单例模式 - 巴基速递

发表于 2022-09-01 | 更新于 2023-04-06 | 设计模式

| 字数总计: 4k | 阅读时长: 13 分钟 | 阅读量: 3357 | 评论数: 4

配套视频课程已更新完毕，大家可通过以下两种方式观看视频讲解：

关注公众号： 爱编程的大丙 ，或者进入 大丙课堂 学习。

苏丙榅

合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。

1. 巴基的订单

在海贼世界中，巴基速递是巴基依靠手下强大的越狱犯兵力，组建的集团海贼派遣公司，它的主要业务是向世界有需要的地方输送雇佣兵（其实是不干好事儿）。

自从从 特拉法尔加罗 和 路飞 同盟击败了 堂吉诃德家族 ，战争的市场对雇佣兵的依赖越来越大。订单便源源不断的来了。此时我们来分析一个问题：巴基是怎么接单并且派单的呢？

简单来说，巴基肯定是有一个账本用于记录下单者信息，下单者的需求以及下单的时间，然后根据下单的先后顺序选择合适的人手进行派单。从程序猿的视角可以这样认为，这个账本其实就相当于一个任务队列：

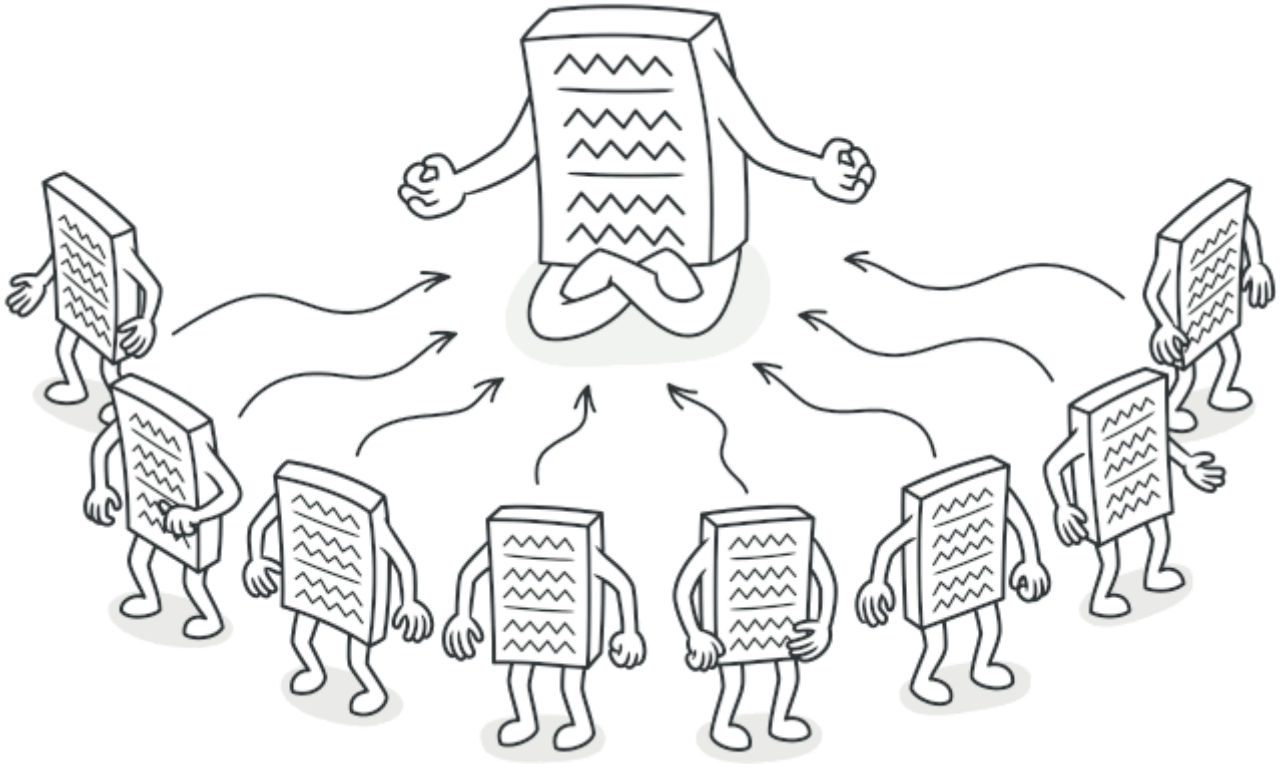
- 有一定的容量，可以存储任务

目录

- 按照下单的先后顺序存储并处理任务 – 典型的队列特性： 先进先出

对于巴基来说把所有的订单全部记录到一个账本上就够了，如果将其平移到项目中，也就意味着应用程序在运行过程中存储任务的任务队列一个足矣，弄太多反而冗余，不太好处理了。

在一个项目中，全局范围内，某个类的实例有且仅有一个，通过这个唯一实例向其他模块提供数据的全局访问，这种模式就叫单例模式。单例模式的典型应用就是任务队列。



2. 独生子女

最新文章

CMake 保姆级教程（下）
2023-03-15

CMake 保姆级教程（上）
2023-03-06

访问者模式 - 再见，香波地群岛
2022-09-22

模板方法模式 - 和平主义者
2022-09-21

状态模式 - 文斯莫克・山治
2022-09-20

如果使用单例模式，首先要保证这个类的实例有且仅有一个，也就是说这个对象是独生子女，如果我们实施计划生育只生一个孩子，不需要也不能给再他增加兄弟姐妹。因此，就必须采取一系列的防护措施。对于类来说以上描述同样适用。涉及一个类多对象操作的函数有以下几个：

- 构造函数： 创建一个新的对象
- 拷贝构造函数： 根据已有对象拷贝出一个新的对象
- 拷贝赋值操作符重载函数： 两个对象之间的赋值

为了把一个类可以实例化多个对象的路堵死，可以做如下处理：

1. 构造函数私有化，在类内部只调用一次，这个是可控的。
   - 由于使用者在类外部不能使用构造函数，所以在类内部创建的这个唯一的对象必须是静态的，这样就可以通过类名来访问了，为了不破坏类的封装，我们都会把这个静态对象的访问权限设置为私有的。
   - 在类中只有它的静态成员函数才能访问其静态成员变量，所以可以给这个单例类提供一个静态函数用于得到这个静态的单例对象。
2. 拷贝构造函数私有化或者禁用（使用 `= delete`）
3. 拷贝赋值操作符重载函数私有化或者禁用（从单例的语义上讲这个函数已经毫无意义，所以在类中不再提供这样一个函数，故将它也一并处理一下。）

由于单例模式就是给类创建一个唯一的实例对象，所以它的 UML 类图是很简单的：

因此，定义一个单例模式的类的示例代码如下：

C++

```cpp
// 定义一个单例模式的类
class Singleton
{
public:
    // = delete 代表函数禁用，也可以将其访问权限设置为私有
    Singleton(const Singleton& obj) = delete;
    Singleton& operator=(const Singleton& obj) = delete;
    static Singleton* getInstance();
private:
    Singleton() = default;
    static Singleton* m_obj;
};
```

在实现一个单例模式的类的时候，有两种处理模式：

- 饿汉模式

3. 饿汉模式

饿汉模式就是在类加载的时候立刻进行实例化，这样就得到了一个唯一的可用对象。关于这个饿汉模式的类的定义如下:

C++

```cpp
// 饿汉模式
class TaskQueue
{
public:
    // = delete 代表函数禁用, 也可以将其访问权限设置为私有
    TaskQueue(const TaskQueue& obj) = delete;
    TaskQueue& operator=(const TaskQueue& obj) = delete;
    static TaskQueue* getInstance()
    {
        return m_taskQ;
    }
private:
    TaskQueue() = default;
    static TaskQueue* m_taskQ;
};
// 静态成员初始化放到类外部处理
TaskQueue* TaskQueue::m_taskQ = new TaskQueue;

int main()
{
```

```
21      TaskQueue* obj = TaskQueue::getInstance();
22  }
```

在 第17行，定义这个单例类的时候，就把这个静态的单例对象创建出来了。当使用者通过 `getInstance()` 获取这个单例对象的时候，它已经被准备好了。

> 注意事项：类的静态成员变量在使用之前必须在类的外部进行初始化才能使用。

4. 懒汉模式

懒汉模式是在类加载的时候不去创建这个唯一的实例，而是在需要使用的时候再进行实例化。

4.1 懒汉模式类的定义

C++

```
1   // 懒汉模式
2   class TaskQueue
3   {
4   public:
5       // = delete 代表函数禁用, 也可以将其访问权限设置为私有
6       TaskQueue(const TaskQueue& obj) = delete;
7       TaskQueue& operator=(const TaskQueue& obj) = delete;
8       static TaskQueue* getInstance()
9       {
10          if(m_taskQ == nullptr)
11          {
```

```
            m_taskQ = new TaskQueue;
        }
        return m_taskQ;
    }
private:
    TaskQueue() = default;
    static TaskQueue* m_taskQ;
};
TaskQueue* TaskQueue::m_taskQ = nullptr;
```

在调用 `getInstance()` 函数获取单例对象的时候，如果在单线程情况下是没有什么问题的，如果是多个线程，调用这个函数去访问单例对象就有问题了。假设有三个线程 同时执行了 `getInstance()` 函数，在这个函数内部每个线程都会 `new` 出一个实例对象。此时，这个任务队列类的实例对象不是一个而是 3 个，很显然这与单例模式的定义是相悖的。

4.2 线程安全问题

双重检查锁定

对于饿汉模式是没有线程安全问题的，在这种模式下访问单例对象的时候，这个对象已经被创建出来了。 要解决懒汉模式的线程安全问题，最常用的解决方案就是使用互斥锁。可以将创建单例对象的代码使用互斥锁锁住，处理代码如下：

C++

```
class TaskQueue
{
public:
```

```
4       // = delete 代表函数禁用，也可以将其访问权限设置为私有
5       TaskQueue(const TaskQueue& obj) = delete;
6       TaskQueue& operator=(const TaskQueue& obj) = delete;
7       static TaskQueue* getInstance()
8       {
9           m_mutex.lock();
10          if (m_taskQ == nullptr)
11          {
12              m_taskQ = new TaskQueue;
13          }
14          m_mutex.unlock();
15          return m_taskQ;
16      }
17  private:
18      TaskQueue() = default;
19      static TaskQueue* m_taskQ;
20      static mutex m_mutex;
21  };
22  TaskQueue* TaskQueue::m_taskQ = nullptr;
23  mutex TaskQueue::m_mutex;
```

在上面代码的 10~13 行这个代码块被互斥锁锁住了，也就意味着不论有多少个线程，同时执行这个代码块的线程只能是一个（相当于是严重限行了，在重负载情况下，可能导致响应缓慢）。我们可以将代码再优化一下：

C++

```
1   class TaskQueue
2   {
3   public:
4       // = delete 代表函数禁用，也可以将其访问权限设置为私有
```

```
5       TaskQueue(const TaskQueue& obj) = delete;
6       TaskQueue& operator=(const TaskQueue& obj) = delete;
7       static TaskQueue* getInstance()
8       {
9           if (m_taskQ == nullptr)
10          {
11              m_mutex.lock();
12              if (m_taskQ == nullptr)
13              {
14                  m_taskQ = new TaskQueue;
15              }
16              m_mutex.unlock();
17          }
18          return m_taskQ;
19      }
20  private:
21      TaskQueue() = default;
22      static TaskQueue* m_taskQ;
23      static mutex m_mutex;
24  };
25  TaskQueue* TaskQueue::m_taskQ = nullptr;
26  mutex TaskQueue::m_mutex;
```

改进的思路就是在加锁、解锁的代码块外层有添加了一个 `if判断` （第 9 行），这样当任务队列的实例被创建出来之后，访问这个对象的线程就不会再执行加锁和解锁操作了（只要有了单例类的实例对象，限行就解除了），对于第一次创建单例对象的时候线程之间还是具有竞争关系，被互斥锁阻塞。上面这种通过 两个嵌套的 `if` 来判断单例对象是否为空的操作就叫做双重检查锁定。

双重检查锁定的问题

假设有两个线程 A、B，当线程 A 执行到 第 8 行 时在线程 A 中 TaskQueue 实例对象 被创建，并赋值给 m_taskQ 。

C++

```
1  static TaskQueue* getInstance()
2  {
3      if (m_taskQ == nullptr)
4      {
5          m_mutex.lock();
6          if (m_taskQ == nullptr)
7          {
8              m_taskQ = new TaskQueue;
9          }
10         m_mutex.unlock();
11     }
12     return m_taskQ;
13 }
```

但是实际上 m_taskQ = new TaskQueue; 在执行过程中 对应的机器指令可能会被重新排序 。正常过程如下：

- 第一步：分配内存用于保存 TaskQueue 对象。
- 第二步：在分配的内存中构造一个 TaskQueue 对象（初始化内存）。
- 第三步：使用 m_taskQ 指针指向分配的内存。

但是被重新排序以后执行顺序可能会变成这样：

- 第一步：分配内存用于保存 TaskQueue 对象。
- 第二步：使用 m_taskQ 指针指向分配的内存。
- 第三步：在分配的内存中构造一个 TaskQueue 对象（初始化内存）。

这样重排序并不影响单线程的执行结果，但是在多线程中就会出问题。如果线程 A 按照第二种顺序执行机器指令，执行完前两步之后失去 CPU 时间片被挂起了，此时线程 B 在第 3 行处进行指针判断的时候 `m_taskQ` 指针是不为空的，但这个指针指向的内存却没有被初始化，最后线程 `B` 使用了一个没有被初始化的队列对象就出问题了（出现这种情况是概率问题，需要反复的大量测试问题才可能会出现）。

在 C++11 中引入了原子变量 `atomic`，通过原子变量可以实现一种更安全的懒汉模式的单例，代码如下：

C++

```C++
class TaskQueue
{
public:
    // = delete 代表函数禁用，也可以将其访问权限设置为私有
    TaskQueue(const TaskQueue& obj) = delete;
    TaskQueue& operator=(const TaskQueue& obj) = delete;
    static TaskQueue* getInstance()
    {
        TaskQueue* queue = m_taskQ.load();
        if (queue == nullptr)
        {
            // m_mutex.lock();  // 加锁：方式1
            lock_guard<mutex> locker(m_mutex);  // 加锁：方式2
            queue = m_taskQ.load();
```

```
            if (queue == nullptr)
            {
                queue = new TaskQueue;
                m_taskQ.store(queue);
            }
            // m_mutex.unlock();
        }
        return queue;
    }

    void print()
    {
        cout << "hello, world!!!" << endl;
    }
private:
    TaskQueue() = default;
    static atomic<TaskQueue*> m_taskQ;
    static mutex m_mutex;
};
atomic<TaskQueue*> TaskQueue::m_taskQ;
mutex TaskQueue::m_mutex;

int main()
{
    TaskQueue* queue = TaskQueue::getInstance();
    queue->print();
    return 0;
}
```

上面代码中使用原子变量 `atomic` 的 `store()` 方法来存储单例对象，使用 `load()` 方法来加载单例对象。在原子变量中这两个函数在处理指令的时候默认的原子顺序是 `memory_order_seq_cst`（顺序原子操作 - `sequentially consistent`），使用顺序约束原子操作库，整个函数执行都将保证顺序执行，并且不会出现数据竞态（data races），不足之处就是使用这种方法实现的懒汉模式的单例执行效率更低一些。

静态局部对象

在实现懒汉模式的单例的时候，相较于双重检查锁定模式有一种更简单的实现方法并且不会出现线程安全问题，那就是使用静态局部局部对象，对应的代码实现如下：

C++

```cpp
class TaskQueue
{
public:
    // = delete 代表函数禁用，也可以将其访问权限设置为私有
    TaskQueue(const TaskQueue& obj) = delete;
    TaskQueue& operator=(const TaskQueue& obj) = delete;
    static TaskQueue* getInstance()
    {
        static TaskQueue taskQ;
        return &taskQ;
    }
    void print()
    {
        cout << "hello, world!!!" << endl;
    }

```

```
17  private:
18      TaskQueue() = default;
19  };
20
21  int main()
22  {
23      TaskQueue* queue = TaskQueue::getInstance();
24      queue->print();
25      return 0;
26  }
```

在程序的 第 9、10 行 定义了一个 静态局部队列对象，并且将这个对象作为了唯一的单例实例。 使用这种方式之所以是线程安全的，是因为在 C++11 标准中有如下规定，并且这个操作是在编译时由编译器保证的：

> 如果指令逻辑进入一个未被初始化的声明变量，所有并发执行应当等待该变量完成初始化。

最后总结一下懒汉模式和饿汉模式的区别：

> 懒汉模式的缺点是在创建实例对象的时候有安全问题，但这样可以减少内存的浪费（如果用不到就不去申请内存了）。饿汉模式则相反，在我们不需要这个实例对象的时候，它已经被创建出来，占用了一块内存。对于现在的计算机而言，内存容量都是足够大的，这个缺陷可以被无视。

5. 替巴基写一个任务队列

作为程序猿的我们，如果想给巴基的账本升级成一个应用程序，首要任务就是设计一个单例模式的任务队列，那么就需要赋予这个类一些属性和方法：

1. 属性：
   - 存储任务的容器，这个容器可以选择使用 STL中的队列（queue）
   - 互斥锁，多线程访问的时候用于保护任务队列中的数据
2. 方法：主要是对任务队列中的任务进行操作
   - 任务队列中任务是否为空
   - 往任务队列中添加一个任务
   - 从任务队列中取出一个任务
   - 从任务队列中删除一个任务

根据分析，就可以把这个饿汉模式的任务队列的单例类定义出来了：

C++

```cpp
#include <iostream>
#include <queue>
#include <mutex>
#include <thread>
using namespace std;

class TaskQueue
{
```

```
public:
    // = delete 代表函数禁用, 也可以将其访问权限设置为私有
    TaskQueue(const TaskQueue& obj) = delete;
    TaskQueue& operator=(const TaskQueue& obj) = delete;
    static TaskQueue* getInstance()
    {
        return &m_obj;
    }
    // 任务队列是否为空
    bool isEmpty()
    {
        lock_guard<mutex> locker(m_mutex);
        bool flag = m_taskQ.empty();
        return flag;
    }
    // 添加任务
    void addTask(int data)
    {
        lock_guard<mutex> locker(m_mutex);
        m_taskQ.push(data);
    }
    // 取出一个任务
    int takeTask()
    {
        lock_guard<mutex> locker(m_mutex);
        if (!m_taskQ.empty())
        {
            return m_taskQ.front();
        }
        return -1;
    }
```

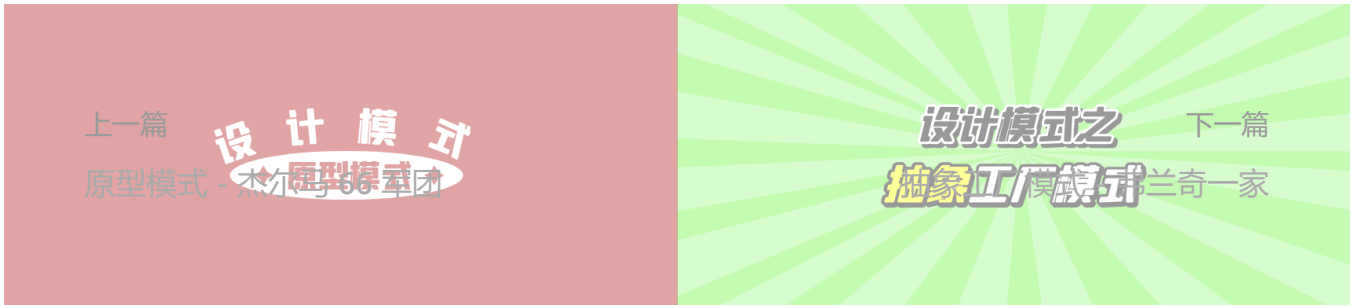
```
40      // 删除一个任务
41      bool popTask()
42      {
```

在上面的程序中有以下几点需要说明一下：

- 正常情况下，任务队列中的任务应该是一个函数指针（这个指针指向的函数中有需要执行的任务动作），此处进行了简化，用一个整形数代替了任务队列中的任务。
- 任务队列中的互斥锁保护的是单例对象的中的数据也就是任务队列中的数据，上面所说的线程安全指的是在创建单例对象的时候要保证这个对象只被创建一次，和此处完全是两码事儿，需要区别看待。
- 关于 ☞ lock_guard 的使用 不懂的可以跳转到这里。
- 在 `main()` 函数中创建了两个子线程
  - 关于 ☞ chrono 库的使用 不懂的可以跳转到这里。
  - 关于 ☞ thread 类的使用 不懂的可以跳转到这里。
  - 关于 ☞ this_thread 命名空间的使用 不懂的可以跳转到这里。
  - `t1线程` 的处理动作是往任务队列中添加任务， `t2线程` 的处理动作是从任务队列中取任务，为了保证能够取出所有的任务，此处需要让 `t2线程` 的执行晚并且慢一些。

设计模式

打赏



相关推荐

评论

昵称 邮箱 网址(http://)

来都来了, 说点什么吧...

提交

4 评论

Anonymous Chrome 110.0.0.0 Windows 11

2023-04-07

回复

直接饿汉模式 YYDS，不初始化的拉出去砍了

Anonymous Chrome 107.0.0.0 Linux

2023-03-13

回复

记得用 g++ 命令时加上 - lpthread

Anonymous Chrome 81.0.4044.145 Android 12

2023-02-13

回复

还得是最后一种方法好，直接静态局部变量简洁方便

Anonymous Chrome 108.0.0.0 Windows 11

2023-01-16

回复

老师，类图的属性 m_obj 和方法 getInstance () 是静态的，应该要有下划线。