Singleton单件模式

保证一个类仅有一个实例，并且提供一个该实例的全局访问点。

特殊的类，必须保证它们在系统中只存在一个实例，才能确保它们的逻辑正确性、以及良好的效率。

动机：提供一种机制来保证一个类只有一个实例。（设计者的责任）

单例模式的实现一：线程非安全版本

又叫懒汉式

```
class Singleton {
   private:
    Singleton();
    Singleton(const Singleton& other);

   public:
    static Singleton* getInstance();
    static Singleton* m_instance;
};
Singleton* Singleton::m_instance = nullptr;

Singleton* Singleton::getInstance() {
    if (m_instance == nullptr) {
        m_instance = new Singleton();
    }
    return m_instance;
}
```

适合：单线程
当多个线程同时进入if的时候就会可能出现对象被创建多个。

单例模式的实现二：线程安全版本，但锁的代价过高

```
class Singleton {
   private:
    Singleton();
    Singleton(const Singleton& other);

   public:
    static Singleton* getInstance();
    static Singleton* m_instance;
};
Singleton* Singleton::m_instance = nullptr;
Singleton* Singleton::getInstance() {
    Lock();
    if (m_instance == nullptr) {
        m_instance = new Singleton();
    }
    return m_instance;
}
```

适用：多线程，但不适用高并发

单例模式的实现三：双检查锁，但由于内存读写reorder不安全（永远不要用这个）

reorder通常认为指定序列会按我们假设的顺序进行
m_instance = new Singleton();

假象顺序：
1.先分配内存
2.调用构造器
3.将内存地址给m_instance

但是在汇编中可能132步骤进行
则在第二步(3)就不是nullptr了，这时如果有个新的线程进来，则直接返回对象
这里返回的对象只是分配了一个内存，可能还没调用构造,所以会返回一个错误的对象

这就是reorder不安全，所以我们避免使用这种实现方法

```
class Singleton {
   private:
    Singleton();
    Singleton(const Singleton& other);

   public:
    static Singleton* getInstance();
    static Singleton* m_instance;
};
Singleton* Singleton::m_instance = nullptr;
Singleton* Singleton::getInstance() {
    if (m_instance == nullptr) {
        Lock();
        if (m_instance == nullptr) {
            m_instance = new Singleton();
        }
    }
    return m_instance;
}
```

单例模式的实现四：最优的实现

(可是为什么我跑不起来！！！)

```
#include <atomic>
#include <iostream>
#include <mutex>
#include <thread>
using namespace std;

class Singleton {
private:
    Singleton() { cout << 1 << endl; };
    Singleton(const Singleton& other);

public:
    static Singleton* getInstance();
    static std::atomic<Singleton*> m_instance;
    static std::mutex m_mutex;
```

```
};

// C++ 11版本之后的跨平台实现(volatile)
Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}

int main() {
    Singleton* instance = Singleton::getInstance();
    return 0;
}
```

单例伴随着进程的生命周期， 常驻内存， 不需要程序员来释放（实际上， 人为释放是有风险的）。 如果进程终结， 对应的堆内存自动被回收， 不会泄露。

单例模式的实现五：【c++ 11版本最简洁的跨平台方案】

局部静态变量不仅只会初始化一次，而且还是线程安全的。

```
class Singleton {
   public:
    // 注意返回的是引用。
    static Singleton& getInstance() {
        static Singleton m_instance;  //局部静态变量
        return m_instance;
    }

   private:
    Singleton();  //私有构造函数，不允许使用者自己生成对象
    Singleton(const Singleton& other);
};

int main() {
    Singleton& instance = Singleton::getInstance();
    return 0;
}
```

这种单例被称为Meyers' Singleton 。这种方法很简洁，也很完美，但是注意：

- gcc 4.0之后的编译器支持这种写法。
- C++11及以后的版本（如C++14）的多线程下，正确。
- C++11之前不能这么写。