翻译：honghaoz 校对：numbbbbb

模式（Patterns）

本页内容包括：

模式（pattern）代表了单个值或者复合值的结构。例如，元组`(1, 2)`的结构是逗号分隔的，包含两个元素的列表。因为模式代表一种值的结构，而不是特定的某个值，你可以把模式和各种同类型的值匹配起来。比如，`(x, y)`可以匹配元组`(1, 2)`，以及任何含两个元素的元组。除了将模式与一个值匹配外，你可以从合成值中提取出部分或全部，然后分别把各个部分和一个常量或变量绑定起来。

在Swift中，模式出现在变量和常量的声明（在它们的左侧），`for-in`语句和`switch`语句（在他们的case标签）中。尽管任何模式都可以出现在`switch`语句的case标签中，但在其他情况下，只有通配符模式（wildcard pattern），标识符模式（identifier pattern）和包含这两种模式的模式才能出现。

你可以为通配符模式（wildcard pattern），标识符模式（identifier pattern）和元组模式（tuple pattern）指定类型注释，用来限制这种模式只匹配某种类型的值。

模式的语法：

pattern → wildcard-patterntype-annotationopt

pattern → identifier-patterntype-annotationopt

pattern → value-binding-pattern

pattern → tuple-patterntype-annotationopt

pattern → enum-case-pattern

pattern → type-casting-pattern

pattern → expression-pattern

通配符模式（Wildcard Pattern）

通配符模式匹配并忽略任何值，包含一个下划线（_）。当你不关心被匹配的值时，可以使用此模式。例如，下面这段代码进行了`1...3`的循环，并忽略了每次循环的值：

```
        for _ in 1...3 {
            // Do something three times.
        }
```

通配符模式的语法：

wildcard-pattern → _

标识符模式（Identifier Pattern）

标识符模式匹配任何值，并将匹配的值和一个变量或常量绑定起来。例如，在下面的常量申明中，`someValue`是一个标识符模式，匹配了类型是`Int`的`42`。

```
        let someValue = 42
```

当匹配成功时，`42`被绑定（赋值）给常量`someValue`。

当一个变量或常量申明的左边是标识符模式时，此时，标识符模式是隐式

的值绑定模式（value-binding pattern）。

标识符模式的语法：

identifier-pattern → identifier

值绑定模式（Value-Binding Pattern）

值绑定模式绑定匹配的值到一个变量或常量。当绑定匹配值给常量时，用关键字`let`,绑定给变量时，用关键之`var`。

标识符模式包含在值绑定模式中，绑定新的变量或常量到匹配的值。例如，你可以分解一个元组的元素，并把每个元素绑定到相应的标识符模式中。

```
        let point = (3, 2)
        switch point {
            // Bind x and y to the elements of point.
        case let (x, y):
            println("The point is at (\(x), \(y)).")
        }
        // prints "The point is at (3, 2).”
```

在上面这个例子中，`let`将元组模式`(x, y)`分配到各个标识符模式。因为这种行为，`switch`语句中`case let (x, y):`和`case (let x, let y):`匹配的值是一样的。

值绑定模式的语法：

value-binding-pattern → var pattern | let pattern

元组模式（Tuple Pattern）

元组模式是逗号分隔的列表，包含一个或多个模式，并包含在一对圆括号

中。元组模式匹配相应元组类型的值。

你可以使用类型注释来限制一个元组模式来匹配某种元组类型。例如，在常量申明`let (x, y): (Int, Int) = (1, 2)`中的元组模式`(x, y): (Int, Int)`，只匹配两个元素都是`Int`这种类型的元组。如果仅需要限制一个元组模式中的某几个元素，只需要直接对这几个元素提供类型注释即可。例如，在`let (x: String, y)`中的元组模式，只要某个元组类型是包含两个元素，且第一个元素类型是`String`，则被匹配。

当元组模式被用在`for-in`语句或者变量或常量申明时，它可以包含通配符模式，标识符模式或者其他包含这两种模式的模式。例如，下面这段代码是不正确的，因为`(x, 0)`中的元素`0`是一个表达式模式：

```
        let points = [(0, 0), (1, 0), (1, 1), (2, 0),
(2, 1)]
        // This code isn't valid.
        for (x, 0) in points {
            /* ... */
        }
```

对于只包含一个元素的元组，括号是不起作用的。模式匹配那个单个元素的类型。例如，下面是等效的：

```
        let a = 2         // a: Int = 2
        let (a) = 2       // a: Int = 2
        let (a): Int = 2 // a: Int = 2
```

元组模式的语法：

tuple-pattern → (tuple-pattern-element-list opt)

tuple-pattern-element-list → tuple-pattern-element | tuple-pattern-element, tuple-pattern-element-list

tuple-pattern-element → pattern

枚举案例模式（Enumeration Case

Pattern）

枚举案例模式匹配现有的枚举类型的某种案例。枚举案例模式仅在`switch`语句中的`case`标签中出现。

如果你准备匹配的枚举案例有任何关联的值，则相应的枚举案例模式必须指定一个包含每个关联值元素的元组模式。关于使用`switch`语句来匹配包含关联值枚举案例的例子，请参阅`Associated Values`.

枚举案例模式的语法：

enum-case-pattern → type-identifier opt . enum-case-name tuple-pattern opt

类型转换模式（Type-Casting Patterns）

有两种类型转换模式，`is`模式和`as`模式。这两种模式均只出现在`switch`语句中的`case`标签中。`is`模式和`as`模式有以下形式：

```
        is type
        pattern as type
```

`is`模式匹配一个值，如果这个值的类型在运行时（runtime）和`is`模式右边的指定类型（或者那个类型的子类）是一致的。`is`模式和`is`操作符一样，他们都进行类型转换，但是抛弃了返回的类型。

`as`模式匹配一个值，如果这个值的类型在运行时（runtime）和`as`模式右边的指定类型（或者那个类型的子类）是一致的。一旦匹配成功，匹配的值的类型被转换成`as`模式左边指定的模式。

关于使用`switch`语句来匹配`is`模式和`as`模式值的例子，请参阅`Type Casting for Any and AnyObject`。

类型转换模式的语法：

type-casting-pattern → is-pattern as-pattern

is-pattern → istype

as-pattern → patternastype

表达式模式（Expression Pattern）

表达式模式代表了一个表达式的值。这个模式只出现在`switch`语句中的`case`标签中。

由表达式模式所代表的表达式用Swift标准库中的`~=`操作符与输入表达式的值进行比较。如果`~=`操作符返回`true`，则匹配成功。默认情况下，`~=`操作符使用`==`操作符来比较两个相同类型的值。它也可以匹配一个整数值与一个`Range`对象中的整数范围，正如下面这个例子所示：

```
        let point = (1, 2)
        switch point {
        case (0, 0):
            println("(0, 0) is at the origin.")
        case (-2...2, -2...2):
            println("(\(point.0), \(point.1)) is near
the origin.")
        default:
            println("The point is at (\(point.0), \
(point.1)).")
        }
        // prints "(1, 2) is near the origin.”
```

你可以重载`~=`操作符来提供自定义的表达式行为。例如，你可以重写上面的例子，以实现用字符串表达的点来比较`point`表达式。

```
        // Overload the ~= operator to match a string
with an integer
        func ~=(pattern: String, value: Int) -> Bool
{
            return pattern == "\(value)"
```

```
        }
        switch point {
        case ("0", "0"):
            println("(0, 0) is at the origin.")
        case ("-2...2", "-2...2"):
            println("(\(point.0), \(point.1)) is near
the origin.")
        default:
            println("The point is at (\(point.0), \
(point.1)).")
        }
        // prints "(1, 2) is near the origin.”
```

表达式模式的语法：

expression-pattern → expression