第 10 章 通过定义来规避错误

异常处理是软件系统中最糟糕的复杂性来源之一。处理特殊情况的代码在本质上比处理正常情况的代码更难编写，并且开发人员经常在定义异常时不考虑异常的处理方式。本章讨论了异常为何不成比例地增加复杂性，然后展示了如何简化异常处理。本章主要的教训是应该减少必须处理异常的地方的数量。在一些情况下，可以修改操作的语义，以便正常行为可以处理所有情况，从而无需报告任何异常情况（这也就是本章的主题）。

10.1 为什么异常会增加复杂性

我使用“异常”一词来指代任何会改变程序中正常控制流程的不常见条件。许多编程语言都包含某种形式化的异常机制，该机制允许异常由较低层级的代码抛出并由更高层级的代码捕获。但是，即使不使用形式化的异常报告机制，异常也可能发生，例如当某个方法返回一个特殊值指示其未完成其正常行为时。所有这些形式的异常都会增加复杂性。

一段特定的代码可能会以几种不同的方式遇到异常：

- 调用方可能会提供错误的参数或配置信息。
- 调用的方法可能无法完成请求的操作。例如，I/O 操作可能失败，或者所需的资源可能不可用。
- 在分布式系统中，网络数据包可能会丢失或延迟，服务器可能无法及时响应，或者节点间可能会以意想不到的方式进行通信。
- 该代码可能会检测到缺陷（bug）、内部不一致或未准备处理的情况。

大型系统必须应对许多特殊情况，特别是在它们是分布式的或需要容错的情况下。异常处理可以占系统中所有代码的很大一部分。

异常处理代码天生就比正常情况下的代码更难写。异常中断了正常的代码流，它通常意味着有些事情没有像预期的那样工作以及操作无法按计划完成。当异常发生时，程序员可以用两种方法处理它，每种方法都很复杂。第一种方法是尽管有错误但仍然向前推进并完成正在进行的工作。例如，如果一个网络数据包丢失，它可以被重发；如果数据损坏了，也许可以从冗余副本中恢复数据。第二种方法是中止正在进行的操作，向上报告异常。但是，中止可能很复杂，因为异常可能发生在系统状态不一致的地方（某个数据结构可能已经部分初始化）；异常处理代码必须恢复一致性，例如通过撤销发生异常之前所做的任何更改。

此外，异常处理代码还可能导致更多的异常。考虑重新发送丢失的网络数据包的情况。也许该数据包实际上并没有丢失，但是只是被延迟了。在这种情况下，重新发送数据包将导致重复的数据包到达对节点；这引入了节点必须处理的新的例外条件。或者，考虑从冗余副本恢复丢失的数据的情况：如果冗余副本也丢失了怎么办？在恢复期间发生的次要异常通常比主要异常更加微妙和复杂。如果通过中止正在进行的操作来处理异常，则必须将此异常作为另一个异常报告给调用方。为了防止无休止的异常级联，开发人员最终必须找到一种在不引入更多异常的情况下处理异常的方法。

语言对异常的支持往往是冗长而笨拙的，这使得异常处理代码难以阅读。例如，考虑以下代码，该代码使用 Java 对对象序列化和反序列化的支持从文件中读取 `tweet` 的集合：

```
try (
    FileInputStream fileStream = new FileInputStream(fileName);
    BufferedInputStream bufferedStream = new
BufferedInputStream(fileStream);
    ObjectInputStream objectStream = new
ObjectInputStream(bufferedStream);
) {
    for (int i = 0; i < tweetsPerFile; i++) {
        tweets.add((Tweet) objectStream.readObject());
    }
}
catch (FileNotFoundException e) {
    ...
}
catch (ClassNotFoundException e) {
    ...
}
catch (EOFException e) {
    // Not a problem: not all tweet files have full
    // set of tweets.
}
catch (IOException e) {
    ...
}
catch (ClassCastException e) {
    ...
}
```

在没有考虑实际处理异常的代码的情况下，只是基本的 `try-catch` 样板代码就比正常情况下的操作代码所占的代码行更多。很难将异常处理代码与普通情况代码相关联：例如，每个异常的生成位置都不明显。另一种方法是将代码分解为许多不同的 `try` 块。在极端情况下，每行可能产生异常的代码都需要单独的 `try` 块。这样可以清楚地说明异常发生的位置，但是 `try` 块本身会破坏代码流，并使代码难以阅读。此外，某些异常处理代码可能最终会在多个 `try` 块中重复。

确保异常处理代码是否会真正起作用是困难的。某些异常（例如 I/O 错误）在测试环境中不易生成，因此很难测试处理它们的代码。异常在运行的系统中很少发生，因此异常处理代码很少执行。代码缺陷可能会长时间未被发现，并且当最终需要异常处理代码时，它很有可能无法正常工作（我最喜欢的一句话是：“从未执行过的代码默认是无法工作的”）。最近的一项研究发现，分布式数据密集型系统中超过 90% 的灾难性故障是由不正确的错误处理引起的 [1]。当异常处理代码失败时，很难调试该问题，因为它很少发生。

10.2 异常过多

程序员通过定义不必要的异常加剧了与异常处理有关的问题。大多数程序员被教导检测和报告错误很重要。他们通常将其解释为“检测到的错误越多越好”。这导致了一种过度防御的风格，任何看起来有点可疑的东西都会被异常拒绝，从而导致不必要的异常激增，增加了系统的复杂性。

在设计 Tcl 脚本语言时，我自己就犯了这个错误。Tcl 包含一个 `unset` 命令，可用于删除变量。我定义的 `unset` 会在变量不存在时抛出错误。当时我认为，如果有人试图删除一个不存在的变量，那么它一定是一个代码缺陷，所以 Tcl 应该报告它。然而，`unset` 最常见的用途之一是清理以前操作创建的临时状态。通常很难准确预测创建了什么状态，尤其是在操作中途被中止的场景里。因此，最简单的方法是删除所有可能已经创建的变量。`unset` 的定义使得这种情况很尴尬：开发人员最终会用 `catch` 语句捕获并忽略 `unset` 抛出的错误。回顾过去，`unset` 命令的设计是我在 Tcl 设计中犯下的最大错误之一。

使用异常来避免处理困难的情况是很诱人的：与其想出一种整洁的方法来处理它，不如抛出一个异常并将问题转移给调用者。有人可能会争辩说，这种方法可以赋予调用者权力，因为它允许每个调用者以不同的方式处理异常。然而，如果你不知道做什么去处理特殊情况，调用者也很有可能不知道该做什么。在这种情况下生成异常只会将问题传递给其他人，并增加系统的复杂性。

类抛出的异常是其接口的一部分：**具有大量异常的类具有复杂的接口，并且比具有较少异常的类浅**。异常是接口中特别复杂的元素。它可以在被捕获之前通过多个堆栈层级向上传播，因此它不仅影响方法的调用者，而且还可能影响更高级别的调用者（及其接口）。

抛出异常很容易，处理它们很困难。因此，异常的复杂性来自异常处理代码。减少由异常处理引起的复杂性破坏的最佳方法是 **减少必须处理异常的位置的数量**。本章的其余部分将讨论减少异常处理程序数量的四种技术。

10.3 通过定义来规避错误

消除异常处理复杂性的最好方法是设计好您的 API，使其没有异常要处理：这就是 **通过定义来规避错误**。这看似亵渎神灵，但在实践中非常有效。考虑上面讨论的 Tcl `unset` 命令。与其让 `unset` 在被要求删除未知变量抛出错误，不如让它简单地不做任何事情而直接返回。我应该稍微修改一下 `unset` 的定义：与其用来删除一个变量，不如用来确保一个变量不再存在。根据第一个定义，如果变量不存在，则 `unset` 不能执行其工作，因此生成异常是说得通的。使用第二个定义，对不存在的变量名调用 `unset` 是很自然的。在这种情况下，它的工作已经完成，因此可以简单地返回。不再有错误需要上报。

10.4 示例：Windows 中的文件删除

文件删除提供了如何通过定义来规避错误的另一个示例。Windows 操作系统不允许删除已在进程中打开的文件。对于开发人员和用户来说，这是一个长期存在的槽点。为了删除一个正在使用的文件，用户必须在系统中搜索以找到已打开这个文件的进程，并终止该进程。有时用户会直接放弃这么做并重新启动系统，只是为了删除文件。

Unix 操作系统更优雅地定义了文件删除。在 Unix 中，如果在删除文件时打开了文件，则 Unix 不会立即删除该文件。而是将文件标记为删除，然后删除操作就成功返回了。该文件名已从其目录中删除，因此其他进程无法再打开该旧文件，并且可以创建具有相同名称的新文件，但现有文件数据将保留。已经打开该文件的进程可以继续读取和正常写入文件。一旦所有访问进程都关闭了文件，便最终释放其数据。

Unix 删除文件的方式规避了两种不同的错误。首先，如果文件当前正在使用中，则删除操作不再返回错误而是成功返回，该文件最终也将被删除。其次，删除正在使用的文件不会使正在使用该文件的进程抛出异常。解决此问题的一种可能方法是立即删除文件并将所有已打开的文件句柄标记为禁用，其他进程对已删除文件的任何读取或写入尝试均将失败。但是，此方法将产生新的需要那些进程处理的错误。相反，Unix 允许他们继续正常访问文件，延迟文件删除通过定义规避了这个错误。

Unix 允许进程继续读取和写入已删除的文件可能看起来很奇怪，但是我从未遇到过因此引起严重问题的情况。对于开发人员和用户，Unix 删除文件的设计比 Windows 的设计要容易相处得多。

10.5 示例：Java 中的 substring 方法

作为最后一个示例，请考虑 Java 的 `String` 类及其 `substring` 方法。给定一个字符串中的两个索引，`substring` 方法返回从第一个索引给定的字符开始并以第二个索引之前的字符结束的子字符串。但是，如果两个索引中的任何一个超出字符串的范围，`substring` 方法将抛出 `IndexOutOfBoundsException`。此异常是不必要的，并且会使此方法的使用复杂化。我经常发现自己处于一个或两个索引可能不在字符串范围内的情况，并且我想提取字符串中与指定范围重叠的所有字符。不幸的是，这要求我检查每个索引并将它们向上舍入为零或向下舍入到字符串的末尾，导致本来单行的方法调用变成了 5 到 10 行代码。

如果 Java 子字符串方法自动执行此调整，则将更易于使用，因此它实现了以下 API：“返回索引大于或等于 `beginIndex` 且小于 `endIndex` 的字符串的字符（如果有）。” 这是一个简单自然的 API，它规避了 `IndexOutOfBoundsException` 异常。现在，即使一个或两个索引均为负，或者 `beginIndex` 大于 `endIndex`，该方法的行为也已明确定义。这种方法简化了方法的 API，同时增加了其功能，因此使方法更深。许多其他语言都采用了这种无错误的方式。例如，Python 对于超出范围的列表切片返回空结果。

当我主张通过定义来规避错误时，人们有时会反驳说抛出错误会捕捉到代码缺陷。如果错误都被定义规避了，那会不会导致有更多缺陷的软件出现？也许这就是 Java 开发人员决定 `substring` 方法应该抛出异常的原因。尽量抛出错误的方式可能会捕获一些代码缺陷，但也会增加复杂性，从而导致其他代码缺陷。在尽量抛出错误的方式中，开发人员必须编写额外的代码来避免或忽略错误，这增加了出现代码缺陷的可能性。或者，他们可能会忘记编写额外的代码，在这种情况下，运行时可能会抛出意外的错误。相比之下，通过定义来规避错误将简化 API，并减少必须编写的代码量。

总体而言，减少代码缺陷的最好方法是简化软件。

10.6 屏蔽异常

减少必须处理异常的地方数量的第二种技术是异常屏蔽。使用这种方法，可以在系统的较低级别上检测和处理异常情况，因此，更高级别的软件无需知道该情况。异常屏蔽在分布式系统中尤其常见。例如，在诸如 TCP 的网络传输协议中，由于各种原因（例如损坏和拥塞），可能会丢弃数据包。TCP 在其实现中通过重新发送丢失的数据包来屏蔽数据包的丢失，因此所有数据最终都将送达，并且客户端不会察觉到丢失的数据包。

NFS 网络文件系统中出现了一个更具争议性的屏蔽异常的示例。如果 NFS 文件服务器由于任何原因崩溃或无法响应，客户端将一遍又一遍地向服务器发出请求，直到问题最终得到解决。客户端上的底层文件系统代码不会向调用应用程序报告任何异常。正在执行的这个操作（及应用程序）只是挂起，直到操作可以成功完成。如果挂起持续的时间超过一小段时间，则 NFS 客户端将在用户控制台上输出 “NFS 服务器 xyzzy 无法响应仍在尝试访问” 之类的消息。

NFS 用户经常抱怨他们的应用程序在等待 NFS 服务器恢复正常运行时会被挂起。许多人建议 NFS 应该终止操作并抛出异常而不是挂起。但是，报告异常会使情况更糟而不是更好。应用程序在无法访问其文件的情况下也没什么好做的。一种可能性是应用程序重试文件操作，但这仍然会使应用程序挂起，并且在 NFS 层级中的一个位置执行重试会比在每个应用程序中的每个文件系统调用处执行重试更容易（编译器应不必为此担心！）。另一种选择是让应用程序中止并将错误返回给调用者。调用者不太可能知道该怎么做，因此他们也将中止，导致用户工作环境崩溃。用户在文件服务器关闭时仍然无法完成任何工作，并且一旦文件服务器恢复工作，他们将不得不重新启动所有应用程序。

因此，最好的替代方法是让 NFS 屏蔽错误并挂起应用程序。通过这种方法，应用程序不需要任何代码来处理服务器问题，并且一旦服务器恢复运行，它们就可以无缝恢复。如果用户厌倦了等待，他们总是可以手动中止应用程序。

异常屏蔽并非在所有情况下都有效，但是在它起作用的情况下它是一个强大的工具。它导致了更深的类，因为它减少了类的接口（用户需要注意的异常更少）并以屏蔽异常的代码形式添加了功能。异常屏蔽是下沉复杂性的一个例子。

10.7 异常聚合

减少与异常相关的复杂性的第三种技术是异常聚合。异常聚合的思想是用一个代码段处理多个异常。与其为多个单独的异常编写不同的处理程序，不如用一个处理程序在一个地方将它们全部处理。

考虑如何处理 Web 服务器中的参数缺失的情况。Web 服务器实现 URL 的集合。服务器收到传入的 URL 时，会分派到特定的服务方法来处理该 URL 并生成响应。该 URL 包含用于生成响应的各种参数。每个服务方法都将调用一个较底层的方法（将其称为 `getParameter`）以从 URL 中提取所需的参数。如果 URL 不包含所需的参数，则 `getParameter` 会抛出异常。

当参加软件设计课程的学生实现这样的服务器时，他们中的许多人将对 `getParameter` 的每个不同调用包装在单独的异常处理程序中以捕获 `NoSuchParameter` 异常，如图 10.1 所示。这导致大量的处理程序，所有这些处理程序基本上都执行相同的操作（生成错误响应）。

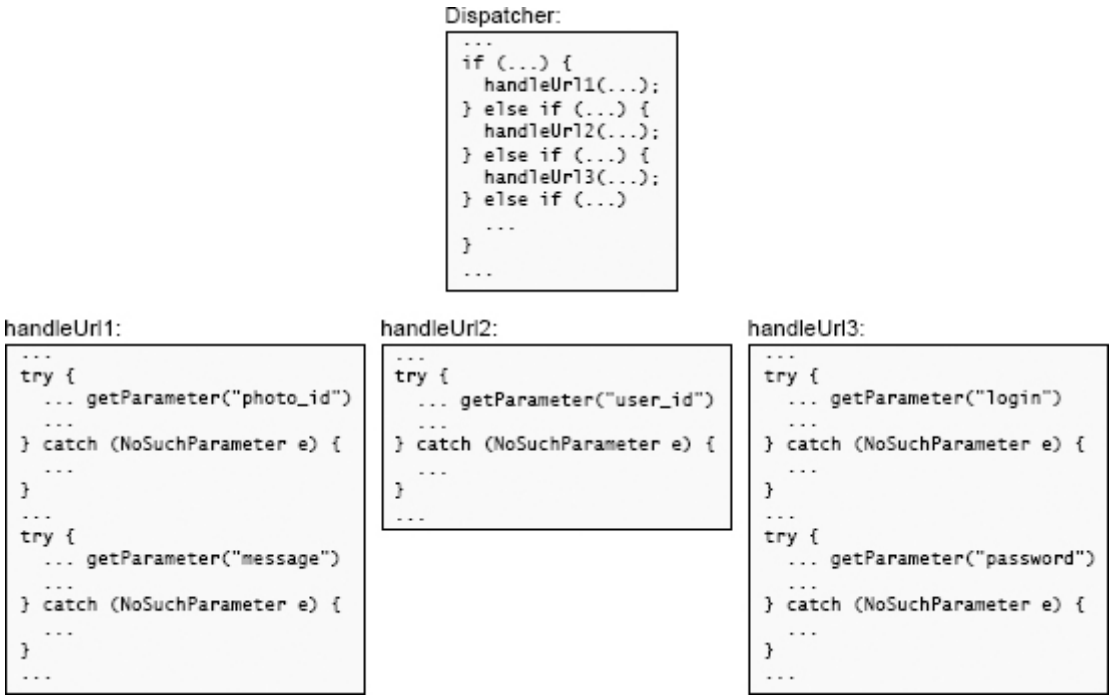

图 10.1：顶部的代码将请求分派给 Web 服务器中的几种方法之一，每种方法都处理一个特定的 URL。每个方法（底部）都使用传入 HTTP 请求中的参数。在此图中，每个对 `getParameter` 的调用都有一个单独的异常处理程序。这导致了重复的代码。

更好的方法是聚合异常。让它们传播到 Web 服务器的顶层调度方法，而不是在单个服务方法中捕获异常，如图 10.2 所示。此方法中的单个处理程序可以捕获所有异常，并为缺失的参数生成适当的错误响应。

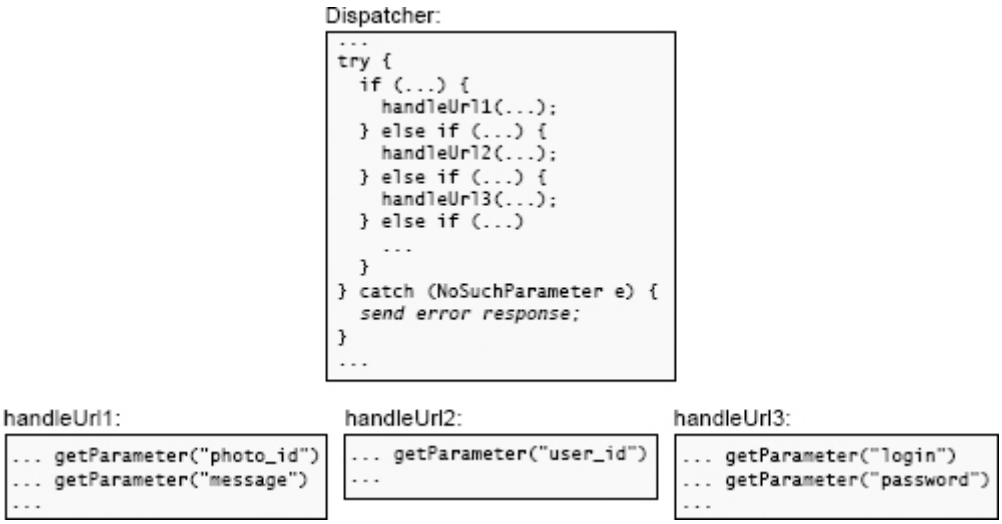

图 10.2：此代码在功能上等效于图 10.1，但是异常处理已聚合：分派器中的单个异常处理程序从所有特定于 URL 的方法中捕获所有 NoSuchParameter 异常。

聚合异常的方式可以在 Web 示例中更进一步。处理网页时，除了缺少参数外，还有许多其他错误。例如，参数可能没有正确的类型（服务方法期望的参数是整数，但值为 “xyz”），或者用户可能无权执行所请求的操作。在每种情况下，错误都应导致错误响应，仅在响应中包含的具体错误消息有所不同（“URL 中不存在参数 'quantity'”，或者，“'quantity' 参数的值 'xyz' 不正确；必须为正整数”）。因此，所有导致错误响应的条件都可以使用单个顶层的异常处理程序进行处理。错误消息可以在引发异常时生成，并作为变量包含在异常记录中。例如，`getParameter` 将生成 “URL 中不存在参数 'quantity'” 消息。顶层处理程序从异常中提取消息，并将其合并到错误响应中。

从封装和信息隐藏的角度来看，上一段中描述的异常聚合具有良好的属性。顶层的异常处理程序封装了有关如何生成错误响应的知识，但对特定错误一无所知。它仅使用异常中提供的错误消息。`getParameter` 方法封装了有关如何从 URL 提取参数的知识，并且还知道如何以人类可读的形式描述提取的错误。这两个信息密切相关，因此将它们放在同一位置是说得通的。但是，`getParameter` 对 HTTP 错误响应的语法一无所知。随着向 Web 服务器中添加了新功能，可能会创建类似 `getParameter` 有自己的异常的新方法。如果新方法抛出异常的方式和 `getParameter` 一样（继承自同一基类并且包含错误信息），现存系统不用做任何更改就可以集成新的方法：顶层的异常处理程序会自动为新方法生成相应的错误响应。

此示例说明了一种用于异常处理的通用设计模式。如果系统要处理一系列的请求，则定义一个异常以中止当前请求、清除系统状态并继续下一个请求非常有用。异常被捕获在系统请求处理循环顶部附近的单个位置。该异常可以在处理请求过程中的任何时候被抛出以将请求终止。可以在不同的条件下定义该异常的不同子类。这种类型的异常应该与对整个系统有致命影响的异常区分开来。

如果异常在被处理之前在堆栈中传播到了多个层级，则异常聚合最有效，这允许在同一个地方处理来自更多方法的更多异常。这与异常屏蔽相反：异常屏蔽通常在异常被底层代码处理的情况下效果最好。对于异常屏蔽，底层方法通常是被许多其他方法使用的库方法，因此，允许传播异常会增加需要处理该异常的位置数量。异常屏蔽和异常聚合的相似之处在于，这两种方式都将异常处理程序置于可以捕获最多异常的位置，从而消除了许多本来需要创建的异常处理程序。

异常聚合的另一个例子是 RAMCloud 存储系统的崩溃恢复。RAMCloud 系统由一组存储服务器组成，这些存储服务器保留每个对象的多个副本，因此系统可以从各种故障中恢复。例如，如果一台服务器崩溃并丢失其所有数据，RAMCloud 会使用存储在其他服务器上的副本来重建丢失的数据。错误也可能在较小的范围内发生。例如，服务器可能发现单个对象已损坏。

对于每种不同类型的错误，RAMCloud 没有单独的恢复机制。相反，RAMCloud 将许多较小的错误“升级”为较大的错误。原则上，RAMCloud 可以通过从备份副本中恢复一个损坏的对象来处理这个损坏的对象。然而，它并不这样做。相反，如果它发现一个损坏的对象，它会使包含该对象的服务器崩溃。RAMCloud 使用这种方法是因为崩溃恢复非常复杂，而且这种方法最小化了必须创建的不同恢复机制的数量。为崩溃的服务器创建恢复机制是不可避免的，因此 RAMCloud 对其他类型的恢复也使用相同的机制。这减少了必须编

写的代码量，而且这还意味着服务器崩溃恢复将更频繁地被调用。因此，恢复机制中的代码缺陷更有可能被发现和修复。

将损坏的对象升级为服务器崩溃的一个缺点是它大大增加了恢复成本。这在 RAMCloud 中不是问题，因为对象损坏非常罕见。但是，错误升级对于经常发生的错误可能没有意义。举一个例子，在服务器的任何网络数据包丢失时使服务器崩溃是不切实际的。

考虑异常聚合的一种方式是，它用可以处理多种情况的单个通用机制替换了几种针对特定情况而量身定制的专用机制。这再次说明了通用机制的好处。

10.8 让程序崩溃?

减少与异常处理相关的复杂性的第四种技术是使应用程序崩溃。在大多数应用程序中，有些错误是不值得去处理的。通常，这些错误很难或不可能处理，而且很少发生。针对这些错误的最简单的操作是打印诊断信息，然后中止应用程序。

一个示例是在存储分配期间发生的“内存不足”错误。考虑一下 C 语言中的 `malloc` 函数，如果它无法分配所需的内存块，则该函数将返回 `NULL`。这是一个不合适的行为，因为它假定 `malloc` 的每个调用者都将检查返回值并在内存不足的情况下采取适当的措施。应用程序包含许多对 `malloc` 的调用，因此在每次调用后都检查结果将增加相当大的复杂性。如果程序员忘记了检查（这很有可能），那么当内存用完时应用程序将解引用空指针并导致崩溃，从而掩盖了实际问题。

此外，当应用程序发现内存已用完时，它也没什么好做的了。原则上，应用程序可以寻找不需要的内存以释放它，但是，如果应用程序有不需要的内存，它可能已经释放了它，这将首先防止内存不足的错误。当今的系统具有如此大的内存，以至于内存几乎永远不会耗尽。如果是这样，通常表明应用程序中存在代码缺陷。因此，尝试处理内存不足错误几乎没有道理。这会带来太多的复杂性，而带来的收益却太少。

更好的方式是定义一个新的 `ckalloc` 方法，该方法调用 `malloc`、检查结果、并在内存耗尽时中止应用程序和输出错误消息。应用程序从不直接调用 `malloc`，它总是调用 `ckalloc`。

在较新的语言（例如 C++ 和 Java）中，如果内存耗尽，则 `new` 运算符将引发异常。捕获此异常没有什么意义，因为异常处理程序很有可能还会尝试分配内存，这也会失败。动态分配的内存是任何现代应用程序中的基本元素，如果内存耗尽，则继续应用程序是没有意义的。最好在检测到错误后立即崩溃。

还有许多其他错误的示例，当这些错误出现时使应用程序崩溃是说得通的。对于大多数程序，如果在读取或写入打开的文件时发生 I/O 错误（例如磁盘硬错误），或者无法打开网络套接字，则应用程序没有什么办法从在错误中恢复，因此中止程序并输出清晰的错误信息是明智之举。这些错误很少发生，因此它们不太可能影响应用程序的整体可用性。如果应用程序遇到内部错误（如数据结构不一致），中止程序并输出清晰的错误信息也是合适的。这样的情况可能表明程序中存在代码缺陷。

当特定错误出现时是否可以接受应用程序崩溃取决于具体的应用程序。对于复制存储系统，因 I/O 错误而中止是不适合的，相反，系统必须使用复制的数据来恢复丢失的任何信息。恢复机制将给程序增加相当大的复杂性，但是恢复丢失的数据是该系统为用户提供的价值的重要组成部分。

10.9 做过头了

通过定义来规避错误或将其屏蔽在模块内部，仅在模块外部不需要该异常信息时才有意义。对于本章中的示例，例如 Tcl 的 `unset` 命令和 Java 的 `substring` 方法，都是如此。在极少数情况下，调用者关心异常检测到的特殊情况，还有其他方法可以获取此信息。

但是，有时候会做得过头。在用于网络通信的模块中，一个学生团队屏蔽了所有网络异常：如果发生网络错误，则模块将其捕获、丢弃并继续进行，就好像问题没发生一样。这意味着使用该模块的应用程序无法确定消息是否丢失或节点服务器是否发生故障，而没有这些信息，就不可能构建健壮的应用程序。在这种情况下，模块必须暴露异常，即使它们增加了模块接口的复杂性。

异常与软件设计中的许多其他领域一样，您必须确定哪些是重要的，以及哪些是不重要的。不重要的事物应该被隐藏起来，它们越多越好。但是，当某件事很重要时，必须将其暴露出来（第 21 章将更详细地讨论这个主题）。

10.10 结论

任何形式的特殊情况都使代码更难以理解，并增加了发生代码缺陷的可能性。本章重点讨论异常，异常是特殊情况代码的最重要来源之一，并讨论了如何减少必须处理异常的地方的数量。做到这一点的最佳方法是重新定义语义以消除错误条件。对于无法通过定义规避的异常，您应该寻找机会将它们在底层屏蔽，以使其影响有限，或者将多个特殊情况的处理程序聚合到一个更通用的处理程序中。总之，这些技术会对整个系统的复杂性产生重大影响。

[1] 丁元 等人，“简单的测试可以防止最关键的故障：对分布式数据密集型系统中的生产故障的分析”，2014 USENIX 操作系统设计和实施大会。