C++单例模式

在单例模式中，**懒汉模式（Lazy Initialization）** 和 **饿汉模式（Eager Initialization）** 是两种不同的实例化策略，分别适用于不同的场景。以下是它们的核心区别和实现细节：

---

1. 懒汉模式（Lazy Initialization）

核心思想：延迟初始化，只有第一次请求单例时才创建实例。
特点：

- **优点**：节省资源（避免实例未被使用时占用内存）。
- **缺点**：需处理线程安全问题（多线程环境下可能重复创建实例）。

代码实现（C++）

```
class LazySingleton {
public:
    static LazySingleton* getInstance() {
        if (instance == nullptr) {              // 检查1
            std::lock_guard<std::mutex> lock(mutex);  // 加锁
            if (instance == nullptr) {          // 检查2（双重检查锁）
                instance = new LazySingleton();
            }
        }
        return instance;
    }

    // 删除拷贝和赋值
    LazySingleton(const LazySingleton&) = delete;
    LazySingleton& operator=(const LazySingleton&) = delete;

private:
    static LazySingleton* instance;  // 懒汉模式使用指针
    static std::mutex mutex;         // 锁（用于线程安全）

    LazySingleton() {}               // 私有构造函数
    ~LazySingleton() {}
};

// 静态成员初始化
LazySingleton* LazySingleton::instance = nullptr;
std::mutex LazySingleton::mutex;
```

适用场景：

- 实例初始化开销大，但可能不会被使用。
- 对程序启动速度敏感（避免启动时加载耗时资源）。
- 需支持动态配置或依赖外部条件初始化。

---

2. 饿汉模式（Eager Initialization）

核心思想：提前初始化，在程序启动时（或类加载时）直接创建实例。
特点：

- **优点**：线程安全（无需加锁）。
- **缺点**：可能浪费资源（实例未被使用时仍占用内存）。

代码实现（C++）

```
class EagerSingleton {
public:
    static EagerSingleton& getInstance() {
        return instance;  // 直接返回已初始化的实例
    }

    // 删除拷贝和赋值
    EagerSingleton(const EagerSingleton&) = delete;
    EagerSingleton& operator=(const EagerSingleton&) = delete;

private:
    static EagerSingleton instance;  // 饿汉模式使用静态对象

    EagerSingleton() {}              // 私有构造函数
    ~EagerSingleton() {}
};

// 静态成员初始化（程序启动时即创建）
EagerSingleton EagerSingleton::instance;
```

适用场景：

- 实例初始化简单且资源占用小。
- 单例必须提前初始化（如全局配置、日志系统）。
- 多线程环境下需避免锁开销。

懒汉模式 vs 饿汉模式对比

特性	懒汉模式	饿汉模式
初始化时机	第一次调用 `getInstance()` 时	程序启动时（或类加载时）
线程安全	需手动加锁（或依赖 C++11 静态变量）	天然线程安全（无竞态条件）
资源占用	按需占用内存	提前占用内存（可能浪费资源）
性能开销	首次调用时有锁开销	无锁开销
实现复杂度	较高（需处理双重检查锁）	简单
适用场景	资源敏感、延迟初始化	简单单例、高频访问、线程安全需求

---

3. Meyers 单例（懒汉模式的优化版）

Scott Meyers 提出的单例实现是懒汉模式的升级版，利用 **局部静态变量** 的特性，在 C++11 及更高标准中实现线程安全和极简代码：

```
class MeyersSingleton {
public:
    static MeyersSingleton& getInstance() {
        static MeyersSingleton instance;  // C++11 起线程安全
        return instance;
    }

    // 删除拷贝和赋值
    MeyersSingleton(const MeyersSingleton&) = delete;
    MeyersSingleton& operator=(const MeyersSingleton&) = delete;

private:
    MeyersSingleton() {}   // 私有构造函数
    ~MeyersSingleton() {}  // 私有析构函数
};
```

优势：

- **自动线程安全**（C++11 标准保证）。
- **自动销毁**（程序结束时调用析构函数）。
- **代码简洁**（无需锁或指针管理）。

如何选择模式？

1. **优先 Meyers 单例**：
    - 若使用 C++11+，Meyers 单例是懒汉模式的最佳实践，兼具线程安全和简洁性。
2. **选择饿汉模式**：
    - 单例必须提前初始化（如日志系统）。
    - 实例简单且无依赖，无需延迟加载。
3. **传统懒汉模式**：
    - 仅在不支持 C++11 的旧代码中，需手动处理锁和指针。

总结

- **懒汉模式**：资源敏感、延迟初始化，但需处理线程安全。
- **饿汉模式**：简单高频、天然线程安全，但可能浪费资源。
- **Meyers 单例**：现代 C++ 中的懒汉模式终极方案，推荐优先使用。