18讲理论四：接口隔离原则有哪三种应用原则中的“接口”该如何理解

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：

```
public interface UserService {
  boolean register(String cellphone, String password);
  boolean login(String cellphone, String password);
  UserInfo getUserInfoById(long id);
  UserInfo getUserInfoByCellphone(String cellphone);
}

public class UserServiceImpl implements UserService {
  //...
}
```

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：

```
public interface UserService {
  boolean register(String cellphone, String password);
  boolean login(String cellphone, String password);
  UserInfo getUserInfoById(long id);
  UserInfo getUserInfoByCellphone(String cellphone);
}

public interface RestrictedUserService {
  boolean deleteUserByCellphone(String cellphone);
  boolean deleteUserById(long id);
}

public class UserServiceImpl implements UserService, RestrictedUserService {
  // ...省略实现代码...
}
```

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

现在我们再换一种理解方式，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。

```
public class Statistics {
  private Long max;
  private Long min;
  private Long average;
  private Long sum;
  private Long percentile99;
  private Long percentile999;
  //...省略constructor/getter/setter等方法...
}

public Statistics count(Collection<Long> dataSet) {
  Statistics statistics = new Statistics();
  //...省略计算逻辑...
  return statistics;
}
```

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：

```
public Long max(Collection<Long> dataSet) { //... }
public Long min(Collection<Long> dataSet) { //... }
public Long average(Colletion<Long> dataSet) { //... }
// ...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。

```
public class RedisConfig {
    private ConfigSource configSource; //配置中心（比如zookeeper）
    private String address;
    private int timeout;
    private int maxTotal;
    //省略其他配置: maxWaitMillis,maxIdle,minIdle...

    public RedisConfig(ConfigSource configSource) {
        this.configSource = configSource;
    }

    public String getAddress() {
        return this.address;
    }
    //...省略其他get()、init()方法...

    public void update() {
      //从configSource加载配置到address/timeout/maxTotal...
    }
}

public class KafkaConfig { //...省略... }
public class MysqlConfig { //...省略... }
```

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：

```
public interface Updater {
  void update();
}

public class RedisConfig implemets Updater {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
}

public class KafkaConfig implements Updater {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
}

public class MysqlConfig { //...省略其他属性和方法... }

public class ScheduledUpdater {
    private final ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();;
    private long initialDelayInSeconds;
    private long periodInSeconds;
    private Updater updater;

    public ScheduleUpdater(Updater updater, long initialDelayInSeconds, long periodInSeconds) {
        this.updater = updater;
        this.initialDelayInSeconds = initialDelayInSeconds;
        this.periodInSeconds = periodInSeconds;
    }

    public void run() {
        executor.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                updater.update();
            }
        }, this.initialDelayInSeconds, this.periodInSeconds, TimeUnit.SECONDS);
    }
}

public class Application {
  ConfigSource configSource = new ZookeeperConfigSource(/*省略参数*/);
  public static final RedisConfig redisConfig = new RedisConfig(configSource);
  public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
```

```
  public static final MySqlConfig mysqlConfig = new MysqlConfig(configSource);

  public static void main(String[] args) {
    ScheduledUpdater redisConfigUpdater = new ScheduledUpdater(redisConfig, 300, 300);
    redisConfigUpdater.run();

    ScheduledUpdater kafkaConfigUpdater = new ScheduledUpdater(kafkaConfig, 60, 60);
    redisConfigUpdater.run();
  }
}
```

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：

```
public interface Updater {
  void update();
}

public interface Viewer {
  String outputInPlainText();
  Map<String, String> output();
}

public class RedisConfig implemets Updater, Viewer {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
  @Override
  public String outputInPlainText() { //... }
  @Override
  public Map<String, String> output() { //...}
}

public class KafkaConfig implements Updater {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
}
```

```
public class MysqlConfig implements Viewer {
  //...省略其他属性和方法...
  @Override
  public String outputInPlainText() { //... }
  @Override
  public Map<String, String> output() { //...}
}

public class SimpleHttpServer {
  private String host;
  private int port;
  private Map<String, List<Viewer>> viewers = new HashMap<>();

  public SimpleHttpServer(String host, int port) {//...}

  public void addViewers(String urlDirectory, Viewer viewer) {
    if (!viewers.containsKey(urlDirectory)) {
      viewers.put(urlDirectory, new ArrayList<Viewer>());
    }
    this.viewers.get(urlDirectory).add(viewer);
  }

  public void run() { //... }
}

public class Application {
    ConfigSource configSource = new ZookeeperConfigSource();
    public static final RedisConfig redisConfig = new RedisConfig(configSource);
    public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
    public static final MySqlConfig mysqlConfig = new MySqlConfig(configSource);

    public static void main(String[] args) {
        ScheduledUpdater redisConfigUpdater =
            new ScheduledUpdater(redisConfig, 300, 300);
        redisConfigUpdater.run();

        ScheduledUpdater kafkaConfigUpdater =
            new ScheduledUpdater(kafkaConfig, 60, 60);
        redisConfigUpdater.run();

        SimpleHttpServer simpleHttpServer = new SimpleHttpServer(“127.0.0.1”, 2389);
        simpleHttpServer.addViewer("/config", redisConfig);
        simpleHttpServer.addViewer("/config", mysqlConfig);
```

```
        simpleHttpServer.run();
    }
}
```

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。

```
public interface Config {
  void update();
  String outputInPlainText();
  Map<String, String> output();
}

public class RedisConfig implements Config {
  //...需要实现Config的三个接口update/outputIn.../output
}

public class KafkaConfig implements Config {
  //...需要实现Config的三个接口update/outputIn.../output
}

public class MysqlConfig implements Config {
  //...需要实现Config的三个接口update/outputIn.../output
}

public class ScheduledUpdater {
  //...省略其他属性和方法..
  private Config config;

  public ScheduleUpdater(Config config, long initialDelayInSeconds, long periodInSeconds) {
      this.config = config;
      //...
  }
  //...
}
```

```
public class SimpleHttpServer {
  private String host;
  private int port;
  private Map<String, List<Config>> viewers = new HashMap<>();

  public SimpleHttpServer(String host, int port) {//...}

  public void addViewer(String urlDirectory, Config config) {
    if (!viewers.containsKey(urlDirectory)) {
      viewers.put(urlDirectory, new ArrayList<Config>());
    }
    viewers.get(urlDirectory).add(config);
  }

  public void run() { //... }
}
```

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：

```
public class ApiMetrics implements Viewer {//...}
public class DbMetrics implements Viewer {//...}

public class Application {
    ConfigSource configSource = new ZookeeperConfigSource();
    public static final RedisConfig redisConfig = new RedisConfig(configSource);
    public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
    public static final MySqlConfig mySqlConfig = new MySqlConfig(configSource);
    public static final ApiMetrics apiMetrics = new ApiMetrics();
    public static final DbMetrics dbMetrics = new DbMetrics();

    public static void main(String[] args) {
        SimpleHttpServer simpleHttpServer = new SimpleHttpServer(“127.0.0.1”, 2389);
        simpleHttpServer.addViewer("/config", redisConfig);
        simpleHttpServer.addViewer("/config", mySqlConfig);
        simpleHttpServer.addViewer("/metrics", apiMetrics);
        simpleHttpServer.addViewer("/metrics", dbMetrics);
        simpleHttpServer.run();
    }
}
```

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？

```
/**
 * Atomically increments by one the current value.
 * @return the previous value
 */
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。

精选留言

辣么大

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）

2019-12-13 08:48

李小四

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。

2019-12-13 10:18

时光流逝，而我们在干嘛?

老师可以每次课对上一次课的思考题做下解答吗

2019-12-13 08:17

作者回复

集中答疑一下吧 课都提前录好了

2019-12-13 08:49

NoAsk

单一职责原则针对的是模块、类、接口的设计。getAnd
Increase()虽然集合了获取和增加两个功能，但是它作为对atomicInteger的值的常用方法，提供对其值的常规操作，是满足单一原则的。

从单一原则的下面这个解释考虑，是不满足接口隔离原则的。“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。”，用户可能调用获取或增加的其中一个方法，再或者先调用增加再调用获取increaseAndGet()方法。

这是我个人理解，还望大家指正。

2019-12-13 07:44

北岛明月

符合SRP 也符合ISP 。

理由是这个方法完成的逻辑就是一个功能：新增和返回旧值。只不过是两步操作罢了。如果你想获取，就用get方法，自增就用increment 方法。都有提供哇。

SRP：老师在文中说，实际上，要从中做出选择，我们不能脱离具体的应用场景。所以我认为是符合的。

ISP: 可以参考老师说的这句话：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。
我们调用这个方法肯定是要用它的整个功能，而不是其中的一个新增或自增功能。

2019-12-13 08:29

小晏子

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。

2019-12-13 09:05

墨雨

单一职责是针对于模块、类在具体的需求业务场景下是否符合只做一件事情的原则。
而接口隔离原则就细化到了接口方面，我是这样理解的，如果接口中的某些方法对于需要继承实现它的类来说是多余的，那么这个接口的设计就不符合接口隔离原则，可以考虑再拆分细化。
对于课后思考题，他只对该数做了相关操作符合单一职责原则。但从接口、函数来看它实现了两个功能，获取整数及给该整数加一，是不符合接口隔离原则的。
不知道我这样考虑是否正确，望指正

2019-12-13 08:48

DullBird

感觉是属于比较难判断是否单一职责的内容，顺便回头翻了一下单一职责的章节:在理解一下，单一职责主要还是要结合业务，getAndIncrement的方法实现了: 原子的获取并且新增的这一职责，如果拆开成get和Increment的话，就需要外层加锁处理原子的获取并新增操作，对于业务不太合适。
从接口隔离的原则看，调用这个方法的类，本身就是依赖这个接口，所以并没有违反。
想到一个问题：
如果一个类中，有n个查询的业务接口，根据姓名查，根据年纪查，根据地址查(假设不是参数控制，而是拆成3个接口)。那么不同调用方依赖这个类的时候，有可能是根据姓名查，有可能根据年纪查，如果都拆开了。那么接口是不是粒度太细了

2019-12-14 15:27

小海

回答课后讨论题得结合具体的场景和运行环境。AtomicInteger的getAndIncrement()函数的职责很单一，就是"获取当前值并递增"这一步原子操作，有人说这是两步操作，这个函数是运行在多线程并发环境下，在这种环境下把获取当前值和递增拆分成两个函数会获得错误的结果，而该函数内部封装了两步操作使其成为一个原子操作，从这个角度任意一方都是另一方的附属品，两者必须同时完成而不能拆分，如果仅仅是为了获取当前值或者递增那完全可以使用该类的其它函数。从调用方的角度，必然是同时用到了获取当前值和递增两个功能，而不是部分功能，明白该函数设计的"单一职责"，就知道它符合SRP和ISP，不要试

图去拆分一个原子操作。

2019-12-14 10:24

黄林晴

思考题:

个人感觉，不符合单一职责，也不符合接口隔离，因为函数做了两件事，不应该把获取当前值和值加1放在一起，因为

1.用户可能需要-1 *1等其他运算操作再返回原始值，这样就要n个方法每个方法中都有返回原始值的操作。

2.用户可能只想运算操作，不想运算后暴露原始结果

3.如果用户以后还想获取操作后的值，这个函数就不能同时返回两个值了

希望大家指正

2019-12-13 08:27

Chen

getAndIncrement()符合接口隔离原则，这是不是一个大而全的函数，而是一个细粒度的函数，跟count++的功能类似。

2019-12-13 07:53

LYy

SIP: 如果调用方不完全需要接口提供的全部功能，那么就需要审视接口是否可以进一步拆分。

2020-01-05 00:43

陈拾柒

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~

2019-12-19 09:04

作者回复

不一样呢你再看看

2019-12-20 07:08

Geek_e9b8c4

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

2019-12-13 18:31

Jxin

1.我主观认为都符合。

1.从名命来看，这个方法要做两件事，事实上它也只做了这两件事。所以这个方法的实现满足这个方法抽象的功能范围，耦合单一职责原则。

2.从使用方来看，返回数值并递增是这类原子类的常规使用场景（longaddr没这方法，蛋疼）。所以对于使用方而言，这个方法包含的两个功能都是其所需要的，所以它也满足接口隔离原则。

2019-12-13 13:13

下雨天

老师提到三种接口的情况可以这样理解:

1. 接口定义前设计原则(理解成OOP中语法接口或未定义的函数):尽量单一，细粒度!

2.接口定义后设计原则(理解成粗粒度的一组集合或函数):已有接口按需(调用者或者新功能)拆分或单独定义接口!

2019-12-13 09:38

静静聆听

老师，我觉得接口隔离原则重在隔离二字，单一只是辅助功能

2020-02-15 14:20

巨龙的力量啊

接口小粒度划分的同时，也要结合实际情况；接口功能是否隔离，可以看使用方的调用情况，是否都用到该接口的所有功能，如果只用到部分，就说明，接口在当前情况下设计是不合理的，不便于拓展。

2020-02-11 16:37

空空

单一职责原则，侧重于针对某个特定接口，要满足功能的独立性和唯一性；

接口隔离原则，则强调接口与接口(方法与方法)之间的关系，要满足功能上的相互隔离，

2020-02-10 12:05

谷雨

单一职责和接口隔离的区别，我的理解：

单一职责，是划分域，这部分属于 A，另一部分属于 B。而接口隔离，更侧重不要“给调用者冗余的”，而冗余的本质，其实还是同一域。

所以，单一职责，是域与域，而接口隔离，是域内。

2020-02-08 22:47