

Swift 基本数据类型



扫码试看/订阅
《Swift核心技术与实战》视频课程

变量和常量

声明常量和变量

- 使用关键字 `let` 来声明常量
- 使用关键字 `var` 来声明变量

```
let maximumNumberOfLoginAttempts = 10  
var currentLoginAttempt = 0
```

```
maximumNumberOfLoginAttempts = 9  
currentLoginAttempt = 1
```

Cannot assign to value: 'maximumNumberOfLoginAttempts' is a 'let' constant

声明常量和变量

- 可以在一行中声明多个变量或常量，用逗号分隔

```
var x = 0.0, y = 0.0, z = 0.0
```

类型标注

- 在声明一个变量或常量时提供类型标注，来明确变量或常量能够储存值的类型
- 添加类型标注的方法是在变量或常量的名字后边加一个冒号，再跟一个空格，最后加上要使用的类型名称
- 可以在一行中定义多个相关的变量为相同的类型，用逗号分隔，只要在最后的变量名字后边加上类型标注

```
var welcomeMessage: String
```

```
welcomeMessage = "hello"
```

```
welcomeMessage = 10
```



Cannot assign value of type 'Int' to type 'String'

变量和常量命名

- 常量和变量的名字几乎可以使用任何字符，甚至包括 Unicode 字符
- 常量和变量的名字不能包含空白字符、数学符号、箭头、保留的（或者无效的）Unicode 码位、连线和制表符。也不能以数字开头，尽管数字几乎可以使用在名字其他的任何地方

```
let  $\pi$  = 3.14159
```

```
let 你好 = "你好世界"
```

```
let 🐶🐮 = "dogcow"
```

打印常量和变量

- `print(_:separator:terminator:)`
- 字符串插值

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"

print("欢迎语是\(你好)")
```


基本数据类型

整数

- Swift 提供了 8, 16, 32 和 64 位编码的有符号和无符号整数
- 命名方式：例如 8 位无符号整数的类型是 UInt8，32 位有符号整数的类型是 Int32
- 通过 min 和 max 属性来访问每个整数类型的最小值和最大值

整数

- Swift 提供了一个额外的整数类型： `Int` ， 它拥有与当前平台的原生字相同的长度
- 同时 Swift 也提供 `UInt` 类型， 来表示平台长度相关的无符号整数
- 建议在用到整数的地方都使用 `Int`

浮点类型

- Double: 64 位浮点数, 至少有 15 位数字的精度
- Float: 32 位浮点数, 至少有 6 位数字的精度
- 在两种类型都可以的情况下, 推荐使用 Double 类型。

数值范围

类型	大小 (字节数)	区间值
Int8	1 字节	-128 到 127
UInt8	1 字节	0 到 255
Int32	4 字节	-2147483648 到 2147483647
UInt32	4 字节	0 到 4294967295
Int64	8 字节	-9223372036854775808 到 9223372036854775807
UInt64	8 字节	0 到 18446744073709551615
Float	4 字节	1.2E-38 到 3.4E+38 (~6 digits)
Double	8 字节	2.3E-308 到 1.7E+308 (~15 digits)

Bool

- Bool: true 和 false
- Swift 的类型安全机制会阻止你用一个非布尔量的值替换掉 Bool

⚠ The playground could not continue running because the playground source did not compile successfully.

```
1 import UIKit
2
3 let i = 1
4 if i {
5     print(i)
6 }
7
```

error: basic.playground:4:4: error: 'Int' is not convertible to 'Bool'
if i {
 ^

```
let i = 1
if i == 1 {
    print(i)
}
```

```
1
"1\n"
```

类型别名

- 类型别名是一个为已存在类型定义的一个可选择的名字
- 你可以关键字 `typealias` 定义一个类型的别名
- 当你想通过在一个在上下文中看起来更合适可具有表达性的名字来引用一个已存在的类型时，这时别名就非常有用

```
typealias AudioSample = UInt8  
let sample: AudioSample = 32
```

Tuple

Tuple

- 元组把多个值合并成单一的复合型的值
- 元组内的值可以是任何类型，而且可以不必是同一类型

```
let error = (1, "没有权限")  
print(error.0)  
print(error.1)
```

```
(.0 1, .1 "没有权限")  
"1\n"  
"没有权限\n"
```

元素命名

- 元组中的每一个元素可以指定对应的元素名称
- 如果没有指定名称的元素也可以使用下标的方式来引用

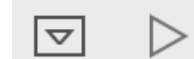
```
let error = (errorCode: 1, errorMessage: "没有权限")  
print(error.errorCode)  
print(error.errorMessage)
```

```
(errorCode 1, errorMessage "没有权限")  
"1\n"  
"没有权限\n"
```

Tuple 修改

- 用 var 定义的元组就是可变元组，let 定义的就是不可变元组
- 不管是可变还是不可变元组，元组在创建后就不能增加和删除元素
- 可以对可变元组的元素进行修改，但是不能改变其类型
- any 类型可以改为任何类型

```
115
116 var error = (errorCode: 1, errorMessage: "没有权限")
117 error.errorCode = 2
118 error.errorCode = "2"
119
120
121
122
```



```
error: basic.playground:118:19: error: cannot assign value of type 'String' to type 'Int'
error.errorCode = "2"
                  ^~~~
```

```
var error:(Any, String) = (1, "没有权限")
error.0 = 2
print(error)
error.0 = "abc"
print(error)
```

```
"(2, "没有权限")\n"
"abc"
>("abc", "没有权限")\n"
```

Tuple 分解

- 以将一个元组的内容分解成单独的常量或变量
- 如果只需要使用其中的一部分数据，不需要的数据可以用下滑线（_）代替

```
let error = (1, "没有权限")
let (errcode, errorMessage) = error
print(errcode)
print(errorMessage)
```

```
(.0 1, .1 "没有权限")

"1\n"
"没有权限\n"
```

```
let error = (1, "没有权限")
let (_, errorMessage) = error
print(errorMessage)
```

```
(.0 1, .1 "没有权限")

"没有权限\n"
```

作为函数返回值

- 使用 Tuple 为函数返回多个值
- 返回值的 Tuple 可以在函数的返回类型部分被命名

```
func writeFile(conent: String) -> (errorCode: Int, errorMessage: String) {  
    //write to file  
    return (1, "没有权限")  
}
```

Optional

为什么需要 Optional

- Objective-C 里的 nil 是无类型的指针
- Objective-C 里面的数组、字典、集合等不允许放入 nil
- Objective-C 所有对象变量都可以为 nil
- Objective-C 只能用在对象上，而在其他地方又用其他特殊值（例如NSNotFound）表示值的缺失

Optional

- 通过在变量类型后面加 ? 表示:
 - 这里有一个值, 他等于 x
- 或者
- 这里根本没有值

Optional

- 你可以通过给可选变量赋值一个 nil 来将之设置为没有值
- 在 Objective-C 中 nil 是一个指向不存在对象的指针
- 在 Swift 中， nil 不是指针，他是值缺失的一种特殊类型，任何类型的可选项都可以设置成 nil 而不仅仅是对象类型

```
var str: String = nil  
var str1 : String? = nil
```

❌ 'nil' cannot initialize specified type 'String'

Optional-If 语句以及强制展开

- 可选项是没法直接使用的
- 需要用!展开之后才能使用(意思是我知道这个可选项里边有值，展开吧)

```
var str: String? = "abc"  
let count = str.count
```

```
error: Optional.playground:21:13: error: value of optional type 'String?' must be unwrapped to refer to member 'count' of  
wrapped base type 'String'  
let count = str.count  
               ^
```

```
Optional.playground:21:13: note: chain the optional using '?' to access member 'count' only for non-'nil' base values  
let count = str.count  
               ^  
               ?
```

```
Optional.playground:21:13: note: force-unwrap using '!' to abort execution if the optional value contains 'nil'  
let count = str.count  
               ^  
               !
```

```
var str: String? = "abc"  
if str != nil {  
    let count = str!.count  
    print(count)  
}
```

"abc"

3

"3\n"

Optional-强制展开

- 使用！来获取一个不存在的可选值会导致运行错误，在使用!强制展开之前必须确保可选项中包含一个非 nil 的值。

```
var str: String?
```

```
let count = str!.count
```



error: Exe...



error

Optional-绑定

- 可以使用可选项绑定来判断可选项是否包含值，如果包含就把值赋给一个临时的常量或者变量
- 可选绑定可以与 if 和 while 的语句用来检查可选项内部的值，并赋值给一个变量或常量
- 同一个 if 语句中包含多可选项绑定，用逗号分隔即可。如果任一可选绑定结果是 nil 或者布尔值为 false，那么整个 if 判断会被看作 false

```
var str: String? = "abc"  
if let actualStr = str {  
    let count = actualStr.count  
    print(count)  
}
```

"abc"

3

"3\n"

Optional-隐式展开

- 有些可选项一旦被设定值之后，就会一直拥有值，在这种情况下，就可以去掉检查的需求，也不必每次访问的时候都进行展开
- 通过在声明的类型后边添加一个叹号（String!）而非问号（String?）来书写隐式展开可选项
- 隐式展开可选项主要被用在 Swift 类的初始化过程中

```
var str: String! = "abc"  
let count = str.count  
print(count)
```

```
"abc"  
3  
"3\n"
```

Optional-可选链

- 可选项后面加问号
- 如果可选项不为 nil, 返回一个可选项结果, 否则返回 nil

```
var str: String? = "abc"  
let count = str?.count  
let lastIndex = count - 1
```

error: Optional.playground:30:17: error: value of optional type 'Int?' must be unwrapped to a value of type 'Int'
let lastIndex = count - 1
 ^

Optional.playground:30:17: note: coalesce using '??' to provide a default when the optional value contains 'nil'
let lastIndex = count - 1
 ^
 ?? default value

Optional.playground:30:17: note: force-unwrap using '!' to abort execution if the optional value contains 'nil'
let lastIndex = count - 1
 ^
 !

```
var str: String? = "abc"  
let count = str?.count  
if count != nil {  
    let lastIndex = count! - 1  
    print(lastIndex)  
}
```

```
"abc"  
3  
  
2  
"2\n"
```

Optional-实现探究

Optional-实现探究

- Optional 其实是标准库里的一个 enum 类型
- 用标准库实现语言特性的典型

Swift > Optional > Optional

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {  
  
    /// The absence of a value.  
    ///  
    /// In code, the absence of a value is typically written using the `nil`  
    /// literal rather than the explicit `.none` enumeration case.  
    case none  
  
    /// The presence of a value, stored as `Wrapped`.  
    case some(Wrapped)  
  
    /// Creates an instance that stores the given value.  
    public init(_ some: Wrapped)
```

```
public init(_ some: Wrapped)
```

```
/// Creates an instance that stores the given value.
```


Optional-实现探究

- Optional.none 就是 nil
- Optional.some 则包装了实际的值

```
var str: Optional<String> = "abc"  
if let actualStr = str {  
    let count = actualStr.count  
    print(count)  
}
```

"abc"

3

"3\n"

Optional-展开实现

- 泛型属性 `unsafelyUnwrapped`

```
/// The wrapped value of this instance, unwrapped without checking whether
/// the instance is `nil`.
///
/// The `unsafelyUnwrapped` property provides the same value as the forced
/// unwrap operator (postfix `!`). However, in optimized builds (`-O`), no
/// check is performed to ensure that the current instance actually has a
/// value. Accessing this property in the case of a `nil` value is a serious
/// programming error and could lead to undefined behavior or a runtime
/// error.
///
/// In debug builds (`-Onone`), the `unsafelyUnwrapped` property has the same
/// behavior as using the postfix `!` operator and triggers a runtime error
/// if the instance is `nil`.
///
/// The `unsafelyUnwrapped` property is recommended over calling the
/// `unsafeBitCast(_:)` function because the property is more restrictive
/// and because accessing the property still performs checking in debug
/// builds.
///
/// - Warning: This property trades safety for performance. Use
///   `unsafelyUnwrapped` only when you are confident that this instance
///   will never be equal to `nil` and only after you've tried using the
///   postfix `!` operator.
@inlinable public var unsafelyUnwrapped: Wrapped { get }
```

Optional-展开实现

- 理论上我们可以直接调用 `unsafelyUnwrapped` 获取可选项的值

```
var str: String? = "abc"  
let count = str.unsafelyUnwrapped.count  
print(count)
```

"abc"

3

"3\n"

字符串-初始化

初始化空串

- 字面量
- 初始化器语法
- isEmpty 检查是否为空串

```
var emptyString = ""  
var anotherEmptyString = String()  
  
if emptyString.isEmpty {  
    print("Nothing to see here")  
}
```

字面量

- 字符串字面量是被双引号（"）包裹的固定顺序文本字符
- Swift 会为 str 常量推断类型为 String

```
let str = "some string"
```

多行字面量

- 多行字符串字面量是用三个双引号引起来的一系列字符
- 多行字符串字面量把所有行包括在引号内，开始和结束默认不会有换行符
- 当你的代码中在多行字符串字面量里包含了换行，那个换行符同样会成为字符串里的值。如果你想要使用换行符来让你的代码易读，却不想让换行符成为字符串的值，那就在那些行的末尾使用反斜杠（ \ ）

```
let softWrappedQuotation = ""  
The White Rabbit put on his spectacles. "Where shall I begin,  
please your Majesty?" he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on |  
till you come to the end; then stop."  
""  
print(softWrappedQuotation)
```

```
The White Rabbit put on his spectacles. "Where shall  
I begin,  
please your Majesty?" he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on  
till you come to the end; then stop."...
```

```
let softWrappedQuotation1 = ""|  
The White Rabbit put on his spectacles. "Where shall I begin, \  
please your Majesty?" he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on \  
till you come to the end; then stop."  
""  
print(softWrappedQuotation1)
```

```
The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?"  
he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop."
```

多行字面量

- 要让多行字符串字面量起始或结束于换行，就在第一或最后一行写一个空行
- 多行字符串可以缩进以匹配周围的代码。双引号（`"""`）前的空格会告诉 Swift 其他行前应该有多少空白是需要忽略的
- 如果你在某行的空格超过了结束的双引号（`"""`），那么这些空格会被包含

```
let linesWithIndentation = """  
    This line doesn't begin with whitespace.  
    This line begins with four spaces.  
    This line doesn't begin with whitespace.  
    """
```

Space ignored ————

Appears in string ————

字符串里的特殊字符

- 转义特殊字符 `\0` (空字符), `\\` (反斜杠), `\t` (水平制表符), `\n` (换行符), `\r` (回车符), `\"` (双引号) 以及 `\'` (单引号)
- 任意的 Unicode 标量, 写作 `\u{n}`, 里边的 `n` 是一个 1-8 位的16 进制数字, 其值是合法 Unicode 值
- 可以在多行字符串字面量中包含双引号 (`"`) 而不需转义。要在多行字符串中包含文本 `"""`, 转义至少一个双引号

```
let wiseWords = "\"Imagination is more important than knowledge\" -  
    Einstein"  
// "Imagination is more important than knowledge" - Einstein  
let dollarSign = "\u{24}" // $, Unicode scalar U+0024  
let blackHeart = "\u{2665}" // ♥, Unicode scalar U+2665  
let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

```
"""Imagination is more important than knowledge" - Einstein"
```

```
"$"
```

```
"♥"
```

```
"💖"
```

扩展字符串分隔符（Raw String）

- 在字符串字面量中放置扩展分隔符来在字符串中包含特殊字符而不让它们真的生效
- 把字符串放在双引号（"）内并由井号（#）包裹
- 如果字符串里有"# 则首尾需要两个##
- 如果你需要字符串中某个特殊符号的效果，使用匹配你包裹的井号数量的井号并在前面写转义符号 \

```
38 let str = #"Line 1\nLine 2"#  
39 let str1 = #"Line 1\#nLine 2"#  
40 let str2 = ###"Line 1\###nLine 2"###  
41 print(str)  
42 print(str1)  
43 print(str2)
```



```
Line 1\nLine 2  
Line 1  
Line 2  
Line 1  
Line 2
```

字符串-操作

字符串的可变性

- var 指定的可以修改
- let 指定的不可修改
- 对比 Objective-C (NSString 和 NSMutableString)

```
var variableString = "Horse"  
variableString += " and carriage"  
// variableString is now "Horse and carriage"
```

```
let constantString = "Highlander"  
constantString += " and another Highlander"  
// this reports a compile-time error - a constant string cannot be modified
```

字符串是值类型

- String 值在传递给方法或者函数的时候会被复制过去
- 赋值给常量或者变量的时候也是一样
- Swift 编译器优化了字符串使用的资源，实际上拷贝只会在确实需要的时候才进行

```
var str: String = "abc"  
var str1 = str  
print(str == str1)  
str += "def"  
print(str)  
print(str1)  
print(str == str1)
```

```
"abc"  
"abc"  
"true\n"  
"abcdef"  
"abcdef\n"  
"abc\n"  
"false\n"
```

操作字符

- for-in 循环遍历 String 中的每一个独立的 Character
- Character 类型
- String 值可以通过传入 Character 数组来构造

```
60 for character in "Dog!🐶" {
61     print(character)
62 }
63
64 let catCharacters: [Character] = ["C",
65     "a", "t", "!", "🐱"]
66 let catString = String(catCharacters)
67 print(catString)
```

(5 times)

["C", "a", "t", "!", "🐱"]

"Cat!🐱"

"Cat!🐱\n"

D
o
g
!
🐶
Cat!🐱

字符串拼接

- 使用加运算符（ + ）创建新字符串
- 使用加赋值符号（ += ）在已经存在的 String 值末尾追加一个 String 值
- 使用 String 类型的 append() 方法来可以给一个 String 变量的末尾追加 Character 值

字符串插值

- 字符串插值是一种从混合常量、变量、字面量和表达式的字符串字面量构造新 String 值的方法
- 每一个你插入到字符串字面量的元素都要被一对圆括号包裹，然后使用反斜杠前缀
- 类似于 NSString 的 stringWithFormat 方法，但是更加简便，更强大

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is
               \((Double(multiplier) * 2.5))"
```

```
3
"3 times 2.5 is 7.5"
```


字符串插值

- 可以在扩展字符串分隔符中创建一个包含在其他情况下会被当作字符串插值的字符
- 要在使用扩展分隔符的字符串中使用字符串插值，在反斜杠后使用匹配首尾井号数量的井号

```
print("#Write an interpolated string in Swift using  
    \multiplier).")  
print("#6 times 7 is \#(6 * 7).")
```

```
"Write an interpolated string in Swift using \multiplier).\n"
```

```
"6 times 7 is 42.\n"
```


字符串-访问和修改

字符串索引

- 每一个 String 值都有相关的索引类型，String.Index，它相当于每个 Character 在字符串中的位置
- startIndex 属性来访问 String 中第一个 Character 的位置。endIndex 属性就是 String 中最后一个字符后的位置
- endIndex 属性并不是字符串下标脚本的合法实际参数
- 如果 String 为空，则 startIndex 与 endIndex 相等

```
let greeting = "Guten Tag!"  
greeting[greeting.startIndex]
```

```
"Guten Tag!"  
"G"
```

```
let greeting = "Guten Tag!"  
greeting[1]  'subscript(_:)' is unavailable: cannot subscript Strin...
```

```
/// A position of a character or code unit in a string.  
public struct Index {  
}
```

字符串索引

- 使用 `index(before:)` 和 `index(after:)` 方法来访问给定索引的前后
- 要访问给定索引更远的索引，你可以使用 `index(_:offsetBy:)`
- 使用 `indices` 属性来访问字符串中每个字符的索引

```
let greeting = "Guten Tag!"  
greeting[greeting.startIndex]  
greeting[greeting.index(before: greeting endIndex)]  
greeting[greeting.index(after: greeting.startIndex)]  
let index = greeting.index(greeting.startIndex,  
    offsetBy: 7)  
greeting[index]
```

```
"Guten Tag!"  
"G"  
"!"  
"u"  
String.Index  
  
"a"
```

插入

- 插入字符，使用 `insert(_:at:)` 方法
- 插入另一个字符串的内容到特定的索引，使用 `insert(contentsOf:at:)` 方法

```
var welcome = "hello"  
welcome.insert("!", at:  
    welcome.endIndex)  
  
welcome.insert(contentsOf: " there",  
    at: welcome.index(before:  
        welcome.endIndex))
```

```
"hello"  
"hello!"
```

```
"hello there!"
```

删除

- 移除字符，使用 `remove(at:)` 方法
- 移除一小段特定范围的字符串，使用 `removeSubrange(_:)` 方法

```
welcome.remove(at: welcome.index(before:
    welcome.endIndex))
print(welcome)
let range = welcome.index(welcome.endIndex,
    offsetBy: -6)..
```

```
"!"
"hello there\n"
{[_rawBits 327936], [_rawBits 720896]}
"hello"
"hello\n"
```

字符串-子串和字符串比较

子字符串

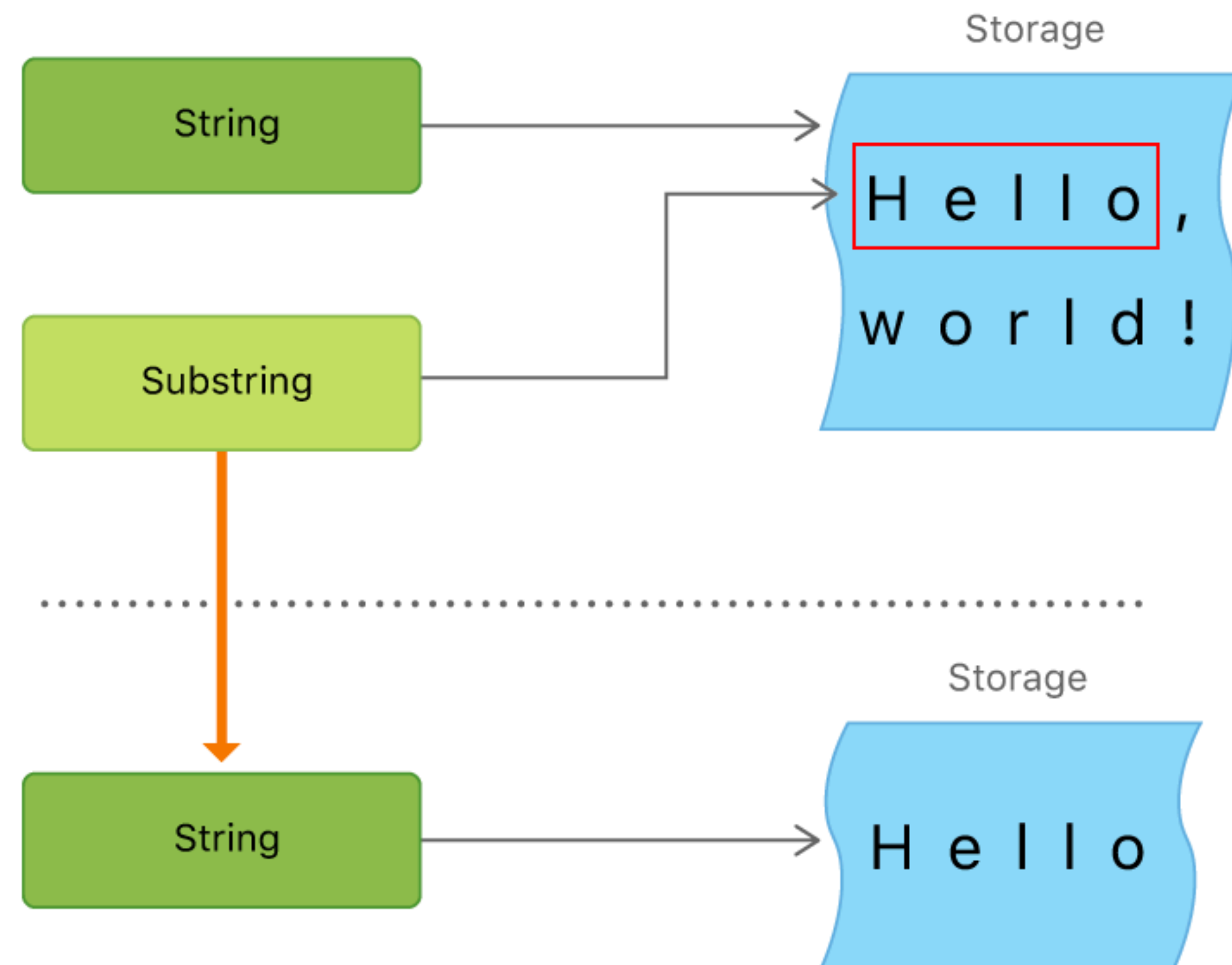
- 使用下标或者类似 `prefix(_:)` 的方法得到的子字符串是 `Substring` 类型
- `Substring` 拥有 `String` 的大部分方法
- `Substring` 可以转成 `String` 类型

```
let greeting = "Hello, world!"  
let index = greeting.index(of: ",") ?? greeting endIndex  
let beginning = greeting[..<index]  
let newString = String(beginning)
```

```
"Hello, world!"  
String.Index  
"Hello"  
"Hello"
```


子字符串

- 子字符串重用一部分原字符串的内存
- 修改字符串或者子字符串之前都不需要花费拷贝内存的代价
- String 和 Substring 都遵循 StringProtocol 协议，也就是说它基本上能很方便地兼容所有接受 StringProtocol 值的字符串操作函数



字符串比较

- 字符串和字符相等性 (==和!=)
- 前缀相等性 hasPrefix(_:)
- 后缀相等性 hasSuffix(_:)

```
var welcome = "hello, world"  
var welcome1 = "hello"  
print(welcome == welcome1)  
print(welcome.hasPrefix("hello"))  
print(welcome.hasSuffix("world"))
```

```
"hello, world"  
"hello"  
"false\n"  
"true\n"  
"true\n"
```



扫码试看/订阅
《Swift核心技术与实战》视频课程