【设计模式】可能是东半球最透彻的单例模式讲解

GeeksPlanets 陈同学在搬砖 2020-01-01 20:57

单例模式的应用场景

设想咱们现在在写一个用户界面程序
要实现的功能是点击一个按钮
创建一个窗口
实现的方法也很简单 在按钮的回调函数里面 new一个窗口类

存在问题1:

如果我们现在要求点击按钮
只能生成一个窗口(无论后面再点击多少次 就只能生成一个窗口) ,
这样上面的方式就不可行了 因为上面的实现方式 你点击了几次按钮
就会生成几个窗口,

改进路子:

于是 我们的解决方案就是
1.在按钮回调函数外面把窗口类声明全局对象
但是不new 这样的话声明出来的这个窗口类对象就是null的

2.在按钮回调函数调用new构造窗口类对象之前 对其进行判断是否为null
如果是null 说明这个窗口类对象还没有new过 也就目前还没生成窗口
故这就调用new给他创建这个窗口
如果不是null 说明这个窗口类对象已经new过 创建出来了
也就是已经有一个窗口了 故就跳过不再创建窗口

存在问题2:

这样子问题好像已经解决了 但是我们还想让这个功能更完美一点
还存在的一个小问题是
就是每次按钮类回调函数去创建窗口类对象的时候
总得让按钮类回调函数里面调用if ,else去判断当前窗口类对象的唯一性
这样其实是不利于代码封装和代码维护的

改进路子:

于是我们就想能不能对窗口类本身进行改进
让他能够自己去保证自己创建对象时候的唯一性,
这样就不用每次在按钮类回调函数里面煞费苦心的
去判断当前是不是只创建了一个窗口类对象

总结:

总结一下就是
有时候我们想让一个类能够保证自己仅仅能生成一个实例化对象
(对应上面的窗口类去保证自己创建对象时候的唯一性)

并提供一个访问它的全局访问点，该实例被所有程序模块共享。
(对应着上面按钮类在回调函数里面访问窗口类对象)

单例模式的设计思路

应该怎么保证上面窗口类在按钮类的回调函数里面调用的时候
能够 不用if else判断 而由窗口类对象自己去保证唯一性呢?

把大象装冰箱 分三步
第一步
首先应该不能让外界调用new的方式去创建他的对象 因为如果这样的话
外部还是可以产生多个对象了。
我们还是得引入`if else`来判断 ,
所以我们就应该阻止其他程序建立对象,
阻止的方式就是把构造函数声明为私有

第二步
其次应该让这个窗口类对象应在作为静态成员对象声明在自己类的内部声明
因为这样的话 我们就可以在类里面实现对这个对象的唯一性判断
(由于它是静态的 各个类对象之间会共有这一个成员对象)

第三步
最后 应该在判断完这个静态成员类对象唯一以后
应该有一个全局唯一接口(静态成员函数)让外面访问这个对象

总结

总结一下实现步骤：
1.私有化构造函数，不让其他程序创建的对象初始化。
2.在本类中new一个静态本类对象。
3.定义一个静态成员函数，
它的功能是让其他程序可以通过这个函数获取到本类的对象。

单例模式的代码实现

介绍

有了上面的理论讲解 ,对应这上面的步骤来看看代码该怎么写
有两种写法 观察一下下面两种写法有什么不同

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

```
class Singleton
{
//1.声明构造函数为私有
private:
  Singleton(){};
//2.声明一个本类的对象
private:
  static Singleton* instance;
//3.对外提供本类对象的公共接口
public:
  static Singleton* getInstance()
{

    return instance;
  }
};
Singleton* Singleton::instance=new Singleton;
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
class Singleton
{
//1.声明构造函数为私有
private:
  Singleton(){};
//2.声明一个本类的对象
private:
  static Singleton* instance;
//3.对外提供本类对象的公共接口
public:
  static Singleton* getInstance()
{
    if(instance==NULL)
    {
       instance=new Singleton;
    }

    return instance;
  }
};
Singleton* Singleton::instance=NULL;
```

仔细观察上面两种写法 你会发现两者最大的不同就在于

第一种 成员静态对象是声明的时候就new好了
第二种 成员静态对象在声明的时候没有new好,
而是到外面要调用这个对象的时候 也就是调用getInstance的时候才new好了

第一种叫饿汉模式 也就是在类加载的时候就饿了,非要创建实例,
不考虑创建对象的内存开销

第二种

叫懒汉模式 也就是,类加载时绝对不会创建实例的,
只有在需要使用的时候才创建实例对象.

总结一下就是
饿汉式：我很饿，我一上来就要new个对象！
懒汉式：我先不new对象，等我干活的时候，我再去new

饿汉式

```
1
2
3
```

```
class Singleton
{
//1.声明构造函数为私有
private:
  Singleton(){};
//2.声明一个本类的对象
private:
  static Singleton* instance;
//3.对外提供本类对象的公共接口
public:
  static Singleton* getInstance()
{

    return instance;
  }
};
Singleton* Singleton::instance=new Singleton;
拖曳以移動
```

饿汉式有两大缺点:

1.资源的占用:
饿汉模式 在类创建的就得new好它的静态成员对象 故占用空间

2.类中静态成员 初始化顺序不确定所导致的异常.
也就是说类中的两个静态成员
getInstance和Instance不同编译单元中的初始化顺序是未定义的,
如果在Instance初始化完成之前调用
getInstance() 方法会返回一个未定义的实例

懒汉式

```
class Singleton
{
//1.声明构造函数为私有
private:
  Singleton(){};
//2.声明一个本类的对象
private:
  static Singleton* instance;
//3.对外提供本类对象的公共接口
public:
  static Singleton* getInstance()
{
    if(instance==NULL)
    {
       instance=new Singleton;
    }

    return instance;
  }
};
Singleton* Singleton::instance=NULL;
```

对于懒汉式,也存在两个问题

1.内存泄露

①析构函数没有被执行：
程序退出时, 析构函数没被执行.
这在某些设计不可靠的系统上会导致资源泄漏,
想一想上面的Instance指向的空间什么时候释放呢?
更严重的问题是，该实例的析构函数什么时候执行?
如果在类的析构行为中有必须的操作,
比如关闭文件，释放外部资源，那么上面的代码无法实现这个要求。
我们需要一种方法，正常的删除该实例。

2.线程安全

②线程不安全：我们注意到在 static Singleton* getInstance() 方法中,
是通过 if 语句判断 静态实例变量 是否被初始化来觉得是否进行初始化,
那么在多线程中就有可能出现多次初始化的问题。
比方说，有两个多线程同时进入到这个方法中,
同时执行 if 语句的判断,
那么就会出现两次两次初始化静态实例变量的情况。

懒汉式改进----用嵌套类解决内存泄漏

对于上面的第一个问题
一种方案是可以在程序结束时调用GetInstance(),
并对返回的指针掉用delete操作。这样做可以实现功能,
但不仅很丑陋，而且容易出错。
因为这样的附加代码很容易被忘记,
而且也很难保证在delete之后，没有代码再调用GetInstance函数。

一个妥善的方法是让这个类自己知道在合适的时候把自己删除,
或者说把删除自己的操作挂在操作系统中的某个合适的点上,
使其在恰当的时候被自动执行。
我们知道，程序在结束的时候，系统会自动析构所有的全局变量。
事实上，系统也会析构所有的类的静态成员变量,
就像这些静态成员也是全局变量一样。
利用这个特征，我们可以在单例类中定义一个这样的静态成员的嵌套类,
而它的唯一工作就是在析构函数中删除单例类的实例

```
1
2
3
4
5
```

```
class Singleton
{
private:
  static Singleton* instance;
private:
  Singleton() { };
  ~Singleton() { };
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
private:
  class Deletor {
  public:
    ~Deletor() {
      if(Singleton::instance != NULL)
        delete Singleton::instance;
    }
  };
  static Deletor deletor;
public:
  static Singleton* getInstance() {
    if(instance == NULL) {
      instance = new Singleton();
    }
    return instance;
  }
};

// init static member
Singleton* Singleton::instance = NULL;
```

懒汉式的改进-----用双重检测锁DLC解决线程安全

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
class singleton
{
protected:
    singleton()
    {
        pthread_mutex_init(&mutex);
    }
private:
    static singleton* p;
public:
    static pthread_mutex_t mutex;
    static singleton* initance();
};

pthread_mutex_t singleton::mutex;
singleton* singleton::p = NULL;
singleton* singleton::initance()
{
    if (p == NULL)
    {
        pthread_mutex_lock(&mutex);
        if (p == NULL)
            p = new singleton();
        pthread_mutex_unlock(&mutex);
    }
    return p;
}
```

拖曳以移動

这里面的关键是为什么要在getinstance里面作两次if(p==NULL)判断呢?
<大话设计模式>中讲解的很清楚

懒汉式的改进-----用内部静态变量解决线程安全

```
class singleton
{
protected:
    singleton()
    {
        pthread_mutex_init(&mutex);
    }
public:
    static pthread_mutex_t mutex;
    static singleton* getInstance();
    int a;
};

pthread_mutex_t singleton::mutex;
singleton* singleton::getInstance()
{
    pthread_mutex_lock(&mutex);
    static singleton obj;
    pthread_mutex_unlock(&mutex);
    return &obj;
}
```

在getInstance函数里定义一个静态的实例，
也可以保证拥有唯一实例，在返回时只需要返回其指针就可以了。
推荐这种实现方法，真得非常简单。

后记

要知道单例模式更完美的一个实现可以参考一下boost库中单例模式的一个实现