

42 | 单例模式（中）：我为什么不推荐使用单例模式？又有何替代方案？

王争 · 设计模式之美



单例模式（中）

上一节课中，我们通过两个实战案例，讲解了单例模式的一些应用场景，比如，避免资源访问冲突、表示业务概念上的全局唯一类。除此之外，我们还学习了 Java 语言中，单例模式的几种实现方法。如果你熟悉的是其他编程语言，不知道你课后有没有自己去对照着实现一下呢？

尽管单例是一个很常用的设计模式，在实际的开发中，我们也确实经常用到它，但是，有些人认为单例是一种反模式（anti-pattern），并不推荐使用。所以，今天，我就针对这个说法详细地讲讲这几个问题：单例这种设计模式存在哪些问题？为什么会被称为反模式？如果不用单例，该如何表示全局唯一类？有何替代的解决方案？

话不多说，让我们带着这些问题，正式开始今天的学习吧！

单例存在哪些问题？

大部分情况下，我们在项目中使用单例，都是用它来表示一些全局唯一类，比如配置信息类、连接池类、ID 生成器类。单例模式书写简洁、使用方便，在代码中，我们不需要创建对象，

直接通过类似 `IdGenerator.getInstance().getId()` 这样的方法来调用就可以了。但是，这种使用方法有点类似硬编码（hard code），会带来诸多问题。接下来，我们就具体看看到底有哪些问题。

1. 单例对 OOP 特性的支持不友好

我们知道，OOP 的四大特性是封装、抽象、继承、多态。单例这种设计模式对于其中的抽象、继承、多态都支持得不好。为什么这么说呢？我们还是通过 `IdGenerator` 这个例子来讲解。

 复制代码

```
1 public class Order {
2     public void create(...) {
3         //...
4         long id = IdGenerator.getInstance().getId();
5         //...
6     }
7 }
8
9 public class User {
10    public void create(...) {
11        // ...
12        long id = IdGenerator.getInstance().getId();
13        //...
14    }
15 }
```

`IdGenerator` 的使用方式违背了基于接口而非实现的设计原则，也就违背了广义上理解的 OOP 的抽象特性。如果未来某一天，我们希望针对不同的业务采用不同的 ID 生成算法。比如，订单 ID 和用户 ID 采用不同的 ID 生成器来生成。为了应对这个需求变化，我们需要修改所有用到 `IdGenerator` 类的地方，这样代码的改动就会比较大。

 复制代码

```
1 public class Order {
2     public void create(...) {
3         //...
4         long id = IdGenerator.getInstance().getId();
5         // 需要将上面一行代码，替换为下面一行代码
```

```
6     long id = OrderIdGenerator.getIntance().getId();
7     //...
8 }
9 }
10
11 public class User {
12     public void create(...) {
13         // ...
14         long id = IdGenerator.getInstance().getId();
15         // 需要将上面一行代码，替换为下面一行代码
16         long id = UserIdGenerator.getIntance().getId();
17     }
18 }
```

除此之外，单例对继承、多态特性的支持也不友好。这里我之所以会用“不友好”这个词，而非“完全不支持”，是因为从理论上来讲，单例类也可以被继承、也可以实现多态，只是实现起来会非常奇怪，会导致代码的可读性变差。不明白设计意图的人，看到这样的设计，会觉得莫名其妙。所以，一旦你选择将某个类设计成到单例类，也就意味着放弃了继承和多态这两个强有力的面向对象特性，也就相当于损失了可以应对未来需求变化的扩展性。

2. 单例会隐藏类之间的依赖关系

我们知道，代码的可读性非常重要。在阅读代码的时候，我们希望一眼就能看出类与类之间的依赖关系，搞清楚这个类依赖了哪些外部类。

通过构造函数、参数传递等方式声明的类之间的依赖关系，我们通过查看函数的定义，就能很容易识别出来。但是，单例类不需要显示创建、不需要依赖参数传递，在函数中直接调用就可以了。如果代码比较复杂，这种调用关系就会非常隐蔽。在阅读代码的时候，我们就需要仔细查看每个函数的代码实现，才能知道这个类到底依赖了哪些单例类。

3. 单例对代码的扩展性不友好

我们知道，单例类只能有一个对象实例。如果未来某一天，我们需要在代码中创建两个实例或多个实例，那就要对代码有比较大的改动。你可能会说，会有这样的需求吗？既然单例类大部分情况下都用来表示全局类，怎么会需要两个或者多个实例呢？

实际上，这样的需求并不少见。我们拿数据库连接池来举例解释一下。

在系统设计初期，我们觉得系统中只应该有一个数据库连接池，这样能方便我们控制对数据库连接资源的消耗。所以，我们把数据库连接池类设计成了单例类。但之后我们发现，系统中有些 SQL 语句运行得非常慢。这些 SQL 语句在执行的时候，长时间占用数据库连接资源，导致其他 SQL 请求无法响应。为了解决这个问题，我们希望将慢 SQL 与其他 SQL 隔离开来执行。为了实现这样的目的，我们可以在系统中创建两个数据库连接池，慢 SQL 独享一个数据库连接池，其他 SQL 独享另外一个数据库连接池，这样就能避免慢 SQL 影响到其他 SQL 的执行。

如果我们将数据库连接池设计成单例类，显然就无法适应这样的需求变更，也就是说，单例类在某些情况下会影响代码的扩展性、灵活性。所以，数据库连接池、线程池这类的资源池，最好还是不要设计成单例类。实际上，一些开源的数据库连接池、线程池也确实没有设计成单例类。

4. 单例对代码的可测试性不友好

单例模式的使用会影响到代码的可测试性。如果单例类依赖比较重的外部资源，比如 DB，我们在写单元测试的时候，希望能通过 mock 的方式将它替换掉。而单例类这种硬编码式的使用方式，导致无法实现 mock 替换。


除此之外，如果单例类持有成员变量（比如 IdGenerator 中的 id 成员变量），那它实际上相当于一种全局变量，被所有的代码共享。如果这个全局变量是一个可变全局变量，也就是说，它的成员变量是可以被修改的，那我们在编写单元测试的时候，还需要注意不同测试用例之间，修改了单例类中的同一个成员变量的值，从而导致测试结果互相影响的问题。关于这一点，你可以回过头去看下 [第 29 讲](#) 中的“其他常见的 Anti-Patterns：全局变量”那部分的代码示例和讲解。

5. 单例不支持有参数的构造函数

单例不支持有参数的构造函数，比如我们创建一个连接池的单例对象，我们没法通过参数来指定连接池的大小。针对这个问题，我们来看下都有哪些解决方案。


第一种解决思路是：创建完实例之后，再调用 init() 函数传递参数。需要注意的是，我们在使用这个单例类的时候，要先调用 init() 方法，然后才能调用 getInstance() 方法，否则代码会

抛出异常。具体的代码实现如下所示：

 复制代码

```
1 public class Singleton {
2     private static Singleton instance = null;
3     private final int paramA;
4     private final int paramB;
5
6     private Singleton(int paramA, int paramB) {
7         this.paramA = paramA;
8         this.paramB = paramB;
9     }
10
11     public static Singleton getInstance() {
12         if (instance == null) {
13             throw new RuntimeException("Run init() first.");
14         }
15         return instance;
16     }
17
18     public synchronized static Singleton init(int paramA, int paramB) {
19         if (instance != null){
20             throw new RuntimeException("Singleton has been created!");
21         }
22         instance = new Singleton(paramA, paramB);
23         return instance;
24     }
25 }
26
27 Singleton.init(10, 50); // 先init, 再使用
28 Singleton singleton = Singleton.getInstance();
```

第二种解决思路是：将参数放到 getInstance() 方法中。具体的代码实现如下所示：

 复制代码


```
1 public class Singleton {
2     private static Singleton instance = null;
3     private final int paramA;
4     private final int paramB;
5
6     private Singleton(int paramA, int paramB) {
7         this.paramA = paramA;
8         this.paramB = paramB;
```

```

9    }
10
11    public synchronized static Singleton getInstance(int paramA, int paramB) {
12        if (instance == null) {
13            instance = new Singleton(paramA, paramB);
14        }
15        return instance;
16    }
17 }
18
19 Singleton singleton = Singleton.getInstance(10, 50);

```

不知道你有没有发现，上面的代码实现稍微有点问题。如果我们如下两次执行 `getInstance()` 方法，那获取到的 `singleton1` 和 `singleton2` 的 `paramA` 和 `paramB` 都是 10 和 50。也就是说，第二次的参数（20，30）没有起作用，而构建的过程也没有给与提示，这样就会误导用户。这个问题如何解决呢？留给你自己思考，你可以在留言区说说你的解决思路。


 复制代码

```

1 Singleton singleton1 = Singleton.getInstance(10, 50);
2 Singleton singleton2 = Singleton.getInstance(20, 30);

```

第三种解决思路是：将参数放到另外一个全局变量中。具体的代码实现如下。`Config` 是一个存储了 `paramA` 和 `paramB` 值的全局变量。里面的值既可以像下面的代码那样通过静态常量来定义，也可以从配置文件中加载得到。实际上，这种方式是最值得推荐的。

 复制代码

```

1 public class Config {
2     public static final int PARAM_A = 123;
3     public static final int PARAM_B = 245;
4 }
5
6 public class Singleton {
7     private static Singleton instance = null;
8     private final int paramA;
9     private final int paramB;
10
11     private Singleton() {
12         this.paramA = Config.PARAM_A;
13         this.paramB = Config.PARAM_B;


```

```
14     }
15
16     public synchronized static Singleton getInstance() {
17         if (instance == null) {
18             instance = new Singleton();
19         }
20         return instance;
21     }
22 }
```

有何替代解决方案？


刚刚我们提到了单例的很多问题，你可能会说，即便单例有这么多问题，但我不用不行啊。我业务上有表示全局唯一类的需求，如果不用单例，我怎么才能保证这个类的对象全局唯一呢？

为了保证全局唯一，除了使用单例，我们还可以用静态方法来实现。这也是项目开发中经常用到的一种实现思路。比如，上一节课中讲的 ID 唯一递增生成器的例子，用静态方法实现一下，就是下面这个样子：

 复制代码

```
1 // 静态方法实现方式
2 public class IdGenerator {
3     private static AtomicLong id = new AtomicLong(0);
4
5     public static long getId() {
6         return id.incrementAndGet();
7     }
8 }
9 // 使用举例
10 long id = IdGenerator.getId();
```

不过，静态方法这种实现思路，并不能解决我们之前提到的问题。实际上，它比单例更加不灵活，比如，它无法支持延迟加载。我们再来看看有没有其他办法。实际上，单例除了我们之前讲到的使用方法之外，还有另外一种使用方法。具体的代码如下所示：

 复制代码

```
1 // 1. 老的使用方式
2 public demofunction() {
```



```
3    //...
4    long id = IdGenerator.getInstance().getId();
5    //...
6 }
7
8 // 2. 新的使用方式：依赖注入
9 public demofunction(IdGenerator idGenerator) {
10     long id = idGenerator.getId();
11 }
12 // 外部调用demofunction()的时候，传入idGenerator
13 IdGenerator idGenerator = IdGenerator.getInsance();
14 demofunction(idGenerator);
```

基于新的使用方式，我们将单例生成的对象，作为参数传递给函数（也可以通过构造函数传递给类的成员变量），可以解决单例隐藏类之间依赖关系的问题。不过，对于单例存在的其他问题，比如对 OOP 特性、扩展性、可测性不友好等问题，还是无法解决。

所以，如果要完全解决这些问题，我们可能要从根上，寻找其他方式来实现全局唯一类。实际上，类对象的全局唯一性可以通过多种不同的方式来保证。我们既可以通过单例模式来强制保证，也可以通过工厂模式、IOC 容器（比如 Spring IOC 容器）来保证，还可以通过程序员自己来保证（自己在编写代码的时候自己保证不要创建两个类对象）。这就类似 Java 中内存对象的释放由 JVM 来负责，而 C++ 中由程序员自己负责，道理是一样的。

对于替代方案工厂模式、IOC 容器的详细讲解，我们放到后面的章节中讲解。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要掌握的重点内容。

1. 单例存在哪些问题？

单例对 OOP 特性的支持不友好

单例会隐藏类之间的依赖关系

单例对代码的扩展性不友好

单例对代码的可测试性不友好

单例不支持有参数的构造函数


2. 单例有什么替代解决方案？

为了保证全局唯一，除了使用单例，我们还可以用静态方法来实现。不过，静态方法这种实现思路，并不能解决我们之前提到的问题。如果要完全解决这些问题，我们可能要从根上，寻找其他方式来实现全局唯一类了。比如，通过工厂模式、IOC 容器（比如 Spring IOC 容器）来保证，由程序员自己来保证（自己在编写代码的时候自己保证不要创建两个类对象）。

有人把单例当作反模式，主张杜绝在项目中使用。我个人觉得这有点极端。模式没有对错，关键看你怎么用。如果单例类并没有后续扩展的需求，并且不依赖外部系统，那设计成单例类就没有太大问题。对于一些全局的类，我们在其他地方 new 的话，还要在类之间传来传去，不如直接做成单例类，使用起来简洁方便。

课堂讨论

1. 如果项目中已经用了很多单例模式，比如下面这段代码，我们该如何在尽量减少代码改动的情况下，通过重构代码来提高代码的可测试性呢？

 复制代码

```
1 public class Demo {
2     private UserRepo userRepo; // 通过构造哈函数或IOC容器依赖注入
3
4     public boolean validateCachedUser(long userId) {
5         User cachedUser = CacheManager.getInstance().getUser(userId);
6         User actualUser = userRepo.getUser(userId);
7         // 省略核心逻辑：对比cachedUser和actualUser...
8     }
9 }
```

2. 在单例支持参数传递的第二种解决方案中，如果我们两次执行 `getInstance(paramA, paramB)` 方法，第二次传递进去的参数是不生效的，而构建的过程也没有给与提示，这样就会误导用户。这个问题如何解决呢？

```
1 Singleton singleton1 = Singleton.getInstance(10, 50);  
2 Singleton singleton2 = Singleton.getInstance(20, 30);
```

欢迎留言和我分享你的思考和见解。如果有收获，也欢迎你把文章分享给你的朋友。

AI智能总结

单例模式在实际开发中常用，但存在一些问题，有人认为它是一种反模式。单例模式对面向对象编程的特性支持不友好，违背了抽象、继承和多态原则，导致代码扩展性差。此外，它隐藏了类之间的依赖关系，降低了代码可读性，也不友好于代码的扩展性和可测试性。文章提出了替代方案，包括使用静态方法、通过init()函数传递参数、将参数放到getInstance()方法中，以及将参数放到另外一个全局变量中。这些方法能解决单例模式存在的问题，但仍需从根本上寻找其他方式来实现全局唯一类，如工厂模式、IOC容器等。总之，单例模式并非绝对反模式，关键在于合理使用。文章还就如何通过重构代码提高代码的可测试性和解决单例支持参数传递的问题展开了讨论。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (80)

最新 精选



Roger宇

2020-05-15

想问一下老师，所谓两个资源池，慢的请求独占一个的设计，如何知道一个sql请求会快会慢？快与慢更多是在运行后才知道的，已经进去运行了还怎么保证独占呢？除非有机制可以在处理sql请求之前评估可能需要的时间。

作者回复: 根据业务来估算的，有些业务本身就很慢，就判定为慢sql



👍 4



小晏子

2020-02-07

课堂讨论，

1. 把代码“User cachedUser = CacheManager.getInstance().getUser(userId);”单独提取出来做成一个单独的函数，这样这个函数就可以进行mock了，进而方便测试validateCachedUs

er。

2. 可以判断传进来的参数和已经存在的instance里面的两个成员变量的值，如果全部相等，就直接返回已经存在的instance，否则就新建一个instance返回。示例如下：

```
public synchronized static Singleton getInstance(int paramA, int paramB) {  
    if (instance == null) {  
        instance = new Singleton(paramA, paramB);  
    } else if (instance.paramA == paramA && instance.paramB == paramB) {  
        return instance;  
    } else {  
        instance = new Singleton(paramA, paramB);  
    }  
    return instance;  
}
```

共 21 条评论 >

👍 122



J.Smile

2020-02-07

模式没有对错，关键看你怎么用。这句话说的很对，所以其实所谓单例模式的缺点这种说法还是有点牵强！

共 2 条评论 >

👍 81



Richie

2020-03-08

课堂讨论第2点，我认为应该先搞清楚需求，为什么需要在getInstance()方法中加参数，想要达到什么目的？

这里分两种情况讨论一下：

1. 如果的确需要一个全局唯一的类，并且这个类只能被初始化一次，那么应该采用文中提到的第三种解决思路，即将所需参数放到全局的配置文件中，从而避免多次初始化参数被忽略或者抛出运行时异常的问题；
2. 如果是要根据不同参数构造出不同的对象，并且相同参数的对象只被构造一次，那么应该改成在Singleton类中维护一个HashMap，然后每次调用getInstance()方法的时候，根据参数去判断对象是否已经存在了（可以采用双重检测），存在则直接返回，不存在再去创建，然后存储，返回。个人理解，这应该是单例+简单工厂的结合。

共 3 条评论 >

👍 67



李小四

2020-02-22

设计模式_42:

作业

1. 可以把单例的对象以依赖注入的方式传入方法;
2. 第二次调用时, 如果参数发生了变化, 应该抛出异常。

感想

坦白讲, 一直以使用双重检测沾沾自喜。。。现在看来, 要不要使用单例要比使用那种单例的实现方式更需要投入思考。

共 1 条评论 >

👍 31



林子er

2020-04-23

由于单例本身存在的一系列缺点, 而单例一般又都是全局的, 因而一般我们项目中很少直接使用单例, 而是通过容器注入, 让容器充当单例和工厂。有时候我们甚至使用伪单例, 即类本身并不是单例的, 而是通过容器保证单例性, 实际编程中按照约定只通过容器获取该实例。

参数化单例实际中是通过Map解决的, 即同样的参数才返回同一个实例, 不同的参数返回不同的实例, 为了保证实例不会太多, 一般可传的参数我们会事先做了限制, 比如只能使用配置文件中配置的 (如数据库连接池的名称)



👍 21



webmin

2020-02-07

1. 如果项目中已经用了很多单例模式, 比如下面这段代码, 我们该如何在尽量减少代码改动的情况下, 通过重构代码来提高代码的可测试性呢?

CacheManager.getInstance(long userId)中增加Mock开关, 如:

```
private User mockUser;

public CacheManager.setMockObj(User mockUser)

public User getInstance(long userId) {
    if(mockUser != null && mockUser.getUserId() == userId) {
        return mockUser
    }
}
```

2. 在单例支持参数传递的第二种解决方案中, 如果我们两次执行 getInstance(paramA, paramB) 方法, 第二次传递进去的参数是不生效的, 而构建的过程也没有给与提示, 这样就会误导用户。这个问题如何解决呢?

第一次构造Instance成功时需要记录paramA和paramB，在以后的调用需要匹配paramA与paramB构造成功Instance时的参数是否一至，不一至时需要抛出异常。

共 3 条评论>

👍 8



小麦

2020-04-03

不太能理解的使用方式违背了基于接口而非实现的设计原则，比如 spring 中的 service 类一般也是单例的，也是继承接口，controller 的调用也是基于接口，不觉得有什么问题啊，如果实现类变了，也只是改注入而已啊。

共 3 条评论>

👍 6



黄林晴

2020-02-07

打卡



👍 6



@二十一大叔

2022-09-27 来自上海

1.

```
public class Demo {
```

```
    private UserRepo userRepo; // 通过构造哈函数或IOC容器依赖注入
```

```
    private CacheManager cacheManager; // 将获取CacheManager对象提出来，通过  
    依赖注入的方式初始化
```

```
    public Demo(CacheManager cacheManager){  
        this.cacheManager = cacheManager;  
    }
```

```
    public boolean validateCachedUser(long userId) {  
        User cachedUser = getCachedUser(userId);  
        User actualUser =userRepo.getUser(userId);  
        // 省略核心逻辑：对比cachedUser和actualUser...  
    }
```

```
    public User getCachedUser(long userId){  
        return cacheManager.getInstance().getUser(userId);  
    }
```

```
}
```

```
static class MockManager extends CacheManager {  
    private static MockManager mockManager;
```

```
    private MockManager(){}  
  
    public static MockManager getInstance(){  
        //todo  
        return mockManager;  
    }  
  
    public static User getUser(long userId){  
        // 返回mock数据  
        return new User(userId);  
    }  
}
```

```
public static void main(String[] args) {  
    CacheManager cacheManager = MockManager().getInstance();  
    Demo demo = new Demo(cacheManager);  
    User user = demo.getCachedUser(123L);  
}
```

共 1 条评论 >

 5