35讲实战一（下）：手把手带你将ID生成器代码从“能用”重构为“好用”

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。

```
public class IdGenerator {
  private static final Logger logger = LoggerFactory.getLogger(IdGenerator.class);

  public static String generate() {
    String id = "";
    try {
      String hostName = InetAddress.getLocalHost().getHostName();
      String[] tokens = hostName.split("\\.");
      if (tokens.length > 0) {
        hostName = tokens[tokens.length - 1];
      }
      char[] randomChars = new char[8];
      int count = 0;
      Random random = new Random();
      while (count < 8) {
        int randomAscii = random.nextInt(122);
        if (randomAscii >= 48 && randomAscii <= 57) {
          randomChars[count] = (char)('0' + (randomAscii - 48));
          count++;
        } else if (randomAscii >= 65 && randomAscii <= 90) {
          randomChars[count] = (char)('A' + (randomAscii - 65));
          count++;
        } else if (randomAscii >= 97 && randomAscii <= 122) {
          randomChars[count] = (char)('a' + (randomAscii - 97));
          count++;
        }
      }
      id = String.format("%s-%d-%s", hostName,
              System.currentTimeMillis(), new String(randomChars));
    } catch (UnknownHostException e) {
      logger.warn("Failed to get the host name.", e);
    }

    return id;
  }
}
```

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，那新的实现类就不好取名了，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，主要的目的是为了方便后续灵活地替换实现类。而LogTraceIdGenerator、UserIdGenerator、OrderIdGenerator三个雷从命名上来看，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。

```
IdGenearator idGenerator = new LogTraceIdGenerator();
替换为:
IdGenearator 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IdGenerator，命名为RandomIdGenerator、SequenceIdGenerator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：

```
public interface IdGenerator {
  String generate();
}

public interface LogTraceIdGenerator extends IdGenerator {
}

public class RandomIdGenerator implements IdGenerator {
  private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.class);

  @Override
  public String generate() {
    String substrOfHostName = getLastfieldOfHostName();
    long currentTimeMillis = System.currentTimeMillis();
    String randomString = generateRandomAlphameric(8);
    String id = String.format("%s-%d-%s",
            substrOfHostName, currentTimeMillis, randomString);
    return id;
  }

  private String getLastfieldOfHostName() {
    String substrOfHostName = null;
    try {
      String hostName = InetAddress.getLocalHost().getHostName();
      String[] tokens = hostName.split("\\.");
      substrOfHostName = tokens[tokens.length - 1];
      return substrOfHostName;
    } catch (UnknownHostException e) {
      logger.warn("Failed to get the host name.", e);
    }
    return substrOfHostName;
  }
```

```
  private String generateRandomAlphameric(int length) {
    char[] randomChars = new char[length];
    int count = 0;
    Random random = new Random();
    while (count < length) {
      int maxAscii = 'z';
      int randomAscii = random.nextInt(maxAscii);
      boolean isDigit= randomAscii >= '0' && randomAscii <= '9';
      boolean isUppercase= randomAscii >= 'A' && randomAscii <= 'Z';
      boolean isLowercase= randomAscii >= 'a' && randomAscii <= 'z';
      if (isDigit|| isUppercase || isLowercase) {
        randomChars[count] = (char) (randomAscii);
        ++count;
      }
    }
    return new String(randomChars);
  }
}

//代码使用举例
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。

```
public class RandomIdGenerator implements IdGenerator {
  private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.class);
```

```
@Override
public String generate() {
  String substrOfHostName = getLastfieldOfHostName();
  long currentTimeMillis = System.currentTimeMillis();
  String randomString = generateRandomAlphameric(8);
  String id = String.format("%s-%d-%s",
          substrOfHostName, currentTimeMillis, randomString);
  return id;
}

private String getLastfieldOfHostName() {
  String substrOfHostName = null;
  try {
    String hostName = InetAddress.getLocalHost().getHostName();
    substrOfHostName = getLastSubstrSplittedByDot(hostName);
  } catch (UnknownHostException e) {
    logger.warn("Failed to get the host name.", e);
  }
  return substrOfHostName;
}

@VisibleForTesting
protected String getLastSubstrSplittedByDot(String hostName) {
  String[] tokens = hostName.split("\\.");
  String substrOfHostName = tokens[tokens.length - 1];
  return substrOfHostName;
}

@VisibleForTesting
protected String generateRandomAlphameric(int length) {
  char[] randomChars = new char[length];
  int count = 0;
  Random random = new Random();
  while (count < length) {
    int maxAscii = 'z';
    int randomAscii = random.nextInt(maxAscii);
    boolean isDigit= randomAscii >= '0' && randomAscii <= '9';
    boolean isUppercase= randomAscii >= 'A' && randomAscii <= 'Z';
    boolean isLowercase= randomAscii >= 'a' && randomAscii <= 'z';
    if (isDigit|| isUppercase || isLowercase) {
      randomChars[count] = (char) (randomAscii);
      ++count;
    }
```

```
    }
    return new String(randomChars);
  }
}
```

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。

```
public String generate();
private String getLastfieldOfHostName();
@VisibleForTesting
protected String getLastSubstrSplittedByDot(String hostName);
@VisibleForTesting
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）：

```
public class RandomIdGeneratorTest {
  @Test
  public void testGetLastSubstrSplittedByDot() {
    RandomIdGenerator idGenerator = new RandomIdGenerator();
    String actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1.field2.field3");
    Assert.assertEquals("field3", actualSubstr);

    actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1");
    Assert.assertEquals("field1", actualSubstr);

    actualSubstr = idGenerator.getLastSubstrSplittedByDot("field1#field2$field3");
    Assert.assertEquals("field1#field2#field3", actualSubstr);
  }
```

```
  }

  // 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
  // 这部分优化留在第36、37节课中讲解
  @Test
  public void testGetLastSubstrSplittedByDot_nullOrEmpty() {
    RandomIdGenerator idGenerator = new RandomIdGenerator();
    String actualSubstr = idGenerator.getLastSubstrSplittedByDot(null);
    Assert.assertNull(actualSubstr);

    actualSubstr = idGenerator.getLastSubstrSplittedByDot("");
    Assert.assertEquals("", actualSubstr);
  }

  @Test
  public void testGenerateRandomAlphameric() {
    RandomIdGenerator idGenerator = new RandomIdGenerator();
    String actualRandomString = idGenerator.generateRandomAlphameric(6);
    Assert.assertNotNull(actualRandomString);
    Assert.assertEquals(6, actualRandomString.length());
    for (char c : actualRandomString.toCharArray()) {
      Assert.assertTrue(('0' < c && c > '9') || ('a' < c && c > 'z') || ('A' < c && c < 'Z'));
    }
  }

  // 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
  // 这部分优化留在第36、37节课中讲解
  @Test
  public void testGenerateRandomAlphameric_lengthEqualsOrLessThanZero() {
    RandomIdGenerator idGenerator = new RandomIdGenerator();
    String actualRandomString = idGenerator.generateRandomAlphameric(0);
    Assert.assertEquals("", actualRandomString);

    actualRandomString = idGenerator.generateRandomAlphameric(-1);
    Assert.assertNull(actualRandomString);
  }
}
```

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。

```
/**
 * Id Generator that is used to generate random IDs.
 *
 * <p>
 * The IDs generated by this class are not absolutely unique,
 * but the probability of duplication is very low.
 */
public class RandomIdGenerator implements IdGenerator {
  private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.class);

  /**
   * Generate the random ID. The IDs may be duplicated only in extreme situation.
   *
   * @return an random ID
   */
```

```
  @Override
  public String generate() {
    //...
  }

  /**
   * Get the local hostname and
   * extract the last field of the name string splitted by delimiter '.'.
   *
   * @return the last field of hostname. Returns null if hostname is not obtained.
   */
  private String getLastfieldOfHostName() {
    //...
  }

  /**
   * Get the last field of {@hostName} splitted by delemiter '.'.
   *
   * @param hostName should not be null
   * @return the last field of {@hostName}. Returns empty string if {@hostName} is empty string.
   */
  @VisibleForTesting
  protected String getLastSubstrSplittedByDot(String hostName) {
    //...
  }

  /**
   * Generate random string which
   * only contains digits, uppercase letters and lowercase letters.
   *
   * @param length should not be less than 0
   * @return the random string. Returns empty string if {@length} is 0
   */
  @VisibleForTesting
  protected String generateRandomAlphameric(int length) {
    //...
  }
}
```

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. 即便是非常简单的需求，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内部被吐掉了，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。

精选留言

chanllenge

public class RandomIdGenerator implements LogTraceIdGenerator，应该是这么写吧？

2020-01-22 15:21

作者回复

代码有点问题，我更新一下，抱歉

2020-01-27 17:21

Yang

1.应该需要继续抛出，因为在实际的业务开发中，会有对应的异常处理器，抛出可以让调用者明白哪出错了，而不是只是简单的打印日志。

2.命名getLastSubstrSplittedByDot替换成getLastSubstrByDelimiter，具体要看需求会不会经常变化，如果经常变化，替换没有任何问题，因为有可能后面根据别的符号来分割，这种情况下我个人认为getLastFiledOfHostName()函数命名应该替换成getLastFiled()，命名不应该暴露太多细节，要是以后不是根据HostName获取最后一个字段呢，之前的所有用到该命名的地方都需要替换，不然可读性不是很好。

如果需求不经常变化，那文中的命名就足够了。

2020-01-22 08:06

辣么大

这两期争哥讲重构，我把Uncle Bob的《重构2》的第一章看了，大呼过瘾。自己也要操刀试一下！

他和Kent Beck强调重构时要用baby step（小步骤），什么是baby step呢？就是一次改一小点，例如改一个变量名字都需要进行 modify-build-test的步骤。

对于争哥的例子，我参考Uncle Bob书中的方法：

第一步、先写好测试

第二步、开始逐步重构（baby step）
第三步、修改-> 测试

经过重构之后代码总计50行。重构之后代码易读，且结构清晰。
https://github.com/gdhucoder/Algorithms4/blob/master/designpattern/u35/RandomLogTraceIDGenerator.java
2020-01-22 22:07


小晏子
在获取主机名失败的时候，generate函数应该能正常返回，因为是随机id，所以只要有个满足要求的id就行了，用户并不关心能不能拿到主机名字，所以在获取主机名失败的时候，可以返回一个默认的主机名，之后在拼接上时间戳和随机数也是满足需求的id，所以我认为generate函数在主机名失败的时候应该使用默认主机名正常返回。另外对于小王的异常处理我认为是可以捕获处理的，只是不能该让整个函数都返回一个空id，而是应该捕获异常时使用一个默认主机名继续后面的逻辑。
第二个问题：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？
我认为是合理的，命名和代码的逻辑不绑定，避免了以后修改代码逻辑还要修改函数名的麻烦，比如将来可能不用点去分割hostname了，用空格分割，这时byDot函数名就不合适了，如果修改，那么所有使用到这个函数的地方都要改，大大增加了出错的概率。
2020-01-22 09:07


Wings
争哥，我是看了你的算法之美后立刻看到你出设计模式之美就立刻买。可是专栏更新到现在快一半，老实说，我觉得内容真的很基础甚至脱离实际开发，很多都是浅尝辄止。专栏一开始渲染了好多说会有很多可落地的代码，可目前为止看到的都是很虚无聊会或者是大家早就知道的东西。如果可以的话，能否在后续课程多分享一些真正的企业级的代码设计和重构呢?
2020-01-23 15:51

作者回复
抱歉没有呢，让你失望了，不过，我还会出新课的，以后你就别买我的课程了。因为新课估计也会让你失望的~

说实话，我觉得的我写的很好，而且很结合实际开发，很多人留言说我写的好，当然也有人根本不识货！如果你觉得哪一个不能落地，能具体指出来吗？或者你觉得哪篇写的不好，网上或者哪本书籍讲的比我讲的好，你指出来。不然你随口一说，无凭无据，那不就瞎喷吗？说实话啥都不懂瞎喷的人太多了，我也不可能一个一个的喷回去，没意思，如果你觉得有写的不好地方，你大可就事论事列举出来，我倒是会很认真的思考改进，不然，我就只能当你是喷子了啊 哥们 。而且，你能说下什么是真正企业级的吗？我工作10多年，搞不清楚什么才是真正企业级的呢。。。

你可以看下我写的这篇文章：公众号”小争哥“ 看看下面的留言：
https://mp.weixin.qq.com/s/Od95pFonyLo7IlB3THa8Tw

2020-01-27 16:09


辣么大
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

2020-01-22 22:15

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。
2020-01-27 16:34

Monday

异常不是被“吐掉”了吧，是被 “吞掉”了吧

2020-02-11 22:39

Ken张云忠

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

2020-01-22 13:28

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

2020-01-27 16:36

evolution

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?

2020-01-22 10:18

作者回复

感谢认可，暂时没有呢

2020-01-27 16:51

牛顿的烈焰激光剑

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?

2020-01-25 00:19

作者回复

有可能hostname会改变，你的代码就获取不到最新的hostname

2020-01-27 16:02

全时N多只

34行代码是不是写错了?

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

2020-01-22 15:54

作者回复

好像没有吧

2020-01-27 16:54

Ken张云忠

小争个,我有个问题:

接口LogTraceIdGenerator 继承 IdGenerator,RandomIdGenerator实现IdGenerator,但是在使用时却是LogTraceIdGenerator logTraceIdGenerator = new RandomIdGenerator();

这里是需要类型强转的,另外这里强转在运行时是会失败的,因为RandomIdGenerator与LogTraceIdGenerator在类型上没有继承和实现关系.

这里该怎么理解?

2020-01-22 13:23

作者回复

代码有点问题，我更新一下，抱歉~~

2020-01-27 17:22

路阳

LogTraceIdGenerator logTraceIdGenerator = new RandomIdGenerator()； 这个代码的问题，老师还是抽空改一下吧

2020-02-21 14:32

aoe

小争哥一波英文注释，猝不及防啊

2020-02-18 16:56

楊_宵夜

争哥, RandomIdGeneratorTest 的代码, 即使是 new RandomIdGenerator() , 也是不能够访问 protected 方法的呀, 实测过用 JUnit框架 也不能...

你这个 Test 是不是放在了 RandomIdGenerator 同一个包底下了咧?

2020-02-16 13:54

马哲富

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！

2020-02-15 13:20

作者回复

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。

2020-02-15 13:29

Dimple

年后回来复工了，在家学习动力不足，赶紧在公司也抠出时间来学习。实战篇的思想是我要学习的地方，对我来说，往后针对项目需求的规划会更好。

2020-02-11 16:34

慕容引刀

个人觉得小争哥的文章还是很给力的，前边的铺垫很多是在讲解代码中可能出现的问题以及如何发现问题。一般来说在清楚代码中存在哪些的问题的情况下就会去寻找解决方案，很容易发现一些设计模式就是为了解决某些问题而存在的。同时因为清楚问题所在，就很容易了解掌握这些设计模式。

2020-02-09 16:07

苟麒睿

对于问题一，我觉得是否往上继续抛出得看情况，如果异常的时候比方说null的时候，也要有一个固定返回值例如直接返回字符串"null"，那就不用往上抛而是在函数里面直接返回，我觉得更加好。对于问题二，我觉得如果函数内的逻辑会经常出现变化了化，可以替换成getLastSubstrByDelimiter，不会经常变化了化应该不用改。

过年各种事情在学习上有点懈怠了，之前的文章也没来得及多多复习，落下了满多，得抓紧补补了，同时也不能忘了前面的复习，温故而知新嘛

2020-02-09 12:21

DullBird

1. hostname不能影响业务逻辑，设置默认值，并且还有随机值可以区分。

2. 通用的比较合适，但是看有没有复用这个代码的必要，否则暂时不需要改动

2020-02-09 11:12