口

```
```python
from abc import ABC, abstractmethod

class AttentionStrategy(ABC):
    @abstractmethod
    def compute_attention(self, query, key, value, **kwargs): ...
    @abstractmethod
    def validate_constraints(self, **kwargs): ...

class FlashAttentionStrategy(AttentionStrategy):
    def compute_attention(self, query, key, value, **kwargs):
        ...

    def validate_constraints(self, **kwargs):
        ...
```
```

### ### 2. 策略工厂

```
```python
class AttentionStrategyFactory:
    _strategies = {}
```

```
@classmethod
def register_strategy(cls, backend_name: str, strategy_class: type):
    cls._strategies[backend_name] = strategy_class
```

```
@classmethod
def create_strategy(cls, backend_name: str) -> AttentionStrategy:
    if backend_name not in cls._strategies:
        raise ValueError(f"Unknown backend: {backend_name}")
    return cls._strategies[backend_name]()
```

# 注册策略

```
AttentionStrategyFactory.register_strategy("flash", FlashAttentionStrategy)
AttentionStrategyFactory.register_strategy("xformers", XFormersAttentionStrategy)
...
```

目录

一、 缺陷诊断

1.1 利用大模型发现问题

二、 重构采用的设计模式介绍

2.1 建造者模式 (Builder Pattern)

2.2 策略模式 (Strategy Pattern)

三、 重构过程以及效果

3.1 重构过程

3.2 效果展示

1

2

6

7

10

15

16

19

## 2.1 建造者模式 (Builder Pattern)

### ① 建造者模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

#### 2.1.1 引入

假设有这样一个复杂对象，在对其进行构造时需要构造众多成员变量和嵌套对象。这些初始化代码通常深藏于一个包含众多参数的构造函数中，且散落在客户端代码的多个位置。

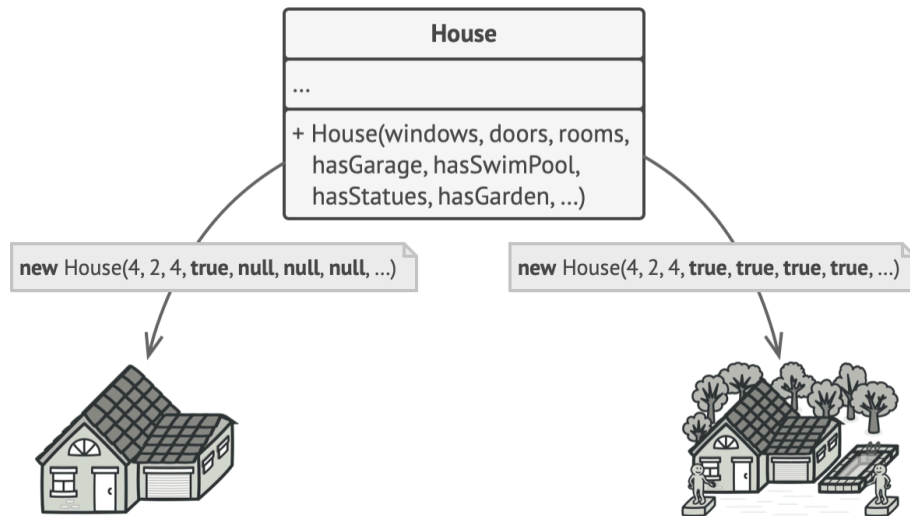


图 2 一个有复杂构造函数的 House 类



## 2.1 建造者模式 (Builder Pattern)

### 建造者模式的解决方案

将对象构造代码从产品类中抽取出来，并将其放在一个名为 **建造者** 的独立对象中。

将对象构造过程划分为一组步骤，比如 `buildWalls` 创建墙壁和 `buildDoor` 创建房门等。每次创建对象时，都需要通过建造者对象执行一系列步骤。重点在于无需调用所有步骤，而只需调用创建特定对象配置所需的那些步骤即可。

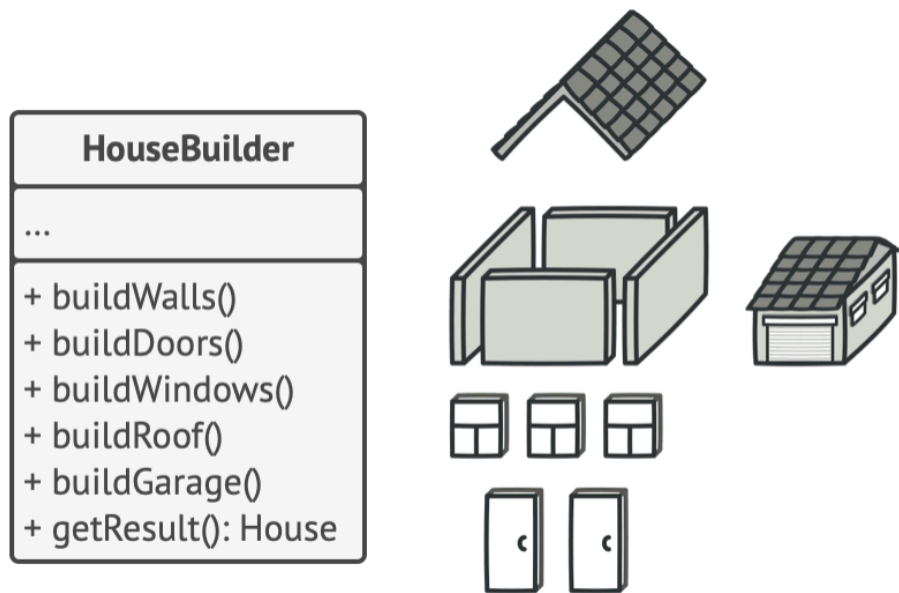


图 3 建造者：HouseBuilder

## 2.1 建造者模式 (Builder Pattern)

### 2.1.2 在代码中的应用

统一 DiffusionPipeline 各组件的构建过程，解决训练脚本中的代码重复和不一致问题。

核心类：DiffusionPipelineBuilder，提供链式配置和组件管理

DiffusionPipelineBuilder 提供一些方法：

- from\_pretrained(), add\_component(), with\_vae(), with\_text\_encoder() 等用于灵活配置和构建不同的扩散管道。
- build() 方法根据配置组装并返回最终的 DiffusionPipeline 实例或者组件 dict。

## 2.2 策略模式 (Strategy Pattern)

### 2.2.1 模式介绍

#### ① 策略模式

定义一系列算法，将每个算法封装起来，并使它们可以互换。策略模式让算法独立于使用它的客户而变化。

- 完成一项任务，往往可以有多种不同的方式，每一种方式称为一个策略，我们可以根据环境或者条件的不同选择不同的策略来完成该项任务。

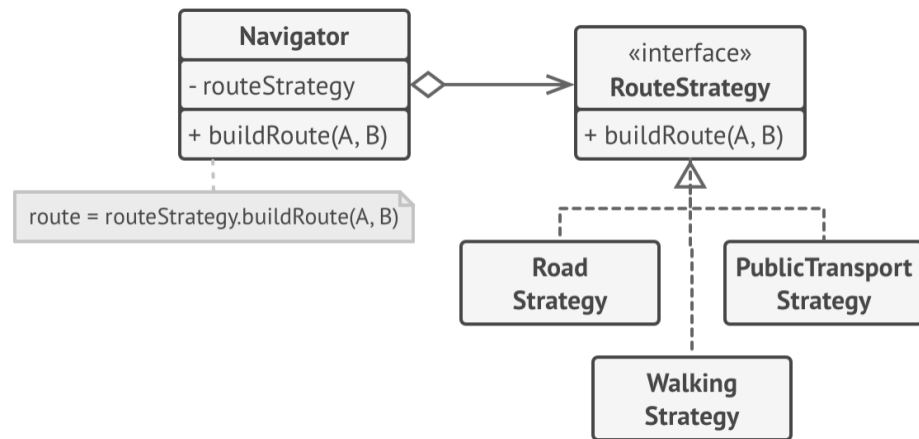


图 4 一些路径规划策略

## 2.2 策略模式 (Strategy Pattern)

策略模式建议找出负责用许多不同方式完成特定任务的类，然后将其中的算法抽取到一组被称为策略的独立类中。

名为上下文的原始类必须包含一个成员变量来存储对于每种策略的引用。上下文并不执行任务，而是将工作委派给已连接的策略对象。

上下文不负责选择符合任务需要的算法——客户端会将所需策略传递给上下文。实际上，上下文并不十分了解策略，它会通过同样的通用接口与所有策略进行交互，而该接口只需暴露一个方法来触发所选策略中封装的算法即可。

因此，上下文可独立于具体策略。这样你就可在不修改上下文代码或其他策略的情况下添加新算法或修改已有算法了。

## 2.2 策略模式 (Strategy Pattern)

### 2.2.2 重构中的应用

#### 2.2.2.1 问题背景

Diffusers 库支持多种 attention 后端（如 FlashAttention、xFormers、PyTorch 原生等），用于优化不同硬件上的性能。但原始实现存在一些问题：

- 扩展困难：新增后端需修改多处代码（如枚举、注册、检查函数）。
- 维护复杂：函数式实现难以测试和调试。
- 类型不安全：缺乏抽象接口，易出错。

## 2.2 策略模式 (Strategy Pattern)

目前原有的实现是基于注册表模式管理后端

```
1 @_AttentionBackendRegistry.register(AttentionBackendName.FLASH)
2 def _flash_attention(query, key, value, **kwargs):
3     return flash_attn_func(q=query, k=key, v=value, **kwargs)
```

python

这个 `_AttentionBackendRegistry.register` 装饰器会在全局的注册表中将后端名称映射到对应的函数。

## 2.2 策略模式 (Strategy Pattern)

引入抽象策略接口，将函数式实现转换为类结构

- 抽象策略接口：AttentionStrategy 基类
- 具体策略类：FlashAttentionStrategy、XFormersAttentionStrategy 等，封装各自的实现细节
- 工厂模式：AttentionStrategyFactory 根据名称实例化对应策略类
- 约束检查：共同的检查移到基类

```
1 class AttentionStrategy(ABC):  
2     @abstractmethod  
3     def compute_attention(self, query, key, value, **kwargs):  
4         pass  
5  
6 class FlashAttentionStrategy(AttentionStrategy):  
7     def compute_attention(self, query, key, value, **kwargs):  
8         return flash_attn_func(q=query, k=key, v=value, **kwargs)
```

python

目录

一、 缺陷诊断

1.1 利用大模型发现问题

二、 重构采用的设计模式介绍

2.1 建造者模式 (Builder Pattern)

2.2 策略模式 (Strategy Pattern)

三、 重构过程以及效果

3.1 重构过程

3.2 效果展示

1

2

6

7

10

15

16

19



## 3.1 重构过程

### 3.1.1 构造单元测试

测试驱动开发 (TDD) 思想，先编写测试用例，再进行重构

减少大模型重构过程中可能发生的错误

```
1  def test_config_override():  
2      """测试配置覆盖"""  
3      print("\n测试 5: 配置覆盖")  
4      print("-" * 50)  
5  
6      try:  
7          builder = DiffusionPipelineBuilder()  
8  
9          # 设置配置  
10         builder.with_config_override(  
11             guidance_scale=7.5,
```

[python](#)

```
12         num_inference_steps=50
13     )
14
15     if "guidance_scale" in builder.config_overrides and "num_inference_steps" in
builder.config_overrides:
16         print(f"✓ 配置覆盖成功")
17         print(f"   - guidance_scale: {builder.config_overrides['guidance_scale']}")
18         print(f"   - num_inference_steps: {builder.config_overrides['num_inference_steps']}")
19         return True
20     else:
21         print(f"配置未正确设置")
22         return False
23 except Exception as e:
24     print(f"测试失败: {e}")
25     return False
```

## 3.1 重构过程

### 3.1.2 大模型辅助重构

整理设计文档，结合代码库当作  
上下文

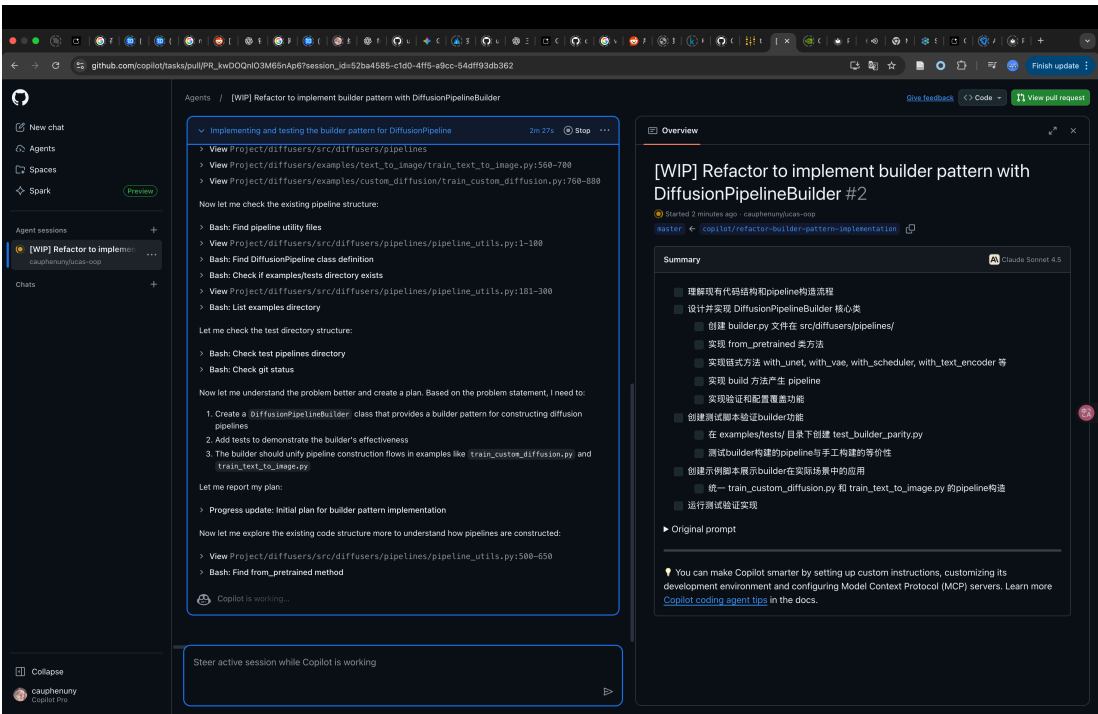


图 5 Coding Agent

## 3.2 效果展示

### 3.2.1 Builder



图 6 Builder 类图

### 3.2 效果展示

传统方式 (train\_text\_to\_image.py)

python

```
1 # 需要 8+ 行重复代码
2 noise_scheduler = DDPMScheduler.from_pretrained(
3     args.pretrained_model_name_or_path,
4     subfolder="scheduler"
5 )
6 tokenizer = CLIPTokenizer.from_pretrained(
7     args.pretrained_model_name_or_path,
8     subfolder="tokenizer", revision=args.revision
9 )
10 text_encoder = CLIPTextModel.from_pretrained(
11     args.pretrained_model_name_or_path,
12     subfolder="text_encoder",
13     revision=args.revision, variant=args.variant
14 )
15 vae = AutoencoderKL.from_pretrained(
```

Builder 方式

python

```
1 # 只需 4 行代码
2 builder =
3     DiffusionPipelineBuilder.from_pretrained(
4         args.pretrained_model_name_or_path,
5         revision=args.revision,
6         variant=args.variant,
7     )
8 # 链式配置和冻结
9 builder.with_vae(builder.components["vae"],
10                 freeze=True)
11 builder.with_text_encoder(builder.components["text_
12                 freeze=True)
```

```
12     args.pretrained_model_name_or_path,  
13     subfolder="vae", revision=args.revision,  
14     variant=args.variant  
15 )  
16 unet = UNet2DConditionModel.from_pretrained(  
17     args.pretrained_model_name_or_path,  
18     subfolder="unet",  
19     revision=args.non_ema_revision  
20 )  
21 # 手动冻结组件  
22 vae.requires_grad_(False)  
23 text_encoder.requires_grad_(False)
```

```
12 # 构造时传递参数  
13 builder.with_scheduler(DDIMScheduler,  
14     num_train_timesteps=1000)  
15 pipe = builder.build()
```

3.2.2 Attention Strategy

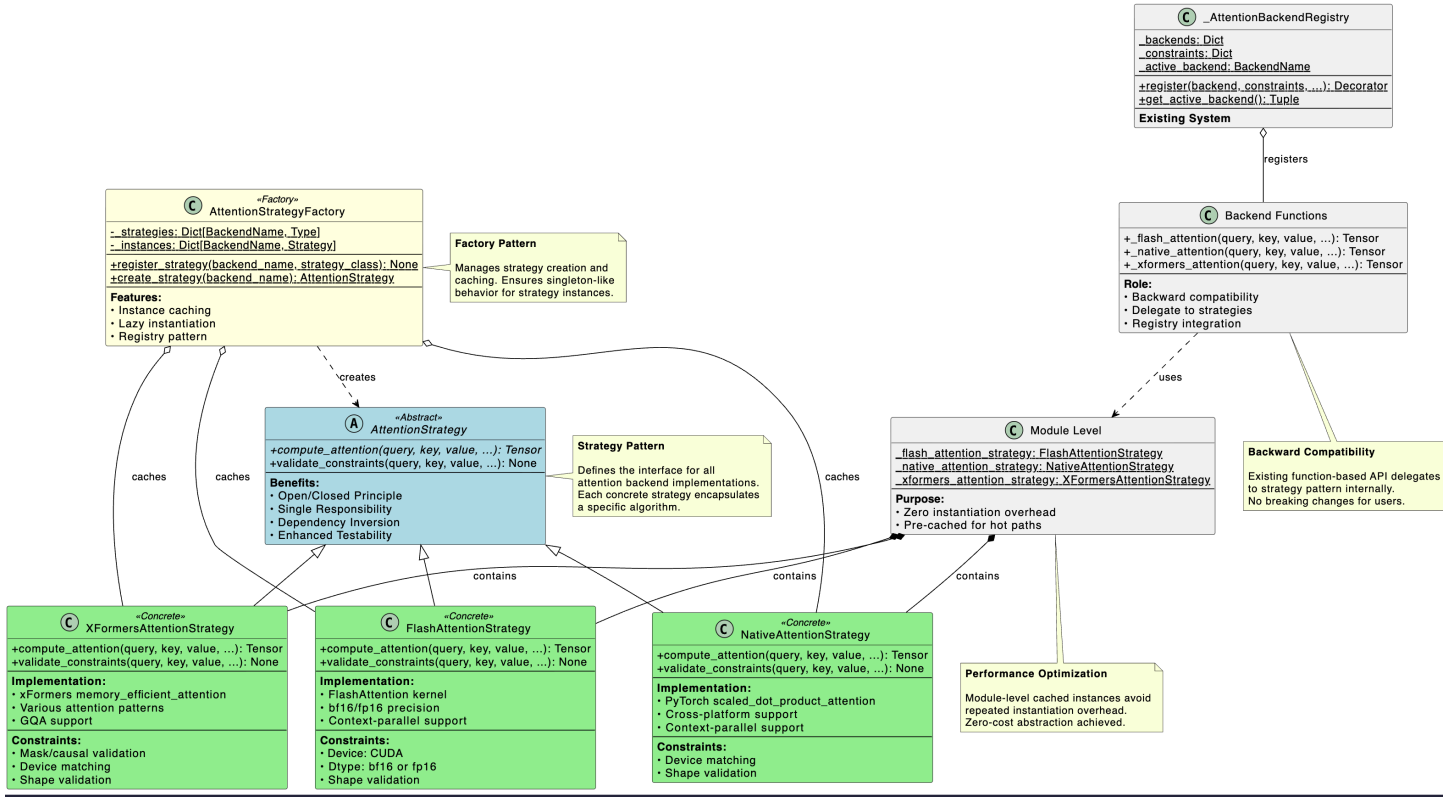


图 7 Attention Strategy 类图

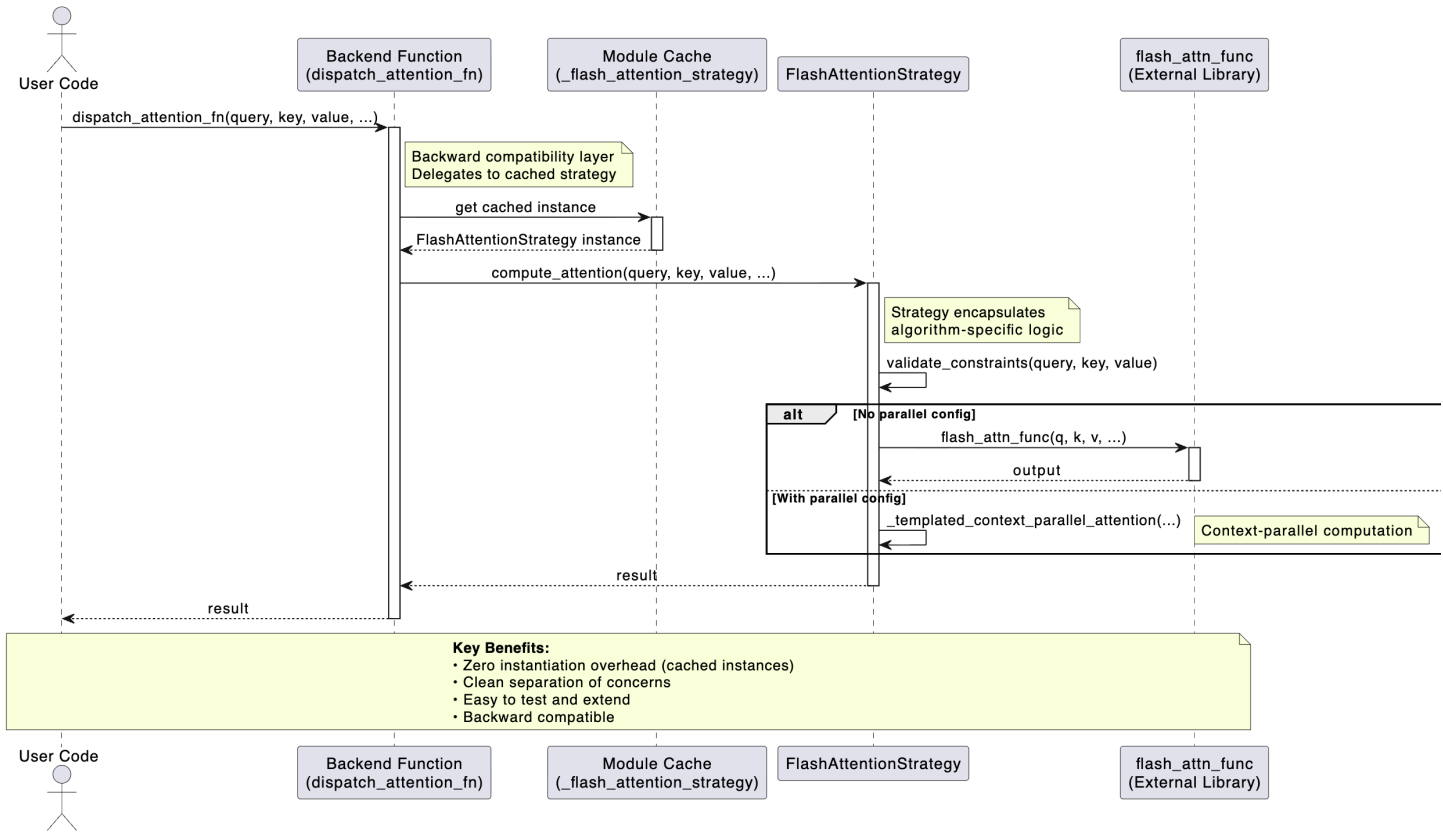


图 8 执行过程



## 3.2 效果展示

### 使用示例

```
1 # 内部使用策略模式
2 # 自动选择合适的 attention 后端
3 from diffusers.models.attention_dispatch import dispatch_attention_fn
4
5 # 根据硬件和配置自动选择策略
6 output = dispatch_attention_fn(
7     backend="FLASH", # 或 "XFORMERS", "NATIVE"
8     query=query, key=key, value=value
9 )
```

python

Thanks!