

Defensive Programming

第 8 章

防御式编程

cc2e.com/0861 内容

- 8.1 保护程序免遭非法输入数据的破坏：第 188 页
- 8.2 断言：第 189 页
- 8.3 错误处理技术：第 194 页
- 8.4 异常：第 198 页
- 8.5 隔离程序，使之包容由错误造成的损害：第 203 页
- 8.6 辅助调试的代码：第 205 页
- 8.7 确定在产品代码中该保留多少防御式代码：第 209 页
- 8.8 对防御式编程采取防御的姿态：第 210 页

相关章节

- 信息隐藏：第 5.3 节中的“隐藏秘密（信息隐藏）”
- 为变更而设计：第 5.3 节中的“找出容易改变的区域”
- 软件架构：第 3.5 节
- 软件构建中的设计：第 5 章
- 调试：第 23 章



KEY POINT

防御式编程并不是说让你在编程时持“防备批评或攻击”的态度——“它就是这么工作！”这一概念来自防御式驾驶。在防御式驾驶中要建立这样一种思维，那就是你永远也不能确定另一位司机将要做什么。这样才能确保在其他人做出危险动作时你也不会受到伤害。你要承担起保护自己的责任，哪怕是其他司机犯的错误。防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据。更一般地说，其核心想法是要承认程序都会有问题，都需要被修改，聪明的程序员应该根据这一点来编程序。

本章就是要讲述如何面对严酷的非法数据的世界、在遇到“绝不会发生”的事件以及其他程序员犯下的错误时保护你自己。如果你是一位有经验的程序员，那么可以略过下面关于对输入数据进行处理的一节，而直接进入第 8.2 节“断言”。

8.1

Protecting Your Program from Invalid Inputs 保护程序免遭非法输入数据的破坏

在学校里你可能听说过“垃圾进，垃圾出（garbage in, garbage out.）”这句话。这句话说的是软件开发领域的“出门概不退换”原则：让用户自己操心自己的事。



对已形成产品的软件而言，仅仅“垃圾进，垃圾出”还不够。不管进来什么，好的程序都不会生成垃圾，而是做到“垃圾进，什么都不出”、“进来垃圾，出去是出错提示”或“不许垃圾进来”。按今天的标准来看，“垃圾进，垃圾出”已然成为缺乏安全性的差劲程序的标志。

通常有三种方法来处理进来垃圾的情况。

检查所有来源于外部的数据的值 当从文件、用户、网络或其他外部接口中获取数据时，应检查所获得的数据值，以确保它在允许的范围内。对于数值，要确保它在可接受的取值范围内；对于字符串，要确保其不超长。如果字符串代表的是某个特定范围内的数据（如金融交易 ID 或其他类似数据），那么要确认其取值合乎用途，否则就应该拒绝接受。如果你在开发需要确保安全的应用程序，还要格外注意那些狡猾的可能是攻击你的系统的数据，包括企图令缓冲区溢出的数据、注入的 SQL 命令、注入的 HTML 或 XML 代码、整数溢出以及传递给系统调用的数据，等等。

检查子程序所有输入参数的值 检查子程序输入参数的值，事实上和检查来源于外部的数值一样，只不过数据是来自于其他子程序而非外部接口。第 8.5 节“隔离程序，使之包容由错误造成的损害”阐述了一种实用方法可用于确定哪些子程序需要检查其输入数据。

决定如何处理错误的输入数据 一旦检测到非法的参数，你该如何处理它呢？根据情况的不同，你可以从十几种不同的方案中选择其一，在本章后面第 8.3 节“错误处理技术”中会详细描述这些技术。

防御式编程是本书所介绍的其他提高软件质量技术的有益辅助手段。防御式编码的最佳方式就是在一开始不要在代码中引入错误。使用迭代式设计、编码前先写伪代码、写代码前先写测试用例、低层设计检查等活动，都有助于防止引入错误。因此，要在防御式编程之前优先运用这些技术。所幸的是，你可以把防御式编程和其他技术结合起来使用。

正如图 8-1 所示，防范看似微小的错误，收获可能远远超出你的想象。本章的剩余部分将介绍用于检查外部数据、检查输入参数和处理错误输入数据的许多可选技术。



图8-1 西雅图90号州际浮桥的一部分在一场风暴中沉没了，原因是未遮盖浮箱，而在风暴中进水，使得桥体过重而无法继续漂浮。在建设时要防范一些小事情，它们的严重性往往超过你的预期

8.2 Assertions 断言

断言 (assertion) 是指在开发期间使用的、让程序在运行时进行自检的代码 (通常是一个子程序或宏)。断言为真，则表明程序运行正常，而断言为假，则意味着它已经在代码中发现了意料之外的错误。举例来说，如果系统假定一份客户信息文件所含的记录数不能超过 50 000，那么程序中可以包含一个断定记录数小于等于 50 000 的断言。只要记录数小于等于 50 000，这一断言都会默默无语。然而一旦记录数超过 50 000，它就会大声地“断言”说程序中存在一个错误。



断言对于大型的复杂程序或可靠性要求极高的程序来说尤其有用。通过使用断言，程序员能更快速地排查出因修改代码或者别的原因，而弄进程序里的不匹配的接口假定和错误等。

一个断言通常含有两个参数：一个描述假设为真时的情况的布尔表达式，和一个断言为假时需要显示的信息。下面是假定变量 denominator (分母) 的值应为非零值时 Java 断言的写法：

Java示例：断言

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.;"
```

这个断言声明 denominator 不会等于 0。其中第一个参数，denominator != 0，是个布尔表达式，其结果为 true 或者 false。第二个参数是当第一个参数为 false 时——即断言为假时——要打印的消息。

断言可以用于在代码中说明各种假定，澄清各种不希望的情形。可以用断言检查如下这类假定：

- 输入参数或输出参数的取值处于预期的范围内；
- 子程序开始（或者结束）执行时文件或流是处于打开（或关闭）的状态；
- 子程序开始（或者结束）执行时，文件或流的读写位置处于开头（或结尾）处；
- 文件或流已用只读、只写或可读可写方式打开；
- 仅用于输入的变量的值没有被子程序所修改；
- 指针非空；
- 传入子程序的数组或其他容器至少能容纳 X 个数据元素；
- 表已初始化，存储着真实的数值；
- 子程序开始（或结束）执行时，某个容器是空的（或满的）；
- 一个经过高度优化的复杂子程序的运算结果和相对缓慢但代码清晰的子程序的运算结果相一致。

当然，这里列出的只是一些基本假定，你在子程序中还可以包括更多可以用断言来说明的假定。

正常情况下，你并不希望用户看到产品代码中的断言信息；断言主要是用于开发和维护阶段。通常，断言只是在开发阶段被编译到目标代码中，而在生成产品代码时并不编译进去。在开发阶段，断言可以帮助查清相互矛盾的假定、预料之外的情况以及传给子程序的错误数据等。在生成产品代码时，可以把断言编译进目标代码里去，以免降低系统的性能。

Building Your Own Assertion Mechanism 建立自己的断言机制

交叉参考 建立自己的断言子程序，是“深入一种语言去编程”而不仅是“在一种语言上编程”的很好例子。关于二者的区别，请见第 34.4 节。

包括 C++、Java 和 Microsoft Visual Basic 在内的很多语言都支持断言。如果你用的语言不直接支持断言语句，自己写也是很容易的。C++中标准的 assert 宏并不支持文本信息。下面的例子给出了一个使用 C++宏改进的 ASSERT 实现：

C++示例：一个实现断言的宏

```
#define ASSERT( condition, message ) { \
    if ( !(condition) ) { \
        LogError( "Assertion failed: ", \
                  #condition, message ); \
        exit( EXIT_FAILURE ); \
    } \
}
```

Guidelines for Using Assertions 使用断言的指导建议

下面是关于使用断言的一些指导性建议。

用错误处理代码来处理预期会发生的状况，用断言来处理绝不应该发生的状况 断言是用来检查永远不该发生的情况，而错误处理代码（error-handling code）是用来检查不太可能经常发生的非正常情况，这些情况是能在写代码时就预料到的，且在产品代码中也要处理这些情况。错误处理通常用来检查有害的输入数据，而断言是用于检查代码中的 bug。

用错误处理代码来处理反常情况，程序就能够很从容地对错误做出反映。如果在发生异常情况的时候触发了断言，那么要采取的更正的措施就不仅仅是对错误做出恰当的反映了——而是应该修改程序的源代码并重新编译，然后发布软件的新版本。

有种方式可以让你更好地理解断言，那就是把断言看做是可执行的注解——你不能依赖它来让代码正常工作，但与编程语言中的注释相比，它能更主动地对程序中的假定做出说明。

避免把需要执行的代码放到断言中 如果把代码写在断言里，那么当你关闭断言功能时，编译器很可能就把这些代码排除在外了。比如说，你写了这么一个断言：

交叉参考 你也可以把这个例子看做是把多行语句放入一行中而引起的问题。第31.5节“每行只写一个语句”中有更多的例子。

Visual Basic示例：一种危险的断言使用方法

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```

这段代码的问题在于，如果未编译断言语句，那么其中用于执行操作的代码也就不会被编译。应该把需要执行的语句提取出来，并把其运算结果赋给状态变量，再对这些状态变量进行判断。下面这样使用断言就很安全：

Visual Basic示例：安全地使用断言

```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

推荐阅读 要想更深入地了解前条件和后条件，请阅读《Object-Oriented Software Construction》(Meyer 1997)一书。

用断言来注解并验证前条件和后条件 前条件（*preconditions*）和后条件（*postconditions*）是一种名为“契约式设计（*design by contract*）”的程序设计和开发方法的一部分（Meyer 1997）。使用前条件和后条件时，每个子程序或类与程序的其余部分都形成了一份契约。

前条件是子程序或类的调用方代码在调用子程序或实例化对象之前要确保为真的属性。前条件是调用方代码对其所调用的代码要承担的义务。

后条件是子程序或类在执行结束后要确保为真的属性。后置条件是子程序或类对调用方代码所承担的责任。

断言是用来说明前条件和后条件的有利工具。也可以用注释来说明前条件和后条件，但断言却能动态地判断前条件和后条件是否为真。

在下面这个例子中，就使用了断言来说明 Velocity（速度）子程序的前条件和后条件：

Visual Basic示例：使用断言来记述前条件和后条件

```
Private Function Velocity ( _
    ByVal latitude As Single, _
    ByVal longitude As Single, _
    ByVal elevation As Single _
) As Single

    ' Preconditions
    Debug.Assert( -90 <= latitude And latitude <= 90 )
    Debug.Assert( 0 <= longitude And longitude < 360 )
    Debug.Assert( -500 <= elevation And elevation <= 75000 )1
```

¹ 译注：latitude 是纬度，longitude 是经度，elevation 是海拔高度。

```
...
' Postconditions
Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

' return value
Velocity = returnVelocity
End Function
```

如果变量 `latitude`、`longitude` 和 `elevation` 都是来源于系统外部，那么就应该用错误处理代码来检查和处理非法的数值，而不是使用断言。而如果变量的值是源于可信的系统内部，并且这段程序是基于这些值不会超出合法范围的假定而设计，使用断言则是非常合适的。

交叉参考 关于健壮性的更多内容，请参考本章 8.3 节中的“健壮性与正确性”。

对于高健壮性的代码，应该先使用断言再处理错误 对于每种可能出错的条件，通常子程序要么使用断言，要么使用错误处理代码来进行处理，但是不会同时使用二者。一些专家主张只须使用一种处理方法即可（Meyer 1997）。

然而，现实世界中的程序和项目通常都很混乱，仅仅依赖断言还是不够的。如果开发的是一个大型的、长期运行的系统，那么系统的不同部分可能会由不同的设计人员来设计，耗时可能会超过 5 到 10 年。不同设计师们将在不同的时期工作，还跨越了多个版本。在系统开发的不同时间点，他们在设计时会关注不同的技术。设计人员也可能在地理位置上相互分离，特别是当系统某些部分是外包给别的公司做的时候。程序员在系统生命周期的不同时期会采用不同的编码规范。在一个大型项目的开发团队里，有些程序员明显比其他人更谨慎，因此有的代码部分的复查会比其他代码更严格一些。有些程序员所做的单元测试比其他人更彻底。当测试团队分布在不同的地理位置，并且受到商业压力而导致每次发行版本的测试覆盖范围都不尽相同时，你根本无法指望详尽的系统级别的回归测试。

在这种环境中，可能同时用断言和错误处理代码来处理同一个错误。以 Microsoft Word 为例，在其代码中对应该始终为真的条件都加上了断言，但同时也用错误处理代码处理了这些错误，以应对断言失败的情况。对于那些像 Word 这样特大规模、复杂且生命周期很长的应用程序而言，断言是非常有用的，因为断言可以帮助在开发阶段排查出尽可能多的错误。然而这样的应用程序实在太复杂了（有着上百万行的源代码），而且都经历了多次的修改，以至于想要在软件交付之前发现并纠正一切错误是不现实的，所以在发布的产品中错误也同样需要处理。

下面就说明如何把这一规则应用到 Velocity 一例中去：

Visual Basic示例：使用断言来说明前条件和后条件

```

Private Function Velocity ( _
    ByRef latitude As Single, _
    ByRef longitude As Single, _
    ByRef elevation As Single _
) As Single

    ' Preconditions
    Debug.Assert ( -90 <= latitude And latitude <= 90 )
    Debug.Assert ( 0 <= longitude And longitude < 360 )
    Debug.Assert ( -500 <= elevation And elevation <= 75000 )
    ...

    ' Sanitize input data. Values should be within the ranges asserted above,
    ' but if a value is not within its valid range, it will be changed to the
    ' closest legal value
    If ( latitude < -90 ) Then
        latitude = -90
    ElseIf ( latitude > 90 ) Then
        latitude = 90
    End If
    If ( longitude < 0 ) Then
        longitude = 0
    ElseIf ( longitude > 360 ) Then
        ...

```

这里是断言代码。
这里是在运行时
处理错误输入数
据的代码。

8.3 Error-Handling Techniques 错误处理技术

断言可以用于处理代码中不应发生的错误。那么又该如何处理那些预料中可能要发生的错误呢？根据所处情形的不同，你可以返回中立值（neutral value）、换用下一个正确数据、返回与前次相同的值、换用最接近的有效值、在日志文件中记录警告信息、返回一个错误码、调用错误处理子程序或对象、显示出错信息或者关闭程序——把这些技术结合起来使用。

下面就来详细说明这些可用的技术。

返回中立值 有时，处理错误数据的最佳做法就是继续执行操作并简单地返回一个没有危害的数值。比如说，数值计算可以返回 0，字符串操作可以返回空字符串，指针操作可以返回一个空指针，等等。如果视频游戏中的绘图子程序接收到一个错误的颜色输入，那么它可以用默认的背景色或前景色继续绘制。当然，对于显示癌症病人 X 光片的绘图子程序而言，最好还是不要显示某个“中立值”。在这种情况下，关闭程序也比让它显示错误的病人数据要好。

换用下一个正确的数据 在处理数据流的时候，有时只需返回下一个正确的数据即可。如果在读数据库记录并发现其中一条记录已经损坏时，你可以继续读下去直到又找到一条正确记录为止。如果你以每秒 100 次的速度读取体温计的数据，那么如果某一次得到的数据有误，你只需再等上 1/100 秒然后继续读取即可。

返回与前次相同的数据 如果前面提到的体温计读取软件在某次读取中没有获得数据，那么它可以简单地返回前一次的读取结果。根据这一应用领域的情况，温度在 1/100 秒的时间内不会发生太大改变。而在视频游戏里，如果你发现要用一种无效的颜色重绘屏幕的某个区域，那么可以简单地使用上一次绘图时使用的颜色。但如果你正在管理自动取款机上的交易，你可能不希望用“和最后一次相同的答案”了，因为那可是前一个用户的银行账号！

换用最接近的合法值 在有些情况下，你可以选择返回最接近的那个合法值，就像前面的 *velocity* 例子里那样。在从已经校准的仪器上取值时，这种方法往往是很合理的。比如说，温度计也许已经校准在 0 到 100 摄氏度之间。如果你检测到一次小于 0 的读取结果，那你可以把它替换为 0，即最接近的那个合法值。如果发现结果大于 100，那么你可以把它替换为 100。在操作字符串时，如果发现某个字符串的长度小于 0，你也可以用 0 代替。当我倒车时，汽车就是用这种方法来处理错误的。因为车上的速度表无法显示负的速度，所以当我倒车时它只是简单地显示 0——即最接近的合法值。

把警告信息记录到日志文件中 在检测到错误数据的时候，你可以选择在日志文件（log file）中记录一条警告信息，然后继续执行。这种方法可以同其他的错误处理技术结合使用，比如说换用最接近的合法值、换用下一个正确的数据等。如果你用到了日志文件，要考虑是否能够安全地公开它，或是否需要对其进行加密或施加其他方式的保护。

返回一个错误码 你可以决定只让系统的某些部分处理错误。其他部分则不在本地（局部）处理错误，而只是简单地报告说有错误发生，并信任调用链上游的某个子程序会处理该错误。通知系统其余部分已经发生错误可以采用下列方法之一：

- 设置一个状态变量的值
- 用状态值作为函数的返回值
- 用语言内建的异常机制抛出一个异常

在这种情况下，与确定特定的错误报告机制相比，更为重要的是要决定系统里的哪些部分应该直接处理错误，哪些部分只是报告所发生的错误。如果安全性很重要，请确认调用方的子程序总会检查返回的错误码。

调用错误处理子程序或对象 另一种方法则是把错误处理都集中在一个全局的错误处理子程序或对象中。这种方法的优点在于能把错误处理的职责集中到一起，从而让调试工作更为简单。而代价则是整个程序都要知道这个集中点并与之紧密耦合。如果你想在其他系统中重用其中的某些代码，那就得把错误处理代码一并带过去。

这种方法对代码的安全性有一个重要的影响。如果代码发生了缓冲区溢出，那么攻击者很可能已经篡改了这一处理程序或对象的地址。这样一来，一旦在应用程序运行期间发生缓冲区溢出，再使用这种方法就不再安全了。

当错误发生时显示出错消息 这种方法可以把错误处理的开销减到最少，然而它也可能会让用户界面中出现的信息散布到整个应用程序中。当你需要创建一套统一协调的用户界面时，或当你想让用户界面部分与系统的其他部分清晰地分开时，或当你想把软件本地化到另一种不同的语言时，都会面临挑战。还有，当心不要告诉系统的潜在攻击者太多东西。攻击者有时能利用错误信息来发现如何攻击这个系统。

用最妥当的方式在局部处理错误 一些设计方案要求在局部解决所有遇到的错误——而具体使用何种错误处理方法，则留给设计和实现会遇到错误的这部分系统的程序员来决定。

这种方法给予每个程序员很大的灵活度，但也带来显著的风险，即系统的整体性能将无法满足对其正确性或可靠性的需求（马上还会更具体地讲这个问题）。根据开发人员最终用以处理特定错误的方法不同，这样做也有可能导致与用户界面相关的代码散布到整个系统中，从而又使程序面临那些与显示出错消息相关的问题。

关闭程序 有些系统一旦检测到错误发生就会关闭。这一方法适用于人身安全攸关（safety-critical）的应用程序。举例来说，如果用作控制治疗癌症病人的放疗设备的软件接收到了错误的放射剂量输入数据，那么怎样处理这一错误最好呢？应该使用与上一次相同的数值吗？应该用最接近的合法值吗？应该使用中立值吗？在这种情况下，关闭程序是最佳的选择。哪怕重启机器也比冒险施放错误的放射剂量要好得多。

一种类似的做法可以用来提高 Microsoft Windows 操作系统的安全性。在默认情况下，即使系统的安全日志已经满了，Windows 仍会继续运行。但你可以配置 Windows，让它在安全日志满的时候停止服务。在信息安全攸关（security-critical）的环境中这样做是明智的。

Robustness vs. Correctness

健壮性与正确性

正如前面视频游戏和 X 光机的例子告诉我们的，处理错误最恰当的方式要根据出现错误的软件的类别而定。这两个例子还表明，错误处理方式有时更侧重于正确性，而有时则更侧重于健壮性。开发人员倾向于非形式地使用这两个术语，但严格来说，这两个术语在程度上是截然相反的。**正确性 (correctness)** 意味着永不返回不准确的结果，哪怕不返回结果也比返回不准确的结果好。然而，**健壮性 (robustness)** 则意味着要不断尝试采取某些措施，以保证软件可以持续地运转下去，哪怕有时做出一些不够准确的结果。

人身安全攸关的软件往往更倾向于正确性而非健壮性。不返回结果也比返回错误的结果要好。放射线治疗仪就是体现这一原则的好例子。

消费类应用软件往往更注重健壮性而非正确性。通常只要返回一些结果就比软件停止运行要强。我所用的字处理软件有时会在屏幕下方显示半行文字。如果它检测到这一情况，难道我期望字处理软件退出吗？当然不。我知道等下次再按 Page Up 或 Page Down 键之后屏幕就会刷新，随后显示状态也就恢复正常了。

High-Level Design Implications of Error Processing

高层次设计对错误处理方式的影响



既然有这么多的选择，你就必须注意，应该在整个程序里采用一致的方式处理非法的参数。对错误进行处理的方式会直接关系到软件能否满足在正确性、健壮性和其他非功能性指标方面的要求。确定一种通用的处理错误参数的方法，是架构层次（或称高层次）的设计决策，需要在那里的某个层次上解决。

一旦确定了某种方法，就要确保始终如一地贯彻这一方法。如果你决定让高层的代码来处理错误，而低层的代码只需简单地报告错误，那么就要确保高层的代码真的处理了错误！有些语言允许你忽略“函数返回的是错误码”这一事实——在 C++ 中，你无须对函数的返回值做任何处理——但千万不要忽略错误信息！检查函数的返回值。即使你认定某个函数绝对不会出错，也无论如何要去检查一下。防御式编程全部的重点就在于防御那些你未曾预料到的错误。

这些指导建议对于系统函数和你自己写的函数都是成立的。除非你已确立了一套不对系统调用进行错误检查的架构性指导建议，否则请在每个系统调用后检查错误码。一旦检测到错误，就记下错误代号和它的描述信息。

8.4 Exceptions 异常

异常是把代码中的错误或异常事件传递给调用方代码的一种特殊手段。如果在一个子程序中遇到了预料之外的情况，但不知道该如何处理的话，它就可以抛出一个异常，就好比是举起双手说“我不知道该怎么处理它——我真希望有谁知道该怎么办！”一样。对出错的前因后果不甚了解的代码，可以把对控制权转交给系统中其他能更好地解释错误并采取措施的部分。

还可以用异常来清理一段代码中存在的杂乱逻辑，正如第 17.3 节中“用 try-finally 重写”一例所示。异常的基本结构是：子程序使用 `throw` 抛出一个异常对象，再被调用链上层其他子程序的 `try-catch` 语句捕获。

几种流行的编程语言在实现异常机制时各有千秋。表 8-1 总结了其中三种语言在这方面的主要差异：

表8-1 支持几种流行的编程语言的表达式

跟异常相关的特性	C++	Java	Visual Basic
支持 <code>try-catch</code> 语句	支持	支持	支持
支持 <code>try-catch-finally</code> 语句	不支持	支持	支持
能抛出什么	<code>std::exception</code> 对象或 <code>std::exception</code> 派生类的 对象；对象指针；对象引用； <code>string</code> 或 <code>int</code> 等数据类型	<code>Exception</code> 对象 或 <code>Exception</code> 派 生类的对象	<code>Exception</code> 对象或 <code>Exception</code> 派生类的对 象
未捕获的异常 所造成的影响	调用 <code>std::unexpected()</code> 函 数，该函数在默认情况下将调 用 <code>std::terminate()</code> ，而这一 函数在默认情况下又将调 用 <code>abort()</code>	如果是一个“受检 异常（ <code>checked exception</code> ）”则终 止正在执行的线 程；如果是“运 行时异常（ <code>runtime exception</code> ）”则不产 生任何影响	终止程序执 行
必须在类的接口中 定义可能会抛出的 异常	否	是	否
必须在类的接口中 定义可能会捕获的 异常	否	是	否

把异常当做正常处理逻辑的一部分的那种程序，都会遭受与所有典型的意大利面条式代码同样的可读性和可维护性问题。

—Andy Hunt 和 Dave Thomas

异常和继承有一点是相同的，即：审慎明智地使用时，它们都可以降低复杂度；而草率粗心地使用时，只会让代码变得几乎无法理解。下面给出的一些建议可以让你在使用异常时扬长避短，并避免与之相关的一些难题。

用异常通知程序的其他部分，发生了不可忽略的错误 异常机制的优越之处在于它能提供一种无法被忽略的错误通知机制（Meyers 1996）。其他的错误处理机制有可能会导致错误在不知不觉中向外扩散，而异常则消除了这种可能性。

只在真正例外的情况下才抛出异常 仅在真正例外的情况下才使用异常——换句话说，就是仅在其他编码实践方法无法解决的情况下才使用异常。异常的应用情形跟断言相似——都是用来处理那些不仅罕见甚至永远不该发生的情况。

异常需要你做出一个取舍：一方面它是一种强大的用来处理预料之外的情况的途径，另一方面程序的复杂度会因此增加。由于调用子程序的代码需要了解被调用代码中可能会抛出的异常，因此异常弱化了封装性。同时，代码的复杂度也有所增加，这与在第 5 章“软件构建中的设计”中提出的软件首要技术使命——管理复杂度——是背道而驰的。

不能用异常来推卸责任 如果某种的错误情况可以在局部处理，那就应该在局部处理掉它。不要把本来可以在局部处理掉的错误当成一个未被捕获的异常抛出去。

避免在构造函数和析构函数中抛出异常，除非你在同一地方把它们捕获 当从构造函数和析构函数里抛出异常时，处理异常的规则马上就会变得非常复杂。比如说在 C++ 里，只有在对象已完全构造之后才可能调用析构函数，也就是说，如果在构造函数的代码中抛出异常，就不会调用析构函数，从而造成潜在的资源泄漏（Meyers 1996, Stroustrup 1997）。在析构函数中抛出异常也有类似复杂的规则。

语言律师可能会认为记忆这些规则是小事一桩，但智力平凡的程序员很难记住这些规则。所以，应该养成好编程习惯，不要一上来就写这类代码，从而轻松地避免由此产生的额外的复杂度。

交叉参考 关于维护一致的接口抽象的详情，请参见第 6.2 节中的“好的抽象”。

在恰当的抽象层次抛出异常 子程序应在其接口中展现出一致的抽象，类也是如此。抛出的异常也是程序接口的一部分，和其他具体的数据类型一样。

当你决定把一个异常传给调用方时，请确保异常的抽象层次与子程序接口的抽象层次是一致的。这个例子说明了应该避免什么样的做法：



Java反例：抛出抽象层次不一致的异常的类

```
class Employee {
    ...
    public TaxId GetTaxId() throws EOFException {
        ...
    }
    ...
}
```

此处声明的异常
位于不一致的抽
象层次。

`GetTaxId()` 把更低层的 `EOFException` (文件结束, end of file) 异常返回给了它的调用方。它本身并不拥有这一异常，但却通过把更低层的异常传递给其调用方，暴露了自身的一些实现细节。这就使得子程序的调用方代码不是与 `Employee` 类的代码耦合，而是与比 `Employee` 类层次更低的抛出 `EOFException` 异常的代码耦合起来了。这样做既破坏了封装性，也减低了代码的智力上的可管理性 (intellectual manageability)。

与之相反，`GetTaxId()` 代码应抛回一个与其所在类的接口相一致的异常，就像下面这样：

Java示例：一个在一致的抽象层次上抛出异常的类

```
class Employee {
    ...
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {
        ...
    }
    ...
}
```

这里声明的异常
则位于一致的抽
象层次。

`GetTaxId()` 里的异常处理代码可能只需要把一个 `io_disk_not_ready` (磁盘 IO 未就绪) 异常映射为 `EmployeeDataNotAvailable` (雇员数据不可用) 异常就好了，因为这样一来就能充分地保持接口的抽象性。

在异常消息中加入关于导致异常发生的全部信息 每个异常都是发生在代码抛出异常时所遇到的特殊情况下。这一信息对于读取异常消息的人们来说是很有价值的，因此要确保该消息中含有为理解异常抛出原因所需的信息。如果异常是

因为一个数组下标错误而抛出的，就应在异常消息中包含数组的上界、下界以及非法的下标值等信息。

避免使用空的 catch 语句 有时你可能会试图敷衍一个不知该如何处理的异常，比如这个例子：



Java示例：忽略异常的错误做法

```
try {
    ...
    // 很多代码
    ...
} catch ( AnException exception ) {
}
```

这种做法就意味着：要么是 try 里的代码不对，因为它无故抛出了一个异常；要么是 catch 里的代码不对，因为它没能处理一个有效的异常。确定一下错误产生的根源，然后修改 try 或 catch 二者其一的代码。

偶尔你也可能会遇到某个较低层次上的异常，它确实无法表现为调用方抽象层次上的异常。如果确实如此，至少需要写清楚为什么采用空的 catch 语句是可行的。你也可以用注释或向日志文件中记录信息来对这一情况进行“文档化”，就像下面这样：

Java示例：忽略异常的正确做法

```
try {
    ...
    // lots of code
    ...
} catch ( AnException exception ) {
    LogError( "Unexpected exception" );
}
```

了解所用函数库可能抛出的异常 如果你所用的编程语言不要求子程序或类定义它可能抛出的异常，那你一定要了解所用的函数库都会抛出哪些异常。未能捕获由函数库抛出的异常将会导致程序崩溃，就如同未能捕获由自己代码抛出的异常一样。如果函数库没有说明它可能抛出哪些异常，可以通过编写一些原型代码来演练该函数库，找出可能发生的异常。

考虑创建一个集中的异常报告机制 有种方法可以确保异常处理的一致性，即创建一个集中的异常报告机制。这个集中报告机制能够为一些与异常有关的信息提供一个集中的存储，如所发生的异常种类、每个异常该被如何处理以及如何格式化异常消息等。

下面就是一个简单的异常处理器，它只是简单地打印出诊断信息：

深入阅读 关于这一技术更详细的阐述，请参见《Practical Standards for Microsoft Visual Basic.NET》(Foxall 2003)。

```
Visual Basic示例：集中的异常报告机制（第一部分）
Sub ReportException( _
    ByVal className, _
    ByVal thisException As Exception _
)
    Dim message As String
    Dim caption As String

    message = _
        "Exception: " & thisException.Message & "." & ControlChars.CrLf & _
        "Class: " & className & ControlChars.CrLf & _
        "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
    caption = "Exception"
    MessageBox.Show( message, caption, MessageBoxButtons.OK, _
        MessageBoxIcon.Exclamation )

End Sub
```

你可以像这样在代码中使用这个通用的异常处理器：

```
Visual Basic示例：集中的异常报告机制（第二部分）
Try
    ...
Catch exceptionObject As Exception
    ReportException( CLASS_NAME, exceptionObject )
End Try
```

这个版本的 ReportException() 代码非常简单。而在实际的应用程序中，你可以根据异常处理的需要开发或简或繁的代码。

如果确定要创建一个集中的异常报告机制，请一定要考虑第 8.3 节中“调用错误处理子程序或对象”所讲到的和集中错误处理相关的事宜。

把项目中对异常的使用标准化 为了保持异常处理尽可能便于管理，你可以用以下几种途径把对异常的使用标准化。

- 如果你在用一种像 C++一样的语言，其中允许抛出多种多样的对象、数据及指针的话，那么就应该为到底可以抛出哪些种类的异常建立一个标准。为了与其他语言相兼容，可以考虑只抛出从 std::exception 基类派生出的对象。

- 考虑创建项目的特定异常类，它可用做项目中所有可能抛出的异常的基类。这样就能把记录日志、报告错误等操作集中起来并标准化。
- 规定在何种场合允许代码使用 throw-catch 语句在局部对错误进行处理。
- 规定在何种场合允许代码抛出不在局部进行处理的异常。
- 确定是否要使用集中的异常报告机制。
- 规定是否允许在构造函数和析构函数中使用异常。

交叉参考 关于错误处理的更多可选方案，请参阅本章前面第 8.3 节“错误处理技术”。

考虑异常的替换方案 一些编程语言对异常的支持已有 5~10 年甚至更久的历史了，但关于如何才能安全使用异常的传统与经验仍然很少。

有些程序员用异常来处理错误，只是因为他所用的编程语言提供了这种特殊的错误处理机制。你心里应该自始至终考虑各种各样的错误处理机制：在局部处理错误、使用错误码来传递错误、在日志文件中记录调试信息、关闭系统或其他的一些方式等。仅仅因为编程语言提供了异常处理机制而使用异常，是典型的“为用而用”；这也是典型的“在一种语言上编程”而非“深入一种语言去编程”的例子。（有关这两者的区别，请参阅第 4.3 节“你在技术浪潮中的位置”和第 34.4 节“以所用语言编程，但思路不受其约束”）

最后，请考虑你的程序是否真的需要处理异常。就像 Bjarne Stroustrup 所指出的，应对程序运行时发生的严重错误的最佳做法，有时就是释放所有已获得的资源并终止程序执行，而让用户去重新用正确的输入数据再次运行程序（Stroustrup 1997）。

8.5

Barricade Your Program to Contain the Damage Caused by Errors 隔离程序，使之包容由错误造成的损害

隔栏（barricade）是一种容损策略（damage-containment strategy）。这与船体外壳上装备隔离舱的原因是类似的。如果船只与冰山相撞导致船体破裂的话，隔离舱就被封闭起来，从而保证船体的其余部位不会受到影响。这也与建筑物里的防火墙很相像。在发生火灾时，建筑物里的防火墙能阻止火势从建筑物的一个部位向其他部位蔓延。（隔栏过去叫“防火墙”，但现在“防火墙”这一术语常用来指代阻止恶意网络攻击的设备。）

以防御式编程为目的而进行隔离的一种方法，是把某些接口选定为“安全”区域的边界。对穿越安全区域边界的数据进行合法性校验，并当数据非法时做

出敏锐的反映。图 8-2 展示了这一概念。

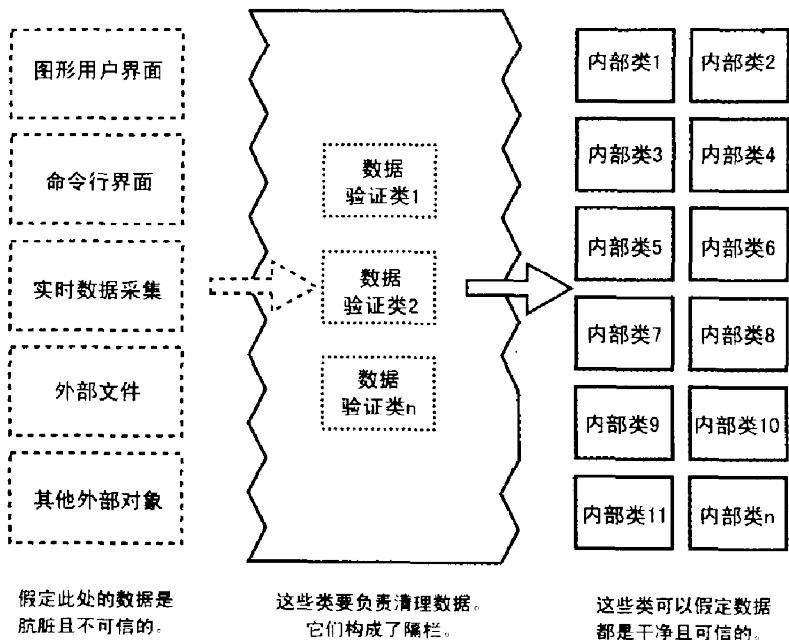


图8-2 让软件的某些部分处理“不干净的”数据，而让另一些部分处理“干净的”数据，即可让大部分代码无须再担负检查错误数据的职责

也同样可以在类的层次采用这种方法。类的公用方法可以假设数据是不安全的，它们要负责检查数据并进行清理。一旦类的公用方法接受了数据，那么类的私用方法就可以假定数据都是安全的了。

也可以把这种方法看做是手术室里使用的一种技术。任何东西在允许进入手术室之前都要经过消毒处理。因此手术室内的任何东西都可以认为是安全的。这其中最核心的设计决策是规定什么可以进入手术室，什么不可以进入，还有把手术室的门设在哪里——在编程中也就是规定，哪些子程序可认为是在安全区域里的，哪些又是在安全区域外的，哪些负责清理数据。完成这一工作最简单的方法是在得到外部数据时立即进行清理，不过数据往往需要经过一层以上的清理，因此多层清理有时也是必需的。

在输入数据时将其转换为恰当的类型 输入的数据通常都是字符串或数字的形式。这些数据有时要被映射为“是”或“否”这样的布尔类型，有时要被映射为像 Color_Red、Color_Green 和 Color_Blue 这样的枚举类型。在程序中长时间传递类型不明的数据，会增加程序的复杂度和崩溃的可能性——比如说有人在需要输入颜色枚举值的地方输入了“是”。因此，应该在输入数据后立即将其转换到恰当的类型。

Relationship Between Barricades and Assertions 隔栏与断言的关系

隔栏的使用使断言和错误处理有了清晰的区分。隔栏外部的程序应使用错误处理技术，在那里对数据做的任何假定都是不安全的。而隔栏内部的程序里就应使用断言技术，因为传进来的数据应该已在通过隔栏时被清理过了。如果隔栏内部的某个子程序检测到了错误的数据，那么这应该是程序里的错误而不是数据里的错误。

隔栏的使用还展示了“在架构层次上规定应该如何处理错误”的价值。规定隔栏内外的代码是一个架构层次上的决策。

8.6 Debugging Aids 辅助调试的代码

防御式编程的另一重要方面是使用调试助手（辅助调试的代码），调试助手非常强大，可以帮助快速地检测错误。

Don't Automatically Apply Production Constraints to the Development Version 不要自动地把产品版的限制强加于开发版之上

深入阅读 关于使用调试代码来进行防御式编程的更多内容，请参阅《Writing Solid Code》(Maguire 1993)

程序员们常常有这样一个误区，即认为产品级软件的种种限制也适用于开发中的软件。产品级的软件要求能够快速地运行，而开发中的软件则允许运行缓慢。产品级的软件要节约使用资源，而开发中的软件在使用资源时可以比较奢侈。产品级的软件不应向用户暴露可能引起危险的操作，而开发中的软件则可以提供一些额外的、没有安全网的操作。

我曾参与编写的一个程序中大量地使用了四重链表（quadruply linked list）。链表的代码是很容易出错的，链表本身的结构很容易损坏。因此我给程序加了一个菜单项来检测链表的完整性。

在调试模式下，Microsoft Word 在空闲循环中加入了一些代码，它们每隔几秒钟就检查一次 Document 对象的完整性。这样既有助于快速检测到数据的损坏，也方便了对错误的诊断。

应该在开发期间牺牲一些速度和对资源的使用，来换取一些可以让开发更顺畅的内置工具。



KEY POINT

Introduce Debugging Aids Early 尽早引入辅助调试的代码

你越早引入辅助调试的代码，它能够提供的帮助也越大。通常，除非被某个错误反复地纠缠，否则你是不愿意花精力去编写一些调试辅助的代码的。然而，如果你一遇到问题马上就编写或使用前一个项目中用过的某个调试助手的话，它就会自始至终在整个项目中帮助你。

Use Offensive Programming 采用进攻式编程

交叉参考 关于处理异常情况的更多细节，请参考第 15.2 节中的“使用 case 语句的诀窍”。

通常情况下死程序所造成的损失要比残废的程序少得多。
—Andy Hunt 和 Dave Thomas

应该以这么一种方式来处理异常情况：在开发阶段让它显现出来，而在产品代码运行时让它能够自我恢复。Michael Howard 和 David LeBlanc 把这种方式称为“进攻式编程（offensive programming）”（Howard and LeBlanc 2003）。

假设你有一段 case 语句，期望用它处理 5 类事件。在开发期间，应该让针对默认情况的 case 分支（即 default case 子句）显示警告信息说：“嗨！这儿还有一种没有处理的情况！改程序吧！”然而，在最终的产品代码里，针对默认情况的处理则应更稳妥一些，比如说可以在错误日志文件中记录该消息。

下面列出一些可以让你进行进攻式编程的方法。

- 确保断言语句使程序终止运行。不要让程序员养成坏习惯，一碰到已知问题就按回车键把它跳过。让问题引起的麻烦越大越好，这样它才能被修复。
- 完全填充分配到的所有内存，这样可以让你检测到内存分配错误。
- 完全填充已分配到的所有文件或流，这样可以让你排查出文件格式错误。
- 确保每一个 case 语句中的 default 分支或 else 分支都能产生严重错误（比如说让程序终止运行），或者至少让这些错误不会被忽视。
- 在删除一个对象之前把它填满垃圾数据。
- 让程序把它的错误日志文件用电子邮件发给你，这样你就能了解到在已发布的软件中还发生了哪些错误——如果这对于你所开发的软件适用的话。

有时候，最好的防守正是大胆进攻。在开发时惨痛地失败，能让你在发布产品后不会败得太惨。

Plan to Remove Debugging Aids 计划移除调试辅助的代码

如果你是写程序给自己用，那么把调试用的代码都留在程序里可能并无大碍。但如果是商用软件，则此举会使软件的体积变大且速度变慢，从而给程序造成巨大的性能影响。要事先做好计划，避免调试用的代码和程序代码纠缠不清。下面是一些可以采用的方法。

交叉参考 关于版本控制的详细情况，请参考第28.2节“配置管理”。

使用类似ant和make这样的版本控制工具和make工具 版本控制工具可以从同一套源码编译出不同版本的程序。在开发模式下，你可以让make工具把所有的调试代码都包含进来一起编译。而在产品模式下，又可以让make工具把那些你不希望包含在商用版本中的调试代码排除在外。

使用内置的预处理器 如果你所用的编程环境里有一个预处理器——比如C++开发环境——你可以用编译器开关来包含或排除调试用的代码。你既可以直接使用预处理器，也可以写一个能与预处理器指令同时使用的宏。下面是一个直接使用预处理器编写代码的例子：

要想在编译时包含调试用的代码，使用#define来定义DEBUG符号。要想将其排除在外，则不要定义DEBUG符号。

C++示例：直接使用预处理器来控制调试用的代码

```
#define DEBUG
...
#if defined( DEBUG )
// debugging code
...
#endif
```

这一用法可以有几种变化。比如说，除了可以直接定义DEBUG以外，你还可以给它赋一个值，然后就可以判断其数值，而不仅是去判断它是否已经定义了。这么做可以让你区分不同级别的调试代码。你可能希望让某些调试代码永远留在程序里，这时就可以用类似#if DEBUG > 0这样的语句把这些代码括起来。另一些调试代码可能只是针对一些特定的用途，你可以用类似#if DEBUG == POINTER_ERRORR这样的语句把这些代码括起来。在另外一些地方，你可能想设置调试级别，这时就可以写类似#if DEBUG > LEVEL_A这样的语句。

如果你不喜欢让#if defined()一类语句散布在代码里的各处，那么可以写一个预处理器宏来完成同样的任务。这里是一个例子：

根据是否定义DEBUG符号，可以选择是否编译此处的代码。

C++示例：使用预处理器宏来控制调试用的代码

```
#define DEBUG
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif
...
DebugCode(
    statement 1;
    statement 2;
    ...
    statement n;
);
```

和前面第一个使用预处理器的例子一样，这种方法在使用时也可以有多种变化，这使得它能够处理更复杂的情况，而不仅仅是要么包含所有调试代码、要么排除全部调试代码这么简单。

交叉参考 关于预处理器和自行编写预处理器的更多信息，请参考第 30.3 节中的“宏预处理器”。

编写你自己的预处理器 如果某种语言没有包含一个预处理器，你也可以很容易自己写一个，用于包含或排除调试代码。首先确立一套声明调试代码的规则，然后就遵循这个规则编写一个预编译器。例如，在 Java 里你可以写一个预编译器来处理//#BEGIN DEBUG 和//#END DEBUG 关键字。写一个脚本来调用该预处理器，然后再编译经过处理之后的代码。从长远看，这样做会为你节省时间，而且也可以避免“不慎编译了未经过预处理的代码”的情况。

交叉参考 关于 stub 的详情，请参阅第 22.5 节中的“为测试各个子程序构造脚手架”。

使用调试存根 (debuging stubs) 很多情况下，你可以调用一段子程序进行调试检查。在开发阶段，该子程序可能要执行若干操作之后才能把控制权交还给其调用方代码。而在产品代码里，你可以用一个存根子程序 (stub routine) 来替换这个复杂的子程序，而这段 stub 子程序要么立即把控制权交还调用方，要么是执行几项快速的操作就返回。这种方法仅会带来很小的性能损耗，并且比自己编写预处理器要快一些。把开发版本和产品版本的 stub 子程序都保留起来，以便将来可以随时在两者之间来回切换。

你可以先写一个检查传入的指针是否有效的子程序：

C++示例：一段使用调试stub的子程序

```
void DoSomething(
    SOME_TYPE *pointer;
    ...
) {
    // check parameters passed in
    CheckPointer( pointer );
    ...
}
```

这行代码将调用检查指针的子程序。

在开发阶段，CheckPointer() 子程序会对传入的指针进行全面检查。这一检测可能相当耗时，但一定要非常有效，比如说这样：

C++示例：在开发阶段检查指针的子程序

```
void CheckPointer( void *pointer ) {
    // 执行第1项检查——可能是检查它不为NULL
    // 执行第2项检查——可能是检查它的地址是合法的
    // 执行第3项检查——可能是检查它所指向的数据完好无损
    ...
    // 执行第n项检查——...
}
```

这个子程序检查任何传入的指针。在开发阶段，可用它来执行你能想到的任意多项的检查。

当代码准备妥当，即将要编译为产品时，你可能不希望这项指针检查影响性能。这时你就可以用这下面个子程序来代替前面的那段代码：

这个子程序仅是
立即返回调用。

C++示例：在产品代码中检查指针的子程序
►void CheckPointer(void *pointer) {
 // no code; just return to caller
}

就计划移除调试代码而言，这里列出的方法还算不上完整，但它们应该已经为你提供足够多的想法，并让你了解到该如何因地制宜地使用这些方法了。

8.7

Determining How Much Defensive Programming to Leave in Production Code 确定在产品代码中该保留多少防御式代码

防御式编程中存在这么一种矛盾的观念，即在开发阶段你希望错误能引人注意——你宁愿看它的脸色，也不想冒险地去忽视它。但在产品发布阶段，你却想让错误能尽可能地偃旗息鼓，让程序能十分稳妥地恢复或停止。下面就给出一些指导建议，帮助你来决定哪些防御式编程工具可以留在产品代码里，而哪些应该排除在外。

保留那些检查重要错误的代码 你需要确定程序的哪些部分可以承担未检测出错误而造成的后果，而哪些部分不能承担。比如说你在开发一个电子表格程序，如果是在屏幕刷新部分的代码中存在未检测出的错误，你可能可以忍受，因为错误造成的主要后果无非是把屏幕搞乱。但如果是在计算引擎部分的代码中存在问题的话，你就无法接受了，因为这种错误可以导致用户的电子表格中出现难以察觉的错误结果。对于大多数用户来说，他们宁愿忍受屏幕乱作一团，也不愿意因为算错税额而被国税局审计。

去掉检查细微错误的代码 如果一个错误带来的影响确实微乎其微的话，可以把检查它的代码去掉。在前面的例子中，你可以把检查电子表格屏幕刷新的代码去掉。这里的“去掉”并不是指永久地删除代码，而是指利用版本控制、预编译器开关或其他技术手段来编译不包含这段特定代码的程序。如果程序所占的空间不是问题，你也可以把错误检查代码保留下，但应该让它不动声色地把错误信息记录在日志文件里。

去掉可以导致程序硬性崩溃的代码 正如我所说的，当你的程序在开发阶段检测到了错误，你肯定想让它尽可能地引人注意，以便能修复它。实现这一目的最好方法通常就是让程序在检测到错误后打印出一份调试信息，然后崩溃退出。这种方法甚至对于细微的错误也很有用。

然而在生成产品的时候，软件的用户需要在程序崩溃之前有机会保存他们的工作成果，为了让程序给他们留出足够的保存时间，用户甚至会忍受程序表现出的一些怪异行为。相反，如果程序中的一些代码导致了用户工作成果的丢失，那么无论这些代码对帮助调试程序并最终改善程序质量有多大的贡献，用户也不会心存感激的。因此，如果你的程序里存在着可能导致数据丢失的调试代码，一定要把它们从最终软件产品中去掉。

保留可以让程序稳妥地崩溃的代码 如果你的程序里有能够检测出潜在严重错误的调试代码，那么应该保留那些能让程序稳妥地崩溃的代码。以火星探路者号（Mars Pathfinder）为例，它的工程师有意地在其中保留了一些调试代码。在火星探路者号着陆之后便发生了一个故障。喷气推进实验室（JPL）的工程师们得以利用保留下来的辅助调试的代码诊断出问题所在，并把修复后的代码上传给火星探路者号，从而使得火星探路者号圆满地完成了任务（March 1999）。

为你的技术支持人员记录错误信息 可以考虑在产品代码中保留辅助调试用的代码，但要改变它们的工作方式，以便与最终产品软件相适应。如果你开发时在代码里大量地使用了断言来中止程序的执行，那么在发布产品时你可以考虑把断言子程序改为向日志文件中记录信息，而不是彻底去掉这些代码。

确认留在代码中的错误消息是友好的 如果你在程序中留下了内部错误消息，请确认这些消息的用语对用户而言是友好的。有一次，一个使用我早先编写的程序的用户打电话跟我说，她得到了这样一条消息：“出现指针分配错误，Dog Breath!” 幸亏她还算有幽默感。有一种常用而且有效的方法，就是通知用户说发生了“内部错误”，再留下可供报告该错误的电子邮件地址或电话号码即可。

8.8 Being Defensive About Defensive Programming 对防御式编程采取防御的姿态

什么东西太多了都不是好事——威士忌酒除外。
—Mark Twain
(马克·吐温)

过度的防御式编程也会引起问题。如果你在每一个能想到的地方用每一种能想到的方法检查从参数传入的数据，那么你的程序将会变得臃肿而缓慢。更糟糕的是，防御式编程引入的额外代码增加了软件的复杂度。防御式编程引入的代码也并非不会有缺陷，和其他代码一样，你同样能轻而易举地在防御式编程添加的代码中找到错误——尤其是当你随手编写这些代码时更是如此。因此，要考虑好什么地方你需要进行防御，然后因地制宜地调整你进行防御式编程的优先级。

cc2e.com/0868

CHECKLIST: Defensive Programming

核对表：防御式编程

一般事宜

- 子程序是否保护自己免遭有害输入数据的破坏？
- 你用断言来说明编程假定吗？其中包括了前条件和后条件吗？
- 断言是否只是用来说明从不应该发生的情况？
- 你是否在架构或高层设计中规定了一组特定的错误处理技术？
- 你是否在架构或高层设计中规定了是让错误处理更倾向于健壮性还是正确性？
- 你是否建立了隔栏来遏制错误可能造成的破坏？是否减少了其他需要关注错误处理的代码的数量？
- 代码中用到辅助调试的代码了吗？
- 如果需要启用或禁用添加的辅助助手的话，是否无须大动干戈？
- 在防御式编程时引入的代码量是否适宜——既不过多，也不过少？
- 你在开发阶段是否采用了进攻式编程来使错误难以被忽视？

异常

- 你在项目中定义了一套标准化的异常处理方案吗？
- 是否考虑过异常之外的其他替代方案？
- 如果可能的话，是否在局部处理了错误而不是把它当成一个异常抛到外部？
- 代码中是否避免了在构造函数和析构函数中抛出异常？
- 所有的异常是否都与抛出它们的子程序处于同一抽象层次上？
- 每个异常是否都包含了关于异常发生的所有背景信息？
- 代码中是否没有使用空的 catch 语句？（或者如果使用空的 catch 语句确实很合适，那么明确说明了吗？）

安全事宜

- 检查有害输入数据的代码是否也检查了故意的缓冲区溢出、SQL 注入、HTML 注入、整数溢出以及其他恶意输入数据？
- 是否检查了所有的错误返回码？
- 是否捕获了所有的异常？
- 出错消息中是否避免出现有助于攻击者攻入系统所需的信息？

Additional Resources

更多资源

cc2e.com/0875 请参阅下列有关防御式编程的资源：

Security

安全

Howard, Michael, and David LeBlanc. 《*Writing Secure Code*》, 2d Ed. Redmond, WA: Microsoft Press, 2003. Howard 和 LeBlanc 在本书中涵盖了关于信任输入的安全隐患。这本书让人大开眼界，它展现了到底有多少种方法能够攻破一个程序——其中一些与软件构建技术相关，而更多的则与之无关。本书跨越了从需求分析、设计、编码到测试的全部内容。

Assertions

断言

Maguire, Steve. 《*Writing Solid Code*》. Redmond, WA: Microsoft Press, 1993. 书中的第 2 章十分精彩地讨论了断言的使用，并列举了一些知名 Microsoft 产品中使用断言的有趣示例。

Stroustrup, Bjarne. 《*The C++ Programming Language*》, 3d Ed. Reading, Mass.: Addison Wesley, 1997. 第 24.3.7.2 节描述了在 C++ 中实现断言的若干变化，包括断言与前条件和后条件之间的关系。

Meyer, Bertrand. 《*Object-Oriented Software Construction*》, 2d Ed. New York: Prentice Hall PTR, 1997。这本书中有关于前条件和后条件的权威论述。

Exceptions

异常

Meyer, Bertrand. 《*Object-Oriented Software Construction*》, 2d Ed. New York: Prentice Hall PTR, 1997. 本书第 12 章中有关于异常处理方法的详细讨论。

Stroustrup, Bjarne.《*The C++ Programming Language*》,3d Ed. Reading, Mass.: Addison Wesley, 1997. 书中第 14 章有关于在 C++ 中处理异常的详尽阐述。其中的 14.11 小节还针对在 C++ 中处理异常总结出了 21 项精彩的诀窍。

Meyers, Scott.《*More Effective C++: 35 New Ways to Improve Your Programs and Designs*》. Reading, Mass.: Addison Wesley, 1996. 书中第 9-15 项描述了在 C++ 中进行异常处理的若干细节问题。

Arnold, Ken, James Gosling, and David Holmes.《*The Java Programming Language*》,3d Ed. Boston, Mass.: Addison Wesley, 2000. 书中第 8 章探讨了在 Java 中进行异常处理的问题。

Bloch, Joshua.《*Effective Java Programming Language Guide*》. Boston, Mass.: Addison Wesley, 2001. 书中的第 39~47 页描述了 Java 中异常处理的各种细节问题。

Foxall, James.《*Practical Standards for Microsoft Visual Basic .NET*》. Redmond, WA: Microsoft Press, 2003. 本书第 10 章讲述了在 Visual Basic 中的异常处理。

Key Points

要点

- 最终产品代码中对错误的处理方式要比“垃圾进，垃圾出”复杂得多。
- 防御式编程技术可以让错误更容易发现、更容易修改，并减少错误对产品代码的破坏。
- 断言可以帮助人尽早发现错误，尤其是在大型系统和高可靠性的系统中，以及快速变化的代码中。
- 关于如何处理错误输入的决策是一项关键的错误处理决策，也是一项关键的高层设计决策。
- 异常提供了一种与代码正常流程角度不同的错误处理手段。如果留心使用异常，它可以成为程序员们知识工具箱中的一项有益补充，同时也应该在异常和其他错误处理手段之间进行权衡比较。
- 针对产品代码的限制并不适用于开发中的软件。你可以利用这一优势在开发中添加有助于更快地排查错误的代码。