<https://www.cnblogs.com/dupengcheng/p/7205527.html>

单例模式是任何面向对象语言绕不过的，单例模式是很有必要的，接下来我用最朴素的语言来解释和记录单例模式的学习。

* 什么是单例模式？

单例模式就是一个类只能被实例化一次 ，更准确的说是**只能有一个实例化的对象的类。**

* 创建一个单例模式的类（初想）

一个类只能有一个实例化的对象，那么这个类就要禁止别人new出来，或者通过直接定义一个对象出来

[复制代码](javascript:void(0);)

class CAR

{

public:

CAR(){}

~CAR(){}

};

CAR a;

CAR \*b = new CAR;

[复制代码](javascript:void(0);)

很明显这样的类可以被程序员用上面这两种方式实例化。那么考虑，如何禁止用上面的这两种方式实例化一个类呢？

如果把构造函数私有化，很明显上面这两种方法都会默认的去调用构造函数，当构造函数是private或者protected时，构造函数将无法从外部调用。

[复制代码](javascript:void(0);)

class CSingleton

{

private:

CSingleton()

{

}

};

int main()

{

CSingleton t;

CSingleton \*tt = new CSingleton;

}

[复制代码](javascript:void(0);)

上面的代码选择了这样实例化类，很明显编译器会报错，因为私有化的构造函数无法被外部调用

error: ‘CSingleton::CSingleton()’ is private

既然构造函数是私有了，那么他就只能被类内部的成员函数调用，所以我们可以搞一个共有函数去供外部调用，然后这个函数返回一个对象，为了保证多次调用这个函数返回的是一个对象，我们可以把类内部要返回的对象设置为静态的，就有了下面的代码：

[复制代码](javascript:void(0);)

class CSingleton

{

private:

CSingleton()

{

}

static CSingleton \*p;

public:

static CSingleton\* getInstance()

{

if(p == NULL)

p = new CSingleton();

return p;

}

};

CSingleton\* CSingleton::p = NULL;

[复制代码](javascript:void(0);)

我们在主函数调用来测试一下

[复制代码](javascript:void(0);)

int main()

{

CSingleton \*t = CSingleton::getInstance();

CSingleton \*tt = CSingleton::getInstance();

cout << t << endl << tt << endl;

}

[复制代码](javascript:void(0);)

结果是

0x1c59c0  
0x1c59c0

两个地址一样，证明我们的单例类的正确的，原理其实很简单，第一次调用获取实例的函数时，静态类的变量指针空，所以会创建一个对象出来，第二次调用就不是空了，直接返回第一次的对象指针（地址）。

同时思考另一个问题，如果两个线程同时获取实例化对象呢？显然是不行的，会出现两个线程同时要对象的时候指针还都是空的情况就完了，想到这种情况你肯定会毫不犹豫的去加个锁。（进一步思考）

[复制代码](javascript:void(0);)

class CSingleton

{

private:

CSingleton()

{

pthread\_mutex\_init(&mtx,0);

}

static CSingleton \*p;

public:

static pthread\_mutex\_t mtx;

static CSingleton\* getInstance()

{

if(p == NULL)

{

pthread\_mutex\_lock(&mtx);

p = new CSingleton();

pthread\_mutex\_unlock(&mtx);

}

return p;

}

};

pthread\_mutex\_t CSingleton::mtx;

CSingleton\* CSingleton::p = NULL;

[复制代码](javascript:void(0);)

上面的代码就是加锁之后的了，你可以用下面的方法调用

[复制代码](javascript:void(0);)

void\* fun1(void \*)

{

while(1)

{

CSingleton \*pt = CSingleton::getInstance();

cout << "fun1: pt\_addr = " << pt << endl;

Sleep(1000);

}

}

void\* fun2(void \*)

{

while(1)

{

CSingleton \*pt = CSingleton::getInstance();

cout << "fun2: pt\_addr = " << pt << endl;

Sleep(1000);

}

}

void callSingleton()

{

pthread\_mutex\_init(&CSingleton::mtx,0);

pthread\_t pt\_1 = 0;

pthread\_t pt\_2 = 0;

int ret = pthread\_create(&pt\_1,0,&fun1,0);

if(ret != 0)

{

printf("error\n");

}

ret = pthread\_create(&pt\_2,0,&fun2,0);

if(ret != 0)

{

printf("error\n");

}

pthread\_join(pt\_1,0);

pthread\_join(pt\_2,0);

}

[复制代码](javascript:void(0);)

你可以这样在fun1,fun2中随意的去实例化这个类了，运行结果如下

fun1: pt\_addr = 0xb85a38  
fun2: pt\_addr = 0xb85a38  
fun1: pt\_addr = 0xb85a38  
fun2: pt\_addr = 0xb85a38  
fun1: pt\_addr = 0xb85a38  
fun2: pt\_addr = 0xb85a38

总结一下我们的这种实现单例模式的方式：类中声明一个静态的本类指针，再写一个public的函数来让这个指针指向我们新创建的实例，返回这个指针（这个实例的地址），并进行加锁，这个对象就永远只有一份，然后单例模式就实现了。

[复制代码](javascript:void(0);)

class CSingleton

{

private:

CSingleton()

{

pthread\_mutex\_init(&mtx,0);

}

public:

static pthread\_mutex\_t mtx;

static CSingleton\* getInstance()

{

pthread\_mutex\_lock(&mtx);

static CSingleton obj;

pthread\_mutex\_unlock(&mtx);

return &obj;

}

};

pthread\_mutex\_t CSingleton::mtx;

[复制代码](javascript:void(0);)

也可以像上面的代码一样把静态对象的放到函数里面，这样就省的在去外部声明一下了，只要返回一个静态类的地址，就算这个函数执行完也不会被销毁，它被保存在静态区和全局变量差不多。

再次总结：只要返回一个本类对象的地址就好了，这个地址要是静态的。别忘记加锁。

而且上面这种方式也被人们成为懒汉模式，为什么叫懒汉？因为这样的方式只有在我调用 CSingleton::getInstance(); 的时候才会返回一个实例化的对象，懒死了，我不要你你就不给我，是不是？

下面这种方式就和上面的不同，人家还没要，我就忍不住先给人家准备好了，如饥似渴，所以也叫饿汉模式。

我们注意到上面第一种方式，类中的静态变量要先被外部声明，否则编译器不会为它分配空间，像这样 CSingleton\* CSingleton::p = NULL; 其实我们可以在这一步就new一个对象出来，因为p是CSingleton的成员，它是可以调用构造函数的哦，于是我们改成这样就是饿汉模式了

[复制代码](javascript:void(0);)

class CSingleton

{

private:

CSingleton()

{

}

static CSingleton \*p;

public:

static CSingleton\* getInstance()

{

return p;

}

};

CSingleton\* CSingleton::p = new CSingleton();

[复制代码](javascript:void(0);)

我们这样锁也不用加了，因为我们调用 CSingleton::getInstance() 之前这个类就已经被实例化了，我们调用这个函数的目地只是为了得到这个对象的地址。饿汉模式就实现了

总结：利用静态变量和私有化构造函数的特性来实现单例模式。搞一个静态的自身类指针，然后把构造函数私有化，这样new的时候就只能让本类中的成员调用，然后不择手段在类内部new出这个对象，并提供一种方法供外部得到这个对象的地址。