## 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。

### 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：

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);

kafkaConfigUpdater.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);

kafkaConfigUpdater.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，并且返回未増之前的值。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() {//...}