自表达代码

书写可读代码之道

附带大量例子代码

```
if (dir == FORWARD) {
    through();
}
if (dir == TURN_LEFT) {
    turn(RIGHT);
    turn(RIGHT);
    turn(RIGHT);
}
```

王　洪　亮

自表达代码

书写可读代码之道

王洪亮

This book is for sale at http://leanpub.com/self_expressive_code_sc

This version was published on 2014-05-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

前言 **1**

致谢 **2**

什么是自表达代码 **3**

第 **I** 部分可读性 **5**

第一章低可读性之殇 5

1.1 恼人的 Bug 6

1.2 难以理解 7

1.3 难以扩展或变更 9

1.4 容易做入 Bug 12

1.5 难以重构 13

第二章追本溯源 15

延伸阅读 **17**

前言

我们编写代码的时候会遇到可读性的问题。会为了一个命名，一个结构而苦苦思索，生怕别人来读的时候读不懂。我们也总是会遇到一些难以读懂的代码，试图理解这种代码的意思也是一种具有挑战性的工作。

什么样的代码才是低可读性的代码？是什么导致了代码可读性的降低？又怎么解决呢？这些问题似乎都很难。

受到编程语言的语法限制，程序必须按照一定的格式来书写。有些人会认为采用自然语言去书写代码似乎并不现实。的确，我们无法像说话一样去写代码，但是我们可以尽量的采用自然语言去描述我们的程序。而编程语言又给我们提供了很多的好用的符号帮助我们达成这个目标。

而对于非英语母语的程序员来说，单词量不够导致了命名时无法采用合适的名字。另外，由于语言习惯的问题，不会想到构建一些介词的类或者方法来增强可读性。有些时候，即使对于英语母语的程序员来说，也同样会遇到易读性问题。这些问题可能并非来自名称，而是来自结构、架构、算法。

总之，提升代码可读性并非易事。

本书将会全面的介绍一种提升代码可读性的方法——自表达代码 (Self-Expressive Code)。如其名称所暗示，自表达代码表示代码本身就能够表述其意思。程序员可以快速而简单的了解代码的意思。自表达代码追求一种能够让人一看就懂得代码书写方式。

Java 到目前为止还是非常流行的一种编程语言，本书选用了 Java 作为其例子语言。

致谢

下面是为本书出版提供了有益帮助的人员名单

- 陈金洲
- 刘成章
- 韩奎奎

愿本书不仅为非英语母语程序员，也为英语母语的程序员们带来有益的帮助。

本书参考了下列图书

- 设计模式：可复用面向对象软件的基础[1]，机械工业出版社，2000 年 9 月作者：[美] Erich Gamma 、Richard Helm 、Ralph Johnson、John Vlissides，译者：李英军、马晓星、蔡敏、刘建中 ISBN: 9787111075752
- 重构：改善既有代码的设计[2]，人民邮电出版社，2010 年 04 月，作　者：[美] 福勒译者: 熊节 ISBN: 9787115221704
- 代码整洁之道[3]，人民邮电出版社，2010 年 01 月，作　者：(美国) 马丁 ISBN: 9787115216878
- 编写可读代码的艺术[4]，机械工业出版社 2012 年 7 月作者：Boswell, D. 、Foucher, T. 译者：尹哲、郑秀雯，ISBN: 9787111385448

[1] http://baike.baidu.com/view/66964.htm#1
[2] http://baike.baidu.com/view/2547526.htm
[3] http://baike.baidu.com/view/2708403.htm#1
[4] http://baike.baidu.com/view/8805043.htm

什么是自表达代码

在汇编语言中，代码是面向机器写的。人们难以阅读代码，只能猜测其意思。几十年过去了，编程语言有了飞跃的发展，变得越来越易读，编程语言也从面向机器变成了面向程序员。

尽管如此，现实世界中，，即使采用了自然语言的方式进行了命名，人们仍旧要面对大量的用 Java 或 C# 书写的难以阅读的代码。有一个开放源代码的流行平台上，有一段代码充斥着大量的低可读性代码。读了这些代码，感觉当初的程序员是面向机器书写的，而不是面向读者。我们无法查证当初的真正原因, 但是我们仍然可以分析这些代码。

这些代码包含很多这样的代码：

1. 同一段 if-else 对不同的变量进行判断；
2. 非英语命名；
3. 长分支；
4. 长方法；
5. 大量的重复与类似；
6. 很多意思模糊的变量。

这些特点说明了当初的程序员没有充分的考虑读者的感受。这是一个第三方公司提供的源代码，他们在这个流行平台上开放源代码也许是为了品牌效应或者什么其他东西。然而，这样的代码只能让他们适得其反。读了他们的代码之后以至于让我担心他们公司出产的医疗器械产品的质量。但愿二者是使用不同的流程来管理的。

那么，怎么才能够写出高可读性的代码呢?

举例来说，在一个游戏程序中，玩家可以控制一个英雄 (hero) 在场景 (scene) 里去攻击敌人 (enemy)。相应的代码可能会这么写：

```
1  scene.fight(hero, enemy, Kongfu.FIST);
```

在上述代码中，读者无法了解到底是英雄攻击了敌人，还是敌人攻击了英雄。

那么，修订一下代码

```
1  hero.fight(enemy, Kongfu.FIST);
```

受到编程语言的语法限制，常量必须带有其类名。代码的意思已经清晰了，但是仍然有些面向机器的感觉。然后，一个更好的修订版出现了：

```
1  hero.fight(enemy).with("FIST");
```

这段代码中，fight() 方法返回一个包含 with() 方法的 Fight 对象，该 with() 方法传入一个功夫的名称，通过一个 KongfuFactory 创建一个 Kongfu 对象再来计算伤害。这段处理带来了更多的类，但是代码却更清晰易懂了。

实现功能只是代码最基本的职责，但是保持代码的可读性、可扩展性、可变更性同样很重要。正如本节前面所述，代码是写给人看的，所以我建议尽量使用自然语言来书写代码。我把这种书写代码的方式称为“自表达代码”，在自表达代码问世之前，已经有两个类似的名词“自文档代码”[5]和“自描述代码”[6]存在了。它们的意思可能很相近，但是我觉得“表达”一词更具有表现力。所以，我把这个命名为了“自表达代码”。

[5] http://en.wikipedia.org/wiki/Self-documenting

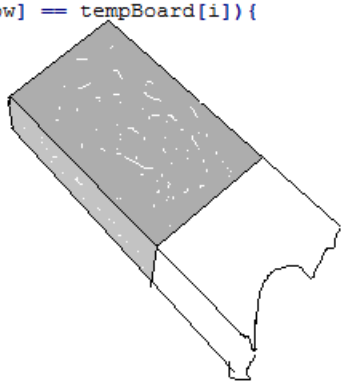
[6] http://en.wikipedia.org/wiki/Self-describing

第 I 部分可读性

```
private List<int[]> solutions = new ArrayList<int[]>();
private static final int ARRAY_SIZE = 8;
private int[] tempBoard = new int[ARRAY_SIZE];

private void putQueenAt(int row, int column) {
    tempBoard[row] = column;
}

private boolean hasConfliction(int row) {
    for (int i = 0; i < row; i ++) {
        if (Math.abs(tempBoard[row] - tempBoard[i]) == Math.abs(row - i) ||
                tempBoard[row] == tempBoard[i]){
            return true;
        }
    }
    return false;
}
```



第一章低可读性之殇

当程序员阅读别人的代码的时候，会对代码产生感觉。就像访问网站一样，如果网站让用户不舒服，用户会关掉离开。而代码不够好的话，程序员会怎么反应呢？恐怕最多的就是说难听的话了吧？最难堪的是，说完了难听的话之后，程序员发现这原来是他（她）自己写的。很显然，低可读性代码是让程序员不爽的。

低可读性代码可能是一个:

- 鼠标滚轮测试程序

长文件、长方法、长分支，这些都强迫程序员不停地滚动鼠标滚轮。这种代码从另外一个角度来说，就是一个鼠标滚轮测试程序。

- 眼力测试程序

  类似的代码让程序员像玩“大家来找茬”一样，让程序员小心翼翼的查找二者的差别。

- 记忆力测试程序

  类似于 a, b, c 或者 count*1, count*2 的名称，让读者不得不去记忆它们的意思。

- 耐性测试程序

  长名称隐藏了关键信息，需要程序员停下来思考。没有格式化的代码让程序员耐住性子去阅读。深度嵌套让程序员不得不认真思考其中的关系。

低可读性代码可能具有下列特质:

- 无意义的命名

  方法名: doProcess, invokeMethod, executeService.

  变量名: returnValue, tempValue, list, result.

- 方言

  不使用英语会让其他读者无法完全明白名称的含义。即使他们拥有共同的语言也不行。

  中文的’YouYu’ 至少有八种意思。

  日文的’ikou’ 至少有四种意思。

- 自造词

  自造词会让其他读者很苦恼。有的时候甚至带来误解。比如：’float DBL’。会让读者在看到 DBL 的时候误以为是 double 的缩写。

- 脑筋急转弯

  用复杂的表达把简单的问题复杂化。比如下面的代码:

```
1 if ((hardKeyboardHidden == 1) == false)
```

低可读性代码同样的也很难变更，扩展或者修复 Bug。低可读性代码都有一个共同的特点：让读者停下来思考。而且它会降低效率和质量。

1.1 恼人的 Bug

程序员不可避免的要面对 Bug，而如果 Bug 很容易被修复的话，程序员恐怕也不会多讨厌 Bug。但是现实世界中，很多 bug 是难以重现的，难以跟踪的，难以修复的，甚至容易造成降级（修改完 Bug 之后引起了新 Bug 或者旧 Bug 重新出现了）。这恐怕是程序员们对 Bug 感到头疼的主要原因吧。

- 难以重现

  一个可能的原因是造成 Bug 的原因是多种因素的共同作用。而让这些因素都达到当时的状态是有困难的。而这些因素往往都是存在于代码内部，而不是业务逻辑。这折射出来的是代码内部增加了影响因素。如果可以的话，减少这些因素会让 Bug 的重现率提高。

- 难以跟踪

  - 一个可能的原因是 Bug 是一行比较复杂的代码造成的。下列代码发生了 NullPointerException 的话，究竟是哪个变量造成的呢?

```
1 product.setUpdater(user.getFullName());
```

  - 另一个可能的原因是对于第三方库的使用方式不当，比如，Spring framework 允许通过配置 xml 来运行系统，而 xml 中的错误很难被发现。
  - 第三种可能的原因是错误消息并不能引导到真正的原因，例如，Ebean 允许用户更容易的访问数据库，但是如果 Server 名称配置错误的话，用户会得到一个 NullPointerException 而陷入错误的方向去寻找问题。直到仔细阅读了 Ebean 的相关代码才能够发现问题的真正原因。

- 难以修复

  历尽千辛万苦，程序员终于找到了问题的原因，但是却发现 Bug 难以修复。因为那是一个有重大影响性的代码，牵一发动全身。需要太多的测试，而时间却不够用了。

- 降级 Bug

  终于修复了 Bug，也能够正常工作了，结果却由于本次修改在另外一个地方造成了新的 Bug。

这些都在低可读性代码中发生的频率较高。

1.2 难以理解

一个新程序员加入了一个团队，他（她）需要先阅读代码而后才能够碰这些代码。但是如果代码可读性较差，那么理解代码所需要的时间也就较长。在这个新程序员弄明白代码之前，他 (她) 是无法修复 Bug 或者是对应变更的。同样的，当一个程序员向另一个程序员移交自己的代码的时候也存在这样的问题。而这个过程很大程度上取决于代码有多好懂，如果代码易读好懂，那么这个过程所花费的时间就短，反之则长。

下面是一些常见的导致代码可读性降低的问题。

- 消极表达

  消极词汇中包含了一个" 否" 的意思，而如果把这个“否”放入一个条件表达式，就会变成一个绕口令：“否定的否定是肯定”如果恰好这个词汇不是形容词，那么代码就会变得更难懂了。比如：

```
if ((hardKeyboardHidden == 1) == false)
```

- 循环引用

  如果对象 A 中包含了对象 B，而恰好，对象 B 由于某种原因需要引用对象 A，这时候二者就构成了循环引用。就像下面的代码：

```
class Container {
    Context context;
    Product product;
}
```

```
class Product {
    //This is for calling of 'container.context'
    Container container;
}
```

  这段代码中，如果访问 **container.product.container.product** 会让代码变得非常诡异。而实际上，这里的问题在于从属关系错误。

- 多种意思

  如果一个方法中的局部变量有多种意思，那么读者就不容易判断在某一行上这个变量的含义了。下面的代码中的 flag 就让人费解。

```
public String registerLesson(long userId, long lessonId) {
    int flag = 0;

    flag = checkUserExist(userId);

    if (flag == 0) {
        return "User does not exist.";
    }

    flag = checkLessonExist(lessonId);

    if (flag == 0) {
        return "Lesson does not exist.";
    }

    return "";
}
```

- 多种分支

  分支是通过条件进行分歧的，但是条件应该针对同一个变量的不同的值进行分支处理的，如果两个连在一起的分支是针对不同条件进行判断的话，就说不通了。

```
public void method(int flag, String name) {
    if (flag == 0) {
        //process 1;
    } else if (name.equals("abc")) {
        //process 2;
    }
}
```

- 暧昧命名

  一个方法的名称如果没有透露出它的意思，读者不得不阅读其中的每一行代码来弄明白它的真实作用。这会花费不少时间。

  下面是一个暧昧命名的例子。

```
public class Keyboard {
    //Pressed the 'alt' key or alter to another keyboard layout?
    public void alt() {
    }
}
```

- 不一致的文档

  很多维护工程师都会因为代码没有值得信赖的文档或者注释而苦恼，这意味着代码是唯一的文档。很不幸，大多数这种情况下的代码也不怎么好读。

1.3 难以扩展或变更

软件会发生变更。当需求变更到来时，需要花费多少时间呢？可变更性越高的代码所要花费的时间就越短。

- 可变更性

  某系统中有个一个widget[7]放置模块，该功能具备磁铁效果。而磁铁效果的算法会因为各种尺寸的 Widget 而变化。但是这个功能却很难被对应，因为其中的 move 方法太复杂了。

[7] http://developer.android.com/guide/topics/appwidgets/index.html

```
private Widget[] widgets;

private int focus;

public void move(int dir) {
    if (dir == 1) {// move right
        if (widgets[focus].width == 1) {
            int next = 0;
            next = findNext();
            if (widgets[next].height == 2) {
                if (widgets[next].y == widgets[focus].y) {
                    //omitted
                }
                //..omitted.
            } else if (widgets[next].height == 3) {
                if (widgets[next].y == widgets[focus].y) {
                    //omitted
                }
                //..omitted.
            }
        }
    } else (dir == -1) {  //move left
        //omitted
    }
}
```

在这个复杂的算法中，程序员几乎罗列了所有的移动 Widget 的可能性，任何新的可能性都会让这个方法膨胀。最终版的代码已经非常臃肿了。任何一个小变动都会让这个方法变得不稳定。

- 可扩展性

  一个 Android 平台的输入法系统需要支持额外的输入设备：遥控器和游戏手柄。

  下面是该系统的一个框架性的代码描述：

```
public class Keyboard {
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (hardKeyboardConnected == 0) {
            //treat with soft keyboard
        } else {
            //treat with hard keyboard
            if (keyCode >= KeyEvent.KEYCODE_A
                && keyCode <= KeyEvent.KEYCODE_Z) {
            } else if (keyCode == KeyEvent.KEYCODE_ENTER) {
            } else if (keyCode == KeyEvent.KEYCODE_SPACE) {
            } else if (keyCode == KeyEvent.KEYCODE_SHIFT) {
            }
        }
    }
}
```

当插入遥控器或者游戏手柄这样的输入设备的时候，并不会像键盘一样被自动检测，所以，程序员就对代码就做了如下的调整。

```
public class Keyboard {
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (hardKeyboardConnected == 0) {
            //treat with soft keyboard
        } else {
            //treat with hard keyboard
            if (keyCode >= KeyEvent.KEYCODE_A
                && keyCode <= KeyEvent.KEYCODE_Z) {
            } else if (keyCode == KeyEvent.KEYCODE_ENTER) {
            } else if (keyCode == KeyEvent.KEYCODE_SPACE) {
            } else if (keyCode == KeyEvent.KEYCODE_SHIFT) {
            } else if (keyCode >= KeyEvent.KEYCODE_BUTTON_1
                && keyCode <= KeyEvent.KEYCODE_BUTTON_10) {  //for remote control
            } else if (keyCode >= KeyEvent.KEYCODE_BUTTON_A
                && keyCode <= KeyEvent.KEYCODE_BUTTON_C) {  //for game pad
            } else if (keyCode >= KeyEvent.KEYCODE_BUTTON_X
                && keyCode <= KeyEvent.KEYCODE_BUTTON_Z) {  //for game pad
            }
        }
    }
}
```

按照这个趋势发展下去，代码将会变得臃肿不堪，而 Bug 也会层出不穷。

1.4 容易做入 Bug

可读性低的代码比可读性高的代码更容易做入 Bug。比如，可读性低的代码往往有更高的圈复杂度[8]。

- 不一致性

  当调用一个声明了一个动作，却执行了两个动作的方法是很容易产生 Bug 的。因为调用时还同时执行了一个未被声明的动作。

```
public void update() {
    reloadData();
    saveFile();
}
```

- 重复与类似

  当需要调用方法 A，却调用了一个类似的方法 B 的时候也可能产生 Bug。

```
public void clear() {
    //remove all elements
}
```

```
public void clean() {
    //remove all dump data.
}
```

- 暧昧

  当调用一个名称和实现都不是那么好懂的代码的时候，也会迷惑。

```
public int[] countDifferentSeries(int[][] basketTwoDArray) {
    int count = 0;
        //omitted.
    return differentSeriesCount;
}
```

  这个方法用 count 开头，但是却返回一个数组。这就不是很容易在短时间内理解的。如果想对数组中的元素分别进行统计，那么在一个循环中分别统计会不会更好呢?

- 复杂度

  深度嵌套的代码难读又难懂。

[8] http://en.wikipedia.org/wiki/Cylomatic_complexity

```
private Widget[] widgets;

private int focus;

public void move(int dir) {
    if (dir == 1) {// move right
        if (widgets[focus].width == 1) {
            int next = 0;
            next = findNext();
            if (widgets[next].height == 2) {
                if (widgets[next].y == widgets[focus].y) {
                    //omitted
                }
                //..omitted.
            } else if (widgets[next].height == 3) {
                if (widgets[next].y == widgets[focus].y) {
                    //omitted
                }
                //..omitted.
            }
        }
    } else (dir == -1) {  //move left
        //omitted
    }
}
```

修改这种代码会很容易引入 Bug。

1.5 难以重构

重构[9]是一种在不改变代码行为的前提条件下对代码解耦合的技术。低可读性的代码的真正含义难以理解，所以，重构起来也就有一定困难。

- 风格不一致

  有些程序员不在乎代码的风格。在他们的代码中，这两种风格同时存在。

```
a=a+1;
```

[9] http://en.wikipedia.org/wiki/Refactor

```
a = a + 1;
```

在这个示例中，如果读者想要搜索什么的话，都不知道应该以哪种风格为准。只好在检索框中加入新的东西——正则表达式。

而且这会让新的程序员很困惑，到底我应该遵循哪种风格呢?

采用不同风格的代码也会因为减少被发现重复的机会而错过重构的机会。

- 用词不统一

  不光代码风格，代码中的术语或者词语不统一也会让读者困惑。

  在一个类中，save 表示 insert 而，另外一个文件中 save 表示 update，那么为什么不使用 insert 和 update 来规避这种误解呢?

  在一个类中，采用 remove 来减少元素个数，而另一个类中采用 delete，那么为什么不统一以便于管理呢?

  而采用不同词语命名的同一功能，则可能被错过重构的机会。

- 处理不统一

  在一个系统中对同一个对象采用不同的方式处理也会需要额外的时间来理解差异。在一个输入法系统中，输入的英文字母会根据当前的转换规则被转换成对应的自然语言。

  下面的代码简单的展示了这个设计的思路。

```
public interface TransferEngine {
    public List<String> transfer();
}
```

```
public class InputMethodService {
    public List<String> transfer() {
        String text = getTypedAlphabets();
        TransferEngine engine = getSelectedTransferEngine();
        List<String> candidates = engine.transfer(text);
        return candidates;
    }
}
```

当增加新的规则的时候，它应该是 TransferEngine 的一个实现

```java
public class HiraganaEngine implements TransferEngine {
    public List<String> transfer(String text) {
        //...
    }
}
```

但是作者却按照下面的方式修改了代码：

```java
public class InputMethodService {

    public List<String> transfer() {
        String text = getTypedAlphabets();
        TransferEngine engine = getSelectedTransferEngine();

        List<String> candidates = null;

        if (engine instanceof SymbolEngine) {
            candidates = new ArrayList<String>();
            candidates.add(text);
            return candidates;
        }

        candidates = engine.transfer(text);
        return candidates;
    }
}
```

不统一的处理会因为代码增加了僵硬性而难以重构。

- 僵硬

  代码中如果声明了很多全局变量、局部变量，并且贯穿了处理导致分解的时候代码难以拆分，难以变更。

- 粘滞

  当修改一处代码的时候，导致另外一个地方也要同时修改。这种代码就是粘滞性代码。

第二章追本溯源

代码可读性到底是怎么被程序员给弄低了的呢?

- 耍酷

  有些时候，有些资深程序员会写一些让人看不懂的代码以显示他们的水平。这的确很酷，但是这既不值得羡慕也不值得推荐。

  下面是的代码是人民币大写转换代码中的一部分。

```
const string NUM_FORMAT_MONEY = "#L#E#D#C#K#E#D#C#J#E#D#C#I#E#D#C#H#E#D#C#G#E#D#C\
#F#E#D#C#.0B0A";
const string PATTERN = @"((?<=-|^)[^1-9]*)|((?'z'0)[0A-E]*((?=[1-9])|(?'-z'(?=[F-\
L\.]|$))))|((?'b'[F-L])(?'z'0)[0A-L]*((?=[1-9])|(?'-z'(?=[\.]|$))))";
```

  这代码可以工作，但是没人了解它为什么能够工作。用了几个复杂的正则表达式进行货币转换无法带来什么启示。谁能够从这段代码获益呢？如果正好这段代码有一个 Bug，谁能修复它呢？也许这段代码有更高的运行效率吧，但是为了提高非常非常小的计算时间而牺牲可读性值得吗？本书不会教你如何耍酷，而且也不推荐你在商业应用中耍酷。

- 加密

  另一种情况，是程序员为了能够留住客户，于是他们把自己的代码“加密”了，然后发送给客户。这样，谁也无法维护这段加密了的代码。商业诡计也超出了本书的讨论范畴。

- 压缩

  还有一种情况是为了加速 javascript 的下载，所以缩减了代码的尺寸，就像 jQuery 的压缩版一样。

- 不知道

  最常见的情况，其实是程序员并不知道如何才能够写出高可读性的代码。这也是本书要讨论的重点。

延伸阅读

第二章追本溯源

2.1 英语能力

2.2 编程语言语法

2.3 命名

2.4 架构

2.5 结构

2.6 注释

2.7 风格

2.8 限制性规则

2.9 管理人员的误导

第三章一次可读性改进体验

3.1 单词

3.2 风格

3.3 算法

3.4 修订

第 II 部分可读性改进

第四章正确的使用英语

4.1 常见错误

4.2 前缀、后缀

4.3 成对词

4.4 缩写

4.5 词性

4.6 语法

4.7 时态

4.8 消极表达

第五章利用编程语言特性

5.1 注解

5.2 异常

5.3 反射

5.4 泛型

5.5 Lambda 表达式

第六章命名的改进

6.1 包的命名

6.2 接口的命名

6.3 类的命名

6.4 枚举的命名

6.5 注解的命名

6.6 方法的命名

6.7 变量的命名

6.8 常量的命名

6.9 同位语

6.10 关键词

6.11 其他改进

第七章架构的改进

7.1 垂直分割与水平分割

7.2 内部类

7.3 匿名类

7.4 关系

7.5 有限取值范围

7.6 防止类爆炸

7.7 抽象

7.8 职责

第八章结构的改进

8.1 缩短长方法

8.2 减少参数个数

8.3 减少嵌套层数

8.4 消除重复和类似

8.5 解耦合

8.6 减少多因素影响

8.7 避免强制类型转换

8.8 缩短长方法

8.9 去除无用代码

8.10 去除临时变量

第九章注释的改进

9.1 JavaDoc

9.2 FIXME, TODO, XXX

9.3 版权

9.4 类说明

9.5 方法注释

9.6 处理的注释

9.7 注释掉的代码

第十章风格的改进

10.1 缩进

10.2 对齐

10.3 空格

10.4 代码区域

10.5 行长与 Tab

10.6 换行

10.7 括号

10.8 长度与深度

第十一章脚本的改进

11.1 面向对象的 Javascript

11.2 JSP, HTML

11.3 SQL

11.4 操作系统脚本

第十二章配置文件的改进

12.1 XML

12.2 JSON

12.3 属性文件

12.4 Yaml

12.5 CSS

第十三章符号的改进

13.1 点

13.2 逻辑运算

13.3 运算符号

13.4 引号

13.5 字符串组合参数

第十四章算法的改进

14.1 递归调用

14.2 用 Map 代替 List

14.3 用 List 代替 Index

14.4 应用领域特定语言

14.5 应用设计模式

14.6 用单线程代替多线程

第十五章测试代码的可读性

15.1 断言

15.2 测试用例

15.3 测试语法

15.4 测试夹具

15.5 测试代码的稳定性

第十六章其他改进

16.1 文件与文件夹

16.2 URL

16.3 动态编程语言

16.4 混合编程

16.5 Log 的记录

16.6 版本履历

第 III 部分可读性探索

第十七章代码自动生成器

17.1 自动生成的代码的问题

17.2 布局编辑器

17.3 动态代码生成

第十八章 IDE 的帮助

18.1 自动格式化

18.2 拼写检查

18.3 样式检查

18.4 自动完成

第十九章代码复查

19.1 机器的工作

19.2 速读

19.3 代码的 5S

19.4 看起来很好

19.5 编码约定

19.6 代码复查的误区

第二十章遗留系统的改善

20.1 确保先有测试

20.2 先利其器

20.3 局部改善

20.4 整体改善

20.5 推翻重写

第二十一章演进式设计

21.1 从第一行开始保持可读性

21.2 保持代码整洁

21.3 够用就好

21.4 想像一下它将如何被调用

21.5 使用第三方库

21.6 养成习惯

21.7 测试的时机

第二十二章可读代码之旅

22.1 创建 Android IME 服务

22.2 布局

22.3 输入设备

22.4 处理按键 (输入)

22.5 处理按键 (操作)

22.6 控制输入 (触摸屏)