# 单例模式

# 概述

单例模式（Singleton Pattern）是一种创建型设计模式，确保一个类在整个系统生命周期内仅有一个实例，并提供一个全局访问点来访问该实例。其主要目的是控制实例的数量和生命周期，通常用于管理配置对象、日志对象、线程池等。

# 目的

1. **节省资源**：一个类只有一个实例，避免了实例的重复创建，从而节省了系统资源。
2. **方便控制**：在某些需要操作公共资源的场景中，使用单例模式可以避免多个实例导致的复杂操作，方便统一管理。

# 线程安全

## 什么是线程安全？

线程安全是指在拥有共享数据的多条线程并行执行的程序中，通过同步机制确保各个线程都可以正常且正确地执行，不会出现数据污染或数据竞争等问题。

## 如何保证线程安全？

1. **加锁**：对共享资源加锁，确保每个资源在同一时间至多被一个线程访问。
2. **线程私有资源**：让每个线程拥有自己的资源，如使用threadlocal可以为每个线程维护一个私有的本地变量。

# 分类

· **懒汉式（Lazy Initialization）**：在第一次使用时才创建实例，常见于单线程环境。

· **饿汉式（Eager Initialization）**：类加载时就创建实例，天然线程安全。

· **双重检查锁（Double-Checked Locking）**：懒汉式的改进，确保线程安全和效率。

· **静态内部类**：使用C++11的静态局部变量来实现懒汉式单例，这种实现称为Meyer’s Singleton，是线程安全的。

· **使用智能指针实现的线程安全懒汉式单例：**使用了双重检查锁和std::shared\_ptr来管理单例对象的生命周期。这种方式既保证了线程安全，也简化了内存管理。

· **使用 std::call\_once 实现的懒汉单例:**利用std::call\_once和std::once\_flag确保单例的线程安全初始化，代码简洁且易于理解，适用于多线程环境。

# 特点

1. **私有构造函数和析构函数**：防止外部直接创建或销毁实例。
2. **私有拷贝构造函数和赋值运算符**：防止通过拷贝或赋值创建新实例，确保实例唯一性。
3. **全局访问点**：提供一个静态方法获取唯一实例。

# 单例模式实现

## 普通懒汉式单例（线程不安全）

这种实现方式在多线程环境下不安全，因此不推荐使用。

#include <iostream>

class SingletonLazy {

private:

static SingletonLazy\* instance;

SingletonLazy() {

std::cout << "SingletonLazy instance created" << std::endl;

}

public:

static SingletonLazy\* getInstance() {

if (instance == nullptr) {

instance = new SingletonLazy();

}

return instance;

}

};

// 初始化静态成员

SingletonLazy\* SingletonLazy::instance = nullptr;

int main() {

SingletonLazy\* singleton1 = SingletonLazy::getInstance();

SingletonLazy\* singleton2 = SingletonLazy::getInstance();

if (singleton1 == singleton2) {

std::cout << "Both instances are the same (Lazy)" << std::endl;

}

return 0;

}

## 线程安全的懒汉式单例

推荐使用C++11的静态局部变量来实现懒汉式单例，这种实现称为Meyer’s Singleton，是线程安全的。

#include <iostream>

class SingletonStaticInner {

private:

SingletonStaticInner() {

std::cout << "SingletonStaticInner instance created" << std::endl;

}

public:

static SingletonStaticInner& getInstance() {

static SingletonStaticInner instance;

return instance;

}

};

int main() {

SingletonStaticInner& singleton1 = SingletonStaticInner::getInstance();

SingletonStaticInner& singleton2 = SingletonStaticInner::getInstance();

if (&singleton1 == &singleton2) {

std::cout << "Both instances are the same (Static Inner)" << std::endl;

}

return 0;

}

## 加锁的懒汉式单例

通过互斥锁实现线程安全。

#include <iostream>

#include <mutex>

class SingletonDCL {

private:

static SingletonDCL\* instance;

static std::mutex mtx;

SingletonDCL() {

std::cout << "SingletonDCL instance created" << std::endl;

}

public:

static SingletonDCL\* getInstance() {

//首先检查实例是否已经被创建，如果已经创建，则直接返回实例

if (instance == nullptr) {

std::lock\_guard<std::mutex> lock(mtx); //加锁

//在加锁后，再次检查实例是否已经被创建。这是因为在第一次检查//后可能有其他线程已经创建了实例。

if (instance == nullptr) {

instance = new SingletonDCL();

}

}

return instance;

}

};

// 初始化静态成员

SingletonDCL\* SingletonDCL::instance = nullptr;

std::mutex SingletonDCL::mtx;

int main() {

SingletonDCL\* singleton1 = SingletonDCL::getInstance();

SingletonDCL\* singleton2 = SingletonDCL::getInstance();

if (singleton1 == singleton2) {

std::cout << "Both instances are the same (DCL)" << std::endl;

}

return 0;

}

## 使用智能指针实现的线程安全懒汉单例

使用了双重检查锁和std::shared\_ptr来管理单例对象的生命周期。这种方式既保证了线程安全，也简化了内存管理。

#include <iostream>

#include <memory>

#include <mutex>

class Singleton {

public:

// 获取单例实例的方法，返回shared\_ptr

static std::shared\_ptr<Singleton> getInstance() {

if (instance == nullptr) {

std::lock\_guard<std::mutex> lock(instanceMutex); // 加锁

if (instance == nullptr) { // 双重检查

instance = std::shared\_ptr<Singleton>(new Singleton());

}

}

return instance;

}

// 打印方法，用于测试

void print() {

std::cout << "Hello World." << std::endl;

}

// 析构函数

~Singleton() {

std::cout << "Singleton instance destroyed" << std::endl;

}

private:

// 私有构造函数，防止外部创建实例

Singleton() {

std::cout << "Singleton instance created" << std::endl;

}

// 禁用拷贝构造函数和赋值运算符

Singleton(const Singleton&) = delete;

Singleton& operator=(const Singleton&) = delete;

// 静态变量声明

static std::shared\_ptr<Singleton> instance;

static std::mutex instanceMutex;

};

// 初始化静态成员

std::shared\_ptr<Singleton> Singleton::instance = nullptr;

std::mutex Singleton::instanceMutex;

int main() {

auto singleton1 = Singleton::getInstance();

singleton1->print();

auto singleton2 = Singleton::getInstance();

singleton2->print();

if (singleton1 == singleton2) {

std::cout << "Both instances are the same" << std::endl;

}

return 0;

}

## 使用 std::call\_once 实现的懒汉单例

利用std::call\_once和std::once\_flag确保单例的线程安全初始化，代码简洁且易于理解，适用于多线程环境。

#include <iostream>

#include <memory>

#include <mutex>

// 单例类的定义

class Singleton {

public:

// 获取单例实例的方法，返回shared\_ptr

static std::shared\_ptr<Singleton> getSingleton() {

std::call\_once(singletonFlag, [] {

singleton = std::shared\_ptr<Singleton>(new Singleton());

});

return singleton;

}

// 打印方法，用于测试

void print() {

std::cout << "Hello World." << std::endl;

}

// 析构函数

~Singleton() {

std::cout << \_\_PRETTY\_FUNCTION\_\_ << std::endl;

}

private:

// 私有构造函数，防止外部创建实例

Singleton() {

std::cout << \_\_PRETTY\_FUNCTION\_\_ << std::endl;

}

// 禁用拷贝构造函数和赋值运算符

Singleton(const Singleton&) = delete;

Singleton& operator=(const Singleton&) = delete;

// 静态变量声明

static std::shared\_ptr<Singleton> singleton;

static std::once\_flag singletonFlag;

};

// 初始化静态成员

std::shared\_ptr<Singleton> Singleton::singleton = nullptr;

std::once\_flag Singleton::singletonFlag;

int main() {

auto singleton1 = Singleton::getSingleton();

singleton1->print();

auto singleton2 = Singleton::getSingleton();

singleton2->print();

if (singleton1 == singleton2) {

std::cout << "Both instances are the same" << std::endl;

}

return 0;

}

## 饿汉式单例

饿汉式在类加载时就创建实例，天然线程安全，但如果实例创建过程比较重，会导致启动慢或浪费资源。

#include <iostream>

class SingletonEager {

private:

static SingletonEager\* instance;

SingletonEager() {

std::cout << "SingletonEager instance created" << std::endl;

}

public:

static SingletonEager\* getInstance() {

return instance;

}

};

// 初始化静态成员

SingletonEager\* SingletonEager::instance = new SingletonEager();

int main() {

SingletonEager\* singleton1 = SingletonEager::getInstance();

SingletonEager\* singleton2 = SingletonEager::getInstance();

if (singleton1 == singleton2) {

std::cout << "Both instances are the same (Eager)" << std::endl;

}

return 0;

}

# 总结

单例模式是一种常用的设计模式，用于确保一个类在整个系统生命周期内只有一个实例，并提供一个全局访问点。实现单例模式时需要考虑线程安全问题，可以通过懒汉式、饿汉式等不同方式实现。C++11引入的特性使得实现线程安全的单例模式更加简单和高效。