# 背景

对于系统中的某些类来说，只有一个实例很重要，例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；**一个系统只能有一个计时工具或ID（序号）生成器**。

在我们点击某一个工具箱按钮时，希望每次点击后都只弹出一个菜单，而不是每次点击后弹出多个菜单，这就要求这个菜单只被实例化一次，这就是单例模式的一个应用场景。

如何保证一个类只有一个实例并且这个实例易于被访问呢？定义一个静态变量或全局变量可以确保对象随时都可以被访问，但**不能防止我们实例化多个对象**。

一个更好的解决办法是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

**“对象性能”模式：**

面向对象很好地解决了“抽象”的问题，但是必不可免地要付出一定的代价。对于通常情况来讲，面向对象的成本大都可以忽略不计。但是某些情况下，面向对象所带来的成本必须谨慎处理。于是，出现了为性能和安全而设计的单例模式。

# 定义

单例模式(Singleton Pattern)：单例模式确保某一个类**只有一个实例**，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它**提供全局访问的方法**。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

# 分类

## 饿汉式

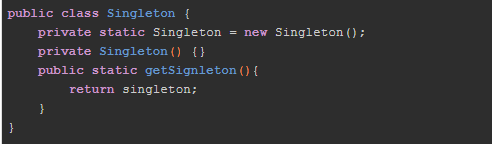
饿汉法就是在第一次引用该类的时候就创建对象实例，而不管实际是否需要创建。

一上来就先实例化，但若类没有使用的话，就有点浪费资源类。

**思路：**

我们应通过将构造函数设置为非公有来保证其不会被用户代码随意创建。而在类型实例访问函数中，我们通过局部静态变量达到实例仅有一个的要求。另外，通过该静态变量，我们可以将该实例的创建延迟到实例访问函数被调用时才执行，以提高程序的启动速度。

**代码：**



public class Singleton {

private static instance = new Singleton();

private Singleton() {}

public static getInstance() {

return m\_instance;

}

}

**优点：**这样做的好处是编写简单，无需关注线程安全问题

**缺点：**

1、如果在一个大环境下使用了过多的饿汉单例，则会生产出过多的实例对象，无论你是否要使用他们。

**2、无法做到延迟创建对象**。但是我们很多时候都希望对象可以尽可能地延迟加载，从而减小负载，所以就需要下面的懒汉法。

**完整代码：**

#define CRT SECURE NO WARNINGS

#include<iostream>

#include<string>

using namespace std;

//饿汉式

class Singleton\_hungry{

class Garbo{

~Garbo(){

if (pSingleton != NULL) {

delete pSingleton;

}

}

};//垃圾回收类

//静态成员变量

private:

static Singleton\_hungry\*pSingleton;//指向对象的指针

static Garbo garbo;

//当程序退出时，会析构garbo，也会删除堆中的饿汉对象

//https://blog.csdn.net/libaineu2004/article/details/79391423

//进程结束后，进程的所有内存都将被释放，包括堆上的内存泄露的内存

//成员函数

public:

static Singleton\_hungry\*getInstance(){

return pSingleton;

}

#if 0

//只有一个，不敢这样释放（不应该提供用户这个方法，在对象的析构函数中释放）

//（因为这个是全局变量，如果某些地方不小心释放了会影响别处使用）

static void freeSpace(){

if (pSingleton != NULL){

delete pSingleton;

}

}

#endif

//对象

//成员变量

private:

Singleton\_hungry(){

cout << "我是饿汉构造!" << endl;

}

};

Singleton\_hungry\* Singleton\_hungry::pSingleton = new Singleton\_hungry;

int main()

{

cout << "进入main函数" << endl;

Singleton\_hungry\* p3 = Singleton\_hungry::getInstance();

Singleton\_hungry\* p4 = Singleton\_hungry::getInstance();

if (p3 == p4){

cout << "两个指针指向同一块内存空间,是单例!" << endl;

}

else{

cout << "不是单例模式!" << endl;

}

return 0;

}

## 懒汉/饱汉式

饱汉模式，很饱不着急，延迟加载，啥时候用啥时候创建实例，存在线程安全问题。

实例在开始时为空，第一次加载后才实例化。

可节约一些资源，但在并发时有可能出现多个单例。

**优点：**延时加载，用的时候才会生产对象。

**缺点：**需要保证同步，付出效率的代价。

**单例模式类图：**



### 单线程

这种写法是最简单的，由私有构造器和一个公有静态工厂方法构成，在工厂方法中对singleton进行null判断，如果是null就new一个出来，最后返回singleton对象。这种方法可以实现延时加载，但是有一个致命弱点：线程不安全。如果有两条线程同时调用getSingleton()方法，就有很大可能导致重复创建对象。

public class Singleton {

private static Singleton singleton = null;

private Singleton(){}

public static Singleton getSingleton() {

if(singleton == null) singleton = new Singleton();

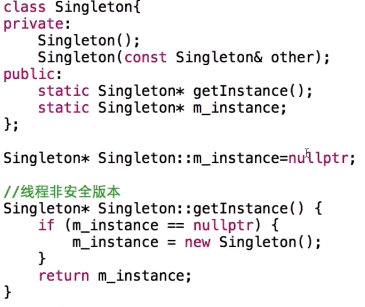
return singleton;

}

}

**优点：**资源利用率高，不执行getInstance()就不会被实例，可以执行该类的其他静态方法。

**缺点：**第一次加载时不够快，线程不安全，多线程使用不必要的同步开销大。



注：

构造函数设置为private，这样就堵塞了外部去new对象的途径，获取对象的方法getInstance()必须使用static，这样它是独立于类对象的，可以直接访问，因为在它调用的时候还没有构造对象。

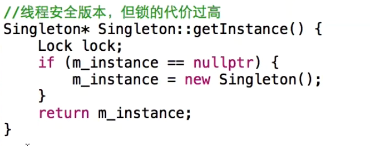
Thread1如果执行到m\_instance == nullptr处，判断为真，可以执行new Singleton操作，同时Thread2也执行到该处，判断m\_instance == nullptr成立，这样都进入new操作了。

### 多线程

饱汉模式为了保证线程安全，就用synchronized关键字标识了方法。之所以被称为“饱汉”，因为它很饱，不急着生产实例，在需要的时候才会生产。

#### 直接加锁

使用直接加锁方式，但是代价太高了：



Singleton\* Singleton::getInstance() {

Lock lock;

if (nullptr == m\_instance) {

m\_instance = new Singleton();

}

return m\_instance;

}

**优点：**线程安全

**缺点：**代价太高

**考虑线程安全的写法**

这种写法考虑了线程安全，将对singleton的null判断以及new的部分使用synchronized进行加锁。同时，对singleton对象使用volatile关键字进行限制，保证其对所有线程的可见性，并且禁止对其进行指令重排序优化。如此即可从语义上保证这种单例模式写法是线程安全的。注意，这里说的是语义上，实际使用中还是存在小坑的。

public class Singleton {

private static volatile Singleton singleton = null;

private Singleton(){}

public static Singleton getSingleton(){

synchronized (Singleton.class){

if(singleton == null){

singleton = new Singleton();

}

}

return singleton;

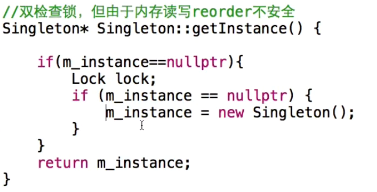
}

}

#### 双检查锁

双重锁模式，是饱汉模式的优化，进行双重判断，当已经创建过实例对象后就无需加锁，提高效率。也是一种推荐使用的方式。

可以使用双检查锁，但由于内存读写reorder不安全（可以理解为实际不可用）：



Singleton\* Singleton::getInstance() {

if (nullptr == m\_instance) {

Lock lock;

if (nullptr == m\_instance) {

m\_instance = new Singleton();

}

}

return m\_instance;

}

只有对象为空的时候才加锁，加完锁后再判空，防止在加锁的过程中被另一个线程调用new，即进行双检查：第一次检查是避免代价过高的问题，第二次检查是防止多线程问题。

**reorder问题：**一般new的执行过程认为是分配内存->构造函数初始化->返回地址，但是实际上可能是分配内存->返回地址->构造函数这种错乱的顺序。如果线程1reorder，另外一个线程2判断非空直接返回pSingleton，但是这个对象实例是无法正常使用的，它只是一个还未调用构造函数初始化的内存。

所以编译器需要解决这类问题，即编译器不能优化。

**兼顾线程安全和效率的写法**

虽然上面这种写法是可以正确运行的，但是其效率低下，还是无法实际应用。因为每次调用getSingleton()方法，都必须在synchronized这里进行排队，而真正遇到需要new的情况是非常少的。所以，就诞生了第三种写法：

public class Singleton {

private static volatile Singleton singleton = null;

private Singleton(){}

public static Singleton getSingleton(){

if(singleton == null){

synchronized (Singleton.class){

if(singleton == null){

singleton = new Singleton();

}

}

}

return singleton;

}

}

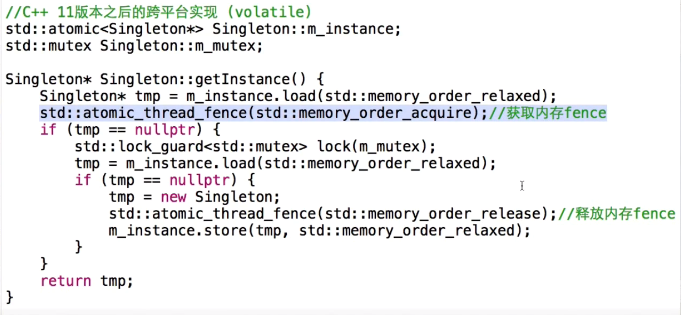
这种写法被称为“双重检查锁”，顾名思义，就是在getSingleton()方法中，进行两次null检查。看似多此一举，但实际上却极大提升了并发度，进而提升了性能。为什么可以提高并发度呢？就像上文说的，在单例中new的情况非常少，绝大多数都是可以并行的读操作。因此在加锁前多进行一次null检查就可以减少绝大多数的加锁操作，执行效率提高的目的也就达到了。

**优点：**资源利用率高，不执行getInstance()就不被实例，可以执行该类其他静态方法

**缺点：**第一次加载时反应不快，由于java内存模型一些原因偶尔失败

#### volatile

为了解决双检查reorder问题，C++11采用volatile（Java使用关键字volatile）：



std::atomic<Singleton\*> Singleton::m\_instance;

std::mutex Singleton::m\_mutex;

Singleton\* Singleton::getInstance() {

Singleton\* tmp = m\_instance.load(std::memory\_order\_relaxed);

std::atomic\_thread\_fence(std::memory\_order\_acuire);//获取内存fence

if (nullptr == tmp) {

tmp = new Singleton;

std::atomic\_thread\_fence(std::memory\_order\_release);//释放内存fence

m\_instance.store(tmp,std::memory\_order\_relaxed);

}

return tmp;

}

其实这个关键字有两层语义。第一层语义相信大家都比较熟悉，就是可见性。可见性指的是在一个线程中对该变量的修改会马上由工作内存（Work Memory）写回主内存（Main Memory），所以会马上反应在其它线程的读取操作中，即看到的都是最新的结果。

顺便一提，工作内存和主内存可以近似理解为实际电脑中的高速缓存和主存，工作内存是线程独享的，主存是线程共享的。volatile的第二层语义是禁止指令重排序优化。大家知道我们写的代码（尤其是多线程代码），由于编译器优化，在实际执行的时候可能与我们编写的顺序不同。

编译器只保证程序执行结果与源代码相同，却不保证实际指令的顺序与源代码相同。这在单线程看起来没什么问题，然而一旦引入多线程，这种乱序就可能导致严重问题。volatile关键字就可以从语义上解决这个问题。

注意，前面反复提到 “从语义上讲是没有问题的”，但是很不幸，禁止指令重排优化这条语义直到 jdk1.5 以后才能正确工作。此前的 JDK 中即使将变量声明为 volatile 也无法完全避免重排序所导致的问题。所以，在 jdk1.5 版本前，双重检查锁形式的单例模式是无法保证线程安全的。

**完整代码：**

#define CRT SECURE NO WARNINGS

#include<iostream>

#include<string>

using namespace std;

/\*在外面不能利用这个类去创建对象

实现单例步骤：

1.构造函数私有化（控制外部访问权限）

2.增加静态私有的当前类的指针变量（否则无法创建类对象）

3.提供静态对外接口,可以让用户获得单例对象

\*/

//懒汉式

class Singleton\_lazy{

//静态

private:

static Singleton\_lazy \*pSingleton;

public:

static Singleton\_lazy\* getInstance(){

//方案一：加锁Lock lock

/\*线程非安全的，可以在该new操作前加锁，但是加锁代价太大

另一个线程获取锁失败需要等待

这里是获取对象实例，对于读操作的线程其实都是浪费的，没必要加锁

\*/

if (pSingleton == NULL){

//Lock lock;

//方案二：双检查锁，但由于内存读写reorder不安全

pSingleton = new Singleton\_lazy;

//只有调用getInstance才会new对象，故称“懒汉”

}

return pSingleton;

}

static void freeSpace(){

if (pSingleton != NULL){

delete pSingleton;

}

}

//对象

private:

Singleton\_lazy(){

cout << "懒汉式创建" << endl;

}

};

Singleton\_lazy \* Singleton\_lazy::pSingleton=NULL;

//在类外部初始化，不分配内存

int main()

{

cout << "main函数" << endl;

Singleton\_lazy \*p1=Singleton\_lazy::getInstance();

Singleton\_lazy \*p2 = Singleton\_lazy::getInstance();

if (p1 == p2){

cout << "两个指针指向同一块内存空间,是单例!" << endl;

}

else{

cout << "不是单例模式!" << endl;

}

Singleton\_lazy::freeSpace();

return 0;

}

**多线程下的懒汉式：**

**懒汉式遇到多线程是不安全的！饿汉式是线程安全的！**

#define CRT SECURE NO WARNINGS

#include<iostream>

#include<string>

using namespace std;

/\*在外面不能利用这个类去创建对象

实现单例步骤：

1.构造函数私有化

2.增加静态私有的当前类的指针变量

3.提供静态对外接口,可以让用户获得单例对象

\*/

//懒汉式

class Singleton\_lazy{

//静态

private:

static Singleton\_lazy \*pSingleton;

public:

static Singleton\_lazy\* getInstance(){

if (pSingleton == NULL){

//当两个线程到这时，有可能都会读取为NULL，从而创建多个对象，所以需加锁

pSingleton = new Singleton\_lazy;

}

return pSingleton;

}

static void freeSpace(){

if (pSingleton != NULL){

delete pSingleton;

}

}

//对象

private:

Singleton\_lazy(){

cout << "懒汉式创建" << endl;

}

};

Singleton\_lazy \* Singleton\_lazy::pSingleton=NULL;

int main()

{

cout << "main函数" << endl;

//多线程需要考虑加锁

Singleton\_lazy \*p1=Singleton\_lazy::getInstance();

Singleton\_lazy \*p2 = Singleton\_lazy::getInstance();

if (p1 == p2){

cout << "两个指针指向同一块内存空间,是单例!" << endl;

}

else{

cout << "不是单例模式!" << endl;

}

Singleton\_lazy::freeSpace();

return 0;

}

## 静态内部类

有没有一种延时加载，并且能保证线程安全的简单写法呢？我们可以把Singleton实例放到一个静态内部类中，这样就避免了静态实例在Singleton类加载的时候就创建对象，并且由于静态内部类只会被加载一次，所以这种写法也是线程安全的：

public class Singleton {

private static class Holder {

private static Singleton singleton = new Singleton();

}

private Singleton(){}

public static Singleton getSingleton(){

return Holder.singleton; //静态类方式获取实例

}

}

但是，上面提到的所有实现方式都有两个共同的缺点：

都需要额外的工作 (Serializable、transient、readResolve()) 来实现序列化，否则每次反序列化一个序列化的对象实例时都会创建一个新的实例。可能会有人使用反射强行调用我们的私有构造器（如果要避免这种情况，可以修改构造器，让它在创建第二个实例的时候抛异常）。枚举写法 当然，还有一种更加优雅的方法来实现单例模式，那就是枚举写法：

public enum Singleton {

INSTANCE;

private String name;

public String getName(){

return name;

}

public void setName(String name){

this.name = name;

}

}

使用枚举除了线程安全和防止反射强行调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。因此，Effective Java 推荐尽可能地使用枚举来实现单例。

**优点：**资源利用率高，不执行getInstance()不被实例，可以执行该类其他静态方法

**缺点：**第一次加载时反应不够快

## 选择

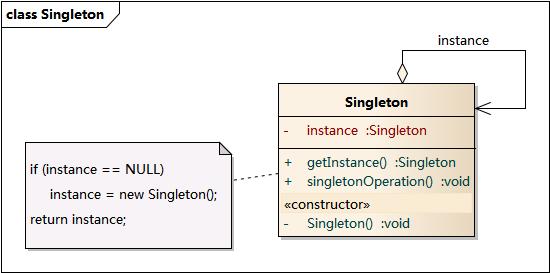
|  |  |  |
| --- | --- | --- |
| 单例模式 | 懒汉模式 | 饿汉模式 |
| 概念 | 在类加载时不创建实例，采用延迟加载的方式，在运行调用时创建实例 | 在类加载的时候，就完成初始化 |
| 特点 | 类加载速度快，但是运行时获取对象的速度较慢（时间换空间）。 | 类加载较慢，但获取对象速度快（空间换时间）。 |
| 延迟加载（lazy loading） | 具备 | 不具备 |
| 线程安全 | 线程不安全 | 线程安全 |

一般采用饿汉式，若对资源十分在意可以采用静态内部类，不建议采用懒汉式及双重检测。

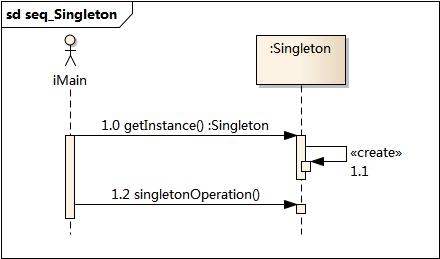
# 模式结构

单例模式包含如下角色：

Singleton：单例



# 时序图



# 代码实现

#include <iostream>

#include "Singleton.h"

using namespace std;

int main(int argc, char \*argv[])

{

Singleton \* sg = Singleton::getInstance();

sg->singletonOperation();

return 0;

}

#include "Singleton.h"

#include <iostream>

using namespace std;

Singleton \* Singleton::instance = NULL;

Singleton::Singleton(){

}

Singleton::~Singleton(){

delete instance;

}

Singleton\* Singleton::getInstance(){

if (instance == NULL)

{

instance = new Singleton();

}

return instance;

}

void Singleton::singletonOperation(){

cout << "singletonOperation" << endl;

}

# 分析

单例模式的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点。单例模式包含的角色只有一个，就是单例类——Singleton。单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法，该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

在单例模式的实现过程中，需要注意如下三点：

单例类的构造函数为私有（可以设置为protected以允许子类派生）；

提供一个自身的静态私有成员变量；

提供一个公有的静态工厂方法。

注：Singleton模式一般不要支持拷贝构造函数和Clone接口，因为这有可能导致多个对象实例，与Singleton模式的初衷违背。

# 特点

## 优点

提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的概念。

由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。

允许可变数目的实例。我们可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

## 缺点

由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。

单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。

滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；现在很多面向对象语言(如Java、C#)的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致对象状态的丢失。

# 适用环境

在以下情况下可以使用单例模式：

系统只需要一个实例对象（线程池、缓存、硬件设备等），如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。

客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。

在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。

一个具有自动编号主键的表可以有多个用户同时使用，但数据库中只能有一个地方分配下一个主键编号，否则会出现主键重复，因此该主键编号生成器必须具备唯一性，可以通过单例模式来实现。

# 实例

在操作系统中，打印池(Print Spooler)是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。现使用单例模式来模拟实现打印池的设计。