

动态评估取舍

——高程序员的 45 个习惯之一

“性能、生产力、优雅、成本以及上市时间，在软件开发过程中都是至关重要的因素。每一项都必须达到最理想状态。”

可能曾经身处这样的团队：管理层和客户将很大一部分注意力都放在应用的界面展示上。也有这样的团队，其客户认为性能表现非常重要。在团队中，你可能会发现，有这样一个开发主管或者架构师，他会强调遵守“正确”的范式比其他任何事情都重要。对任何单个因素如此独断地强调，而不考虑它是否是项目成功的必要因素，必然导致灾难的发生。

强调性能的重要性情有可原，因为恶劣的性能表现会让一个应用在市场上铩羽而归。然而，如果应用的性能已经足够好了，还有必要继续投入精力让其运行得更快一点吗？大概不用了吧。一个应用还有很多其他方面的因素同样重要。与其花费时间去提升千分之一的性能表现，也许减少开发投入，降低成本，并尽快让应用程序上市销售更有价值。

举例来说，考虑一个必须要与远程 Windows 服务器进行通讯的 .NET Windows 应用程序。可以选择使用 .NET Remoting 技术或 Web Services 来实现这个功能。现在，针对使用 Web Services 的提议，有些开发者会说：“我们要在 Windows 之间进行通信，通常此类情况下，推荐使用 .NET Remoting。而且，Web Services 很慢，我们会遇到性能问题。”嗯，一般来说确实是这样。

然而，在这个例子中，使用 Web Services 很容易开发。对 Web Services 的性能测试表明 XML 文档很小，并且相对应用程序自己的响应时间来讲，花在创建和解析 XML 上的时间几乎可以忽略不计。使用 Web Services 不但可以在短期内节省开发时间，且在此后团队被迫使用第三方提供的服务时，Web Services 也是个明智的选择。

Andy 说。。。。

过犹不及

我曾经遇到这样一个客户，他们坚信可配置性的重要性，致使他们的应用有大概 10 000 个可配置变量。新增代码变得异常艰难，因为要花费大量时间来维护配置应用程序和数据库。但是他们坚信需要这种程度的灵活性，因为每个客户都有不同的需求，需要不同的设置。

可实际上，他们只有 19 个客户，而且预计将来也不会超过 50 个。他们并没有很好地去权衡。

考虑这样一个应用，从数据库中读取数据，并以表格方式显示。你可以使用一种优雅的、面向对象的方式，从数据库中取数据，创建对象，再将它们返回给 UI 层。在 UI 层中，你再从对象中拆分出数据，并组织为表格方式显示。除了看起来优雅之外，这样做还有什么好处吗？

也许你只需要让数据层返回一个 dataset 或数据集合，然后用表格显示这些数据即可。这样还可以避免对象创建和销毁所耗费的资源。如果需要的只是数据展示，为什么要创建对象去自找麻烦呢？不按书上说的 OO 方式来做，可以减少投入，同时获得性能上的提升。当

然，这种方式有很多缺点，但问题的关键是要多长个心眼儿，而不是总按照习惯的思路去解决问题。

总而言之，要想让应用成功，降低开发成本与缩短上市时间，二者的影响同样重要。由于计算机硬件价格日益便宜，处理速度日益加快，所以可在硬件上多投入以换取性能的提升，并将节省下来的时间放在应用的其他方面。

当然，这也不完全对。如果硬件需求非常庞大，需要一个巨大的计算机网格以及众多的支持人员才能维持其正常运转（比如类似 Google 那样的需求），那么考虑就要向天平的另一端倾斜了。

但是谁来最终判定性能表现已经足够好，或是应用的展现已经足够“炫”了呢？客户或是利益相关者必须进行评估，并做出相关决定（见第 45 页中习惯 10）。如果团队认为性能上还有提升的空间，或者觉得可以让某些界面看起来更吸引人，那么就去咨询一下利益相关者，让他们决定应将重点放在哪里。

没有适宜所有状况的最佳解决方案。你必须对手上的问题进行评估，并选出最合适的解决方案。每个设计都是针对特定问题的——只有明确地进行评估和权衡，才能得出更好的解决方案。

没有最佳解决方案（No best solution）

动态评估权衡

考虑性能、便利性、生产力、成本和上市时间。如果性能表现足够了，就将注意力放在其他因素上。不要为了感觉上的性能提升或者设计的优雅，而将设计复杂化。

切身感受

即使不能面面俱到，你也应该觉得已经得到了最重要的东西——客户认为有价值的特性。

平衡的艺术

如果现在投入额外的资源和精力，是为了将来可能得到的好处，要确认投入一定要得到回报（大部分情况下，是不会有回报的）。真正的高性能系统，从一开始设计时就在这个方向努力。

过早的优化是万恶之源。

过去用过的解决方案对当前的问题可能适用，也可能不适用。不要事先预设结论，先看看现在是什么状况。

代码要清晰地表达意图

—— 高效程序员的 45 个习惯之一

“可以工作而且易于理解的代码挺好，但是让人觉得聪明更加重要。别人给你钱是因为你脑子

好使，让我们看看你到底有多聪明。”

Hoare 谈软件设计

C.A.R. Hoare_

设计软件有两种方式。一种是设计得尽量简单，并且明显没有缺陷。另一种方式是设计得尽量复杂，并且没有明显的缺陷。

我们大概都见过不少难以理解和维护的代码，而且（最坏的是）还有错误。当开发人员像一群旁观者见到 UFO 一样围在代码四周，同样也感到恐惧、困惑与无助时，这个代码的质量就可想而知了。如果没有人理解一段代码的工作方式，那这段代码还有什么用呢？

开发代码时，应该更注重可读性，而不是只图自己方便。代码被阅读的次数要远远超过被编写的次数，所以在编写的时候值得花点功夫让它读起来更加简单。实际上，从衡量标准上来看，代码清晰程度的优先级应该排在执行效率之前。

例如，如果默认参数或可选参数会影响代码可读性，使其更难以理解和调试，那最好明确地指明参数，而不是在以后让人觉得迷惑。

在改动代码以修复 bug 或者添加新功能时，应该有条不紊地进行。首先，应该理解代码做了什么，它是如何做的。接下来，搞清楚将要改变哪些部分，然后着手修改并进行测试。作为第 1 步的理解代码，往往是最难的。如果别人给你的代码很容易理解，接下来的工作就省心多了。要敬重这个黄金法则，你欠他们一份情，因此也要让你自己的代码简单、便于阅读。

明白地告诉阅读程序的人，代码都做了什么，这是让其便于理解的一种方式。让我们看一些例子。

```
coffeeShop.PlaceOrder(2);
```

通过阅读上面的代码，可以大致明白这是要在咖啡店中下一个订单。但是，2 到底是什么意思？是意味着要两杯咖啡？要再加两次？还是杯子的大小？要想搞清楚，唯一的方式就是去看方法定义或者文档，因为这段代码没有做到清晰易懂。

所以我们不妨添加一些注释。

```
coffeeShop.PlaceOrder(2 /* large cup */);
```

现在看起来好一些了，但是注释有时候是用来对写得很差的代码进行补偿的（见第 105 页中习惯 26：用代码沟通）。

Java 5 与 .NET 中有枚举值的概念，我们不妨使用一下。使用 C#，我们可以定义一个名为 CoffeeCupSize 的枚举，如下所示。

```
public enum CoffeeCupSize  
{
```

```
Small,  
Medium,  
Large  
    }
```

接下来就可以用它来下单要咖啡了。

```
coffeeShop.PlaceOrder(CoffeeCupSize.Large);
```

这段代码就很明白了，我们是要一个大杯 [①] 的咖啡。

作为一个开发者，应该时常提醒自己是否有办法让写出的代码更容易理解。下面是另一个例子。

```
Line 1 public int compute(int val)  
    - {  
    - int result = val << 1;  
    - //... more code ...  
    5 return result;  
    - }
```

第 3 行中的位移操作符是用来干什么的？如果善于进行位运算，或者熟悉逻辑设计或汇编编程，就会明白我们所做的只是把 `val` 的值乘以 2。

PIE 原则

所写的代码必须明确表达你的意图，而且必须富有表现力。这样可以使代码更易于被别人阅读和理解。代码不让人迷惑，也就减少了发生潜在错误的可能。代码要清晰地表达意图。

但对没有类似背景的人们来说，又会如何——他们能明白吗？也许团队中有一些刚刚转行做开发、没有太多经验的成员。他们会挠头不已，直到把头发抓下来 [②]。代码执行效率也许很高，但是缺少明确的意图和表现力。

用位移做乘法，是在对代码进行不必要且危险的性能优化。`result=val*2` 看起来更加清晰，也可以达到目的，而且对于某种给定的编译器来说，可能效率更高（积习难改，见第 34 页的习惯 7）。不要表现得好像很聪明似的，要遵循 PIE 原则：代码要清晰地表达意图。

要是违反了 PIE 原则，造成的问题可就不只是代码可读性那么简单了——它会影响到代码的正确性。下列代码是一个 C# 方法，试图同步对 `CoffeeMaker` 中 `MakeCoffee()` 方法进行调用。

```
Public void MakeCoffee()  
{  
    lock(this)  
    {
```

```
    // ... operation  
  }  
}
```

这个方法的作者想设置一个临界区（critical section）——任何时候最多只能有一个线程来执行 *operation* 中的代码。要达到这个目的，作者在 *CoffeeMaker* 实例中声明了一个锁。一个线程只有获得这个锁，才能执行这个方法。（在 Java 中，会使用 `synchronized` 而不是 `lock`，不过想法是一样的。）

对于 Java 或 .NET 程序员来说，这样写顺理成章，但是其中有两个小问题。首先，锁的使用影响范围过大；其次，对一个全局可见的对象使用了锁。我们进一步来看看这两个问题。

假设 *Coffeemaker* 同时可以提供热水，因为有些人希望早上能够享用一点伯爵红茶。我想同步 `GetWater()` 方法，因此调用其中的 `lock(this)`。这会同步任何在 *CoffeeMaker* 上使用 `lock` 的代码，也就意味着不能同时制作咖啡以及获取热水。这是开发者原本的意图吗还是锁的影响范围太大了？通过阅读代码并不能明白这一点，使用代码的人也就迷惑不已了。

同时，`MakeCoffee()` 方法的实现在 *CoffeeMaker* 对象上声明了一个锁，而应用的其他部分都可以访问 *CoffeeMaker* 对象。如果在一个线程中锁定了 *CoffeeMaker* 对象实例，然后在另外一个线程中调用那个实例之上的 `MakeCoffee()` 方法呢？最好的状况也会执行效率很差，最坏的状况会带来死锁。

让我们在这段代码上应用 PIE 原则，通过修改让它变得更加明确吧。我们不希望同时有两个或更多的线程来执行 `MakeCoffee()` 方法。那为什么不能为这个目的创建一个对象并锁定它呢？

```
Private object makeCoffeeLock = new Object();
```

```
Public void MakeCoffee()  
{  
    lock (makeCoffeeLock)  
    {  
        // ... operation  
    }  
}
```

这段代码解决了上面的两个问题——我们通过指定一个外部对象来进行同步操作，而且更加明确地表达了意图。

在编写代码时，应该使用语言特性来提升表现力。使用方法名来传达意向，对方法参数的命名要帮助读者理解背后的想法。异常传达的信息是哪些可能会出问题，以及如何进行防御式编程，要正确地使用和命名异常。好的编码规范可以让代码变得易于理解，同时减少不必要的注释和文档。

要编写清晰的而不是讨巧的代码

向代码阅读者明确表明你的意图。可读性差的代码一点都不聪明。

切身感受

应该让自己或团队的其他任何人，可以读懂自己一年前写的代码，而且只读一遍就知道它的运行机制。

平衡的艺术

现在对你显而易见的事情，对别人可能并不显然，对于一年以后的你来说，也不一定显然。不妨将代码视作不知道会在未来何时打开的一个时间胶囊。

不要明日复明日。如果现在不说的话，以后你也不会做的。

有意图的编程并不是意味着创建更多的类或者类型。这不是进行过分抽象的理由。

使用符合当时情形的耦合。例如，通过散列表进行松耦合，这种方式适用于在实际状况中就是松耦合的组件。不要使用散列表存储紧密耦合的组件，因为这样没有明确表示出你的意图。

[①] 对星巴克的粉丝来说，这是指 *venti*。

[②] 没错，那不是一块秃顶，而是一个编程机器的太阳能电池板。

记录问题解决日志

—— 高效程序员的 45 个习惯之习惯 33

“在开发过程中是不是经常遇到似曾相识的问题？这没关系。以前解决过的问题，现在还是可以解决掉的。”

面对问题（并解决它们）是开发人员的一种生活方式。当问题发生时，我们希望赶紧把它解决掉。如果一个熟悉的问题再次发生，我们会希望记起第一次是如何解决的，而且希望下次能够更快地把它搞定。然而，有时一个问题看起来跟以前遇到的完全一样，但是我们却不记得是如何修复的了。这种状况时常发生。

不能通过 Web 搜索获得答案吗？毕竟互联网已经成长为如此令人难以置信的信息来源，我们也应该好好加以利用。从 Web 上寻找答案当然胜过仅靠个人努力解决问题。可这是非常耗费时间的过程。有时可以找到需要的答案，有时除了找到一大堆意见和建议之外，发现不了实质性的解决方案。看到有多少开发人员遇到同样的问题，也许会感觉不错，但我们需要的是一个解决办法。

要想得到更好的效果，不妨维护一个保存曾遇到的问题以及对应解决方案的日志。这样，当问题发生时，就不必说：“嘿，我曾碰到过这个问题，但是不记得是怎么解决的了。”可以快速搜索以前用过的方法。工程师们已经使用这种方式很多年了，他们称之为 每日日志（*daylog*）。

不要在同一处跌倒两次

Don't get burned twice

可以选择符合需求的任何格式。下面这些条目可能会用得上。

问题发生日期。

问题简述。

解决方案详细描述。

引用文章或网址，以提供更多细节或相关信息。

任何代码片段、设置或对话框的截屏，只要它们是解决方案的一部分，或者可以帮助更深入地理解相关细节。

要将日志保存为可供计算机搜索的格式，就可以进行关键字搜索以快速查找细节。图 7-1 展示了一个简单的例子，其中带有超链接以提供更多信息。

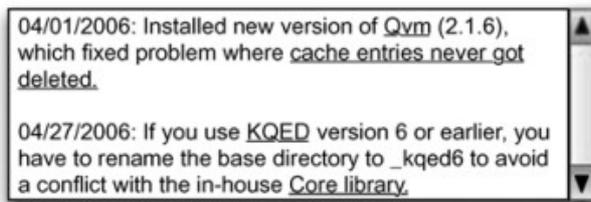


图 7-1 带有超链接的解决方案条目示例

如果面临的问题无法在日志中找到解决方案，在问题解决之后，要记得马上将新的细节记录到日志中去。

要共享日志给其他人，而不仅仅是靠一个人维护。把它放到共享的网络驱动器中，这样其他人也可以使用。或者创建一个 Wiki，并鼓励其他开发人员使用和更新其内容。

维护一个问题及其解决方案的日志。

保留解决方案是修复问题过程的一部分，以后发生相同或类似问题时，就可以很快找到并使用了。

切身感受

解决方案日志应该作为思考的一个来源，可以在其中发现某些特定问题的细节。对于某些类似但是有差异的问题，也能从中获得修复的指引。

平衡的艺术

记录问题的时间不能超过在解决问题上花费的时间。要保持轻量级和简单，不必达到对外发布式的质量。

找到以前的解决方法非常关键。使用足够的关键字，可以帮助你在需要的时候发现需要的条目。

如果通过搜索 Web，发现 没人 曾经遇到同样的问题，也许搜索的方式有问题。

要记录发生问题时应用程序、应用框架或平台的特定版本。同样的问题在不同的平台或版本上可能表现得不同。

要记录团队做出一个重要决策的原因。否则，在 6~9 个月之后，想再重新回顾决策过程的时候，这些细节就很难再记得了，很容易发生互相指责的情形。

附件下载：



【连载】优秀程序员的 45 个习惯之 34——警告就是错误

2009-12-29 17:00:06

标签: [\[推送到技术圈\]](#)

警告就是错误

—— 高效程序员的 45 个习惯之习惯 34

“编译器的警告信息只不过是给过分小心和过于书呆子气的人看的。它们只是警告而已。如果导致的后果很严重，它们就是错误了，而且会导致无法通过编译。所以干脆忽略它们就是了。”

当程序中出现一个编译错误时，编译器或是构建工具会拒绝产生可执行文件。我们别无选择 —— 必须要先修正错误，再继续前行。

然而，警告却是另外一种状况。即使代码编译时产生了警告，我们还是可以运行程序。那么忽略警告信息继续开发代码，会导致什么状况呢？这样做等于是坐在了一个嘀嗒作响的定时炸弹上，而且它很有可能在最糟糕的时刻爆炸。

有些警告是过于挑剔的编译器的良性副产品，有些则不是。例如，一个关于未被使用的变量的警告，可能不会产生什么恶劣影响，但却有可能是暗示某些变量被错误使用了。

最近在一家客户那里，Venkat 发现一个开发中的应用有多于 300 个警告。其中一个被开发人员忽略的警告是这样：

```
Assignment in conditional expression is always constant;  
did you mean to use == instead of = ?  
条件表达式中的赋值总为常量，你是否要使用 == 而不是 = ?
```

相关代码如下：

```
if (theTextBox.Visible = true)  
...
```

也就是说，if 语句总是会评估为 true，无论不幸的 theTextBox 变量是什么状况。看到类似这样真正的错误被当作警告忽略掉，真是令人感到害怕。

看看下面的 C# 代码：

```
public class Base  
{  
    public virtual void foo()  
    {  
        Console.WriteLine("Base.foo");  
    }  
}
```

```
}  
  
public class Derived : Base  
{  
    public virtual void foo()  
    {  
        Console.WriteLine("Derived.foo ");  
    }  
}  
  
class Test  
{  
    static void Main(string[] args)  
    {  
        Derived d = new Derived();  
        Base b = d;  
        d.foo();  
        b.foo();  
    }  
}
```

在使用 Visual Studio 2003 默认的项目设置对其进行编译时，会看到如此信息“构建：1 个成功，0 失败，0 跳过”显示在 Output 窗口的底部。运行程序，会得到这样的输出：

```
Derived.foo  
Base.foo
```

但这不是我们预期的结果。应该看到两次对 Derived 类中 foo 方法的调用。是哪里出错了？如果仔细查看 [输出] 窗口，可以发现这样的警告信息：

```
Warning. Derived.foo hides inherited member Base.foo  
To make the current member override that implementation,  
add the override keyword. Otherwise, you' d add the new keyword.
```

这明显是一个 错误 —— 在 Derived 类的 foo() 方法中，应该使用 override 而不是 virtual 。[\[1\]](#) 想象一下，有组织地忽略代码中类似这样的错误会导致什么样的后果。代码的行为会变得无法预测，其质量会直线下降。

可能有人会说优秀的单元测试可以发现这些问题。是的，它们可以起到帮助作用（而且也应该使用优秀的单元测试）。可如果编译器可以发现这种问题，那为什么不利用它呢？这可以节省大量的时间和麻烦。

要找到一种方式让编译器将警告作为错误提示出来。如果编译器允许调整警告的报告级别，那就把级别调到最高，让任何警告不能被忽略。例如，GCC 编译器支持 -Werror 参数，

在 Visual Studio 中，开发人员可以改变项目设置，将警告视为错误。

对于一个项目的警告信息来说，至少也要做到这种地步。然而，如果采取这种方式，就要对创建的每个项目去进行设置。如果可以尽量以全局化的方式来进行设置就好了。

比如，在 Visual Studio 中，开发人员可以修改项目模板（查看 *.NET Gotchas* [Sub05] 获取更多细节），这样在计算机上创建的任何项目，都会有同样的完整项目设置。在当前版本的 Eclipse 中，可以按照这样的顺序修改设置：Windows→Preferences→Java→Compiler→Errors/Warnings。如果使用其他的语言或 IDE，花一些时间来找出如何在其中将警告作为错误处理吧。

在修改设置的时候，要记得在构建服务器上使用的持续集成工具中，修改同样的设置选项。（要详细了解持续集成，查看在第 87 页上的习惯 21。）这个小小的设置，可以大大提升团队签入到源码控制系统中的代码质量。

在开始一个项目的时候，要把相关的设置都准备好。在项目进行到一半的时候，突然改变警告设置，有可能会带来颠覆性的后果，导致难以控制。

编译器可以轻易处理警告信息，可是你不能。

将警告视为错误

签入带有警告的代码，就跟签入有错误或者没有通过测试的代码一样，都是极差的做法。签入构建工具中的代码不应该产生任何警告信息。

切身感受

警告给人的感觉就像……哦，警告。它们就某些问题给出警告，来吸引开发人员的注意。
平衡的艺术

虽然这里探讨的主要是编译语言，解释型语言通常也有标志，允许运行时警告。使用相关标志，然后捕获输出，以识别并最终消除警告。

由于编译器的 bug 或是第三方工具或代码的原因，有些警告无法消除。如果确实没有应对之策的话，就不要再浪费更多时间了。但是类似的状况很少发生。

应该经常指示编译器：要特别注意别将无法避免的警告作为错误进行提示，这样就不用费力去查看所有的提示，以找到真正的错误和警告。

弃用的方法被弃用是有原因的。不要再使用它们了。至少，安排一个迭代来将它们（以及它们引起的警告信息）安全地移除掉。

如果将过去开发完成的方法标记为弃用方法，要记录当前用户应该采取何种变通之策，以及被弃用的方法将会在何时一起移除。

[1] 这对 C++ 程序员来讲是一个潜伏的陷阱。在 C++ 中代码可以按预期方式工作。

【连载】优秀程序员的 45 个习惯之 35——对问题各个击破

2009-12-29 17:02:38

标签: [\[推送到技术圈\]](#)

对问题各个击破

—— 高效程序员的 45 个习惯之习惯 35

“逐行检查代码库中的代码确实很令人恐惧。但是要调试一个明显的错误，只有去查看整个系统的代码，而且要全部过一遍。毕竟你不知道问题可能发生在什么地方，这样做是找到它的唯一方式。”

单元测试（在第 76 页，第 5 章）带来的积极效应之一，是它会强迫形成代码的分层。要保证代码可测试，就必须把它从周边代码中解脱出来。如果代码依赖其他模块，就应该使用 mock 对象，来将它从其他模块中分离开。这样做不但让代码更加健壮，且在发生问题时，也更容易定位来源。

否则，发生问题时有可能无从下手。也许可以先使用调试器，逐行执行代码，并试图隔离问题。也许在进入到感兴趣的部分之前，要运行多个表单或对话框，这会导致更难发现问题的根源。你会发现自己陷入整个系统之中，徒然增加了压力，而且降低了工作效率。

大型系统非常复杂 —— 在执行过程中会有很多因素起作用。从整个系统的角度来解决问题，就很难区分开，哪些细节对要定位的特定问题产生影响，而哪些细节没有。

答案很清晰：不要试图马上了解系统的所有细节。要想认真调试，就必须将有问题的组件或模块与其他代码库分离开来。如果有单元测试，这个目的就已经达到了。否则，你就得开动脑筋了。

比如，在一个时间紧急的项目中（哪个项目的时间不紧急呢），Fred 和 George 发现他们面对的是一个严重的数据损毁问题。要花很多精力才能知道哪里出了问题，因为开发团队没有将数据库相关的代码与其他的应用代码分离开。他们无法将问题报告给软件厂商，当然不能把整个代码库用电子邮件发给人家！

于是，他们俩开发了一个小型的原型系统，并展示了类似的症状；然后将其发送给厂商作为实例，并询问他们的专家意见，使用原型帮助他们对问题理解得更清晰。

而且，如果他们无法在原型中再现问题的话，原型也可以告诉他们可以工作的代码示例，这也有助于分离和发现问题。

识别复杂问题的第一步，是将它们分离出来。既然不可能在半空中试图修复飞机引擎，为什么还要试图在整个应用中，诊断其中某个组成部分的复杂问题呢？当引擎被从飞机中取出来，而且放在工作台上之后，就更容易修复了。同理，如果可以隔离出发生问题的模块，也更容易修复发生问题的代码。

可是，很多应用的代码在编写时没有注意到这一点，使得分离变得特别困难。应用的各个构成部分之间会彼此纠结：想把这个部分单独拿出来，其他的会紧随而至。在这些状况下，最好花一些时间把关注的代码提取出来，而且创建一个可让其工作的测试环境。

对问题各个击破，这样做有很多好处：通过将问题与应用其他部分隔离开，可以将关注点直接放在与问题相关的议题上；可以通过多种改变，来接近问题发生的核心——你不可能针对正在运行的系统来这样做。可以更快地发现问题的根源所在，因为只与所需最小数量的相关代码发生关系。

隔离问题不应该只在交付软件之后才着手。在构建系统原型、调试和测试时，各个击破的战略都可以起到帮助作用。

对问题各个击破

在解决问题时，要将问题域与其周边隔离开，特别是在大型应用中。

切身感受

面对必须要隔离的问题时，感觉就像在一个茶杯中寻找一根针，而不是大海捞针。

平衡的艺术

如果将代码从其运行环境中分离后，问题消失不见了，这有助于隔离问题。

另一方面，如果将代码从其运行环境中分离后，问题还在，这也有助于隔离问题。

以二分查找的方式来定位问题是很有用的。也就是说，将问题空间分为两半，看看哪一半包含问题。再将包含问题的一半进行二分，并不断重复这个过程。

在向问题发起攻击之前，先查找你的解决问题日志（见第 129 页上的习惯 33）。

【连载】优秀程序员的 45 个习惯之 37——提供有用的错误信息

2009-12-29 17:28:34

标签：[\[推送到技术圈\]](#)

提供有用的错误信息

—— 高效程序员的 45 个习惯之习惯 37

“不要吓着用户，吓程序员也不行。要提供给他们干净整洁的错误信息。要使用

类似‘用户错误。替换，然后继续。’这样让人舒服的词句。”

当应用发布并且在真实世界中得到使用之后，仍然会发生这样那样的问题。比如计算模块可能出错，与数据库服务器之间的连接也可能丢失。当无法满足用户需求时，要以优雅的方式进行处理。

类似的错误发生时，是不是只要弹出一条优雅且带有歉意的信息给用户就足够了？并不尽然。当然了，显示通用的信息，告诉用户发生了问题，要好过由于系统崩溃造成应用执行错误的动作，或者直接关闭（用户会因此感到困惑，并希望知道问题所在）。然而，类似“出错了”这样的消息，无法帮助团队针对问题做出诊断。用户在给支持团队打电话报告问题时，我们希望他们提供足够多且好的信息，以帮助尽快识别问题所在。遗憾的是，用很通用的错误消息，是无法提供足够的数据的。

针对这个问题，常用的解决方案是记录日志：当发生问题时，让应用详细记录错误的相关数据。错误日志最起码应该以文本文件的形式维护。不过也许可以发布到一个系统级别的事件日志中。可以使用工具来浏览日志，产生所有日志信息的 RSS feed，以及诸如此类的辅助方式。

记录日志很有用，可是单单这样做是不够的：开发人员认真分析日志，可以得到需要的数据；但对于不幸的用户来说，起不到任何帮助作用。如果展示给他们类似下图中的信息，他们还是一点头绪都没有——不知道自己到底做错了什么，应该怎么做可以绕过这个错误，或者在给技术支持打电话时，应该报告什么。

如果你注意的话，在开发阶段就能发现这个问题的早期警告。作为开发人员，经常要将自己假定为用户来测试新功能。要是错误信息很难理解，或者无助于定位错误的话，就可以想想真正的用户和支持团队，遇到这个问题时会有多么困难了（见图 7-2）。

图 7-2 无用的异常信息

例如，假定登录 UI 调用了应用的中间层，后台向数据访问层发送了一个请求。由于无法连接数据库，数据访问层抛出一个异常。这个异常被中间层用自己的异常包裹起来，并继续向上传递。那么 UI 层应该怎么做呢？它至少应该让用户知道发生了系统错误，而不是由用户的输入引起的。

接下来，用户会打电话并且告诉我们他无法登录。我们怎么知道问题的实

质是什么呢？日志文件可能有上百个条目，要找到相关的细节非常困难。

实际上，不妨在显示给用户的信息中提供更多细节。好比说，可以看到是哪条 SQL 查询或存储过程发生了错误；这样可以很快找到问题并且修正，而不是浪费大把的时间去盲目地碰运气。不过另一方面，在生产系统中，向用户显示数据连接问题的特定信息，不会对他们有多大帮助。而且有可能吓他们一跳。

一方面要提供给用户清晰、易于理解的问题描述和解释，使他们有可能寻求变通之法。另一方面，还要提供具备关于错误的详细技术细节给用户，这样方便开发人员寻找代码中真正的问题所在。

下面是一种同时实现上述两个目的的方式：图中显示了清晰的错误说明信息。该错误信息不只是简单的文本，还包括了一个超链接。用户、开发人员、测试人员都可以由此链接得到更多信息，如图 7-3、图 7-4 所示。



图 7-3 带有更多细节链接的异常信息

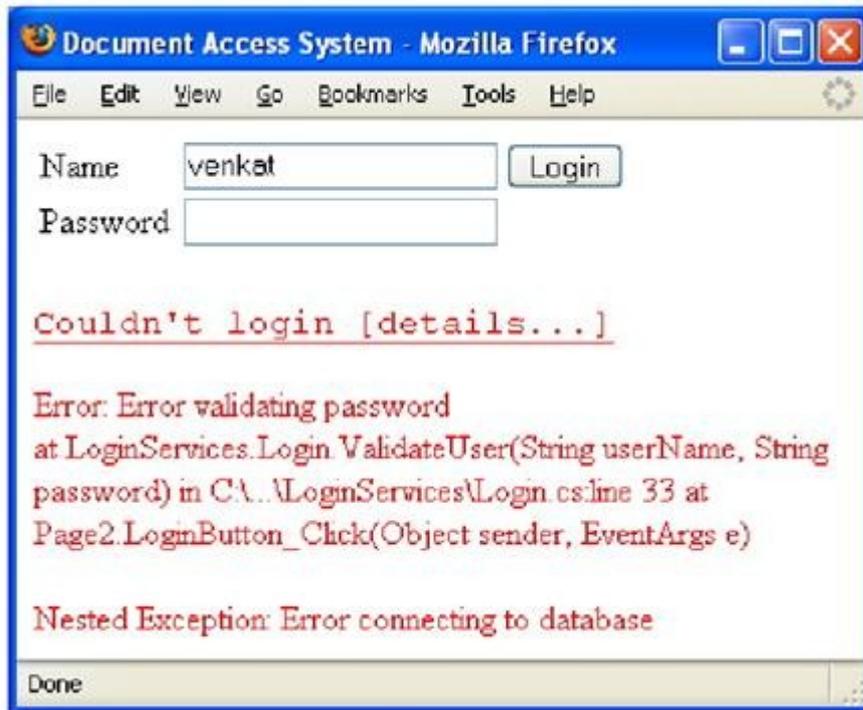


图 7-4 供调试用的完整详细信息

进入链接的页面，可以看到异常（以及所有嵌套异常）的详细信息。在开发时，我们可能希望只要看到这些细节就好了。不过，当应用进入生产系统后，就不能把这些底层细节直接暴露给用户了，而要提供链接，或是某些访问错误日志的入口。支持团队可以请用户点击错误信息，并读出错误日志入口的相关信息，这样支持团队可以很快找到错误日志中的特定细节。对于独立系统来说，点击链接，有可能会将错误信息通过电子邮件发送到支持部门。

除了包括出现问题的详细数据外，日志中记录的信息可能还有当时系统状态的一个快照（例如 Web 应用的会话状态）。[④]

使用上述信息，系统支持团队可以重建发生问题的系统状态，这样对查找和修复问题非常有效。

错误报告对于开发人员的生产率，以及最终的支持活动消耗成本，都有很大的影响。在开发过程中，如果定位和修复问题让人倍受挫折，就考虑使用更加积极主动的错误报告方式吧。调试信息非常宝贵，而且不易获得。不要轻易将其丢弃。

展示有用的错误信息

提供更易于查找错误细节的方式。发生问题时，要展示出尽量多的支持细节，不过别让用户陷入其中。

区分错误类型

程序缺陷。这些是真正的 bug，比如 `NullPointerException`、缺少主键等。用户或者系统管理员对此束手无策。

环境问题。该类别包括数据库连接失败，或是无法连接远程 Web Services、磁盘空间满、权限不足，以及类似的问题。程序员对此没有应对之策，但是用户也许可以找到变通的方法，如果提供足够详细的信息，系统管理员应该可以解决这些问题。

用户错误。程序员与系统管理员不必担心这些问题。在告知是哪里操作的问题后用户可以重新来过。

通过追踪记录报告的错误类型，可以为受众提供更加合适的建议。

切身感受

错误信息有助于问题的解决。当问题发生时，可以详细研究问题的细节描述和发生上下文。

平衡的艺术

像“无法找到文件”这样的错误信息，就其本身而言无助于问题的解决。“无法打开 /andy/project/main.yaml 以供读取”这样的信息更有效。

没有必要等待抛出异常来发现问题。在代码关键点使用断言以保证一切正常。当断言失败时，要提供与异常报告同样详细的信息。

在提供更多信息的同时，不要泄露安全信息、个人隐私、商业机密，或其他敏感信息（对于基于 Web 的应用，这一点尤其重要）。

提供给用户的信息可以包含一个主键，以便于在日志文件或是审核记录中定位相关内容。

[①] 有些安全敏感的信息不应该被暴露，甚至不可以记录到日志中去，其中包括密码、银行账户等。

【连载】优秀程序员的 45 个习惯之 39——架构师必须写代码

2009-12-29 17:23:13

标签：[\[推送到技术圈\]](#)

架构师必须写代码

——高效程序员的 45 个习惯之习惯 39

“我们的专家级架构师 Fred 会提供设计好的架构，供你编写代码。他经验丰富，拿的薪水很高，所以不要用一些愚蠢的问题或者实现上的难点，来浪费他的时间。”

软件开发业界中有许多挂着架构师称号的人。作为作者的我们，不喜欢这个称号

原因如下：架构师应该负责设计和指导，但是许多名片上印着“架构师”的人配不上这个称号。作为架构师，不应该只是画一些看起来很漂亮的设计图，说一些像“黑话”一样的词汇，使用一大堆设计模式——这样的设计通常不会有效的。

不可能在 PowerPoint 幻灯片中
进行编程

You can't code in

PowerPoint

这些架构师通常在项目开始时介入，绘制各种各样的设计图，然后在重要的代码实现开始之前离开。有太多这种“PowerPoint 架构师”了，由于得不到反馈，他们的架构设计工作也不会有很好的收效。

一个设计要解决眼前面临的特定问题，随着设计的实现，对问题的理解也会发生改变。想在开始实现之前，就做出一个很有效的详细设计是非常困难的（见第 48 页上的实践 11）。因为没有足够的上下文，能得到的反馈也很少，甚至没有。设计会随着时间而演进，如果忽略了应用的现状（它的具体实现），要想设计一个新的功能，或者完成某个功能的提升是不可能的。

作为设计人员，如果不能理解系统的具体细节，就不可能做出有效的设计。只通过一些高度概括的、粗略的设计图是没有办法达成对系统的理解的。

这就像是尝试仅仅通过查看地图来指挥一场战役——一旦开打，仅有计划是不够的。战略上的决策也许可以在后方进行，但是战术决策——影响成败的决策——需要对战场状况的明确了解。

可逆性

“程序员修炼之道”丛书中指出不存在所谓的最终决策。没有哪个决策做出之后，就是板上钉钉了。实际上，就时间性来看，不妨把每个重要的决策，都看作沙上堆砌的城堡，它们都是在变化之前所做出的预先规划。

新系统的设计者 **Donald E. Knuth**

新系统的设计者必须要亲自投入到实现中去。

正像 Knuth 说的，好的设计者必须能够卷起袖子，加入开发队伍，毫不犹豫地参与实际编程。真正的架构师，如果不被允许参与编码的话，他们会提出强烈的抗议

有一句泰米尔谚语说：“只有一张蔬菜图无法做出好的咖喱菜。”与之类似，纸上的设计也无法产生优秀的应用。设计应该被原型化，经过测试，当然还有验证——它是要进化的。实现可用的设计，这是设计者或者说架构师的责任。

Martin Fowler 在题为“*Who needs an Architect ?*”的文章中提到：一个真正的架构师“……应该指导开发团队，提升他们的水平，以解决更为复杂的问题”。他接着说：“我认为架构师最重要的任务是：通过找到移除软件设计不可逆性的方式，从而去除所谓架构的概念。”增强可逆性是注重实效的软件实现方式的关键构成部

分。

要鼓励程序员参与设计。主力程序员应该试着担任架构师的角色，而且可以从事多种不同的角色。他会负责解决设计上的问题，同时也不会放弃编码的工作。如果开发人员不愿意承担设计的责任，要给他们配备一个有良好设计能力的人。程序员在拒绝设计的同时，也就放弃了思考。

优秀的设计从积极的程序员那里开始演化

积极的编程可以带来深入的理解。不要使用不愿意编程的架构师——不知道系统的真实情况，是无法展开设计的。

切身感受

架构、设计、编码和测试，这些工作给人的感觉就像是同一个活动 —— 开发 —— 的不同方面。感觉它们彼此之间应该是不可分割的。

平衡的艺术

如果有一个首席架构师，他可能没有足够的时间来参与编码工作。还是要让他参与，但是别让他开发在项目关键路径上的、工作量最大的代码。

不要允许任何人单独进行设计，特别是你自己。

连载】优秀程序员的 45 个习惯之 42——允许大家自己想办法

阅读:356 次 评论:0 条 更新时间:2010-01-05

允许大家自己想办法

—— 高效程序员的 45 个习惯之习惯 42

“你这么聪明，直接把干净利落的解决方案告诉团队其他人就好了。不用浪费时间告诉他们为什么这样做。”

“授人以鱼，三餐之需；授人以渔，终生之用。”告诉团队成员解决问题的方法，也要让他们知道如何解决问题的思路，这也是成为指导者的一部分。

了解上个实践 —— 成为指导者 —— 之后，也许有人会倾向于直接给同事一个答案，以继续完成工作任务。要是只提供一些小指引给他们，让他们自己想办法找到答案，又会如何？

这并不是多么麻烦的事情；不要直接给出像“42”这样的答案，应该问你的队友：“你有没有查看在事务管理者与应用的锁处理程序之间的交互关系？”

这样做有下面几点好处。

- 你在帮助他们学会如何解决问题。
- 除了答案之外，他们可以学到更多东西。
- 他们不会再就类似的问题反复问你。
- 这样做，可以帮助他们在你不能回答问题时自己想办法。
- 他们可能想出你没有考虑到的解决方法或者主意。这是最有趣的——你也可以学到新东西。

如果有人还是没有任何线索，那就给更多提示吧（或者甚至是答案）。如果有人提出某些想法，不妨帮他们分析每种想法的优劣之处。如果有人给出的答案或解决方法更好，那就从中汲取经验，然后分享你的体会吧。这对双方来说都是极佳的学习经验。

作为指导者，应该鼓励、引领大家思考如何解决问题。前面提到过亚里士多德的话：“接纳别人的想法，而不是盲目接受，这是受过教育的头脑的标志。”应该接纳别人的想法和看问题的角度，在这个过程中，自己的头脑也得到了拓展。

如果整个团队都能够采纳这样的态度，可以发现团队的知识资本有快速的提升，而且将会完成一些极其出色的工作成果。

给别人解决问题的机会

指给他们正确的方向，而不是直接提供解决方案。每个人都能从中学到不少东西。

切身感受

感觉不是在以填鸭式的方式给予别人帮助。不是有意掩饰，更非讳莫如深，而是带领大家找到自己的解决方案。

平衡的艺术

- 用问题来回答问题，可以引导提问的人走上正确的道路。

- 如果有人真的陷入胶着状态，就不要折磨他们了。告诉他们答案，再解释为什么是这样。

【连载】优秀程序员的 45 个习惯之 45——及时通报进展与问题

阅读:210 次 评论:2 条 更新时间:2010-01-15

及时通报进展与问题

—— 高效程序员的 45 个习惯之习惯 45

“管理层、项目团队以及业务所有方，都仰仗你来完成任务。如果他们想知道进展状况，会主动找你要的。还是埋头继续做事吧。”

接受一个任务，也就意味着做出了要准时交付的承诺。不过，遇到各种问题从而导致延迟，这种情形并不少见。截止日期来临，大家都等着你在演示会议上展示工作成果。如果你到会后通知大家工作还没有完成，会有什么后果？除了感到窘迫，这对你的事业发展也没有什么好处。

如果等到截止时间才发布坏消息，就等于是为经理和技术主管提供了对你进行微观管理（micromanagement）的机会。他们会担心你再次让他们失望，并开始每天多次检查你的工作进度。你的生活就开始变得像 呆伯特 的漫画一样了。

假定现在你手上有一个进行了一半的任务，由于技术上的难题，看起来不能准时完成了。如果这时积极通知其他相关各方，就等于给机会让他们提前找出解决问题的方案。也许他们可以向另外的开发人员寻求帮助，也许他们可以将工作重新分配给更加熟悉相关技术的人，也许他们可以提供更多需要的资源，或者调整目前这个迭代中要完成的工作范围。客户会愿意将这个任务用其他同等重要的任务进行交换的。

及时通报进展与问题，有情况发生时，就不会让别人感到突然，而且他们也很愿意了解目前的进展状况。他们会知道何时应提供帮助，而且你也获得了他们的信任。

发送电子邮件，用即时贴传递信息，或快速电话通知，这都是通报大家的传统方式。还可以使用 Alistair Cockburn 提出的“信息辐射器”。^[①]信息辐射器类似于墙上的海报，提供变更的信息。路人可以很方便地了解其中的内容。以推送的方式传递信息，他们就不必再来问问题了。信息辐射器中可以展示目前的任务进度，和团队、管理层或客户可能会感兴趣的其他内容。

也可以使用海报、网站、Wiki、博客或者 RSS。只要让人们可以有规律地查看到需要的信息，这就可以了。

整个团队可以使用信息辐射器来发布他们的状态、代码设计、研究出的好点子等内容。现在只要绕着团队的工作区走一圈，就可以学到不少新东西，而且管理层也就可以知道目前的状况如何了。

及时通报进展与问题

发布进展状况、新的想法和目前正在关注的主题。不要等着别人来问项目状态如何。

切身感受

当经理或同事来询问工作进展、最新的设计，或研究状况时，不会感到头痛。

平衡的艺术

- 每日立会（见第 148 页中习惯 38）可以让每个人都能明确了解最新的进展和形势。
- 在展示进度状况时，要照顾到受众关注的细节程度。举例来说，CEO 和企业主是不会关心抽象基类设计的具体细节的。
- 别花费太多时间在进展与问题通报上面，还是应该保证开发任务的顺利完成。
- 经常抬头看看四周，而不是只埋头于自己的工作。