一、什么是单例

单例 Singleton 是设计模式的一种，其特点是只提供唯一一个类的实例,具有全局变量的特点，在任何位置都可以通过接口获取到那个唯一实例;

具体运用场景如：

设备管理器，系统中可能有多个设备，但是只有一个设备管理器，用于管理设备驱动;

数据池，用来缓存数据的数据结构，需要在一处写，多处读取或者多处写，多处读取;

二、C++单例的实现

2.1 基础要点

全局只有一个实例：static 特性，同时禁止用户自己声明并定义实例（把构造函数设为 private）

线程安全

禁止赋值和拷贝

用户通过接口获取实例：使用 static 类成员函数

2.2 C++ 实现单例的几种方式

2.2.1 有缺陷的懒汉式

懒汉式(Lazy-Initialization)的方法是直到使用时才实例化对象，也就说直到调用get\_instance() 方法的时候才 new 一个单例的对象。好处是如果被调用就不会占用内存。

成员函数后面加上default和delete。

Delete:禁止被使用。

Default:

C++ 的类有四类特殊成员函数，它们分别是：默认构造函数、析构函数、拷贝构造函数以及拷贝赋值运算符。

这些类的特殊成员函数负责创建、初始化、销毁，或者拷贝类的对象。

如果程序员没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，则编译器会隐式的为这个类生成一个默认的特殊成员函数。

// C++11 标准引入了一个新特性："=default"函数。程序员只需在函数声明后加上“=default;”，就可将该函数声明为 "=default"函数，编译器将为显式声明的 "=default"函数自动生成函数体

//编译器将为函数自动生成函数体。

X() = default; //该函数比用户自己定义的默认构造函数获得更高的代码效率

// "=default"函数特性仅适用于类的特殊成员函数，且该特殊成员函数没有默认参数。

"=default"函数既可以在类体里（inline）定义，也可以在类体外（out-of-line）定义。

"=delete"函数特性还可用于禁用类的某些转换构造函数，从而避免不期望的类型转换.

X4(double)

{

}

X4(int) = delete;

"=delete"函数特性还可以用来禁用某些用户自定义的类的 new 操作符，从而避免在自由存储区创建类的对象

class X5

{

public:

void \*operator new(size\_t) = delete;

void \*operator new[](size\_t) = delete;

};

class Singleton

{

private:

  Singleton(){

    std::cout<<"constructor called!"<<std::endl;

  }

  Singleton(Singleton&)=delete;

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

  static Singleton\* m\_instance\_ptr;

public:

  ~Singleton()

  {

      std::cout<<"destruct called"<<std::endl;

  }

  static Singleton\* get\_instance()

  {

      if(m\_instance\_ptr==NULL)

      {

          m\_instance\_ptr=new Singleton();

      }

      return m\_instance\_ptr;

  }

};

Singleton\* Singleton::m\_instance\_ptr=NULL;

int main()

{

    Singleton \*instance=Singleton::get\_instance();

    Singleton \*instance\_1=Singleton::get\_instance();

}

可以看到，获取了两次类的实例，却只有一次类的构造函数被调用，表明只生成了唯一实例，这是个最基础版本的单例实现，他有哪些问题呢？

线程安全的问题,当多线程获取单例时有可能引发竞态条件：第一个线程在if中判断 m\_instance\_ptr是空的，于是开始实例化单例;同时第2个线程也尝试获取单例，这个时候判断m\_instance\_ptr还是空的，于是也开始实例化单例;这样就会实例化出两个对象,这就是线程安全问题的由来; 解决办法:加锁

内存泄漏. 注意到类中只负责new出对象，却没有负责delete对象，因此只有构造函数被调用，析构函数却没有被调用;因此会导致内存泄漏。解决办法： 使用共享指针;

class Singleton

{

private:

Singleton(Singleton&) = delete;

Singleton & operator=(Singleton&) = delete;

Singleton() {

std::cout << "gouzao" << std::endl;

}

static std::shared\_ptr<Singleton> m\_instance;

static std::mutex m\_mutex;

public:

~Singleton()

{

std::cout << "xigou" << std::endl;

}

static std::shared\_ptr<Singleton> get\_instance()

{

if (m\_instance == NULL)

{

std::lock\_guard<std::mutex> lock(m\_mutex);

if (m\_instance == NULL)

{

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

}

}

return m\_instance;

}

};

std::shared\_ptr<Singleton> Singleton::m\_instance = NULL;

std::mutex Singleton::m\_mutex;

int main()

{

std::shared\_ptr<Singleton> m = Singleton::get\_instance();

std::shared\_ptr<Singleton> n = Singleton::get\_instance();

std::cout << "Hello World!\n";

}

hared\_ptr和mutex都是C++11的标准，以上这种方法的优点是

基于 shared\_ptr, 用了C++比较倡导的 RAII思想，用对象管理资源,当 shared\_ptr 析构的时候，new 出来的对象也会被 delete掉。以此避免内存泄漏。

加了锁，使用互斥量来达到线程安全。这里使用了两个 if判断语句的技术称为双检锁；好处是，只有判断指针为空的时候才加锁，避免每次调用 get\_instance的方法都加锁，锁的开销毕竟还是有点大的。

不足之处在于： 使用智能指针会要求用户也得使用智能指针，非必要不应该提出这种约束; 使用锁也有开销; 同时代码量也增多了，实现上我们希望越简单越好。

2.2.3 最推荐的懒汉式单例(magic static )——局部静态变量

class Singleton {

private:

Singleton()

{

std::cout << "constructor called" << std::endl;

}

Singleton(Singleton&)=delete;

Singleton& operator=(Singleton&) = delete;

public:

~Singleton()

{

std::cout << "destructor called" << std::endl;

}

static Singleton& get\_instance()

{

static Singleton instance;

return instance;

}

};

int main()

{

Singleton &m = Singleton::get\_instance();

Singleton &n = Singleton::get\_instance();

std::cout << "Hello World!\n";

}

这种方法又叫做 Meyers' SingletonMeyer's的单例， 是著名的写出《Effective C++》系列书籍的作者 Meyers 提出的。所用到的特性是在C++11标准中的Magic Static特性：

If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.

如果当变量在初始化的时候，并发同时进入声明语句，并发线程将会阻塞等待初始化结束。

这样保证了并发线程在获取静态局部变量的时候一定是初始化过的，所以具有线程安全性。

C++静态变量的生存期 是从声明到程序结束，这也是一种懒汉式。

这是最推荐的一种单例实现方式：

通过局部静态变量的特性保证了线程安全 (C++11, GCC > 4.3, VS2015支持该特性);

不需要使用共享指针，代码简洁；

注意在使用的时候需要声明单例的引用 Single& 才能获取对象。

另外网上有人的实现返回指针而不是返回引用

static Singleton\* get\_instance(){

static Singleton instance;

return &instance;

}

这样做并不好，理由主要是无法避免用户使用delete instance导致对象被提前销毁。还是建议大家使用返回引用的方式。

3.单例的模板

3.1 CRTP 奇异递归模板模式实现

template<typename T>

class Singleton {

public:

static T& get\_instance() {

static T instance;

return instance;

}

virtual ~Singleton() {

std::cout << "destructor called!" << std::endl;

}

Singleton(const Singleton&) = delete;

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

protected:

Singleton() {

std::cout << "constructor called!" << std::endl;

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Example:

// 1.friend class declaration is requiered!

// 2.constructor should be private

class DerivedSingle :public Singleton<DerivedSingle> {

// !!!! attention!!!

// needs to be friend in order to

// access the private constructor/destructor

friend class Singleton<DerivedSingle>;

public:

DerivedSingle(const DerivedSingle&) = delete;

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

private:

DerivedSingle() = default;

};

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

DerivedSingle& instance1 = DerivedSingle::get\_instance();

DerivedSingle& instance2 = DerivedSingle::get\_instance();

return 0;

}

以上实现一个单例的模板基类，使用方法如例子所示意，子类需要将自己作为模板参数T 传递给 Singleton<T> 模板; 同时需要将基类声明为友元，这样才能调用子类的私有构造函数。

基类模板的实现要点是：

构造函数需要是 protected，这样子类才能继承；

使用了奇异递归模板模式CRTP(Curiously recurring template pattern)

get instance 方法和 2.2.3 的static local方法一个原理。

在这里基类的析构函数可以不需要 virtual ，因为子类在应用中只会用 Derived 类型，保证了析构时和构造时的类型一致。

3.2

3.2.1奇异递归模板模式（curiously recurring template pattern，CRTP）是C++模板编程时的一种惯用法（idiom）：把派生类作为基类的模板参数。更一般地被称作F-bound polymorphism，是一类F 界量化.

3.2.2

CRTP的特点

继承自模板类；

使用派生类作为模板参数特化基类；

template <typename T>

class Base

{ ...

};

// use the derived class itself as a template parameter of the base class

class Derived : public Base<Derived>

{

...

};

这样做的目的是在基类中使用派生类，从基类的角度来看，派生类其实也是基类，通过向下转换[downcast],因此，基类可以通过static\_cast把其转换到派生类，从而使用派生类的成员，形式如下：

template <typename T>

class Base

{

public:

void doWhat()

{

T& derived = static\_cast<T&>(\*this);

// use derived...

}

};