单例模式

题目链接

单例模式-小明的购物车

什么是单例设计模式

单例模式是一种**创建型设计模式**， 它的核心思想是保证**一个类只有一个实例，并提供一个全局访问点来访问这个实例。**

- 只有一个实例的意思是，在整个应用程序中，只存在该类的一个实例对象，而不是创建多个相同类型的对象。
- 全局访问点的意思是，为了让其他类能够获取到这个唯一实例，该类提供了一个全局访问点（通常是一个静态方法），通过这个方法就能获得实例。

为什么要使用单例设计模式呢

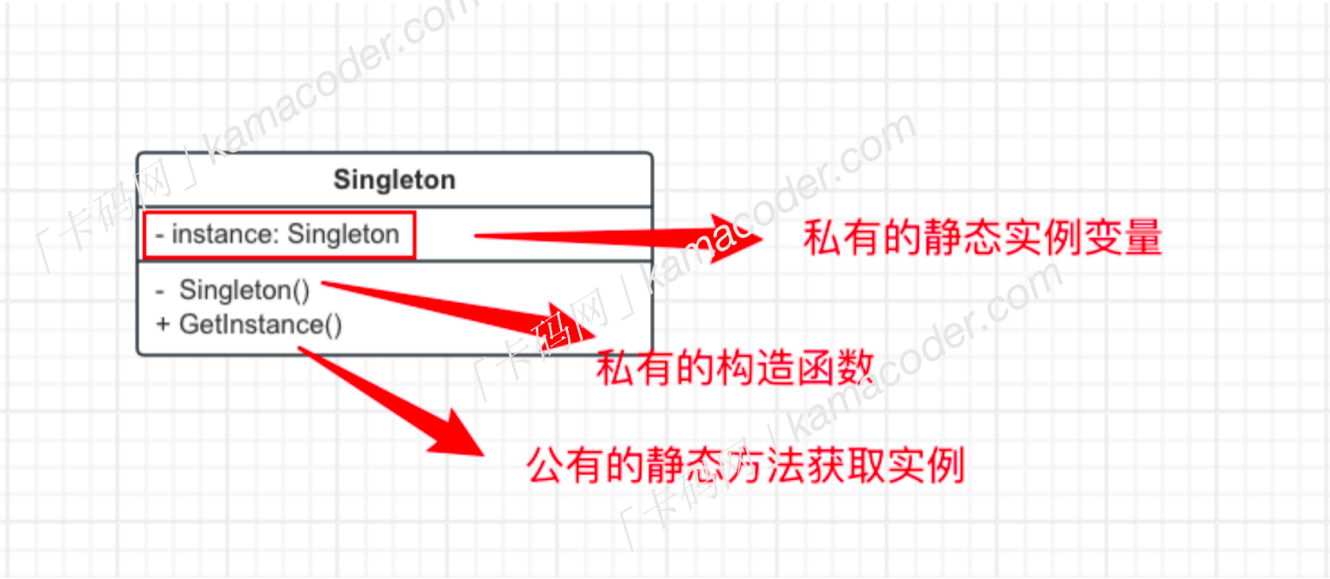
简易来说，单例设计模式有以下几个优点让我们考虑使用它：

- 全局控制：保证只有一个实例，这样就可以严格的控制客户怎样访问它以及何时访问它，简单的说就是对唯一实例的受控访问（引用自《大话设计模式》第21章）
- 节省资源：也正是因为只有一个实例存在，就避免多次创建了相同的对象，从而节省了系统资源，而且多个模块还可以通过单例实例共享数据。
- 懒加载：单例模式可以实现懒加载，只有在需要时才进行实例化，这无疑会提高程序的性能。

单例设计模式的基本要求

想要实现一个单例设计模式，必须遵循以下规则：

- 私有的构造函数：防止外部代码直接创建类的实例
- 私有的静态实例变量：保存该类的唯一实例
- 公有的静态方法：通过公有的静态方法来获取类的实例



单例设计模式的实现

单例模式的实现方式有多种，包括懒汉式、饿汉式等。

饿汉式指的是在**类加载时就已经完成了实例的创建**，不管后面创建的实例有没有使用，先创建再说，所以叫做
“饿汉”。

而懒汉式指的是只有在请求实例时才会创建，如果在首次请求时还没有创建，就创建一个新的实例，如果已经创建，就返回已有的实例，意思就是**需要使用了再创建**，所以称为“懒汉”。

在多线程环境下，由于饿汉式在程序启动阶段就完成了实例的初始化，因此不存在多个线程同时尝试初始化实例的问题，但是懒汉式中多个线程同时访问 `getInstance()` 方法，并且在同一时刻检测到实例没有被创建，就可能会同时创建实例，从而导致多个实例被创建，这种情况下我们可以采用一些同步机制，例如使用互斥锁来确保在任何时刻只有一个线程能够执行实例的创建。

> *举个例子，你和小明都发现家里没米了，在你们没有相互通知的情况下，都会去超市买一袋米，这样就重复购买了，违背了单例模式。*

下面以Java的代码作为实例，说明单例设计模式的基本写法：

1. 饿汉模式：实例在类加载时就被创建，这种方式的实现相对简单，但是实例有可能没有使用而造成资源浪费。

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

2. 懒汉模式：第一次使用时才创建

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }
    // 使用了同步关键字来确保线程安全，可能会影响性能
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

在懒汉模式的基础上，可以使用双重检查锁来提高性能。

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
```

```
            }
            return instance;
        }
    }
```

什么时候使用单例设计模式

说了这么多，那在什么场景下应该考虑使用单例设计模式呢？可以结合单例设计模式的优点来看。

1. 资源共享

多个模块共享某个资源的时候，可以使用单例模式，比如说应用程序需要一个全局的配置管理器来存储和管理配置信息、亦或是使用单例模式管理数据库连接池。

2. 只有一个实例

当系统中某个类只需要一个实例来协调行为的时候，可以考虑使用单例模式， 比如说管理应用程序中的缓存，确保只有一个缓存实例，避免重复的缓存创建和管理，或者使用单例模式来创建和管理线程池。

3. 懒加载

如果对象创建本身就比较消耗资源，而且可能在整个程序中都不一定会使用，可以使用单例模式实现懒加载。

在许多流行的工具和库中，也都使用到了单例设计模式，比如Java中的`Runtime`类就是一个经典的单例，表示程序的运行时环境。此外 Spring 框架中的应用上下文 (`ApplicationContext`) 也被设计为单例，以提供对应用程序中所有 bean 的集中式访问点。

本道题目代码

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        ShoppingCartManager cart = ShoppingCartManager.getInstance();
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNext()) {
            String itemName = scanner.next();
            int quantity = scanner.nextInt();

            // 获取购物车实例并添加商品
```

```
            cart.addToCart(itemName, quantity);
        }

        // 输出购物车内容
        cart.viewCart();
    }
}

class ShoppingCartManager {

    // 饿汉模式实现单例
    private static final ShoppingCartManager instance = new
ShoppingCartManager();

    // 购物车存储商品和数量的映射
    private Map<String, Integer> cart;

    // 私有构造函数
    private ShoppingCartManager() {
        cart = new LinkedHashMap<>();
    }

    // 获取购物车实例
    public static ShoppingCartManager getInstance() {
        return instance;
    }

    // 添加商品到购物车
    public void addToCart(String itemName, int quantity) {
        cart.put(itemName, cart.getOrDefault(itemName, 0) + quantity);
    }

    // 查看购物车
    public void viewCart() {
        for (Map.Entry<String, Integer> entry : cart.entrySet()) {
            System.out.println(entry.getKey() + " " + entry.getValue());
        }
    }
}
```

其他语言版本

Java版本

懒汉+双重锁检查

```
import java.util.Scanner;
import java.util.ArrayList;

class ShoppingCart {
    // 购物车类的单例实例变量，使用volatile关键字确保线程安全
    private static volatile ShoppingCart instance;
    // 存储商品名称
    private static ArrayList<String> productNames = new ArrayList<>();
    // 存储商品数量
    private static ArrayList<Integer> productQuantities = new ArrayList<>
();

    // 私有构造函数，防止外部直接创建ShoppingCart对象
    private ShoppingCart() {

    }

    // 获取购物车单例实例的方法，确保线程安全
    public static ShoppingCart getInstance() {
        if (instance == null) {
            synchronized (ShoppingCart.class) {
                if (instance == null) {
                    instance = new ShoppingCart();
                }
            }
        }
        return instance;
    }

    // 添加商品到购物车的方法
    public void Add(String name, int quantity) {
        productNames.add(name);
        productQuantities.add(quantity);
        System.out.println(name + " " + quantity);
    }
}

public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = ShoppingCart.getInstance();
```

```
        Scanner scanner = new Scanner(System.in);

        String inputLine;
        // 循环读取用户输入，直到用户输入"exit"
        while (scanner.hasNextLine()) {
            inputLine = scanner.nextLine();

            if ("exit".equalsIgnoreCase(inputLine)) {
                break;
            }

            // 使用空格分割输入的字符串，获取商品名称和数量
            String[] parts = inputLine.split(" ");

            // 确保输入格式正确，即包含两个部分：商品名称和数量
            if (parts.length == 2) {
                // 商品名称
                String name = parts[0];
                // 商品数量
                int quantity;

                try {
                    // 将第二部分转换为整数
                    quantity = Integer.parseInt(parts[1]);
                    cart.Add(name, quantity);
                } catch (NumberFormatException e) {
                    // 如果转换失败，输出错误信息
                    System.out.println("转换失败，请重新输入");
                }
            } else {
                // 如果输入格式不正确，输出错误信息
                System.out.println("如果输入格式不正确，请重新输入");
            }
        }

        scanner.close();
    }
}
```

CPP

```
#include <iostream>
#include <map>

using namespace std;
```

```cpp
class ShoppingCartManager {
public:
    // 获取购物车实例
    static ShoppingCartManager& getInstance() {
        static ShoppingCartManager instance;
        return instance;
    }

    // 添加商品到购物车
    void addToCart(const string& itemName, int quantity) {
        cart[itemName] += quantity;
    }

    // 查看购物车
    void viewCart() const {
        for (const auto& item : cart) {
            cout << item.first << " " << item.second << endl;
        }
    }

private:
    // 私有构造函数
    ShoppingCartManager() {}

    // 购物车存储商品和数量的映射
    map<string, int> cart;
};

int main() {
    string itemName;
    int quantity;

    while (cin >> itemName >> quantity) {
        // 获取购物车实例并添加商品
        ShoppingCartManager& cart = ShoppingCartManager::getInstance();
        cart.addToCart(itemName, quantity);
    }

    // 输出购物车内容
    const ShoppingCartManager& cart = ShoppingCartManager::getInstance();
    cart.viewCart();

    return 0;
}
```

Python

1. 饿汉模式

```
class Singleton:
    instance = None

    def __init__(self):
        if Singleton.instance is not None:
            raise Exception("Only one instance of Singleton class is
allowed")
        else:
            Singleton.instance = self
```

2. 懒汉模式

```

class Singleton:

 __instance: Singleton = None

 @staticmethod
 def getInstance():
     if Singleton.__instance is None:
  Singleton.__instance = Singleton()
     return Singleton.__instance
```

3. 懒汉+双重锁检查

```
import threading

class Singleton:
        __instance = None
        __lock = threading.Lock()

        @classmethod
        def getInstance(cls):
            if cls.__instance is None:
                with cls.__lock:
                    if cls.__instance is None:
                        # cls.__instance = super().__new__(cls)
                        cls.__instance = Singleton()
            return cls.__instance
```

Go

```go
package main

import (
 "bufio"
 "fmt"
 "os"
 "strings"
 "sync"
)

// ShoppingCartManager 实现购物车管理
type ShoppingCartManager struct {
 cart   map[string]int
 keys   []string // 用于保存插入顺序的键
 mutext sync.Mutex
}

var once sync.Once
var instance *ShoppingCartManager

// getInstance 获取购物车实例
func getInstance() *ShoppingCartManager {
 once.Do(func() {
  instance = &ShoppingCartManager{
   cart: make(map[string]int),
  }
 })
 return instance
}

// addToCart 将商品添加到购物车
func (scm *ShoppingCartManager) addToCart(itemName string, quantity int) {
 scm.mutext.Lock()
 defer scm.mutext.Unlock()

 // 检查是否已经存在，不存在则添加到 keys 中
 if _, exists := scm.cart[itemName]; !exists {
  scm.keys = append(scm.keys, itemName)
 }

 scm.cart[itemName] += quantity
}

// viewCart 查看购物车内容并按照插入顺序输出
```

```
func (scm *ShoppingCartManager) viewCart() {
 scm.mutext.Lock()
 defer scm.mutext.Unlock()

 for _, itemName := range scm.keys {
  quantity := scm.cart[itemName]
  fmt.Printf("%s %d\n", itemName, quantity)
 }
}

func main() {
 cart := getInstance()
 scanner := bufio.NewScanner(os.Stdin)

 for scanner.Scan() {
  input := scanner.Text()
  if input == "" {
   break
  }
  parts := strings.Fields(input)
  itemName := parts[0]
  quantity := 0
  if len(parts) > 1 {
   fmt.Sscanf(parts[1], "%d", &quantity)
  }

  // 获取购物车实例并添加商品
  cart.addToCart(itemName, quantity)
 }

 // 输出购物车内容
 cart.viewCart()
}
```

Typescript

```
interface IProduct {
  name: string;
  quantity: number;
}

interface IShoppingCartManager {
  add(name: string, quantity: number);
  viewCart();
}
```

```
class ShoppingCartManager implements IShoppingCartManager {
  private static instance = new ShoppingCartManager();
  private products: IProduct[] = [];

  private constructor() {}

  static getInstance() {
    return ShoppingCartManager.instance;
  }

  add(name: string, quantity: number) {
    this.products.push({ name, quantity });
  }

  viewCart() {
    this.products.forEach(({ name, quantity }) => {
      console.log(name, quantity);
    });
  }
}

(function () {
  const instance = ShoppingCartManager.getInstance();

  instance.add("Apple", 3);
  instance.add("Banana", 2);
  instance.add("Orange", 5);
  instance.viewCart();
})();
```