

## 21 | 怎么设计一个简单又直观的接口？

2019-02-20 范学雷

代码精进之路

[进入课程 >](#)



讲述：刘飞

时长 11:45 大小 26.90M



我们前面聊过接口规范，开放的接口规范是使用者和实现者之间的合约。既然是合约，就要成文、清楚、稳定。合约是好东西，它可以让代码之间的组合有规可依。但同时它也是坏东西，让接口的变更变得困难重重。

接口设计的困境，大多数来自于接口的稳定性要求。摆脱困境的有效办法不是太多，其中最有效的一个方法就是要**保持接口的简单直观**。那么该怎么设计一个简单直观的接口呢？

### 从问题开始

软件接口的设计，要从真实的问题开始。

一个解决方案，是从需要解决的现实问题开始的。要解决的问题，可以是用户需求，也可以是现实用例。面对要解决的问题，我们要把大问题分解成小问题，把小问题分解成更小的问题，直到呈现在我们眼前的是公认的事实或者是可以轻易验证的问题。

比如说，是否可以授权一个用户使用某一个在线服务呢？这个问题就可以分解为两个小问题：

1. 该用户是否为已注册的用户？
2. 该用户是否持有正确的密码？

我们可以使用思维导图来描述这个分解。



分解问题时，我们要注意分解的问题一定要“相互独立，完全穷尽”（Mutually Exclusive and Collectively Exhaustive）。这就是 MECE 原则。使用 MECE 原则，可以帮助我们用最合理的条理化和最大的完善度理清思路。

如何理解这个原则呢？

先来说一下“相互独立”这个要求。问题分解后，我们要仔细琢磨，是不是每一个小问题都是独立的，都是可以区分的事情。

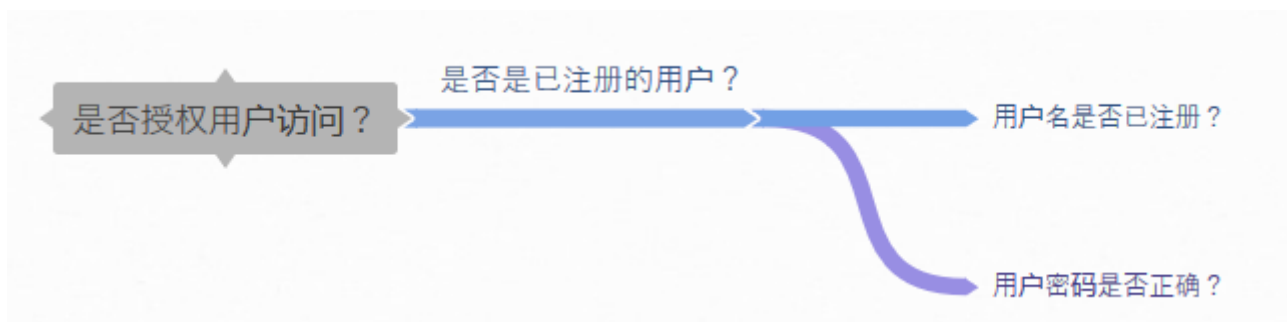
我们以上面的分解为例子，仔细看会发现这种划分是有问题的。因为只有已经注册的用户，才会持有正确的密码。而且，只有持有正确密码的用户，才能够被看作是注册用户。这两个小问题之间，存在着依赖关系，就不能算是“相互独立”。

我们要消除掉这种依赖关系。

变更后，就需要两个层次的表达。第一个层次问题是，该用户是否为已注册的用户？这个问题，可以进一步分解为两个更小的问题：用户持有的用户名是否已注册？用户持有的密码是否匹配？

1. 该用户是否是已注册的用户？
  - a. 用户名是否已注册？
  - b. 用户密码是否正确？

这种描述的思维导图，和上面的相比，已经有了很大的差别。



除了每一项都要独立之外，我们还要琢磨，是不是把所有能够找到的因素，都找到了？也就是说，我们是否穷尽了所有的内容，做到了“完全穷尽”？

你可能早已经注意到了上述问题分解的缺陷。如果一个服务，对所有的注册用户开放，上面的分解就是完备的。否则，我们就漏掉了一个重要的内容，不同的注册用户，可以访问的服务可能是不同的。也就是说如果没有访问的权限，那么即使用户名和密码正确也无法访问相关的服务。

如果我们将漏掉的加上，这个问题的分解可以进一步表示为：

1. 该用户是否是已注册的用户？
  - a. 用户名是否已注册？
  - b. 用户密码是否正确？

2. 该用户是否有访问的权限？



完成上述的分解后，对于是否授权用户访问一个服务这个问题，我们就会有一个清晰的思路了。

## 为什么从问题开始？

为什么我们要遵循“相互独立，完全穷尽”的原则呢？

只有完全穷尽，才能把问题解决掉。否则，这个解决方案就是有漏洞的，甚至是无效的。

只有相互独立，才能让解决方案简单。否则，不同的因素纠缠在一起，既容易导致思维混乱，也容易导致不必要的复杂。

还有一个问题，我们也要清楚地理解。那就是，为什么要从问题开始呢？

从问题开始，是为了让我们能够找到一条主线。然后，围绕这条主线，去寻找解决问题的办法，而不是没有目标地让思维发散。这样，也可以**避免需求膨胀和过度设计**。

比如说，如果没有一条主线牵制着，按照面向对象编程的思路，我们看到“用户”两个字，马上就会有无限的联想。是男的还是女的呀？姓啥名谁呀？多大岁数了？家住哪儿啊？一系列问题都会冒出来，然后演化成一个庞大的对象。但事实上，对于上面的授权访问问题，我们根本不需要知道这些。


## 自然而来的接口

把大问题分解成小问题，再把小问题分解成更小的问题。在这个问题逐层分解的过程中，软件的接口以及接口之间的联系，也就自然而然地产生了。这样出来的接口，逻辑直观，职责清晰。对应的，接口的规范也更容易做到简单、稳定。

还记得我们前面说过的 Java 的命名规范吗？Java 类的标识符使用名词或者名词短语，接口的标识符使用名词、名词短语或者形容词，方法的标识符使用动词或者动词短语。这背后的逻辑是，Java 类和接口，通常代表的是一个对象；而 Java 的方法，通常代表的是一个动作。

我们在分解问题的过程中，涉及到的关键的动词和动词短语、名词和名词短语或者形容词，就是代码中类和方法的现实来源。比如，从上面的问题分解中，我们很容易找到一个基础的小问题：用户名是否已注册。这个小问题，就可以转换成一个方法接口。

我们前面讨论过这个接口。下面，我们再来看看这段使用过的代码，你有没有发现什么不妥的地方？

 复制代码


```
1 /**
2  * Check if the {@code userName} is a registered name.
3  *
4  * @return true if the {@code userName} is a registered name.
5  */
6 boolean isRegisteredUser(String userName) {
7     // snipped
8 }
```

不知道你看到没有，这个方法的命名是不妥当的。

根据前面的问题分解，我们知道，判断一个用户是不是注册用户，需要两个条件：用户名是否注册？密码是否正确？

上面例子中，这个方法的参数，只有一个用户名。这样的话，只能判断用户名是不是已经被注册，还判断不了使用这个用户名的用户是不是真正的注册用户。

如果我们把方法的名字改一下，就会更符合这个方法的职能。

 复制代码

```
1 /**
2  * Check if the {@code userName} is a registered name.
3  *
4  * @return true if the {@code userName} is a registered name.
5  */
6 boolean isRegisteredUserName(String userName) {
7     // snipped
8 }
```

如果你已经理解了我们前面的问题分解，你就会觉得原来的名字有点儿刺眼或者混乱。这就是问题分解带给我们的好处。问题的层层简化，会让接口的逻辑更直观，职责更清晰。这种好处，也会传承给后续的接口设计。

## 一个接口一件事

前面，我们提到过一行代码只做一件事情，一块代码只做一件事情。一个接口也应该只做一件事情。

如果一行代码一件事，那么一块代码有七八行，不是也应该做七八件事情吗？怎么能说是一件事情呢？这里我们说的“事情”，其实是在某一个层级上的一个职责。授权用户访问是一件完整、独立的事情；判断一个用户是否已注册也是一件完整、独立的事情。只是这两件事情处于不同的逻辑级别。也就是说，一件事情，也可以分几步完成，每一步也可以是更小的事情。有了逻辑级别，我们才能分解问题，接口之间才能建立联系。

对于一件事的划分，我们要注意三点。

1. 一件事就是一件事，不是两件事，也不是三件事。
2. 这件事是独立的。
3. 这件事是完整的。

如果做不到这三点，接口的使用就会有麻烦。

比如下面的这段代码，用于表示在不同的语言环境下，该怎么打招呼。在汉语环境下，我们说“你好”，在英语环境下，我们说“Hello”。

 复制代码

```
1  /**
2   * A {@code HelloWords} object is responsible for determining how to say
3   * "Hello" in different language.
4   */
5  class HelloWords {
6      private String language = "English";
7      private String greeting = "Hello";
8
9      // snipped
10
11     /**
12      * Set the language of the greeting.
13      *
```

```
14     * @param language the language of the greeting.
15     */
16     void setLanguage(String language) {
17         // snipped
18     }
19
20     /**
21     * Set the greetings of the greeting.
22     *
23     * @param language the greetings of the greeting.
24     */
25     void setGreeting(String greeting) {
26         // snipped
27     }
28
29     // snipped
30 }
```

这里涉及两个要素，一个是语言（英语、汉语等），一个是问候语（Hello、你好等）。上面的这段代码，抽象出了这两个要素。这是好的方面。

看起来，有两个独立的要素，就可以有两个独立的方法来设置这两个要素。使用 `setLanguage()` 设置问候的语言，使用 `setGreeting()` 设置问候的问候语。看起来没什么毛病。

但这样的设计对用户是不友好的。因为 `setLanguage()` 和 `setGreeting()` 这两个方法，都不能表达一个完整的事情。只有两个方法合起来，才能表达一件完整的事情。

这种互相依赖的关系，会导致很多问题。比如说：

1. 使用时，应该先调用哪一个方法？
2. 如果语言和问候语不匹配，会出现什么情况？
3. 实现时，需不需要匹配语言和问候语？
4. 实现时，该怎么匹配语言和问候语？

这些问题，使用上面示例中的接口设计，都不好解决。一旦接口公开，软件发布，就更难解决掉了。

## 减少依赖关系




有时候，“一个接口一事情”的要求有点理想化。如果我们的设计不能做到这一点，一定要减少依赖关系，并且声明依赖关系。

一般来说一个对象，总是先要实例化，然后才能调用它的实例方法。构造方法和实例方法之间，就有依赖关系。这种依赖关系，是规范化的依赖关系，有严格的调用顺序限制。编译器可以帮我们检查这种调用顺序。

但是，我们自己设计的实例方法之间的依赖关系，就没有这么幸运了。这就要求我们弄清楚依赖关系，标明清楚依赖关系、调用顺序，以及异常行为。

下面的这段代码，摘录自 OpenJDK。这是一个有着二十多年历史的，被广泛使用的 Java 核心类。这段代码里的三个方法，有严格的调用顺序要求。要先使用 `initSign()` 方法，再使用 `update()` 方法，最后使用 `sign()` 方法。这些要求，是通过声明的规范，包括抛出异常的描述，交代清楚的。

 复制代码

```
1  /*
2   * Copyright (c) 1996, 2018, Oracle and/or its affiliates. All rights reserved.
3   * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
4   *
5   * <snipped>
6   */
7
8  package java.security;
9
10 import java.security.InvalidKeyException;
11 import java.security.PrivateKey;
12 import java.security.SignatureException;
13 import java.security.SignatureSpi;
14
15 /**
16  * The Signature class is used to provide applications the functionality
17  * of a digital signature algorithm. Digital signatures are used for
18  * authentication and integrity assurance of digital data.
19  *
20  * <snipped>
21  *
22  * @since 1.1
23  */
24 public abstract class Signature extends SignatureSpi {
25     // snipped
26
27     /**
28      * Initialize this object for signing. If this method is called
```



```

29     * again with a different argument, it negates the effect
30     * of this call.
31     *
32     * @param privateKey the private key of the identity whose signature
33     * is going to be generated.
34     *
35     * @exception InvalidKeyException if the key is invalid.
36     */
37     public final void initSign(PrivateKey privateKey)
38         throws InvalidKeyException {
39         // snipped
40     }
41
42     /**
43     * Updates the data to be signed or verified, using the specified
44     * array of bytes.
45     *
46     * @param data the byte array to use for the update.
47     *
48     * @exception SignatureException if this signature object is not
49     * initialized properly.
50     */
51     public final void update(byte[] data) throws SignatureException {
52         // snipped
53     }
54
55     /**
56     * Returns the signature bytes of all the data updated.
57     * The format of the signature depends on the underlying
58     * signature scheme.
59     *
60     * <p>A call to this method resets this signature object to the state
61     * it was in when previously initialized for signing via a
62     * call to {@code initSign(PrivateKey)}. That is, the object is
63     * reset and available to generate another signature from the same
64     * signer, if desired, via new calls to {@code update} and
65     * {@code sign}.
66     *
67     * @return the signature bytes of the signing operation's result.
68     *
69     * @exception SignatureException if this signature object is not
70     * initialized properly or if this signature algorithm is unable to
71     * process the input data provided.
72     */
73     public final byte[] sign() throws SignatureException {
74         // snipped
75     }
76
77     // snipped
78 }

```

然而，即使接口规范里交待清楚了严格的调用顺序要求，这种设计也很难说是一个优秀的设计。用户如果不仔细阅读规范，或者是这方面的专家，很难第一眼就对调用顺序有一个直观、准确的认识。

这就引出了另一个要求，接口一定要“皮实”。

## 使用方式要“傻”

所有接口的设计，都是为了最终的使用。方便、皮实的接口，才是好用的接口。接口要很容易理解，能轻易上手，这就是方便。此外还要限制少，怎么用都不容易出错，这就是皮实。

上面的 OpenJDK 例子中，如果三个方法的调用顺序除了差错，接口就不能正常地使用，程序就不能正常地运转。既不方便，也不皮实。

## 小结

今天，我们主要讨论了该怎么设计简单直观的接口这个话题。这是一个很大的话题。我们只讨论了最基本的原则，那就是：

1. 从真实问题开始，把大问题逐层分解为“相互独立，完全穷尽”的小问题；
2. 问题的分解过程，对应的就是软件的接口以及接口之间的联系；
3. 一个接口，应该只做一件事情。如果做不到，接口间的依赖关系要描述清楚。


另外，关于面向对象设计，有一个简称为 SOLID 的面向对象设计五原则。如果你没有了解过这些原则，我也建议你找来看看。也欢迎你在留言区分享你对这些原则的理解和看法。

## 一起来动手

下面的这段代码，摘录自 OpenJDK，是上面那个例子的扩充版。如果从面向对象的角度来看，这样的设计也许是无可厚非的。但是这种设计存在着很多的缺陷，也带来了越来越多的麻烦。这是一个现实存在的问题，直到 OpenJDK 12，这些缺陷还没有改进。

你试着找一找，看看能发现哪些缺陷，有没有改进的办法。欢迎你把发现的缺陷，以及优化的接口公布在讨论区，也可以写一下你的优化思路。说不定，你可以为 OpenJDK 社区，提供一个有价值的参考意见或者改进方案。

也欢迎点击“请朋友读”，和你的朋友一起交流一下这段代码。

 复制代码

```
1  /*
2   * Copyright (c) 1996, 2018, Oracle and/or its affiliates. All rights reserved.
3   * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
4   *
5   * <snipped>
6   */
7
8  package java.security;
9
10 import java.security.InvalidAlgorithmParameterException;
11 import java.security.InvalidKeyException;
12 import java.security.PrivateKey;
13 import java.security.PublicKey;
14 import java.security.SignatureException;
15 import java.security.SignatureSpi;
16 import java.security.spec.AlgorithmParameterSpec;
17
18 /**
19  * The Signature class is used to provide applications the functionality
20  * of a digital signature algorithm. Digital signatures are used for
21  * authentication and integrity assurance of digital data.
22  *
23  * <snipped>
24  *
25  * @since 1.1
26  */
27 public abstract class Signature extends SignatureSpi {
28     // snipped
29
30     /**
31      * Initializes this signature engine with the specified parameter set.
32      *
33      * @param params the parameters
34      *
35      * @exception InvalidAlgorithmParameterException if the given parameters
36      * are inappropriate for this signature engine
37      *
38      * @see #getParameters
39      */
40     public final void setParameter(AlgorithmParameterSpec params)
41         throws InvalidAlgorithmParameterException {
42         // snipped
43     }
44
45     /**
46      * Initializes this object for verification. If this method is called
47      * again with a different argument, it negates the effect
```

```

48     * of this call.
49     *
50     * @param publicKey the public key of the identity whose signature is
51     * going to be verified.
52     *
53     * @exception InvalidKeyException if the key is invalid.
54     */
55     public final void initVerify(PublicKey publicKey)
56         throws InvalidKeyException {
57         // snipped
58     }
59
60     /**
61     * Initialize this object for signing. If this method is called
62     * again with a different argument, it negates the effect
63     * of this call.
64     *
65     * @param privateKey the private key of the identity whose signature
66     * is going to be generated.
67     *
68     * @exception InvalidKeyException if the key is invalid.
69     */
70     public final void initSign(PrivateKey privateKey)
71         throws InvalidKeyException {
72         // snipped
73     }
74
75     /**
76     * Updates the data to be signed or verified, using the specified
77     * array of bytes.
78     *
79     * @param data the byte array to use for the update.
80     *
81     * @exception SignatureException if this signature object is not
82     * initialized properly.
83     */
84     public final void update(byte[] data) throws SignatureException {
85         // snipped
86     }
87
88     /**
89     * Returns the signature bytes of all the data updated.
90     * The format of the signature depends on the underlying
91     * signature scheme.
92     *
93     * <p>A call to this method resets this signature object to the state
94     * it was in when previously initialized for signing via a
95     * call to {@code initSign(PrivateKey)}. That is, the object is
96     * reset and available to generate another signature from the same
97     * signer, if desired, via new calls to {@code update} and
98     * {@code sign}.
99     *

```

```
100     * @return the signature bytes of the signing operation's result.
101     *
102     * @exception SignatureException if this signature object is not
103     * initialized properly or if this signature algorithm is unable to
104     * process the input data provided.
105     */
106     public final byte[] sign() throws SignatureException {
107         // snipped
108     }
109
110     /**
111     * Verifies the passed-in signature.
112     *
113     * <p>A call to this method resets this signature object to the state
114     * it was in when previously initialized for verification via a
115     * call to {@code initVerify(PublicKey)}. That is, the object is
116     * reset and available to verify another signature from the identity
117     * whose public key was specified in the call to {@code initVerify}.
118     *
119     * @param signature the signature bytes to be verified.
120     *
121     * @return true if the signature was verified, false if not.
122     *
123     * @exception SignatureException if this signature object is not
124     * initialized properly, the passed-in signature is improperly
125     * encoded or of the wrong type, if this signature algorithm is unable to
126     * process the input data provided, etc.
127     */
128     public final boolean verify(byte[] signature) throws SignatureException {
129         // snipped
130     }
131 }
```



# 代码精进之路

你写的每一行代码都是你的名片

范学雷

Oracle 首席软件工程师  
Java SE 安全组成员  
OpenJDK 评审成员



新版升级：点击「 请朋友读」，10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 20 | 简单和直观，是永恒的解决方案

下一篇 22 | 高效率，从超越线程同步开始！

## 精选留言 (7)

写留言



hua168

2019-02-20

3

1.说到接口，现在网站接口风格还是 RESTful API，GraphQL用得少吧？

2.我看了之前的文章想起了一个实现的问题：

像我们中小公司人员流失比较大，从0开发一个电商网站的话，很多开发为了赶时间都不愿意写详细的开发文档，代码只是简单的做一下注解，交接文档也写得随便，这种情况如果原始团队的开发换完了， ...

展开

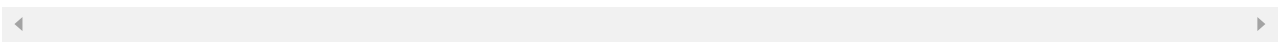
作者回复：#1. 我知道的，RESTful API用的多些。我对这方面的不熟，希望留言区有人可以帮着回答。

#2. 其实，短期内我们很难做到“铁打的营盘，流水的兵”。软件开发还算是一项复杂的活动，很多时候，也会体现出是“流水的营盘，铁打的将”的现象。优秀的程序员，还是要想办法留住

的。有研究表明，替换一个工程师，需要花费平均6到9个月的薪水，甚至是1.5到2年的薪水。如果我们把替换成本变成现有工程师的薪水涨幅，也许事情就简单了很多。

代码没人敢动，一个很重要的原因，就是没有回归测试或者回归测试不完备，我们搞不清修改代码带来的后果。JDK运行在几十亿台设备上，每天都做很多修改。之所以能够做这么大量的修改，除了规范、文档、评审之外，还有大量的回归测试案例。代码修改提交之前，要把相关的回归测试跑一遍。一旦修改带来了兼容性问题，回归测试就会检测出来，工程师就会知道修改带来的影响，会重新考量修改方案。

另外，要用好现代的工具，不能停留在手种刀割的时代。很多工具都是开源软件，搭建起来，形成习惯就可好了。比如，bug管理工具（bug systems），版本控制工具（git, mercurial），这些工具都会留有历史信息，用好了可以更好地理解代码和变更。不使用这些工具，好多有价值的东西，极小一部分留在工程师的脑子里，人走了，价值也就随着走了；大部分都会被岁月冲散。JDK的开发过程中，我经常需要找找十多年前历史信息，看看当初为什么那样设计，要解决的到底是什么问题，变更起来影响可以有多坏。



克里斯

2019-05-03

👍 1

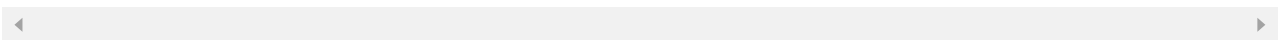
这种强依赖问题属于归纳思维中的时间顺序范畴。

但是时间顺序的过程在代码里表现为 结构顺序，丢失了时间顺序的信息。

为了解决这个问题:...

展开 ▾

作者回复: 这两种都是很好的方法。第一个方案的一个小缺点是如果用户传入参数，内部整理在个别的情况下没有办法自动处理，不过大部分情况下都没有问题。第二种理论上没什么问题，就是对规范和用户的要求都有一点点高，需要规范标注；仔细阅读规范；并且遵守时间顺序。😞，编码的难处就在于要反复地妥协和平衡。



克里斯

2019-05-03

👍 1

作者应该是看了《金字塔原理》这本书。作者基本上到目前的章节，基本都在表述程序员版的《金字塔原理》😄



作者回复: 搜了一下, 是《金字塔原理:麦肯锡40年经典培训教材》吗? 你的阅读面很广啊! 嗯, 有机会我也要买来看看。作为一名老旧式的五道口技校经管的学生, 推荐大家多读读经济管理的书籍, 对产品设计很有帮助的, 大部分的设计思想都逃脱不了经济管理的原理范畴。



**Sisyphus2...**

2019-05-23



Signature 继承 SignatureSpi, 实现签名算法。

setParameter 方法通过传入参数启动 Signature engine, initVerify 初始化 verification 对象, initSign 初始化 signing 对象。

update 更新 signed 或者 verified data, sign 返回二进制签名, verify 验证传入的签名。...

展开 ∨

作者回复: 嗯, 估计你使用小屏幕阅读的。代码的显示在小屏幕上, 的确是个问题。

这个Signature类设计的最大问题, 就来源于update()这个方法。因为签名数据可能很大很大, 这种情况下, 需要分批传入数据, 使用update()方法。当然, 还有更好的解决办法。



**LeasonZ**

2019-03-20



问题: 如何划分一件事?

就像吃饭,简单说吃饭就是拿东西吃,更细一点变成了拿食物->张嘴->咬->咀嚼->吞咽,再细点就变成肌肉运动,电信号之类的.所以划分维度在代码设计时该怎么去把控,这是长久困扰我的一个疑问,希望老师能解答下

作者回复: MESE的原则是划分到事实为止。用到代码上, 划分到有现成方法可以用为止。如果有吞咽这个方法, 划到吞咽就够了。



**Tom**

2019-03-02



签名数据太大, 比如文件图片, 占用内存大, 使用流处理可以减少内存占用吗?



唐名之

2019-02-20



我们习惯这样写，：

```
private final void initVerify(PublicKey publicKey)
    throws InvalidKeyException {
    // snipped
}...
```

展开 ▾

作者回复: 这样适合处理小数据，如果签名数据很大，比如文件，图像，sign方法需要占用的内存太多，占用时间太长（data参数）。verify方法怎么传要签名的数据呢？

