# 第 3 章
# 游戏引擎编程中的高级面向对象技术

**主要内容：**在游戏引擎设计中，面向对象技术的使用程度是游戏引擎封装好坏的重要因素，但是面向对象将涉及软件工程中的许多重要内容，本章主要讲述面向对象中的两个重要技术：设计模式和泛型编程，这两种编程都属于面向对象编程中的高级内容，也是在实际游戏引擎开发中使用最广泛的内容。

**本章重点：**

- 设计模式的基本概念
- 设计模式的方法
- 游戏引擎中所用的设计模式
- 标准模板库
- 容器的使用
- 迭代器的使用

## 3.1 设计模式

设计面向对象软件比较困难，而设计可复用的面向对象软件就更加困难。必须找到相关的对象，以适当的粒度将它们归类，再定义类的接口和继承层次，建立对象之间的基本关系。设计应该对手头的问题有针对性，同时对将来的问题和需求也要有足够的通用性。大家都希望避免重复设计或尽可能少做重复设计。有经验的面向对象设计者会告诉你，要一下子就得到复用性和灵活性好的设计，即使不是不可能的，至少也是非常困难的。一个设计在最终完成之前常要被复用好几次，而且每一次都有所修改。

有经验的面向对象设计者的确能做出良好的设计，而新手则会面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长时间领会良好的面向对象设计是怎么回事儿。有经验的设计者显然知道一些新手所不知道的东西，这又是什么呢？

内行的设计者知道，不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。因此，会在许多面向对象系统中看到类和相互通信的对象（Communicating Object）的重复模式。这些模式解决特定的设计问题，使面向对象设计更灵活优雅，最终复用性更好。它们帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。

以下类比可以帮助说明这一点。小说家和剧本作家很少从头开始设计剧情。他们总是沿袭一些业已存在的模式，像“悲剧性英雄”模式（《麦克白》、《哈姆雷特》等）或“浪漫小说”模式（存在着无数浪漫小说）。同样地，面向对象设计员也沿袭一些模式，像“用对象表示状态”和“修饰对象以便于能容易地添加/删除属性”等。一旦懂得了模式，许多设计决策自然而然就产生了。

设计模式使人们可以更加简单方便地复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式帮助你做出有利于系统复用的选择，避免设计损害了系统复用性。通过提供一个显示类和对象作用关系以及它们之间潜在联系的说明规范，设计模式甚至能够提高已有系统的文档管理和系统维护的有效性。简而言之，设计模式可以帮助设计者更快更好地完成系统设计。

### 3.1.1 设计模式概述

一般而言，一个模式有以下 4 个基本要素：

（1）模式名称（Pattern Name）。一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是设计模式编目工作的难点之一。

（2）问题（Problem）。描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

（3）解决方案（Solution）。描述了设计的组成成分、它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

（4）效果（Consequences）。描述了模式应用的效果及使用模式应权衡的问题。尽管描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

出发点的不同会产生对什么是模式和什么不是模式的理解不同。一个人的模式对另一个人来说可能只是基本构造部件。本书中将在一定的抽象层次上讨论模式。“设计模式”并不描述链表和 hash 表那样的设计，尽管它们可以用类来封装，也可复用；也不包括那些复杂的、特定领域内的对整个应用或子系统的设计。本书中的设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述。

一个设计模式命名、抽象和确定了一个通用设计结构的主要方面，这些设计结构能被用来构造可复用的面向对象设计。设计模式确定了所包含的类和实例，它们的角色、协作方式以及职责分配。每一个设计模式都集中于一个特定的面向对象设计问题或设计要点，描述了什么时候使用它、在另一些设计约束条件下是否还能使用，以及使用的效果和如何取舍。



### 3.1.2 类的接口

设计模式采用多种方法解决面向对象设计者经常碰到的问题。这里给出几个问题以及使用设计模式解决它们的方法。

1. 寻找合适的对象

面向对象程序由对象组成，对象包括数据和对数据进行操作的过程，过程通常称为方法或操作。对象在收到客户的请求（或消息）后，执行相应的操作。

客户请求是使对象执行操作的唯一方法，操作又是对象改变内部数据的唯一方法。由于这些限制，对象的内部状态是被封装的，它不能被直接访问，它的表示对于对象外部是不可见的。

面向对象设计最困难的部分是将系统分解成对象集合。因为要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等，它们都影响着系统的分解，并且这些因素通常还是互相冲突的。

面向对象设计方法学支持许多设计方法。可以写出一个问题描述，挑出名词和动词，进而创建相应的类和操作；或者，可以关注于系统的协作和职责关系；或者，可以对现实世界建模，再将分析时发现的对象转化至设计中。至于哪一种方法最好，并无定论。

设计的许多对象来源于现实世界的分析模型。但是，设计结果所得到的类通常在现实世界中并不存在，有些是像数组之类的低层类，而另一些则层次较高。例如，Composition 模式引入了统一对待现实世界中并不存在的对象的抽象方法。严格反映当前现实世界的模型并不能产生也能反映将来世界的系统。设计中的抽象对于产生灵活的设计是至关重要的。

设计模式帮助你确定并不明显地抽象和描述这些抽象的对象。例如，描述过程或算法的对象现实中并不存在，但它们却是设计的关键部分。Strategy 模式描述了怎样实现可互换的算法族。

State 模式将实体的每一个状态描述为一个对象。这些对象在分析阶段，甚至在设计阶段的早期都并不存在，后来为使设计更灵活、复用性更好才将它们发掘出来。

2. 决定对象的粒度

对象在大小和数目上变化极大。它们能表示下至硬件或上至整个应用的任何事物。那么怎样决定一个对象应该是什么呢？

设计模式很好地解决了这个问题。Facade 模式描述了怎样用对象表示完整的子系统，Flyweight 模式描述了如何支持大量的最小粒度的对象。其他一些设计模式描述了将一个对象分解成许多小对象的特定方法。Abstract Factory 和 Builder 产生那些专门负责生成其他对象的对象。Visitor 和 Command 生成的对象专门负责实现对其他对象或对象组的请求。

3. 指定对象接口

对象声明的每一个操作指定操作名、作为参数的对象和返回值，这就是所谓的操作的型构（Signature）。对象操作所定义的所有操作型构的集合被称为该对象的接口（Interface）。对象接口描述了该对象所能接受的全部请求的集合，任何匹配对象接口中型构的请求都可以发送给该对象。

类型（Type）是用来标识特定接口的一个名字。如果一个对象接受 Window 接口所定义的所有操作请求，那么就说该对象具有 Window 类型。一个对象可以有许多类型，并且不同的对象可以共享同一个类型。对象接口的某部分可以用某个类型来刻画，而其他部分则可以用其他类型刻画。两个类型相同的对象只需要共享它们的部分接口。接口可以包含其他接口作为子集。当一个类型的接口包含另一个类型的接口时，则说它是另一个类型的子类型（Subtype）的超类型（Supertype）。我们常说子类型继承了它的超类型的接口。

在面向对象系统中，接口是基本的组成部分。对象只有通过它们的接口才能与外部交流，如果不通过对象的接口就无法知道对象的任何事情，也无法请求对象做任何事情。对象接口与其功能实现是分离的，不同的对象可以对请求做不同的实现，也就是说，两个有相同接口的对象可以有完全不同的实现。

当给对象发送请求时，所引起的具体操作既与请求本身有关又与接收对象有关。支持相同请求的不同对象可能对请求激发的操作有不同的实现。发送给对象的请求和它的相应操作在运行时刻的连接就称为动态绑定（Dynamic Binding）。

动态绑定是指发送的请求直到运行时刻才受具体的实现的约束。因而，在知道任何有正确接口的对象都将接受此请求时，可以写一个一般的程序，它期待着那些具有该特定接口的对象。进一步讲，动态绑定允许在运行时刻彼此替换有相同接口的对象。这种可替换性就称为多态（Polymorphism），它是面向对象系统中的核心概念之一。多态允许客户对象仅要求其他对象支持特定接口，除此之外对其假设几近于无。多态简化了客户的定义，使得对象间彼此独立，并可以在运行时刻动态改变它们相互的关系。

设计模式通过确定接口的主要组成成分及经接口发送的数据类型来帮助定义接口。设计模式也许还会告诉你接口中不应包括哪些东西。Memento 模式是一个很好的例子，它描述了怎样封装和保存对象内部的状态，以便一段时间后对象能恢复到这一状态。它规定了 Memento 对象必须定义两个接口：一个是允许客户保持和复制 Memento 的限制接口，一个是只有原对象才能使用的用来存储和提取 Memento 中状态的特权接口。

设计模式也指定了接口之间的关系。特别地，它们经常要求一些类具有相似的接口；或它们对一些类的接口做了限制。例如，Decorator 和 Proxy 模式要求 Decorator 和 Proxy 对象的接口与

被修饰的对象和受委托的对象一致。而 Visitor 模式中，Visitor 接口必须反映出 Visitor 能访问的对象的所有类。

4. 对象的实现

至此，我们很少提及到实际上怎样去定义一个对象。对象的实现是由它的类决定的，类指定了对象的内部数据和表示，也定义了对象所能完成的操作。

我们基于 OMT 的表示法，将类描述成一个矩形，其中的类名以黑体表示。操作在类名下面，以常规字体表示。类所定义的任何数据都在操作的下面。类名与操作之间以及操作与数据之间用横线分隔。

返回类型和实例变量类型是可选的，因为并未假设一定要用具有静态类型的实现语言。

对象通过实例化类来创建，此对象被称为该类的实例。当实例化类时，要给对象的内部数据（由实例变量组成）分配存储空间，并将操作与这些数据联系起来。对象的许多类似实例是由实例化同一个类来创建的。

新的类可以由已存在的类通过类继承（Class Inheritance）来定义。当子类（Subclass）继承父类（Parent Class）时，子类包含了父类定义的所有数据和操作。子类的实例对象包含所有子类和父类定义的数据，且它们能完成子类和父类定义的所有操作。以竖线和三角表示子类关系。

抽象类（Abstract Class）的主要目的是为它的子类定义公共接口。一个抽象类将把它的部分或全部操作的实现延迟到子类中，因此一个抽象类不能被实例化。在抽象类中定义却没有实现的操作被称为抽象操作（Abstract Operation）。非抽象类称为具体类（Concrete Class）。

子类能够改进和重新定义它们父类的操作。更具体地说，类能够重定义（Override）父类定义的操作，重定义使得子类能接管父类对请求的处理操作。类继承允许只简单地扩展其他类即可定义新类，从而可以很容易地定义具有相近功能的对象族。

抽象类的类名以斜体表示，以与具体类相区别。抽象操作也用斜体表示。图中可以包括实现操作的伪代码，如果这样，则代码将出现在带有褶角的框中，并用虚线将该褶角框与代码所实现的操作相连。

混入类（Mixin Class）是给其他类提供可选择的接口或功能的类。它与抽象类一样不能实例化。混入类要求多继承。

类继承与接口继承的比较对理解对象的类（Class）与对象的类型（Type）之间的差别非常重要。一个对象的类定义了对象是怎样实现的，同时也定义了对象的内部状态和操作的实现。

但是对象的类型只与它的接口有关，接口即对象能响应的请求的集合。一个对象可以有多个类型，不同类的对象可以有相同的类型。

当然，对象的类和类型是有紧密关系的。因为类定义了对象所能执行的操作，也定义了对象的类型。当我们说一个对象是一个类的实例时，即指该对象支持类所定义的接口。

类继承是一个通过复用父类功能而扩展应用功能的基本机制。它允许根据旧对象快速定义新对象，允许从已存在的类中继承所需要的绝大部分功能，从而几乎无需任何代价就可以获得新的实现。

然而，实现的复用只是成功的一半，继承所拥有的定义具有相同接口的对象族的能力也是很重要的（通常可以从抽象类来继承）。为什么呢？因为多态依赖于这种能力。

当继承被恰当地使用时，所有从抽象类导出的类将共享该抽象类的接口。这意味着子类仅仅添加或重定义操作，而没有隐藏父类的操作。这时，所有的子类都能响应抽象类接口中的请求，

从而子类的类型都是抽象类的子类型。

只根据抽象类中定义的接口来操纵对象有以下两个好处：

（1）客户无须知道他们使用对象的特定类型，只需要对象有客户所期望的接口。

（2）客户无须知道他们使用的对象是用什么类来实现的，只需要知道定义接口的抽象类。

这将极大地减少子系统实现之间的相互依赖关系，也产生了可复用的面向对象设计的如下原则：针对接口编程，而不是针对实现编程。

不将变量声明为某个特定的具体类的实例对象，而是让它遵从抽象类所定义的接口。这是本书设计模式的一个常见主题。

理解对象、接口、类和继承之类的概念对大多数人来说并不难，问题的关键在于如何运用它们写出灵活的、可复用的软件。设计模式将告诉你怎样去做。

1. 继承和组合的比较

面向对象系统中功能复用的两种最常用技术是类继承和对象组合（Object Composition）。正如我们已经解释过的，类继承允许你根据其他类的实现来定义一个类的实现。这种通过生成子类的复用通常被称为白箱复用（White-box Reuse）。术语“白箱”是相对可视性而言的：在继承方式中，父类的内部细节对子类可见。

对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组装或组合对象来获得。对象组合要求被组合的对象具有良好定义的接口。这种复用风格被称为黑箱复用（Black-box Reuse），因为对象的内部细节是不可见的。对象只以“黑箱”的形式出现。

继承和组合各有优缺点。类继承是在编译时刻静态定义的，且可直接使用，因为程序设计语言直接支持类继承。类继承可以较方便地改变被复用的实现。当一个子类重定义一些而不是全部操作时，它也能影响它所继承的操作，只要在这些操作中调用了被重定义的操作。

但是类继承也有一些不足之处。首先，因为继承在编译时刻就定义了，所以无法在运行时刻改变从父类继承的实现。更糟的是，父类通常至少定义了部分子类的具体表示。因为继承对子类揭示了其父类的实现细节，所以继承常被认为“破坏了封装性”。子类中的实现与它的父类有如此紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化。

当需要复用子类时，实现上的依赖性就会产生一些问题。如果继承下来的实现不适合解决新的问题，则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。一个可用的解决方法就是只继承抽象类，因为抽象类通常提供较少的实现。

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。组合要求对象遵守彼此的接口约定，进而要求更仔细地定义接口，而这些接口并不妨碍你将一个对象和其他对象一起使用。这还会产生良好的结果：因为对象只能通过接口访问，所以并不破坏封装性；只要类型一致，运行时刻还可以用一个对象来替代另一个对象；更进一步，因为对象的实现是基于接口写的，所以实现上存在较少的依赖关系。

对象组合对系统设计还有另一个作用，即优先使用对象组合有助于保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物。另一方面，基于对象组合的设计会有更多的对象（而有较少的类），且系统的行为将依赖于对象间的关系而不是被定义在某个类中。

这导出了面向对象设计的第二个原则：优先使用对象组合，而不是类继承。

理想情况下，不应为获得复用而去创建新的构件。应该能够只使用对象组合技术，通过

组装已有的构件来获得需要的功能。但是事实很少如此，因为可用构件的集合实际上并不足够丰富。使用继承的复用使得创建新的构件要比组装旧的构件来得容易。这样，继承和对象组合常一起使用。

然而，经验表明：设计者往往过度使用了继承这种复用技术。但依赖于对象组合技术的设计却有更好的复用性（或更简单）。你将会看到设计模式中一再使用对象组合技术。

2. 委托

委托（Delegation）是一种组合方法，它使组合具有与继承同样的复用能力。在委托方式下，有两个对象参与处理一个请求，接受请求的对象将操作委托给它的代理者（Delegate）。这类似于子类将请求交给它的父类处理。使用继承时，被继承的操作总能引用接受请求的对象，C++中通过 this 成员变量，Smalltalk 中则通过 self。委托方式为了得到同样的效果，接受请求的对象将自己传给被委托者（代理人），使被委托的操作可以引用接受请求的对象。

举例来说，可以在窗口类中保存一个矩形类的实例变量来代理矩形类的特定操作，这样窗口类可以复用矩形类的操作，而不必像继承时那样定义成矩形类的子类。也就是说，一个窗口拥有一个矩形，而不是一个窗口就是一个矩形。窗口现在必须显式地将请求转发给它的矩形实例，而不是像以前它必须继承矩形的操作。

引用名是可选的。委托的主要优点在于它便于运行时刻组合对象操作以及改变这些操作的组合方式。假定矩形对象和圆对象有相同的类型，只需简单地用圆对象替换矩形对象，得到的窗口就是圆形的。

委托与那些通过对象组合以取得软件灵活性的技术一样，具有如下不足之处：动态的、高度参数化的软件比静态软件更难以理解。还有运行低效问题，不过从长远来看人的低效才是更主要的。只有当委托使设计比较简单而不是更复杂时，它才是好的选择。要给出一个能确切告诉你什么时候可以使用委托的规则是很困难的。因为委托可以得到的效率是与上下文有关的，并且还依赖于你的经验。委托最适用于符合特定程式的情形，即标准模式的情形。

有一些模式使用了委托。在 State 模式中，一个对象将请求委托给一个描述当前状态的 State 对象来处理。在 Strategy 模式中，一个对象将一个特定的请求委托给一个描述请求执行策略的对象，一个对象只会有一个状态，但它对不同的请求可以有许多策略。这两个模式的目的都是通过改变受托对象来改变委托对象的行为。在 Visitor 中，对象结构的每个元素上的操作总是被委托到 Visitor 对象。

其他模式则没有这么多地用到委托。Mediator 引进了一个中介其他对象间通信的对象。有时，Mediator 对象只是简单地将请求转发给其他对象；有时，它沿着指向自己的引用来传递请求，使用真正意义的委托。Chain of Responsibility 通过将请求沿着对象链传递来处理请求，有时，这个请求本身带有一个接受请求对象的引用，这时该模式就使用了委托。Bridge 将实现和抽象分离开，如果抽象和一个特定实现非常匹配，那么这个实现可以代理抽象的操作。委托是对象组合的特例。它告诉你对象组合作为一个代码复用机制可以替代继承。

3. 继承和参数化类型的比较

另一种功能复用技术（并非严格的面向对象技术）是参数化类型，也就是类属或模板。它允许在定义一个类型时并不指定该类型所用到的其他所有类型。未经指定的类型在使用时以参数形式提供。例如，一个列表类能够以它所包含元素的类型来进行参数化。如果想声明一个 Integer 列表，只需将 Integer 类型作为列表参数化类型的参数值；声明一个 String 列表，只需提供 String

类型作为参数值。语言的实现将会为各种元素类型创建相应的列表类模板的定制版本。

参数化类型提供了除类继承和对象组合之外的第三种方法来组合面向对象系统中的行为。许多设计可以使用这三种技术中的任何一种来实现。实现一个以元素比较操作为可变元的排序例程，可有如下方法：

（1）通过子类实现该操作（Template Method 的一个应用）。

（2）实现为传给排序例程的对象的职责。

（3）作为 C＋＋模板类的参数，以指定元素比较操作的名称。

这些技术存在着极大的不同之处。对象组合技术允许在运行时刻改变被组合的行为，但是它存在间接性，比较低效。继承允许提供操作的默认实现，并通过子类重定义这些操作。参数化类型允许改变类所用到的类型。但是继承和参数化类型都不能在运行时刻改变。哪一种方法最佳，取决于你设计和实现的约束条件。

参数化类型在像 Smalltalk 那样的编译时刻不进行类型检查的语言中是完全不必要的。

一个面向对象程序运行时刻的结构通常与它的代码结构相差较大。代码结构在编译时刻就被确定下来了，它由继承关系固定的类组成。而程序的运行时刻结构是由快速变化的通信对象网络组成的。事实上两个结构是彼此独立的，试图由一个去理解另一个就好像试图从静态的动植物分类去理解活生生的生态系统的动态性；反之亦然。

考虑对象聚合和相识的差别以及它们在编译和运行时刻的表示是多么的不同。聚合意味着一个对象拥有另一个对象或对另一个对象负责，一般我们称一个对象包含另一个对象或者是另一个对象的一部分。聚合意味着聚合对象和其所有者具有相同的生命周期。

相识意味着一个对象仅仅知道另一个对象。有时相识也被称为“关联”或“引用”关系。相识的对象可能请求彼此的操作，但是它们不为对方负责。相识是一种比聚合要弱的关系，它只标识了对象间较松散的耦合关系。可以用普通的箭头线表示相识，尾部带有菱形的箭头线表示聚合，实例聚合和相识很容易混淆，因为它们通常以相同的方法实现。C++中，聚合可以通过定义表示真正实例的成员变量来实现，但更通常的是将这些成员变量定义为实例指针或引用；相识也是以指针或引用来实现的。

从根本上讲，是聚合还是相识是由你的意图而不是由显示的语言机制决定的。尽管它们之间的区别在编译时刻的结构中很难看出来，但这些区别还是很大的。聚合关系使用较少且比相识关系更持久；而相识关系则出现频率较高，但有时只存在于一个操作期间，相识也更具动态性，使得它在源代码中更难被辨别出来。

程序的运行时刻结构和编译时刻结构存在这么大的差别，很明显代码不可能揭示关于系统如何工作的全部。系统的运行时刻结构更多地受到设计者的影响，而不是编程语言。对象和它们的类型之间的关系必须更加仔细地设计，因为它们决定了运行时刻程序结构的好坏。

许多设计模式（特别是那些属于对象范围的）显式地记述了编译时刻和运行时刻结构的差别。Composite 和 Decorator 对于构造复杂的运行时刻结构特别有用。Observer 也与运行时刻结构有关，但这些结构对于不了解该模式的人来说是很难理解的。Chain of Responsibility 也产生了继承所无法展现的通信模式。总之，只有理解了模式，才能清楚代码中的运行时刻结构。

获得最大限度复用的关键在于对新需求和已有需求发生变化时的预见性，要求你的系统设计要能够相应地改进。

为了设计适应这种变化且具有健壮性的系统，必须考虑系统在它的生命周期内会发生怎样的

变化。一个不考虑系统变化的设计在将来就有可能需要重新设计。这些变化可能是类的重新定义和实现，修改客户和重新测试。重新设计会影响软件系统的许多方面，并且未曾料到的变化总是代价巨大的。

设计模式可以确保系统能以特定方式变化，从而帮助你避免重新设计系统。每一个设计模式允许系统结构的某个方面的变化独立于其他方面，这样产生的系统对于某一种特殊变化将更健壮。

下面列出一些导致重新设计的一般原因，以及解决这些问题的设计模式：

（1）通过显式地指定一个类来创建对象。在创建对象时指定类名将使你受特定实现的约束而不是特定接口的约束。这会使未来的变化更复杂。要避免这种情况，应该间接地创建对象。

（2）对特殊操作的依赖。当你为请求指定一个特殊的操作时，完成该请求的方式就固定下来了。为避免把请求代码写死，你将可以在编译时刻或运行时刻很方便地改变响应请求的方法。

（3）对硬件和软件平台的依赖。外部的操作系统接口和应用编程接口在不同的软硬件平台上是不同的。依赖于特定平台的软件将很难移植到其他平台上，甚至都很难跟上本地平台的更新。所以设计系统时限制其平台相关性就很重要了。

（4）对对象表示或实现的依赖。知道对象怎样表示、保存、定位或实现的客户在对象发生变化时可能也需要变化。对客户隐藏这些信息能阻止连锁变化。

（5）算法依赖。算法在开发和复用时常常被扩展、优化和替代。依赖于某个特定算法的对象在算法发生变化时不得不变化，因此有可能发生变化的算法应该被孤立起来。

（6）紧耦合。紧耦合的类很难独立地被复用，因为它们是互相依赖的。紧耦合产生单块的系统，要改变或删掉一个类，必须理解和改变其他许多类。这样的系统是一个很难学习、移植和维护的密集体。

松散耦合提高了一个类本身被复用的可能性，并且系统更易于学习、移植、修改和扩展。设计模式使用抽象耦合和分层技术来提高系统的松散耦合性。

（7）通过生成子类来扩充功能。通常很难通过定义子类来定制对象。每一个新类都有固定的实现开销（初始化、终止处理等）。定义子类还需要对父类有深入的了解。如重定义一个操作可能需要重定义其他操作。一个被重定义的操作可能需要调用继承下来的操作。并且子类方法会导致类爆炸，因为即使对于一个简单的扩充，也不得不引入许多新的子类。

一般的对象组合技术和具体的委托技术是继承之外组合对象行为的另一种灵活方法。新的功能可以通过以新的方式组合已有对象而不是通过定义已存在类的子类的方式加到应用中去。另一方面，过多地使用对象组合会使设计难以理解。许多设计模式产生的设计中，可以定义一个子类，且将它的实例和已存在的实例进行组合来引入定制的功能。

（8）不能方便地对类进行修改。有时不得不改变一个难以修改的类。也许你需要源代码而又没有（对于商业类库就有这种情况），或者可能对类的任何改变会要求修改许多已存在的其他子类。设计模式提供在这些情况下对类进行修改的方法。

这些例子反映了使用设计模式有助于增强软件的灵活性。这种灵活性所具有的重要程度取决于你将要建造的软件系统。下面来看一看设计模式在开发如下 3 类主要软件中所起的作用：应用程序、工具箱和框架。

（1）应用程序。

如果将要建造像文档编辑器或电子制表软件这样的应用程序（Application Program），那么它

的内部复用性、可维护性和可扩充性是要优先考虑的。内部复用性确保你不会做多余的设计和实现。设计模式通过减少依赖性来提高内部复用性。松散耦合也增强了一类对象与其他多个对象协作的可能性。例如，通过孤立和封装每一个操作来消除对特定操作的依赖，可使在不同上下文中复用一个操作变得更简单。消除对算法和表示的依赖可达到同样的效果。

当设计模式被用来对系统分层和限制对平台的依赖性时，它们还会使一个应用更具可维护性。通过显示怎样扩展类层次结构和怎样使用对象复用，它们可增强系统的易扩充性。同时，耦合程度的降低也会增强可扩充性。如果一个类不过多地依赖其他类，扩充这个孤立的类还是很容易的。

（2）工具箱。

一个应用经常会使用来自一个或多个被称为工具箱的预定义类库中的类。工具箱是一组相关的、可复用的类的集合，这些类提供了通用的功能。工具箱的一个典型例子就是列表、关联表单、堆栈等类的集合，C++的 I/O 流库是另一个例子。工具箱并不强制应用采用某个特定的设计，它们只是为你的应用提供功能上的帮助。工具箱强调的是代码复用，它们是面向对象环境下的“子程序库”。

工具箱的设计比应用设计要难得多，因为它要求对许多应用是可用的和有效的。再者，工具箱的设计者并不知道什么应用使用该工具箱及它们有什么特殊需求。这样，避免假设和依赖就变得很重要，否则会限制工具箱的灵活性，进而影响它的适用性和效率。

（3）框架。

框架是构成一类特定软件可复用设计的一组相互协作的类。例如，一个框架能帮助建立适合不同领域的图形编辑器，如艺术绘画、音乐作曲和机械。另一个框架也许能帮助你建立针对不同程序设计语言和目标机器的编译器。而再一个也许能帮助你建立财务建模应用。你可以定义框架抽象类的应用相关的子类，从而将一个框架定制为特定应用。

框架规定了你的应用的体系结构。它定义了整体结构、类和对象的分割、各部分的主要责任、类和对象怎么协作，以及控制流程。框架预定义了这些设计参数，以便于应用设计者或实现者能集中精力于应用本身的特定细节。框架记录了其应用领域共同的设计决策，因而框架更强调设计复用，尽管框架常包括具体的立即可用的子类。

这个层次的复用导致了应用和它所基于的软件之间的反向控制（Inversion of Control）。当使用工具箱（或传统的子程序库）时，需要写应用软件的主体并且调用想复用的代码。而当使用框架时，应该复用应用的主体，写主体调用的代码。不得不以特定的名字和调用约定来写操作的实现，但这会减少需要做出的设计决策。

不仅可以更快地建立应用，而且应用还具有相似的结构。它们很容易维护，且用户看来也更一致。另一方面，你也失去了一些表现创造性的自由，因为许多设计决策无须你来作出。

如果说应用程序难以设计，那么工具箱就更难了，而框架则是最难的。框架设计者必须冒险决定一个要适应于该领域的所有应用的体系结构。任何对框架设计的实质性修改都会大大降低框架所带来的好处，因为框架对应用的最主要贡献在于它所定义的体系结构。因此设计的框架必须尽可能地灵活、可扩充。

更进一步讲，因为应用的设计如此依赖于框架，所以应用对框架接口的变化是极其敏感的。当框架演化时，应用不得不随之演化。这使得松散耦合更加重要，否则框架的一个细微变化都将引起强烈反应。

刚才讨论的主要设计问题对框架设计而言最具重要性。一个使用设计模式的框架比不用设计模式的框架更可能获得高层次的设计复用和代码复用。成熟的框架通常使用了多种设计模式。设计模式有助于获得无须重新设计即可适用于多种应用的框架体系结构。

当框架和它所使用的设计模式一起写入文档时，我们可以得到另外一个好处。了解设计模式的人能较快地洞悉框架，甚至不了解设计模式的人也可以从产生框架文档的结构中受益。加强文档工作对于所有软件而言都是重要的，但对于框架其重要性显得尤为突出。学会使用框架常常是一个必须克服很多困难的过程。设计模式虽然无法彻底克服这些困难，但它通过对框架设计的主要元素做更显式的说明可以降低框架学习的难度。

因为模式和框架有些类似，人们常常对它们有怎样的区别和它们是否有区别感到疑惑。它们最主要的不同在于如下 3 个方面：

（1）设计模式比框架更抽象。框架能够用代码表示，而设计模式只有其实例才能表示为代码。框架的威力在于它们能够使用程序设计语言写出来，它们不仅能被学习，也能被直接执行和复用。而本书中的设计模式在每一次被复用时，都需要被实现。设计模式还解释了它的意图、权衡和设计效果。

（2）设计模式是比框架更小的体系结构元素。一个典型的框架包括了多个设计模式，而反之绝非如此。

（3）框架比设计模式更加特例化。框架总是针对一个特定的应用领域。一个图形编辑器框架可能被用于一个工厂模拟，但它不会被错认为是一个模拟框架。而本书收录的设计模式几乎能被用于任何应用。当然比我们的模式更特殊的设计模式也是可能的（如分布式系统和并发程序的设计模式），尽管这些模式不会像框架那样描述应用的体系结构。

框架变得越来越普遍和重要。它们是面向对象系统获得最大复用的方式。较大的面向对象应用将会由多层彼此合作的框架组成。应用的大部分设计和代码将来自于它所使用的框架或受其影响。

### 3.1.3 游戏引擎中所用的设计模式

RPG 游戏框架采用到常用的单件、对象工厂等模式。这些模式主要体现在游戏框架层次接口规范、对象创建、对象管理以及相关功能调用上。早期的 RPG 游戏采用面向过程的方法为主要设计和实现办法。当游戏规模达到一定级别后，游戏框架的各个模块就会出现命名重复、接口功能难以管理、全局对象过多等致命的问题。采用面向对象的设计方法并辅助采用部分简单的设计模式可以基本解决这类问题。

1. 单件模式

对于整个游戏中只有一个实例存在的情况，使用单件模式，管理者就是这种模式的产物。该单件模式图描述了单件类（Singlelon）的组成，其中包括实例接口 histance、单件操作接口 singletonOperationo 和单件、数据提取接口 GetsingletonData，实例句柄存储如 queInstance 和单件的数据集合 singletonData，任何对单件的应用都是先通过 Instance 获实例后再进行类本身的方法调用。singleton 模式要求一个类有且仅有一个实例，并且提供了一个全局的访问点。

这就提出了一个问题：如何绕过常规的构造器提供一种机制来保证一个类只有一个实例？客户程序在调用某一个类时，它是不会考虑这个类是否只能有一个实例等问题的，所以，这应该是类设计者的责任，而不是类使用者的责任。从另一个角度来说，singleton 模式其实也是一种职责型模式。因为创建了一个对象，这个对象扮演了独一无二的角色，在这个单独的对象实例中，它

集中了它所属类的所有权力，同时它也肩负了行使这种权力的职责。

2. 对象工厂模式

对象工厂用于灵活的对象层次的实例创建。想要创建的对象类型，它包含了自己的操作，MyDocument 是从 Document 中派生的类型，My Application 是用户自己的对象创建器，而 Application 则负有创建 Document 的责任，它便是对象工厂，通过对 createDocument 的调用，它创建了 MyDocument 的实例，而每一个 MyApplication 必须注册自己的对象类型的创建函数到 Application 中去。

在游戏系统中，经常面临着“一系列相互依赖的对象”的创建工作，同时由于需求的变化，往往存在着更多系列对象的创建工作。如何应对这种变化？如何绕过常规的对象的创建方法（new），提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？

在以下情况下应当考虑使用对象工厂模式：游戏系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。游戏系统有多于一个的产品族，而游戏系统只消费其中某一产品族。同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。游戏系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实例。

3. 观察者模式

观测者模式用于对象之间的信息通知。在逻辑游戏世界构建过程中，需要为某些对象建立一种“通知依赖关系”，一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。使用观察者模式，可以将这种依赖关系弱化，并形成一种稳定的依赖关系，从而实现软件体系结构的松耦合。

适用性：当一个抽象模型有两个方面，其中一个方面依赖于另一方面，将这二者封装在独立的对象中以使它们可以各自独立地改变和复用；当对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象有待改变；当一个对象必须通知其他对象，而它又不能假定其他对象是谁。换言之，不希望这些对象是紧密耦合的。

通过观察者模式，把一对多对象之间的通知依赖关系变得更为松散，大大地提高了程序的可维护性和可扩展性，也很好地符合了开放—封闭原则。

4. 命令模式

命令模式提供很好的过程框架。Command 定义一个 Excute 的方法框架，派生自 Command 的具体的 PasteCommand 实现 Excute 方法，实际的执行是通过对命令的执行方法进行动态绑定而执行 PasteCommand 的方法。

在主程序中，是通过 Command 的对象指针调用 Execute，所以 PasteCommand 无法绕过这一限制，这就体现了过程框架的意义。

在游戏软件系统中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合，比如要对行为进行“记录、撤消/重做、事务”等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将“行为请求者”与“行为实现者”解耦，将一组行为抽象为对象，可以实现二者之间的松耦合？这就需要命令模式。

将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

效果及实现要点：Command 模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“将行为抽象为对象”，实现命令接口的具体命令对象有时候根据需要可能会保存一些额外的状态信息。

通过使用 composite 模式，可以将多个命令封装为一个“复合命令”（Macrocommand）。Command 模式与 C#中的 Delegate 有些类似。但两者定义行为接口的规范有所区别：命令以面向对象中的“接口—实现”来定义行为接口规范，更严格、更符合抽象原则；Delegate 以函数签名来定义行为接口规范，更灵活，但抽象能力比较弱。

使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个、几百个，甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

适用性：使用命令模式作为在面向对象系统中的替代。call back 讲的便是先将一个函数登记上，然后在以后调用此函数。需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。系统需要支持命令的撤消（Undo）。命令对象可以把状态存储起来，等到客户端需要撤消命令所产生的效果时，可以调用 Undo 方法把命令所产生的效果撤消掉。命令对象还可以提供 Redo 方法，以供客户端在需要时再重新实施命令效果。如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志读回所有的数据更新命令，重新调用 Execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

5. 模板方法（TemplateMethod）

TemplateMethod 模式是比较简单的设计模式之一，但它却是代码复用的一项基本的技术，在类库中尤其重要。定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

实现要点：TemplateMethod 模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。除了可以灵活应对子步骤的变化外，“不用调用我，让我来调用你”的反向控制结构是 TemplateMethod 的典型应用。在具体实现方面，被 TemplateMethod 调用的虚方法可以具有实现，也可以没有任何实现（抽象方法、纯虚方法），但一般推荐将它们设置为 Protected 方法。

适用性：采用模板方法模式可以一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是 Opdyke 和 Johnson 所描述过的“重分解以一般化”的一个很好的例子。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。模板方法只在特定点调用 Hok 操作，这样就只允许在这些点进行扩展。

6. 迭代器模式（Iterator）

在面向对象的游戏软件设计中，经常会遇到一类集合对象，这类集合对象的内部结构可能有着各种各样的实现，但是归结起来有两点需要我们去关心：

（1）集合内部的数据存储结构。

（2）遍历集合内部的数据。面向对象设计原则中有一条是类的单一职责原则，所以要尽可

能地去分解这些职责，用不同的类去承担不同的职责。Iterator 模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

意图：提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

效果及实现要点：迭代抽象，访问一个聚合对象的内容而无需暴露它的内部表示。迭代多态，为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。迭代器的健壮性考虑，遍历的同时更改迭代器所在的集合结构会导致问题。

适用性：访问一个聚合对象的内容而无需暴露它的内部表示。支持对聚合对象的多种遍历。为遍历不同的聚合结构提供一个统一的接口，即支持多态迭代。

Iterator 模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

## 3.2 STL 使用基础

C++标准模板库（Standard Template Library，STL）是 C++语言的重要组成部分，提供了多个方面的特性，为游戏开发者提供了很多方便，所以本节将抛砖引玉地讲述 STL 的使用基础，概述 STL 最基础、最常用的使用方法。如果读者对 STL 感兴趣并希望深入了解的话，可以参考相关的书，可以是像《C++ primer》这样权威的 C++参考书，也可以是专门讲述 STL 的书籍。在游戏开发中，一个提高开发效率的最有效的方法就是使用模板库。

### 3.2.1 标准模板库简介

C++标准模板库存在名字空间 std 中，用一组头文件的方式呈现。这些文件表明了这个库的重要组成部分。在 C++标准模板库中主要有以下几个类型的库：

（1）容器。能保存其他对象。许多计算机都涉及到建立各种对象的汇集，以及对这些汇集的操作。逐个字符地读进一个字符串、逐个字符串地将该串打印出来都是这种工作的实际例子。一个以保存一批对象为主要用途的类称为容器。C++标准库中将包括如下容器：

- <vector>：T 的一维数组。
- <list>：T 的双向链表。
- <deque>：T 的双端队列。
- <stack>：T 的堆栈。
- <map>：T 的关联数组。
- <set>：T 的集合。
- <multimap>：关联数组。
- <multiset>：集合，值可以重复出现。

（2）迭代器。提供一种机制，使标准算法能通用于标准容器和类型。迭代器是一种处于容器中的元素序列的非常一般而有用的概念。C++标准库中包括迭代器：<iterator>，即迭代器及迭代器支持。

（3）算法。标准常用的算法，主要包括非修改性的序列操作、修改性的序列操作、序列操

作、序列排序、集合算法、堆操作、最大和最小算法、排列等算法。主要存在以下算法库：

- <algorithm>：通用算法
- <stdlib>：bsearch()和 qsort()。

对于以上模板库的内容，考虑到本书不是主要讲述 C＋＋的书籍，所以本书主要讲述容器和迭代器的使用方法。

### 3.2.2 容器和迭代器使用基础

在本书中容器方面的使用讲述以下 4 个库：<vector>、<list>、<deque>、<stack>，因为这是游戏编程中最常用的库。

1. Vector（向量）

标准的 vector 是定义在名字空间 std 中的一个模板，由<vector>给出。该模板是一个动态的数组，将提供快速访问数组中的任何一个元素，并可以实现排序功能。其定义如下：

template < class Type,　class Allocator = allocator<Type> > class vector

变量的定义说明如下：

Type：存储在向量中的数据。

Allocator：存储分配器对象，用以了解向量的分配和释放内存情况。这个变量是可选的，其默认值是 allocator<Type>。

该函数将可以提供在数组末端插入或删除元素的功能。这个容器善于在数组的开始和末端删除元素。

其包括如下操作方法：

（1）构造函数：进行模板类的构造和初始化。由于 vector 具有多个构造函数，其声明如下：

```
vector();
explicit vector(
   const Allocator& _Al
);
explicit vector(
   size_type _Count
);
vector(
   size_type _Count,
   const Type& _Val
);
vector(
   size_type _Count,
   const Type& _Val,
   const Allocator& _Al
);
vector(
   const _vector& _Right
);
template<class InputIterator>
   vector(
      InputIterator _First,
```

```
      InputIterator _Last
   );
template<class InputIterator>
   vector(
      InputIterator _First,
      InputIterator _Last,
      const Allocator& _Al
   );
```

其变量的定义为：

_Al：分配器用于该对象。get_allocator 将返回分配器对象。

_Count：向量的初始长度。

_Val：向量数组的初始值。

_Right：通过拷贝构造向量。

_First：拷贝的第一个值的位置。

_Last：拷贝的最后一个值的位置。

现给出该函数应用的一个实例：

```
#include <vector>
#include <iostream>
int main()
{
   using namespace std;
   vector <int>::iterator v1_Iter, v2_Iter, v3_Iter, v4_Iter, v5_Iter;

   // 创建一个空向量 v0
   vector <int> v0;

   // 创建一个有 3 个元素的向量 v1，其元素的默认值为 0
   vector <int> v1( 3 );

   //创建一个有 5 个值的向量 v2，其初始值为 2
   vector <int> v2( 5, 2);

   //创建一个有 3 个值的向量 v3，其初始值为 1 和从分配器中获得向量 v2
   vector <int> v3( 3, 1, v2.get_allocator() );

   // 拷贝向量 v2 到 v4
   vector <int> v4( v2 );

   //拷贝向量 v4 的第一个元素到第三个元素到向量 v5
  vector <int> v5( v4.begin() + 1, v4.begin() + 3 );

   cout << "v1 =" ;
   for ( v1_Iter = v1.begin() ; v1_Iter != v1.end() ; v1_Iter++ )
      cout << " " << *v1_Iter;
   cout << endl;

   cout << "v2 =" ;
```

```
   for ( v2_Iter = v2.begin() ; v2_Iter != v2.end() ; v2_Iter++ )
      cout << " " << *v2_Iter;
   cout << endl;

   cout << "v3 =" ;
   for ( v3_Iter = v3.begin() ; v3_Iter != v3.end() ; v3_Iter++ )
      cout << " " << *v3_Iter;
   cout << endl;

   cout << "v4 =" ;
   for ( v4_Iter = v4.begin() ; v4_Iter != v4.end() ; v4_Iter++ )
      cout << " " << *v4_Iter;
   cout << endl;

   cout << "v5 =";
   for ( v5_Iter = v5.begin() ; v5_Iter != v5.end() ; v5_Iter++ )
      cout << " " << *v5_Iter;
   cout << endl;
}
```

最后输出结果为：

```
v1 = 0 0 0
v2 = 2 2 2 2 2
v3 = 1 1 1
v4 = 2 2 2 2 2
v5 = 2 2
```

分配函数：进行向量的原有值的擦除，并分配现有值，其声明如下：

```
void assign(
   size_type _Count,
   const Type& _Val
);
template<class InputIterator>
   void assign(
      InputIterator _First,
      InputIterator _Last
   );
```

其变量的定义为：

_First：第一个要拷贝的元素的位置。

_Last：最后一个要拷贝的元素的位置。

_Count：插入这个向量的元素个数。

_Val：插入这个向量的元素的值。

现给出该函数应用的一个实例：

```
#include <vector>
#include <iostream>

int main()
{
   using namespace std;
```

```
    vector <int> v1;
    vector <int>::iterator Iter;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );
    v1.push_back( 40 );
    v1.push_back( 50 );

    cout << "v1 = " ;
    for ( Iter = v1.begin(); Iter != v1.end(); Iter++ )
       cout << *Iter << " ";
    cout << endl;

    v1.assign( v1.begin() + 1, v1.begin() + 3 );
    cout << "v1 = ";
    for ( Iter = v1.begin() ; Iter != v1.end() ; Iter++ )
       cout << *Iter << " ";
    cout << endl;

    v1.assign( 7, 4 ) ;
    cout << "v1 = ";
    for ( Iter = v1.begin() ; Iter != v1.end() ; Iter++ )
       cout << *Iter << " ";
    cout << endl;
}
```

该程序输出结果为：

```
v1 = 10 20 30 40 50
v1 = 20 30
v1 = 4 4 4 4 4 4 4
```

擦除函数：在指定位置删除元素，其声明如下：

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
```

其变量的定义为：

_Where：指定被删除元素的位置。

_First：第一个删除元素之前的那个元素的位置。

_Last：最后一个被删除元素的位置。

现给出该函数应用的一个实例：

```
#include <vector>
#include <iostream>

int main()
```

```
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;
   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );
   v1.push_back( 40 );
   v1.push_back( 50 );
   cout << "v1 =" ;
   for ( Iter = v1.begin() ; Iter != v1.end() ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin() );
   cout << "v1 =";
   for ( Iter = v1.begin() ; Iter != v1.end() ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
   v1.erase( v1.begin() + 1, v1.begin() + 3 );
   cout << "v1 =";
   for ( Iter = v1.begin() ; Iter != v1.end() ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

输出结果为：

```
v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50
```

通过上面的实例对向量的使用基本已经有一个相当的认识，表 3-1 给出向量使用的方法函数。

表 3-1　向量的方法函数

| 方法名 | 含义 |
|---|---|
| assign | 擦除向量，并指定元素到空向量 |
| at | 返回向量指定位置的地址的值 |
| back | 返回向量末端的地址的值 |
| begin | 返回指向第一个元素随机进行入迭代器 |
| capacity | 向量现有占用的内存 |
| clear | 擦除向量中的所有元素 |
| empty | 测试向量容器是否为空 |
| end | 返回指向最后一个元素随机进行入迭代器 |
| erase | 在指定位置删除元素 |
| front | 返回向量首端的地址的值 |
| get_allocator | 返回一个分配器类的对象 |
| insert | 在指定位置插入一个元素或一定长度的元素 |

续表

| 方法名 | 含义 |
| --- | --- |
| max_size | 返回向量的最大长度 |
| pop_back | 删除向量的末尾的元素 |
| push_back | 在向量的末端添加元素 |
| rbegin | 返回指向翻转向量的第一个元素的迭代器 |
| rend | 返回指向翻转向量的最后一个元素的迭代器 |
| resize | 指明新向量的值 |
| reserve | 向量对象的最小内存的长度 |
| size | 返回向量元素的个数 |
| swap | 交换两个向量的元素 |
| vector | 向量的构造函数 |

2. 链表（list）

STL 链表类是一个容器模板类，可以有效地实现任意位置的元素的插入和删除。该容器是以双向链表数据结构存储元素，其定义如下：

```
template <
   class Type,
   class Allocator=allocator<Type>
>
class list
```

变量定义如下：

Type：存储在链表中的数据的类型。

Allocator：存储分配器对象，用以了解链表的分配和释放内存情况。这个变量是可选的，其默认值是 allocator<Type>。

构造函数：进行模板类的构造和初始化，由于 list 具有多个构造函数，其声明如下：

```
list();
explicit list(
   const Allocator& _Al
);
explicit list(
   size_type _Count
);
list(
   size_type _Count,
   const Type& _Val
);
list(
   size_type _Count,
   const Type& _Val,
   const Allocator& _Al
);
list(
   const _list& _Right
```

```
);
template<class InputIterator>
   list(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator >
   list(
      InputIterator _First,
      InputIterator _Last,
      const Allocator& _Al
   );
```

其变量的定义为：

_Al：分配器用于该对象，get_allocator 将返回分配器对象。

_Count：链表的初始长度。

_Val：链表的初始值。

_Right：通过拷贝构造链表。

_First：拷贝的第一个值的位置。

_Last：拷贝的最后一个值的位置。

下面给出该函数的一个使用实例：

```
#include <list>
#include <iostream>

int main()
{
   using namespace std;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

   // 创建一个新链表 c0
   list <int> c0;

   //创建一个新链表 c1，该链表有 3 个元素，并赋予其初始值为默认值 0
   list <int> c1( 3 );

   //创建一个新链表 c2，该链表有 5 个元素，并赋予其初始值为 2
   list <int> c2( 5, 2 );

  //创建一个新链表 c3，该链表有 3 个元素，并赋予其初始值为 1 和 c2 的分配器
   list <int> c3( 3, 1, c2.get_allocator() );

   // 创建 c2 的拷贝链表 c4
   list <int> c4(c2);

   //创建 c4 的第一个元素到第二个元素的拷贝链表 c5
   c4_Iter = c4.begin();
   c4_Iter++;
   c4_Iter++;
```

```
    list <int> c5( c4.begin(), c4_Iter );

    //  创建 c4 的第一个元素到第三个元素的拷贝链表 c6，并使用相同的分配器
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    c4_Iter++;
    list <int> c6( c4.begin(), c4_Iter, c2.get_allocator() );

    cout << "c1 =";
    for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
       cout << " " << *c1_Iter;
    cout << endl;

    cout << "c2 =";
    for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;

    cout << "c3 =";
    for ( c3_Iter = c3.begin(); c3_Iter != c3.end(); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;

    cout << "c4 =";
    for ( c4_Iter = c4.begin(); c4_Iter != c4.end(); c4_Iter++ )
       cout << " " << *c4_Iter;
    cout << endl;

    cout << "c5 =";
    for ( c5_Iter = c5.begin(); c5_Iter != c5.end(); c5_Iter++ )
       cout << " " << *c5_Iter;
    cout << endl;

    cout << "c6 =";
    for ( c6_Iter = c6.begin(); c6_Iter != c6.end(); c6_Iter++ )
       cout << " " << *c6_Iter;
    cout << endl;
}
```

输出结果为：

```
c1 = 0 0 0
c2 = 2 2 2 2 2
c3 = 1 1 1
c4 = 2 2 2 2 2
c5 = 2 2
c6 = 2 2 2
```

list::sort()函数：用以对链表里的元素进行排序，其函数声明如下：

```
void sort();
```

```
template<class Traits>
   void sort(
      Traits _Comp
   );
```

变量_Comp 是用以比较的运算符。

下面是该函数使用的一个实例：

```
#include <list>
#include <iostream>

int main()
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 20 );
   c1.push_back( 10 );
   c1.push_back( 30 );

   cout << "Before sorting: c1 =";
   for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort();
   cout << "After sorting c1 =";
   for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( greater<int>() );
   cout << "After sorting with 'greater than' operation, c1 =";
   for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

最后的输出结果如下：

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

list::splice：移除链表中的元素，并插入另一个链表中，下面是其函数实现的实例：

```
void splice(
   iterator _Where,
   list<Allocator>& _Right
);
void splice(
   iterator _Where,
   list<Allocator>& _Right,
```

```
   iterator _First
);
void splice(
   iterator _Where,
   list<Allocator>& & _Right,
   iterator _First,
   iterator _Last
);
```

变量定义如下：

_Where：指定被删除元素的位置。

_First：第一个删除元素之前的那个元素的位置。

_Last：最后一个被删除元素的位置。

_Where：目标链表需要插入元素的位置。

_Right：需要插入目标链表的源链表。

_First：确定要插入目标链表的左边的范围。

_Last：确定要插入目标链表的右边的范围。

下面是该函数的实例：

```
#include <list>
#include <iostream>

int main()
{
   using namespace std;
   list <int> c1, c2, c3, c4;
   list <int>::iterator c1_Iter, c2_Iter, w_Iter, f_Iter, l_Iter;

   c1.push_back( 10 );
   c1.push_back( 11 );
   c2.push_back( 12 );
   c2.push_back( 20 );
   c2.push_back( 21 );
   c3.push_back( 30 );
   c3.push_back( 31 );
   c4.push_back( 40 );
   c4.push_back( 41 );
   c4.push_back( 42 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
   for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
```

```
    w_Iter = c2.begin();
    w_Iter++;
    c2.splice( w_Iter,c1 );
    cout << "After splicing c1 into c2: c2 =";
    for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
      cout << " " << *c2_Iter;
    cout << endl;

    f_Iter = c3.begin();
    c2.splice( w_Iter,c3, f_Iter );
    cout << "After splicing the first element of c3 into c2: c2 =";
    for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
      cout << " " << *c2_Iter;
    cout << endl;

    f_Iter = c4.begin();
    l_Iter = c4.end();
    l_Iter--;
    c2.splice( w_Iter,c4, f_Iter, l_Iter );
    cout << "After splicing a range of c4 into c2: c2 =";
    for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
      cout << " " << *c2_Iter;
    cout << endl;
  cout << "c1 =";
    for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
    cout << endl;
  }
```

输出结果如下：

```
c1 = 10 11
c2 = 12 20 21
After splicing c1 into c2: c2 = 12 10 11 20 21
After splicing the first element of c3 into c2: c2 = 12 10 11 30 20 21
After splicing a range of c4 into c2: c2 = 12 10 11 30 40 41 20 21
c1 =
```

list::unique 函数：可以移除链表中相邻的同一个元素或者满足其他谓词约束的元素，该函数的声明如下：

```
void unique();
template<class BinaryPredicate>
  void unique(
    BinaryPredicate _Pred
  );
```

变量_Pred 为二值的谓词条件。

下面给出该函数的一个实例：

```
#include <list>
#include <iostream>

int main()
```

```
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter,c3_Iter;
   not_equal_to<int> mypred;

   c1.push_back( -10 );
   c1.push_back( 10 );
   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 20 );
   c1.push_back( -10 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin(); c1_Iter != c1.end(); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.unique();
   cout << "After removing successive duplicate elements, c2 =";
   for ( c2_Iter = c2.begin(); c2_Iter != c2.end(); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   list <int> c3 = c2;
   c3.unique( mypred );
   cout << "After removing successive unequal elements, c3 =";
   for ( c3_Iter = c3.begin(); c3_Iter != c3.end(); c3_Iter++ )
      cout << " " << *c3_Iter;
   cout << endl;
}
```

输出结果如下：

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

通过上面的介绍，应该对链表模板类有了一个了解，表 3-2 给出了其操作方法。

**表 3-2　链表模板类的方法**

| 方法名 | 含义 |
|---|---|
| assign | 擦除链表，并指定到元素到空链表 |
| back | 返回链表末端的地址的值 |
| begin | 返回指向链表第一个元素随机进行入迭代器 |
| clear | 擦除链表中所有元素 |
| empty | 测试链表容器是否为空 |
| end | 返回指向链表最后一个元素随机进行入迭代器 |
| erase | 在链表指定位置删除元素 |

续表

| 方法名 | 含义 |
| --- | --- |
| front | 返回链表首端的地址的值 |
| get_allocator | 返回一个分配器类的对象 |
| insert | 在链表指定位置插入一个元素或一定长度的元素 |
| list | 构造链表 |
| max_size | 返回链表的最大长度 |
| merge | 移除源链表中的元素，插入新链表，并以升序排列新链表中的数据 |
| pop_back | 删除链表的末尾的元素 |
| pop_front | 删除链表的第一个元素 |
| push_back | 在链表的末端添加元素 |
| push_front | 在链表的前端添加元素 |
| rbegin | 返回指向翻转链表的第一个元素的迭代器 |
| remove | 通过匹配一个指定的值擦除链表中的一个元素 |
| remove_if | 通过一个指定的规则擦除链表中的元素 |
| rend | 返回指向翻转链表的最后一个元素的迭代器 |
| resize | 指明新链表的值 |
| reserve | 链表对象的最小内存的长度 |
| size | 返回链表元素的个数 |
| sort | 按照指定的方式进行链表的排序，默认的方式是升序排列 |
| splice | 移除链表中的元素，并插入另一个链表中 |



3. 双向队列（Deque）

STL 双向队列类是一个容器模板类，可以有效地实现任意位置的元素的插入和删除。该容器类似于向量，但是比向量更能支持任一地方的插入和删除。该类的定义如下：

```
template <
   class Type,
   class Allocator=allocator<Type>
>
   class deque
```

变量定义如下：

Type：存储在队列中的数据的类型。

Allocator：存储分配器对象，用以了解队列的分配和释放内存情况。这个变量是可选的，其默认值是 allocator<Type>。

现将其操作方法及其含义列于表 3-3 之中。

表 3-3　链表模板类的方法

| 方法名 | 含义 |
| --- | --- |
| assign | 擦除队列，并指定元素到空队列 |
| at | 返回队列指定位置的地址的值 |
| back | 返回队列末端的地址的值 |

续表

| 方法名 | 含义 |
| --- | --- |
| begin | 返回指向第一个元素随机进行入迭代器 |
| clear | 擦除队列中的所有元素 |
| deque | 队列的构造函数 |
| empty | 测试队列容器是否为空 |
| end | 返回指向最后一个元素随机进行入迭代器 |
| erase | 在指定位置删除元素 |
| front | 返回队列首端的地址的值 |
| get_allocator | 返回一个分配器类的对象 |
| insert | 在指定位置插入一个元素或一定长度的元素 |
| max_size | 返回队列的最大长度 |
| pop_back | 删除队列的末尾的元素 |
| pop_front | 删除队列的前端的元素 |
| push_back | 在队列的末端添加元素 |
| push_front | 在队列之前添加元素 |
| rbegin | 返回指向翻转队列的第一个元素的迭代器 |
| rend | 返回指向翻转队列的最后一个元素的迭代器 |
| resize | 指明新队列的值 |
| size | 返回队列元素的个数 |
| swap | 交换两个队列的元素 |

## 本章小结

设计模式和 STL 都属于面向对象编程中的高级内容，涉及的方面也非常广，是非常深奥的技术，本章只作一个简介性质的讲解，如果读者对这方面比较感兴趣，可以钻研更深奥的内容。在讲完本章之后将讲述游戏中的几何基础。

# 第 4 章
# 三维游戏引擎中的几何基础

**主要内容：**本章主要介绍地形生成过程中所涉及的几何基础及相关技术，首先介绍计算机图形学中所涉及的各种坐标系，然后再介绍通用的几何变换、地形渲染流程和光照纹理等相关技术。

**本章重点：**

- 向量及其运算
- 矩阵及矩阵操作
- 3 种坐标系
- 几何变换
- 四元数

## 4.1 向量及其运算

### 4.1.1 向量的定义

向量（Vector）也称为矢量，简单来说，就是在给定的坐标系（Coordinate System）中描述位置或方向的一组数。在 3D 图形学中，这个坐标系往往是笛卡儿坐标系（Cartesian Coordinates），比如，一个向量可以用(4,5,3)来表示，如图 4-1 所示。向量不同于标量，标量仅有大小，而向量既有大小又有方向。具体在 3D 图形学中，向量可以用来表示粒子的速度、加速度和光线方向等既有大小又有方向的量。总之，向量为在三维空间中表示方向提供了方便。

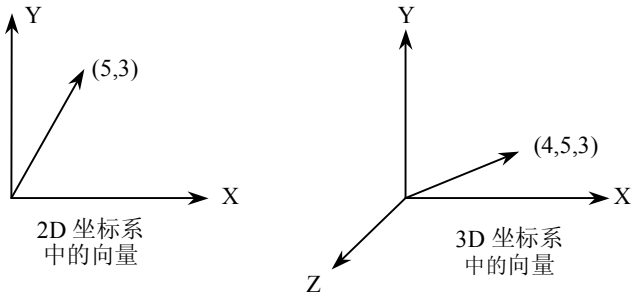


图 4-1　2D 向量和 3D 向量

在 D3DX 库中，用 D3DXVECTOR3 来表示 3D 空间中的三维向量，它是从 D3DVECTOR 继承过来的，D3DVECTOR 结构体定义如以下代码所示：

```
typedef struct D3DVECTOR {
    float x, y, z;
} D3DVECTOR, *LPD3DVECTOR;
```

D3DVECTOR 含有 3 个浮点型变量，分别表示三维向量的一个分量。D3DX 库中对 D3DVECTOR 的派生类 D3DXVECTOR3 的定义如下：

```
typedef struct D3DXVECTOR3 : public D3DVECTOR
{
public:
    // 构造函数
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( CONST D3DXFLOAT16 * );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // 类型转换函数
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // 赋值操作符
    D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );
```

```
    D3DXVECTOR3& operator *= ( FLOAT );
    D3DXVECTOR3& operator /= ( FLOAT );

    // 一元操作符
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // 二元操作符
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;

    // 友元函数操作符
    friend D3DXVECTOR3 operator * ( FLOAT, CONST struct D3DXVECTOR3& );

    // 比较两个向量是否相等操作符
    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;
```



可以看出，与 D3DVECTOR 相比，D3DXVECTOR3 仅仅扩展了许多操作符和函数，便于向量与数、向量与向量之间的数学运算。如果你的 C++学得很好，大概已经看出这些操作符有什么用途了；如果你一头雾水，没有关系，这里将简要讲述上述函数的作用。

前 5 个构造函数允许你从 FLOAT 数组、16 位 FLOAT 数组、D3DVECTOR 向量等创建一个 D3DXVECTOR3 对象。然后是两个类型转换函数，它允许将一个 D3DXVECTOR3 对象直接转换成一个 FLOAT 指针，指向该对象的第一个变量 x。然后重载了 4 个赋值操作符，它们允许你像操作内置数据类型那样操作 D3DXVECTOR3 对象，请看以下代码：

```
D3DXVECTOR3 vec1(1,2,3);
D3DXVECTOR3 vec2(vec1);
D3DXVECTOR3 vec3;
vec3 = vec1 + vec2;
```

执行完毕后，vec2 的大小等于(1,2,3)，vec3 的大小等于(2,4,6)。

然后定义了两个单目运算符，分别为取正“+”和取负操作“-”。请看代码：

```
D3DXVECTOR3 vec4 = - vec2;
```

执行完毕，vec4 的大小为(-1,-2,-3)。

然后定义了 4 个双目运算符，其中加运算符和减运算符允许两个 D3DXVECTOR3 类型向量相加（减），返回一个 D3DXVECTOR3 类型，运算规则为分别将两个向量对应的分量相加（减）。乘运算符和除运算符允许一个 D3DXVECTOR3 向量和一个 FLOAT 类型相乘（除），运算规则为将向量中的每个分量乘（除）以那个 FLOAT 数据，返回一个 D3DXVECTOR3 类型。

上面定义的乘法运算符只允许将一个 D3DXVECTOR3 类型和一个 FLOAT 数据相乘，若是想将一个 FLOAT 数据和一个 D3DXVECTOR3 类型相乘，怎么办呢？这时使用上述的*运算符是错误的，必须另外定义一个友元函数操作符，它含有两个参数；第一个是 FLOAT 数据，第二个是 D3DXVECTOR3 类型向量。

下面给出代码说明作为友元函数操作符的“*”与属于类本身的二元操作符之间的区别：

```
D3DXVECTOR3 vec1(1,2,3);
D3DXVECTOR3 vec2 = vec1 * 2;    // 调用属于类本身的二元操作符
D3DXVECTOR3 vec3 = 2 * vec1;    // 调用友元函数操作符
```

最后两个操作符实现判断两个 D3DXVECTOR3 向量是否相等的功能。这两个操作符都只含有一个 D3DXVECTOR3 参数。只有在每个分量都相等的情况下，才能说明某两个向量是相等的。下面给出代码说明：

```
D3DXVECTOR3 vec1(1,2,3);
D3DXVECTOR3 vec2(vec1);
D3DXVECTOR3 vec3(1,2,4);
ASSERT(vec1 == vec2);
ASSERT(vec1 != vec3);
```

本节仅介绍了 D3DXVECTOR3 类（虽然定义为结构体，但 C++中的结构体与类已经没有很大区别了）的用法，D3DX 数学库中还定义了 D3DXVECTOR2 和 D3DXVECTOR4 向量类，分别是二维向量和四维向量，其用法和 D3DXVECTOR3 都很相似，在此就不详细介绍了。

### 4.1.2 向量的数学运算及在 D3D 中的实现

上一节讲述向量类 D3DXVECTOR3 时，已经介绍了一些简单的向量数学运算，它们都是通过定义在类内部的数学操作符实现的。本节将介绍 D3DX 实现的用于向量数学运算的函数，分别为求向量长度、归一化、点积、叉积等。

1. 向量的长度与归一化

使用向量时，经常遇到的一个问题是计算一个向量的长度。向量的长度在数学上也称为范数（Norm）。从几何上表示，向量的长度是从向量原点到向量终点之间的距离。要计算一个向量的长度，应用以下公式即可：

$$|v|=\sqrt{|v.x|^2+|x.y|^2+|v.z|^2} \tag{4-1}$$

D3DX 库为计算向量长度专门提供了一个函数，其原型为：

```
FLOAT D3DXVec3Length(
  CONST D3DXVECTOR3 * pV
);
```

pV 指向所要求的三维向量，函数返回值即为向量的长度。

知道向量的长度后，即可对其进行归一化（Normalize）处理了，归一化即对向量进行缩放，使其长度为 1.0，并且方向保持不变。对一个向量求归一化，可以使用以下公式：

$$\hat{v}=\frac{v}{|v|}=\frac{v}{\sqrt{|v.x|^2+|v.y|^2+|v.z|^2}} \tag{4-2}$$

D3DX 库为对向量进行归一化也提供了一个函数，其原型为：

```
D3DXVECTOR3 * D3DXVec3Normalize(
  D3DXVECTOR3 * pOut,
  CONST D3DXVECTOR3 * pV
);
```

pV 指向要输入的待归一化处理的向量，归一化之后的向量保存在 pOut 指向的对象中。该函

数还返回一个 D3DXVECTOR3 指针值，该指针值和 pOut 的最终值相等，这样做的好处是该函数可以作为其他函数的参数。

2. 向量的点积

如果你学过线性代数，肯定听过向量的点乘和叉乘的概念，在这里称为点积和叉积。两个向量的点积得到一个数，两个向量的叉积得到一个向量。

点积非常有用，后述章节中光照计算等都要用到这个运算，其运算法则可以用以下公式表示：

$$u \cdot v = u.x \cdot v.x + u.y \cdot v.y + u.z \cdot v.z \tag{4-3}$$

从公式可以看出，点积是将向量的各个分量相乘然后相加，得到一个标量。它还可以用如下公式表示：

$$u \cdot v = |u| \cdot |v| \cdot \cos\theta \tag{4-4}$$

其中 $\theta$ 为两个向量的夹角，它说明点积还意味着它不仅与向量的长度有关，还与两个向量的夹角有关。如果两个向量是垂直的，它们的点积一定为 0。那么可以用点积判断两个向量是否垂直（这个非常重要，特别在游戏编程中经常用到）。

组合公式（4-3）和公式（4-4）还可以发现，任意给定两个向量，可以很容易地求出它们之间的夹角，求夹角的公式为：

$$\theta = \cos^{-1}\left(\frac{u \cdot v}{|u| \cdot |v|}\right) = \cos^{-1}\left(\frac{u.x \cdot v.x + u.y \cdot v.y + u.z \cdot v.z}{|u| \cdot |v|}\right) \tag{4-5}$$

这是一个功能非常强大的工具，也是很多 3D 图形学算法的基础。下面给出向量的点积与角度之间的定性规律，这些规律是非常有用的：

- 如果向量 $u$ 和 $v$ 之间的夹角为 90 度（相互垂直），则 $u \cdot v = 0$。
- 如果向量 $u$ 和 $v$ 之间的夹角大于 90 度（钝角），则 $u \cdot v < 0$。
- 如果向量 $u$ 和 $v$ 之间的夹角小于 90 度（锐角），则 $u \cdot v > 0$。
- 如果向量 $u$ 和 $v$ 之间的夹角为 0 度（相互平行），则 $u \cdot v = |u| \cdot |v|$。

点积还可以用于其他许多运算。在计算机图形学和游戏编程中，要经常计算一个向量在另一个向量上的投影分量大小。假设有一个向量 $v$，它代表游戏中某个角色的运动轨迹，还有另一个向量 $u$，它代表另一个角色的运动轨迹。很多情况下，需要知道 $u$ 在 $v$ 方向上的分量（$\text{Proj}_v(u)$），这时可以使用点积来完成这种计算。

一般来说，计算 $u$ 在 $v$ 上的投影向量的公式如下：

$$\text{Proj}_v(u) = \frac{(u \cdot v) * v}{|v| * |v|} \tag{4-6}$$

特别地，如果 $v$ 为单位向量，则公式可简化为：

$$\text{Proj}_v(u) = (u \cdot v) * v \tag{4-6}$$

点积满足交换律，即有：

$$u \cdot v = v \cdot u \tag{4-7}$$

D3DX 库为向量点积运算提供了一个函数，其原型为：

```
FLOAT D3DXVec3Dot(
  CONST D3DXVECTOR3 * pV1,
  CONST D3DXVECTOR3 * pV2
);
```

函数形式很简单，输入两个向量指针，函数返回它们的点积值，返回值类型为 FLOAT。

3. 向量的叉积

另一种向量乘法是叉积，这个概念也许是最难理解的。两个向量的叉积得到第三个向量，这个向量与原始两个向量相垂直，其大小定义如下（叉积符号定义为×）：

$$|u \times v| = |u| * |v| * \sin\theta \tag{4-8}$$

其中 $\sin\theta$ 为两个向量夹角的正弦值。要计算两个向量叉积的大小，可以建立以下矩阵：

$$u \times v = \begin{vmatrix} i & j & k \\ u.x & u.y & u.z \\ v.x & v.y & v.z \end{vmatrix} \tag{4-9}$$

假设 $n = u \times v$，那么：

$$\begin{aligned} n.x &= u.y * v.z - u.z * v.y \\ n.y &= -(u.x * v.z - u.z * v.x) \\ n.z &= u.x * v.y - u.y * v.x \end{aligned} \tag{4-10}$$

前面已经讲过，两个向量的叉积得到的向量与原来的向量都垂直，这在计算表面的法线向量时非常有用。

注意，叉积不满足交换律，但交换顺序的结果仅仅是改变了叉积结果的符号，即：

$$u \times v = -v \times u \tag{4-11}$$

D3DX 库为向量叉积运算提供了一个函数，其原型为：

```
D3DXVECTOR3 * D3DXVec3Cross(
  D3DXVECTOR3 * pOut,
  CONST D3DXVECTOR3 * pV1,
  CONST D3DXVECTOR3 * pV2
);
```

pV1 和 pV2 两个指针为输入参数，它们保存了两个原始向量的值，叉积所求得的向量保存在 pOut 中。函数还有一个返回值，其值和 pOut 相等，作用是使得 D3DXVec3Cross 函数返回值可以作为一个实参传递给其他函数。

## 4.2 矩阵及矩阵操作

正如以前提到过的，使用矩阵，可以旋转、缩放或移动顶点（进而对整个对象也能做这些变换）。本章会看到怎样按照不同的方法来旋转 5 个立方体：绕 X 轴、绕 Y 轴、绕 Z 轴、绕自定义轴和绕所有其他轴。

在讲述几何变换的具体含义之前，先介绍几何变换的形象描述与它在图形渲染中的作用。当游戏角色在地图上走动时，可以理解为平移（Translation）；当怪物的手臂转动时，可以理解为旋转（Rotation）；假如你的武器像孙悟空的金箍棒一样无限放大，那么你需要缩放（Scaling）。

### 4.2.1 矩阵的概念及其工作原理

矩阵其实是一个高级的数学主题（参看《线性代数》），所以在此只能尽力地、简要地概括一下。矩阵就像是一个表格，有一定数目的行与列；每个格子中都一个数字或表达式。通过特定的矩阵，可以对 3D 对象中的顶点坐标进行运算，来实现类似移动、旋转、缩放这样的操作。在 DirectX

中，矩阵就是 4×4 的二维数组。图 4-2 所示是一个矩阵的例子，这个矩阵能使一个对象（由它的所有顶点）缩放到原来的 5 倍。

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 4-2　矩阵示例

在使用 Direct3D 编程的时候，使用 4×4 的矩阵和 1×4 的行向量来进行各种转换，这么做的想法是这样的：使用一个 4×4 的矩阵 X 来表示一个特定的转换。把一个点的信息放到一个 1×4 的行向量 *v* 里面，那么这两个的积即 *v*×X 的结果就是新的转换出来的向量 *v*'。比如，如果 X 表示一个单位为 10 的平移转换矩阵，*v*=[2,6,-3,1]，那么转换出来 *v*X=[12,6,-3,1]。

也许，你会觉得 3×3 的矩阵更加适合转换，但是你要清楚，只有 4×4 矩阵才能更加适合我们来描述所有的转换。有很多的转换使用 3×3 矩阵是不合适的，如透视投影、平移转换等。注意，使用的是向量矩阵积的形式来进行转换，所以只有把向量增大成 1×4 来和矩阵达成一致，因为 1×3 的向量和 4×4 的矩阵相乘是不合法的。

既然多了一个元素（一般标记它为 w），那么使用第四个元素干什么呢？在表示一个点的时候，把这个 w 设为 1，这样就可以很好地进行转换了。因为向量没有位置的概念，那么向量也就没有平移的概念，任何对向量进行的平移操作就没有意义。为了防止对向量进行平移操作，在进行平移操作的时候，把 1×4 的行向量的 w 设置为 0。比如，在表示一个点的时候，*P*(p1,p2,p3) 对应的向量就应该是 *P*(p1,p2,p3,1)，而在表示一个向量的时候（注意这个向量不表示点），那么就把 w 设置为 0，像 *V*(v1,v2,v3)变为 1×4 的向量就是 *V*(v1,v2,v3,0)。

明白了吗？（似乎越说越糊涂），那么我再次说清楚一点。只有点才可以进行转换操作，所以，如果一个向量表示一个点，那么就设 w=1，如果不是点，而是一般的向量，比如表示一个方向，那么设 w=0，这样，进行转换的时候只要看一下 w 就知道能不能转换了。

**注意**：一个 1×4 的行向量既可以表示一个点，也可以表示一个向量，那么使用术语“向量”的时候，指的可能是点，也可能是向量。

那么你一定很奇怪为什么要用 4×4 矩阵而不用 3×3 矩阵进行各种转换呢？主要是 3×3 矩阵只能表示线性变换，而不能表示类似平移转换。比如一个点 p[0,0,0]，对它进行平移一个单位，怎么使用矩阵乘来做呢？没有办法了吧？使用 4×4 矩阵就可以解决了。

那么，矩阵是怎样改变顶点位置的呢？要改变一个顶点的 *x*、*y* 与 *z* 值，应该把它们与某个矩阵相乘。如图 4-3 演示了顶点是怎样与矩阵相乘的。

这其实是一种简单的计算，只需要将 *x*、*y* 和 *z* 值分别与每列上的数相乘再相加，每列上得出的数都是新顶点的一个坐标值，这在图 4-3 中是显而易见的。你可能已经注意到上面的图例中在当前顶点（Current Vertex）的 *z* 值后面还有一个“1”，那是为了向矩阵的运算提供可行性和方便性（最好参看一下相关的数学资料）。

所以，只需要操作矩阵就能完成这些类似旋转、缩放或移动的变换。幸运的是，DirectX 提供了一些函数，能方便地生成一些通常的矩阵。那么，怎样完成既缩放又旋转（复合变换）的变换呢？首先需要两个矩阵：一个用来旋转，一个用来缩放；然后，把两个矩阵相乘，得出一个新的复合矩阵，既缩放又旋转的矩阵；然后利用这个新的矩阵来变换顶点。应该注意的是，矩阵相

乘并不是普通的乘法，而且也不满足交换率：“矩阵 *A*×矩阵 *B*”和“矩阵 *B*×矩阵 *A*”是不相等的。图 4-4 演示了怎样把两个矩阵相乘：要把两个矩阵相乘，需要把第一个矩阵的每一行和第二个矩阵的每一列都相乘。在上面的例子中，把第一个矩阵的第一行的每个元素与第二个矩阵的第一列的对应元素相乘，然后把得出的 4 个结果相加，按此方法，得出了新矩阵的第一行的 4 个元素（Column1～Column 4），其他元素的计算方法依此类推。这可能看起来有些复杂，不过就像上面所提到的，DirectX 提供了一些矩阵操作的函数，所以不要对此太担心。

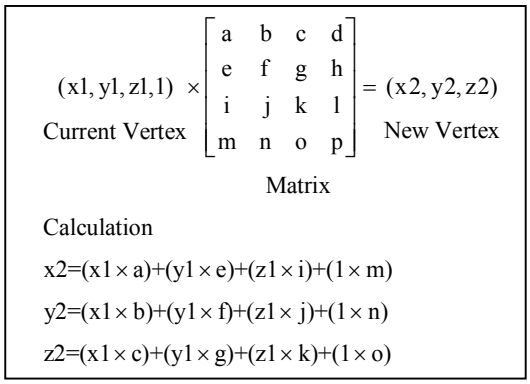


图 4-3　矩阵和矢量（顶点位置）相乘

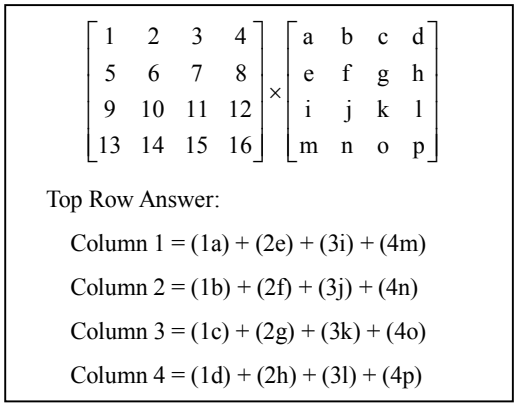


图 4-4　矩阵乘法

D3DX 库提供了 D3DXMATRIX 这个类，它是从 D3DMATRIX 派生过来的，这里给出它们的定义：

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;
```

```
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:
    D3DXMATRIX() {};
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( CONST D3DXFLOAT16 * );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
              FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
              FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
              FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );


    // 入口
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT  operator () ( UINT Row, UINT Col ) const;

    // 类型转换
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // 赋值操作符
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );

    // 取正、取反操作符
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    // 算术操作符
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;

    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );

    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;

} D3DXMATRIX, *LPD3DXMATRIX;
```

D3DXMATRIX 类提供了许多运算符重载，其中入口操作符使得可以像使用数组一样访问其中的元素。比如，想把矩阵中的元素_11 的大小设置成 5，那么只需要如下这样写即可：

```
D3DXMATRIX M;
```

```
M(0, 0) = 5.0f;     // 设置入口 ij 为 11 ～ 5.0f
```

这个类还提供了 5 个赋值运算符，其中“*=”既允许将一个矩阵和一个矩阵相乘，也允许将一个矩阵和一个数相乘。

比如要计算两个矩阵相乘，只要这么做：

```
D3DXMATRIX A(…);            // 初始化 A
D3DXMATRIX B(…);            // 实始化 B
D3DXMATRIX C = A * B;       // C = AB
```

还有很多算术运算符和取正、取反操作符，其原理和 D3DXVECTOR3 类都差不多，这里就不详细叙述了。读者如果有兴趣可以查看 d3dx9math.h 和 d3dx9math.inl 文件是怎样定义和实现它的。

### 4.2.2 矩阵相乘

为了将多个矩阵变换的效果综合在一起，必须将矩阵连接起来。矩阵的连接是通过相乘来实现的。矩阵相乘，必须具备一定的条件，并不是任意两个矩阵都可以相乘。比如两个矩阵，一个是 $m \times n$ 矩阵 $M$，另一个是 $n \times l$ 矩阵 $N$，这两个矩阵才可以相乘，相乘之后得到一个 $m \times l$ 矩阵。一般来说，矩阵相乘需要满足的条件是：被乘矩阵的列数应该等于乘矩阵的行数，就是第一个矩阵的列数应该等于第二个矩阵的行数。

**注意**：矩阵相乘的顺序是至关重要的。例如，先平移后缩放与先缩放后平移绝对是不一样的，平移的因子应用到缩放中去了，而后者平移因子并没有应用到缩放中去。一般来说，先将物体搬到坐标系的原点进行缩放，然后才进行平移；否则，缩放之后的物体形状将和原始物体不一样。

如果先对物体用矩阵 $S$ 来放缩，然后用矩阵 $T$ 对之平移，那么使用下面的矩阵：

$$M = T \cdot S \tag{4-12}$$

在这里 $M$ 只能作为左乘因子对列向量进行变换，如果对行向量进行处理，结果刚好相反（为什么？仔细想一下线性代数中的矩阵知识吧）。

你只需要记住，如果对 3D 物体进行变换但是变换结果并没有按期望的方式表现出来，那么很可能是将矩阵乘法的顺序搞错了，只要试着将顺序颠倒过来，再检验结果是否正确即可。

D3DX 库中将两个矩阵相乘得到另一个矩阵的函数是：

```
D3DXMATRIX * D3DXMATRIXMultiply(
  D3DXMATRIX * pOut,
  CONST D3DXMATRIX * pM1,
  CONST D3DXMATRIX * pM2
);
```

在这里，$pOut = pM1 \times pM2$。

### 4.2.3 矩阵的求逆

矩阵求逆相当于矩阵变换的逆操作。比如将物体从 $A$ 处平移到 $B$ 处称为变换 $T_1$，从 $B$ 处移到 $A$ 处称为变换 $T_2$，那么 $T_1$ 和 $T_2$ 这两个变换互逆，变换表示的矩阵互为逆矩阵。从数学上说，如果

$$T_1 \times T_2 = I \tag{4-13}$$

$I$ 为单位矩阵，则称 $T_1$ 和 $T_2$ 互逆，一般 $T_1$ 的逆矩阵用 $T_1^{-1}$ 表示。现在来讨论平移矩阵、旋转矩阵和缩放矩阵的逆矩阵的求法。

给定平移矩阵：

$$M_T = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-14}$$

则它的逆矩阵为：

$$M_T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -a \\ 0 & 1 & 0 & -b \\ 0 & 0 & 1 & -c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-15}$$

即求平移矩阵的逆矩阵，只需要将第四列的前 3 个数的符号取反即可（不信读者可以试试将这两个矩阵相乘，看结果是否为单位矩阵）。

给定旋转矩阵：

$$M_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-16}$$

要求旋转矩阵的逆矩阵，只需要将 $\theta$ 的符号取反即可（试想，将角度 $\theta$ 的符号取反是不是意味着向相反的方向旋转角度 $\theta$，那么不是回到原始位置了吗？）。又由于 $\cos(-\theta)=\cos(\theta)$，$\sin(-\theta)=-\sin(\theta)$，那么它的逆矩阵为：

$$M_X^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-17}$$

缩放矩阵的逆矩阵就更容易求解了，只需要将缩放因子求倒数即可。

给定缩放矩阵：

$$M_S = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-18}$$

其逆矩阵为：

$$M_S^{-1} = \begin{bmatrix} 1/a & 0 & 0 & 0 \\ 0 & 1/b & 0 & 0 \\ 0 & 0 & 1/c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-19}$$

平移、旋转和缩放 3 种矩阵都是特殊矩阵，其逆矩阵求解相对简单。然而，任意给定一个矩阵，对其求逆就不是那么简单了。任意矩阵求逆的具体求解方法请参考有关数学书籍，这里仅给出 D3DX 库中的矩阵求逆函数：

```
D3DXMATRIX * D3DXMATRIXInverse(
  D3DXMATRIX * pOut,
  FLOAT * pDeterminant,
  CONST D3DXMATRIX * pM
);
```

其中 *pDeterminant* 参数一般不用关心，设置成 NULL 即可。*pM* 为输入矩阵，求逆后的矩阵保存在 *pOut* 指向的内存单元中。

## 4.3 坐标系介绍

在三维地形生成的过程中将涉及 3 种坐标系统，包括模型坐标系、世界坐标系、视图坐标系。

### 4.3.1 模型坐标系（Model Coordinates）

模型坐标系又称为建模空间，这是定义物体的三角形列的坐标系。模型坐标系简化了建模的过程。在物体自己的坐标系中建模比在世界坐标系中直接建模更容易。例如，在模型坐标系中建模不像在世界坐标系中要考虑本物体相对于其他物体的位置、大小、方向关系。

### 4.3.2 世界坐标系（World Coordinates）

一旦构造了各种模型，它们都在自己的模型坐标系中，但是我们需要把它们都放到同一个世界坐标系中。物体从自身坐标系到世界坐标系的变换叫做世界变换。世界变换通常是用平移、旋转、缩放操作来设置模型在世界坐标系中的位置、大小、方向。世界变换就是通过各物体在世界坐标系中的位置、大小和方向等相互之间的关系来建立所有物体。

### 4.3.3 视图坐标系（View Coordinates）

世界坐标系中的几何图与摄像机是相对于世界坐标系而定义的。然而在世界坐标系中当摄像机任意放置和定向时，投影和其他一些操作会变得困难或低效。为了使事情变得更简单，将摄像机平移变换到世界坐标系的原点并把它的方向旋转至朝向 $Z$ 轴的正方向，当然，世界坐标系中的所有物体都将随着摄像机的变换而作相同的变换。这个变换就叫做视图坐标系变换，得到的新坐标系叫视图坐标系。

## 4.4 几何变换

在可视化的过程中，通常需要以某种方式对一系列的向量进行变换。通常用到的变换包括平移、缩放和旋转。对一组点的平移，只要将每个点简单地加上一个偏移向量即可实现。统一缩放（即等量地缩放向量的每一个坐标）可以通过数量乘积来实现。非统一缩放、旋转以及其他更复杂的变换则需要利用矩阵乘积来实现。

### 4.4.1 通用变换

通常可将 $n \times n$ 可逆矩阵 $M$ 看成是从一个坐标系到另一个坐标系的变换矩阵。$M$ 的列给出了坐标轴从原坐标系到新坐标系的映射。例如，假设 $M$ 是一个 $n \times n$ 可逆矩阵，当 $M$ 与向量 $(1,0,0)^T$ 、

$(0,1,0)^T$ 和 $(0,1,0)^T$ 相乘时，可以得到：

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} M_{11} \\ M_{21} \\ M_{31} \end{bmatrix}$$

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} M_{11} \\ M_{21} \\ M_{31} \end{bmatrix} \tag{4-20}$$

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} \\ M_{21} \\ M_{31} \end{bmatrix}$$

类似地，$M^{-1}$ 的列给出了坐标轴从新坐标系列原坐标系的映射。这样，对于任意给定的线性无关的向量 $U$、$V$ 和 $W$，可以构造一个变换矩阵，该矩阵将这些向量映射到向量 $(1,0,0)^T$ 、$(0,1,0)^T$ 和 $(0,1,0)^T$ 。由 $U$、$V$ 和 $W$ 作为列的矩阵的逆矩阵就具有这种性质：

$$\begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} U_x \\ U_y \\ U_z \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} U_x \\ U_y \\ U_z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{4-21}$$

$$\begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} U_x \\ U_y \\ U_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

多个变换可以串联起来，并且可以将多个变换矩阵的乘积用一个矩阵来表示。假设需要先用矩阵 $M$ 后用矩阵 $G$ 对一个对象进行变换，由于矩阵乘积满足结合律，对于任意向量 $P$ 都有 $G(MP)=(GM)P$，因此只需要存储 $GM$ 得到的矩阵，将该矩阵作为对象的变换矩阵即可。这使得可以对顶点进行无数次的变换，而不需要额外的存储空间和计算开销。

### 4.4.2 缩放变换和旋转变换

用 $a$ 作为系数来缩放向量 $P$，只需要计算 $P'=aP$ 。在三维空间中，这个运算也可以表示为矩阵的乘积：

$$P' = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{4-22}$$

这种缩放称为统一缩放。如果希望在 $x$、$y$ 和 $z$ 轴以不同的值缩放向量，那么可以使用与统一缩放矩阵相似的矩阵，只不过其对角线元素不必都相等。称这种缩放为非统一缩放，它可以表示为如下的矩阵乘积：

$$P' = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{4-23}$$

如果想在 3 个任意轴上进行非统一缩放，则要用到稍微复杂的缩放过程。假设希望以系数 $a$ 沿 $U$ 轴方向，以系数 $b$ 沿 $V$ 轴方向，以系数 $c$ 沿 $W$ 轴方向进行缩放，可以先从坐标系$(U,V,W)$变换到坐标系$(i,j,k)$，然后在$(i,j,k)$坐标系中用等式（4-23）进行缩放运算，最后再还原到$(U,V,W)$坐标系。整个过程可以用下面的矩阵乘积表示：

$$P' = \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{4-24}$$

得到将坐标系绕 $x$、$y$ 或 $z$ 轴旋转 $\theta$ 角的 $3\times3$ 矩阵并不难。这里对绕 A 轴的正角度旋转是这样规定的：当 A 轴指向我们时，所看到的旋转是逆时针方向。

找到二维空间旋转的通用公式。通过变换 $x$ 和 $y$ 的坐标，并将新的 $x$ 坐标取负，即能将位于 $xoy$ 平面上的二维（2D）向量 $P$ 进行 90° 的逆时针旋转。设旋转向量为 $Q$，则有 $P=(-P_y,P_x)$。向量 $P$ 和 $Q$ 组成 $xoy$ 平面的一个正交集，因此 $xoy$ 平面上的任何向量 $P$ 以 $\theta$ 角旋转后形成的二维向量 $P'$ 都可以用分别平行 $P$ 和 $Q$ 的分量来表示。根据普通三角学有：

$$P' = P\cos\theta + Q\sin\theta \tag{4-25}$$

这样，可以用下面的公式得到 $P'$ 分量，即：

$$\begin{aligned} P'_x &= P_x\cos\theta + P_y\sin\theta \\ P'_y &= P_y\cos\theta + P_x\sin\theta \end{aligned} \tag{4-26}$$

可以用矩阵的形式改写上面的公式为：

$$P' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} P \tag{4-27}$$

将单位矩阵的第三行和第三列加入到等式（4-26）中的二维旋转矩阵中，就可以将上面的矩阵扩展为绕 $Z$ 轴的三维旋转。该 $3\times3$ 矩阵可以保证绕 $Z$ 轴旋转时向量的 $Z$ 坐标保持不变，这种旋转操作是要经常用到的。这样，绕 $Z$ 轴旋转 $\theta$ 角度的旋转矩阵 $R_z(\theta)$ 可以表示为：

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4-28}$$

同样，可以分别得到绕 $X$ 轴和 $Y$ 轴旋转 $\theta$ 角度的 $3\times3$ 旋转矩阵 $R_x(\theta)$ 和 $R_y(\theta)$：

$$R_x(\theta)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\theta & -\sin\theta\\ 0 & \sin\theta & \cos\theta\end{bmatrix}$$
$$R_y(\theta)=\begin{bmatrix}\cos\theta & 0 & \sin\theta\\ 0 & 1 & 0\\ -\sin\theta & 0 & \cos\theta\end{bmatrix} \quad (4\text{-}29)$$

### 4.4.3 四维变换

其实，可以用一种统一的数学形式来简洁而优雅地表示这些变换。为了做到这一点，需要将向量从三维扩展到四维，并使用 4×4 矩阵来表示变换操作。给 3D 点 $P$ 增加一个坐标，并将这个扩展的第四个坐标（称之为 $\omega$ 坐标）的值设为 1，这样就将 $P$ 从三维扩展到了四维。这里，构造一个 4×4 变换矩阵 $F$，$F$ 对应于 3×3 矩阵 $M$ 和 3D 变换 $T$，如下：

$$F=\begin{bmatrix}M & T\\ 0 & 1\end{bmatrix}=\begin{bmatrix}M_{11} & M_{12} & M_{13} & T_x\\ M_{21} & M_{22} & M_{23} & T_y\\ M_{31} & M_{32} & M_{33} & T_z\\ 0 & 0 & 0 & 1\end{bmatrix} \quad (4\text{-}30)$$

将这个矩阵乘以向量 $(p_x,p_y,p_z,1)$，就等效于对向量的 $x$、$y$、$z$ 坐标进行变换，同时 $w$ 坐标保持为 1。此外，将形如公式（4-29）的两个矩阵相乘，得到矩阵形式仍符合公式（4-30）。

因此，从公式（4-30）中得到 4×4 矩阵 $F$ 的逆矩阵 $F^{-1}$ 为：

$$F^{-1}=\begin{bmatrix}M^{-1} & -M^{-1}T\\ 0 & 1\end{bmatrix} \quad (4\text{-}31)$$

下面的运算验证了上式的正确性。

$$FF^{-1}=\begin{bmatrix}M & T\\ 0 & 1\end{bmatrix}\begin{bmatrix}M^{-1} & -M^{-1}T\\ 0 & 1\end{bmatrix}=\begin{bmatrix}MM^{-1} & M(-M^{-1}T)+T\\ 0 & 1\end{bmatrix}=\begin{bmatrix}I_3 & 0\\ 0 & 1\end{bmatrix} \quad (4\text{-}32)$$

通常可以将 $n\times n$ 可逆矩阵 $M$ 看成是从一个坐标系到另一个坐标系的变换矩阵。

### 4.4.4 坐标变换

1. 世界坐标系到视图坐标系的变换

在地形几何模型建立完成后，进一步的工作是将地形描述转换到视图坐标系中。对象描述的转换等价于将视图坐标系叠加到世界坐标系的一连串变换。可以用 4.2 节中所描述的变换方法来实现这个转换，具体实现步骤是：

平移视图坐标系原点到世界坐标系原点。进行旋转，分别让 $x_{view}$、$y_{view}$ 和 $z_{view}$ 轴对应到世界坐标的 $x_w$、$y_w$ 和 $z_w$ 轴。如果指定世界坐标点 $p=(x_0,y_0,z_0)$ 为视图坐标系原点，则将视图坐标系原点移到世界坐标系原点的变换是：

$$T=\begin{bmatrix}1 & 0 & 0 & -x_0\\ 0 & 1 & 0 & -y_0\\ 0 & 0 & 1 & -z_0\\ 0 & 0 & 0 & 1\end{bmatrix} \quad (4\text{-}33)$$

将视图坐标系叠加到世界坐标系的组合旋转变换矩阵使用单位向量 $u$、$v$、$n$ 来形成。该变换矩阵为：

$$R=\begin{bmatrix} u_x & u_y & u_z & 0 \\ u_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-34}$$

这里，矩阵 $R$ 的元素是 $uvn$ 轴向量的分量。

将前面的平移和旋转矩阵乘起来获得坐标变换矩阵：

$$M_{wc,vc}=R\cdot T=\begin{bmatrix} u_x & u_y & u_z & -u\cdot p0 \\ u_x & v_y & v_z & -v\cdot p0 \\ n_x & n_y & n_z & -n\cdot p0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-35}$$

该矩阵中的平移因子按 $u$、$v$、$n$ 和 $p_0$ 的向量点积计算而得，$p_0$ 代表从世界坐标系原点到视图坐标系原点的向量。换句话来说，平移因子是每一轴上的负投影（视图坐标系中的负分量 $p_0$ ）。这些矩阵元素的取值为：

$$\begin{aligned} -u\cdot p_0 &= -x_0u_x - y_0u_y - z_0u_z \\ -v\cdot p_0 &= -x_0v_x - y_0v_y - z_0v_z \\ -n\cdot p_0 &= -x_0n_x - y_0n_y - z_0n_z \end{aligned} \tag{4-36}$$

矩阵 $M_{wc,vc}$ 将世界坐标系中的地形描述转换到视图坐标系中。

2. 视图坐标系坐标投影变换成二维图像坐标

三维地形投影到平面上一般采用透视投影，其原理是通过观察平面上指定一个矩形裁剪窗口可得到一个观察体。观察体的底面、顶面和侧面是通过窗口边线相交于投影参考点的平面。这形成一个顶点在投影中心的无限矩形棱锥，添加垂直于 $z_{view}$ 轴且和观察平面平行的前后平面后，形成了一个棱台观察体。棱台观察体由视野范围 $fov$ 和前剪切面距离观察点的远近 $D$ 来定义，如图 4-5 所示。

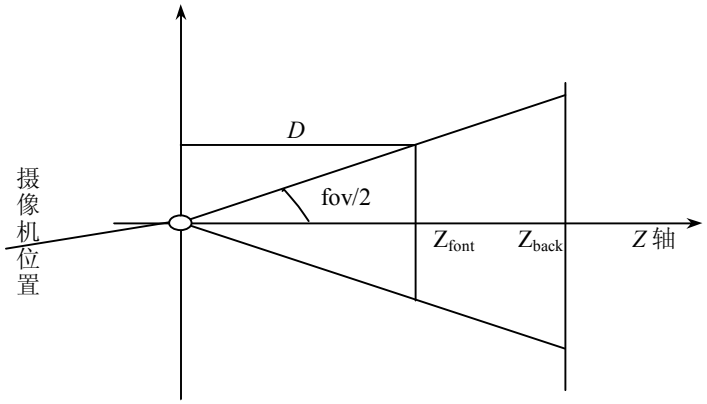


图 4-5　棱台观察体的平面示意图

用变换方法来实现这个转换，具体实现步骤如下：

（1）以前截面的中心为原点可以得到投影矩阵为：

$$M_1=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&\dfrac{1}{D}\\0&0&0&1\end{bmatrix} \tag{4-37}$$

（2）摄像机矩阵通过沿着 $Z$ 轴方向平移到 $-D$ 距离来平移摄像机到原点，平移矩阵如下所示：

$$M_2=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&-D&1\end{bmatrix} \tag{4-38}$$

（3）把这个矩阵和投影矩阵相乘，可以得到不考虑视野范围 $fov$ 和 $Z$ 值深度的混合投影矩阵：

$$M_3=M_2\cdot M_1=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&\dfrac{1}{D}\\0&0&-D&1\end{bmatrix} \tag{4-39}$$

（4）考虑视口的比例，可得投影矩阵：

$$M_4=\begin{bmatrix}w&0&0&0\\0&h&0&0\\0&0&q&1\\0&0&-qz_n&0\end{bmatrix} \tag{4-40}$$

其中，$z_n$ 是近处剪切平面的深度值；$z_f$ 表示远剪切平面的深度值；变量 $w$、$h$ 和 $q$ 的含义如下（注意 $fov_w$ 和 $fov_h$ 表示视口的水平视野和纵向视野）：

$$w=\cot\left(\frac{fov_w}{2}\right)\quad h=\cot\left(\frac{fov_h}{2}\right)\quad q=\frac{z_f}{z_f-z_n} \tag{4-41}$$

3. 二维图像坐标到视口坐标的转换

二维图像坐标到视口坐标的转换是将在屏幕上的东西映射到 2D 显示屏上，可能只需要把它投射到屏幕的某一区域上。投影空间到视口空间的转换矩阵如下：

$$M_{normvoewvol,screen}=\begin{bmatrix}\dfrac{width}{2}&0&0&0\\0&-\dfrac{height}{2}&0&0\\0&0&\max z-\min z&0\\x+\dfrac{width}{2}&\dfrac{height}{2}+y&\min z&1\end{bmatrix} \tag{4-42}$$

## 4.5 3D 编程中的四元数

### 4.5.1 什么是四元数

四元数是数学家 Hamilton 发明的，他将复数从二维空间扩展到四维空间，可以解决三维空间中的许多问题。四元数不是专门为 3D 图形学设计的，但它在 3D 图形学的很多方面确实体现出了相当的优势：

- 3D 相机控制。
- 压缩存储。
- 3D 旋转平滑插值。

四元数是基于复数的，因此首先给出中学时学过的复数的概念。令 $z=a+bi$，其中 $i$ 为虚数的单位，$i^2=-1$。$a$ 为实部，$b$ 为虚部，实部和虚部不能合并，因此可以将复数看成二维平面上的一个点。

将虚部扩展到三维空间，即令：

$$q=w+xi+yj+zk \tag{4-43}$$

其中 $i$、$j$、$k$ 都是虚数单位，可以将它们视为三维坐标系中的 3 个虚轴，它们满足：

$$\begin{aligned} &i^2=j^2=k^2=-1\\ &ij=k,\ ji=-k\\ &jk=i,\ kj=-i\\ &ki=j,\ ik=-j \end{aligned} \tag{4-44}$$

因此，四元数具有 1 个实部、3 个虚部。有时候，还把四元数写成如下形式：

$$q=w+v \tag{4-45}$$

其中 $v=xi+yj+zk$，这样，可以把 $q$ 看成一个实数和一个三维向量的组合（现在，我们离旋转越来越近了，把向量看成三维空间中将要围绕之旋转的轴，把实数看成旋转的角度。怎么样，很清晰了吧，然而并不是这么简单，还要经过一些处理才行！）。

进一步讲述四元数与旋转之间的关系之前，先给出四元数的数学运算。四元数的加减运算很简单，实部与实部相加减，虚部与虚部相加减，这里不花费过多笔墨，需要重点讲述的是四元数的乘法。给出两个四元数：

$$\begin{aligned} q_1&=w_1+x_1i+y_1j+z_1k\\ q_1&=w_2+x_2i+y_2j+z_2k \end{aligned} \tag{4-46}$$

那么 $q_1\times q_2$（类似于向量的叉积，$q_1\times q_2$ 仍然是一个四元数）等于多少呢？下面仅给出结果，中间推导环节省略。

$$\begin{aligned} q_1\times q_2=&w_1w_2-x_1x_2-y_1y_2-z_1z_2\\ &+(w_1x_2+x_1w_2+y_1z_2-z_1y_2)j\\ &+(w_1y_2+y_1w_2+z_1x_2-x_1z_2)j\\ &+(w_1z_2+z_1w_2+x_1y_2-y_1x_2)k \end{aligned} \tag{4-47}$$

将一个矩阵和四元数都看成一个旋转。矩阵和矩阵相乘需要执行 4×4×4 次乘法运算，而四元数和四元数相乘只需要执行 16 次乘法运算。由此看来，要将多个旋转组合起来使用时，采用四元数乘法比采用矩阵乘法所消耗的计算资源更少。这是将四元数应用于旋转的理由之一。

D3DX 库定义了一个 D3DXQUATERNION 类，专门用于四元数的数学运算：

```
typedef struct D3DXQUATERNION
{
#ifdef __cplusplus
public:
    D3DXQUATERNION() {}
    D3DXQUATERNION( CONST FLOAT * );
    D3DXQUATERNION( CONST D3DXFLOAT16 * );
    D3DXQUATERNION( FLOAT x, FLOAT y, FLOAT z, FLOAT w );

    // 自动类型转换
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // 赋值运算符
    D3DXQUATERNION& operator += ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator -= ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator *= ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator *= ( FLOAT );
    D3DXQUATERNION& operator /= ( FLOAT );

    // 取正、取负运算符
    D3DXQUATERNION  operator + () const;
    D3DXQUATERNION  operator - () const;

    // 算术运算符
    D3DXQUATERNION operator + ( CONST D3DXQUATERNION& ) const;
    D3DXQUATERNION operator - ( CONST D3DXQUATERNION& ) const;
    D3DXQUATERNION operator * ( CONST D3DXQUATERNION& ) const;
    D3DXQUATERNION operator * ( FLOAT ) const;
    D3DXQUATERNION operator / ( FLOAT ) const;

    friend D3DXQUATERNION operator * (FLOAT, CONST D3DXQUATERNION& );

    BOOL operator == ( CONST D3DXQUATERNION& ) const;
    BOOL operator != ( CONST D3DXQUATERNION& ) const;

#endif //_cplusplus
    FLOAT x, y, z, w;
} D3DXQUATERNION, *LPD3DXQUATERNION;
```

这个类重载了大量的操作符，其含义与 D3DXVECTOR3 和 D3DXMATRIX 类似，在这里不做过多讲解。

### 4.5.2 四元数表示旋转

给定一个顶点 $P$，它的坐标是$(a,b,c)$，首先将它扩展成四维向量$[0,a,b,c]$，一般来说，这个向量将不会刚好是单位向量。现在要将这个顶点绕着轴 $n$ 旋转一个角度 $\theta$，则可以用一个四元数 $q$ 来表示这个旋转，且 $q$ 的大小为：

$$q = \cos\theta + n \cdot \sin\theta \tag{4-48}$$

其中 $n$ 为一个向量，它不一定必须是一个单位向量（当然最好是单位向量，下面将看到单位四元数有一些特殊性质）。

D3DX 库提供了一个从轴向量和旋转角度构造一个四元数的函数：

```
D3DXQUATERNION * D3DXQUATERNIONRotationAxis(
  D3DXQUATERNION * pOut,
  CONST D3DXVECTOR3 * pV,
  FLOAT Angle
);
```

pV 为旋转轴，Angle 为旋转角度。

那么如何将 $q$ 用于顶点 $P$ 的旋转呢？令 $p=[0,a,b,c]$， $p'$ 为旋转之后的顶点坐标，则有：

$$p' = q^{-1}pq \tag{4-49}$$

若不信，可以验证一下。需要提一下的是，这里的 $q^{-1}$ 表示 $q$ 的倒数，D3DX 库中求一个四元数的倒数的函数是：

```
D3DXQUATERNION * D3DXQUATERNIONInverse(
  D3DXQUATERNION * pOut,
  CONST D3DXQUATERNION * pQ
);
```

特别地，当四元数是一个单位四元数时（意味着构成四元数的向量 $n$ 为单位向量），它的倒数等于它的共轭。四元数的共轭和复数的共轭类似，比如：

$$q^* = (w + xi + yi + zk)^* = w - xi - yj - zk \tag{4-50}$$

四元数的最大优点体现在旋转插值上，比如相机（相机的概念在后续章节讲到，这里理解成三维向量即可），当前朝向为 $n_1$ 时，要将该相机缓缓地旋转到轴 $n_2$ 上。如果使用欧拉角线性插值，发现并不是平滑地沿着最短路径从当前朝向过渡到目标朝向的。使用四元数来解决旋转插值，问题就迎刃而解了。

D3DX 库提供了一个四元数旋转插值函数：

```
D3DXQUATERNION * D3DXQUATERNIONSlerp(
  D3DXQUATERNION * pOut,
  CONST D3DXQUATERNION * pQ1,
  CONST D3DXQUATERNION * pQ2,
  FLOAT t
);
```

它相当于以下公式：

$$q = q_1 + t \cdot (q_2 - q_1) \tag{4-51}$$

## 本章小结

本章主要从向量、矩阵、几何变换、四元数等方面介绍了三维游戏引擎中的几何基础，这些几何基础将为后面的学习提供理论支撑。从下一章开始进行 Direct3D 中图形编程和相关类的封装的学习。