42讲单例模式（中）：我为什么不推荐使用单例模式又有何替代方案

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。

```
public class Order {
  public void create(...) {
    //...
    long id = IdGenerator.getInstance().getId();
    //...
  }
}

public class User {
  public void create(...) {
    // ...
    long id = IdGenerator.getInstance().getId();
    //...
  }
}
```

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。

```
public class Order {
  public void create(...) {
    //...
    long id = IdGenerator.getInstance().getId();
    // 需要将上面一行代码，替换为下面一行代码
    long id = OrderIdGenerator.getIntance().getId();
    //...
  }
}

public class User {
  public void create(...) {
    // ...
    long id = IdGenerator.getInstance().getId();
    // 需要将上面一行代码，替换为下面一行代码
    long id = UserIdGenerator.getIntance().getId();
  }
}
```

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：

```
public class Singleton {
  private static Singleton instance = null;
  private final int paramA;
  private final int paramB;

  private Singleton(int paramA, int paramB) {
    this.paramA = paramA;
    this.paramB = paramB;
  }

  public static Singleton getInstance() {
    if (instance == null) {
       throw new RuntimeException("Run init() first.");
    }
    return instance;
  }

  public synchronized static Singleton init(int paramA, int paramB) {
    if (instance != null){
       throw new RuntimeException("Singleton has been created!");
    }
    instance = new Singleton(paramA, paramB);
    return instance;
  }
}

Singleton.init(10, 50); // 先init，再使用
Singleton singleton = Singleton.getInstance();
```

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

```
public class Singleton {
  private static Singleton instance = null;
  private final int paramA;
  private final int paramB;

  private Singleton(int paramA, int paramB) {
    this.paramA = paramA;
    this.paramB = paramB;
  }

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

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signleton2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。

```
Singleton singleton1 = Singleton.getInstance(10, 50);
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。

```
public class Config {
  public static final int PARAM_A = 123;
  public static fianl int PARAM_B = 245;
}

public class Singleton {
  private static Singleton instance = null;
  private final int paramA;
  private final int paramB;

  private Singleton() {
    this.paramA = Config.PARAM_A;
    this.paramB = Config.PARAM_B;
  }

  public synchronized static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

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：

```
// 静态方法实现方式
public class IdGenerator {
  private static AtomicLong id = new AtomicLong(0);

  public static long getId() {
    return id.incrementAndGet();
  }
}
// 使用举例
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，还有另外一个种使用方法。具体的代码如下所示：

```
// 1. 老的使用方式
public demofunction() {
  //...
  long id = IdGenerator.getInstance().getId();
  //...
}

// 2. 新的使用方式：依赖注入
public demofunction(IdGenerator idGenerator) {
  long id = idGenerator.getId();
}
// 外部调用demofunction()的时候，传入idGenerator
IdGenerator idGenerator = IdGenerator.getInsance();
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）。

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?

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

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

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?

```
Singleton singleton1 = Singleton.getInstance(10, 50);
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。

精选留言

小晏子

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validateCachedUser。

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，否则就新创建一个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;
}

2020-02-07 09:38

webmin

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。
2020-02-07 13:35

黄林晴
打卡
2020-02-07 02:07

Jeff.Smile
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！
2020-02-07 19:02

Eden Ma

2、instance不为空抛出异常
2020-02-07 12:17

Ken张云忠
1.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?
将单例类中新增一个用于获取测试instance的函数,命名getTestInstance(User testUser),该函数中把需要的测试用例通过参数传入instance当中,当要做测试时就可以通过getTestInstance函数来获取实例得到需要的测试数据.
public boolean validateCachedUser(long userId) {
User actualUser = userRepo.getUser(userId);
//User cachedUser = CacheManager.getInstance().getUser(userId);//生产使用
User cachedUser = CacheManager.getTestInstance(actualUser).getUser(userId);//测试使用
// 省略核心逻辑：对比cachedUser和actualUser...
}
2.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?
第二次调用getInstance时如果带有与之前相同参数就直接返回instance实例;如果参数不相同且业务允许构建新的instance实例就允许再第二次getInstance时构建新的实例,如果业务不允许就在构建时抛出异常.
public synchronized static Singleton getInstance(int paramA, int paramB) {
if (instance == null) {
instance = new Singleton(paramA, paramB);
} else if (this.paramA != paramA || this.paramB != paramB) {
//instance = new Singleton(paramA, paramB);// 业务允许
throw new RuntimeException("Singleton has been created!");// 业务不允许
}
return instance;
}
2020-02-09 15:58

忆水寒
第一个问题，为了增加可测试性，也就是尽量可以测试中间结果。我觉得可以将cacheUser那一行代码和下一行代码分别抽取出来封装　。
第二个问题，可以将参数保存在静态类中，本身这个类新增一个init函数，在new 对象后进行调用init。这样用户可以不需要加

载参数。当然了，如果一定要在getInstance时传入参数，那么也可以校验参数是否和上一次传入的参数是否一致。

2020-02-07 22:09

李小四

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。

2020-02-22 21:43

岁月

讨论题1: 这里测试的不是单例对象, 而是依赖单例的类, 所以单例可以直接用依赖注入的形式传入即可, Demo类不要在内部创建单例对象, 而是直接使用外部传入的单例对象.

讨论题2: 把不同的参数转成字符串作为key, 再用一个字典针对每一个key单独创建单例对象, 这样可以保证传入的参数相同时获取到的对象是同一个, 不同参数对应不同单例对象.

2020-02-19 16:22

Jaybor

第一个问题：把cache user的获取改成函数返回，这样可以在测试的时候对函数进行mock。

第二个问题：此时需要比较已经存在的instance的参数和传入的新参数是否一致，一致就返回instance，不一致直接报错。

2020-02-18 16:51

桂城老托尼

感谢分享，尝试回答下

1,cacheManager作为demo的属性，构造demo对象时 获取单例，而非每次调用方法时获取。

2，compareAndUpdate 思路去解决这个问题，不过虽然能解决问题，但这种用法真的好吗？太刁钻了~

2020-02-16 07:42

FIGNT

1、将单例获取对象的代码抽离单独方法。再对方法mock

2、如果参数不相同，直接报错，告诉使用者如何使用。

```
public synchronized static Singleton getInstance(int paramA, int paramB) {
if (instance == null) {
instance = new Singleton(paramA, paramB);
}else{
if(paramA == this.paramA && paramB == this.paramB){
return instance;
}else{
throw new RuntimeException("illegal argument,please use getInstance("+this.paramA+","+this.paramB+") method");
}
}
}
```

2020-02-15 17:46

小喵喵

2.可以使用反射修改单例中参数信息

2020-02-14 13:40

L

课堂讨论: 1. 把 User cachedUser = CacheManager.getInstance().getUser(userId); 抽出来, 变成一个函数,这样就可以对函数 mock 了

2. 在进 getInstance函数的时候, 对全局变量 instance 置空

2020-02-13 15:07

whistleman

打卡～

2020-02-13 08:26

小刀

第二次调用getInstance时如果带有与之前相同参数就直接返回instance实例;如果参数不相同且业务允许构建新的instance实例就允许再第二次getInstance时构建新的实例,如果业务不允许就在构建时抛出异常.

2020-02-11 12:18

bin

第一个问题：把 CacheManager.getInstance().getUser() 封装成一个函数,可以用来mock

```
public User getCacheUser(long userId){
}
```

第二个问题，加一个判断，

```
public synchronized static Singleton getInstance(int paramA, int paramB) {
if (instance == null) {
instance = new Singleton(paramA, paramB);
return instance;
}
if( instance != null && (paramA != this.paramA || paramB != this.paramB ) ){
throw new RunTimeException("不能重复初始化");
}
return instance;
}
```

2020-02-10 21:31

天天向上卡索

在 .net core 里，依赖注入模式的使用比较多，对于一些配置会通过options模式来处理，option也可以注入，很灵活

2020-02-10 18:26

Uncle.Wang

我遇到的问题是：在client端开发的时候，工程中有大量单例，这些单例中保存着数据，而这些数据可能是和用户账户相关联的，一旦切换账号，面临reset这些单例的问题，往往存在遗漏。如果大量存在这种单例，会很难维护。

2020-02-09 17:33

守拙

课堂讨论

1. 修改validateCachedUser()方法形参:

```
public boolean validateCachedUser(int userId, CacheManager manager){...}
```

2. 带有参数的getInstance()的一种实现方式:

```
public syncrinized static Singleton getInstance(int paramA, int paramB){
 if(instance == null){
 instance = new Singleton(paramA, paramB);
```

```
}



if(this.paramA != paramA || this.paramB != paramB){

instance = new Singleton(paramA, paramB);

}



return instance;

}
```

2020-02-09 14:00