

35 | 实战一（下）：手把手带你将ID生成器代码从“能用”重构为“好用”

王争 · 设计模式之美



上一节课中，我们结合 ID 生成器代码讲解了如何发现代码质量问题。虽然 ID 生成器的需求非常简单，代码行数也不多，但看似非常简单的代码，实际上还是有很多优化的空间。综合评价一下的话，小王的代码也只能算是“能用”、勉强及格。我们大部分人写出来的代码都能达到这个程度。如果想要在团队中脱颖而出，我们就不能只满足于这个 60 分及格，大家都能做的事情，我们要做得更好才行。

上一节课我们讲了，为什么这份代码只能得 60 分，这一节课我们再讲一下，如何将 60 分的代码重构为 80 分、90 分，让它从“能用”变得“好用”。话不多说，让我们正式开始今天的学习吧！

回顾代码和制定重构计划

为了方便你查看和对比，我把上一节课中的代码拷贝到这里。

```
1 public class IdGenerator {
```

 复制代码

```

2     private static final Logger logger = LoggerFactory.getLogger(IdGenerator.class)
3
4     public static String generate() {
5         String id = "";
6         try {
7             String hostName = InetAddress.getLocalHost().getHostName();
8             String[] tokens = hostName.split("\\.");
9             if (tokens.length > 0) {
10                 hostName = tokens[tokens.length - 1];
11             }
12             char[] randomChars = new char[8];
13             int count = 0;
14             Random random = new Random();
15             while (count < 8) {
16                 int randomAscii = random.nextInt(122);
17                 if (randomAscii >= 48 && randomAscii <= 57) {
18                     randomChars[count] = (char)('0' + (randomAscii - 48));
19                     count++;
20                 } else if (randomAscii >= 65 && randomAscii <= 90) {
21                     randomChars[count] = (char)('A' + (randomAscii - 65));
22                     count++;
23                 } else if (randomAscii >= 97 && randomAscii <= 122) {
24                     randomChars[count] = (char)('a' + (randomAscii - 97));
25                     count++;
26                 }
27             }
28             id = String.format("%s-%d-%s", hostName,
29                             System.currentTimeMillis(), new String(randomChars));
30         } catch (UnknownHostException e) {
31             logger.warn("Failed to get the host name.", e);
32         }
33
34         return id;
35     }
36 }

```

前面讲到系统设计和实现的时候，我们多次讲到要循序渐进、小步快跑。重构代码的过程也应该遵循这样的思路。每次改动一点点，改好之后，再进行下一轮的优化，保证每次对代码的改动不会过大，能在很短的时间内完成。所以，我们将上一节课中发现的代码质量问题，分成四次重构来完成，具体如下所示。

第一轮重构：提高代码的可读性

第二轮重构：提高代码的可测试性

第三轮重构：编写完善的单元测试

第四轮重构：所有重构完成之后添加注释

第一轮重构：提高代码的可读性

首先，我们要解决最明显、最急需改进的代码可读性问题。具体有下面几点：

hostName 变量不应该被重复使用，尤其当这两次使用时的含义还不同的时候；

将获取 hostName 的代码抽离出来，定义为 getLastfieldOfHostName() 函数；

删除代码中的魔法数，比如，57、90、97、122；

将随机数生成的代码抽离出来，定义为 generateRandomAlphameric() 函数；

generate() 函数中的三个 if 逻辑重复了，且实现过于复杂，我们要对其进行简化；

对 IdGenerator 类重命名，并且抽象出对应的接口。

这里我们重点讨论下最后一个修改。实际上，对于 ID 生成器的代码，有下面三种类的命名方式。你觉得哪种更合适呢？



	接口	实现类
命名方式一	IdGenerator	LogTraceIdGenerator
命名方式二	LogTraceIdGenerator	HostNameMillisIdGenerator
命名方式三	LogTraceIdGenerator	RandomIdGenerator


我们来逐一分析一下三种命名方式。

第一种命名方式，将接口命名为 IdGenerator，实现类命名为 LogTraceIdGenerator，这可能是很多人最先想到的命名方式了。在命名的时候，我们要考虑到，以后两个类会如何使用、会

如何扩展。从使用和扩展的角度来分析，这样的命名就不合理了。

首先，如果我们扩展新的日志 ID 生成算法，也就是要创建另一个新的实现类，因为原来的实现类已经叫 LogTraceIdGenerator 了，命名过于通用，那新的实现类就只好取名了，无法取一个跟 LogTraceIdGenerator 平行的名字了。

其次，你可能会说，假设我们没有日志 ID 的扩展需求，但要扩展其他业务的 ID 生成算法，比如针对用户的（UserIdGenerator）、订单的（OrderIdGenerator），第一种命名方式是不是就是合理的呢？答案也是否定的。基于接口而非实现编程，主要的目的是为了后续灵活地替换实现类。而 LogTraceIdGenerator、UserIdGenerator、OrderIdGenerator 三个类从命名上来看，涉及的是完全不同的业务，不存在互相替换的场景。也就是说，我们不可能在有关日志的代码中，进行下面这种替换。所以，让这三个类实现同一个接口，实际上是没有意义的。

 复制代码

```
1 IdGenerator idGenerator = new LogTraceIdGenerator();  
2 替换为：  
3 IdGenerator idGenerator = new UserIdGenerator();
```

第二种命名方式是不是就合理了呢？答案也是否定的。其中，LogTraceIdGenerator 接口的命名是合理的，但是 HostNameMillisIdGenerator 实现类暴露了太多实现细节，只要代码稍微有所改动，就可能需要改动命名，才能匹配实现。

第三种命名方式是我比较推荐的。在目前的 ID 生成器代码实现中，我们生成的 ID 是一个随机 ID，不是递增有序的，所以，命名成 RandomIdGenerator 是比较合理的，即便内部生成算法有所改动，只要生成的还是随机的 ID，就不需要改动命名。如果我们需要扩展新的 ID 生成算法，比如要实现一个递增有序的 ID 生成算法，那我们可以命名为 SequenceIdGenerator。

实际上，更好的一种命名方式是，我们抽象出两个接口，一个是 IdGenerator，一个是 LogTraceIdGenerator，LogTraceIdGenerator 继承 IdGenerator。实现类实现接口

LogTraceIdGenerator，命名为 RandomIdGenerator、SequencIdGenerator 等。这样，实现类可以复用到多个业务模块中，比如前面提到的用户、订单。

根据上面的优化策略，我们对代码进行第一轮的重构，重构之后的代码如下所示：

 复制代码

```
1 public interface IdGenerator {
2     String generate();
3 }
4
5 public interface LogTraceIdGenerator extends IdGenerator {
6 }
7
8 public class RandomIdGenerator implements LogTraceIdGenerator {
9     private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.class);
10
11     @Override
12     public String generate() {
13         String substrOfHostName = getLastfieldOfHostName();
14         long currentTimeMillis = System.currentTimeMillis();
15         String randomString = generateRandomAlphameric(8);
16         String id = String.format("%s-%d-%s",
17             substrOfHostName, currentTimeMillis, randomString);
18         return id;
19     }
20
21     private String getLastfieldOfHostName() {
22         String substrOfHostName = null;
23         try {
24             String hostName = InetAddress.getLocalHost().getHostName();
25             String[] tokens = hostName.split("\\.");
26             substrOfHostName = tokens[tokens.length - 1];
27             return substrOfHostName;
28         } catch (UnknownHostException e) {
29             logger.warn("Failed to get the host name.", e);
30         }
31         return substrOfHostName;
32     }
33
34     private String generateRandomAlphameric(int length) {
35         char[] randomChars = new char[length];
36         int count = 0;
37         Random random = new Random();
38         while (count < length) {
39             int maxAscii = 'z';
40             int randomAscii = random.nextInt(maxAscii);
```

```
41     boolean isDigit= randomAscii >= '0' && randomAscii <= '9';
42     boolean isUppercase= randomAscii >= 'A' && randomAscii <= 'Z';
43     boolean isLowercase= randomAscii >= 'a' && randomAscii <= 'z';
44     if (isDigit|| isUppercase || isLowercase) {
45         randomChars[count] = (char) (randomAscii);
46         ++count;
47     }
48 }
49 return new String(randomChars);
50 }
51 }
52
53 //代码使用举例
54 LogTraceIdGenerator logTraceIdGenerator = new RandomIdGenerator();
```

第二轮重构：提高代码的可测试性

关于代码可测试性的问题，主要包含下面两个方面：

generate() 函数定义为静态函数，会影响使用该函数的代码的可测试性；

generate() 函数的代码实现依赖运行环境（本机名）、时间函数、随机函数，所以 generate() 函数本身的可测试性也不好。


对于第一点，我们已经在第一轮重构中解决了。我们将 RandomIdGenerator 类中的 generate() 静态函数重新定义成了普通函数。调用者可以通过依赖注入的方式，在外部创建好 RandomIdGenerator 对象后注入到自己的代码中，从而解决静态函数调用影响代码可测试性的问题。

对于第二点，我们需要在第一轮重构的基础之上再进行重构。重构之后的代码如下所示，主要包括以下几个代码改动。

从 getLastfieldOfHostName() 函数中，将逻辑比较复杂的那部分代码剥离出来，定义为 getLastSubstrSplittedByDot() 函数。因为 getLastfieldOfHostName() 函数依赖本地主机名，所以，剥离出主要代码之后这个函数变得非常简单，可以不用测试。我们重点测试 getLastSubstrSplittedByDot() 函数即可。

将 `generateRandomAlphameric()` 和 `getLastSubstrSplittedByDot()` 这两个函数的访问权限设置为 `protected`。这样做的目的是，可以直接在单元测试中通过对象来调用两个函数进行测试。

给 `generateRandomAlphameric()` 和 `getLastSubstrSplittedByDot()` 两个函数添加 Google Guava 的 annotation `@VisibleForTesting`。这个 annotation 没有任何实际的作用，只起到标识的作用，告诉其他人说，这两个函数本该是 `private` 访问权限的，之所以提升访问权限到 `protected`，只是为了测试，只能用于单元测试中。

 复制代码

```
1 public class RandomIdGenerator implements LogTraceIdGenerator {
2     private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.
3
4     @Override
5     public String generate() {
6         String substrOfHostName = getLastfieldOfHostName();
7         long currentTimeMillis = System.currentTimeMillis();
8         String randomString = generateRandomAlphameric(8);
9         String id = String.format("%s-%d-%s",
10             substrOfHostName, currentTimeMillis, randomString);
11         return id;
12     }
13
14     private String getLastfieldOfHostName() {
15         String substrOfHostName = null;
16         try {
17             String hostName = InetAddress.getLocalHost().getHostName();
18             substrOfHostName = getLastSubstrSplittedByDot(hostName);
19         } catch (UnknownHostException e) {
20             logger.warn("Failed to get the host name.", e);
21         }
22         return substrOfHostName;
23     }
24
25     @VisibleForTesting
26     protected String getLastSubstrSplittedByDot(String hostName) {
27         String[] tokens = hostName.split("\\.");
28         String substrOfHostName = tokens[tokens.length - 1];
29         return substrOfHostName;
30     }
31
32     @VisibleForTesting
33     protected String generateRandomAlphameric(int length) {
34         char[] randomChars = new char[length];
35         int count = 0;
```

```

36     Random random = new Random();
37     while (count < length) {
38         int maxAscii = 'z';
39         int randomAscii = random.nextInt(maxAscii);
40         boolean isDigit= randomAscii >= '0' && randomAscii <= '9';
41         boolean isUppercase= randomAscii >= 'A' && randomAscii <= 'Z';
42         boolean isLowercase= randomAscii >= 'a' && randomAscii <= 'z';
43         if (isDigit|| isUppercase || isLowercase) {
44             randomChars[count] = (char) (randomAscii);
45             ++count;
46         }
47     }
48     return new String(randomChars);
49 }
50 }

```


在上一节课的课堂讨论中，我们提到，打印日志的 Logger 对象被定义为 static final 的，并且在类内部创建，这是否影响到代码的可测试性？是否应该将 Logger 对象通过依赖注入的方式注入到类中呢？

依赖注入之所以能提高代码可测试性，主要是因为，通过这样的方式我们能轻松地用 mock 对象替换依赖的真实对象。那我们为什么要 mock 这个对象呢？这是因为，这个对象参与逻辑执行（比如，我们要依赖它输出的数据做后续的计算）但又不可控。对于 Logger 对象来说，我们只往里写入数据，并不读取数据，不参与业务逻辑的执行，不会影响代码逻辑的正确性，所以，我们没有必要 mock Logger 对象。

除此之外，一些只是为了存储数据的值对象，比如 String、Map、UseVo，我们也没必要通过依赖注入的方式来创建，直接在类中通过 new 创建就可以了。

第三轮重构：编写完善的单元测试

经过上面的重构之后，代码存在的比较明显的问题，基本上都已经解决了。我们现在为代码补全单元测试。RandomIdGenerator 类中有 4 个函数。

 复制代码

```


1 public String generate();
2 private String getLastfieldOfHostName();

```



```
3 @VisibleForTesting
4 protected String getLastSubstrSplittedByDot(String hostName);
5 @VisibleForTesting
6 protected String generateRandomAlphameric(int length);
```

我们先来看后两个函数。这两个函数包含的逻辑比较复杂，是我们测试的重点。而且，在上一步重构中，为了提高代码的可测试性，我们已经将这两个部分代码跟不可控的组件（本机名、随机函数、时间函数）进行了隔离。所以，我们只需要设计完备的单元测试用例即可。具体的代码实现如下所示（注意，我们使用了 JUnit 测试框架）：

 复制代码

```
1 public class RandomIdGeneratorTest {
2     @Test
3     public void testGetLastSubstrSplittedByDot() {
4         RandomIdGenerator idGenerator = new RandomIdGenerator();
5         String actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1.field2.f
6         Assert.assertEquals("field3", actualSubstr);
7
8         actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1");
9         Assert.assertEquals("field1", actualSubstr);
10
11         actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1#field2#field3")
12         Assert.assertEquals("field1#field2#field3", actualSubstr);
13     }
14
15     // 此单元测试会失败，因为我们在代码中没有处理hostName为null或空字符串的情况
16     // 这部分优化留在第36、37节课中讲解
17     @Test
18     public void testGetLastSubstrSplittedByDot_nullOrEmpty() {
19         RandomIdGenerator idGenerator = new RandomIdGenerator();
20         String actualSubstr = idGenerator.getLastSubstrSplittedByDot(null);
21         Assert.assertNull(actualSubstr);
22
23         actualSubstr = idGenerator.getLastSubstrSplittedByDot("");
24         Assert.assertEquals("", actualSubstr);
25     }
26
27     @Test
28     public void testGenerateRandomAlphameric() {
29         RandomIdGenerator idGenerator = new RandomIdGenerator();
30         String actualRandomString = idGenerator.generateRandomAlphameric(6);
31         Assert.assertNotNull(actualRandomString);
32         Assert.assertEquals(6, actualRandomString.length());
33         for (char c : actualRandomString.toCharArray()) {
```

```

34         Assert.assertTrue(('0' <= c && c <= '9') || ('a' <= c && c <= 'z') || ('
35     }
36 }
37
38 // 此单元测试会失败，因为我们在代码中没有处理length<=0的情况
39 // 这部分优化留在第36、37节课中讲解
40 @Test
41 public void testGenerateRandomAlphameric_lengthEqualsOrLessThanZero() {
42     RandomIdGenerator idGenerator = new RandomIdGenerator();
43     String actualRandomString = idGenerator.generateRandomAlphameric(0);
44     Assert.assertEquals("", actualRandomString);
45
46     actualRandomString = idGenerator.generateRandomAlphameric(-1);
47     Assert.assertNull(actualRandomString);
48 }
49 }

```

我们再来看 generate() 函数。这个函数也是我们唯一一个暴露给外部使用的 public 函数。虽然逻辑比较简单，最好还是测试一下。但是，它依赖主机名、随机函数、时间函数，我们该如何测试呢？需要 mock 这些函数的实现吗？

实际上，这要分情况来看。我们前面讲过，写单元测试的时候，测试对象是函数定义的功能，而非具体的实现逻辑。这样我们才能做到，函数的实现逻辑改变了之后，单元测试用例仍然可以工作。那 generate() 函数实现的功能是什么呢？这完全是由代码编写者自己来定义的。

比如，针对同一份 generate() 函数的代码实现，我们可以有 3 种不同的功能定义，对应 3 种不同的单元测试。

1. 如果我们把 generate() 函数的功能定义为：“生成一个随机唯一 ID”，那我们只要测试多次调用 generate() 函数生成的 ID 是否唯一即可。
2. 如果我们把 generate() 函数的功能定义为：“生成一个只包含数字、大小写字母和中划线的唯一 ID”，那我们不仅要测试 ID 的唯一性，还要测试生成的 ID 是否只包含数字、大小写字母和中划线。
3. 如果我们把 generate() 函数的功能定义为：“生成唯一 ID，格式为：{主机名 substr}-{时间戳}-{8 位随机数}。在主机名获取失败时，返回：null-{时间戳}-{8 位随机数}”，那我们不仅要测试 ID 的唯一性，还要测试生成的 ID 是否完全符合格式要求。

总结一下，单元测试用例如何写，关键看你如何定义函数。针对 generate() 函数的前两种定义，我们不需要 mock 获取主机名函数、随机函数、时间函数等，但对于第 3 种定义，我们需要 mock 获取主机名函数，让其返回 null，测试代码运行是否符合预期。

最后，我们来看下 getLastfieldOfHostName() 函数。实际上，这个函数不容易测试，因为它调用了一个静态函数 (InetAddress.getLocalHost().getHostName();)，并且这个静态函数依赖运行环境。但是，这个函数的实现非常简单，肉眼基本上可以排除明显的 bug，所以我们可以不为其编写单元测试代码。毕竟，我们写单元测试的目的是为了减少代码 bug，而不是为了写单元测试而写单元测试。

当然，如果你真的想要对它进行测试，我们也是有办法的。一种办法是使用更加高级的测试框架。比如 PowerMock，它可以 mock 静态函数。另一种方式是将获取本机名的逻辑再封装为一个新的函数。不过，后一种方法会造成代码过度零碎，也会稍微影响到代码的可读性，这个需要你自己去权衡利弊来做选择。

第四轮重构：添加注释

前面我们提到，注释不能太多，也不能太少，主要添加在类和函数上。有人说，好的命名可以替代注释，清晰的表达含义。这点对于变量的命名来说是适用的，但对于类或函数来说就不一定对了。类或函数包含的逻辑往往比较复杂，单纯靠命名很难清晰地表明实现了什么功能，这个时候我们就需要通过注释来补充。比如，前面我们提到的对于 generate() 函数的 3 种功能定义，就无法用命名来体现，需要补充到注释里面。

对于如何写注释，你可以参看我们在 [第 31 节课](#) 中的讲解。总结一下，主要就是写清楚：做什么、为什么、怎么做、怎么用，对一些边界条件、特殊情况进行说明，以及对函数输入、输出、异常进行说明。

 复制代码

```
1  /**
2   * Id Generator that is used to generate random IDs.
3   *
4   * <p>
5   * The IDs generated by this class are not absolutely unique,
6   * but the probability of duplication is very low.
7   */
```

```

8 public class RandomIdGenerator implements LogTraceIdGenerator {
9     private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.
10
11     /**
12      * Generate the random ID. The IDs may be duplicated only in extreme situation.
13      *
14      * @return an random ID
15      */
16     @Override
17     public String generate() {
18         //...
19     }
20
21     /**
22      * Get the local hostname and
23      * extract the last field of the name string splitted by delimiter '.'.
24      *
25      * @return the last field of hostname. Returns null if hostname is not obtained
26      */
27     private String getLastfieldOfHostName() {
28         //...
29     }
30
31     /**
32      * Get the last field of {@hostName} splitted by delemiter '.'.
33      *
34      * @param hostName should not be null
35      * @return the last field of {@hostName}. Returns empty string if {@hostName} i
36      */
37     @VisibleForTesting
38     protected String getLastSubstrSplittedByDot(String hostName) {
39         //...
40     }
41
42     /**
43      * Generate random string which
44      * only contains digits, uppercase letters and lowercase letters.
45      *
46      * @param length should not be less than 0
47      * @return the random string. Returns empty string if {@length} is 0
48      */
49     @VisibleForTesting
50     protected String generateRandomAlphameric(int length) {
51         //...
52     }
53 }

```

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，你需要掌握的重点内容。

在这节课中，我带你将小王写的凑活能用的代码，重构成了结构更加清晰、更加易读、更易测试的代码，并且为其补全了单元测试。这其中涉及的知识点都是我们在理论篇中讲过的内容，比较细节和零碎，我就不一一带你回顾了，如果哪里不是很清楚，你可以回到前面章节去复习一下。

实际上，通过这节课，我更想传达给你的是下面这样几个开发思想，我觉得这比我给你讲解具体的知识点更加有意义。

1. 即便是非常简单需求，不同水平的人写出来的代码，差别可能会很大。我们要对代码质量有所追求，不能只是凑活能用就好。花点心思写一段高质量的代码，比写 100 段凑活能用的代码，对你的代码能力提高更有帮助。
2. 知其然知其所以然，了解优秀代码设计的演变过程，比学习优秀设计本身更有价值。知道为什么这么做，比单纯地知道怎么做更重要，这样可以避免你过度使用设计模式、思想和原则。
3. 设计思想、原则、模式本身并没有太多“高大上”的东西，都是一些简单的道理，而且知识点也并不多，关键还是锻炼具体代码具体分析的能力，把知识点恰当地用在项目中。
4. 我经常讲，高手之间的竞争都是在细节。大的架构设计、分层、分模块思路实际上都差不多。没有项目是靠一些不为人知的设计来取胜的，即便有，很快也能被学习过去。所以，关键还是看代码细节处理得够不够好。这些细节的差别累积起来，会让代码质量有质的差别。所以，要想提高代码质量，还是要在细节处下功夫。

课堂讨论

1. 获取主机名失败的时候，`generate()` 函数应该返回什么最合适呢？是特殊 ID、null、空字符，还是异常？在小王的代码实现中，获取主机名失败异常在 `IdGenerator` 内部被吞掉了，打印一条报警日志，并没有继续往上抛出，这样的异常处理是否得当？
2. 为了隐藏代码实现细节，我们把 `getLastSubstrSplittedByDot(String hostName)` 函数命名替换成 `getLastSubstrByDelimiter(String hostName)`，这样是否更加合理？为什么？

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

本文深入介绍了通过逐步重构的方式，将一个ID生成器的代码从“能用”重构为“好用”。作者提出了四次重构计划，包括提高代码的可读性、可测试性、编写完善的单元测试以及添加注释。文章详细讲解了第一轮重构的具体步骤，包括对接口和实现类的命名方式进行了深入分析。接着，介绍了第二轮重构，主要包括将generate()函数定义为普通函数，以及对两个函数的访问权限和注释进行了调整。在第三轮重构中，作者为代码补全了单元测试，重点测试了逻辑复杂的部分。最后，文章讨论了如何定义函数的功能，并根据不同的功能定义编写了相应的单元测试。通过实际案例深入讲解了代码重构的过程和技巧，对于想要提高代码质量的开发者具有很高的参考价值。文章还强调了对代码质量的追求、了解优秀代码设计的演变过程、细节处理对代码质量的重要性等开发思想。

© 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 (125)

最新 精选



马哲富

2020-02-15

看到有人说这个专栏写得不好，我忍不住要留个言给这个专栏叫叫好，这个专栏写得很好，非常好，只恨自己水平有限，不能完全吸收，顶这个专栏！

编辑回复: 哈哈，没事的，各有自己的判断，不可能让大家都觉得好，我们虚心相待，尽力而为。遇到问题，解决问题。

共 10 条评论 >

👍 59



辣么大

2020-01-22

对于在ID generator中方法里写到

```
void foo(){
```

```
    Random random = new Random();
```

```
}
```

有个疑问：

- 1、为什么不声明成静态变量？
- 2、能用成员变量么？而不是写成局部变量

作者回复: 也可以，不过尽量的缩小变量的作用域，代码可读性也好，毕竟random只会用在某个函数中，而不是用在多个函数中，放到局部函数中，也符合封装的特性，不暴露太多细节。

共 3 条评论 >

👍 13



evolution

2020-01-22

代码的演变过程，真的是干货满满。不知道争哥有没有架构方面的演变课程？

作者回复: 感谢认可，暂时没有呢

共 2 条评论 >

👍 10



提姆

2020-07-13

老師你好，想問一下有關測試的問題RandomIdGeneratorTest，為什麼不是分幾個Test Case去對generate做測試而是要拆出protected方法去做測試呢？

作者回复: 拆出来的目的并非为了单元测试，更重要的是逻辑清晰，可读性好。之所以设置成protected的，是因为private的没法写单元测试。

共 3 条评论 >

👍 6



牛顿的烈焰激光剑

2020-01-25

老师，对于获取 hostname (getLastfieldOfHostName())，我的想法是用 static 代码块，只在类加载的时候执行一次。请问这样处理的话会不会有什么坏处？

作者回复: 有可能hostname会改变，你的代码就获取不到最新的hostname

共 2 条评论 >

👍 3



一颗大白菜

2020-01-22

34行代码是不是写错了？

```
Assert.assertTrue(('0' < c && c > '9') || ('a' < c && c > 'z') || ('A' < c && c < 'Z'));
```

作者回复: 好像没有吧

共 5 条评论 >

👍 3



Ken张云忠

2020-01-22

读小争哥的注释就是种欣赏,小争哥的英文表达是怎么一步步积累的?

我认为动词和介词是英文的精髓,还有英文的语法

作者回复: 我英语也不好,多花点心思优化一下,实在不行,写中文注释也是可以的

共 3 条评论 >

👍 3



冬渐暖

2020-07-06

看了下您的代码,请教下 针对同一个service,有必要对各种情况都写一个@test吗?平时我都是一个接口一个test,如果有不同的条件,就直接在这个的入参上面改。不然某个测试类的代码会很大,也没有必要对一个接口一个类,而是一个综合业务一个test类。

作者回复: 一般来讲,一个单元测试类对应一个类。你说的可能更像是集成测试了。



👍 2



云宝

2020-11-17

generateRandomAlphameric()方法的测试用例需要改为: `Assert.assertTrue(('0' <= c && c <= '9') || ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z'));`

作者回复: 嗯嗯,我改下



👍



JUNLONG

2020-06-16

测试代码中的testGenerateRandomAlphameric()函数的for循环中的前两个范围判断打错了，应为：('0' < c && c < '9') || ('a' < c && c < 'z') 。
RandomIdGeneratorTest()函数中的一个#打成了\$

作者回复: 嗯嗯，多谢指出，我改下

