

## 15 | 软件设计的接口隔离原则：如何对类的调用者隐藏类的公有方法？

2019-12-25 李智慧

后端技术面试38讲

[进入课程 >](#)



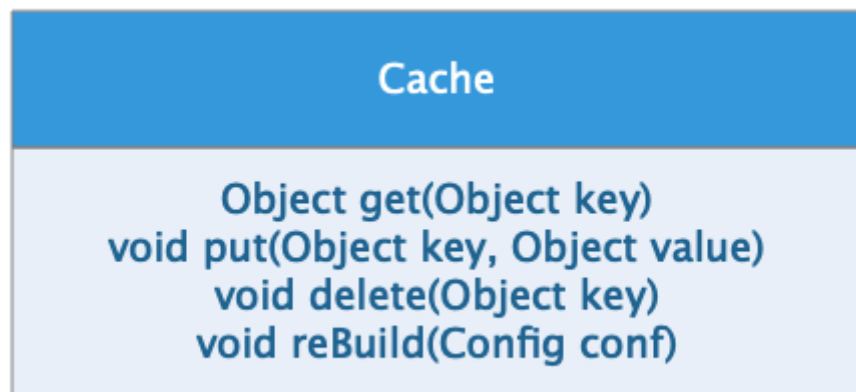
讲述：李智慧

时长 09:49 大小 8.99M



我在阿里巴巴工作期间，曾经负责开发一个统一缓存服务。这个服务要求能够根据远程配置中心的配置信息，在运行期动态更改缓存的配置，可能是将本地缓存更改为远程缓存，也可能是更改远程缓存服务器集群的 IP 地址列表，进而改变应用程序使用的缓存服务。

这就要求缓存服务的客户端 SDK 必须支持运行期配置更新，而配置更新又会直接影响缓存数据的操作，于是就设计出这样一个缓存服务 Client 类。



这个缓存服务 Client 类的方法主要包含两个部分：一部分是缓存服务方法，get()、put()、delete() 这些，这些方法是面向调用者的；另一部分是配置更新方法 reBuild()，这个方法主要是给远程配置中心调用的。

但是问题是，Cache 类的调用者如果看到 reBuild() 方法，并错误地调用了该方法，就可能导致 Cache 连接被错误重置，导致无法正常使用 Cache 服务。所以必须要将 reBuild() 方法向缓存服务的调用者隐藏，而只对远程配置中心的本地代理开放这个方法。

但是 reBuild() 方法是一个 public 方法，**如何对类的调用者隐藏类的公有方法？**

## 接口隔离原则


我们可以使用接口隔离原则解决这个问题。接口隔离原则说：**不应该强迫用户依赖他们不需要的的方法。**

那么如果强迫用户依赖他们不需要的的方法，会导致什么后果呢？

一来，用户可以看到这些他们不需要，也不理解的方法，这样无疑会增加他们使用的难度，如果错误地调用了这些方法，就会产生 bug。二来，当这些方法如果因为某种原因需要更改的时候，虽然不需要但是依赖这些方法的用户程序也必须做出更改，这是一种不必要的耦合。

但是如果一个类的几个方法之间本来就是互相关联的，就像我开头举的那个缓存 Client SDK 的例子，reBuild() 方法必须要在 Cache 类里，这种情况下，如何做到不强迫用户依赖他们不需要的的方法呢？

我们先看一个简单的例子，Modem 类定义了 4 个主要方法，拨号 dial()，挂断 hangup()，发送 send() 和接受 recv()。这四个方法互相存在关联，需要定义在一个类里。

 复制代码

```
1 class Modem {  
2     void dial(String pno);  
3     void hangup();  
4     void send(char c);  
5     void recv();  
6 }  
7
```

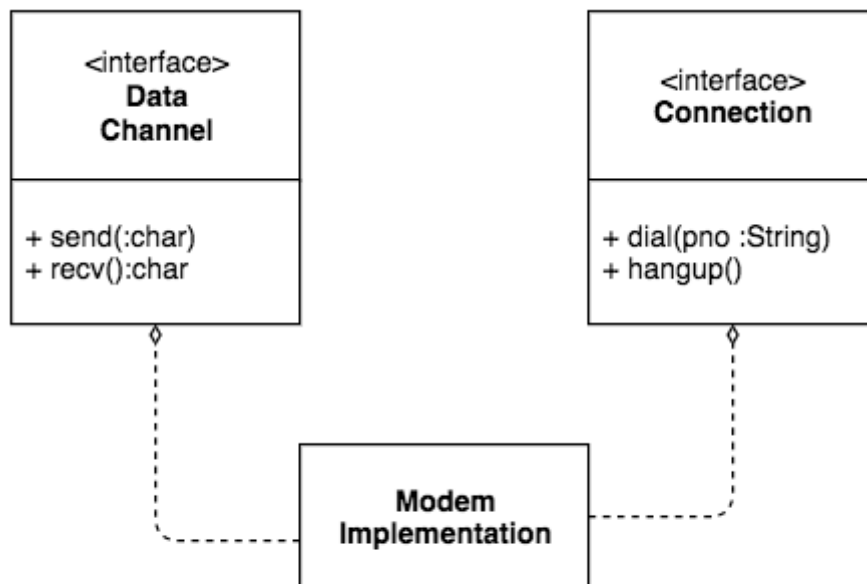
但是对调用者而言，某些方法可能完全不需要，也不应该看到。比如拨号 dial() 和挂断 hangup()，这两个方式是属于专门的网络连接程序的，通过网络连接程序进行拨号上网或者挂断网络。而一般的使用网络的程序，比如网络游戏或者上网浏览器，只需要调用 send() 和 recv() 发送和接收数据就可以了。

强迫只需要上网的程序依赖他们不需要的拨号与挂断方法，只会导致不必要的耦合，带来潜在的系统异常。比如在上网浏览器中不小心调用 hangup() 方法，就会导致整个机器断网，其他程序都不能连接网络。这显然不是系统想要的。

这种问题的解决方法就是通过接口进行方法隔离，Modem 类实现两个接口，DataChannel 接口和 Connection 接口。

DataChannel 接口对外暴露 send() 和 recv() 方法，这个接口只负责网络数据的发送和接收，网络游戏或者网络浏览器只依赖这个接口进行网络数据传输。这些应用程序不需要依赖它们不需要的 dial() 和 hangup() 方法，对应用开发者更加友好，也不会导致因错误的调用而引发的程序 bug。

而网络管理程序则可以依赖 Connection 接口，提供显式的 UI 让用户拨号上网或者挂断网络，进行网络连接管理。



通过使用**接口隔离原则**，我们可以将一个实现类的不同方法包装在不同的接口中对外暴露。应用程序只需要依赖它们需要的方法，而不会看到不需要的方法。

## 一个使用接口隔离原则优化的例子

我们再看一个使用接口隔离原则优化设计的例子。假设我们有个门 Door 对象，这个 Door 对象可以锁上，可以解锁，还可以判断门是否打开。

```
1 class Door {
2     void lock();
3     void unlock();
4     boolean isDoorOpen();
5 }
```

[复制代码](#)

现在我们需要一个 TimedDoor，一个有定时功能的门，如果门开着的时间超过预定时间，就会自动锁门。

我们已经有一个类 Timer，和一个接口 TimerClient：

```
1 class Timer {
2     void register(int timeout, TimerClient client);
```

[复制代码](#)

```
3 }
4
5
6 interface TimerClient {
7     void timeout();
8 }
```

TimerClient 可以向 Timer 注册，调用 register() 方法，设置超时时间。当超时时间到，就会调用 TimerClient 的 timeout() 方法。

那么，我们如何利用现有的 Timer 和 TimerClient 将 Door 改造成一个具有超时自动锁门的 TimedDoor?

比较容易，且直观的办法就是，修改 Door 类，Door 实现 TimerClient 接口，这样 Door 就有了 timeout() 方法，直接将 Door 注册给 Timer，当超时的时候，Timer 调用 Door 的 timeout() 方法，在 Door 的 timeout() 方法里调用 lock() 方法，就可以实现超时自动锁门的操作。

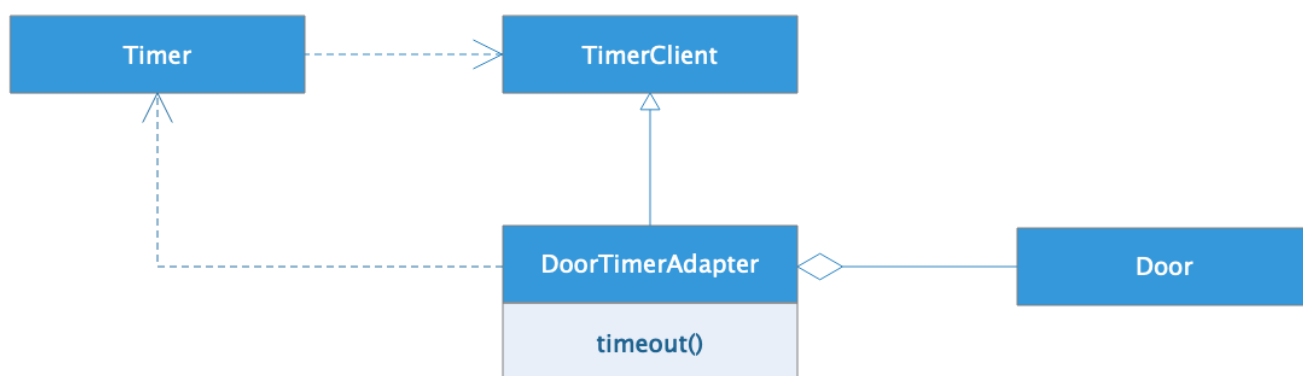
 复制代码

```
1 class Door implements TimerClient {
2     void lock();
3     void unlock();
4     boolean isDoorOpen();
5     void timeout(){
6         lock();
7     }
8 }
```

这个方法简单直接，也能实现需求，但是问题在于使 Door 多了一个 timeout() 方法。如果这个 Door 类想要复用到其他地方，那么所有使用 Door 的程序都不得不依赖一个它们可能根本用不着的方法。同时，Door 的职责也变得复杂，违反了单一职责原则，维护会变得更加困难。这样的设计显然是有问题的。

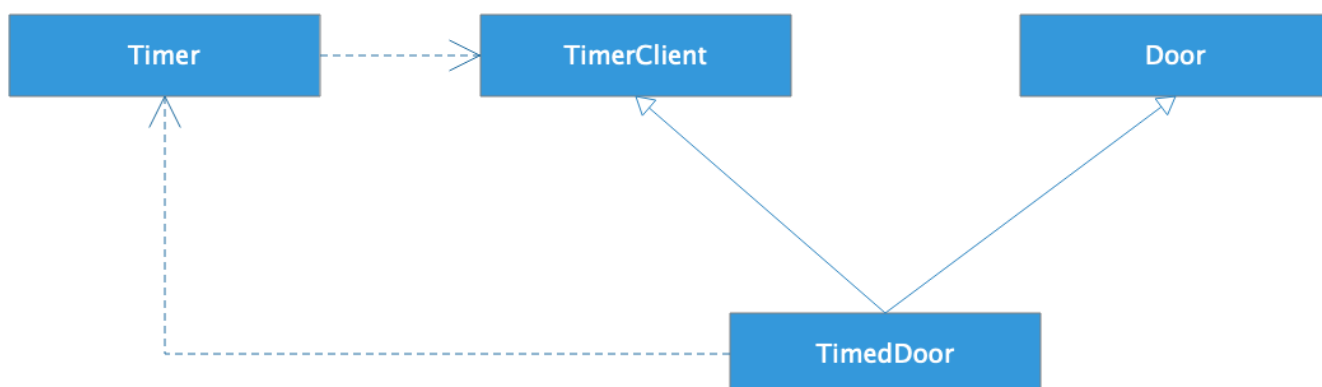
要想解决这些问题，就应该遵循接口隔离原则。事实上，这里有两个互相独立的接口，一个接口是 TimerClient，用来供 Timer 进行超时控制；一个接口是 Door，用来控制门的操作。虽然超时锁门的操作是一个完整的动作，但是我们依然可以使用接口使其隔离。

一种方法是通过委托进行接口隔离，具体方式就是增加一个适配器 DoorTimerAdapter，这个适配器继承 TimerClient 接口实现 timeout() 方法，并将自己注册给 Timer。适配器在自己的 timeout() 方法中，调用 Door 的方法实现超时锁门的操作。



这种场合使用的适配器可能会比较重，业务逻辑比较多，如果超时的时候需要执行较多的逻辑操作，那么适配器的 timeout() 方法就会包含很多业务逻辑，超出了适配器的职责范围。而如果这些逻辑操作还需要使用 Door 的内部状态，可能还需要迫使 Door 做出一些修改。

接口隔离更典型的做法是使用多重继承，跟前面 Modem 的例子一样，TimedDoor 同时实现 TimerClient 接口和继承 Door 类，在 TimedDoor 中实现 timeout() 方法，并注册到 Timer 定时器中。



这样，使用 Door 的程序就不需要被迫依赖 timeout() 方法，Timer 也不会看到 Door 的方法，程序更加整洁，易于复用。

## 接口隔离原则在迭代器设计模式中的应用

Java 的数据结构容器类可以通过 for 循环直接进行遍历，比如：



[复制代码](#)

```
1 List<String> ls = new ArrayList<String>();
2 ls.add("a");
3 ls.add("b");
4 for(String s: ls) {
5     System.out.println(s);
6 }
```

事实上，这种 for 语法结构并不是标准的 Java for 语法，标准的 for 语法在实现上述遍历时应该是这样的：

[复制代码](#)

```
1 for(Iterator<String> itr=ls.iterator();itr.hasNext();) {
2     System.out.println(itr.next());
3 }
```

之所以可以写成上面那种简单的形式，就是因为 Java 提供的语法糖。Java5 以后版本对所有实现了 Iterable 接口的类都可以使用这种简化的 for 循环进行遍历。而我们上面例子的 ArrayList 也实现了这个接口。

Iterable 接口定义如下，主要就是构造 Iterator 迭代器。

[复制代码](#)

```
1 public interface Iterable<T> {
2     Iterator<T> iterator();
3 }
```

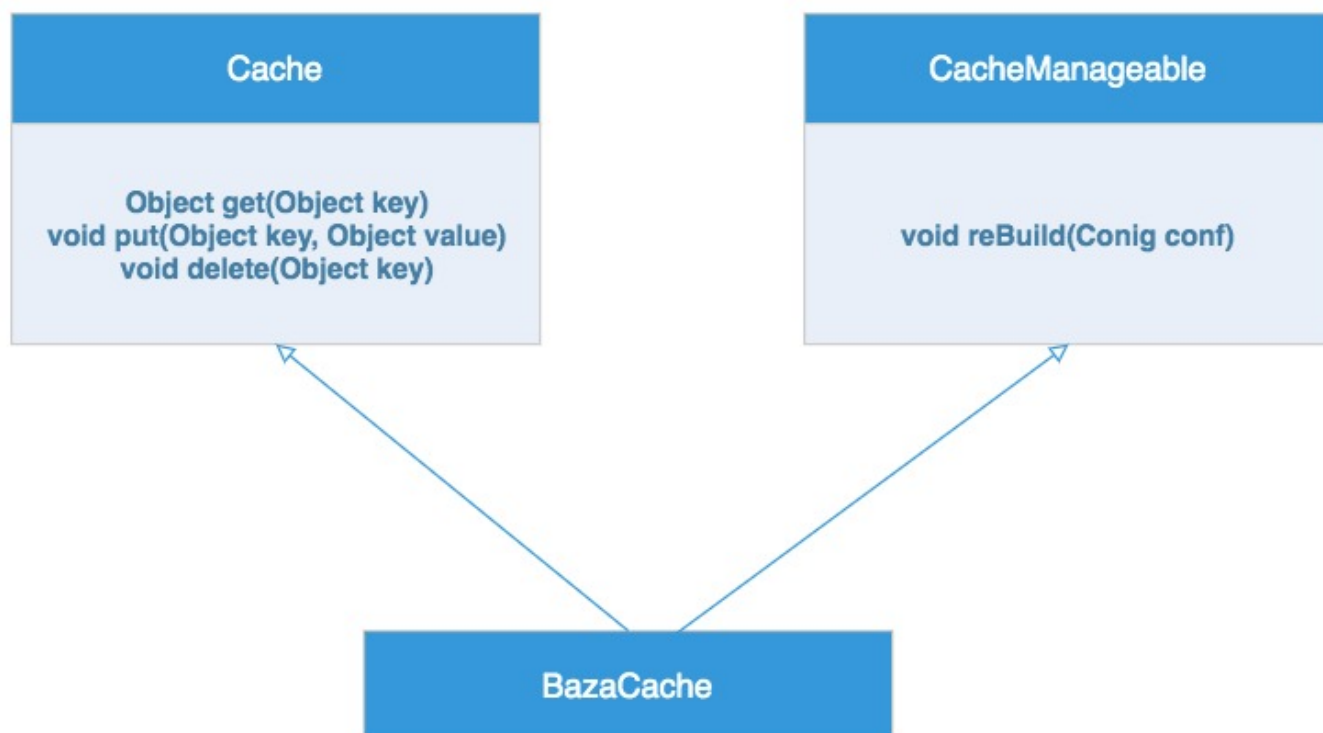
在 Java5 以前，每种容器的遍历方法都不相同，在 Java5 以后，可以统一使用这种简化的遍历语法实现对容器的遍历。而实现这一特性，主要就在于 Java5 通过 Iterable 接口，将容器的遍历访问从容器的其他操作中隔离出来，使 Java 可以针对这个接口进行优化，提供更加便利、简洁、统一的语法。

## 小结

我们再回到开头那个例子，如何让缓存类的使用者看不到缓存重构的方法，以避免不必要的依赖和方法的误用。答案就是使用接口隔离原则，通过多重继承的方式进行接口隔离。

Cache 实现类 BazaCache (Baza 是当时开发的统一缓存服务的产品名) 同时实现 Cache 接口和 CacheManageable 接口, 其中 Cache 接口提供标准的 Cache 服务方法, 应用程序只需要依赖该接口。而 CacheManageable 接口则对外暴露 reBuild() 方法, 使远程配置服务可以通过自己的本地代理调用这个方法, 在运行期远程调整缓存服务的配置, 使系统无需重新部署就可以热更新。

最后的缓存服务 SDK 核心类设计如下:



当一个类比较大的时候, 如果该类的不同调用者被迫依赖类的所有方法, 就可能产生不必要的耦合。对这个类的改动也可能会影响到它的不同调用者, 引起误用, 导致对象被破坏, 引发 bug。

使用接口隔离原则, 就是定义多个接口, 不同调用者依赖不同的接口, 只看到自己需要的方法。而实现类则实现这些接口, 通过多个接口将类内部不同的方法隔离开来。

## 思考题

在你的开发实践中, 你看到过哪些地方使用了接口隔离原则? 你自己开发的代码, 哪些地方可以用接口隔离原则优化?



欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

点击参加 21 天打卡计划 

# 搞定后端技术基础



扫一扫参与小程序话题



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 14 | 软件设计的单一职责原则：为什么说一个类文件打开最好不要超过一屏？

下一篇 16 | 设计模式基础：不会灵活应用设计模式，你就没有掌握面向对象编程

## 精选留言 (6)

 写留言



alex

2019-12-25

老师能加个代码么？自动锁门那块

展开 ∨



2



niuniu

2019-12-25

我觉得关键是合理的定义接口的粒度，实践中不同的场景可能同时需要用到同一个类的多个接口，还是要进行强转，让调用方很不爽。

展开 ∨



**QQ怪**

2019-12-30

又加深理解了，优秀

展开 ∨



**Paul Shan**

2019-12-29

接口隔离原则感觉比较简单，依赖什么就只定义需要的接口，代价是相似的接口被定义好几份，可以用接口间的继承一定程度上消除重复代码。

展开 ∨



**Zend**

2019-12-27

没想过，说实话 在写代码的时候太过于赶进度，没有对代码进行重构，更没有考虑到自己的设计的这个类如果方法都集中在一起不方便同事调用，让同事产生困惑。

展开 ∨



**山猫**

2019-12-25

接口隔离原则好是好，就是写着写着就发现接口越来越多，越来越多，甚至会超过本身类的数量，而且每个接口会只使用一次，这样不如直接用外观模式的IDE自动完成了。

我现在用接口主要用于一些模型的规范性和方法参数规范性。如果需要文章中的功能，会拆开为两个类来写，而不是用两个接口加两个类。...

展开 ∨

