Design Pattern - 单例模式

单例模式概念

> 指一个类只有一个实例，且该类能自行创建这个实例的一种模式。

优缺点

- 单例模式的优点：
  - 单例模式可以保证内存里只有一个实例，减少了内存的开销。
  - 可以避免对资源的多重占用。
  - 单例模式设置全局访问点，可以优化和共享资源的访问。
- 单例模式的缺点：
  - 单例模式一般没有接口，扩展困难。如果要扩展，则除了修改原来的代码，没有第二种途径，违背开闭原则。
  - 在并发测试中，单例模式不利于代码调试。在调试过程中，如果单例中的代码没有执行完，也不能模拟生成一个新的对象。
  - 单例模式的功能代码通常写在一个类中，如果功能设计不合理，则很容易违背单一职责原则。

应用场景

> 对于 Java 来说，单例模式可以保证在一个 JVM 中只存在单一实例。单例模式的应用场景主要有以下几个方面。

1. 需要频繁创建的一些类，使用单例可以降低系统的内存压力，减少 GC。
2. 某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等。
3. 某些类创建实例时占用资源较多，或实例化耗时较长，且经常使用。
4. 某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。
5. 频繁访问数据库或文件的对象。
6. 对于一些控制硬件级别的操作，或者从系统上来讲应当是单一控制逻辑的操作，如果有多个实例，则系统会完全乱套。
7. 当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

懒汉式实现

该模式的特点是类加载时没有生成单例，只有当第一次调用 getInstance() 方法时才去创建这个单例。

1. 默认构造函数是私有的，外部不能进行单例类的实例化；
2. 拷贝构造函数和赋值运算符也是私有的，以禁止拷贝和赋值；
3. 具有一个私有的静态成员指针 instance_，指向唯一的实例；
4. 提供一个公有的静态成员函数用于返回实例，如果实例为NULL，则进行实例化。

```
//Singleton.h
#pragma once

class Singleton {
public:
    static Singleton* getInstance();

private:
```

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

    static Singleton* instance_;
};

//Singleton.cpp
#include <iostream>
#include "Singleton.h"

Singleton* Singleton::instance_ = NULL;

Singleton::Singleton() {}

Singleton::Singleton(const Singleton&) {}

Singleton &Singleton::operator=(const Singleton&) {}

Singleton *Singleton::getInstance() {
    if (NULL == instance_) {
        instance_ = new Singleton();
    }
    return instance_;
}
```

饿汉式实现

与懒汉式单例模式不同之处是，在全局作用域进行单例类的实例化，并用此实例初始化单例类的静态成员指针instance_。

```
//Singleton.h
#pragma once

class Singleton {
public:
    static Singleton* getInstance();

private:
    Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);

    static Singleton* instance_;
};

//Singleton.cpp
#include <iostream>
#include "Singleton.h"

Singleton* Singleton::instance_ = new Singleton();

Singleton::Singleton() {}

Singleton::Singleton(const Singleton&) {}

Singleton &Singleton::operator=(const Singleton&) {}
```

```
Singleton *Singleton::getInstance() {
    return instance_;
}
```

线程安全问题

懒汉式：如果有两个线程同时获取单例类的实例，都发现实例不存在，因此都会进行实例化，就会产生两个实例都要赋值给instance_，这是严重的错误。为了解决这个问题，就要考虑加锁。

线程安全的懒汉式，修改获取实例的方法如下：

```
Singleton *Singleton::getInstance() {
    lock();        //上锁
    if (NULL == instance_) {
        instance_ = new Singleton();
    }
    unlock();

    return instance_;
}
```

但这个获取实例的方法存在性能问题，每次获取实例的时候都要先上锁，之后再解锁，如果有很多线程的话，可能会造成大量线程的阻塞。改进后的实现如下：

```
Singleton *Singleton::getInstance() {
    if (NULL == instance_) {
        lock();        //上锁
        if (NULL == instance_) {
            instance_ = new Singleton();
        }
        unlock();
    }

    return instance_;
}
```

绝大多数情况下，获取实例时都是直接返回实例，这时候不会涉及到上锁、解锁的问题。只有在没有实例的时候，才会涉及上锁、解锁，这种情况是很少的。这个获取实例的方法对性能几乎无影响。

饿汉式：程序运行初期就进行了单例类实例化，不存在上述的线程安全问题。

对象内存释放问题

1. 人为在程序结束之前，调用delete来释放
    - 析构函数内部来 delete.

```
~Singleton()
{
   delete instance_;
}
```

- 问题一：new出来的对象，必须用与之对应的delete显示的来释放，程序并不会自动调用析构函数来析构new出来的对象；
- 问题二：在delete的时候会调用析构函数，析构函数中又调用了delete，然后又调用了析构函数……这样就进入了一个无限的循环之中。
- 改进：

```
int main(int argc, char ** argv)
{
    //...

    delete Singleton::get_instance();
    //...
}
```

2. 通过C标准库的 atexit() 函数注册释放函数

   atexit()函数可以用来注册终止函数。如果打算在main()结束后执行某些操作，可以使用该函数来注册相关函数。

```
void del_singleton_01()
{
    if (Singleton::get_instance())
    {
        delete Singleton::get_instance();
    }
}

int main(int argc, char **argv)
{
    // ...
    atexit(del_singleton_01);
    // ...
}
```

   标准规定atexit()至少可以注册32个终止函数，如果系统中有多个单例，我们可能要注册多个函数，或者在同一个终止函数中释放所有单例对象。但是方式一中的问题依然存在。必须由认为手工注册，且有可能遗漏某个对象。

3. 由单例类提供释放接口

   与方法一相似：

```
class Singleton {
public:
    // ...
    void del_object() {
        if (instance_) {
            delete instance_;
            instance_ = nullptr;
        }
    }
    // ...
};
```

```
int main(int argc, char ** argv)
{
    // ...
    Singleton::get_instance()->del_object();
    // ...
}
```

4. 让操作系统自动释放

   进程结束时，静态对象的生命周期随之结束，其析构函数会被调用来释放对象。利用这一特性，在单例类中声明一个内嵌类，该类的析构函数专门用来释放new出来的单例对象，并声明一个该类类型的static对象。

```
class Singleton {
public:
    // ...
private:
    // ...
    static Singleton * instance_;

    class GarbageCollector {
    public:
        ~GarbageCollector() {
            if (Singleton::instance_) {
                delete Singleton::instance_;
                Singleton::instance_ = nullptr;
            }
        }
    };
    static GarbageCollector gc;
};

// 定义
Singleton::GarbargeCollector Singleton::gc;
// ...
```

5. 另:

   之所以要进行内存的释放，是因为在单例的实现过程中，我们使用了new来创建对象。如果在实现过程中，不使用new，而是使用静态[局部]对象的方式，就不存在释放对象的问题了。

```
class Singleton {
    // ...
    static Singleton instance;
    // ...
};

// ...
Singleton Singleton::instance;
// ...
```