防御性编程
代码攻防编程

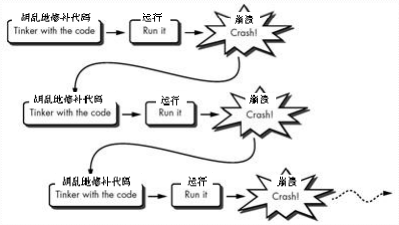
议题

- （1）防御性代码的风格
- （2）编译警告开关的使用
- （3）安全的数据结构
- （4）检查所有的返回值
- （5）审慎处理内存资源
- （6）在声明位置初始化所有变量
- （7）延迟声明变量
- （8）审慎地进行强制转换

什么是防御性编程？

- 顾名思义，防御性编程是一种细致、谨慎的编程方法。为了开发可靠的软件，我们要设计系统中的每个组件，以使其尽可能地“保护”自己。我们通过明确地在代码中对设想进行检查，击碎了未记录下来的设想。这是一种努力，防止（或至少是观察）我们的代码以将会展现错误行为的方式被调用。
- 防御性编程使我们可以尽早发现较小的问题，而不是等到它们发展成大的灾难的时候才发现。

- 你常常可以看到“职业”的开发人员不假思索飞快地编写着代码。他们开发软件的过程可能是这样的:



- 他们不断地受到那些从未有时间验证的错误的打击。这很难说是现代软件工程的进步，但它却不断地发生着。防御性编程帮助我们从一开始就编写正确的软件，而不再需要经历“编写 - 尝试 - 编写 - 尝试……”的循环过程

- 在采用了防御性编程之后，开发软件的过程将变成：

- 当然，防御性编程并不能排除所有的程序错误。但是问题所带来的麻烦将会减少，并易于修改。防御性程序员只是抓住飘落的雪花，而不是被埋葬在错误的雪崩中。
- 防御性编程是一种防卫方式，而不是一种补救形式。我们可以将其与在错误发生之后再来改正错误的调试比较一下。调试就是如何来找到补救的办法。

对防御性编程的误解

- 关于防御性编程，有一些常见的误解。防御性编程并不是：
- **检查错误**
  - 如果代码中存在可能出现错误的情况，无论如何你都应该检查这些错误。这并不是防御性编码。它只是一种好的做法，是编写正确代码的一部分。
- **测试**
  - 测试你的代码并不是防御，而只是开发工作的另一个典型部分。测试工作不是防御性的，这项工作可以验证代码现在是正确的，但不能保证代码在经历将来的修改之后不会出错。即便是拥有了世界上最好的测试工具，也还是会有人对代码进行更改，并使代码进入过去未测试的状态。
- **调试**
  - 在调试期间，你可以添加一些防御性代码，不过调试是在程序出错之后进行的。防御性编程首先是“防止”程序出错的措施（或在错误以不可理解的方式出现之前发现它们，不然就需要整夜的调试）。

支持和反对的意见

- **反对意见**
  - 防御性编程消耗了程序员和计算机的资源。
  - — 它降低了代码的效率；即使是很少的额外代码也需要一些额外的执行时间。对于一个函数或一个类，这也许还不要紧，但是如果一个系统由10万个函数组成，问题就变得严重了。
  - — 每种防御性的做法都需要一些额外的工作。为什么要做这些工作呢？你需要做的已经够多的了，不是吗？只要确保人们正确地使用你的代码就可以了。如果他们使用的方式不正确，那么任何问题也都是他们自己造成的。

- **支持意见**
  - 反驳很有说服力。
  - — 防御性编程可以节省大量的调试时间，使你可以去做更有意义的事情。还记得墨菲吗：凡是可能会被错误地使用的代码，一定会被错误地使用。
  - — 编写可以正确运行、只是速度有些慢的代码，要远远好过大多数时间都正常运行、但是有时候会崩溃的代码（显示器闪烁高亮彩色火花）。
  - — 我们可以设计一些在版本构建中物理移除的防御性代码，以解决性能问题。总之，我们这里所考虑的大部分防御性措施，并不具有任何明显的开销。
  - — 防御性编程避免了大量的安全性问题，这在现代软件开发中是一个重大的问题。避免这些问题可以带来很多好处。
  - 由于市场要求软件的开发更加快速和廉价，我们就需要致力于实现这一目标的技术。不要跳过眼前的这些额外工作，它们可以防止将来的痛苦和项目延迟。

- 有人曾说过：“千万不要归咎于那些完全可以用愚蠢来解释的恶意”。[2] 我们把大部分时间花在防备愚蠢以及错误无效的设想上。事与愿违的是，总是有一些恶意的用户，他们为了达到其设下的不良目标，试图修改和破坏你的代码。
- 防御性编程有助于程序的安全性，可以防范诸如此类恶意的滥用。黑客和病毒制造者常常会利用那些不严谨的代码，以控制某个应用程序，然后实施他们蓄意的破坏计划。这对软件开发的现代世界而言，无疑是个严重的威胁；这个问题涉及到诸如生产效率、金钱和个人隐私等方方面面。
- 软件滥用者形形色色，从利用程序小缺陷的不守规则的用户，到想尽办法非法进入他人系统的职业黑客。有太多的程序员在不经意间为这些人留下了可随意通过的后门。随着网络化计算机的兴起，粗心大意所带来的后果变得愈来愈显著了。
- 许多大型软件开发公司终于意识到了这种威胁，开始认真思考这个问题，将时间和资源投入到严谨的防御性编码工作中。事实上，在受到恶意进攻之后才亡羊补牢是很困难的。我们将在第12章更详细地讨论软件的安全性。

防御性编程技巧

- 到此为止，我们已经了解了足够多的背景知识。那么，这些背景知识对软件工厂中的程序员来说，究竟意味着什么呢？
- 在防御性编程的大框架之下，有许多常识性的规则。人们在想到防御性编程的时候，通常都会想到“断言”，这没有错。我们将在后面对此进行讨论。但是，还是有一些简单的编程习惯可以极大地提高代码的安全性。
- 尽管看上去像是常识，但是这些规则却往往被人们忽视，这就是为什么世界上并不缺少低质量软件的原因。只要程序员们警惕起来，受到足够的督促，更高的安全性和可靠的开发很容易就能够实现。
- 在下面的几页中，将列出防御性编程的一些规则。我们将先从粗略的概览开始，整体地描述防御的技巧、过程和步骤。随着讨论的深入，我们会加入更多的细节，进一步地逐条分析每条代码语句。在这些防御性技巧中，有一些是与具体的编程语言相关的。这很自然——如果你的编程语言会让你射伤到自己的脚，那么你一定要穿上防弹靴。
- 在阅读这些规则时，请对你自己进行一个评估。在这些规则中，现在你遵循的有几条？你打算采纳那些规则？

1.使用好的编码风格和合理的设计

- 我们可以通过采用良好的编程风格，来防范大多数编码错误。这与本篇的其他章节自然地吻合。很多简单的事，如选用有意义的变量名，或者审慎地使用括号，都可以使编码变得更加清晰明了，并减少缺陷出现的可能性。
- 同样地，在投入到编码工作中之前，先考虑大体的设计方案，这也非常关键。“最好的计算机程序的文本是结构清晰的。” 从实现一套清晰的API、一个逻辑系统结构以及一些定义良好的组件角色与责任开始入手，将使你避免以后处处头疼的局面。

2.不要仓促地编写代码

- 闪电式的编程太常见了。使用这种编程方式的程序员会很快地开发出一个函数，马上把这个函数交给编译器来检查语法，接着运行一遍看看能不能用，然后就进入下一个任务。这种方式充满了危险。
- 相反，在写每一行时都三思而后行。可能会出现什么样的错误？你是否已经考虑了所有可能出现的逻辑分支？放慢速度，有条不紊的编程虽然看上去很平凡，但这的确是减少缺陷的好办法。

关键概念 欲速则不达。每敲一个字，都要想清楚你要输入的是什么。

- 在C语言中，有一个会使追求速度的程序员犯错的陷阱，即将“==”错误地输入为“=”。前者为相等关系测试，而后者则是变量赋值。如果你的编译器功能不全（或者关闭了警告功能），你就不会得到相关提示，也就无从得知自己输入了不该输入的东西。
- 一定要在完成与一个代码段相关的所有任务之后，再进入下一个环节。例如，如果你决定先编写主体部分，再加入错误检查和处理，那么一定要确保这两项工作的完成都遵循章法。如果你要推迟错误检查的编写，而直接开始编写超过三个代码段的主体部分，你一定要慎之又慎。你也许真的想随后再回来编写错误检查，但却一而再再而三地向后推迟，这期间你可能会忘记很多上下文，使得接下来的工作更加耗时和琐碎。（当然，到时候你还要面临一些人为设置的最后截止日期。）
- 遵循章法是一种习惯，需要牢记于心并切实贯彻。如果你不立即做正确的事，那么将来你很可能也不会再去做正确的事。现在就行动，不要等到撒哈拉沙漠下雨了才行动。晚做不如早做，因为将来再做将需要遵循更多的章法。

3.不要相信任何人

- 妈妈曾告诉过你，不要和陌生人说话。不幸的是，要想开发一个好的软件，就需要更加愤世嫉俗，对人的天性更加不信任。即便是没有恶意的代码用户，也可能会给你的程序带来麻烦。防御意味着不能相信任何人。
- 下面这些情况可能是给你带来麻烦的原因：
  - 真正的用户 意外地提供了假的输入，或者错误地操作了程序；
  - 恶意的用户 故意造成不好的程序行为；
  - 客户端代码 使用错误的参数调用了你的函数，或者提供了不一致的输入；
  - 运行环境 没有为程序提供足够的服务；
  - 外部程序库 运行失误，不遵从你所依赖的接口协议。

- 你甚至可能会在编写一个函数时犯下愚蠢的错误，或者错误地使用三年前编写的代码，因为你忘记了这些代码究竟是怎样运行的。不要设想所有的一切都运行良好，或者所有的代码都会正确地运行。在你的程序各处都添加安全检查。时刻注意弱点，用更多的防御性代码防止弱点的出现。

关键概念 不要相信任何人毫无疑问，任何人（包括你自己）都可能把缺陷引入你的程序逻辑当中。用怀疑的眼光审视所有的输入和所有的结果，直到你能证明它们是正确的时为止。

编码的目标是清晰，而不是简洁

- 如果要你从简洁（但是有可能让人困惑）的代码和清晰（但是有可能比较冗长）的代码中选择，一定要选那些看上去和预期相符合的代码，即使它不太优雅。例如，将复杂的代数运算拆分为一系列单独的语句，使逻辑更清晰。
- 想一想，谁会是你的代码的读者。这些代码也许需要一位初级程序员来进行维护，如果他不能理解代码的逻辑，那么他肯定会犯一些错误。复杂的结构或不常用的语言技巧可以证明你在运算符优先级方面渊博的知识，但是这些实际上会扼杀代码的可维护性。请保持代码简单。
- 不能维护的代码是不安全的。举一个极端的例子，过于复杂的表达式会使编译器生成错误的代码，许多编译器优化的错误就是因此而造成的。

关键概念 简单就是一种美。不要让你的代码过于复杂。

不要让任何人做他们不该做的修补工作

- 内部的事情就应该留在内部。私人的东西就应该用锁和钥匙保管起来。不要把你的代码初稿示于众人。不管你多么礼貌地恳求，只要你稍不注意，别人就会篡改你的数据，然后自以为是地试着调用“仅用于执行”的例行程序。不要让他们这样做。
- — 在面向对象的语言中，通过将属性设为专用（private）来防止对内部类数据的访问。在C++中，可以考虑使用Cheshire cat/pimpl idiom。
- — 在过程语言中，你仍然可以使用面向对象（OO）的打包概念，将private数据打包在不透明的类型背后，并提供可以操作它们的定义良好的公共函数。
- — 将所有变量保持在尽可能小的范围内。不到万不得已，不要声明全局变量。如果变量可以声明为函数内的局部变量，就不要在文件范围上声明。如果变量可以声明为循环体内的局部变量，就不要在函数范围上声明。

- 何时进行防御性编程？你是否在事情不顺利时才开始这样做？或者在整理一些你不理解的代码时才开始？
- 不，这是不对的，你应该从始到终地使用这些防御性编程的技巧。它们应该成为你的第二天性。成熟的程序员已经从经验中得到教训，在吃过不止一遍的苦头之后，他们才明白了增加预防措施是明智的。
- 在开始编写代码时就应用防御性策略，比改进代码时才应用要容易得多。如果你很晚才试着将这些策略强加进去，就不可能做到万无一失。如果你在问题出现后才开始添加防御性代码，实际上你是在调试，被动地做出反应，而不是积极地防患于未然。
- 然而，在调试的过程中，甚至在添加新的功能时，你将发现一些你希望验证的情况。这常常是添加防御性代码的好时机。

编译时打开所有警告开关

- 大多数语言的编译器都会在你“伤了它们感情的时候”给出一大堆错误信息。当这些编译器碰到潜在的有缺陷代码时（如在赋值之前使用C或C++变量），它们也会给出各种各样的警告。通常情况下，这些警告可以有选择地启用或禁用。
- 如果你的代码中充满了危险的构造，你将会得到数页的警告信息。糟糕的是，通常的反应是禁用编译器的警告功能，或者干脆不理会这些信息。这两种做法都不可取。
- 在任何情况下都要打开你的编译器的警告功能。如果你的代码产生了任何的警告信息，立即修正代码，让编译器的报错声停下来。在启用了警告功能之后，不要对不能安静地完成编译的代码感到满意。警告的出现总是有原因的。即使你认为某个警告无关紧要，也不要置之不理。否则，总有一天这个警告会隐藏一个确实重要的警告。

关键概念 编译器的警告可以捕捉到许多愚蠢的编码错误。在任何情况下都启用它们。确保你的代码可以安安静静地完成编译。

使用静态分析工具

- 编辑器警告是对代码的一次有限的静态分析（即在程序运行之前执行的代码检查）的结果。
- 还有许多独立的静态分析工具可供使用，如用于C语言的lint（以及更多新出的衍生工具）和用于.NET汇编程序的FxCop。你的日常编程工作，应该包括使用这些工具来检查你的代码。它们会比你的编译器挑出更多的错误。

使用安全的数据结构

- 如果你做不到，那么就安全地使用危险的数据结构。
- 最常见的安全隐患大概是由缓冲溢出引起的。缓冲溢出是由于不正确地使用固定大小的数据结构而造成的。如果你的代码在没有检查一个缓冲的大小之前就写入这个缓冲，那么写入的内容总是有可能会超过缓冲的末尾的。
- 这种情况很容易出现，如下面这一小段C语言代码所示：

```
char *unsafe_copy(const char *source)
{
    char *buffer = new char[10];
    strcpy(buffer, source);
    return buffer;
}
```

- 如果source中数据的长度超过10个字符，它的副本就会超出buffer所保留内存的末尾。随后，任何事都可能会发生。数据出错是最好情况下的结果——一些其他数据结构的内容会被覆盖。而在最坏的情况下，恶意用户会利用这个简单的错误，把可执行代码加入到程序堆栈中，并使用它来任意运行他自己的程序，从而劫持了计算机。这类缺陷常常被系统黑客所利用，后果极其严重。
- 避免由于这些隐患而受到攻击其实很简单：不要编写这样的糟糕代码！使用更安全的、不允许破坏程序的数据结构——使用类似C++的string类的托管缓冲。或者
- 对不安全的数据类型系统地使用安全的操作。通过把strcpy更换为有大小限制的字符串复制操作strncpy，就可以使上面的C代码段得到保护。

```
char *safer_copy(const char *source)
{
    char *buffer = new char[10];
    strncpy(buffer, source, 10);
    return buffer;
}
```

检查所有的返回值

- 如果一个函数返回一个值，它这样做肯定是有理由的。检查这个返回值。如果返回值是一个错误代码，你就必须辨别这个代码并处理所有的错误。不要让错误悄无声息地侵入你的程序；忍受错误会导致不可预知的行为。
- 这既适用于用户自定义的函数，也适用于标准库函数。你会发现：大多数难以察觉的错误都是因为程序员没有检查返回值而出现的。不要忘记，某些函数会通过不同的机制（例如，标准C库的errno）返回错误。不论何时，都要在适当的级别上捕获和处理相应的异常。

审慎地处理内存（和其他宝贵的资源）

- 对于在执行期间所获取的任何资源，必须彻底释放。内存是这类资源最常提到的一个例子，但并不是唯一的一个。文件和线程锁也是我们必须小心使用的宝贵资源。做一个好的“管家”。
- 不要因为觉得操作系统会在你的程序退出时清除程序，就不注意关闭文件或释放内存。对于你的代码还会执行多长时间，是否会耗尽所有的文件句柄或占用所有的内存，其实你一无所知。你甚至不能肯定操作系统是否会完全释放你的资源，有的操作系统就不是这样的。
- 有一个学派说：“在确定你的程序可以运行之前，不要担心内存的释放；只有在能够确定之后再添加所有相关的释放操作。”这种观点大错特错，是一种荒谬而且危险的做法。它会使你在使用内存时出现许许多多的错误；你将不可避免地在某些地方忘记释放内存。

关键概念 重视所有稀有的资源。审慎地管理它们的获取和释放。

- Java和.NET使用垃圾回收器来执行这些繁重的清洁工作，所以你可以“忘记”释放资源。让它们进入工作状态，这样在运行时将会不时地进行清扫。这真是一种享受，不过，不要因此而对安全性抱有错误的想法。你仍然需要思考。你必须显式地终止对那些不再需要，或不会被自动清除的对象的引用；不要意外地保留对对象的引用。不太先进的垃圾回收器也很容易会被循环引用蒙蔽（例如，A引用B，B又引用A，除此之外没有对A和B的引用）。这会导致对象永远不会被清除；这是一种难以发现的内存泄漏形式。

在声明位置初始化所有变量

- 这是一个显而易见的问题。如果你初始化了每个变量，它们的用途就会是明确的。依靠像“如果我不初始化它，我就不关心初始值”的经验主义是不安全的。代码将会发展。未初始化的值以后可能随时都会变成问题。
- C和C++使这个问题更加复杂化。如果你意外地使用了一个没有初始化的变量，那么你的程序在每次运行的时候都将得到不同的结果，这取决于当时内存中的垃圾信息是什么。在一个地方声明一个变量，随后再对它进行赋值，在这之后再使用它，这样会为错误打开一个窗口。如果赋值的语句被跳过，你就会花费大量的时间来寻找程序随机出现各种行为的原因。在声明每个变量的时候就对它进行初始化，就可以把这个窗口关上，因为即使初始化时赋的值是错误的，至少出现的错误行为也是可以预知的。
- 比较安全的语言（如Java和C#）通过为所有变量定义初始值，回避了这个易犯的错误。在声明变量的时候对它进行初始化仍然是一种好的做法，这样可以提高代码的明确性。

尽可能推迟一些声明变量

- 尽可能推迟一些声明变量，可以使变量的声明位置与使用它的位置尽量接近，从而防止它干扰代码的其他部分。这样做也使得使用变量的代码更加清晰。你不再需要到处寻找变量的类型和初始化，在附近声明使这些都变得非常明显。
- 不要在多个地方重用同一个临时变量，即使每次使用都是在逻辑上相互分离的区域中进行的。变量重用会使以后对代码重新完善的工作变得异常复杂。每次都创建一个新的变量——编译器会解决任何有关效率的问题。

使用标准语言工具

- 在这方面，C和C++都是一场噩梦。它们的规范有许多不同的版本，使得许多情况成为了其他实现的未定义行为。现如今有很多种编译器，每个编译器都有一些与其他编译器稍有不同的行为。这些编译器大部分是相互兼容的，但是仍然存在大量的绳索会套住你的脖子。
- 明确地定义你正在使用的是哪个语言版本。除非你的项目要求你（最好是有一个好的理由），否则不要将命运交给编译器，或者对该语言的任何非标准的扩展。如果该语言的某个领域还没有定义，就不要依赖你所使用的特定编译器的行为（例如，不要依赖你的C编译器将char作为有符号的值对待，因为其他的编译器并不是这样的）。这样做会产生非常脆弱的代码。当你更新了编译器之后，会发生什么？一位新的程序员加入到开发团队中，如果他不理解那些扩展，会发生什么？依赖于特定编译器的个别行为，将导致以后难以发现的错误。

使用好的诊断信息日志工具

- 当你编写新的代码时，常常会加入很多诊断信息，以确定程序的运行情况。在调试结束后是否应该删除这些诊断信息呢？保留这些信息对以后再次访问代码会带来很多方便，特别是如果在此期间可以有选择地禁用这些信息。
- 有很多诊断信息日志系统可以帮助实现这种功能。这些系统中很多都可以使诊断信息在不需要的时候不带来任何开销；可以有选择地使它们不参加编译。

审慎地进行强制转换

- 大多数语言都允许你将数据从一种类型强制转换（或转换）为另一种类型。这种操作有时比其他操作更成功。如果试着将一个64位的整数转换为较小的8位数据类型，那么其他的56位会怎么样呢？你的执行环境可能会突然抛出异常，或者悄悄地使你数据的完整性降级。很多程序员并不考虑这类事情，所以他们的程序就会表现出不正常的行为。
- 如果你真的想使用强制转换，就必须对之深思熟虑。你所告诉编译器的是：“忘记类型检查吧：我知道这个变量是什么，而你并不知道。”你在类型系统中撕开了一个大洞，并直接穿越过去。这样做很不可靠。如果你犯了任何一种错误，编译器将只会静静地坐在那里小声嘀咕道：“我告诉过你的。”如果你很幸运（例如使用Java或C#），运行时可能会抛出异常以让你了解发生了错误，但这完全依赖于你要进行的是什么转换。
- C和C++对于数据类型的精度并不明确，所以对于数据类型的可互换性不要做任何假设。不要假设int和long的大小相同并且可以相互赋值，即使你在你的平台上侥幸可以这样做。代码可以在平台之间移植，但是糟糕的代码可移植性很差。

细则

- 低级别防御性代码的编写技巧有很多。这些技巧是日常编程工作的组成部分，包含在对现实世界的一种健康的怀疑当中。下面的几条细则值得考虑：
- **提供默认的行为**
- 大多数语言都提供了一条switch语句；这些语言都将碰到default case的执行情况。如果default case是错误的，在代码中将错误情况明示出来。如果一切都正常，也要在代码中明示顺利执行的情况，只有这样维护代码的程序员才会理解程序的执行情况。
- 同样地，如果你要编写一条不带else子句的if语句，停下来想一想，你是否应该处理这个逻辑上的默认情况。
- **遵从语言习惯**
- 这条简单的建议将确保你的读者可以明白你所编写的所有代码。他们做出的错误设想会更少。
- **检查数值的上下限**
- 即使是最基本的计算，也会使数值型变量上溢或下溢。对此要非常注意。语言规范或核心库提供了一些机制，用来确定各个标准类型的大小——别忘了使用这些机制。确保你了解所有可用的数值类型，以及每种类型最适合的情况。
- 检查并确保每一次运算都是可靠稳定的。例如，确保自己一定不要使用可能会造成除0错误的值。
- **正确设置常量**
- C或C++语言的程序员真的应该对常量的设置保持高度警惕，这会让日子好过很多。尽可能将所有可以设置成常量的都设为常量。这样做有两个好处：首先，常量的限制条件可以充当代码记录；其次，常量使编译器可以找到你所犯下的愚蠢错误。这样，你就可以避免修改超出上下限的数据了。

new约束

- 前面我们已经讨论了在编程时我们可能会做的一些设想。但是，我们应当如何把这些设想切实地与我们的软件联系起来，从而使它们不再成为随时会出现的问题呢？很简单，只需要编写一小段额外的代码来检查每种可能出现的情况。这段代码充当了每个设想的记录，使设想从暗处走到了明处。通过这种方式，我们就把约束编入到了程序的功能和行为当中。
- 如果约束被破坏了，我们应当对程序做些什么？遭到破坏的约束不仅仅是一个简单的容易找到和更正的运行时错误（事实上我们已经无时无刻不在做这种检查和处理），它一定是存在于程序逻辑中的一个缺陷。我们对程序可能做出的反应有下面几种：
  - 对问题视而不见，期望最终不会因此出现任何差错。
  - 做一些小的改动，使程序得以继续运行（例如，打印一份诊断报告，或者将错误记录到日志中）。
  - 直接把程序打入冷宫，不让它再运行下去（例如，以立即受控或非受控的方式中止程序）。
- 例如，当字符串指针设为零时，调用C语言的strlen函数是无效的，因为这时指针将会立即被废弃，于是后两种情况就成为了最貌似有理的候选项。也许，最恰当的是立即中止程序，因为使一个空指针失去引用，可以致使未加保护的操作系统中出现各种各样的灾难性后果。
- 在许多不同的情景中都需要运用约束：

- **前置条件**
- 这些条件是在输入一段代码之前必须保持为真的条件。如果前置条件不成立，那是因为客户端代码的缺陷所致。
- **后置条件**
- 这些条件是在编写一段代码之后必须保持为真的条件。如果后置条件不成立，那是因为提供者代码的错误所致。
- **不变条件**
- 这些条件是每当程序的执行到达一个特定点（如循环中、方法调用等等）时都保持为真的条件。如果不变条件不成立，则意味着程序逻辑存在错误。
- **断言**
- 断言是任何其他关于程序在给定位置状态的陈述。
- 如果没有语言的支持，实施上面所列的前两个条件将非常困难——如果一个函数有多个退出点,那么插入后置条件就会非常麻烦。Eiffel在核心语言中支持前置和后置条件，并且也可以确保约束校验不带有任何副作用。
- 虽然很乏味，但是代码中所表达的好的约束可以使你的程序更加清晰，也更易于维护。由于约束构成了代码段之间一个不可改变的契约，所以这一技术也称为“契约式设计”（design by contract）。

约束的内容

- 使用约束可以防止许多不同问题的发生。例如，你可以：
- — 检查所有的数组访问是否都在边界内；
- — 在废弃指针之前断言指针是非零的；
- — 确保函数的参数有效；
- — 在函数的结果返回之前对其进行充分的检查；
- — 在操作对象之前证明它的状态是一致的；
- — 警惕代码中你会写下注释“不应该执行到这里”的任何一段。
- 这些例子中的前两个在C/C++语言中尤为突出。Java和C#在其核心语言中，就如同其他语言一样有其自己规避这些隐患的机制。
- 你究竟应该进行多少约束校验呢？在每一行都设置一个校验就有些太极端了。与很多事情类似，随着程序员越来越成熟，恰当的平衡才会逐渐明朗起来。是多一些好还是少一些好呢？过多的约束校验，有可能会使代码的逻辑变得不明确。“可读性是衡量程序质量的最佳标准：如果一个程序易于阅读，那么这个程序很可能是一个好的程序；如果很难阅读，则很有可能不是好程序。”
- 实际上，在主要的函数中放置前置条件和后置条件，并且在关键的循环中放置不变条件，就已经足够了。

- **移除约束**
- 通常只有在程序构建的开发和调试阶段，才需要这种约束校验。当我们用约束使自己确信（不论正确与否）程序的逻辑是正确的之后，理论上我们就可以移除它们，从而节省很多不必要的运行时开销。
- 感谢神奇的现代技术，这一切是完全可能的。C和C++标准库提供了实施约束的公共机制——断言（assert）。断言起程序防火墙的作用，用于测试其自变量的逻辑。它是作为对开发人员的警告而提供的，向其显示不正确的程序行为；不应该允许断言在面向用户的代码中触发。如果断言的约束满足，则程序继续执行。否则，程序将中止，并产生类似如下的错误信息：

```
bugged.cpp:10:int main():Assertion "1 == 0" failed.
```

- 断言是作为预处理程序的宏实现的，这意味着它在C中比在C++中更自然。也有很多更适合C++的断言库。
- 为了使用断言，你必须包含〈assert.h〉头文件。然后，你就可以在函数中编写类似“assert(ptr != 0);”的语句。神奇的预处理程序使我们可以通过为编译器指定“NDEBUG”标志，来将断言从发行版中剥离出来。所有的断言都将被移除，其自变量也不会被计算。这意味着在发行版中，断言将没有任何开销。
- 断言是否应该被完全移除，是一个值得讨论的问题。有一个学派声称在移除断言之后，你所测试的就会是一段完全不同的代码。其他人则说断言的开销在版本构建中是不可接受的，所以必须将它们移除。（但是有多少人通过检验执行的情况来证明这一点呢？）
- 无论对哪个版本，我们的断言都不能有任何副作用。例如，如果你错误地编写了如下语句，会出现什么情况：

```
int i = pullNumberFromThinAir();
assert(i = 6);
printf("i is %d\n", i);
```

- 显然在调试版本中这个断言不会触发；它的值是6（对于C而言接近于true）。然而在版本构建中，assert行将被完全移除，而printf将产生不同的输出。这可能会成为随后生产版开发过程中的小问题的原因。防止错误检查代码中的错误是很困难的！
- 设想断言可能具有更多副作用的情况并非难事。例如，如果编写语句“assert (invariants());”，而invariants()函数具有副作用，这并不容易发现。
- 由于断言可以从发行版代码中移除，所以只有约束测试可以与断言共同使用，这一点非常重要。与内存分配失败或者文件系统问题一样，真正的错误条件测试应当在一般的代码中进行处理。你不会愿意在你的程序之外对此进行编译！情有可原的运行时错误（无论多么令人不快）应该由永远无法移除的防御性代码检测出来。
- Java语言具有一套类似的断言机制。可以通过JVM上的控件启用或禁用Java的断言机制，该机制将抛出一个异常（java.lang.AssertionError）而不是使程序立即中止。.NET在框架的Debug类中提供了断言机制。

进攻性编程？
有效的进攻就是最好的防守。
——格言

- 我们都应该成为进攻性程序员。
- 如果你发现并修正了一个错误，在修正错误的地方加入一条断言是一个良好的习惯。这样，你就可以确信你不会再重蹈覆辙了。即使没有别的作用，这样做也是对将来维护代码的人员的一个很好的警示。
- 为每个类添加一个名为“bool invariant()”的成员函数，是C++和Java编写类约束的一种常用的技术。（当然，这个函数不应该具有副作用。）现在，在每个调用该不变条件的成员函数的开头和结尾，都可以放置一个断言。（显然，构造函数的开头或者析构函数的结尾都不应该有断言。）例如，一个circle类的不变条件可能会校验“radius != 0”，这将是无效的对象状态，并且可能会导致后续的计算出错（比如除数为零的错误）。

优秀的程序员……	糟糕的程序员……
— 关心他们的代码是否健壮 — 确保每个设想都显式地体现在防御性代码中 — 希望代码对无用信息的输入有正确的行为 — 在编程的时候认真思考他们所编写的代码 — 编写可以保护自己不受其他人（或程序员自己）的愚蠢伤害的代码	— 不愿意去考虑他们的代码出错的情况 — 为集成才发布可能会出错的代码，并希望别人会找到错误 — 将关于如何使用他们代码的信息紧紧攥在手里，并随时都可能将其丢掉 — 很少思考他们正在编写的代码，从而产生不可预知的和不可靠的代码