The Basics

数据类型

基本类型

- 整型 `Int`
- 浮点 `Double` 和 `Float`
- 布尔 `Bool`
- 字符串 `String`

行末不需要分号（除非在一行写多个表达式）

常量与变量

常量/变量与一个标识符和一个类型相关联

- `let` 定义常量
- `var` 定义变量

可以用 `variable:type` 的形式来显式注明类型（否则编译器根据初始值推断类型）

```
var welcomeMessage: String = "Hello"
```

标识符可以是任意 Unicode

一旦声明就不能再次声明或是改变类型或是在常量/变量间转换

可以用 `print(_:separator:terminator:)` 打印变量，可以用 string interpolation 来打印（用括号把标识符包起来，用 backslash 来 escape 左括号）

```
print("The current value of friendlyWelcome is \
(friendlyWelcome)")
// Prints "The current value of friendlyWelcome is Bonjour!"
```

整型

整型可以有多种类型，有符号或无符号，类型名形如 `UInt8` ， `Int32`

可以用 `Int32.min` 获取类型的上下界

`Int` 大小与当前平台有关，32-bit 为 `Int32`，64-bit 为 `Int64`，无符号整型 `UInt` 同理

如非必要，尽量全部使用 `Int`

类型安全

swift 是类型安全的，编译时会进行类型检查

在没有显式注明类型的情况下，编译器会进行类型推断（根据第一次赋值）

数字字面量

- 二进制以 `0b` 开头
- 十六进制以 `0x` 开头
- 八进制以 `0o` 开头

浮点数可以是十进制或者十六进制（`0x` 开头），采用科学计数法时十进制用 `e`，十六进制用 `p`

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

字面值可以用下划线分割，提高可读性

数字类型转换

对一个整型进行超出范围的赋值会在编译期报错

运算操作数类型不同时需要对现有的值进行显式的类型转换（不论是不同大小的整型还是整型和浮点型）

将浮点转换为整型时总是会进行截断

类型别名

用 `typealias` 关键字来为类型定义别名

```
typealias AudioSample = UInt16
```

提高代码可读性

布尔值

两个常量 `true/false`

非布尔值不能代替布尔值（如在条件语句中）

注释

双斜杠 `//`

多行注释（可嵌套）

```
/* multiline
comments */
```

元组

tuple 将多个值打包在一起

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

tuple 的类型可以是任意类型的任意组合，也可以将其分解，如果只需要一部分值，可以用 `_` 代替不需要的值

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404"
```

也可以用下标来访问 tuple 的值

```
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

或是命名后用名来访问

```
let http200Status = (statusCode: 200, description: "OK")
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is \(http200Status.description)")
// Prints "The status message is OK"
```

Optional

当一个值可能不存在时，可以使用 optional 变量，optional 变量有两种情况

- 值存在，此时可以用这个值

- 值不存在，此时变量为空

e. g. 使用 `Int` 来将字符串转换为整型时返回的就是 optional int（因为转换可能失败，参数可能不是合法的整型字符串），记为 `Int?`

一个 optional 变量可以赋值为其应有的类型的值，或是 `nil`

`nil` 不能赋值给 non-optional 的变量/常量

可以用 if 语句处理 optional

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \
(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

在变量后加 `!` 表示确定其非 `nil`，可以使用其中的值

也可以用 if 和 let 将 optional 的值和一个常量 binding

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer
value of \(actualNumber)")
} else {
    print("The string \"\(possibleNumber)\" could not be
converted to an integer")
}
// Prints "The string "123" has an integer value of 123"
```

如果非 `nil`，进入 if 分支并且其值被绑定到 let 后的常量，否则进入 else 分支（也可以用 var 绑定，如果需要修改的话）

一个 if 可以包含多个 binding，关系为 and（即有一个为 `nil` 则不能进入 if 分支）

当可以通过程序推断出一个 optional 非 `nil` 的话，便可以将其隐式地展开，只需声明时将 `?` 换成 `!`

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an
exclamation point

let assumedString: String! = "An implicitly unwrapped optional
string."
let implicitString: String = assumedString // no need for an
exclamation point
```

声明为隐式展开的 optional，当不能用作 optional 时才会强制展开，如果其值为 `nil`，则会产生 runtime error（同展开一个值为 `nil` 的正常 optional），也可以同正常 optional 一样用 if 去检查其值或 binding

错误处理

传递异常

对于会抛出异常的函数，定义时加入 `throws` 关键字

调用会抛出异常的函数要用 `try` 关键字，并用 `catch` 处理抛出的异常

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

可以 catch 不同的异常来分别处理

断言与前提

运行时的 check，不满足则终止程序

不同于异常机制是为了恢复错误，失败的断言表示当前程序无效，无法恢复

- assertion 只在 debug 模式执行
- precondition 在 production 模式也会执行

断言的函数为 `assert(_:_:file:line:)`，前两个参数是要检查的表达式和错误时的提示，提示信息也可以省略

当代码已经完成了检查的过程，可以直接调用 `assertionFailure(_:file:line:)`

前提的函数为 `precondition(_:_:file:line:)`，必须为真才能继续执行，同理也有 `preconditionFailure(_:file:line:)`

可以用 `fatalError(_:file:line:)` 在任何情况下停止程序