单例模式

游戏设计模式 / Design Patterns Revisited

这个章节不同寻常。 其他章节展示如何使用某个设计模式。 这个章节展示如何避免使用某个设计模式。

尽管它的意图是好的，GoF描述的单例模式[GoF]通常弊大于利。 他们强调应该谨慎使用这个模式，但在游戏业界的口口相传中，这一提示经常被无视了。

就像其他模式一样，在不合适的地方使用单例模式就好像用夹板处理子弹伤口。 由于它被滥用得太严重了，这章的大部分都在讲如何回避单例模式， 但首先，让我们看看模式本身。

> 当业界从C语言迁移到面向对象的语言，他们遇到的首个问题是“如何访问实例？”他们知道有要调用的方法，但是找不到实例提供这个方法。 单例（换言之，全局化）是一条简单的解决方案。

单例模式

设计模式 像这样描述单例模式：

> 保证一个类只有一个实例，并且提供了访问该实例的全局访问点。

我们从“并且”那里将句子分为两部分，分别进行考虑。

保证一个类只有一个实例

有时候，如果类存在多个实例就不能正确的运行。 通常发生在类与保存全局状态的外部系统互动时。

考虑封装文件系统的API类。 因为文件操作需要一段时间完成，所以类使用异步操作。 这就意味着可以同时运行多个操作，必须让它们相互协调。 如果一个操作创建文件，另一个操作删除同一文件，封装器类需要同时考虑，保证它们没有相互妨碍。

为了实现这点，对我们封装器类的调用必须接触之前的每个操作。 如果用户可以自由地创建类的实例，这个实例就无法知道另一实例之前的操作。 而单例模式提供的构建类的方式，在编译时保证类只有单一实例。

提供了访问该实例的全局访问点

游戏中的不同系统都会使用文件系统封装类：日志，内容加载，游戏状态保存，等等。 如果这些系统不能创建文件系统封装类的实例，它们如何访问该实例呢？

单例为这点也提供了解决方案。 除了创建单一实例以外，它也提供了一种获得它的全局方法。 使用这种范式，无论何处何人都可以访问实例。 综合起来，经典的实现方案如下：

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // 惰性初始化
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

静态的`instance_`成员保存了一个类的实例， 私有的构造器保证了它是唯一的。 公开的静态方法`instance()`让任何地方的代码都能访问实例。 在首次被请求时，它同样负责惰性实例化该单例。

现代的实现方案看起来是这样的：

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    static FileSystem *instance = new FileSystem();
    return *instance;
  }

private:
  FileSystem() {}
};
```

哪怕是在多线程情况下，C++11标准也保证了本地静态变量只会初始化一次， 因此，假设你有一个现代C++编译器，这段代码是线程安全的，而前面的那个例子不是。

> 当然，单例类本身的线程安全是个不同的问题！这里只保证了它的**初始化**没问题。

为什么我们使用它

看起来已有成效。 文件系统封装类在任何需要的地方都可用，而无需笨重地到处传递。 类本身巧妙地保证了我们不会实例化多个实例而搞砸。它还具有很多其他的优良性质：

- 如果没人用，就不必创建实例。 节约内存和CPU循环总是好的。 由于单例只在第一次被请求时实例化，如果游戏永远不请求，那么它不会被实例化。
- 它在运行时实例化。 通常的替代方案是使用含有静态成员变量的类。 我喜欢简单的解决方案，因此我尽可能使用静态类而不是单例，但是静态成员有个限制：自动初始化。 编译器在`main()`运行前初始化静态变量。 这就意味着不能使用在程序加载时才获取的信息（举个例子，从文件加载的配置）。 这也意味着它们的相互依赖是不可靠的——编译器可不保证以什么样的顺序初始化静态变量。

惰性初始化解决了以上两个问题。 单例会尽可能晚地初始化，所以那时它需要的所有信息都应该可用了。 只要没有环状依赖，一个单例在初始化它自己的时甚至可以引用另一个单例。

- 可继承单例。 这是个很有用但通常被忽视的能力。 假设我们需要跨平台的文件系统封装类。 为了达到这一点，我们需要它变成文件系统抽象出来的接口，而子类为每个平台实现接口。 这是基类：

```
class FileSystem
{
public:
  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void  writeFile(char* path, char* contents) = 0;
};
```

然后为一堆平台定义子类：

```
class PS3FileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // 使用索尼的文件读写API……
  }

  virtual void writeFile(char* path, char* contents)
  {
    // 使用索尼的文件读写API……
  }
};

class WiiFileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // 使用任天堂的文件读写API……
  }

  virtual void writeFile(char* path, char* contents)
  {
    // 使用任天堂的文件读写API……
  }
};
```

下一步，我们把`FileSystem`变成单例：

```
class FileSystem
{
public:
  static FileSystem& instance();

  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void  writeFile(char* path, char* contents) = 0;

protected:
  FileSystem() {}
};
```

灵巧之处在于如何创建实例：

```
FileSystem& FileSystem::instance()
{
  #if PLATFORM == PLAYSTATION3
    static FileSystem *instance = new PS3FileSystem();
  #elif PLATFORM == WII
    static FileSystem *instance = new WiiFileSystem();
  #endif

  return *instance;
}
```

通过一个简单的编译器转换，我们把文件系统包装类绑定到合适的具体类型上。 整个代码库都可以使用`FileSystem::instance()`接触到文件系统，而无需和任何平台相关的代码耦合。耦合发生在为特定平台写的`FileSystem`类实现文件中。

大多数人解决问题到这个程度就已经够了。 我们得到了一个文件系统封装类。 它工作可靠，它全局有效，只要请求就能获取。 是时候提交代码，开怀畅饮了。

为什么我们后悔使用它

短期来看，单例模式是相对良性的。 就像其他设计决策一样，我们需要从长期考虑。 这里是一旦我们将一些不必要的单例写进代码，会给自己带来的麻烦：

它是一个全局变量

当游戏还是由几个家伙在车库中完成时，榨干硬件性能比象牙塔里的软件工程原则更重要。C语言和汇编程序员前辈能毫无问题地使用全局变量和静态变量，发布好游戏。 但随着游戏变得越来越大，越来越复杂，架构和管理开始变成瓶颈， 阻碍我们发布游戏的，除了硬件限制，还有生产力限制。

所以我们迁移到了像C++这样的语言， 开始将一些从软件工程师前辈那里学到的智慧应用于实际。 其中一课是全局变量有害的诸多原因：

- 理解代码更加困难。 假设我们在查找其他人所写函数中的漏洞。 如果函数没有碰到任何全局状态，脑子只需围着函数转， 只需搞懂函数和传给函数的变量。

  > 计算机科学家称不接触不修改全局状态的函数为“纯”函数。 纯函数易于理解，易于编译器优化， 易于完成优雅的任务，比如记住缓存的情况并继续上次调用。
  >
  > 完全使用纯函数是有难度的，但其好处足以引诱科学家创造像Haskell这样**只**使用纯函数的语言。

  现在考虑函数中间是个对`SomeClass::getSomeGlobalData()`的调用。为了查明发生了什么，得追踪整个代码库来看看什么修改了全局变量。你真的不需要讨厌全局变量，直到你在凌晨三点使用`grep`搜索数百万行代码，搞清楚哪一个错误的调用将一个静态变量设为了错误的值。

- 促进了耦合的发生。 新加入团队的程序员也许不熟悉你们完美、可维护、松散耦合的游戏架构， 但还是刚刚获得了第一个任务：在岩石撞击地面时播放声音。 你我都知道这不需要将物理和音频代码耦合，但是他只想着把任务完成。 不幸的是，我们的`AudioPlayer`是全局可见的。 所以之后一个小小的`#include`，新队员就打乱了整个精心设计的架构。

如果不用全局实例实现音频播放器，那么哪怕他确实用`#include`包含了头文件，他还是啥也做不了。 这种阻碍给他发送了一个明确的信号，这两个模块不该接触，他需要另辟蹊径。通过控制对实例的访问，你控制了耦合。

- 对并行不友好。 那些在单核CPU上运行游戏的日子已经远去。 哪怕完全不需要并行的优势，现代的代码至少也应考虑在多线程环境下工作。 当我们将某些东西转为全局变量时，我们创建了一块每个线程都能看到并访问的内存， 却不知道其他线程是否正在使用那块内存。 这种方式带来了死锁，竞争状态，以及其他很难解决的线程同步问题。

像这样的问题足够吓阻我们声明全局变量了， 同理单例模式也是一样，但是那还没有告诉我们应该如何设计游戏。 怎样不使用全局变量构建游戏？

有几个对这个问题的答案（这本书的大部分都是由答案构成）， 但是它们并非显而易见。 与此同时，我们得发布游戏。 单例模式看起来是万能药。 它被写进了一本关于面向对象设计模式的书中，因此它肯定是个好的设计模式，对吧？ 况且我们已经借助它做了很多年软件设计了。

不幸的是，它不是解药，它是安慰剂。 如果浏览全局变量造成的问题列表，你会注意到单例模式解决不了其中任何一个。 因为单例确实是全局状态——它只是被封装在一个类中。

它能在你只有一个问题的时候解决两个

在GoF对单例模式的描述中，“并且”这个词有点奇怪。 这个模式解决了一个问题还是两个问题呢？如果我们只有其中一个问题呢？ 保证实例是唯一存在的是很有用的，但是谁告诉我们要让每个人都能接触到它？ 同样，全局接触很方便，但是必须禁止存在多个实例吗？

这两个问题中的后者，便利的访问，几乎是使用单例模式的全部原因。 想想日志类。大部分模块都能从记录诊断日志中获益。 但是，如果将`Log`类的实例传给每个需要这个方法的函数，那就混杂了产生的数据，模糊了代码的意图。

明显的解决方案是让`Log`类成为单例。 每个函数都能从类那里获得一个实例。 但当我们这样做时，我们无意地制造了一个奇怪的小约束。 突然之间，我们不再能创建多个日志记录者了。

起初，这不是一个问题。 我们记录单独的日志文件，所以只需要一个实例。 然后，随着开发周期的逐次循环，我们遇到了麻烦。 每个团队的成员都使用日志记录各自的诊断信息，大量的日志倾泻在文件里。 程序员需要翻过很多页代码来找到他关心的记录。

我们想将日志分散到多个文件中来解决这点。 为了达到这点，我们得为游戏的不同领域创造单独的日志记录者： 网络，UI，声音，游戏，玩法。 但是我们做不到。 `Log`类不再允许我们创建多个实例，而且调用的方式也保证了这一点：

```
Log::instance().write("Some event.");
```

为了让`Log`类支持多个实例（就像它原来的那样）， 我们需要修改类和提及它的每一行代码。 之前便利的访问就不再那么便利了。

> 这可能更糟。想象一下你的`Log`类是在多个**游戏**间共享的库中。 现在，为了改变设计，需要在多组人之间协调改变， 他们中的大多数既没有时间，也没有动机修复它。

惰性初始化从你那里剥夺了控制权

在拥有虚拟内存和软性性能需求的PC里，惰性初始化是一个小技巧。 游戏则是另一种状况。初始化系统需要消耗时间：分配内存，加载资源，等等。 如果初始化音频系统消耗了

几百个毫秒，我们需要控制它何时发生。 如果在第一次声音播放时惰性初始化它自己，这可能发生在游戏的高潮部分，导致可见的掉帧和断续的游戏体验。

同样，游戏通常需要严格管理在堆上分配的内存来避免碎片。 如果音频系统在初始化时分配到了堆上，我们需要知道初始化在何时发生， 这样我们可以控制内存待在堆的哪里。

> 对象池模式一节中有内存碎片的其他细节。

因为这两个原因，我见到的大多数游戏都不使用惰性初始化。 相反，它们像这样实现单例模式：

```
class FileSystem
{
public:
  static FileSystem& instance() { return instance_; }

private:
  FileSystem() {}

  static FileSystem instance_;
};
```

这解决了惰性初始化问题，但是损失了几个单例确实比原生的全局变量优良的特性。 静态实例中，我们不能使用多态，在静态初始化时，类也必须是可构建的。 我们也不能在不需要这个实例的时候，释放实例所占的内存。

与创建一个单例不同，这里实际上是一个简单的静态类。 这并非坏事，但是如果你需要的是静态类，为什么不完全摆脱`instance()`方法， 直接使用静态函数呢？ 调用`Foo::bar()`比`Foo::instance().bar()`更简单， 也更明确地表明你在处理静态内存。

> 通常使用单例而不是静态类的理由是， 如果你后来决定将静态类改为非静态的，你需要修改每一个调用点。 理论上，用单例就不必那么做，因为你可以将实例传来传去，像普通的实例方法一样使用。
>
> 实践中，我从未见过这种情况。 每个人都在使用`Foo::instance().bar()`。 如果我们将Foo改成非单例，我们还是得修改每一个调用点。 鉴于此，我更喜欢简单的类和简单的调用语法。

那该如何是好

如果我现在达到了目标，你在下次遇到问题使用单例模式之前就会三思而后行。 但是你还是有问题需要解决。你应该使用什么工具呢？ 这取决于你试图做什么，我有一些你可以考虑的选项，但是首先.....

看看你是不是真正地需要类

我在游戏中看到的很多单例类都是“管理器”——那些类存在的意义就是照顾其他对象。 我曾看到一些代码库中，几乎所有类都有管理器： 怪物，怪物管理器，粒子，粒子管理器，声音，声音管理器，管理管理器的管理器。 有时候，它们被叫做“系统”或“引擎”，但是思路还是一样的。

管理器类有时是有用的，但通常它们只是反映出作者对OOP的不熟悉。思考这两个特制的类：

```
class Bullet
{
public:
  int getX() const { return x_; }
  int getY() const { return y_; }

  void setX(int x) { x_ = x; }
  void setY(int y) { y_ = y; }

private:
  int x_, y_;
};

class BulletManager
{
public:
  Bullet* create(int x, int y)
  {
    Bullet* bullet = new Bullet();
    bullet->setX(x);
    bullet->setY(y);

    return bullet;
  }

  bool isOnScreen(Bullet& bullet)
  {
    return bullet.getX() >= 0 &&
           bullet.getX() < SCREEN_WIDTH &&
           bullet.getY() >= 0 &&
           bullet.getY() < SCREEN_HEIGHT;
  }

  void move(Bullet& bullet)
  {
    bullet.setX(bullet.getX() + 5);
  }
};
```

也许这个例子有些蠢，但是我见过很多代码，在剥离了外部的细节后是一样的设计。 如果你看看这个代码，`BulletManager`很自然应是一个单例。 无论如何，任何有`Bullet`的对象都需要管理，而你又需要多少个`BulletManager`实例呢？

事实上，这里的答案是零。 这里是我们如何为管理类解决“单例”问题：

```
class Bullet
{
public:
  Bullet(int x, int y) : x_(x), y_(y) {}

  bool isOnScreen()
  {
    return x_ >= 0 && x_ < SCREEN_WIDTH &&
           y_ >= 0 && y_ < SCREEN_HEIGHT;
  }

  void move() { x_ += 5; }

private:
  int x_, y_;
};
```

好了。没有管理器，也没有问题。 糟糕设计的单例通常会“帮助”另一个类增加代码。 如果可以，把所有的行为都移到单例帮助的类中。 毕竟，OOP就是让对象管理好自己。

但是在管理器之外，还有其他问题我们需要寻求单例模式帮助。 对于每种问题，都有一些后续方案可供参考。

将类限制为单一的实例

这是单例模式帮你解决的一个问题。 就像在文件系统的例子中那样，保证类只有一个实例是很重要的。 但是，这不意味着我们需要提供对实例的公众，全局访问。 我们想要减少某部分代码的公众部分，甚至让它在类中是私有的。 在这些情况下，提供一个全局接触点消弱了整体架构。

> 举个例子，我们也许想把文件系统包在**另一**层抽象中。

我们希望有种方式能保证同事只有一个实例而无需提供全局接触点。 有好几种方法能做到。这是其中之一：

```
class FileSystem
{
public:
  FileSystem()
  {
    assert(!instantiated_);
    instantiated_ = true;
  }

  ~FileSystem() { instantiated_ = false; }

private:
  static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

这个类允许任何人构建它，如果你试图构建超过一个实例，它会断言并失败。 只要正确的代码首先创建了实例，那么就保证了没有其他代码可以接触实例或者创建自己的实例。 这个类保证满足了它关注的单一实例，但是它没有指定类该如何被使用。

> **断言** 函数是一种向你的代码中添加限制的方法。 当assert()被调用时，它计算传入的表达式。 如果结果为true，那么什么都不做，游戏继续。 如果结果为false，它立刻停止游戏。 在debug build时，这通常会启动调试器，或至少打印失败断言所在的文件和行号。
>
> assert()表示，“我断言这个总该是真的。如果不是，那就是漏洞，我想**立刻**停止并处理它。”这使得你可以在代码区域之间定义约束。 如果函数断言它的某个参数不能为NULL，那就是说，“我和调用者定下了协议：传入的参数不会NULL。”
>
> 断言帮助我们在游戏发生预期以外的事时立刻追踪漏洞，而不是等到错误最终显现在用户可见的某些事物上。 它们是代码中的栅栏，围住漏洞，这样漏洞就不能从制造它的代码边逃开。

这个实现的缺点是只在运行时检查并阻止多重实例化。 单例模式正相反，通过类的自然结构，在编译时就能确定实例是单一的。

为了给实例提供方便的访问方法

便利的访问是我们使用单例的一个主要原因。 这让我们在不同地方获取需要的对象更加容易。 这种便利是需要付出代价的——在我们不想要对象的地方，也能轻易地使用。

通用原则是在能完成工作的同时，将变量写得尽可能局部。 对象影响的范围越小，在处理它时，我们需要放在脑子里的东西就越少。 在我们拿起有全局范围影响的单例对象前，先

考虑考虑代码中其他获取对象的方式：

- 传进来。 最简单的解决办法，通常也是最好的，把你需要的对象简单地作为参数传给需要它的函数。 在用其他更加繁杂的方法前，考虑一下这个解决方案。

  > 有些人使用术语“依赖注入”来指代它。不是代码**出来**调用某些全局量来确认依赖，而是依赖通过参数被**传进**到需要它的代码中去。 其他人将“依赖注入”保留为对代码提供更复杂依赖的方法。

  考虑渲染对象的函数。为了渲染，它需要接触一个代表图形设备的对象，管理渲染状态。 将其传给所有渲染函数是很自然的，通常是用一个名字像`context`之类的参数。

  另一方面，有些对象不该在方法的参数列表中出现。 举个例子，处理AI的函数可能也需要写日志文件，但是日志不是它的核心关注点。 看到`Log`出现在它的参数列表中是很奇怪的事情，像这样的情况，我们需要考虑其他的选项。

  > 像日志这样散布在代码库各处的是“横切关注点”(cross-cutting concern)。 小心地处理横切关注点是架构中的持久挑战，特别是在静态类型语言中。
  >
  > 面向切面编程被设计出来应对它们。

- 从基类中获得。 很多游戏架构有浅层但是宽泛的继承层次，通常只有一层深。 举个例子，你也许有`GameObject`基类，每个游戏中的敌人或者对象都继承它。 使用这样的架构，很大一部分游戏代码会存在于这些“子”推导类中。 这就意味着这些类已经有了对同样事物的相同获取方法：它们的`GameObject`基类。 我们可以利用这点：

  ```
  class GameObject
  {
  protected:
    Log& getLog() { return log_; }

  private:
    static Log& log_;
  };

  class Enemy : public GameObject
  {
    void doSomething()
    {
      getLog().write("I can log!");
    }
  };
  ```

  这保证任何`GameObject`之外的代码都不能接触`Log`对象，但是每个派生的实体都确实能使用`getLog()`。 这种使用protected函数，让派生对象使用的模式， 被涵盖在子类沙箱这章中。

  > 这也引出了一个新问题，“GameObject是怎样获得Log实例的？”一个简单的方案是，让基类创建并拥有静态实例。
  >
  > 如果你不想要基类承担这些，你可以提供一个初始化函数传入Log实例， 或使用服务定位器模式找到它。

- 从已经是全局的东西中获取。 移除所有全局状态的目标令人钦佩，但并不实际。 大多数代码库仍有一些全局可用对象，比如一个代表了整个游戏状态的`Game`或`World`对象。

我们可以让现有的全局对象捎带需要的东西，来减少全局变量类的数目。 不让Log，FileSystem和AudioPlayer都变成单例，而是这样做：

```
class Game
{
public:
  static Game& instance() { return instance_; }

  // 设置log_, et. al. ……

  Log&         getLog()         { return *log_; }
  FileSystem&  getFileSystem()  { return *fileSystem_; }
  AudioPlayer& getAudioPlayer() { return *audioPlayer_; }

private:
  static Game instance_;

  Log         *log_;
  FileSystem  *fileSystem_;
  AudioPlayer *audioPlayer_;
};
```

这样，只有Game是全局可见的。 函数可以通过它访问其他系统。

```
Game::instance().getAudioPlayer().play(VERY_LOUD_BANG);
```

> 纯粹主义者会声称这违反了Demeter法则。我则声称这比一大坨单例要好。

如果，稍后，架构被改为支持多个Game实例（可能是为了流处理或者测试），Log，FileSystem，和AudioPlayer都不会被影响到——它们甚至不知道有什么区别。 缺陷是，当然，更多的代码耦合到了Game中。 如果一个类简单地需要播放声音，为了访问音频播放器，上例中仍然需要它知道游戏世界。

我们通过混合方案解决这点。 知道Game的代码可以直接从它那里访问AudioPlayer。 而不知道的代码，我们用上面描述的其他选项来提供AudioPlayer。

- 从服务定位器中获得。 目前为止，我们假设全局类是具体的类，比如Game。 另一种选项是定义一个类，存在的唯一目标就是为对象提供全局访问。 这种常见的模式被称为服务定位器模式，有单独讲它的章节。

单例中还剩下什么

剩下的问题，何处我们应该使用真实的单例模式？ 说实话，我从来没有在游戏中使用全部的GoF模式。 为了保证实例是单一的，我通常简单地使用静态类。 如果这无效，我使用静态标识位，在运行时检测是不是只有一个实例被创建了。

书中还有一些其他章节也许能有所帮助。 子类沙箱模式通过分享状态，给实例以类的访问权限而无需让其全局可用。 服务定位器模式确实让一个对象全局可用，但它给了你如何设置对象的灵活性。