

18 | 理论四：接口隔离原则有哪三种应用？原则中的“接口”该如何理解？

王争 · 设计模式之美



上几节课中，我们学习了 SOLID 原则中的单一职责原则、开闭原则和里式替换原则，今天我们学习第四个原则，接口隔离原则。它对应 SOLID 中的英文字母“I”。对于这个原则，最关键就是理解其中“接口”的含义。那针对“接口”，不同的理解方式，对应在原则上也有不同的解读方式。除此之外，接口隔离原则跟我们之前讲到的单一职责原则还有点儿类似，所以今天我也会具体讲一下它们之间的区别和联系。

话不多说，现在就让我们正式开始今天的学习吧！

如何理解“接口隔离原则”？

接口隔离原则的英文翻译是“Interface Segregation Principle”，缩写为 ISP。Robert Martin 在 SOLID 原则中是这样定义它的：“Clients should not be forced to depend upon

interfaces that they do not use。”直译成中文的话就是：客户端不应该被强迫依赖它不需要的接口。其中的“客户端”，可以理解为接口的调用者或者使用者。

实际上，“接口”这个名词可以用在很多场合中。生活中我们可以用它来指插座接口等。在软件开发中，我们既可以把它看作一组抽象的约定，也可以具体指系统与系统之间的 API 接口，还可以特指面向对象编程语言中的接口等。

前面我提到，理解接口隔离原则的关键，就是理解其中的“接口”二字。在这条原则中，我们可以把“接口”理解为下面三种东西：

一组 API 接口集合


单个 API 接口或函数

OOP 中的接口概念

接下来，我就按照这三种理解方式来详细讲一下，在不同的场景下，这条原则具体是如何解读和应用的。

把“接口”理解为一组 API 接口集合

我们还是结合一个例子来讲解。微服务用户系统提供了一组跟用户相关的 API 给其他系统使用，比如：注册、登录、获取用户信息等。具体代码如下所示：


 复制代码

```
1 public interface UserService {
2     boolean register(String cellphone, String password);
3     boolean login(String cellphone, String password);
4     UserInfo getUserInfoById(long id);
5     UserInfo getUserInfoByCellphone(String cellphone);
6 }
7
8 public class UserServiceImpl implements UserService {
9     //...
10 }
```

现在，我们的后台管理系统要实现删除用户的功能，希望用户系统提供一个删除用户的接口。这个时候我们该如何来做呢？你可能会说，这不是很简单吗，我只需要在 UserService 中新添加一个 deleteUserByCellphone() 或 deleteUserById() 接口就可以了。这个方法可以解决问题，但是也隐藏了一些安全隐患。

删除用户是一个非常慎重的操作，我们只希望通过后台管理系统来执行，所以这个接口只限于给后台管理系统使用。如果我们把它放到 UserService 中，那所有使用到 UserService 的系统，都可以调用这个接口。不加限制地被其他业务系统调用，就有可能导致误删用户。

当然，最好的解决方案是从架构设计的层面，通过接口鉴权的方式来限制接口的调用。不过，如果暂时没有鉴权框架来支持，我们还可以从代码设计的层面，尽量避免接口被误用。我们参照接口隔离原则，调用者不应该强迫依赖它不需要的接口，将删除接口单独放到另外一个接口 RestrictedUserService 中，然后将 RestrictedUserService 只打包提供给后台管理系统来使用。具体的代码实现如下所示：


 复制代码

```
1 public interface UserService {
2     boolean register(String cellphone, String password);
3     boolean login(String cellphone, String password);
4     UserInfo getUserInfoById(long id);
5     UserInfo getUserInfoByCellphone(String cellphone);
6 }
7
8 public interface RestrictedUserService {
9     boolean deleteUserByCellphone(String cellphone);
10    boolean deleteUserById(long id);
11 }
12
13 public class UserServiceImpl implements UserService, RestrictedUserService {
14     // ...省略实现代码...
15 }
```

在刚刚的这个例子中，我们把接口隔离原则中的接口，理解为一组接口集合，它可以是某个微服务的接口，也可以是某个类库的接口等等。在设计微服务或者类库接口的时候，如果部分接口只被部分调用者使用，那我们就需要将这部分接口隔离出来，单独给对应的调用者使用，而不是强迫其他调用者也依赖这部分不会被用到的接口。

把“接口”理解为单个 API 接口或函数

现在再换一种理解方式，把接口理解为单个接口或函数（以下为了方便讲解，我都简称为“函数”）。那接口隔离原则就可以理解为：函数的设计要功能单一，不要将多个不同的功能逻辑在一个函数中实现。接下来，我们还是通过一个例子来解释一下。

 复制代码

```
1 public class Statistics {
2     private Long max;
3     private Long min;
4     private Long average;
5     private Long sum;
6     private Long percentile99;
7     private Long percentile999;
8     //...省略constructor/getter/setter等方法...
9 }
10
11 public Statistics count(Collection<Long> dataSet) {
12     Statistics statistics = new Statistics();
13     //...省略计算逻辑...
14     return statistics;
15 }
```

在上面的代码中，count() 函数的功能不够单一，包含很多不同的统计功能，比如，求最大值、最小值、平均值等等。按照接口隔离原则，我们应该把 count() 函数拆成几个更小粒度的函数，每个函数负责一个独立的统计功能。拆分之后的代码如下所示：

 复制代码

```
1 public Long max(Collection<Long> dataSet) { //... }
2 public Long min(Collection<Long> dataSet) { //... }
3 public Long average(Collection<Long> dataSet) { //... }
4 // ...省略其他统计函数...
```

不过，你可能会说，在某种意义上讲，count() 函数也不能算是职责不够单一，毕竟它做的事情只跟统计相关。我们在讲单一职责原则的时候，也提到过类似的问题。实际上，判定功能是否单一，除了很强的主观性，还需要结合具体的场景。

如果在项目中，对每个统计需求，Statistics 定义的那几个统计信息都有涉及，那 count() 函数的设计就是合理的。相反，如果每个统计需求只涉及 Statistics 罗列的统计信息中一部分，比如，有的只需要用到 max、min、average 这三类统计信息，有的只需要用到 average、sum。而 count() 函数每次都会把所有的统计信息计算一遍，就会做很多无用功，势必影响代码的性能，特别是在需要统计的数据量很大的时候。所以，在这个应用场景下，count() 函数的设计就有点不合理了，我们应该按照第二种设计思路，将其拆分成粒度更细的多个统计函数。

不过，你应该已经发现，接口隔离原则跟单一职责原则有点类似，不过稍微还是有点区别。单一职责原则针对的是模块、类、接口的设计。而接口隔离原则相对于单一职责原则，一方面它更侧重于接口的设计，另一方面它的思考的角度不同。它提供了一种判断接口是否职责单一的标准：通过调用者如何使用接口来间接地判定。如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。

把“接口”理解为 OOP 中的接口概念

除了刚讲过的两种理解方式，我们还可以把“接口”理解为 OOP 中的接口概念，比如 Java 中的 interface。我还是通过一个例子来给你解释。

假设我们的项目中用到了三个外部系统：Redis、MySQL、Kafka。每个系统都对应一系列配置信息，比如地址、端口、访问超时时间等。为了在内存中存储这些配置信息，供项目中的其他模块来使用，我们分别设计实现了三个 Configuration 类：RedisConfig、MysqlConfig、KafkaConfig。具体的代码实现如下所示。注意，这里我只给出了 RedisConfig 的代码实现，另外两个都是类似的，我这里就不贴了。

 复制代码

```
1 public class RedisConfig {
2     private ConfigSource configSource; //配置中心（比如zookeeper）
3     private String address;
4     private int timeout;
5     private int maxTotal;
6     //省略其他配置：maxWaitMillis,maxIdle,minIdle...
7
8     public RedisConfig(ConfigSource configSource) {
9         this.configSource = configSource;
10    }
```


```

11     public String getAddress() {
12         return this.address;
13     }
14     //...省略其他get()、init()方法...
15
16     public void update() {
17         //从configSource加载配置到address/timeout/maxTotal...
18     }
19 }
20
21 public class KafkaConfig { //...省略... }
22 public class MysqlConfig { //...省略... }
23

```

现在，我们有一个新的功能需求，希望支持 Redis 和 Kafka 配置信息的热更新。所谓“热更新（hot update）”就是，如果在配置中心中更改了配置信息，我们希望在不用重启系统的情况下，能将最新的配置信息加载到内存中（也就是 RedisConfig、KafkaConfig 类中）。但是，因为某些原因，我们并不希望对 MySQL 的配置信息进行热更新。

为了实现这样一个功能需求，我们设计实现了一个 ScheduledUpdater 类，以固定时间频率（periodInSeconds）来调用 RedisConfig、KafkaConfig 的 update() 方法更新配置信息。具体的代码实现如下所示：

 复制代码

```

1  public interface Updater {
2      void update();
3  }
4
5  public class RedisConfig implements Updater {
6      //...省略其他属性和方法...
7      @Override
8      public void update() { //... }
9  }
10
11 public class KafkaConfig implements Updater {
12     //...省略其他属性和方法...
13     @Override
14     public void update() { //... }
15 }
16
17 public class MysqlConfig { //...省略其他属性和方法... }
18

```

```

19 public class ScheduledUpdater {
20     private final ScheduledExecutorService executor = Executors.newSingleThreadSc
21     private long initialDelayInSeconds;
22     private long periodInSeconds;
23     private Updater updater;
24
25     public ScheduledUpdater(Updater updater, long initialDelayInSeconds, long peri
26         this.updater = updater;
27         this.initialDelayInSeconds = initialDelayInSeconds;
28         this.periodInSeconds = periodInSeconds;
29     }
30
31     public void run() {
32         executor.scheduleAtFixedRate(new Runnable() {
33             @Override
34             public void run() {
35                 updater.update();
36             }
37         }, this.initialDelayInSeconds, this.periodInSeconds, TimeUnit.SECONDS);
38     }
39 }
40
41 public class Application {
42     ConfigSource configSource = new ZookeeperConfigSource(/*省略参数*/);
43     public static final RedisConfig redisConfig = new RedisConfig(configSource);
44     public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
45     public static final MySqlConfig mysqlConfig = new MysqlConfig(configSource);
46
47     public static void main(String[] args) {
48         ScheduledUpdater redisConfigUpdater = new ScheduledUpdater(redisConfig, 300,
49             redisConfigUpdater.run());
50
51         ScheduledUpdater kafkaConfigUpdater = new ScheduledUpdater(kafkaConfig, 60, 6
52             kafkaConfigUpdater.run());
53     }
54 }


```

刚刚的热更新的需求我们已经搞定了。现在，我们又有了一个新的监控功能需求。通过命令行来查看 Zookeeper 中的配置信息是比较麻烦的。所以，我们希望能有一种更加方便的配置信息查看方式。

我们可以在项目中开发一个内嵌的 SimpleHttpServer，输出项目的配置信息到一个固定的 HTTP 地址，比如：<http://127.0.0.1:2389/config>。我们只需要在浏览器中输入这个地

址，就可以显示出系统的配置信息。不过，出于某些原因，我们只想暴露 MySQL 和 Redis 的配置信息，不想暴露 Kafka 的配置信息。

为了实现这样一个功能，我们还需要对上面的代码做进一步改造。改造之后的代码如下所示：

 复制代码

```
1 public interface Updater {
2     void update();
3 }
4
5 public interface Viewer {
6     String outputInPlainText();
7     Map<String, String> output();
8 }
9
10 public class RedisConfig implements Updater, Viewer {
11     //...省略其他属性和方法...
12     @Override
13     public void update() { //... }
14     @Override
15     public String outputInPlainText() { //... }
16     @Override
17     public Map<String, String> output() { //... }
18 }
19
20 public class KafkaConfig implements Updater {
21     //...省略其他属性和方法...
22     @Override
23     public void update() { //... }
24 }
25
26 public class MysqlConfig implements Viewer {
27     //...省略其他属性和方法...
28     @Override
29     public String outputInPlainText() { //... }
30     @Override
31     public Map<String, String> output() { //... }
32 }
33
34 public class SimpleHttpServer {
35     private String host;
36     private int port;
37     private Map<String, List<Viewer>> viewers = new HashMap<>();
38
39     public SimpleHttpServer(String host, int port) { //... }
40 }
```



```

41     public void addViewers(String urlDirectory, Viewer viewer) {
42         if (!viewers.containsKey(urlDirectory)) {
43             viewers.put(urlDirectory, new ArrayList<Viewer>());
44         }
45         this.viewers.get(urlDirectory).add(viewer);
46     }
47
48     public void run() { //... }
49 }
50
51 public class Application {
52     ConfigSource configSource = new ZookeeperConfigSource();
53     public static final RedisConfig redisConfig = new RedisConfig(configSource);
54     public static final KafkaConfig kafkaConfig = new KafkaConfig(configSource);
55     public static final MySqlConfig mysqlConfig = new MySqlConfig(configSource);
56
57     public static void main(String[] args) {
58         ScheduledUpdater redisConfigUpdater =
59             new ScheduledUpdater(redisConfig, 300, 300);
60         redisConfigUpdater.run();
61
62         ScheduledUpdater kafkaConfigUpdater =
63             new ScheduledUpdater(kafkaConfig, 60, 60);
64         kafkaConfigUpdater.run();
65
66         SimpleHttpServer simpleHttpServer = new SimpleHttpServer("127.0.0.1", 238
67         simpleHttpServer.addViewer("/config", redisConfig);
68         simpleHttpServer.addViewer("/config", mysqlConfig);
69         simpleHttpServer.run();
70     }
71 }


```

至此，热更新和监控的需求我们就都实现了。我们来回顾一下这个例子的设计思想。

我们设计了两个功能非常单一的接口：Updater 和 Viewer。ScheduledUpdater 只依赖 Updater 这个跟热更新相关的接口，不需要被强迫去依赖不需要的 Viewer 接口，满足接口隔离原则。同理，SimpleHttpServer 只依赖跟查看信息相关的 Viewer 接口，不依赖不需要的 Updater 接口，也满足接口隔离原则。

你可能会说，如果我们不遵守接口隔离原则，不设计 Updater 和 Viewer 两个小接口，而是设计一个大而全的 Config 接口，让 RedisConfig、KafkaConfig、MysqlConfig 都实现这个

Config 接口，并且将原来传递给 ScheduledUpdater 的 Updater 和传递给 SimpleHttpServer 的 Viewer，都替换为 Config，那会有什么问题呢？我们先来看一下，按照这个思路来实现的代码是什么样的。


 复制代码

```
1 public interface Config {
2     void update();
3     String outputInPlainText();
4     Map<String, String> output();
5 }
6
7 public class RedisConfig implements Config {
8     //...需要实现Config的三个接口update/outputIn.../output
9 }
10
11 public class KafkaConfig implements Config {
12     //...需要实现Config的三个接口update/outputIn.../output
13 }
14
15 public class MysqlConfig implements Config {
16     //...需要实现Config的三个接口update/outputIn.../output
17 }
18
19 public class ScheduledUpdater {
20     //...省略其他属性和方法..
21     private Config config;
22
23     public ScheduledUpdater(Config config, long initialDelayInSeconds, long periodIn
24         this.config = config;
25         //...
26     }
27     //...
28 }
29
30 public class SimpleHttpServer {
31     private String host;
32     private int port;
33     private Map<String, List<Config>> viewers = new HashMap<>();
34
35     public SimpleHttpServer(String host, int port) {//...}
36
37     public void addViewer(String urlDirectory, Config config) {
38         if (!viewers.containsKey(urlDirectory)) {
39             viewers.put(urlDirectory, new ArrayList<Config>());
40         }
41         viewers.get(urlDirectory).add(config);
```

```
42     }
43
44     public void run() { //... }
45 }
```

这样的设计思路也是能工作的，但是对比前后两个设计思路，在同样的代码量、实现复杂度、同等可读性的情况下，第一种设计思路显然要比第二种好很多。为什么这么说呢？主要有两点原因。

首先，第一种设计思路更加灵活、易扩展、易复用。因为 Updater、Viewer 职责更加单一，单一就意味了通用、复用性好。比如，我们现在又有一个新的需求，开发一个 Metrics 性能统计模块，并且希望将 Metrics 也通过 SimpleHttpServer 显示在网页上，以方便查看。这个时候，尽管 Metrics 跟 RedisConfig 等没有任何关系，但我们仍然可以让 Metrics 类实现非常通用的 Viewer 接口，复用 SimpleHttpServer 的代码实现。具体的代码如下所示：

 复制代码

```
1 public class ApiMetrics implements Viewer { //... }
2 public class DbMetrics implements Viewer { //... }
3
4 public class Application {
5     ConfigSource configSource = new ZookeeperConfigSource();
6     public static final RedisConfig redisConfig = new RedisConfig(configSource);
7     public static final KafkaConfig kafkaConfig = new KafkaConfig(configSource);
8     public static final MySqlConfig mySqlConfig = new MySqlConfig(configSource);
9     public static final ApiMetrics apiMetrics = new ApiMetrics();
10    public static final DbMetrics dbMetrics = new DbMetrics();
11
12    public static void main(String[] args) {
13        SimpleHttpServer simpleHttpServer = new SimpleHttpServer("127.0.0.1", 238
14        simpleHttpServer.addViewer("/config", redisConfig);
15        simpleHttpServer.addViewer("/config", mySqlConfig);
16        simpleHttpServer.addViewer("/metrics", apiMetrics);
17        simpleHttpServer.addViewer("/metrics", dbMetrics);
18        simpleHttpServer.run();
19    }
20 }
```

其次，第二种设计思路在代码实现上做了一些无用功。因为 Config 接口中包含两类不相关的接口，一类是 update()，一类是 output() 和 outputInPlainText()。理论上，KafkaConfig 只需要实现 update() 接口，并不需要实现 output() 相关的接口。同理，MysqlConfig 只需要实现 output() 相关接口，并需要实现 update() 接口。但第二种设计思路要求 RedisConfig、KafkaConfig、MySqlConfig 必须同时实现 Config 的所有接口函数（update、output、outputInPlainText）。除此之外，如果我们要往 Config 中继续添加一个新的接口，那所有的实现类都要改动。相反，如果我们的接口粒度比较小，那涉及改动的类就比较少。

重点回顾

今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

1. 如何理解“接口隔离原则”？

理解“接口隔离原则”的重点是理解其中的“接口”二字。这里有三种不同的理解。

如果把“接口”理解为一组接口集合，可以是某个微服务的接口，也可以是某个类库的接口等。如果部分接口只被部分调用者使用，我们就需要将这部分接口隔离出来，单独给这部分调用者使用，而不强迫其他调用者也依赖这部分不会被用到的接口。

如果把“接口”理解为单个 API 接口或函数，部分调用者只需要函数中的部分功能，那我们就需要把函数拆分成粒度更细的多个函数，让调用者只依赖它需要的那个细粒度函数。

如果把“接口”理解为 OOP 中的接口，也可以理解为面向对象编程语言中的接口语法。那接口的设计要尽量单一，不要让接口的实现类和调用者，依赖不需要的接口函数。


2. 接口隔离原则与单一职责原则的区别

单一职责原则针对的是模块、类、接口的设计。接口隔离原则相对于单一职责原则，一方面更侧重于接口的设计，另一方面它的思考角度也是不同的。接口隔离原则提供了一种判断接口的职责是否单一的标准：通过调用者如何使用接口来间接地判定。如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。

课堂讨论

今天课堂讨论的话题是这样的：

java.util.concurrent 并发包提供了 AtomicInteger 这样一个原子类，其中有一个函数 getAndIncrement() 是这样定义的：给整数增加一，并且返回未增之前的值。我的问题是，这个函数的设计是否符合单一职责原则和接口隔离原则？为什么？

 复制代码

```
1 /**
2  * Atomically increments by one the current value.
3  * @return the previous value
4  */
5 public final int getAndIncrement() {//...}
```

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

接口隔离原则在软件设计中的应用是本文的重点。作者通过三种不同的理解方式来解释接口隔离原则的应用：将“接口”理解为一组API接口集合，将“接口”理解为单个API接口或函数，以及将“接口”理解为OOP中的接口概念。通过具体的例子和代码实现，深入浅出地解释了接口隔离原则的应用和意义。文章通过设计Updater和Viewer两个功能非常单一的接口，实现了热更新和监控的需求。这种设计思路更加灵活、易扩展、易复用，同时避免了无用功。总之，本文通过清晰的技术指导，帮助读者快速了解接口隔离原则在软件设计中的重要性和实际应用。文章还提到了接口隔离原则与单一职责原则的区别，以及针对java.util.concurrent并发包提供的AtomicInteger类中的getAndIncrement()函数是否符合单一职责原则和接口隔离原则的讨论。这些内容为读者提供了深入的技术思考和讨论。

© 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 (186)

最新 精选



码到成功

2019-12-13

老师可以每次课对上一次课的思考题做下解答吗

作者回复: 集中答疑一下吧 课都提前录好了

共 3 条评论 >

👍 30



陈拾柒

2019-12-19

为什么觉得老师说的，对于接口的三种理解，第一种理解和第三种理解说的是同一件事情~

作者回复: 不一样呢你再看看

共 10 条评论 >

👍 3



郑自明

2020-08-26

Java8 Interface有default method 这样新加的method不需要所有相关类再实现了。但对这些类而言 不就违背了接口隔离原则么？

作者回复: 语法归语法，看你怎么用了，这个跟原则不冲突的。



庚小庚

2020-07-09

接口隔离原则，如果是从API接口角度，我觉得应该是从实现方来看，比如我实现某个接口，我只想实现这个接口的部分功能，其他功能用不上，那么就要考虑这个接口是否符合隔离原则，能否进行粒度拆分，这样也更灵活，针对函数来讲，应该就是从调用者的角度来看，比如我只想统计商品总量，而其他的统计结果，你不要给我，其实我们现在做的项目，都是前后端分离的，让前端调用，很多时候，我们项目中，会有一个很大的用户接口，包括一大堆信息，但是实际上前端只想获取用户的姓名或则手机号。但是你却给我了一大堆，后端也是为了图方便，只写一个接口，反正所有用户的信息全部塞到里面，那从这个角度，是不是也不符合接口隔离原则呢

作者回复: 是的，我觉得应该不过过于大而全，但拆的过细也不好，你可以结合着门面模式看下





辣么大

2019-12-13

Java.util.concurrent.atomic包下提供了机器底层级别实现的多线程环境下原子操作，相比自己实现类似的功能更加高效。

AtomicInteger提供了

intValue() 获取当前值

incrementAndGet() 相当于++i

getAndIncrement相当于i++

从getAndIncrement实现“原子”操作的角度上来说，原子级别的给整数加一，返回未加一之前的值。它的职责是明确的，是符合单一职责的。

从接口隔离原则上看，也是符合的，因为AtomicInteger封装了原子级别的整数操作。

补充：

多线程环境下如果需要计数的话不需旧的值时，推荐使用LongAdder或者LongAccumulator（CoreJava上说更加高效，但我对比了AtomicLong和LongAdder，没感觉效率上有提高，可能是例子写的不够准确。测试代码见 <https://github.com/gdhucoder/Algorithms4/tree/master/designpattern/u18> 希望和小伙伴们一起讨论）

共 15 条评论 >

👍 159



李小四

2019-12-13

设计模式_18

纯理论分析，这么设计是不符合“接口隔离”原则的，毕竟，get是一个操作，increment是另一个操作。

结合具体场景，Atomic类的设计目的是保证操作的原子性，专门看了一下AtomicInteger的源码，发现没有单独的 increment 方法，然后思考了一下线程同步时的的问题，场景需要保证 get 与 increment 中间不插入其他操作，否则函数的正确性无法保证，从场景的角度，它又是符合原则的。

共 7 条评论 >

👍 105



星溯

2021-04-01

老师此题大有深意，我们可以从此思考题中方法的设计来深化对单一职责和接口隔离的理解：接口隔离，强调的是调用方，是否只使用了接口中的部分功能？若是，则违反接口隔离，应当细粒度拆分接口，从这个例子看，调用方诉求与方法名完全一致，通过方法内部封装两个操作，实现原子性，达成了调用方的最终目的，不多不少。

单一职责，不强调是否为调用方，只要能某一角度观察出，一个模块/类/方法，负责了多于一件事情，就可判定其破坏了单一职责，基于此经典理论，不假以深层次思考的角度出发，从方法本身的命名（做两件事）就可断定，它一定是破坏了单一职责的，应该拆分为两个操作。但我们可以结合老师说的，判定职责是否单一，要懂得结合业务场景，业务需求，此方法，其实就是要通过JDK提供的CAS乐观自选锁（方法最终依赖硬件指令集原语，Compare And Swap）从“原语”这一词的含义看，其实也是同时、原子性地做了一件“完整”的事情，因此，考虑这一点，是可以判定它符合单一职责的。

而这其实正是单一职责判定结果，往往见仁见智的原因：基于不同的角度，不同的立场，不同的业务理解，往往可以得到不同的判定结果，但不必纠结，判定过程中用到的思想才是精髓。

共 5 条评论 >

👍 61



Geek_e9b8c4

2019-12-13

总结成思维导图了，链接 <https://blog.csdn.net/dingshuo168/article/details/103531805>



👍 36



M

2020-02-08

接口隔离原则：我只要我想要的，不想要的别给我

共 2 条评论 >

👍 17



小晏子

2019-12-13

思考题：

先看是否符合单一职责原则，这个函数的功能是加1然后返回之前的值，做了两件事，是不符合单一职责原则的！

但是却符合接口隔离原则，从调用者的角度来看的话，因为这个类是Atomic类，需要的所有操作都是原子的，所以为了满足调用者需要原子性的完成加一返回的操作，提供一个这样的接口是必要的，满足接口隔离原则。

共 1 条评论 >

👍 17