第 4 章 模块应该是深的

管理软件复杂性最重要的技术之一就是将系统设计成开发人员在任何给定时间只需要面对整体复杂性的一小部分。这种方法称为 *模块化设计*，本章介绍其基本原则。

4.1 模块化设计

在模块化设计中，软件系统被分解为相对独立的 *模块* 集合。模块可以采用多种形式，例如类、子系统或服务。在理想的世界中，每个模块都将完全独立于其他模块：开发人员可以在任何模块中工作，而无需了解任何其他模块。在这个世界里，系统的复杂性就是其中最糟糕的模块的复杂性。

不幸的是，这种理想是无法实现的。模块必须通过调用彼此的函数或方法来协同工作。结果，模块必须相互了解。模块之间将存在 *依赖关系*：如果一个模块发生更改，则可能需要更改其他模块以进行匹配。例如，方法的参数在方法本身与调用该方法的任何代码之间创建了依赖关系。如果更改了要求的参数，则必须修改该方法的所有调用以符合新的签名。依赖关系可以采用许多其他形式，并且它们可能非常微妙。比如，除非先调用一个方法，否则另外一个方法就不会正常工作。模块化设计的目标是最大程度地减少模块之间的依赖性。

为了识别和管理依赖关系，我们将每个模块分为两个部分：*接口* 和 *实现*。接口包含了开发人员在使用这个模块时必须知道的所有内容。通常，接口描述模块做什么，而不描述模块如何做；而实现则包含了接口如何做的代码。在特定模块中工作的开发人员必须了解该模块的接口和实现，以及由该模块调用的任何其他模块的接口。除了该模块以外，开发人员应该无需了解其他模块的实现。

考虑一个实现平衡树的模块。该模块可能包含复杂的代码，以确保树保持平衡。但是，此复杂性对于模块用户而言是不可见的。用户可以看到一个相对简单的接口，用于调用在树中插入、删除和获取节点的操作。要调用插入操作，调用者只需提供新节点的键和值即可，而遍历树和拆分节点的机制在接口中不可见。

就本书而言，模块是具有接口和实现的任何代码单元。面向对象编程语言中的每个类都是一个模块。类中的方法或非面向对象语言中的函数也可以视为模块：每个模块都有一个接口和一个实现，并且可以将模块化设计技术应用于它们。更高层级的子系统和服务也是模块。它们的接口可能采用不同的形式，例如内核调用或 HTTP 请求。本书中有关模块化设计的许多讨论都集中在类的设计上，但是这些技术和概念也适用于其他种类的模块。

最好的模块通常其接口比其实现简单得多。这样的模块具有两个优点。首先，简单的接口可以将模块强加于系统其余部分的复杂性降至最低。其次，如果以不更改其接口的方式修改了一个模块，则该修改不会影响其他模块。如果模块的接口比其实现简单得多，则可以在不影响其他模块的情况下更改模块的许多方面。

4.2 接口中有什么?

模块的接口包含两种信息：形式化信息和非形式化信息。接口的形式化部分在代码中明确指定，并且其中一些可以通过编程语言检查其正确性。例如，方法的形式化接口是其签名，其中包括参数的名称和类型、返回值的类型以及有关该方法引发的异常的信息。大多数编程语言都确保对方法的每次调用都提供了正确数量和类型的参数以匹配其签名。类的形式化接口包括其所有公有方法的签名以及任何公有变量的名称和类型。

每个接口还包括了非形式化的元素。这些元素无法以编程语言可以理解或执行的方式进行指定。接口的非正形式化部分包括其高层级的行为，例如函数可能被设计为会删除由其参数之一所命名的文件。如果对类的使用存在限制（也许必须先调用一个方法才能调用另一个），则这些约束也是类接口的一部分。通常，如果开发人员需要了解特定信息才能使用模块，则该信息是模块接口的一部分。接口的非形式化方面只能使用注释来描述，而编程语言并不能确保描述是完整或准确的 [1]。对于大多数接口，非形式化的部分要比形式化的部分更大和更复杂。

明确指定接口的好处之一是，它可以准确指示开发人员使用关联模块所需要知道的内容。这有助于消除第 2.2 节中描述的“未知的未知”问题。

4.3 抽象

抽象 这个术语与模块化设计的思想紧密相关。**抽象是实体的简化视图，其中省略了不重要的细节。** 抽象是有用的，因为它们使我们更容易思考和操纵复杂的事物。

在模块化编程中，每个模块以其接口的形式提供抽象。该接口提供了模块功能的简化视图；从模块抽象的角度来看，实现的细节并不重要，因此在接口中将其省略。

在抽象的定义中，“不重要”一词至关重要。从抽象中忽略的不重要的细节越多越好。但是，只能在细节确实不重要的情况下才可以将其从抽象中省略。抽象可能通过两种方式出错。首先，它可能包含了并非真正重要的细节。当这种情况发生时，它会使抽象变得不必要的复杂，从而增加了使用抽象的开发人员的认知负荷。第二个错误是抽象忽略了真正重要的细节。这导致了模糊性：仅查看抽象的开发人员将不会获得正确使用抽象所需的全部信息。忽略重要细节的抽象是 *错误的抽象*：它可能看起来很简单，但实际上并非如此。设计抽象的关键就是要识别什么是重要的，并在设计过程中将重要的信息最小化。

例如，考虑一个文件系统。文件系统提供的抽象省略了许多细节，例如用于选择存储设备上的哪些块用于存储给定文件中的数据的机制。这些详细信息对于文件系统的用户而言并不重要（只要系统提供足够的性能即可）。但是，文件系统实现的一些细节对用户很重要。大多数文件系统将数据缓存在主内存中，并且它们可能会延迟将新数据写入存储设备以提高性能。一些应用程序（例如数据库）需要确切地知道何时将数据写入存储设备，以便它们可以确保在系统崩溃后数据仍将保留。因此，将数据刷新到辅助存储的规则必须在文件系统的接口中可见。

我们不仅依靠抽象来管理编程中的复杂性，抽象在日常生活中也无处不在。微波炉包含复杂的电子设备，可将交流电转换为微波辐射并将该辐射分布到整个烹饪腔中。幸运的是，用户看到的是一个简单得多的抽象，它由几个按钮控制微波的定时和强度。汽车提供了一种简单的抽象概念，使我们可以在不了解电动机、电池电源管理、防抱死制动、巡航控制

等机制的情况下驾驶它们。

4.4 深模块

最好的模块是那些提供强大功能但具有简单接口的模块。我用“深”一词来描述这样的模块。为了形象化深度的概念，假设每个模块都由一个矩形表示，如图 4.1 所示。每个矩形的面积与模块实现的功能成比例。矩形的顶部边缘代表模块的接口；边缘的长度表示接口的复杂性。最好的模块很深：它们在简单的接口后隐藏了许多功能。深模块是一个很好的抽象，因为其内部复杂性的很小一部分对其用户可见。

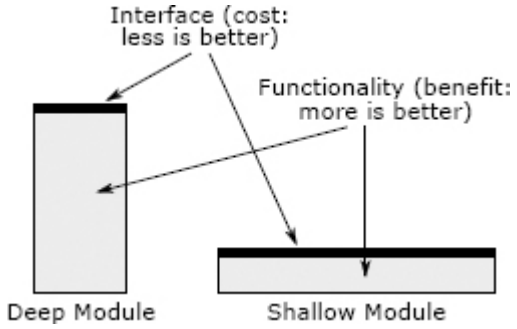

图 4.1：深浅模块。最好的模块很深：它们允许通过简单的接口访问许多功能。浅层模块是具有相对复杂的接口的模块，但功能不多：它不会掩盖太多的复杂性。

模块的深度是一种考虑成本与收益的方式。模块提供的好处是其功能。模块的成本（就系统复杂性而言）是其接口。模块的接口代表了模块强加给系统其余部分的复杂性：接口越小越简单，引入的复杂性就越小。最好的模块是那些收益最大、成本最低的模块。接口是个好东西，但更多或更大的接口不一定更好！

Unix 操作系统及其后代（例如 Linux）提供的文件 I/O 机制是深层接口的一个很好的例子。I/O 只有五个基本系统调用，带有简单签名：

```
int open(const char* path, int flags, mode_t permissions);
ssize_t read(int fd, void* buffer, size_t count);
ssize_t write(int fd, const void* buffer, size_t count);
off_t lseek(int fd, off_t offset, int referencePosition);
int close(int fd);
```

其中的 `open` 系统调用采用层次化的文件名，例如 `/a/b/c`，并返回一个整型的 *文件描述符*，该描述符用于引用打开的文件。`open` 的其他参数提供可选信息，例如打开文件后是否进行读取或写入、如果不存在现有文件则是否应创建新文件，以及如果创建新文件则文件的访问权限。`read` 和 `write` 系统调用在应用程序内存和文件的缓冲区之间传输信息。`close` 结束对文件的访问。大多数文件是按顺序访问的，因此这是默认设置。但是，可以通过 `lseek` 系统调用来更改当前访问位置以实现随机访问。

Unix I/O 接口的现代实现需要成千上万行代码，这些代码可以解决诸如以下的复杂问题：

- 如何在磁盘上表示文件以支持高效率的访问？
- 如何存储目录，以及如何处理层次化的路径名以查找它们所引用的文件？
- 如何进行权限管控，以使一个用户无法修改或删除另一用户的文件？

- 如何实现文件访问？例如，如何在中断处理程序和后台代码之间划分功能，以及这两个元素如何安全通信？
- 在同时访问多个文件时使用什么调度策略？
- 如何将最近访问的文件数据缓存在内存中以减少磁盘访问次数？
- 如何将各种不同的辅助存储设备（例如磁盘和闪存驱动器）合并到单个文件系统中？

所有这些问题，以及更多的问题，都被 Unix 文件系统的实现解决了。对于使用这些系统调用的程序员来说，它们是不可见的。多年来，Unix I/O 接口的实现已经发生了根本性的发展，但是五个基本的内核调用并没有改变。

深模块的另一个示例是诸如 Go 或 Java 之类的语言中的垃圾收集器。这个模块根本没有接口。它在后台进行隐形操作以回收未使用的内存。由于垃圾收集消除了用于释放对象的接口，因此向系统中添加垃圾回收实际上会缩小其总体接口。垃圾收集器的实现非常复杂，但这种复杂性对程序员是隐藏的。

诸如 Unix I/O 和垃圾收集器之类的深模块提供了强大的抽象，因为它们易于使用，隐藏了巨大的实现复杂性。

4.5 浅模块

另一方面，浅模块是其接口与其提供的功能相比相对复杂的模块。例如，实现链表的类很浅。操作链表不需要太多代码（插入或删除元素仅需几行代码），因此链表抽象不会隐藏很多细节。链表接口的复杂性几乎与其实现的复杂性一样高。类似于链表的浅类有时是不可避免的，它们也是有用的，但是它们在管理复杂性方面没有提供太多帮助。

这是一个浅方法的极端示例，该浅层方法来自软件设计的课程项目：

```
private void addNullValueForAttribute(String attribute) {
    data.put(attribute, null);
}
```

从管理复杂性的角度来看，此方法会使情况变得更糟，而不是更好。该方法不提供任何抽象，因为其所有功能都可以通过其接口看到。例如，调用者可能需要知道该属性将存储在 `data` 变量中。考虑接口并不比考虑完整实现简单。如果正确地文档化了这个方法，则文档将比该方法的代码长。与调用方直接操作数据变量相比，调用该方法所花费的键盘敲击数量甚至更多。该方法增加了复杂性（以供开发人员学习的新接口的形式），但没有提供任何补偿收益。

 危险信号：浅模块 

> 浅模块是一个接口相对于其提供的功能而言较为复杂的模块。浅模块在对抗复杂性方面无济于事，因为它们提供的好处（不必了解它们在内部如何工作）被学习和使用其接口的成本所抵消。小模块往往很浅。

4.6 多类症

不幸的是，深类的价值在今天并未得到广泛认可。编程中的传统观点是，类应该 *小* 而不是深。学生们经常被教导说，类的设计中最重要的事情是将较大的类分成较小的类。对于方法，通常会给出相同的建议："任何长于 N 行的方法都应分为多种方法"（N 可以低至 10）。这种方法导致了大量的浅类和方法，这增加了整体的系统复杂性。

"类应该小"的极端做法是我称之为 *多类症* 的综合症，这是由于错误地认为"类是好的，所以类越多越好"所导致的。在遭受多类症的系统中，鼓励开发人员最小化每个新类的功能：如果您想要更多的功能，请引入更多的类。多类症可能导致每个类自身都很简单，但是却增加了整个系统的复杂性。小类不会贡献太多功能，因此必须有很多小类，但每个小类都有自己的接口。这些接口的累积会在系统层级产生巨大的复杂性。由于每个类都需要样板代码，小类也容易导致冗长的编程风格。

4.7 示例：Java 和 Unix I/O

如今，最常见的多类症案例之一是 Java 类库。Java 语言本身并不意味着大量的小类，但是多类症的文化似乎已在 Java 编程社区中扎根。例如，多年以来，Java 程序员要打开文件以便从文件中读取序列化的对象，必须创建三个不同的对象：

```
FileInputStream fileStream = new FileInputStream(fileName);

BufferedInputStream bufferedStream = new
BufferedInputStream(fileStream);

ObjectInputStream objectStream = new
ObjectInputStream(bufferedStream);
```

`FileInputStream` 对象仅提供基本的 I/O：它不能执行缓冲的 I/O，也不能读取或写入序列化的对象。`BufferedInputStream` 对象将缓冲功能添加到 `FileInputStream`，而 `ObjectInputStream` 添加了读取和写入序列化对象的功能。一旦文件被打开，上面代码中的前两个对象 `fileStream` 和 `bufferedStream` 将永远不会被使用，以后的所有操作都使用 `objectStream`。

特别令人烦恼（并且容易出错）的是，必须通过创建一个单独的 `BufferedInputStream` 对象来显式请求缓冲功能。如果开发人员忘记创建该对象，将没有缓冲，并且 I/O 将变慢。也许 Java 开发人员会争辩说，并不是每个人都希望对文件 I/O 使用缓冲，因此不应将其内置到基本机制中。他们也可能会争辩说，最好单独提供缓冲能力，以便人们可以选择是否使用它。提供选择是好的，但是 **设计接口时应该使常见情况尽可能简单** （请参阅第 2.1 节的公式）。几乎每个文件 I/O 用户都希望缓冲，因此默认情况下应提供缓冲。对于不需要缓冲的少数情况，该库可以提供一种禁用它的机制。禁用缓冲的机制的任何机制都应该在接口中清晰地分离（例如，通过提供不同的 `FileInputStream` 构造函数，或者通过提供禁用或替换缓冲机制的方法），这样大多数开发人员甚至不需要知道其存在。

相反，Unix 系统调用的设计者使常见情况变得简单。例如，他们认识到顺序 I/O 是最常见的，因此他们将其作为默认行为。通过使用 `lseek` 系统调用，随机访问仍然相对容易实现，但是仅执行顺序访问的开发人员无需了解该机制。如果一个接口具有许多功能，但是大多数开发人员只需要了解其中的一些功能，那么该接口的有效复杂性就是常用功能的复杂性。

4.8 结论

通过将模块的接口与其实现分开，我们可以将实现的复杂性对系统的其余部分隐藏起来。模块的用户只需要了解模块接口提供的抽象。在设计类和其他模块时，最重要的事情是使它们足够深，以使它们具有适用于常见用例的简单接口，但仍提供重要的功能。这样就能够最大化地隐藏掉复杂性。

[1] 当前存在一些编程语言（主要是在研究社区中），可以在其中使用某种规范语言来对方法或功能的整体行为进行形式化的描述，也可以自动地检查该规范以确保它与实现相匹配。一个有趣的问题是，这样的形式化规范是否可以代替接口的非形式化部分。我目前的观点是，用英语描述的接口比使用形式化的规范语言编写的接口对开发人员来说更直观和易于理解。