

# 1

## 导论

---

如果你可以站到每一位框架使用者的肩上去编码和解释它应该如何使用，那么本书中的准则就都是多余的。作为框架作者，本书介绍的准则为你提供了一系列工具，使你能够创造出介于框架作者和框架使用者之间的通用语言。例如，将操作作为属性提供给用户，而不是公开一个方法给用户，这可以向用户传达应该如何使用该操作的信息。

在 PC 时代初期，开发应用的主要工具是语言编译器，一个很小的标准库集合和操作系统应用编程的原始接口（API），这是一套非常基础的编程工具集。

开发者使用如此基础的工具来开发应用，在这一过程中，他们发现了越来越多的重复代码，这些代码可以通过更高阶的 API 进行抽象。操作系统供应商发现，如果他们可以为开发者提供这些更高阶的 API，那么就可以让开发者以更低的成本为他们的操作系统开发应用。这样，在其操作系统上运行的应用数量就会增加，就更利于操作系统去吸引那些需要各种应用的终端用户。同时，独立的工具和组件供应商也迅速意识到提升 API 抽象级别所带来的商机。

同一时期，业界慢慢开始接受了面向对象的设计，并认识到它在可扩展性和可复用性方面的重要性<sup>1</sup>。当可复用库供应商采用面向对象编程（OOP）来开发其高级 API 时，框架的概念就随之产生了。即，应用开发者不再从无到有地开发应用，框架为其提供了

---

<sup>1</sup> 面向对象语言不是唯一可用于开发可扩展、可复用库的编程语言，但是它们在普及可复用性和可扩展性这一概念的过程中扮演了关键角色。可扩展性和可复用性是面向对象编程（OOP）哲学中的重要组成部分，对面向对象编程的采用反过来也增加了人们对其好处的认识。

大部分所需的代码，然后开发者可以对其进行自定义和连接<sup>1</sup>以形成应用。

随着越来越多的供应商开始提供组件，这些组件可以被拼接到同一个应用中实现复用，开发者发现一些组件并不能很好地组合在一起。他们的应用看起来像是一座由不同的建筑工人建起来的房子，更糟糕的是，这些建筑工人之间完全没有交流！同样地，当更大比例的应用源码被用于 API 调用，而不是被用于书写标准的语言结构时，开发者开始抱怨现在他们不得不读/写多种语言：一种编程语言和几种可复用组件所使用的“语言”。这对于开发者的生产力来说有显著的负面影响——生产力是框架成功的主要元素之一。很明显，需要制定通用的规则，以确保可复用组件的一致性和无缝集成。

现在的许多应用开发平台都规定了一些设计约定，在为这个平台设计框架时必须遵循这些约定。如果框架不遵循这些约定，那么它们就不能很好地和该平台集成在一起，也会使得那些尝试使用它们的开发者扫兴，处于竞争劣势而最终失去市场。成功的往往是那些自洽、合理并且设计精良的框架。

## 1.1 设计精良的框架的特质

那么，如何定义一个设计精良的框架，以及如何实现这个框架呢？有许多因素——如性能、可靠性、安全性、依赖管理等——影响软件质量。框架显然应当坚持相同的质量标准，但是框架与其他软件不同的是，框架是由一系列可复用的 API 组成的，这为设计高质量的框架提出了一系列特别的考量因素。

### 1.1.1 设计精良的框架是简单的

许多框架并不缺乏能力，因为随着需求变得更加清晰，添加更多新的功能是相当容易的。相反，当计划压力、功能蔓延或者渴望去满足每个微不足道的边界场景占据了开发流程时，常常会牺牲简单性。然而，简单性是每一个框架必须拥有的特性。如果觉得当前的功能设计过于复杂，最好的办法就是把该功能从当前的发布版本中移除，在下一次发布前花更多的时间去做正确的设计。正如框架设计者常说的：“你总是可以新增，但

---

<sup>1</sup> 现在有很多关于面向对象（OO）设计的批评，其声称 OO 所承诺的可复用性没有实现。面向对象设计并不是可复用性的保证，我们也不确定它是否曾承诺过。然而，面向对象设计提供了自然的结构来表达可复用单元（类型）、控制可扩展性（虚成员）并促进解耦（抽象）

是永远不能移除”。如果感觉设计不合适，你最好把它拿掉，不然你很可能会后悔为什么没有这么做。

■ **CHRIS SELLS** 为了测试一个 API 是否“简单”，我喜欢将其提交给一个被我称之为“客户端优先编程”的测试。首先，尝试描述该软件库是做什么的，然后要求开发者根据他或她对这个软件库的想法（在没有实际查看你的软件库的情况下）来编写一个程序，看开发者所写出来的和你自己实现的是否大体一致。和不同的开发者一起多做几次这个测试。如果他们中的大多数人都写出了相似的代码，并且和你写的不一致，那么他们是对的，而你是错的，你应当适当地更新该软件库。

我发现这个方法非常有用，我在设计 API 的过程中，常常先编写我所期望的客户端代码，然后实现一个软件库来匹配它。当然，你必须在简单性和试图提供的功能的固有复杂性之间取得平衡，这正是你的计算机科学专业学位的价值所在。

本书中所描述的许多准则都是试图在功能和简单性之间取得适当的平衡，特别是第 2 章，介绍了最成功的一批框架设计者所使用的一些基本方法，这些方法兼顾了简单性和功能。

### 1.1.2 设计精良的框架设计成本高昂

优秀的框架设计并不是凭空产生的，它是花费大量时间与资源，努力工作带来的结果。如果你不想在设计中投入真金白银，你就不要妄想可以创造出设计精良的框架。

■ **STEPHEN TOUB** 对于我们中那些发现自己参加的会议比理想情况下还要多的人来说，计算一次会议的成本或许是一个有趣且又令人谦卑的爱好：房间里面的人数乘以与会者平均每小时预估的工资，再乘以会议的频率。在这种情况下，我参加过的最昂贵的会议是我们的 .NET API 审阅会议，我们审阅计划中的新 API，并决定是否以及如何推进它。这些花费都是值得的，因为 API 设计中的不一致或者错误最终都会导致更大的负面开销，设计精良的、可被集成的 API 的价值“大于它们各个部分的总和”。

框架设计应该是开发过程中明确且独立的一部分<sup>1</sup>。它必须是明确的，因为它需要适当的设计，调配人手，并最终被执行。它必须是独立的，因为它不能只是实现流程的一部分；否则，常见的结果是，框架最终是由实现流程结束后，恰好保留下来的那些公开类型与成员所组成的。

最好的框架设计要么由明确负责框架设计的人来完成，要么由那些能够在开发过程中适时担任框架设计师角色的人来完成。混淆职责是错误的，会导致在设计中暴露实现细节，而这些细节对框架的最终用户来说本应该是不可见的<sup>2</sup>。

**JEREMY BARTON** 作为领域专家，要产出一个好的 API 也是非常困难的。你了解问题空间究竟有多复杂，你的提案已经把问题简化到了本质——这是简单性的巅峰。然而，事实并非如此，那些不熟悉细节的人依然会告诉你它太复杂了，并且他们很有可能是对的。

即便是那些由.NET API 审阅团队成员所提出的 API 提案，在审阅过程中也会发生变化。在 API 审阅过程中引入那些在这个功能领域不是专家的人，可以显著提高任意 API 提案的质量。

### 1.1.3 设计精良的框架充满权衡

世界上并没有完美的设计。设计就是要做出取舍，为了做出正确的决定，你需要了解有哪些选项，以及了解它们的优势和不足。如果你发现自己在设计过程中不需要做出任何取舍，那么极有可能是你忽略了某些重大的问题，而不是找到了“银弹”。

本书中描述的实践是作为指导性原则（Guideline）而不是作为规则（Rule）提出的，这正是因为框架设计需要管理权衡。其中的一些准则讨论了所涉及的权衡部分，甚至为特定的场景提供了替代方案。

- 
- 1 不要误解，以为这是对前置设计流程的认可。事实上，过度的 API 设计流程是一种浪费，因为 API 在实现后总是需要对它们进行调整。但是，API 设计流程必须独立于实现流程，并且必须被纳入产品周期的每一个部分：计划阶段（哪些 API 可以满足用户需求）、设计阶段（为了得到正确的 API，我们愿意在功能上进行哪些权衡）、开发阶段（我们有没有分配时间来试用框架，看看最终结果如何）和维护阶段（在框架的发展过程中，我们是否降低了设计质量）。
  - 2 原型是框架设计流程中最重要的部分，而且原型和实现非常不同。

#### 1.1.4 设计精良的框架会借鉴过往经验

大多数成功的框架都会借鉴现有的成熟设计并在此基础上进行构建。或许，在框架设计中引入全新的解决方案是每个人都梦寐以求的，但是那需要极其小心谨慎才有可能做到。随着新概念数量的增加，整体设计的正确性愈难保持。

■ **CHRIS SELLS** 请不要试图在软件库设计中进行“创新”，让你的软件库的 API 尽可能“乏味”吧！你需要做的是使功能（而不是 API）变得有趣吸引人。

■ **JEREMY BARTON** 全新的解决方案最好还是留给那些交叉领域的问题。这有助于你了解新解决方案的真正价值，以及让你的用户理解为什么他们需要学习“额外的事务”。

改变是一件糟糕的事情，除非改头换面。只是在某个小的方面变得好一点儿，可能并不值得你的用户花时间去学习如何使用新方法。

本书中包含的准则都是基于我们在设计 .NET 核心库的过程中获得的经验的，它们鼓励借鉴那些经得起时间考验的事物，并让我们对那些没能做到这一点的保持警惕。我们希望你能以这些优秀的实践作为开始，并进一步改进它们。本书第 9 章介绍了大量可行的通用设计模式。

#### 1.1.5 设计精良的框架旨在不断发展

如何在未来发展你的框架？这需要你思考应该做出哪些取舍。一方面，框架设计者可以在设计过程中花费更多的时间和精力，有时候，额外的复杂度甚至可以用“以防万一”来形容。另一方面，仔细考量可以避免所引入的东西随着时间的推移而退化，甚至更糟，到后面不能保持向后兼容性<sup>1</sup>。通常来说，最好将新功能推迟到下一次发布，而不是在当前发布的版本中引入它。

任何时候，当你在权衡一个设计时，都应当思考这个决定将会如何影响框架的后续发展。本书中提出的准则考虑到了这一重要问题。

---

<sup>1</sup> 在本书中，并没有详细地讨论向后兼容性，但是它和可靠性、安全性和性能一样，也应被视为框架设计的基本要素之一

### 1.1.6 设计精良的框架是完整统一的

现代框架需要能够很好地与大量不同的开发工具、编程语言、应用模型等集成在一起，云计算和其他面向服务器的工作负载模式意味着为特定应用模型做框架设计的时代已经结束了，在框架设计中无须思考合适的工具支持或者不需要与开发者社区所使用的编程语言进行适当集成的时代也已经结束了。

### 1.1.7 设计精良的框架是一致的

一致性是设计精良的框架的核心特质，是影响生产力的最重要因素。一致性的框架使得开发者可以将知识从他们已知的部分转移到他们正试图学习的部分。一致性也能帮助开发者快速认识到，对于特定功能领域来说，设计的哪些部分是真正独特的，需要特别关注，哪些又是司空见惯的设计模式和惯例。

一致性差不多是本书的核心主题了，几乎每一条准则都部分地受到一致性的影响。其中的第3~5章大概可以说是最重要的章节了，因为它们包含了涉及一致性的核心准则。我们提供了这些准则来帮助你成功构建框架。下一章介绍了一般软件库设计所涉及的准则。

## 2

# 框架设计基础

---

要设计一个成功的通用框架，在设计时就一定要考虑到有着不同需求、技能和背景的开发者。框架设计者所面临的一大挑战之一是：既要满足多样化的用户群体的功能需求，又要保持框架本身的简单性。

框架设计者的另一个重要目标是提供统一的编程模型——无论开发者编写的是哪种类型的应用<sup>1</sup>，或者，如果框架在运行时支持多种语言，则无论开发者使用的是哪种编程语言，都应该具有统一的编程模型。

通过采用已被广泛接受的设计原则和遵循本章所描述的准则，你可以创造出一个功能一致的框架，其可以满足使用不同编程语言构建不同类型应用的开发者的需求。

✓ **DO** 要设计功能强大且易于使用的框架。

设计精良的框架使得实现简单的应用场景变得容易。同时，它不会妨碍用户进一步实现更复杂的场景，尽管可能会更困难一些。正如 Alan Kay 所说：“让简单的事情保持简单，让复杂的事情成为可能。”

这一准则同样与 Pareto 原则（即“二八定律”）相关。Pareto 原则表示，在任何情况下，只有 20% 是重要的，剩下的 80% 则是微不足道的。在设计一个框架时，应该专注于重要的 20% 的场景和接口。换言之，在框架设计中，我们应该把功夫花在框架中最常被使用的那部分功能上。

---

<sup>1</sup> 举例来说，如果一个框架组件是可用的，那么无论是在控制台应用、Windows Forms 应用还是在 ASP.NET 应用中，它都应该具有相同的编程模型。

✓ **DO** 要了解具有不同编程风格、需求和技能水平的开发者，并明确地为他们进行设计。

■ **PAUL VICK** 为 Visual Basic (VB) 开发者设计框架没有“银弹”。我们的用户范围很广，从第一次使用编程工具的小白到构建大规模商业应用的行业老手。设计一个吸引 VB 开发者的框架，关键是要让他们能够以最少的麻烦和困扰完成工作。设计一个使用概念最少的框架是一个好主意——不是因为 VB 开发者不能处理概念，而是因为他们不得不停下来思考与手头工作无关的概念，从而导致工作流程被打断。VB 开发者的目标通常不是学习一些有趣或令人兴奋的新概念，也不是要去欣赏你的设计在智力上的纯粹和简单性，他们的目的是要完成工作并继续前进。

■ **KRZYSZTOF CWALINA** 为像你一样的用户做设计很容易，而为与你不同的人做设计则非常困难。有太多的 API 是由领域专家设计的，坦率地说，它们只对领域专家有利。问题是，大多数开发者不是、永远不会是、也不需要成为现代应用中所有技术领域的专家。

■ **BRAD ABRAMS** 尽管惠普公司著名的座右铭“为下一个工作台的工程师而构建”对于推动软件项目的质量和完整性是有用的，但对于 API 的设计来说，它却具有误导性。例如，Microsoft Word 团队的开发人员清楚地知道，他们自己不是 Word 的目标客户。我的母亲才是目标客户。因此，Word 团队放入了更多我的母亲可能认为有用的功能，而不是开发团队认为有用的功能。尽管这在 Word 这样的应用程序中是显而易见的，但我们在设计 API 时往往会忽略这个原则。我们倾向于只为自己设计 API，忽视了客户的应用场景。

✓ **DO** 要了解各种编程语言，并为之做设计。

许多编程语言都实现了对通用语言运行时 (CLR) 和 .NET 的支持，其中的一些语言和你用来实现 API 的语言之间可能会有很大的差别，通常需要额外的考量来保证你的 API 在这些语言上能正常运行。

举例来说，开发者在使用能够和 .NET 交互的动态类型语言（如 PowerShell 和 IronPython 等）时，如果所使用的 API 要求他们创建某些具有特性（Attribute）的自定义类型，这时候他们就极有可能无法正常使用该 API。

另一个例子是，F# 语言不支持用户自定义的隐式转换运算符。因此，如果 API 使

用隐式转换运算符来简化其调用模式，那么在 F# 中，该 API 可能并不见得会易于使用。

■ **JAN KOTAS** 为现有编程语言的“最小公分母”做设计阻碍了 .NET 平台的发展。近年来，为了实现创新，人们不再强调这一点，如特性 `Span<T>` 的引入。C# 和 F# 引进新的语言特性以启用 `Span<T>`，而其他编译到 .NET 的语言（如 Visual Basic）则尚未支持它，因此这些语言无法使用新的基于 `Span<T>` 的 API。

■ **JEREMY BARTON** 虽然我们确实在没有 VB 支持的情况下添加了 `Span<T>`，但是建议使用基于数组的替代方法（请参见 9.12 节）。尽管这在一定程度上是基于可用性给出的建议，但是最终仍然回归到本准则，多种语言都可以与 .NET CLR 进行交互。

本书还介绍了其他关于不同编程语言支持的特别注意事项。

## 2.1 渐进式框架

为广大开发者、应用场景和编程语言设计出一个唯一的框架是困难且代价高昂的。过去，框架供应商为特定的开发者群体提供了若干产品来应对特定的场景。例如，微软提供的 Visual Basic API 面向简单性和相对受限的场景进行了优化，Win32 API 则针对功能和灵活性进行了优化，即使这意味着牺牲了使用上的便利性。其他框架，如 MFC 和 ATL，也是针对特定的开发者群体和应用场景的。

尽管这种多框架模式已被证明是提供 API 的一种成功方式，这些 API 对特定的开发者群体来说功能强大且易于使用，但是它有着明显的缺点。主要缺点<sup>1</sup>是，众多的框架使得使用其中某个框架的开发者很难将他们的知识转移到下一个技能水平或应用场景中（通常需要使用不同的框架）。例如，当开发者需要使用更强大的功能来实现另一个不同的应用时，他们将面临非常陡峭的学习曲线，因为他们需要学习一种几乎全新的编程模式，如图 2.1 所示。

<sup>1</sup> 其他缺点包括：那些基于其他框架进行包装的框架上市时间会更慢、工作重复、缺乏通用工具。

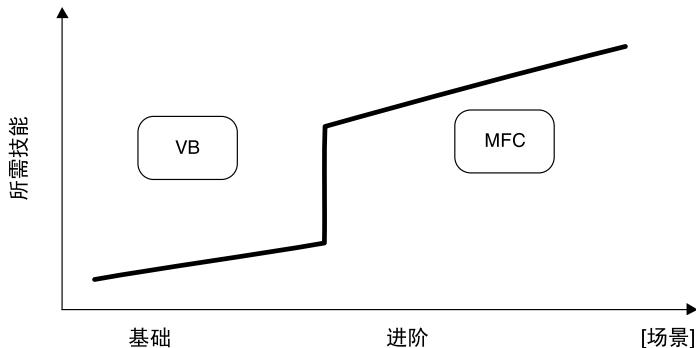


图 2.1 多框架平台学习曲线

**ANDERS HEJLSBERG** 在 Windows 早期，你可以直接使用 Windows API。要编写一个应用程序，你只需要先启动 C 语言编译器，`#include windows.h`，创建一个 `winproc`，再处理窗口消息——基本上老式 Petzold 风格的 Windows 编程就是如此。尽管这种方式可以工作，但它既不是特别具有生产力，也不是特别容易使用。

随着时间的推移，基于 Windows API 的各种编程模型相继出现。VB 选择拥抱快速应用开发（Rapid Application Development, RAD），利用 VB，你可以实例化一张表单，将组件拖曳到表单上，然后为之编写事件处理器。你的代码通过委托来运行。

在 C++ 的世界中，我们使用 WFC 和 ATL，采用了完全不同的方式。这里涉及的关键概念是子类化，开发者从大型的面向对象库中派生出子类。尽管这可以给你带来更多的功能和更强大的表达能力，但是和 VB 的组合模型相比，它并不具备同等的易用性和生产力。

如果你看看上面这张图就会发现，其中一个问题是你在编程模型上的选择也必然成为你对编程语言的选择。这很糟糕。如果你是一个经验丰富的 MFC 开发者，现在你需要用 VB 编写几行代码，你已有的经验并不能直接被拿过来使用。同样，即使你非常了解 VB，你的那些知识也不能帮助你使用 MFC。

面对不同的编程模型，API 同样不具备一致的可用性。面对不同的问题，每个模型都凭空捏造出它们自己的解决方案，但是实际上其可能是所有模型共同面临的核心问题。例如，如何处理文件 I/O，如何处理字符串格式化，如何处理安全性、线程，等等。

.NET 框架所做的就是统一所有的这些模型。无论你使用的是哪种编程语言，也无论你面向的是哪种编程模型，在任何地方，它都可以为你提供可用且一致的 API。

■ **PAUL VICK** 值得注意的是，这种统一是有代价的。在编写框架时，有一个无法解决的矛盾——是应该暴露大量的功能给用户，使得用户可以对其行为表现进行各种控制，还是应该保持概念的简单，只为用户提供相对来说更有限的功能。在大多数情况下，没有“银弹”，在功能和简单性上做出取舍是不可避免的。在设计 .NET 框架的过程中，大量的工作被投入到实现二者之间的平衡上。我认为，我们今天仍然在继续做这件事情。

更好的办法是提供一个渐进式框架，它是一个面向广大开发者的框架，允许开发者由浅入深地拓展他们的知识。.NET 框架正是这样一个框架，它提供了平滑的学习曲线，如图 2.2 所示。

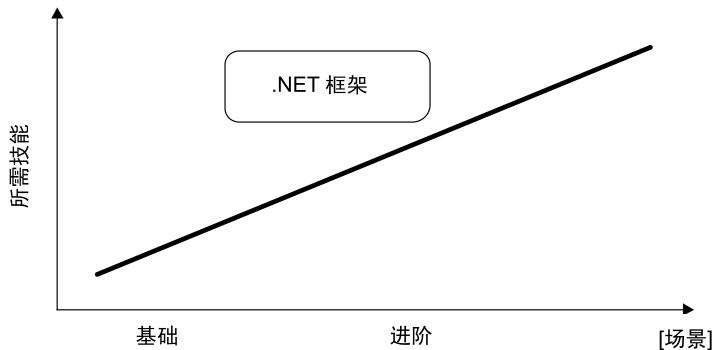


图 2.2 渐进式框架学习曲线

实现这样一个有着较低起点的平滑学习曲线的框架是困难的，但也绝非是不可能的。困难在于，它要求我们使用一种全新的框架设计方法，需要更深厚的设计知识，也有着更高的设计成本。幸运的是，本章和本书中所描述的准则都旨在指导你完成这个困难的设计过程，并最终帮助你设计一个出色的渐进式框架。

你应该始终牢记开发者社区是非常庞大的，从录制宏指令的办公室白领到底层设备驱动的作者，不一而足。任何试图去服务所有这些用户的框架最终都会变得一团糟，到头来甚至无法满足任何一个用户。渐进式框架的目标是在广大的开发者中尽量去拓宽它的用户群体，但不是去满足每一个潜在的开发者。这意味着那些不属于该目标群体的开发者将需要特定的 API。

## 2.2 框架设计基本原则

提供一个功能强大且易于使用的开发平台是.NET 的首要目标之一，如果你正在扩展它的话，这也应该是你的目标之一。第一版.NET 框架实际上已经为开发者提供了功能强大的 API，但是一些开发者还是觉得框架中的某些部分很难使用。

■ **RICO MARIANI** 另一方面，不仅要使 API 本身易于使用，还要确保开发者能按照 API 所设计的正确方式来使用它。想清楚你应该提供怎样的模式，确保开发者可以在最自然的方式下使用你提供的系统并得到正确的结果，即使它受到攻击也能保证安全，还要能提供出色的性能，确保它不会被开发者错误地使用。几年前我曾写过：

### 成功之坑

与通过反复尝试，在旅途中遭遇各种“惊喜”，最后登上顶峰，或者横穿沙漠的那种成功形成鲜明的对比，我们希望用户可以通过使用我们的平台和框架更简单地获得成功。反之，如果我们使它变得更容易导致问题的话，则是我们的失败。

真正的生产力是使开发者能够容易地创造出优秀的产品——而不是能够容易地生产垃圾。要构建成功之坑。

用户反馈和可用性调研表明，很大一部分 VB 开发者在学习 VB.NET 时会遇到问题。部分问题源自一个简单的事实——.NET 和 VB 6.0 库是不同的，但是也有一些是 API 设计导致的可用性问题。解决这些问题，成为微软在 .NET Framework 2.0 开发中的首要事项。

本节中描述的这些原则以标签“框架设计原则”来标识，旨在解决上述问题。它们主要是为了帮助框架设计者避免那些会造成严重不良后果的错误设计，这些错误是从许多的可用性调研和用户反馈中总结而来的。我们相信，这些原则是设计任何通用框架的核心所在。一些原则和建议有重叠，这也从另一个角度证明了其正确性。

### 2.2.1 场景驱动设计原则

框架往往包含了大量的 API，这对于实现具有强大功能和表现力的高级场景是必要的。然而，绝大多数开发实际上都围绕着一组常见的场景展开，只依赖完整框架中相对来说比较核心的一部分。为了提升框架使用者整体的生产力，应该将大量的投入集中于那些在最常见的场景所使用的 API 的设计中，这至关重要。

为此，框架设计应该侧重于一组通用场景，使整个设计过程都由场景来驱动。我们

建议，框架设计者首先把自己当作框架使用者，为主要的使用场景写一些代码，然后再设计对象模型来支撑这些代码片段<sup>1</sup>。

### ■ 框架设计原则

框架设计必须从一组使用场景和实现这些场景的代码示例开始。

**■ KRZYSZTOF CWALINA** 我想在刚才阐释的原则上补充一点，“要设计出一个出色的框架，根本没有其他方法”。如果我只能挑选一条设计原则放到本书中，那么就只能是它了。如果我不是写一本书，而只是写一篇简要的文章来介绍在 API 设计中什么是重要的，我依然会挑选这一原则。

框架设计者经常犯这样的错误，首先（运用各种设计原则）设计对象模型，然后根据最终实现的 API 来编写示例代码。问题是，许多设计原则（包括最常用的那些面向对象设计原则）都是为了最终实现的可维护性来优化的，而不是为了所产出 API 的可用性。它们最适合框架内部的架构设计，但对于大型框架的公开 API 来说并不适合。

在设计一个框架时，首先应该产出场景驱动的 API 规范（参考“附录 C”）。这个规范可以独立于功能规范，也可以是一个较大的规范文档的其中一部分。对于后者来说，API 规范在位置和时间上都应先于功能规范。

这个规范应该包含场景，在给定的技术领域内，将前 5~10 个场景列出来，并给出实现这些场景的代码示例。当你的 API 或者代码示例使用了新的或其他不常用的语言特性时，你应当考虑使用至少一种别的语言再写一遍示例，因为有的时候，使用不同语言编写的代码会有非常大的差异。

使用不同的编码风格（使用语言独有特性）来编写这些场景相关代码也很重要。例如，VB.NET 对大小写不敏感，所以示例也应该反映这一点。C# 代码应该遵循第 3 章中所描述的标准。

✓ **DO** 要确保 API 设计规范是任何功能（包含公开可访问的 API）设计的核心。

“附录 C”中包含了满足该准则的设计规范示例。

✓ **DO** 要为主要功能领域定义主要使用场景。

API 规范应该包含描述主要场景的内容，并给出实现相应场景的代码示例。该内容

---

<sup>1</sup> 这与测试驱动开发（TDD）或基于用例的流程是相似的，但是仍有一些区别。TDD 更为重量级，因为除了驱动 API 设计，它还有其他目标。相较于一个个独立的 API 调用，用例通常被用来抽象更上层的问题。

应该紧跟在执行概览的后面，平均每个功能领域（如文件 I/O）应该有 5~10 个主要场景。

- ✓ **DO** 要确保场景切合适当的抽象水平。它们应该和终端用户的使用情况大致一致。  
例如，从文件中读取数据是一个很好的场景，但是打开文件、从文件中读取一行文本或者关闭文件都不是好的场景，它们的粒度太细了。
- ✓ **DO** 要先为主要场景编写代码示例，再定义对象模型来支持这些代码示例。  
例如，要设计一个 API 来测量代码运行的时间，你可能会写出如下场景代码示例：

```
// 场景一：测量经过的时间
Stopwatch watch = Stopwatch.StartNew();
DoSomething();
Console.WriteLine(watch.Elapsed);

// 场景二：重用 Stopwatch
Dim watch As Stopwatch = Stopwatch.StartNew()
DoSomething();
Console.WriteLine(watch.ElapsedMilliseconds)

watch.Reset()
watch.Start()
DoSomething()
Console.WriteLine(watch.Elapsed)
```

通过这些代码示例可以得到如下对象模型：

```
public class Stopwatch {
    public static Stopwatch StartNew();

    public void Start();
    public void Reset();

    public TimeSpan Elapsed { get; }
    public long ElapsedMilliseconds { get; }
    ...
}
```

■ **JOE DUFFY** 作为软件开发者，我们乐于创造强大且有意思的新功能，然后把它们分享给其他开发者，这正是 API 设计的乐趣之一。但是，退后一步，客观评估你所热衷的某个新功能在现实世界中能否真正发挥作用是极其困难的。要鉴别一个新功能是否被需要并确定其理想的使用方式，“使用场景”是我所知道的最佳方法。发掘场景实际上是非常困难的，因为它需要将技术能力和对客户需求的理解进行独特的结合。当你完成之后，你也只能基于本能的感受和直觉做出一系列决定，或许可以交付

一些有用的 API，但是仍然有风险做出令你自己后悔的决定。当有疑虑时，最好的方式就是把这个功能先拿出来，等更好地理解了需求之后再添加回去。

■ **STEPHEN TOUB** 添加新的 API 很有趣，但是添加的每一个 API 都是有一定的代价的。有时候，代价“只是”设计、开发、测试、写文档和维护功能（包括功能与运行时的结合）等工作。然而，遗憾的是，在通常情况下，当下添加的 API 实际上限制了将来某人添加其他更受欢迎的或更具影响力的新 API 的能力，因为它的功能可能会和该 API 冲突。为此，我们在选择添加什么样的新 API 时需要深思熟虑，因为这有可能阻碍了属于未来的创新。如果一定要问“我希望我们从未添加这个 API”这个问题，我敢说，我们中的许多人都能举出自己的例子。至少我就有一些。

- ✓ **DO** 要以至少两种不同的语言来编写主要场景代码示例（如 C# 和 F#）。最好确保所选择的语言有明显不同的语法、风格和能力。

■ **PAUL VICK** 如果你正在编写一个可以供多种语言使用的框架，那么实际了解几种编程语言（都是 C 语言风格的编程语言不算数）是非常有益的。我们发现，有时候一个 API 只能在某一种编程语言中正常工作，因为设计（和测试）这个 API 的人只懂得那一种编程语言。请多了解几种 .NET 语言，并按照设计好的正确方式来使用它们。在像 .NET 框架一样的多语言平台上，期待全世界都只用你的语言是行不通的。

- ✓ **CONSIDER** 建议使用动态类型语言如 PowerShell 或 IronPython 来编写主要场景代码示例。

设计不适合动态类型语言的 API 是很容易的。这类语言在处理泛型方法，以及依赖特性或需要创建强类型的 API 时通常会遇到问题。

- ✗ **DON'T** 在设计框架的公开 API 时，不要只依赖标准设计方法。  
标准设计方法（包括面向对象设计方法）都是针对最终实现的可维护性来优化的，它们并没有针对所产出 API 的可用性进行优化。场景驱动设计结合原型设计、可用性调研和一定次数的迭代优化是一种更好的方式。

■ **CHRIS ANDERSON** 每一个开发者都有属于其自己的原则，使用其他建模方式也并没有什么根本性错误，问题往往是在于结果。框架设计最好的开始方式是编写那

些你希望开发者去编写的代码——把它当成某种形式的测试驱动开发。你编写了最佳代码，它会反过来为你指出你期望构建的对象模型。

### 2.2.1.1 可用性调研

在广大开发者中进行框架原型的可用性调研是场景驱动设计的关键。使用为主要场景而设计的 API，对它们的作者来说可能很简单，但是对其他开发者来说并非一定如此。

理解开发者会如何处理每一个主要场景可以帮助我们洞察框架的设计，了解它应该如何很好地满足所有目标开发者的需求。出于这个原因，进行可用性调研——以正式或非正式的方式——是框架设计流程的重要组成部分。

如果在调研中发现大多数开发者都不能实现其中的某个场景，或者他们采取的方式和设计者期待的方式不一致，则表明该 API 应该被重新设计。

■ **KRZYSZTOF CWALINA** 在 .NET Framework 1.0 发布之前，我们没有对命名空间 System.IO 中的类型进行可用性测试。所以在发布后不久，我们收到了很多关于 System.IO 可用性的负面反馈。我们非常意外，并决定在 8 个随机用户中进行可用性调研，这 8 个人没有一个能成功在 30 分钟内从文件中读取出文本信息。我们认为，部分原因是文档搜索引擎存在问题，以及样本覆盖率不足。不过，更明显的是，API 本身的可用性有问题。如果在产品发布前做过调研，我们就能够消除大部分用户的不满，且能够避免在不引入破坏性变更的前提下试图修复主要功能区的 API 所带来的开销。

■ **BRAD ABRAMS** 对于 API 设计者来说，没有什么经历比坐在单向镜的后面，看着一个又一个开发者被自己设计的 API 挫败，最终无法完成任务，更能使他们深刻认识到其 API 的可用性问题。在 1.0 版本发布后的针对 System.IO 的可用性调研中，我自己经历了各种情绪。当看着一个又一个开发者不能完成这个简单的任务时，我的情感从傲慢到怀疑，然后是沮丧，最后下定决心去解决 API 中存在的问题。

■ **CHRIS SELLS** 可用性调研可以是正式的，前提是你要有钱，有时间。实际上，通过找到一些接近这个库目标用户的开发者，让他们运行一下你提出的 API，就可以得到 80% 的反馈结果。不要让“可用性调研”吓到你，以至于什么都不去做。你只需要把它当成这类“嗨，请帮忙看一眼这个”的调研来看待。

■ **STEVEN CLARKE** 我们发现，与其花费大量精力来计划、设计和进行需要大量参与者的规模调研来试图覆盖尽可能多的方面，不如在整个 API 开发过程中进行一系列较小的、更专注的调研。在每项调研中，我们只需要少量的参与者，专注于 API 的一个设计问题或一个领域。我们利用从中学到的经验知识对设计进行迭代，然后，在一两个星期后再针对更新的设计或 API 的其他领域发起另一项调研。这种持续学习的方式意味着，在整个设计过程中，有持续不断的源于客户的反馈在为我们提供信息，而不是在整个过程中的某个特定节点上一次性传递大量信息。

理想的 API 可用性调研应该基于广大目标开发者群体所使用的真实的开发环境、编辑器和文档，在现实情况下，最好是在产品周期的初期而不是后期进行可用性调研，所以不要因为产品还没准备好就推迟调研。

■ **STEPHEN TOUB** 你甚至不需要真正的实现来了解 API 的可用性。然而，最好还是让开发者可以运行一些东西来查看他们的实验结果，早期的设计反馈可以通过对其进行编码和编译的 API 来获得，这意味着所有的实现都可以是 no-ops 或者 throws；这不重要，因为它们不会被调用。开发者是否直观地找到了所涉及的相关类型？他们是否能够识别用于访问功能的模式？IntelliSense 能否以有意义的方式帮助指导他们使用 API？他们处理问题的方式是否和你假设的一样？他们是否经常搜索一些名字不同的东西？

通常来说，正式的可用性调研对小的开发团队和那些面向少量开发者的框架来说是不现实的。在这种情况下，非正式的调研是行之有效的方案。将一个初具雏形的库提供给不熟悉其设计的开发者，要求他们用 30 分钟写一个简单的程序，观察他们如何应对，这是找出那些最令人烦恼的 API 设计问题的有效方法。

✓ **DO** 要组织可用性调研来测试主要场景 API。

应该在开发周期的初期组织这些调研，因为严重的可用性问题往往需要进行大量的设计修改。在理想的情况下，大多数开发者都应该能够在不出现大问题的前提下为主要场景编写代码，如果他们做不到，则表明你应该重新设计相关 API。尽管重新设计是一种代价高昂的做法，但是我们发现，从长远来看，它实际上可以节省更多的资源，因为在不破坏现有代码的情况下修复不可用的 API 的成本是巨大的。

下一节将介绍设计 API 的重要性，以便在最初接触的时候不会让人感到沮丧。这就是所谓的低门槛原则。

## 2.2.2 低门槛原则

今天，很多开发者都希望能快速学习新框架的基础知识，他们希望在一定的基础上对框架的某些部分进行试用，只有当他们对某些功能感兴趣或需要使用更复杂的场景时，他们才会花时间来深入了解整个架构。初次接触就遇到设计糟糕的 API，会给开发者留下框架复杂难用的持久印象，进而使得一些开发者不愿使用这个框架。这就是为什么对于框架来说，为那些只是想试用框架的开发者提供一个较低的入门门槛是非常重要的。

### ■ 框架设计原则

框架必须简化上手试用的难度，为非专业用户提供较低的入门门槛。

许多开发者都希望通过试用 API 来搞清楚它究竟做了什么，然后逐步调整他们的代码来得到其想要的结果。

**■ PAUL VICK** 大多数开发者，无论他们使用的是什么语言，都是边做边学的。文档可以帮助你初步了解应该发生什么，但是我们都知道，除非你深入其中，开始捣鼓并尝试做出一些有用的东西，否则你永远不会真正了解某些东西是如何工作的。特别是 Visual Basic，鼓励通过这种方式来编程。虽然我们从不回避提前规划，但是我们努力使学习和编程成为一个连续的流程。编写不言而喻的 API，不需要开发者掌握复杂的知识就能够使用它们，例如，如何与多个对象或 API 交互，这可以有效促进这一流程（事实上，这似乎适用于大多数编程语言，不仅仅是 Visual Basic）。

有些 API 可以被测试，而有些则不然。为了便于试用，API 必须做到如下几点：

- 让与常见编程任务相关的类型和成员更容易识别。对于一个用于存放通用场景 API 的命名空间，如果其包含了 500 种类型，而实际上只有其中少数类型在通用场景中很重要，这是不容易试用的。对于面向主线场景的类型来说，如果其中的很多成员都是针对高级使用场景的，那么也是同样的。

**■ CHRIS ANDERSON** 在 Windows Presentation Foundation (WPF) 项目初期，我们遇到了差不多同样的问题。我们有一个基类型 `Visual`，几乎所有的其他元素都是从它派生出来的。但问题是，它所引入的成员和派生出元素的对象模型有冲突，特别是围绕子节点的问题。对于子视图渲染，`Visual` 只有一个单一的层次结构，但是我们的元素想要引入特定域的子元素（例如，一个 `TabControl` 只接收 `TabPage`s）。

我们的解决方案是创建一个 `VisualOperations` 类（该类具有作用于 `Visual` 的静态成员），而不是使每个元素的对象模型都变得复杂。

- 让开发者可以立即使用 API，不管它是否能达到开发者最终想要得到的效果。如果一个框架需要大量的初始化过程，或者依赖几个类型的实例化，然后把它们组合到一起，这是不易于试用的。同样地，如果 API 没有便捷重载（重载成员只有较短的参数列表），或者为属性设置了很糟的默认值，这也为那些想要尝试 API 的开发者设置了较高的壁垒。

**CHRIS ANDERSON** 把对象模型看作一张地图：你必须放置清晰的标志来解释如何从一个地方到另一个地方。你希望一个属性可以清楚地向人们展示它是做什么的、它需要什么样的值、赋值之后会发生什么。指向一个不能清楚地表明其派生类型是什么的抽象基类型是非常糟糕的事情。WPF 中的动画就是一个这样的例子：用于动画的类型是 `Timeline`，但是整个命名空间里没有什么是以单词“`Timeline`”结尾的。事实上，`Animation` 继承自 `Timeline`，还有很多其他类型，诸如 `DoubleAnimation` 和 `ColorAnimation` 等，但是属性类型和用于填充属性的有效项之间没有任何联系。

- 让发现和修复由于 API 被错误使用而带来的问题变得简单。例如，API 抛出的异常应该清楚地描述修复这个问题需要做些什么。

**CHRIS SELLS** 在编程的过程中，我特别喜欢这样的错误信息，它们指明了我哪里做错了，以及如何去解决这个问题。但是很多时候，我得到的只有前者，而我真正关心的其实是后者。

以下准则，将帮助你确保你的框架适合那些想通过动手实验来学习的开发者。

**✓ DO** 要确保每个功能领域的命名空间只包含那些用于通用场景的类型，面向高级场景的类型应该被放在子命名空间中。

例如，`System.Net` 命名空间只提供了面向网络编程主线场景的 API，更高级的 `Socket` API 被放在 `System.Net.Sockets` 这个子命名空间下。

**ANTHONY MOORE** 这条准则反过来讲也是成立的，可以这样表述：“不要把一个命名空间中最常用的类型埋到许多不常用的类型里面”。`StringBuilder` 正是这样

一个例子，我们后来希望能在一开始就把它包含到 `System` 命名空间下。它存在于 `System.Text` 中，但它的使用频率比这个命名空间里的其他类型高得多，同时它与其他类型也不是很相关。

尽管如此，这是 `System` 命名空间中唯一受此反向规则影响的类型。在大多数情况下，我们不得不忍受的是，有太多不常用的类型存在于这个命名空间下。

✓ **DO** 要为构造函数和方法提供简单的重载。简单的重载意味着只有非常少的参数，且所有的参数都是基础类型。

✗ **DON'T** 不要让面向主线场景的类型拥有面向高级场景的成员。

■ **BRAD ABRAMS** 在.NET 框架的设计中，一条重要的原则是通过减法来做加法。即，通过移除框架中的功能（也许它从未被添加到框架中），我们实际上可以让开发者变得更具生产力，因为他们只需要处理更少的概念。不支持多继承，是一个在 CLR 层面通过减法来做加法的经典范例。

✗ **DON'T** 不要让开发者在最基本的场景中显式地实例化一个以上的类型。

■ **KRZYSZTOF CWALINA** 图书出版商说，一本书的销量与该书中等式的数量成反比。该定律的框架设计者版本是：使用你的框架的开发者数量与前 10 个简单场景中需要显式调用的构造函数的数量成反比。

✗ **DON'T** 在为基本场景编程前，不要要求用户执行任何大规模的初始化。

主线场景 API 应该被设计成只需要最少量的初始化过程。在理想的情况下，使用为基本场景设计的类型，一个默认的构造函数或一个只有简单参数的构造函数就已经足够了。

```
var zipCodes = new Dictionary<string,int>();
zipCodes.Add("Redmond",98052);
zipCodes.Add("Sammamish",98074);
```

如果某个初始化过程是必要的，那么由于未执行所要求的初始化过程而导致的异常应该清楚地说明需要执行的操作。

■ **STEVEN CLARKE** 自从本书第1版发行以来，我们已经在这个领域中做了重要的可用性调研。我们一次又一次地观察到，那些需要大量初始化的类型大大提高了用

户入门的门槛。其后果是，一些开发者会选择不去使用这些类型，而是变相寻找看起来可以完成这项工作的其他类型，最终，一些开发者会错误地使用这些类型，只有少数开发者可以找到正确使用这些类型的方法。

ADO.NET 就是这样一个例子，我们的用户发现某个功能领域很难使用，因为它需要大量初始化工作，即使在最简单的场景中，用户也需要理解几种类型之间复杂的交互和依赖关系，甚至在一个简单的场景中，用户也必须实例化几个对象（如 `DataSet`、`DataAdapter`、`SqlConnection` 和 `SqlCommand` 的实例）并将它们关联在一起。需要指出的是，在 .NET Framework 2.0 中，通过添加辅助类的方式解决了许多同类型问题，大大简化了基本场景。

✓ **DO** 在可能的情况下，要为所有的属性和参数（使用便捷重载）提供合适的默认值。

关于这个概念，`System.Messaging.MessageQueue` 是一个很好的示例。组件只需要传递一个表明路径的字符串到它的构造函数中，再调用 `Send` 方法，就可以发送消息了。消息的优先级、加密算法和其他消息属性可以通过在简单场景的基础上增加代码来进行自定义。

```
var ordersQueue = new MessageQueue(path);
ordersQueue.Send(order); // 使用默认的优先级、加密算法等。
```

不能盲目地运用该准则。如果默认值可能会使用户误入歧途，则框架设计者应避免提供默认值。例如，默认值永远不应该导致安全漏洞或者糟糕的可执行代码。

■ **STEPHEN TOUB** 在设计默认值时，了解 API 的主要用例是很重要的，而且应尽可能预测未来会出现的用例。我的前 10 个“我希望我可以重新来做”的案例之一来自 `System.Threading.Tasks`，在这个 API 的设计之初，我们主要专注于基于 CPU 的并行性，但是随着时间的变化，主要的用例最终变成了基于 IO 的异步。一些初始的默认值更适合前者，而对于后者来说，其变成了危害。随着时间的推移，我们在添加易于使用的 API 的过程中解决了这些问题，但是对于仍在使用最早 API 的开发者而言，最初的问题和由此带来的困难依然存在。

■ **JEREMY BARTON** 这里的对比非常重要，很难找到合适的平衡点。.NET Cryptography API 包含了大量的类型，它们提供了友好且安全的默认值。遗憾的是，“友好”是不变的，但“安全”是一个动态的目标。有些时候，你为用户提供默认值是在帮助他们，但是有些时候，这又会对他们造成伤害。

✓ **DO** 要使用异常来传达 API 的不正确使用。

异常应该明确地描述导致异常的原因和开发者应该如何修改代码来解决问题。例如，`EventLog` 组件要求在写入事件之前设置好 `Source` 属性，如果在 `WriteEntry` 被调用前没有设置 `Source`，将会抛出这样的异常：“在写入事件日志之前没有设置 `Source` 属性”。

■ **STEVEN CLARKE** 在我们的可用性调研中，我们观察到许多开发者认为异常是 API 可以提供的最佳类型的文档。它提供的指导始终围绕着开发者要实现的目标，并且它真正支持被众多开发者青睐的边做边学的方法。

下一节将介绍使对象模型尽可能自文档化的重要性。

### 2.2.3 对象模型自文档化原则

许多框架都是由成百上千种类型和更多的成员及参数所组成的。开发者在使用这样的框架的过程中，需要大量的指导，以及频繁地去回忆 API 的意图和正确的使用方法。它自身的参考文档不能满足这一需求。如果需要参考文档来回答最简单的问题，一是可能很耗时，二是打断了开发者的工作流。而且，如前所述，许多开发者更喜欢通过反复试验来编码，只有在直觉失效时才会诉诸文档。

出于上述这些原因，设计不需要开发者每次执行简单的任务时都要查阅文档的 API 非常重要。我们发现，遵循一套简单的准则可以帮助开发者生成相对自文档化的直观 API。

#### ■ 框架设计原则

在简单场景中，框架必须可用且不需要文档。

■ **CHRIS SELLS** 在预计开发者将怎样学习使用你的框架时，永远不要低估 IntelliSense 的作用。如果你的 API 符合直觉，那么 IntelliSense 将可以满足一个新开发者 80% 的需求。要优化 IntelliSense。

■ **KRZYSZTOF CWALINA** 参考文档仍然是框架中很重要的一部分，设计出完全自文档化的 API 是不可能的。不同的开发者，根据他们的技术水平和过去的经验，他们会认为框架中的不同部分是不解自明的。同时，对于那些会花时间预先了解框架

总体设计的用户来说，文档仍然是至关重要的。对于所有这些用户来说，信息丰富、简洁且完整的文档与不解自明的对象模型一样至关重要。

✓ **DO** 要确保 API 符合直觉，且在基本场景中，要让开发者不需要参考文档就能够成功地使用 API。

✓ **DO** 要为所有的 API 提供出色的文档。

✓ **DO** 要为通用场景中的重要 API 提供代码示例来说明其用法。

不是所有的 API 都可以不解自明，一些开发者希望能够在使用 API 之前彻底地了解它们。

为了使框架可以自文档化，在选择名称和类型、设计异常等方面必须要小心。下面介绍了与自文档化 API 设计有关的一些重要的考虑因素。

### 2.2.3.1 命名

使框架自文档化最简单但也最常被忽略的方式，就是为最常见场景中所使用的类型保留简单且直观的名称。框架设计者经常为那些不怎么通用、大多数用户都不怎么在意的类型“烧”掉最好的名称。

例如，以 `File` 命名一个抽象基类，然后提供一个具体的类型 `NtfsFile`，如果所有用户在使用 API 之前都了解其中的继承关系，那么也没什么问题。但是，如果用户不了解这层关系，那么他们首先使用（且通常不会成功）的将是 `File` 类型。尽管该命名在面向对象设计的意义上可以很好地工作（毕竟，`NtfsFile` 是一种文件），但它无法通过可用性测试，因为 `File` 是大多数开发者直觉上认为应该使用的名称。

■ **KRZYSZTOF CWALINA** .NET 框架的设计者花了大量时间来讨论主要类型的命名替代方案。.NET 中的大多数标识符都有精心挑选的名称。一些命名得不那么好的情况是由于专注于概念和抽象而非主要场景导致的。

另一个建议是使用描述性的标识符名称，它应清楚地说明每个方法的作用，以及每种类型和参数所代表的含义。框架设计者在选择标识符的名称时不应该担心它过于冗长。例如，`EventLog.DeleteEventSource(string source, string machineName)` 看起来可能相当冗长，但是我们认为它具有肯定的可用性价值。

描述性的方法名只适用于那些简单的具有清晰语义的方法。这是我们应该遵循避免复杂的语义这个很好的通用设计原则的另一个原因。

总的来说，你在选择标识符的名称时应该格外小心。命名选择是框架设计者不得不做的重要抉择之一。在 API 发布后再去改变标识符的名称，将极其困难并且代价高昂。

✓ **DO** 要使关于标识符命名选择的讨论成为规范审阅的一个重要部分。

大多数场景会以哪种类型开头？在尝试实现此场景时，大多数人首先想到的名称是什么？用户首先想到的是通用类型的名称吗？例如，由于“File”是大多数人在处理文件 I/O 场景时会想到的名称，因此用于文件访问的主要类型应被命名为 `File`。同时，还应该讨论最常用类型的最常用方法和它们所有的参数。是不是任何熟悉你的技术，但不熟悉这个特定设计的人都能够快速、正确、轻松地识别和调用这些方法？

✗ **DON'T** 不要担心在使 API 自文档化的过程中使用冗长的标识符名称。

大多数标识符名称都应该清楚地表明每个方法的作用，以及每种类型和参数所代表的含义。

■ **BRENT RECTOR** 开发者阅读的标识符名称比其键入的名称多数百倍，甚至数千倍。现代编辑器更是将打字这样的琐事减少到最少，更长的名称使开发者可以通过 IntelliSense 更快地找到合适的类型或成员。此外，长期而言，那些使用具有良好类型命名的代码更容易理解和维护。

针对 C 语言家族开发者的一个特别注意事项是：要摆脱使用隐晦的标识符命名这一习惯带来的生产力下降，要从这个枷锁中解放出来。

✓ **CONSIDER** 在设计过程的前期就要让技术作家参与进来。他们可以成为一个很好的资源，帮助你发现那些命名不当的和很难向用户解释的设计。

✓ **CONSIDER** 要为最常用的类型保留最佳的类型名称。

如果你相信自己会在以后的版本中添加更多的高级 API，则请放心地在框架的第一个版本中为后续的 API 保留最佳名称。

■ **ANTHONY MOORE** 即使你从未想过要在以后使用该名称，也仍然有其他理由要求你避免使用过于笼统的名称。更具体的名称有助于使 API 变得更容易理解和可读。如果有人在代码中看到通用名称，则其可能会假定这是一个非常通用的应用程序，因此，对更特定的东西使用通用名称具有一定的误导性。此外，更具描述性的名称还有助于使用者分辨出类型与哪些场景或技术相关联。

### 2.2.3.2 异常

异常在自文档化框架设计中扮演了重要角色。通过异常消息，可以向开发者传达 API 正确的使用方法。例如，下面这段代码将会抛出一个消息为“在写入事件日志之前没有设置 `Source` 属性”的异常。

```
// C#
var log = new EventLog();
// log 没有设置 Source 属性
log.WriteEntry("Hello World");
```

✓ **DO** 要利用异常消息向开发者传达框架的使用错误。

例如，如果用户在使用 `EventLog` 这个组件时忘记了设置 `Source` 属性，那么任何依赖该属性的调用都应该在异常消息中声明这一点。第 7 章为异常及异常消息的设计提供了更多的指导。

### 2.2.3.3 强类型

强类型或许是决定 API 直观性的最重要因素。显然，调用 `Customer.Name` 要比调用 `Customer.Properties["Name"]` 容易。此外，以 `String` 形式返回名称的 `Name` 属性要比直接返回的 `Object` 更可用。

在有些情况下，使用属性包（property bag）、后期绑定调用（late-bound call）和其他弱类型的 API 也是必要的，但是它们应该只是该规则的一个例外，而不是通用实践。此外，设计者应该考虑为用户在非强类型 API 层上执行的最常见操作提供强类型辅助方法。例如，`Customer` 类型可能有一个属性包，但也应该为大多数常见属性（如 `Name`、`Address` 等）提供强类型 API 支持。

✓ **DO** 要尽一切可能提供强类型 API。

不要完全依赖弱类型 API，如属性包。在必须要用到属性包时，也要为属性包中最常用的属性提供强类型支持。

■ **VANCE MORRISON** 强类型（有更好的 IntelliSense 支持）是 .NET 框架比一般的 COM API 更容易“在编程中学习”的重要原因。我仍不时地需要使用由 COM 提供的功能，只要它是强类型，我就可以很好地使用。但是，当需要使用枚举值时，API 经常返回或接收一个泛型对象，或字符串参数，或传递的 DWORD，我需要花费 10 倍的时间来搞清楚到底需要传递什么。

### 2.2.3.4 一致性

与用户已经熟悉的现有 API 保持一致是设计自文档化框架的另一个强有力要素。这包括与其他的 .NET API 以及一些遗留的 API 保持一致。尽管如此，你不应该以遗留的 API 或设计不当的现有框架 API 为借口，以避开本书中所述的任何准则，但也不应该在没有理由的情况下随意更改符合标准的既定模式与设计。

✓ **DO** 要确保和 .NET 以及用户可能与之进行交互的其他框架的一致性。

一致性对于可用性来说非常重要。如果你的 API 和框架中用户熟悉的某些部分相似，那么他（她）将会认为你的设计自然且直观。你的 API 与其他 .NET API 的区别，应仅限于你的特定 API 的一些独有之处。

### 2.2.3.5 有限的抽象

通用场景 API 不应该使用太多的接口和抽象类，它应该符合系统的物理结构或众所周知的逻辑。

正如前面所提到的，标准的面向对象设计方法是针对代码的可维护性来优化的。这很合理，因为维护成本是开发软件产品的总体开销中最大的一部分。提升可维护性的方式之一就是使用诸如接口、抽象类这样的抽象。因此，现代设计方法倾向于使用大量的抽象。

问题在于，具有大量抽象概念的框架迫使用户在开始实现哪怕是最简单的场景之前就得成为框架架构的专家。然而，很多开发者并不渴望也没有业务上的理由来成为一名熟知该框架提供的所有 API 的专家。对于简单的场景，开发者要求 API 足够简单易用，且不需要他们来了解整个功能领域如何组合在一起。这是标准设计方法没有对其进行优化也从未声称要优化的问题。

当然，抽象在框架设计中有它们的地位。例如，抽象在提升框架的可测试性和一般可扩展性方面非常有用。由于设计良好的抽象，这种可扩展性通常是可能的。第6章讨论了如何设计可扩展 API，帮助你在过量的可扩展性与过少的可扩展性之间取得适当的平衡。

✗ **AVOID** 避免在主线场景 API 中使用太多的抽象。

■ **KRYSZTOF CWALINA** 抽象几乎总是必要的，但是太多的抽象表示系统过度工程化。框架设计者应该仔细地为客户设计，而不是为了自己的智力享受。

■ **JEFF PROSISE** 有过多抽象的设计也可能会影响性能。我曾经与一位客户合作，该客户对产品进行了重新设计，加入了大量的面向对象设计。他们将“所有”内容都用类来建模，最终得到了一个嵌套深得荒谬的对象层次结构。本来重新设计的部分目的是提高性能，但是“改进”后的软件运行速度比原来慢了4倍！

■ **VANCE MORRISON** 任何曾“乐于”调试C++ STL库的人都明白抽象是一把双刃剑。太多的抽象和代码会使之变得很难理解，因为你必须记住场景中所有抽象名称的真正含义。过度地使用泛型和继承是你可能过度泛化的常见症状。

■ **CHRIS SELLS** 老话常说，计算机科学中的任何问题都可以通过加一层抽象来解决。遗憾的是，开发者遇到的问题往往都是由它们造成的。

#### 2.2.4 分层架构原则

不是所有的开发者都被要求来解决同一类问题，不同的开发者通常需要所使用的框架提供不同层次的抽象和不同数量的控制器。一些经常使用C++和C#的开发者看重API的表现力和功能，我们将此类型的API称为底层(low-level) API，因为它们通常提供底层的抽象。相反，一些经常使用C#和VB.NET的开发者则更看重API的生产力和简单性，我们把这类API称为高级(high-level) API，因为它们提供了更高级别的抽象。通过使用分层设计，构建一个单一框架来满足这些截然不同的需求是完全可能的。

##### ■ 框架设计原则

分层设计使单一框架能同时提供功能和易用性。

■ **PAUL VICK** 将Visual Basic迁移到.NET平台的一部分原因是，许多VB开发者需要使用底层API来访问特定的功能，然而，我们提供的高级API并不具备这样的能力。VB开发者在开始时可能会花费大量的时间使用高级API来快速开发应用程序，但这并不能改变大多数开发者迟早需要调整或优化应用程序的事实。为了优化应用程序，开发者通常需要引入底层API来实现额外一小部分功能。因此，底层API的设计应当充分考虑到VB开发者。

要构建一个面向广大开发者的单一框架，一般准则是将 API 集合进行拆分，底层类型暴露出它所具备的所有能力，高级类型则应该基于更底层的实现封装出更方便的 API。

这是一个非常强大且简单的技巧。如果只有单层 API，你往往不得不在更复杂的设计和放弃某些场景的支持之间做出选择。拥有一个低阶的功能层，为将高级 API 真正用于主线场景提供了自由。

在某些情况下，我们可能并不需要其中的某个层次。例如，一些功能领域可能就只会暴露底层 API。

.NET JSON API 正是这种分层设计的一个例子。为了功能和表现力，`Utf8JsonReader` 提供了一个底层的 JSON 语法分析器，允许开发者针对 JSON 中的个别 token 进行编码。然而，.NET 也有 `JsonDocument` 和 `JsonElement` 类型，它们是基于 `Utf8JsonReader` 实现的，允许开发者面向高级的概念编程，例如文档结构，他们不需要关心对象深度的计数或转义字符串。类型具有一致性的行为和标识符，只是不同的层次面向的是不同的场景和受众。

对于 API 分层，有两种主要的命名空间拆解方式：

- 将不同的层次划分到不同的命名空间中。
- 将所有层次暴露在同一个命名空间中。

#### 2.2.4.1 将不同的层次划分到不同的命名空间中

拆分框架的一种方式是将高级和底层的类型放在不同但又相关联的命名空间中。这样做的好处是，当开发者需要实现更复杂的场景时，可以在主流场景中将底层类型隐藏起来，而又不至于将它们置于遥不可及的地方。

与 .NET 网络相关的 API 正是以这种方式来拆分的，有底层的 `System.Net.Sockets.Socket` 类型、中级的 `System.Net.Security.SslStream` 类型和高级的 `System.Net.Http.HttpClient` 类型。`HttpClient` 的实现最终依赖 `Socket` 和 `SslStream`，但是大多数需要使用 HTTP 的开发者可以直接使用 `HttpClient`，无须用到底层类型。

绝大多数框架应该遵循这种命名空间拆分方式。

#### 2.2.4.2 将所有层次暴露在同一个命名空间中

另一种拆分的方式是将高级和底层的类型放在同一个命名空间中。好处是，当我们有需要时，它能够自动回退到更复杂的功能上。缺点是，将复杂的类型放在同一个命名空间下会使一些场景实现变得更加困难，即使我们没有用到这些复杂的类型。

这种拆分适用于简单的功能，例如，`System.Text` 命名空间既包含底层类型，如 `Encoder` 类和 `Decoder` 类，也包含高级的 `Encoding` 类的子类。

■ **STEVEN CLARKE** 仔细考虑分层 API 的运行时行为。例如，如果开发者在某一个层上工作，确保其不会捕获从不同的层抛出的异常。确保在编写、阅读和理解代码时，开发者只需要真正关心某一层中发生的事情，并且可以将其他层安全地视为黑盒。

- ✓ **CONSIDER** 建议采用分层架构，针对生产力来优化高级 API，针对功能和表现力来优化底层 API。
- ✗ **AVOID** 避免在底层 API 很复杂（如包含很多类型）的情况下，将底层 API 和高级 API 放到同一个命名空间下。
- ✓ **DO** 要确保一个功能领域的各个层次能被很好地集成在一起。开发者应该能够使用其中任意一个层次来进行编程，当他们需要将相应的代码更改为使用另一个层次来实现时，不需要重写整个应用程序。

## 总结

---

在设计一个框架时，很重要的一点是要意识到框架的受众是形形色色的，无论是他们的需求还是技能水平，皆是如此。遵循本章中所介绍的原则，将确保你的框架可以为广大开发者群体所使用。