表达式

在 Swift 中，有四种类型的表达式：前缀表达式，二元表达式， 基本表达式和后缀表达式。计算表达式会返回值、导致副作用，或者二者都有。

前缀表达式和二元表达式允许你给简单表达式应用各种运算符。基本表达式是概念上最简单的表达式种类，它们提供了一种访问值的方法。后缀表达式，如前缀表达式和二元表达式一般，后缀允许你建立更复杂的表达式，例如函数调用和成员访问。下面的章节中会详细介绍每种表达式。

> GRAMMAR OF AN EXPRESSION
>
> expression → try-operator$_{opt}$ prefix-expression binary-expressions$_{opt}$
>
> expression-list → expression expression , expression-list

前缀表达式

*前缀表达式*由可选的前缀运算符和一个表达式组合而成。前缀运算符接收一个参数，之后是表达式。

更多关于这些运算符的行为，请参阅基本运算符和高级运算符。

更多关于 Swift 标准库运算符的信息，请参阅Swift标准库运算符引用。

除了标准库运算符 ，你需要在把变量作为输入输出形式参数传递给函数调用表达式时，在形式参数名前紧跟 & 。更多详细信息以及相关示例，请参阅输入输出形式参数。

> GRAMMAR OF A PREFIX EXPRESSION
>
> prefix-expression → prefix-operator$_{opt}$ postfix-expression
>
> prefix-expression → in-out-expression
>
> in-out-expression → & identifier

Try 运算符

一个 *try 表达式*由一个 try 运算符和一个可抛出错误的表达式组成。具体形式如下：

try expression

*可选 try 表达式*由一个 try? 运算符和一个可抛出错误的表达式组成。具体形式如下：

try? expression

如果*表达式*不能抛出错误，那么可选 try 表达式的值就是一个包含了*表达式*值的可选项，否则，这个 try 表达式的值就是 nil 。

*强制 try 表达式*由一个 try! 运算符和一个表达式可抛出错误的表达式组成。具体形式如下：

try! expression

如果*表达式*抛出了错误，就会引发运行时错误。

当二元运算符左边的表达式标记为 try ， try? 或是 try! ，那运算符适用于整个二元表达式。就是说，你可以使用括号来明确运算符的作用域。

```
1  sum = try someThrowingFunction() + anotherThrowingFunction()   // try applies
2  to both function calls
3  sum = try (someThrowingFunction() + anotherThrowingFunction()) // try applies
   to both function calls
   sum = (try someThrowingFunction()) + anotherThrowingFunction() // Error: try
   applies only to the first function call
```

一个 try 表达式不能出现在二元运算符的右边，除非二元运算符是赋值运算符或者 try 表达式是用括号括起来的。

更多关于try，try?和try!的信息以及示例，请参阅错误处理

> GRAMMAR OF A TRY EXPRESSION
>
> try-operator → try　try?　try!

二元表达式

*二元表达式*由一个中缀二元运算符和两个表达式作为左实际参数和右实际参数，形式如下：

left-hand argument　operator　right-hand argument

更多关于运算符的使用信息，请参阅基本运算符和高级运算符

更多关于 Swift 提供的运算符信息，请参阅 Swift 标准库运算符

> 注意：
>
> 在解析时，由二元运算符组成的表达式会呈现一个水平列表。该列表会通过运算符的优先级转化为树。例如，表达式 2 + 3 * 5 可以理解为五个元素的水平列表， 2 ， + ， 3 ， * ，和 5 ，这个过程将其转换成（2 + (3 * 5)）。

> GRAMMAR OF A PREFIX EXPRESSION
>
> binary-expression → binary-operatorprefix-expression
>
> binary-expression → assignment-operatortry-operator$_{opt}$prefix-expression
>
> binary-expression → conditional-operatortry-operator$_{opt}$prefix-expression
>
> binary-expression → type-casting-operator
>
> binary-expressions → binary-expression binary-expressions$_{opt}$

赋值运算符

*赋值运算符*会给指定的表达式赋一个新的值，具体形式如下：

expression

*表达式*的值设置给通过计算该*值*所得到的值。如果*表达式*是元组，*值*必须与元组的元素数量匹配。(内嵌元组是允许的。)赋值从*值*的每一部分到*表达式*的相关部分执行，例如：

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a is "test", b is 12, c is 3, and 9.45 is ignored
```

赋值运算符不会返回任何值。

> GRAMMAR OF AN ASSIGNMENT OPERATOR
>
> assignment-operator → `=`

三元条件运算符

*三元条件运算符*会基于条件的值来对两个给定值中的一个进行计算，具体形式如下：

condition ? expression used if true : expression used if false

如果*条件*计算为 true ，条件运算符计算第一个表达式并返回它的值。否则，计算第二个表达式并返回其值。没有使用的表达式不会进行计算。

使用三元条件运算符的例子，请参阅三元条件运算符。

> GRAMMAR OF A CONDITIONAL OPERATOR
>
> conditional-operator → `?` try-operator$_{opt}$expression `:`

类型转换运算符

一共有四种类型转换运算符： is 运算符， as 运算符， as? 运算符，和 as! 运算符，它们具有如下形式：

expression is type

expression as type

expression as? type

expression as! type

is 运算符在运行时检查*表达式*是否可以转换为指定的*类型*。如果*表达式*可以转换为指定*类型*返回 ture ；否则返回 false 。

as 当在编译时确定成功时执行转换，比如向上转换或者桥接。向上转换允许你使用一个表达式作为类型的父类实例，不需要使用中介变量。下面的两种方法是等价的：

```
func f(_ any: Any) { print("Function for Any") }
func f(_ int: Int) { print("Function for Int") }
let x = 10
f(x)
// Prints "Function for Int"
let y: Any = x
f(y)
// Prints "Function for Any"
f(x as Any)
// Prints "Function for Any"


```

桥接能让你把一个 Swift 标准库类型例如 String 作为一个与Foundation类型例如 NSString 使用，不需要创建一个新的实例。更多关于桥接的信息，请参阅*Using Swift with Cocoa and Objective-C (Swift 3.0.1)* 中的 *Working with Cocoa Data Types*

as? 运算符条件性地转换*表达式*到指定的*类型*。 as? 运算符返回特定类型的可选项。在运行时，如果转换成功，*表达式*的值包装成可选项返回；否则，返回的值就是 nil 。如果转换指定*类型*时必定成功或者失败，就会出现编译时错误。

as! 运算符执行*表达式*到指定*类型*的强制转换。 as! 运算符返回一个指定*类型*的值，而不是可选类型。如果转换失败，运行时错误。 x as! T 和 (x as? T)! 的效果是一样的。

更多关于类型转换和类型转换运算符使用例子的信息，请参阅类型转换

> GRAMMAR OF A TYPE-CASTING OPERATOR
>
> type-casting-operator → `is` type
>
> type-casting-operator → `as` type
>
> type-casting-operator → `as` `?` type
>
> type-casting-operator → `as` `!` type

基本表达式

*基本表达式*是最基础的表达式类型。它们自身就可以作为表达式单独使用，也可以和其他符号组成前缀表达式、二元表达式和后缀表达式。

> GRAMMAR OF A PRIMARY EXPRESSION
>
> primary-expression → identifiergeneric-argument-clause$_{opt}$
>
> primary-expression → literal-expression
>
> primary-expression → self-expression
>
> primary-expression → superclass-expression
>
> primary-expression → closure-expression
>
> primary-expression → parenthesized-expression
>
> primary-expression → tuple-expression
>
> primary-expression → implicit-member-expression
>
> primary-expression → wildcard-expression
>
> primary-expression → selector-expression
>
> primary-expression → key-path-expression

字面量表达式

字面量表达式要么由普通字面量组成(例如字符串和数字)， 要么是数组或字典的字面量、playground 字面量，要么就是下面的特殊字面量：

Literal	Type	Value
#file	String	它出现的位置应该是文件名。
#line	Int	它出现位置应该是行数。
#column	Int	它出现的位置应该是列数。
#function	String	它出现位置应该是声明的名称。
#dsohandle	UnsafeRawPointer	它出现的位置应该使用 DSO（动态共享对象）处理。

在函数中， #function 会的值就是那个函数的名字，在方法里就是那个方法的名字，在属性设置器和读取器中则是属性的名字，在特殊的成员例如 init 或 subscript 就是关键字的名字，在最顶层文件，就就是当前模块的名字。

当作为函数或是方法的默认值时，特殊字面量的值取决于默认值表达式调用的时候。

```
1  func logFunctionName(string: String = #function) {
2     print(string)
3  }
4  func myFunction() {
5     logFunctionName() // Prints "myFunction()".
6  }
```

*数组字面量*是值的有序集合。它有如下形式：

[value 1 , value 2 , ...]

数组中最后的表达式后面可以跟着一个可选的逗号。数组值的字面量是 [T] ，这个 T 就是其中表达式的类型，如果表达式有多个类型， T 就是他们最接近的公共父类。空的数组写时使用一对空的方括号创建指定类型的空数组。

```
1  var emptyArray: [Double] = []
```

*字典字面量*是无序键值对的集合 ，它们具有如下形式：
[key 1 : value 1 : key 2 : value 2 , ...]

字典中最后的的表达式可以跟着一个可选的逗号。这个字典的字面量类型是 [Key: Value] , 其中 Key 是它的键表达式的类型, Value 的是它的值表达式的类型。如果表达式有多个类型， Key 和 Value 是它们各自值最接近的的公共类型。空字典的创建写做一对方括号中加一个冒号（ [:] ）来与空数组区分。你可以使用一个空的字典字面量创建一个特定类型的字典。

```
1  var emptyDictionary: [String: Double] = [:]
```

*Playground 字面量*是 Xcode 用来在程序编辑器中创建可交互的颜色、文件、或是图片的字面量，Playground 字面量在 Xcode 外是一种用特殊的语法表示的纯文本。

更多关于在 Xcode 中使用 playground 字面量的信息，请参阅 Xcode Help > Use playgrounds > Add a literal.

> GRAMMAR OF A LITERAL EXPRESSION
>
> literal-expression → literal
>
> literal-expression → array-literal dictionary-literal playground-literal
>
> literal-expression → `#file` `#line` `#column` `#function`
>
> array-literal → `[` array-literal-items$_{opt}$ `]`
>
> array-literal-items → array-literal-item `,`$_{opt}$ array-literal-item `,` array-literal-items
>
> array-literal-item → expression
>
> dictionary-literal → `[` dictionary-literal-items `]` `[` `:` `]`
>
> dictionary-literal-items → dictionary-literal-item `,`$_{opt}$ dictionary-literal-item `,` dictionary-literal-items
>
> dictionary-literal-item → expression `:` expression
>
> playground-literal → `#colorLiteral` `(` `red` `:` expression `,` `green` `:` expression `,` `blue` `:` expression `,` `alpha` `:` expression `)`
>
> playground-literal → `#fileLiteral` `(` `resourceName` `:` expression `)`
>
> playground-literal → `#imageLiteral` `(` `resourceName` `:` expression `)`

Self 表达式

self 表达式是一个明确当前类型或实例类型的显式引用，它有以下形式：

self

self. member name

self[subscript index]

self(initializer arguments)

self.init(initializer arguments)

在初始化器，下标，或是实例方法中， self 指代它出现位置的当前实例的类型。在类型方法中， self 指代它出现位置的类型。

当访问成员时， self 表达式用于指定范围，当指定范围中有另一个相同的变量名字是消除歧义，例如函数形式参数，例如：

```
class SomeClass {
    var greeting: String
    init(greeting: String) {
        self.greeting = greeting
    }
}
```

在值类型的异变方法中，你可以对 self 赋一个新的实例值。例如：

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

> GRAMMAR OF A SELF EXPRESSION
>
> self-expression → `self` self-method-expression self-subscript-expression self-initializer-expression
>
> self-method-expression → `self` `.` identifier
>
> self-subscript-expression → `self` `[` expression-list `]`
>
> self-initializer-expression → `self` `.` `init`

父类表达式

*父类表达式*使类互动于它的父类。它有以下形式：

super. member name

super[subscript index]

super.init(initializer arguments)

第一种形式用来访问父类中的成员。第二种形式用来访问父类的下标。第三种形式用来访问父类的初始化器。

子类可以在它们的成员、下标和初始化器的实现中使用父类表达式，来使用父类的实现。

> GRAMMAR OF A SUPERCLASS EXPRESSION
>
> superclass-expression → superclass-method-expression superclass-subscript-expression superclass-initializer-expression
>
> superclass-method-expression → `super` `.` identifier
>
> superclass-subscript-expression → `super` `[` expression-list `]`
>
> superclass-initializer-expression → `super` `.` `init`

闭包表达式

*闭包表达式*创建闭包，也就是其他语言中所谓的 *lambda* 或*匿名函数*。类似函数声明，闭包包含了执行语句，并且还会捕捉到其所在环境的常量和变量。它具有如下形式：

{(parameters) -> return type in

statements

}

形式参数和函数声明中的形式参数具有相同的形式，请参阅函数声明

还有几个特别的形式，会使闭包写得更简洁：

- 闭包可以省略它形式参数类型或它的返回类型抑或二者都省略。如果你省略了参数名字和它们的类型，也要省略语句中关键字 in 。如果省略的类型无法被推断，那么就会产生编译时错误；
- 闭包可以省略它的形式参数名。这样的话形式参数会被隐式命名 $ 后跟它所在的位置： $0 ， $1 ， $2 ,以此类推；
- 闭包只有一个表达式组成，那么那个表达式的值就是闭包的返回值。在表达式类型推断的时候也被推断为闭包返回类型。

下面的表达式是一样的：

```
1   myFunction {
2       (x: Int, y: Int) -> Int in
3       return x + y
4   }
5   myFunction {
6       (x, y) in
7       return x + y
8   }
9   myFunction { return $0 + $1 }
10  myFunction { $0 + $1 }
11
12
13
```

更多关于把闭包作为实际参数传递给函数的信息，请参阅函数调用表达式。

捕获列表

默认情况下，闭包表达式会使用强引用捕捉它所在的环境范围内的常量和变量。你可以使用*捕获列表*来显式地控制闭包如何捕获值。

捕获列表写在形式参数列表之前用在方括号内用逗号分隔表达式列表。如果你使用了捕获列表，你必须也使用 in 关键字 ，即使你省略了参数名，参数类型和返回值类型。

捕获列表中的条目会在创建闭包的时候初始化。捕获列表中的每一个条目，都会用常量来初始化并包含闭包所在环境中与它名称相同的常量或变量的值。例如下面的代码，a 在捕获列表中但是 b 不在，所以他们有不同的行为。

```
var a = 0
var b = 0
let closure = { [a] in
    print(a, b)
}
a = 10
b = 10
closure()
// Prints "0 10"

```

在一在闭包范围内，这有两个 a ，但是只有一个变量名字叫 b 。在闭包创建的时候会用闭包范围外的 a 的值初始化闭包范围内的 a ，但是他们毫无联系。这意思就是改变一个 a 的值不会影响到另一个 a 的值，相比之下，闭包范围内外 b 都是同一个变量，在闭包范围内外改变值，改变的都是同一个。

当捕获的变量类型是引用的时候就没什么区别了。例如，下面的代码中有两个变量 x ，一个常量在闭包内，一个变量在外，但是它们引用的是同一个对象因为是引用语义。

```
class SimpleClass {
    var value: Int = 0
}
var x = SimpleClass()
var y = SimpleClass()
let closure = { [x] in
    print(x.value, y.value)
}
x.value = 10
y.value = 10
closure()
// Prints "10 10"

```

如果表达式的类型是类，你可以在捕获列表达式使用 weak 和 unowned 修饰它，闭包会用弱引用和无主引用来获取表达式的值。

```
myFunction { print(self.title) }                  // strong capture
myFunction { [weak self] in print(self!.title) }  // weak capture
myFunction { [unowned self] in print(self.title) } // unowned capture
```

你可以在捕获列表中将任意表达式的值绑定到捕获列表。当闭包创建的时候表达式就会计算，并且会按照指定的类型捕获。例如：

```
// Weak capture of "self.parent" as "parent"
myFunction { [weak parent = self.parent] in print(parent!.title) }
```

更多关于闭包表达式的信息，请参阅闭包表达式。更多关于捕获列表的例子，请参阅解决闭包的循环强引用。

> GRAMMAR OF A CLOSURE EXPRESSION
>
> closure-expression → `{` closure-signature$_{opt}$ statements$_{opt}$ `}`
>
> closure-signature → capture-list$_{opt}$ closure-parameter-clause `throws`$_{opt}$ function-result$_{opt}$ `in`
>
> closure-signature → capture-list `in`
>
> closure-parameter-clause → `(` `)` | `(` closure-parameter-list `)` | identifier-list
>
> closure-parameter-list → closure-parameter | closure-parameter `,` closure-parameter-list
>
> closure-parameter → closure-parameter-name type-annotation$_{opt}$
>
> closure-parameter → closure-parameter-name type-annotation `...`
>
> closure-parameter-name → identifier
>
> capture-list → `[` capture-list-items `]`
>
> capture-list-items → capture-list-item | capture-list-item `,` capture-list-items
>
> capture-list-item → capture-specifier$_{opt}$ expression
>
> capture-specifier → `weak` | `unowned` | `unowned(safe)` | `unowned(unsafe)`

隐式成员表达式

*隐式成员表达式*是一种缩写方式访问成员的类型，例如枚举或是类方法，可以通过上下文推断出类型，它具有如下形式

. member name

例如

```
var x = MyEnumeration.someValue
x = .anotherValue
```

> GRAMMAR OF A IMPLICIT MEMBER EXPRESSION
>
> implicit-member-expression → . identifier

括号表达式

*括号表达式*由括号和表达式组成。你可以用括号显式指定表达式组的优先级。括号不会改变表达式的类型——例如，(1) 的类型就是 Int 。

> GRAMMAR OF A PARENTHESIZED EXPRESSION
>
> parenthesized-expression → (expression)

元组表达式

*元组表达式*由括号括起来用逗号分隔的表达式组成。每个表达式之前都有一个可选的标识符，用（：）分割，它具有如下形式：

(identifier 1 : expression 1 , identifier 2 : expression 2 , ...)

元组表达式可以一个表达式都没有，也可以包含两个或是更多的表达式。单个表达式用括号括起来就是括号表达式了。

> GRAMMAR OF A TUPLE EXPRESSION
>
> tuple-expression → ( ) (tuple-element , tuple-element-list)
>
> tuple-element-list → tuple-element tuple-element , tuple-element-list
>
> tuple-element → expression identifier : expression

通配符表达式

通配符表达式可以在赋值时显式地忽略一个值。例如， 10 赋值给 x ， 20 被忽略：

```
(x, _) = (10, 20)
// x is 10, and 20 is ignored
```

> GRAMMAR OF A WILDCARD EXPRESSION
>
> wildcard-expression → _

Key-Path 表达式

*Key-Path 表达式*指向类型的属性或者下标。你可以在动态程序任务中使用 key-path 表达式，比如键值观察。它们具有如下形式：

\type name.path

type name 是具体的类型，包括任意泛型形式参数，比如说 String 、 [Int] 或者 Set<Int> 。

path 则是由属性名称、下标、可选链表达式以及强制展开表达式组成。这些 key-path 元素中的任何一个都可以按照任意顺序重复多次。

在编译时，key-path 表达式会被 KeyPath 类实例替代。

要使用 key-path 来访问值，把 key-path 传给 subscript(keyPath:) 下标，这个下标对所有类型可用，比如说：

```
struct SomeStructure {
    var someValue: Int
}
let s = SomeStructure(someValue: 12)
let pathToProperty = \SomeStructure.someValue
let value = s[keyPath: pathToProperty]
// value is 12


```

type name 在接口可以隐式决定类型时能省略不写。下面的代码使用 \.someProperty 来代替 \SomeClass.someProperty ：

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}
let c = SomeClass(someProperty: 10)
c.observe(\.someProperty) { object, change in
    // ...
}

```

path 可以引用 self 来创建身份 key path （ \.self ）。身份 key path 会指向整个实例，所以你可以通过它来一次性访问和改变所有存在变量里的数据。比如说：

```
var compoundValue = (a: 1, b: 2)
// Equivalent to compoundValue = (a: 10, b: 20)
compoundValue[keyPath: \.self] = (a: 10, b: 20)
```

path 可以包含多个属性名称，使用点号分隔，以引用属性中的属性值。这份代码使用 key path 表达式 \OuterStructure.outer.someValue 来访问 OuterStructure 类型里 outer 中的 someValue 属性：

```
struct OuterStructure {
    var outer: SomeStructure
    init(someValue: Int) {
        self.outer = SomeStructure(someValue: someValue)
    }
}
let nested = OuterStructure(someValue: 24)
let nestedKeyPath = \OuterStructure.outer.someValue
let nestedValue = nested[keyPath: nestedKeyPath]
// nestedValue is 24


```

path 可以包含使用方括号来包含下标，只要下标的形式参数遵循 Hashable 协议。这个例子在 key path 中使用下标来访问数组中的第二个元素：

```
let greetings = ["hello", "hola", "bonjour", "안녕"]
let myGreeting = greetings[keyPath: \[String].[1]]
// myGreeting is 'hola'
```

在下标中使用的值可以是一个命名的值或者字面量。如果是值则会使用语义分析来捕捉值。下面的代码给两个key-path都在闭包中使用使用值 index 来访问 greetings 数组中的第三个元素。当 index 被修改，当闭包使用新的元素时，key-path 表达式仍旧引用第三个元素。

```
var index = 2
let path = \[String].[index]
let fn: ([String]) -> String = { strings in strings[index] }
print(greetings[keyPath: path])
// Prints "bonjour"
print(fn(greetings))
// Prints "bonjour"
// Setting 'index' to a new value doesn't affect 'path'
index += 1
print(greetings[keyPath: path])
// Prints "bonjour"
// Because 'fn' closes over 'index', it uses the new value
print(fn(greetings))
// Prints "안녕"
```

path 可以使用可选链以及强制展开。下面的代码在key path使用可选链来访问可选项中的属性：

```
let firstGreeting: String? = greetings.first
print(firstGreeting?.count as Any)
// Prints "Optional(5)"
// Do the same thing using a key path.
let count = greetings[keyPath: \[String].first?.count]
print(count as Any)
// Prints "Optional(5)"
```

你可以混合匹配key path的组合来访问类型中深入嵌套的值。下面的代码通过组合访问一个数组的字典中不同的值和属性。

```
let interestingNumbers = ["prime": [2, 3, 5, 7, 11, 13, 15],
                          "triangular": [1, 3, 6, 10, 15, 21, 28],
                          "hexagonal": [1, 6, 15, 28, 45, 66, 91]]
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]] as Any)
// Prints "Optional([2, 3, 5, 7, 11, 13, 15])"
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]![0]])
// Prints "2"
print(interestingNumbers[keyPath: \[String: [Int]].["hexagonal"]!.count])
// Prints "7"
print(interestingNumbers[keyPath: \[String:
[Int]].["hexagonal"]!.count.bitWidth])
// Prints "64"
```

你可以在平时任何使用函数或者闭包的地方使用 key path 表达式。特殊地，你可以在根类型为 SomeType 和 path 生成 Value 类型值时不使用 (SomeType) -> Value 类型的函数或者闭包而使用 key path 表达式。

```
struct Task {
    var description: String
    var completed: Bool
}
var toDoList = [
    Task(description: "Practice ping-pong.", completed: false),
    Task(description: "Buy a pirate costume.", completed: true),
    Task(description: "Visit Boston in the Fall.", completed: false),
]
// Both approaches below are equivalent.
let descriptions = toDoList.filter(\.completed).map(\.description)
let descriptions2 = toDoList.filter { $0.completed }.map { $0.description }

```

key path 表达式的副作用是它只会在评估表达式的时候计算一次。比如说，如果你在下标中用 key path 表达式调用函数，函数只会作为表达式评估被调用一次，而不是每次使用 key path 时都调用。

```
func makeIndex() -> Int {
    print("Made an index")
    return 0
}
// The line below calls makeIndex().
let taskKeyPath = \[Task][makeIndex()]
// Prints "Made an index"
// Using taskKeyPath doesn't call makeIndex() again.
let someTask = toDoList[keyPath: taskKeyPath]

```

要了解更多关于在代码中与 Objective-C API 中使用key path的信息，见 Keys and Key Paths in *Using Swift with Cocoa and Objective-C (Swift 4.0.3)*。*更多关于键值编程和键值观察者信息，见 Key-Value Coding Programming Guide* 以及 *Key-Value Observing Programming Guide*。

> GRAMMAR OF A KEY-PATH EXPRESSION
>
> key-path-expression → `\` type$_{opt}$ `.` key-path-components
>
> key-path-components → key-path-component key-path-component `.` key-path-components
>
> key-path-component → identifierkey-path-postfixes$_{opt}$ key-path-postfixes
>
> key-path-postfixes → key-path-postfixkey-path-postfixes$_{opt}$
>
> key-path-postfix → `?` `!` `[` function-call-argument-list `]`

Selector 表达式

选择器表达式可以让你使用 Objective-C 中用于引用属性的 getter 或s etter 以及方法的选择器

#selector(method name)

#selector(getter: property name)

#selector(setter: property name)

*方法名*和*属性名*都必须是 Objective-C 运行时中可用的方法和属性的引用。选择器表达式的返回值是 Selector 类型的实例。例如：

```
1   class SomeClass: NSObject {
2       let property: String
3       @objc(doSomethingWithInt:)
4       func doSomething(_ x: Int) {}
5
6       init(property: String) {
7           self.property = property
8       }
9   }
10  let selectorForMethod = #selector(SomeClass.doSomething(_:))
11  let selectorForPropertyGetter = #selector(getter: SomeClass.property)
```

当为属性的 getter 创建选择器的时候，*属性名*可以是变量或常量属性的引用。当为属性的 setter 创建选择器时，*属性名*只能是变量属性的引用。

*方法名*可以包含圆括号以分组，同时可以使用 as 运算符为两个名字相同但是类型不同的方法消除歧义。例如：

```
extension SomeClass {
    @objc(doSomethingWithString:)
    func doSomething(_ x: String) { }
}
let anotherSelector = #selector(SomeClass.doSomething(_:) as (SomeClass) ->
(String) -> Void)
```

由于选择器是编译时创建的，并不是运行时，所以编译器可以检查方法或者属性在运行时是否暴露给 Objective-C 运行时

> 注意
>
> 虽然*方法名*和*属性名*都是表达式，但是它们永远不会参与计算。

更多关于在 Swift 中使用选择器和 Objective-C API 的信息，请参阅 Objective-C Selectors in *Using Swift with Cocoa and Objective-C (Swift 3.0.1)*.

> GRAMMAR OF A SELECTOR EXPRESSION
>
> selector-expression → `#selector` `(` expression `)`
>
> selector-expression → `#selector` `(` `getter:` expression `)`
>
> selector-expression → `#selector` `(` `setter:` expression `)`

Key-Path 字符串表达式

Key-Path 字符串表达式允许你访问 Objective-C 中用于引用属性的字符串，以使用键值编码和键值观察者 API。它具有如下格式：

#KeyPath(property name)

*属性*名必须引用 Objective-C 运行时可用的属性。在编译时，Key-Path 表达式被字符串字面量所取代。例如:

```
@objc class SomeClass: NSObject {
    var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
    func keyPathTest() -> String {
        return #keyPath(someProperty)
    }
}
let c = SomeClass(someProperty: 12)
let keyPath = #keyPath(SomeClass.someProperty)
print(keyPath == c.keyPathTest())
// Prints "true"
if let value = c.value(forKey: keyPath) {
    print(value)
}
// Prints "12"


```

当你在类中使用 Key-Path 字符串表达式时，你可以通过直接写属性名来引用类中的属性，不需要写类名。

```
extension SomeClass {
    func getSomeKeyPath() -> String {
        return #keyPath(someProperty)
    }
}
print(keyPath == c.getSomeKeyPath())
// Prints "true"
```

因为 Key-Path 字符串是在编译时创建的，而不是在运行时，所以编译器可以检查对应属性是否暴露给 Objective-C 运行时

更多关于在Swift中使用选择器和Objective-C API 的信息，请参阅Keys and Key Paths in *Using Swift with Cocoa and Objective-C (Swift 3.0.1)*.更多使用键值编码和键值观察者的例子请参阅 *Key-Value Coding Programming Guide* and *Key-Value Observing Programming Guide*.

> 注意
>
> 尽管*属性名*是表达式，但它不会参与计算。

> GRAMMAR OF A KEY-PATH EXPRESSION
>
> key-path-expression → `#keyPath` `(` expression `)`

后缀表达式

*后缀表达式*给表达式使用后缀运算符或其他后缀语法形成。在语法上，每一个基本表达式也是一个后缀表达式。

更多关于这些运算符的信息，请参阅基本运算符和高级运算符。

更多关于Swift标准库提供的运算符信息，请参阅 *Swift Standard Library Operators Reference*.

> GRAMMAR OF A POSTFIX EXPRESSION
>
> postfix-expression → primary-expression
>
> postfix-expression → postfix-expressionpostfix-operator
>
> postfix-expression → function-call-expression
>
> postfix-expression → initializer-expression
>
> postfix-expression → explicit-member-expression
>
> postfix-expression → postfix-self-expression
>
> postfix-expression → dynamic-type-expression
>
> postfix-expression → subscript-expression
>
> postfix-expression → forced-value-expression
>
> postfix-expression → optional-chaining-expression

函数调用表达式

所有的*函数调用表达式*都是由一个函数名后跟用圆括号括起来的以逗号分隔的实际参数组成。函数调用表达式有以下形式：

function name (argument value 1 , argument value 2)

*函数名*可以是任何值是函数类型的表达式。

如果函数包含了形式参数的名字，函数调用的时候也必须要包括由(:)分隔的实际参数在内。这种函数调用表达式具有以下形式：

function name (argument name 1 : argument value 1 , argument name 2 : argument value 2)

函数调用表达式可以在圆括号后紧跟闭包表达式来包含一个尾随闭包。尾随闭包也是函数的实际参数，排列在圆括号内最后一个实际参数之后。下面的函数调用时等价的：

```
// someFunction takes an integer and a closure as its arguments
someFunction(x: x, f: {$0 == 13})
someFunction(x: x) {$0 == 13}
```

如果尾随闭包是函数唯一的实际参数，括号可以省略。

```
// someFunction takes a closure as its only argument
myData.someMethod() {$0 == 13}
myData.someMethod {$0 == 13}
```

> GRAMMAR OF A FUNCTION CALL EXPRESSION
>
> function-call-expression → postfix-expressionfunction-call-argument-clause
>
> function-call-expression → postfix-expressionfunction-call-argument-clause$_{opt}$trailing-closure
>
> function-call-argument-clause → ( ) | (function-call-argument-list)
>
> function-call-argument-list → function-call-argument | function-call-argument , function-call-argument-list
>
> function-call-argument → expression | identifier : expression
>
> function-call-argument → operator | identifier : operator
>
> trailing-closure → closure-expression

初始化器表达式

*初始化器表达式*可以访问类型的初始化器，它有如下形式：

expression .init(initializer arguments)

你可以在函数调用表达式中使用初始化器表达式来初始化一个新类型的实例。你也可以使用初始化器表达式来把初始化器委托给父类。

```
class SomeSubClass: SomeSuperClass {
    override init() {
        // subclass initialization goes here
        super.init()
    }
}
```

类似函数，初始化可以作为值使用，例如：

```
// Type annotation is required because String has multiple initializers.
let initializer: (Int) -> String = String.init
let oneTwoThree = [1, 2, 3].map(initializer).reduce("", +)
print(oneTwoThree)
// Prints "123"
```

如果你通过名字指定类型，你就可以不使用初始化器表达式直接访问类型的初始化器。在其他情况下，你必须使用初始化器表达式

```
let s1 = SomeType.init(data: 3)  // Valid
let s2 = SomeType(data: 1)       // Also valid
let s3 = type(of: someValue).init(data: 7)  // Valid
let s4 = type(of: someValue)(data: 5)       // Error

```

> GRAMMAR OF AN INITIALIZER EXPRESSION
>
> initializer-expression → postfix-expression . init
>
> initializer-expression → postfix-expression . init (argument-names)

显式成员表达式

*显示成员表达式*允许访问已命名类型、元组或者模型的成员。它由在项目和成员标识以及两者之间的点（.）组成。

expression . member name

命名类型的成员在类中声明或是在扩展中定义，例如：

```
class SomeClass {
    var someProperty = 42
}
let c = SomeClass()
let y = c.someProperty  // Member access
```

元组的成员是按照它们出现的顺序隐式地使用整数命名，从零开始，例如：

```
var t = (10, 20, 30)
t.0 = t.1
// Now t is (20, 20, 30)
```

模型成员访问这个模型顶层声明的成员。

要区分只有实际参数名不同的方法或初始化器，在圆括号中写出参实际数名，实际参数名后紧跟着冒号（:）。没有参数名的实际参数用下划线（_）代替参数名。对于重写方法，使用类型标注。例如：

```
class SomeClass {
   func someMethod(x: Int, y: Int) {}
   func someMethod(x: Int, z: Int) {}
   func overloadedMethod(x: Int, y: Int) {}
   func overloadedMethod(x: Int, y: Bool) {}
}
let instance = SomeClass()
let a = instance.someMethod            // Ambiguous
let b = instance.someMethod(x:y:)      // Unambiguous
let d = instance.overloadedMethod      // Ambiguous
let d = instance.overloadedMethod(x:y:)  // Still ambiguous
let d: (Int, Bool) -> Void  = instance.overloadedMethod(x:y:)  // Unambiguous


```

如果点号（.）出现在一行的开始，它会作为显式成员表达式的一部分，而不是隐式成员表达式。例如，下面展示的一系列方法被分为多行调用：

```
let x = [10, 3, 20, 15, 4]
   .sorted()
   .filter { $0 > 5 }
   .map { $0 * 100 }
```

> GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION
>
> explicit-member-expression → postfix-expression . decimal-digits
>
> explicit-member-expression → postfix-expression . identifiergeneric-argument-clause$_{opt}$
>
> explicit-member-expression → postfix-expression . identifier (argument-names)
>
> argument-names → argument-nameargument-names$_{opt}$
>
> argument-name → identifier :

后缀 self 表达式

后缀 self 表达式有一个表达式由类型名后面加 .self 组成，它具有以下形式：

expression .self

type .self

> GRAMMAR OF A SELF EXPRESSION
>
> postfix-self-expression → postfix-expression . self

下标表达式

*下标表达式*提供了相应的 getter 和 setter 下标以访问相关的下标声明。它具有以下形式：

expression [index expressions]

要计算下标表达式的值，表达式的类型的下标 getter 就会被以索引表达式作为下标的形式参数来调用。要设置它的值，下标的 setter 会以相同的方式调用。

更多关于下标声明的信息，请参阅下标协议声明。

> GRAMMAR OF A SUBSCRIPT EXPRESSION
>
> subscript-expression → postfix-expression [expression-list]

强制取值表达式

当你确定可选项的值不是 nil 时，用*强制取值表达式*来展开它，它具有如下形式：

expression !

如果*表达式*的值不是 nil ，可选项展开后返回非可选的相关类型。否则就是运行时错误。

强制取值表达式展开的值可以修改，要么通过改动自身，要么赋值给其成员。例如：

```
var x: Int? = 0
x! += 1
// x is now 1
var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
someDictionary["a"]![0] = 100
// someDictionary is now ["b": [10, 20], "a": [100, 2, 3]]

```

> GRAMMAR OF A FORCED-VALUE EXPRESSION
>
> forced-value-expression → postfix-expression !

可选链表达式

*可选链表达式*提供了在后缀表达式中使用可选值的简便语法。它具有如下形式：

expression ?

后缀运算符 ？ 会根据表达式形成可选链但是不会改变其值

可选链表达式必须出现在后缀表达式中，并且这会导致后缀表达式以很特殊的方法计算。如果可选链表达式的值为 nil ，那么后缀表达式中所有其他操作都会被忽略并且整个后缀表达式的结果是 nil 。如果可选链表达式不是 nil ，可选链表达式的值会展开以用于后缀表达式其余部分。总之，后缀表达式的值仍然都是可选类型。

如果后缀表达式中包含可选链表达式并嵌套在其他后缀表达式中，只有最外层的表达式返回可选类型。下面的例子中，当 c 不是 nil 的时候，它的值会展开并用于计算 .property 。然后它的值用于 .performAction() 。整个 c ? .property.performAction() 表达式返回一个可选类型的值。

```
var c: SomeClass?
var result: Bool? = c?.property.performAction()
```

下面的例子是上边那个不用可选链的版本：

```
var result: Bool? = nil
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

可选链表达式展开的值是可以被修改的，无论修改值本身还是修改值的成员。如果可选链表达式的值为 nil ，赋值运算符右侧的操作就不会计算了。例如：

```
func someFunctionWithSideEffects() -> Int {
    return 42  // No actual side effects.
}
var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
someDictionary["not here"]?[0] = someFunctionWithSideEffects()
// someFunctionWithSideEffects is not evaluated
// someDictionary is still ["b": [10, 20], "a": [1, 2, 3]]
someDictionary["a"]?[0] = someFunctionWithSideEffects()
// someFunctionWithSideEffects is evaluated and returns 42
// someDictionary is now ["b": [10, 20], "a": [42, 2, 3]]


```

> GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION
>
> optional-chaining-expression → postfix-expression ?