翻译：sg552 校对：numbbbbb

表达式（Expressions）

本页包含内容:

- [前缀表达式（Prefix Expressions）]
- [二元表达式（Binary Expressions）]
- [赋值表达式（Assignment Operator）]
- [三元条件运算符（Ternary Conditional Operator）]
- [类型转换运算符（Type-Casting Operators）]
- [主要表达式（Primary Expressions）]
- [后缀表达式（Postfix Expressions）]

Swift 中存在四种表达式： 前缀（prefix）表达式，二元（binary）表达式，主要（primary）表达式和后缀（postfix）表达式。表达式可以返回一个值，以及运行某些逻辑（causes a side effect）。

前缀表达式和二元表达式就是对某些表达式使用各种运算符（operators）。 主要表达式是最短小的表达式，它提供了获取（变量的）值的一种途径。 后缀表达式则允许你建立复杂的表达式，例如配合函数调用和成员访问。 每种表达式都在下面有详细论述~

表达式的语法

expression → *prefix-expression**binary-expressions*(opt) *expression-list* → *expression*| *expression*,*expression-list*

前缀表达式（Prefix Expressions）

前缀表达式由 前缀符号和表达式组成。（这个前缀符号只能接收一个参数）

Swift 标准库支持如下的前缀操作符:

- ++ 自增1 （increment）

- -- 自减1 （decrement）
- ! 逻辑否 （Logical NOT ）
- ~ 按位否 （Bitwise NOT ）
- + 加（Unary plus）
- - 减（Unary minus）

对于这些操作符的使用，请参见： Basic Operators and Advanced Operators

作为对上面标准库运算符的补充，你也可以对 某个函数的参数使用 '&'运算符。 更多信息，请参见： "In-Out parameters".

前缀表达式的语法

prefix-expression → *prefix-operator* (opt) *postfix-expression* *prefix-expression* → *in-out-expression* *in-out-expression* → *&identifier*

二元表达式（Binary Expressions）

二元表达式由 "左边参数" + "二元运算符" + "右边参数" 组成, 它有如下的形式:

```
left-hand argument operator right-hand argument
```

Swift 标准库提供了如下的二元运算符:

- 求幂相关（无结合，优先级160）
  - << 按位左移（Bitwise left shift）
  - 按位右移（Bitwise right shift）

- 乘除法相关（左结合，优先级150）
  - * 乘
  - / 除
  - % 求余
  - &* 乘法，忽略溢出（ Multiply, ignoring overflow）
  - &/ 除法，忽略溢出（Divide, ignoring overflow）
  - &% 求余, 忽略溢出（ Remainder, ignoring overflow）
  - & 位与（ Bitwise AND）
- 加减法相关（左结合, 优先级140）

    - + 加
    - - 减
    - &+ Add with overflow
    - &- Subtract with overflow
    - | 按位或（Bitwise OR ）
    - ^ 按位异或（Bitwise XOR）
- Range （无结合,优先级 135）
    - .. 半闭值域 Half-closed range
    - ... 全闭值域 Closed range
- 类型转换 （无结合,优先级 132）
    - is 类型检查（ type check）
    - as 类型转换（ type cast）
- Comparative （无结合,优先级 130）
    - < 小于
    - <= 小于等于
    - 大于
    - = 大于等于
    - == 等于
    - != 不等
    - === 恒等于
    - !== 不恒等
    - ~= 模式匹配（ Pattern match）
- 合取（ Conjunctive） （左结合,优先级 120）
    - && 逻辑与（Logical AND）
- 析取（Disjunctive） （左结合,优先级 110）
    - || 逻辑或（ Logical OR）
- 三元条件（Ternary Conditional ）（右结合,优先级 100）
    - ?: 三元条件 Ternary conditional
- 赋值 （Assignment） （右结合, 优先级 90）
    - = 赋值（Assign）
    - *= Multiply and assign
    - /= Divide and assign
    - %= Remainder and assign
    - += Add and assign

- -= Subtract and assign
- <<= Left bit shift and assign
- = Right bit shift and assign

- &= Bitwise AND and assign
- ^= Bitwise XOR and assign
- |= Bitwise OR and assign
- &&= Logical AND and assign
- ||= Logical OR and assign

关于这些运算符（operators）的更多信息，请参见：Basic Operators and Advanced Operators.

注意

在解析时, 一个二元表达式表示为一个一级数组（a flat list）, 这个数组（List）根据运算符的先后顺序，被转换成了一个tree. 例如： 2 + 3 5 首先被认为是： 2, + , 3, , 5. 随后它被转换成 tree （2 + （3 * 5））
二元表达式的语法

binary-expression → *binary-operatorprefix-expression binary-expression* → *assignment-operator*prefix-expression binary-expression → conditional-operator*prefix-expression binary-expression* → *type-casting-operator binary-expressions* → *binary-expressionbinary-expressions*(opt)

赋值表达式（Assignment Operator）

The assigment operator sets a new value for a given expression. It has the following form: 赋值表达式会对某个给定的表达式赋值。 它有如下的形式；

```
expression = value
```

就是把右边的 *value* 赋值给左边的 *expression*. 如果左边的*expression* 需要接收多个参数（是一个tuple ），那么右边必须也是一个具有同样数量参数的tuple. （允许嵌套的tuple）

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a is "test", b is 12, c is 3, and 9.45 is ignored
```
赋值运算符不返回任何值。

赋值表达式的语法

assignment-operator → =

三元条件运算符（Ternary Conditional Operator）

三元条件运算符 是根据条件来获取值。 形式如下：

```
`condition` ? `expression used if true` : `expression used if false`
```
如果 `condition` 是true, 那么返回 第一个表达式的值（此时不会调用第二个表达式）， 否则返回第二个表达式的值（此时不会调用第一个表达式）。

想看三元条件运算符的例子，请参见： Ternary Conditional Operator.

三元条件表达式

`conditional-operator` → ?`expression`:

类型转换运算符（Type-Casting Operators）

有两种类型转换操作符： as 和 is. 它们有如下的形式：

```
`expression` as `type`
`expression` as? `type`
`expression` is `type`
```
as 运算符会把目标表达式转换成指定的类型（specified type），过程如下：

- 如果类型转换成功， 那么目标表达式就会返回指定类型的实例（instance）. 例如：把子类（subclass）变成父类（superclass）时.

- 如果转换失败，则会抛出编译错误（ compile-time error）。
- 如果上述两个情况都不是（也就是说，编译器在编译时期无法确定转换能否成功，） 那么目标表达式就会变成指定的类型的optional.（is an optional of the specified type ） 然后在运行时，如果转换成功， 目标表达式就会作为 optional的一部分来返回， 否则，目标表达式返回nil. 对应的例子是： 把一个 superclass 转换成一个 subclass.

```
class SomeSuperType {}
class SomeType: SomeSuperType {}
class SomeChildType: SomeType {}
let s = SomeType()

let x = s as SomeSuperType  // known to succeed; type
is SomeSuperType
let y = s as Int            // known to fail;
compile-time error
let z = s as SomeChildType  // might fail at runtime;
type is SomeChildType?
```

使用'as'做类型转换跟正常的类型声明，对于编译器来说是一样的。例如：

```
let y1 = x as SomeType  // Type information from 'as'
let y2: SomeType = x    // Type information from an
annotation
```

'is' 运算符在“运行时（runtime）”会做检查。 成功会返回true, 否则 false

The check must not be known to be true or false at compile time. The following are invalid: 上述检查在“编译时（compile time）”不能使用。 例如下面的使用是错误的：

```
"hello" is String
"hello" is Int
```

关于类型转换的更多内容和例子，请参见： Type Casting.

类型转换的语法

type-casting-operator → is*type*| as?(opt)*type*

主要表达式（Primary Expressions）

主要表达式是最基本的表达式。 它们可以跟 前缀表达式，二元表达式，后缀表达式以及其他主要表达式组合使用。

主要表达式的语法

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 → implicit-member-expression primary-expression → wildcard-expression*

字符型表达式（Literal Expression）

由这些内容组成：普通的字符（string, number），一个字符的字典或者数组，或者下面列表中的特殊字符。

字符（Literal）	类型（Type）	值（Value）
*_FILE_*	String	所在的文件名
*_LINE_*	Int	所在的行数
*_COLUMN_*	Int	所在的列数
*_FUNCTION_*	String	所在的function 的名字

在某个函数（function）中，`__FUNCTION__` 会返回当前函数的名字。 在某个方法（method）中，它会返回当前方法的名字。 在某个property 的getter/setter中会返回这个属性的名字。 在init/subscript中 只有的特殊成员（member）中会返回这个keyword的名字，在某个文件的顶端（the top level of a file），它返回的是当前module的名字。

一个array literal，是一个有序的值的集合。 它的形式是：

```
[`value 1`, `value 2`, `...`]
```

数组中的最后一个表达式可以紧跟一个逗号（','）. []表示空数组 。 array literal的type是 T[], 这个T就是数组中元素的type. 如果该数组中有多种type, T则是跟这些type的公共supertype最接近的type.（closest common

supertype）

一个`dictionary literal`是一个包含无序的键值对（key-value pairs）的集合，它的形式是:

```
[`key 1`: `value 1`, `key 2`: `value 2`, `...`]
```

dictionary 的最后一个表达式可以是一个逗号（','）. [:] 表示一个空的 dictionary. 它的type是 Dictionary （这里KeyType表示 key的type, ValueType表示 value的type） 如果这个dictionary 中包含多种 types, 那么KeyType, Value 则对应着它们的公共supertype最接近的type（ closest common supertype）.

字符型表达式的语法

literal-expression → *literal literal-expression* → *array-literal*| *dictionary-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*

self表达式（Self Expression）

self表达式是对 当前type 或者当前instance的引用。它的形式如下:

self self.`member name` self[`subscript index`] self（`initializer arguments`） self.init（`initializer arguments`）

如果在 initializer, subscript, instance method中， self等同于当前type的 instance. 在一个静态方法（static method）, 类方法（class method）中， self等同于当前的type.

当访问 member（成员变量时）， self 用来区分重名变量（例如函数的参数）. 例如， （下面的 self.greeting 指的是 var greeting: String, 而不是 init（greeting: String） ）

```
class SomeClass {
    var greeting: String
    init (greeting: String)  {
```

```
        self.greeting = greeting
    }
}
```
在mutating 方法中， 你可以使用self 对 该instance进行赋值。

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX (deltaX: Double, y deltaY:
Double)  {
        self = Point (x: x + deltaX, y: y + deltaY)
    }
}
```
self表达式的语法

self-expression → self *self-expression* → self.*identifier self-expression* → self[*expression*] *self-expression* → self.init

超类表达式（Superclass Expression）

超类表达式可以使我们在某个class中访问它的超类. 它有如下形式：

```
super.`member name`
super[`subscript index`]
super.init (`initializer arguments`)
```
形式1 用来访问超类的某个成员（member）. 形式2 用来访问该超类的 subscript 实现。 形式3 用来访问该超类的 initializer.

子类（subclass）可以通过超类（superclass）表达式在它们的 member, subscripting 和 initializers 中来利用它们超类中的某些实现（既有的方法或者逻辑）。

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*] *superclass-initializer-expression* → super.init

闭包表达式（Closure Expression）

闭包（closure） 表达式可以建立一个闭包（在其他语言中也叫 lambda, 或者 匿名函数（anonymous function））. 跟函数（function）的声明一样, 闭包（closure）包含了可执行的代码（跟方法主体（statement）类似） 以及接收（capture）的参数。 它的形式如下:

```
{ (parameters) -> return type in
    statements
}
```

闭包的参数声明形式跟方法中的声明一样, 请参见：Function Declaration.

闭包还有几种特殊的形式, 让使用更加简洁:

- 闭包可以省略 它的参数的type 和返回值的type. 如果省略了参数和参数类型，就也要省略 'in'关键字。 如果被省略的type 无法被编译器获知（inferred） ，那么就会抛出编译错误。
- 闭包可以省略参数，转而在方法体（statement）中使用 0, 1, $2 来引用出现的第一个，第二个，第三个参数。
- 如果闭包中只包含了一个表达式，那么该表达式就会自动成为该闭包的返回值。 在执行 'type inference '时，该表达式也会返回。

下面几个 闭包表达式是 等价的:

```
myFunction {
    (x: Int, y: Int) -> Int in
    return x + y
}

myFunction {
    (x, y) in
    return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

关于 向闭包中传递参数的内容，参见： Function Call Expression.

闭包表达式可以通过一个参数列表（capture list） 来显式指定它需要的参

数。 参数列表 由中括号 [] 括起来，里面的参数由逗号','分隔。一旦使用了参数列表，就必须使用'in'关键字（在任何情况下都得这样做，包括忽略参数的名字，type, 返回值时等等）。

在闭包的参数列表（ capture list）中， 参数可以声明为 'weak' 或者 'unowned' .

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

在参数列表中，也可以使用任意表达式来赋值. 该表达式会在 闭包被执行时赋值，然后按照不同的力度来获取（这句话请慎重理解）。（captured with the specified strength. ） 例如：

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

关于闭包表达式的更多信息和例子，请参见： Closure Expressions.

闭包表达式的语法

closure-expression → {*closure-signature*opt*statements*} *closure-signature* → *parameter-clausefunction-result*(opt)in *closure-signature* → *identifier-list function-result*(opt)in *closure-signature* → *capture-listparameter-clause function-result*(opt)in *closure-signature* → *capture-listidentifier-listfunction-result*(opt)in *closure-signature* → *capture-list*in *capture-list* → [*capture-specifierexpression*] *capture-specifier* → weak| unowned| unowned（safe）| unowned（unsafe）

隐式成员表达式（Implicit Member Expression）

在可以判断出类型（type）的上下文（context）中，隐式成员表达式是访问某个type的member（ 例如 class method, enumeration case） 的简洁方法。 它的形式是：

`.member name`

例子：

```
var x = MyEnumeration.SomeValue
x = .AnotherValue
```

隐式成员表达式的语法

implicit-member-expression → .identifier

圆括号表达式（Parenthesized Expression）

圆括号表达式由多个子表达式和逗号','组成。 每个子表达式前面可以有 identifier x: 这样的可选前缀。形式如下：

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

圆括号表达式用来建立tuples， 然后把它做为参数传递给 function. 如果某个圆括号表达式中只有一个 子表达式，那么它的type就是 子表达式的type。例如： （1）的 type是Int, 而不是（Int）

圆括号表达式的语法

parenthesized-expression → （expression-element-list (opt)） expression-element-list → expression-element| expression-element,expression-element-list expression-element → expression| identifier:expression

通配符表达式（Wildcard Expression）

通配符表达式用来忽略传递进来的某个参数。例如：下面的代码中，10被传递给x, 20被忽略（译注：好奇葩的语法。。。）

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

通配符表达式的语法

wildcard-expression → _

后缀表达式（Postfix Expressions）

后缀表达式就是在某个表达式的后面加上 操作符。 严格的讲，每个主要表达式（primary expression）都是一个后缀表达式

Swift 标准库提供了下列后缀表达式：

- ++ Increment
- -- Decrement

对于这些操作符的使用，请参见： Basic Operators and Advanced Operators

后缀表达式的语法

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（Function Call Expression）

函数调用表达式由函数名和参数列表组成。它的形式如下：

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

The function name can be any expression whose value is of a function type.（不用翻译了, 太罗嗦）

如果该function 的声明中指定了参数的名字，那么在调用的时候也必须得写出来. 例如：

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

可以在 函数调用表达式的尾部（最后一个参数之后）加上 一个闭包（closure） ， 该闭包会被目标函数理解并执行。它具有如下两种写法：

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

```
someFunction（x） {$0 == 13}
```
如果闭包是该函数的唯一参数，那么圆括号可以省略。

```
// 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-expressionparenthesized-expression*
function-call-expression → *postfix-expressionparenthesized-expression*(opt)
trailing-closure trailing-closure → *closure-expression*

初始化函数表达式（Initializer Expression）

Initializer表达式用来给某个Type初始化。 它的形式如下：

`expression`.init（`initializer arguments`）

（Initializer表达式用来给某个Type初始化。） 跟函数（function）不同，initializer 不能返回值。

```
var x = SomeClass.someClassFunction // ok
var y = SomeClass.init              // error
```swift

可以通过 initializer 表达式来委托调用（delegate to ）到superclass的initializers.

```swift
class SomeSubClass: SomeSuperClass {
    init（） {
        // subclass initialization goes here
        super.init（）
    }
}
```
initializer表达式的语法

initializer-expression → *postfix-expression*.init

显式成员表达式（Explicit Member Expression）

显示成员表达式允许我们访问type, tuple, module的成员变量。它的形式如下：

`expression.member name`

该member 就是某个type在声明时候所定义（declaration or extension） 的变量, 例如：

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

对于tuple, 要根据它们出现的顺序（0, 1, 2...）来使用:

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

The members of a module access the top-level declarations of that module.（不确定：对于某个module的member的调用，只能调用在top-level声明中的member.）

显示成员表达式的语法

explicit-member-expression → *postfix-expression*.*decimal-digit* *explicit-member-expression* → *postfix-expression*.*identifier*generic-argument-*clause*(opt)

后缀self表达式（Postfix Self Expression）

后缀表达式由 某个表达式 + '.self' 组成. 形式如下：

`expression`.self `type`.self

形式1 表示会返回 expression 的值。例如： x.self 返回 x

形式2：返回对应的type。我们可以用它来动态的获取某个instance的

type。

后缀self表达式的语法

postfix-self-expression → *postfix-expression*.self

dynamic表达式（Dynamic Type Expression）

（因为dynamicType是一个独有的方法，所以这里保留了英文单词，未作翻译, --- 类似与self expression）

dynamicType 表达式由 某个表达式 + '.dynamicType' 组成。

`expression`.dynamicType

上面的形式中， expression 不能是某type的名字（当然了，如果我都知道它的名字了还需要动态来获取它吗）。动态类型表达式会返回"运行时"某个instance的type, 具体请看下面的列子：

```
class SomeBaseClass {
    class func printClassName () {
        println ("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName () {
        println ("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass ()

// someInstance is of type SomeBaseClass at compile
time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName ()
// prints "SomeSubClass"
```

dynamic type 表达式

dynamic-type-expression → *postfix-expression*.dynamicType

附属脚本表达式（Subscript Expression）

附属脚本表达式提供了通过附属脚本访问getter/setter 的方法。它的形式是：

`expression`[`index expressions`]

可以通过附属脚本表达式通过getter获取某个值，或者通过setter赋予某个值.

关于subscript的声明，请参见： Protocol Subscript Declaration.

附属脚本表达式的语法

subscript-expression → *postfix-expression*[*expression-list*]

强制取值表达式（Forced-Value Expression）

强制取值表达式用来获取某个目标表达式的值（该目标表达式的值必须不是nil ）。它的形式如下：

`expression`!

如果该表达式的值不是nil, 则返回对应的值。 否则，抛出运行时错误（runtime error）。

强制取值表达式的语法

forced-value-expression → *postfix-expression*!

可选链表达式（Optional-Chaining Expression）

可选链表达式由目标表达式 + '?' 组成，形式如下：

`expression`?

后缀'?' 返回目标表达式的值，把它做为可选的参数传递给后续的表达式

如果某个后缀表达式包含了可选链表达式，那么它的执行过程就比较特殊： 首先先判断该可选链表达式的值，如果是 nil, 整个后缀表达式都返回nil, 如果该可选链的值不是nil, 则正常返回该后缀表达式的值（依次执行它

的各个子表达式）。在这两种情况下，该后缀表达式仍然是一个optional type（In either case, the value of the postfix expression is still of an optional type）

如果某个"后缀表达式"的"子表达式"中包含了"可选链表达式"，那么只有最外层的表达式返回的才是一个optional type. 例如，在下面的例子中， 如果c 不是nil, 那么 c?.property.performAction（） 这句代码在执行时，就会先获得c 的property方法，然后调用 performAction（）方法。 然后对于 "c?.property.performAction（）" 这个整体，它的返回值是一个optional type.

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

如果不使用可选链表达式，那么 上面例子的代码跟下面例子等价：

```
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

可选链表达式的语法

optional-chaining-expression → *postfix-expression*?