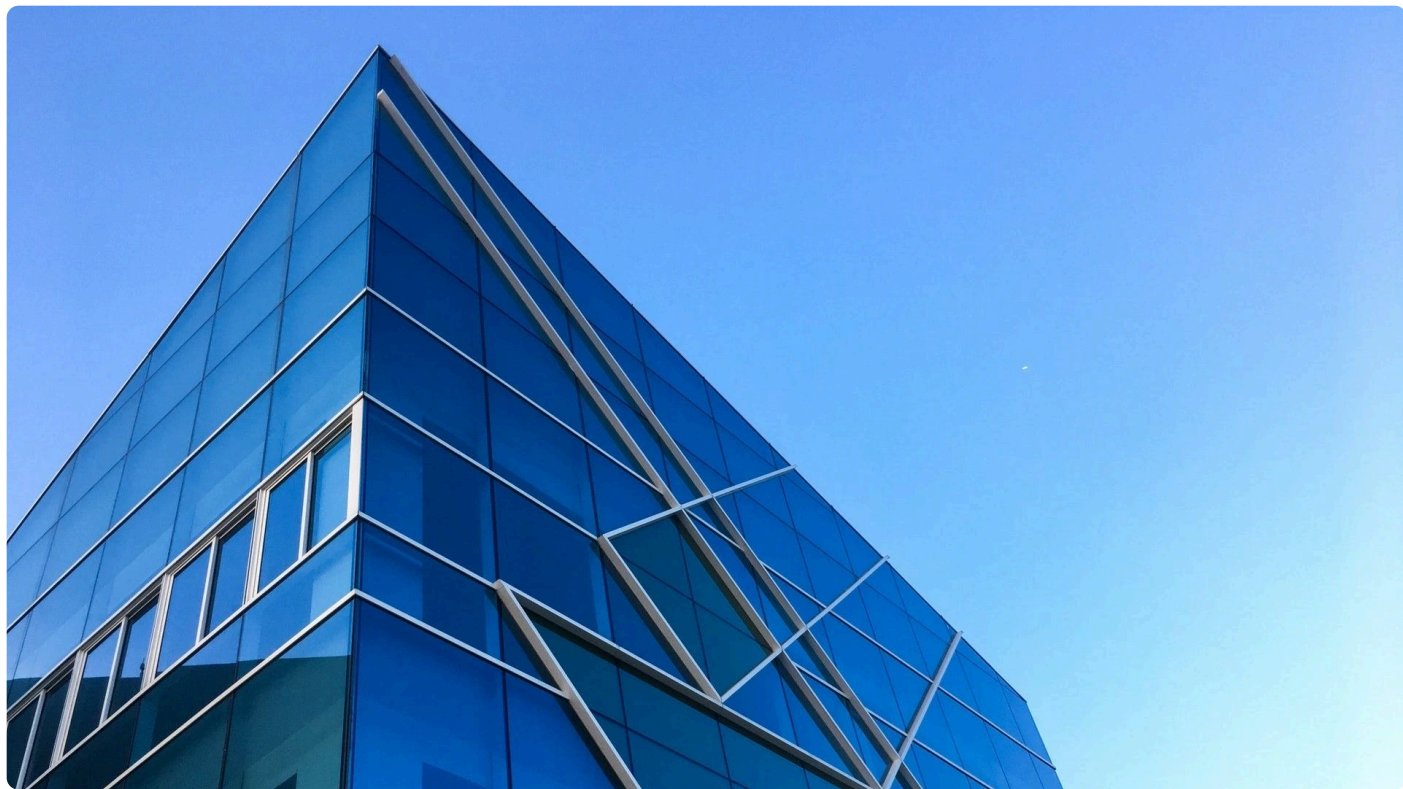


95 | 项目实战二：设计实现一个通用的接口幂等框架（实现）

王争 · 设计模式之美



上一节课，我们讲解了幂等框架的设计思路。在正常情况下，幂等框架的处理流程是比较简单的。调用方生成幂等号，传递给实现方，实现方记录幂等号或者用幂等号判重。但是，幂等框架要处理的异常情况很多，这也是设计的复杂之处和难点之处。比如，代码运行异常、业务系统宕机、幂等框架异常。

虽然幂等框架要处理的异常很多，但考虑到开发成本以及简单易用性，我们对某些异常的处理在工程上做了妥协，交由业务系统或者人工介入处理。这样就大大简化了幂等框架开发的复杂度和难度。

今天，我们针对幂等框架的设计思路，讲解如何编码实现。跟限流框架的讲解相同，对于幂等框架，我们也会还原它的整个开发过程，从 V1 版本需求、最小原型代码讲起，然后讲解如何 review 代码发现问题、重构代码解决问题，最终得到一份易读、易扩展、易维护、灵活、可测试的高质量代码实现。

话不多说，让我们正式开始今天的学习吧！

V1 版本功能需求

上一节课给出的设计思路比较零散，重点还是在讲设计的缘由，为什么要这么设计。今天，我们再重新整理一下，经过上一节课的分析梳理最终得到的设计思路。虽然上一节课的分析很复杂、很烧脑，但思从深而行从简，最终得到的幂等框架的设计思路是很简单的，主要包含下面这样两个主要的功能开发点：

实现生成幂等号的功能；

实现存储、查询、删除幂等号的功能。

因为功能非常简单，所以，我们就不再进行进一步裁剪了。在 V1 版本中，我们会实现上面罗列的所有功能。针对这两个功能点，我们先来说下实现思路。

我们先来看，如何生成幂等号。

幂等号用来标识两个接口请求是否是同一个业务请求，换句话说，两个接口请求是否是重试关系，而非独立的两个请求。接口调用方需要在发送接口请求的同时，将幂等号一块传递给接口实现方。那如何来生成幂等号呢？一般有两种生成方式。一种方式是集中生成并且分派给调用方，另一种方式是直接由调用方生成。

对于第一种生成方式，我们需要部署一套幂等号的生成系统，并且提供相应的远程接口（Restful 或者 RPC 接口），调用方通过调用远程接口来获取幂等号。这样做的好处是，对调用方完全隐藏了幂等号的实现细节。当我们需要改动幂等号的生成算法时，调用方不需要改动任何代码。

对于第二种生成方式，调用方按照跟接口实现方预先商量好的算法，自己来生成幂等号。这种实现方式的好处在于，不用像第一种方式那样调用远程接口，所以执行效率更高。但是，一旦需要修改幂等号的生成算法，就需要修改每个调用方的代码。

并且，每个调用方自己实现幂等号的生成算法也会有问题。一方面，重复开发，违反 DRY 原则。另一方面，工程师的开发水平层次不齐，代码难免会有 bug。除此之外，对于复杂的幂等

号生成算法，比如依赖外部系统 Redis 等，显然更加适合上一种实现方式，可以避免调用方为了使用幂等号引入新的外部系统。

权衡来讲，既考虑到生成幂等号的效率，又考虑到代码维护的成本，我们选择第二种实现方式，并且在此基础上做些改进，由幂等框架来统一提供幂等号生成算法的代码实现，并封装成开发类库，提供给各个调用方复用。除此之外，我们希望生成幂等号的算法尽可能的简单，不依赖其他外部系统。

实际上，对于幂等号的唯一要求就是全局唯一。全局唯一 ID 的生成算法有很多。比如，简单点的有取 UUID，复杂点的可以把应用名拼接在 UUID 上，方便做问题排查。总体上来讲，幂等号的生成算法并不难。

我们再来看，如何实现幂等号的存储、查询和删除。

从现在的需求来看，幂等号只是为了判重。在数据库中，我们只需要存储一个幂等号就可以，不需要太复杂的存储结构，所以，我们不选择使用复杂的关系型数据库，而是选择使用更加简单的、读写更加快速的键值数据库，比如 Redis。

在幂等判重逻辑中，我们需要先检查幂等号是否存在。如果没有存在，再将幂等号存储进 Redis。多个线程（同一个业务实例的多个线程）或者多进程（多个业务实例）同时执行刚刚的“检查 – 设置”逻辑时，就会存在竞争关系（竞态，race condition）。比如，A 线程检查幂等号不存在，在 A 线程将幂等号存储进 Redis 之前，B 线程也检查幂等号不存在，这样就会导致业务被重复执行。为了避免这种情况发生，我们要给“检查 – 设置”操作加锁，让同一时间只有一个线程能执行。除此之外，为了避免多进程之间的竞争，普通的线程锁还不起作用，我们需要分布式锁。

引入分布式锁会增加开发的难度和复杂度，而 Redis 本身就提供了把“检查 – 设置”操作作为原子操作执行的命令：setnx(key, value)。它先检查 key 是否存在，如果存在，则返回结果 0；如果不存在，则将 key 值存下来，并将值设置为 value，返回结果 1。因为 Redis 本身是单线程执行命令的，所以不存在刚刚讲到的并发问题。

最小原型代码实现

V1 版本要实现的功能和实现思路，现在已经很明确了。现在，我们来看下具体的代码实现。还是跟限流框架同样的实现方法，我们先不考虑设计和代码质量，怎么简单怎么来，先写出 MVP 代码，然后基于这个最简陋的版本做优化重构。

V1 版本的功能非常简单，我们用一个类就能搞定，代码如下所示。只用了不到 30 行代码，就搞定了——一个框架，是不是觉得有点不可思议。对于这段代码，你可以先思考下，有哪些值得优化的地方。


 复制代码

```
1 public class Idempotence {
2     private JedisCluster jedisCluster;
3
4     public Idempotence(String redisClusterAddress, GenericObjectPoolConfig config)
5     {
6         String[] addressArray= redisClusterAddress.split(";");
7         Set<HostAndPort> redisNodes = new HashSet<>();
8         for (String address : addressArray) {
9             String[] hostAndPort = address.split(":");
10            redisNodes.add(new HostAndPort(hostAndPort[0], Integer.valueOf(hostAndPort[1]));
11        }
12        this.jedisCluster = new JedisCluster(redisNodes, config);
13    }
14
15    public String genId() {
16        return UUID.randomUUID().toString();
17    }
18
19    public boolean saveIfAbsent(String idempotenceId) {
20        Long success = jedisCluster.setnx(idempotenceId, "1");
21        return success == 1;
22    }
23
24    public void delete(String idempotenceId) {
25        jedisCluster.del(idempotenceId);
26    }
27 }
```

Review 最小原型代码

尽管 MVP 代码很少，但仔细推敲，也有很多值得优化的地方。现在，我们就站在 Code Reviewer 的角度，分析一下这段代码。我把我的所有意见都放到代码注释中了，你可以对照

着代码一块看下。

 复制代码

```
1 public class Idempotence {
2     // comment-1: 如果要替换存储方式，是不是很麻烦呢？
3     private JedisCluster jedisCluster;
4
5     // comment-2: 如果幂等框架要跟业务系统复用jedisCluster连接呢？
6     // comment-3: 是不是应该注释说明一下redisClusterAddress的格式，以及config是否可以传递进
7     public Idempotence(String redisClusterAddress, GenericObjectPoolConfig config)
8         // comment-4: 这段逻辑放到构造函数里，不容易写单元测试呢
9         String[] addressArray= redisClusterAddress.split(";");
10        Set<HostAndPort> redisNodes = new HashSet<>();
11        for (String address : addressArray) {
12            String[] hostAndPort = address.split(":");
13            redisNodes.add(new HostAndPort(hostAndPort[0], Integer.valueOf(hostAndPort[
14        ]
15        this.jedisCluster = new JedisCluster(redisNodes, config);
16    }
17
18    // comment-5: generateId()是不是比缩写要好点？
19    // comment-6: 根据接口隔离原则，这个函数跟其他函数的使用场景完全不同，这个函数主要用在调用方
20    public String genId() {
21        return UUID.randomUUID().toString();
22    }
23
24    // comment-7: 返回值的意义是不是应该注释说明一下？
25    public boolean saveIfAbsent(String idempotenceId) {
26        Long success = jedisCluster.setnx(idempotenceId, "1");
27        return success == 1;
28    }
29
30    public void delete(String idempotenceId) {
31        jedisCluster.del(idempotenceId);
32    }
33 }
```

总结一下，MVP 代码主要涉及下面这样几个问题。

代码可读性问题：有些函数的参数和返回值的格式和意义不够明确，需要注释补充解释一下。genId() 函数使用了缩写，全拼 generateId() 可能更好些！

代码可扩展性问题：按照现在的代码实现方式，如果改变幂等号的存储方式和生成算法，代码修改起来会比较麻烦。除此之外，基于接口隔离原则，我们应该将 genId() 函数跟其他函数分离开来，放到两个类中。独立变化，隔离修改，更容易扩展！

代码可测试性问题：解析 Redis Cluster 地址的代码逻辑较复杂，但因为放到了构造函数中，无法对它编写单元测试。

代码灵活性问题：业务系统有可能希望幂等框架复用已经建立好的 jedisCluster，而不是单独给幂等框架创建一个 jedisCluster。

重构最小原型代码

实际上，问题找到了，修改起来就容易多了。针对刚刚罗列的几个问题，我们对 MVP 代码进行重构，重构之后的代码如下所示。

 复制代码

```
1 // 代码目录结构
2 com.xzg.cd.idempotence
3 --Idempotence
4 --IdempotenceIdGenerator(幂等号生成类)
5 --IdempotenceStorage(接口：用来读写幂等号)
6 --RedisClusterIdempotenceStorage(IdempotenceStorage的实现类)
7
8 // 每个类的代码实现
9 public class Idempotence {
10     private IdempotenceStorage storage;
11
12     public Idempotence(IdempotenceStorage storage) {
13         this.storage = storage;
14     }
15
16     public boolean saveIfAbsent(String idempotenceId) {
17         return storage.saveIfAbsent(idempotenceId);
18     }
19
20     public void delete(String idempotenceId) {
21         storage.delete(idempotenceId);
22     }
23 }
24
25 public class IdempotenceIdGenerator {
26     public String generateId() {
27         return UUID.randomUUID().toString();
28     }
29 }
```

```

28     }
29 }
30
31 public interface IdempotenceStorage {
32     boolean saveIfAbsent(String idempotenceId);
33     void delete(String idempotenceId);
34 }
35
36 public class RedisClusterIdempotenceStorage implements IdempotenceStorage {
37     private JedisCluster jedisCluster;
38
39     /**
40      * Constructor
41      * @param redisClusterAddress the format is 128.91.12.1:3455;128.91.12.2:3452;2
42      * @param config should not be null
43      */
44     public RedisIdempotenceStorage(String redisClusterAddress, GenericObjectPoolCon
45         Set<HostAndPort> redisNodes = parseHostAndPorts(redisClusterAddress);
46         this.jedisCluster = new JedisCluster(redisNodes, config);
47     }
48
49     public RedisIdempotenceStorage(JedisCluster jedisCluster) {
50         this.jedisCluster = jedisCluster;
51     }
52
53     /**
54      * Save {@idempotenceId} into storage if it does not exist.
55      * @param idempotenceId the idempotence ID
56      * @return true if the {@idempotenceId} is saved, otherwise return false
57      */
58     public boolean saveIfAbsent(String idempotenceId) {
59         Long success = jedisCluster.setnx(idempotenceId, "1");
60         return success == 1;
61     }
62
63     public void delete(String idempotenceId) {
64         jedisCluster.del(idempotenceId);
65     }
66
67     @VisibleForTesting
68     protected Set<HostAndPort> parseHostAndPorts(String redisClusterAddress) {
69         String[] addressArray= redisClusterAddress.split(";");
70         Set<HostAndPort> redisNodes = new HashSet<>();
71         for (String address : addressArray) {
72             String[] hostAndPort = address.split(":");
73             redisNodes.add(new HostAndPort(hostAndPort[0], Integer.valueOf(hostAndPort[
74         ]
75         return redisNodes;
76     }

```

接下来，我再总结罗列一下，针对之前发现的问题，我们都做了哪些代码改动。主要有下面这样几点，你可以结合着代码一块看下。

在代码可读性方面，我们对构造函数、`savelfAbsense()` 函数的参数和返回值做了注释，并且将 `genId()` 函数改为全拼 `generateId()`。不过，对于这个函数来说，缩写实际上问题也不大。

在代码可扩展性方面，我们按照基于接口而非实现的编程原则，将幂等号的读写独立出来，设计成 `IdempotenceStorage` 接口和 `RedisClusterIdempotenceStorage` 实现类。

`RedisClusterIdempotenceStorage` 实现了基于 Redis Cluster 的幂等号读写。如果我们需要替换新的幂等号读写方式，比如基于单个 Redis 而非 Redis Cluster，我们就可以再定义一个实现了 `IdempotenceStorage` 接口的实现类：`RedisIdempotenceStorage`。

除此之外，按照接口隔离原则，我们将生成幂等号的代码抽离出来，放到 `IdempotenceIdGenerator` 类中。这样，调用方只需要依赖这个类的代码就可以了。幂等号生成算法的修改，跟幂等号存储逻辑的修改，两者完全独立，一个修改不会影响另外一个。

在代码可测试性方面，我们把原本放在构造函数中的逻辑抽离出来，放到了 `parseHostAndPorts()` 函数中。这个函数本应该是 `Private` 访问权限的，但为了方便编写单元测试，我们把它设置为成了 `Protected` 访问权限，并且通过注解 `@VisibleForTesting` 做了标明。

在代码灵活性方面，为了方便复用业务系统已经建立好的 `jedisCluster`，我们提供了一个新的构造函数，支持业务系统直接传递 `jedisCluster` 来创建 `Idempotence` 对象。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

我们前面花了两节课的时间，用很大的篇幅在讲需求和设计，特别是设计的缘由。而真正到了实现环节，我们只用了不到 30 行代码，就实现了幂等框架。这就很好体现了“思从深而行从

简”的道理。对于不到 30 行代码，很多人觉得不大可能有啥优化空间了，但我们今天还是提出了 7 个优化建议，并且对代码结构做了比较大的调整。这说明，只要仔细推敲，再小的代码都有值得优化的地方。

不过，之前有人建议我举一些大型项目中的例子，最好是上万行代码的那种，不要举这种几十行的小例子。大项目和小项目在编码这个层面，实际上没有太大区别。再宏大的工程、再庞大的项目，也是一行一行写出来的。那些上来就要看上万行代码，分析庞大项目的，大部分都还没有理解编码的精髓。编码本身就是一个很细节的事情，牛不牛也都隐藏在一行一行的代码中。空谈架构、设计、大道理，实际上没有太多意义，对你帮助不大。能沉下心来把细节都做好那才是真的牛！

课堂讨论

1. 针对 MVP 代码，我有两个问题留给你思考。其中一个问题是，`delete()` 是应该返回 `void` 值还是 `boolean` 值？如果删除出错，应该如何处理？另一个问题是，需不需要给幂等号生成算法抽象出一个接口呢？为什么？
2. 在后续的版本规划中，你觉得幂等框架还可以继续扩展哪些功能？或者做哪些优化？如果让你规划第二个版本，你会做哪些东西？

欢迎留言和我分享你的想法。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

设计和实现通用的接口幂等框架是本文的核心内容。文章首先介绍了幂等框架的设计思路和异常处理流程，讨论了幂等号的生成方式，并选择了由调用方自行生成的实现方式。此外，文章还详细介绍了幂等号的存储、查询和删除的实现思路，以及使用Redis作为键值数据库的原因。在最小原型代码实现部分，文章给出了一个简单的Idempotence类的代码实现，展示了如何使用不到30行代码实现一个幂等框架。接着，文章对MVP代码进行了重构，解决了代码可读性、可扩展性、可测试性和灵活性等方面的问题。重构后的代码结构更加清晰，将幂等号的读写独立出来，设计成了IdempotenceStorage接口和RedisClusterIdempotenceStorage实现类。同时，幂等号生成算法也被抽离出来，放到了IdempotenceIdGenerator类中。这样的设计使得代码更易于扩展和维护。总的来说，本文通过简洁清晰的语言，深入浅出地介绍了幂等框架的设计思路和代码实现，为读者提供了一份易读、易扩展、易维护、灵活、可测试的高质量代码实现。

© 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 (35)

最新 精选



高源

2020-06-10

老师讲的真的好后期老师把所讲课程配套代码提供上，我对着再仔细阅读一次结合实际应用到实际开发中，谢谢

作者回复: 可以的，我现在有空了，我最近就能补上，关注我的github吧：

<https://github.com/wangzheng0822>

共 3 条评论 >

👍 12



Jason

2020-06-10

RedisClusterIdempotenceStorage是不是少implements IdempotenceStorage?

作者回复: 嗯嗯 多谢指出~



👍 9



cricket1981

2020-06-26

有一点不明白，框架为什么要对外提供delete接口呢？调用方乱用怎么办？如果是为了处理系统异常，我觉得可以让调用方重新生成新的幂等号再发起请求，而不是依赖调用方在重试之前正确地调用delete接口。

作者回复: 不是重试之前删除，而是在出错之后删除。重试的时候并不知道原来是执行正确还是错误、



👍 3

北纬8°

北纬8°

2020-07-08

老师，能帮菜鸟讲解下幂等咋就实现了，区分请求，是一个新的请求或者是同一个请求再次发送

作者回复: 幂等号相同就是同一个请求。两个请求是否是同一个请求, 由业务发起方来定义 (他说了算), 他觉得某两个请求是同一个请求, 就赋予相同的幂等号。

共 8 条评论 >

👍 1



小晏子

2020-06-10

delete是否返回boolean和如果出错该如何处理这个问题, 要看业务方是处理业务和幂等号的顺序, 如果先存储幂等号, 在做业务, 那么业务没处理成功时, 后续处理就要删除幂等号, 然后重复业务处理, 这就要保证删除幂等号一定要成功, 这样就要返回boolean值; 相反, 如果业务处理成功后在保存幂等号, 那么删除幂等号成功与否都无关, 删除幂等号可以不返回值。幂等号生成算法没必要再开个接口, 因为幂等号生成算法需要稳定性全局性, 否则不同业务用不同算法生成幂等号, 那么幂等框架就可能无法区分不同的业务请求了。

后续版本可以让幂等框架更易用, 不需要过多配置, 比如提供一个注解, 使用方直接在需要幂等的接口上添加上这个注解, 那么这个请求就一定会保证幂等。

共 6 条评论 >

👍 38



Jxin

2020-06-10

1.针对mvp代码, delete不该返回boolean, void即可。因为该方法只有在技术异常 (网络超时, redis节点无法提供服务) 时才会有失败的场景。而我的习惯, 是把技术异常放在最外层处理, 或则代理层处理 (技术异常的处理应该尽量与业务代码分离)。如果delete里面也有业务逻辑, 比如入参检验, 那么我会返回boolean。因为这时候的异常是业务代码该处理的场景, 同时我认为调用方无需知道delete因何失败, 只需要知道delete失败, 所以收敛delete内部业务异常, 对外只以boolean返回值做交互。

2.因为唯一id的需求方并不只有幂等框架, 抽离出来在其他场景也能用, 比如流水id。我认为幂等服务方提供幂等id生成接口给调用方的方式并不合适。这样做是一种资源的浪费。在这个场景, 我只是为了保证幂等id生成代码的去重, 并没有动态上扩缩节点调整负载的诉求。所以幂等id生成代码以公共包的方式被调用方项目依赖, 仅做代码级的复用会好些。

3.抽成公共包 (去重), 提供声明式接口 (易用), 提供配置接口 (灵活)。



👍 19



88591

2020-07-16

能沉下心来把细节都做好那才是真的牛！



👍 8



Jemmy

2020-06-14

优化：IdempotenceldGenerator 可以面向接口编程，未来 uuid 生成方式可能会变。



👍 4



Jackey

2020-06-10

delete还是返回boolean好一点，对删除出错的key可以人工干预或者记录下来统一处理



👍 4



lengrongfu

2020-08-06

作为框架功能，应该要把异常上抛给调用方，让调用方自己决定如何处理；现在通用的功能框架几乎都是这么做的。



👍 3