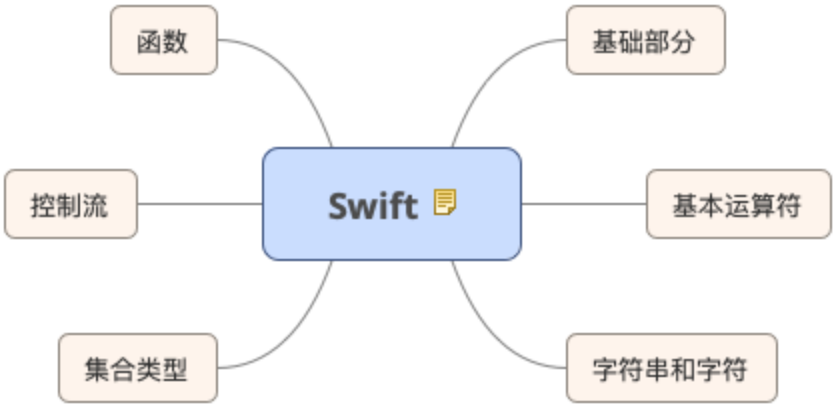
Swift

Swift 1
1. 基础部分 5
1.1. 常量(let)、变量(var) 6
1.1.1. 声明常量和变量 6
1.1.2. 类型注解 7
1.1.3. 常量和变量的命名 7
1.1.4. 输出常量和变量 7
1.2. 注释 7
1.3. 分号 7
1.4. 整数 8
1.4.1. 整数范围 8
1.4.2. int 8
1.4.3. Uint 8
1.5. 浮点数 Double/Float 8
1.6. 类型安全和类型推断 8
1.7. 数值型字面量 9
1.8. 数值型类型转换 9
1.8.1. 整数转换 9
1.8.2. 整数和浮点数转换 9
1.9. 类型别名(类比C++相关) 9
1.10. 布尔值 10
1.11. 元组 10
1.12. 可选类型 10
1.12.1. nil 10
1.12.2. if 语句以及强制解析 11
1.12.3. 可选绑定 11
1.12.4. 隐式解析可选类型 11
1.13. 错误处理 11
1.14. 断言和先决条件 11
1.15. 强制执行先决条件 12
2. 基本运算符 12
2.1. 赋值运算符(注意、小心) 12
2.2. 算术运算符 12
2.2.1. 求余运算符 12
2.2.2. 一元负号运算符 13

2.2.3. 一元正号运算符 13
2.3. 组合赋值运算符 13
2.4. 比较运算符（Comparison Operators） 13
2.5. 三元运算符（Ternary Conditional Operator） 13
2.6. 空合运算符（Nil Coalescing Operator） 13
2.7. 区间运算符（Range Operators) 13
2.7.1. 闭区间运算符 13
2.7.2. 半开区间运算符 13
2.7.3. 单侧区间 13
2.8. 逻辑运算符（Logical Operators） 13
2.8.1. 逻辑非运算符 14
2.8.2. 逻辑与运算符 14
2.8.3. 逻辑或运算符 14
2.8.4. 逻辑运算符组合计算 14
2.8.5. 使用括号来明确优先级 14
3. 字符串和字符 14
3.1. 字符串字面量 15
3.1.1. 多行字符串字面量 16
3.1.2. 字符串字面量的特殊字符 16
3.1.3. 扩展字符串分隔符 16
3.2. 初始化空字符串 16
3.3. 字符串可变性 16
3.4. 字符串是值类型 16
3.5. 使用字符 16
3.6. 连接字符串和字符 16
3.7. 字符串插值 16
3.8. Unicode 16
3.8.1. Unicode 标量 17
3.8.2. 可扩展的字形群集 17
3.9. 计算字符数量 17
3.10. 访问和修改字符串 17
3.10.1. 字符串索引 17
3.10.2. 插入和删除 17
3.11. 子字符串 17
3.12. 比较字符串 17
3.12.1. 字符串/字符相等 17
3.12.2. 前缀/后缀相等 18
3.12.3. 字符串的 Unicode 表示形式 18

4. 集合类型 18
4.1. 集合的可变性 18
4.2. 数组（Arrays） 18
4.2.1. 数组的简单语法 19
4.2.2. 创建一个空数组 19
4.2.3. 创建一个带有默认值的数组 19
4.2.4. 通过两个数组相加创建一个数组 19
4.2.5. 用数组字面量构造数组 19
4.2.6. 访问和修改数组 19
4.2.7. 数组的遍历 19
4.3. 集合（Sets） 19
4.3.1. 集合类型的哈希值 20
4.3.2. 集合类型语法Set<Element> 20
4.3.3. 创建和构造一个空的集合 20
4.3.4. 用数组字面量创建集合 20
4.3.5. 访问和修改一个集合 20
4.3.6. 遍历一个集合 20
4.4. 集合操作(有需要要进一步复习记忆) 20
4.4.1. 基本集合操作 20
4.4.2. 集合成员关系和相等 20
4.5. 字典 20
4.5.1. 字典类型简化语法 Dictionary<Key, Value> 21
4.5.2. 创建一个空字典 21
4.5.3. 用字典字面量创建字典 21
4.5.4. 访问和修改字典 21
4.5.5. 字典遍历 21
5. 控制流 21
5.1. For-In 循环 21
5.2. While 循环 21
5.2.1. While 22
5.2.2. Repeat-While 22
5.3. 条件语句 22
5.3.1. If 22
5.3.2. Switch 22
5.4. 控制转移语句 23
5.4.1. Continue 23
5.4.2. Break 23
5.4.3. 贯穿（Fallthrough） 23

5.4.4. 带标签的语句 23
5.5. *提前退出 guard待进一步学习 注：where条件关键字用 , 代替了 23
5.6. 检测 API 可用性 25
6. 函数 25



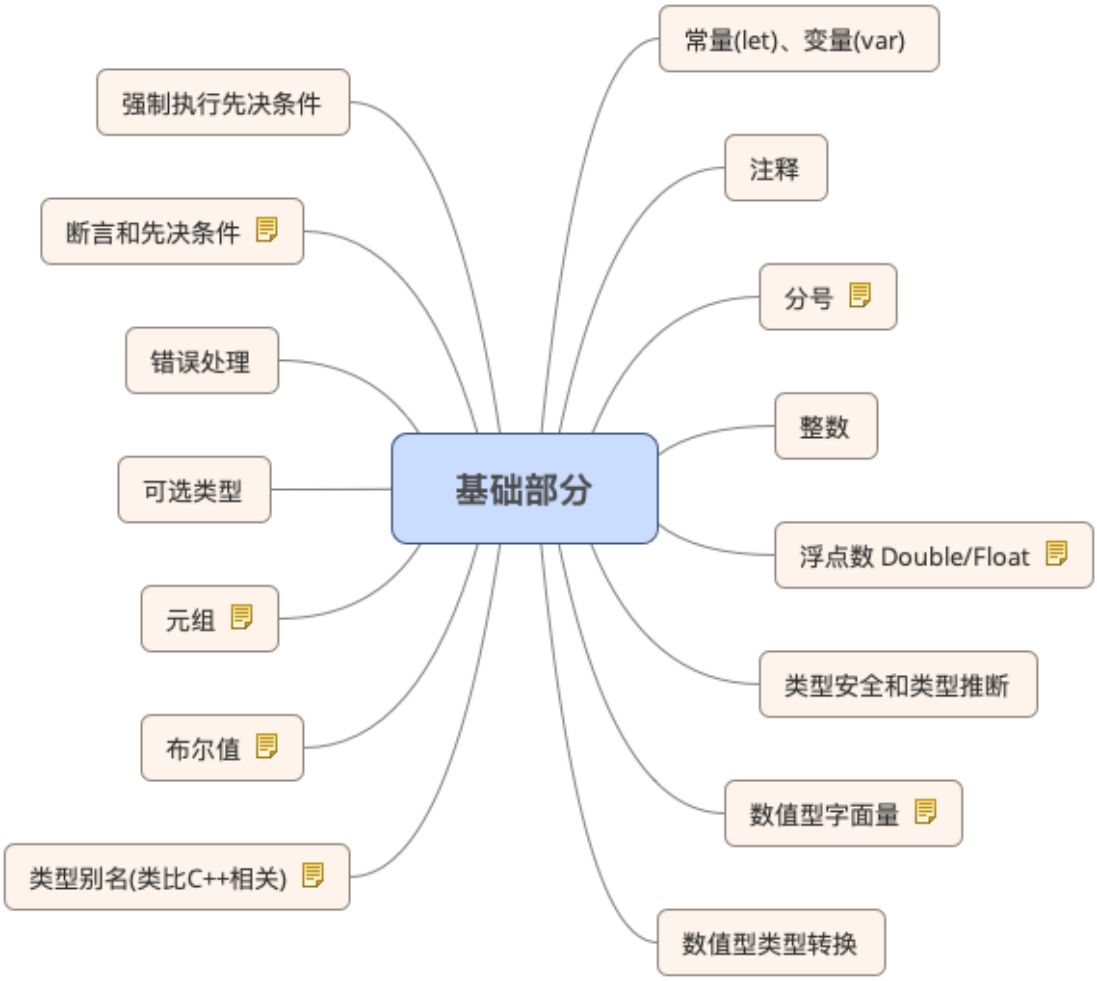
https://swiftgg.gitbook.io/swift/swift-jiao-cheng

Playground

Command+Shit+Enter：运行整个Playground

Shift+Enter：运行截止到某一行代码

1. 基础部分



1.1. 常量(let)、变量(var)

1.1.1. 声明常量和变量

eg:let maximumNumberOfLoginAttempts = 10

1.1.2. 类型注解

eg:var welcomeMessage: String

1.1.3. 常量和变量的命名

eg:let π=3.14
let □=cow
var friendlyWelcome = "hello"
friendlyWelcome="加菲猫"

1.1.4. 输出常量和变量

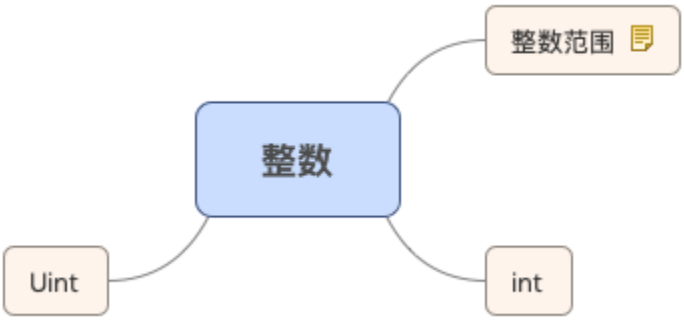
```
import UIKit

var str = "Hello, playground by:nixs Nice"
print(str)

var welcomeMessage: String = "Nice to meet you."
welcomeMessage = "倪新生"
print(welcomeMessage)

let π=3.14
let □="cow"
var friendlyWelcome = "hello"
friendlyWelcome="加菲猫"
print(friendlyWelcome)
print("字符串拼接 \(welcomeMessage) \(π) \(□)")
```

1.2. 注释

1.3. 分号

```
let cat:String="little cat";print(cat)
```

1.4. 整数



1.4.1. 整数范围

```
let minValue = UInt8.min  // minValue 为 0，是 UInt8 类型
let maxValue = UInt8.max  // maxValue 为 255，是 UInt8 类型
print("minValue:\(minValue) ; maxValue:\(maxValue)")
```

1.4.2. int

1.4.3. Uint

1.5. 浮点数 Double/Float

浮点数是有小数部分的数字，比如 3.14159、0.1 和 -273.15。

浮点类型比整数类型表示的范围更大，可以存储比 Int 类型更大或者更小的数字。Swift 提供了两种有符号浮点数类型：

Double 表示64位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。

Float 表示32位浮点数。精度要求不高的话可以使用此类型。

注意

Double 精确度很高，至少有15位数字，而 Float 只有6位数字。选择哪个类型取决于你的代码需要处理的值的范围，在两种类型都匹配的情况下，将优先选择 Double

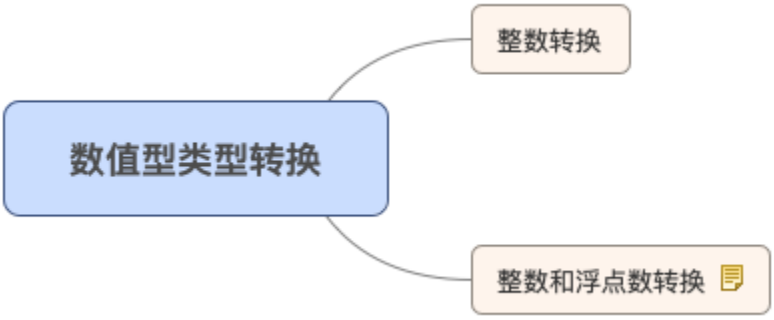
1.6. 类型安全和类型推断

1.7. 数值型字面量

整数字面量可以被写作：
一个十进制数，没有前缀
一个二进制数，前缀是 0b
一个八进制数，前缀是 0o
一个十六进制数，前缀是 0x
下面的所有整数字面量的十进制值都是 17:

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 二进制的17
let octalInteger = 0o21           // 八进制的17
let hexadecimalInteger = 0x11     // 十六进制的17
```

1.8. 数值型类型转换



1.8.1. 整数转换

1.8.2. 整数和浮点数转换

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi 等于 3.14159，所以被推测为 Double 类型
print(pi)
```

1.9. 类型别名(类比C++相关)

```
typealias AudioSample = UInt16
```

```
print(AudioSample.max)
```

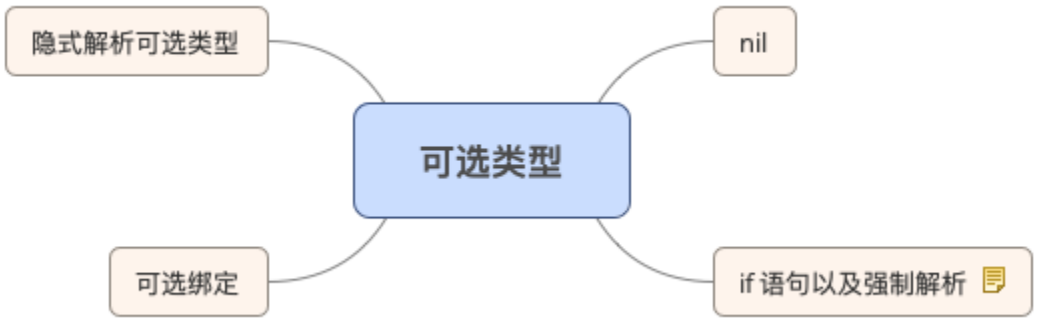
1.10. 布尔值

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

1.11. 元组

```
let http404Error = (404,"Not Found")
let (statusCode,statusMsg) = http404Error

print("The status code is \(statusCode)")
print("The status code is \(statusMsg)")

//let rgbColor = (r:(223/255),g:(124/255),b:(25/255))
let rgbColor = (r:5,g:8.90,b:3.14)
print("The color r is \(rgbColor.r)")
print("The color g is \(rgbColor.g)")
print("The color b is \(rgbColor.b)")

let stu = (name:"nixs",sex:"male",age:28)
print("name of stu is \(stu.name)")
print("name of sex is \(stu.sex)")
print("name of age is \(stu.age)")
```

1.12. 可选类型



1.12.1. nil

1.12.2. if 语句以及强制解析

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber 被推测为类型 "Int?"， 或者类型 "optional Int"

var serverResponseCode: Int? = 404
// serverResponseCode 包含一个可选的 Int 值 404
serverResponseCode = nil
// serverResponseCode 现在不包含值

var surveyAnswer: String?
// surveyAnswer 被自动设置为 nil

if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// 输出“convertedNumber contains some integer value.”
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// 输出“convertedNumber has an integer value of 123.”
```

1.12.3. 可选绑定

1.12.4. 隐式解析可选类型

1.13. 错误处理

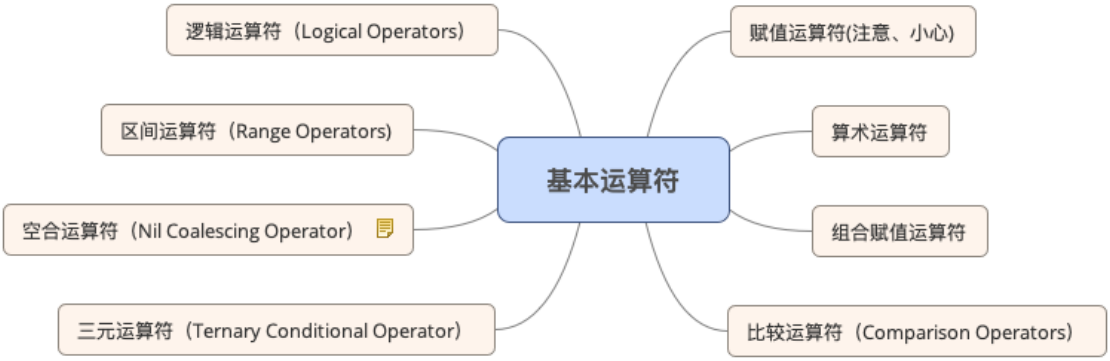
1.14. 断言和先决条件

```
let age = -3
assert(age <= 0, "A person's age cannot be less than zero")
assert(age >= 0, "A person's age cannot be less than zero")
// 因为 age < 0，所以断言会触发,程序就不会往下走了

if age > 10 {
    print("You can ride the roller-coaster or the ferris wheel.")
```

```
} else if age > 0 {
    print("You can ride the ferris wheel.")
} else {
    assertionFailure("A person's age can't be less than zero.")
}
```

1.15. 强制执行先决条件

2. 基本运算符



2.1. 赋值运算符(注意、小心)

2.2. 算术运算符

2.2.1. 求余运算符

求余运算符（a % b）是计算 b 的多少倍刚刚好可以容入 a，返回多出来的那部分（余数）。

注意

求余运算符（%）在其他语言也叫取模运算符。但是严格说来，我们看该运算符对负数的操作结果，「求余」比「取模」更合适些。

2.2.2. 一元负号运算符

2.2.3. 一元正号运算符

2.3. 组合赋值运算符

2.4. 比较运算符（Comparison Operators）

2.5. 三元运算符（Ternary Conditional Operator）

2.6. 空合运算符（Nil Coalescing Operator）

```
let defaultColorName = "red"
var userDefinedColorName: String?   //默认值为 nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName 的值为空，所以 colorNameToUse 的值为 "red"
print(colorNameToUse)
```

2.7. 区间运算符（Range Operators)

2.7.1. 闭区间运算符

2.7.2. 半开区间运算符

2.7.3. 单侧区间

2.8. 逻辑运算符（Logical Operators）

2.8.1. 逻辑非运算符

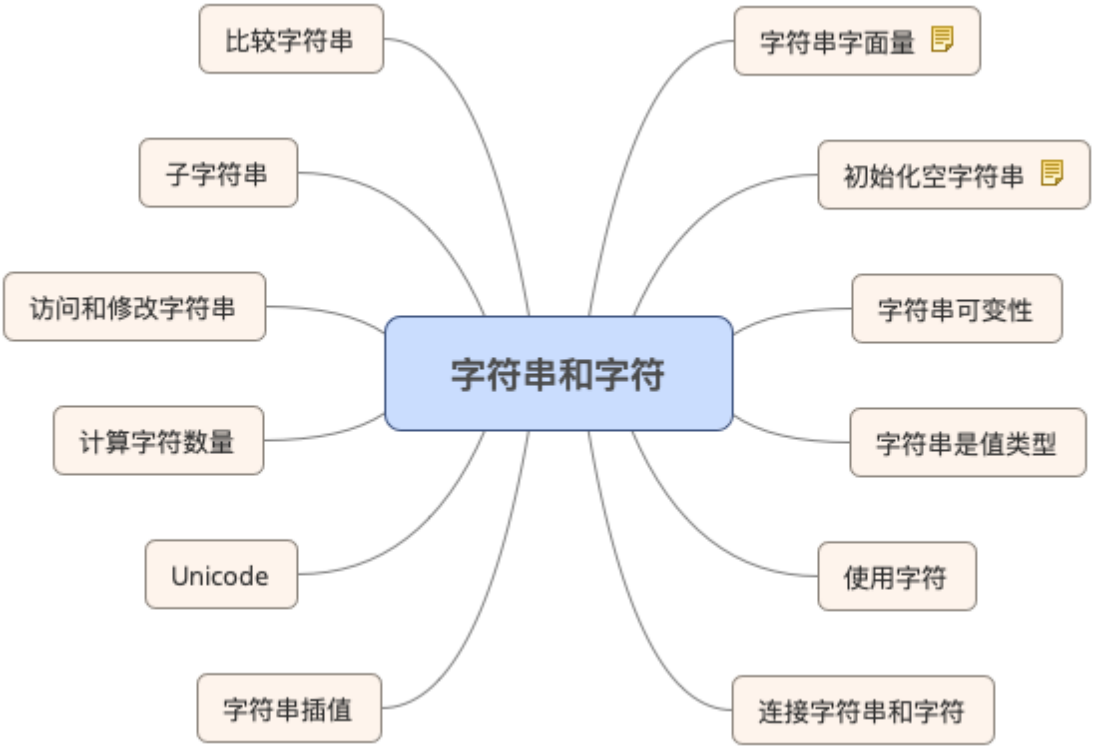
2.8.2. 逻辑与运算符

2.8.3. 逻辑或运算符

2.8.4. 逻辑运算符组合计算

2.8.5. 使用括号来明确优先级

3. 字符串和字符



3.1. 字符串字面量


```
import UIKit

let quotation = """
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."
```

```
"牛叉叉"
"""
print(quotation)
```

3.1.1. 多行字符串字面量

3.1.2. 字符串字面量的特殊字符

3.1.3. 扩展字符串分隔符

3.2. 初始化空字符串

```
var emptyString = ""               // 空字符串字面量
var anotherEmptyString = String()  // 初始化方法
// 两个字符串均为空并等价。
```

3.3. 字符串可变性

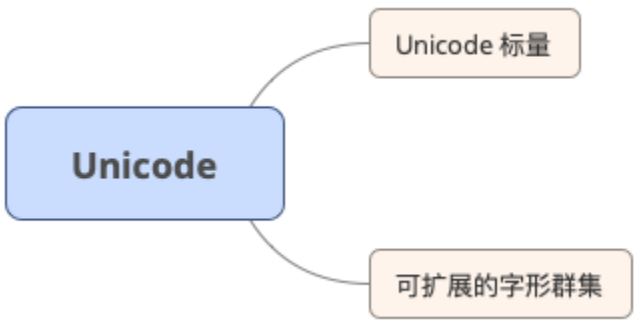
3.4. 字符串是值类型

3.5. 使用字符

3.6. 连接字符串和字符

3.7. 字符串插值

3.8. Unicode

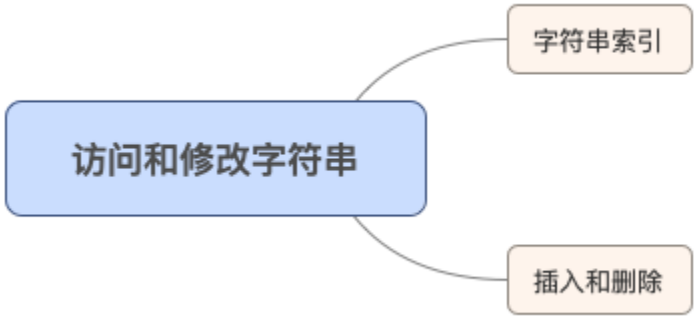


3.8.1. Unicode 标量

3.8.2. 可扩展的字形群集

3.9. 计算字符数量

3.10. 访问和修改字符串



3.10.1. 字符串索引

3.10.2. 插入和删除

3.11. 子字符串

3.12. 比较字符串

3.12.1. 字符串/字符相等

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
```

```
if quotation == sameQuotation {
    print("These two strings are considered equal")
}
// 打印输出“These two strings are considered equal”
```

3.12.2. 前缀/后缀相等

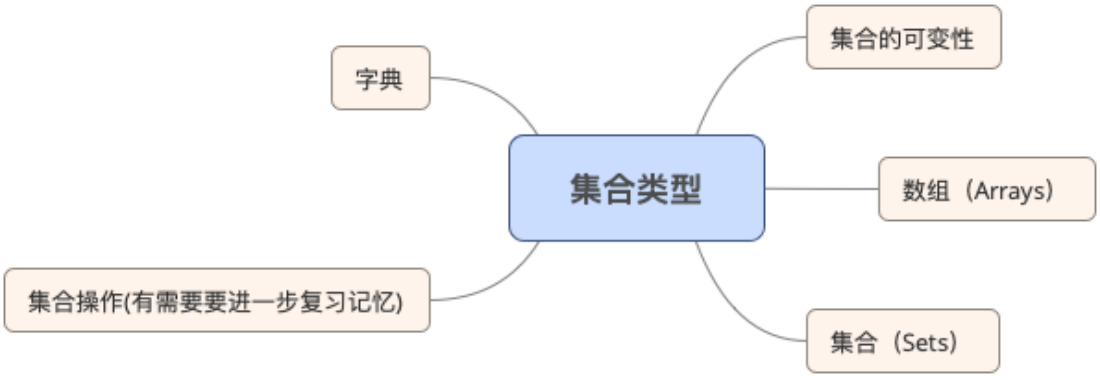
3.12.3. 字符串的 Unicode 表示形式

UTF-8 表示

UTF-16 表示

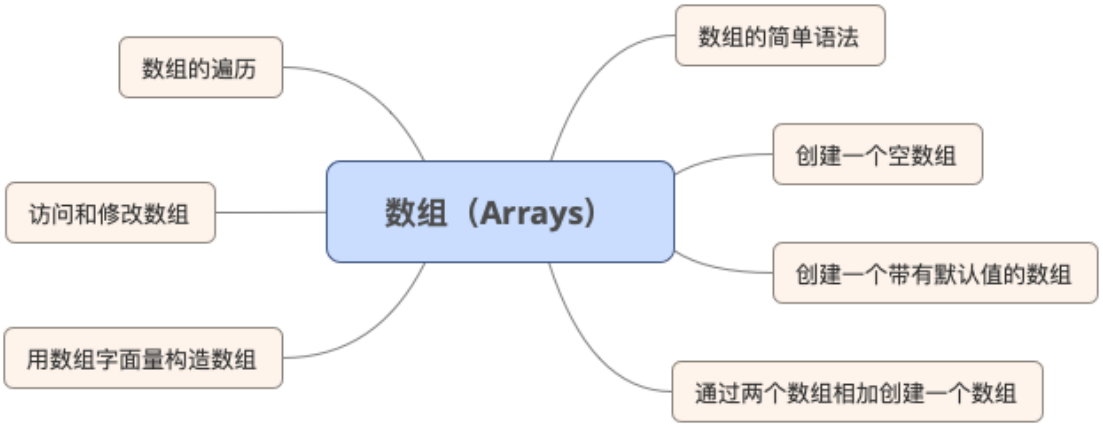
Unicode 标量表示

4. 集合类型



4.1. 集合的可变性

4.2. 数组（Arrays）



4.2.1. 数组的简单语法

4.2.2. 创建一个空数组

4.2.3. 创建一个带有默认值的数组

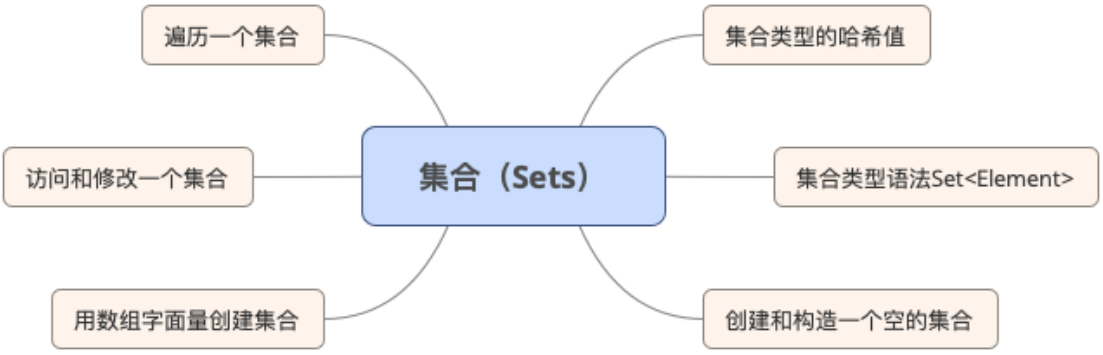
4.2.4. 通过两个数组相加创建一个数组

4.2.5. 用数组字面量构造数组

4.2.6. 访问和修改数组

4.2.7. 数组的遍历

4.3. 集合（Sets）



4.3.1. 集合类型的哈希值

4.3.2. 集合类型语法Set<Element>

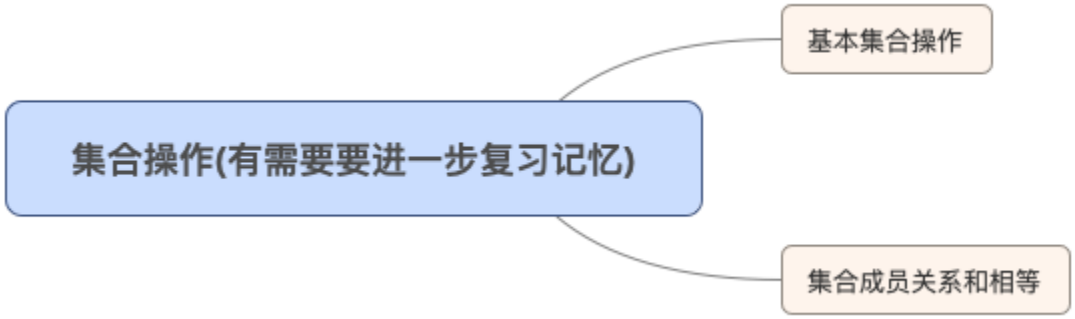
4.3.3. 创建和构造一个空的集合

4.3.4. 用数组字面量创建集合

4.3.5. 访问和修改一个集合

4.3.6. 遍历一个集合

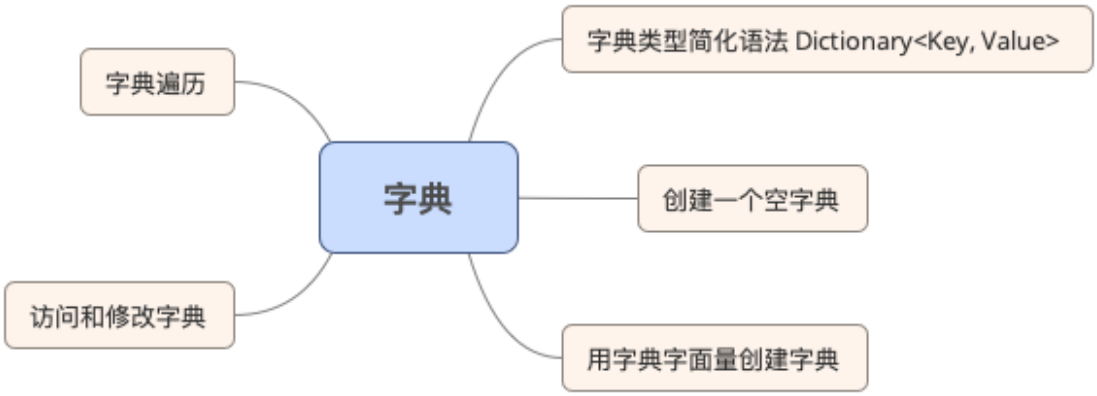
4.4. 集合操作(有需要要进一步复习记忆)



4.4.1. 基本集合操作

4.4.2. 集合成员关系和相等

4.5. 字典



4.5.1. 字典类型简化语法 Dictionary<Key, Value>

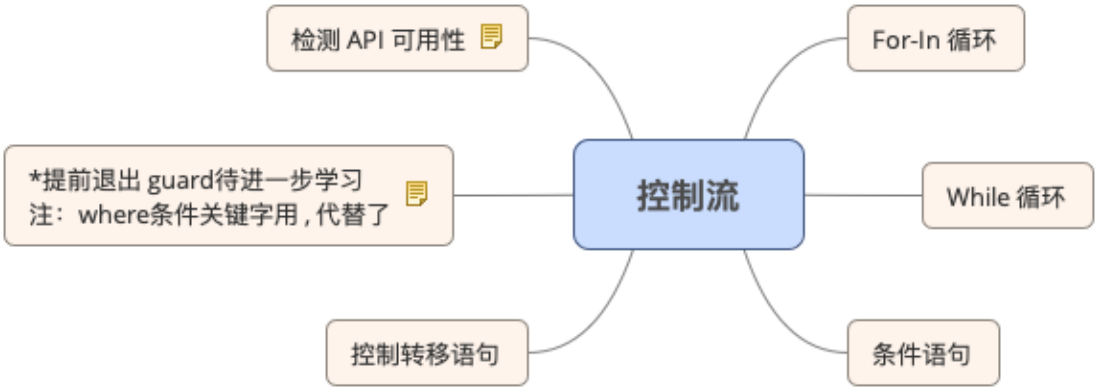
4.5.2. 创建一个空字典

4.5.3. 用字典字面量创建字典

4.5.4. 访问和修改字典

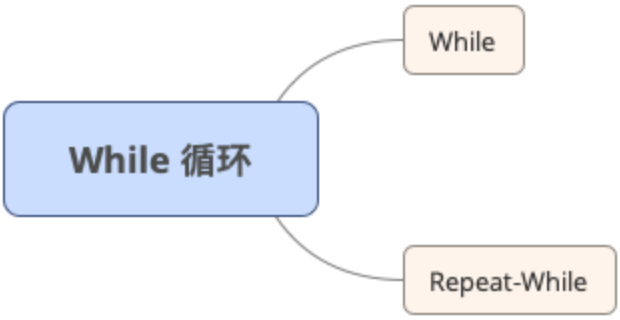
4.5.5. 字典遍历

5. 控制流



5.1. For-In 循环

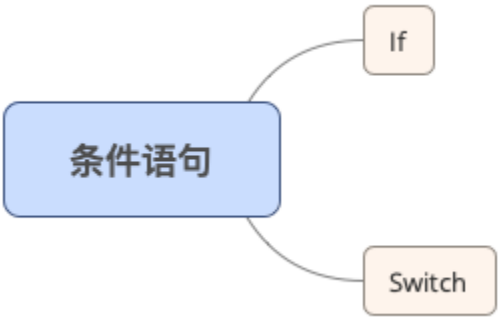
5.2. While 循环



5.2.1. While

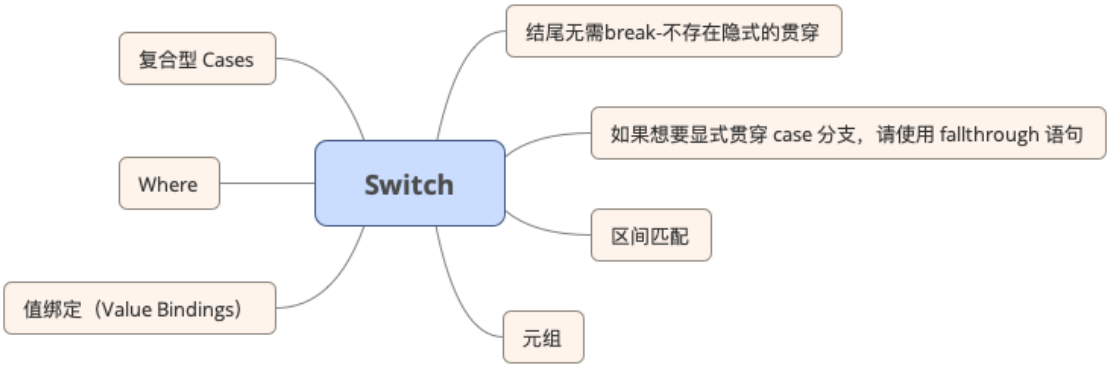
5.2.2. Repeat-While

5.3. 条件语句



5.3.1. If

5.3.2. Switch



结尾无需**break**-不存在隐式的贯穿

如果想要显式贯穿 **case** 分支，请使用 **fallthrough** 语句

区间匹配

元组

值绑定（**Value Bindings**）

Where

复合型 **Cases**

5.4. 控制转移语句

5.4.1. Continue

5.4.2. Break

5.4.3. 贯穿（Fallthrough）

5.4.4. 带标签的语句

5.5. *提前退出 guard待进一步学习

注：**where**条件关键字用 **,** 代替了

参考1：https://www.jianshu.com/p/df9431e1940a

参考2：https://www.jianshu.com/p/9fff7621ed75

swift 中 guard 关键字的使用

注意事项：

1.guard关键字必须使用在函数中。

2.guard关键字必须和else同时出现。

3.guard关键字只有条件为false的时候才能走else语句 相反执行后边语句。

例子：

```
class CloseRange{
    let start: Int
    let end: Int
    init?(startValue: Int , endValue: Int) {

        guard startValue < endValue else {
            print("结束值 应大于 起始值")
            return nil
        }

        self.start = startValue
        self.end = endValue
    }
}

let customRange = CloseRange(startValue: 3, endValue: 5)
if let customRange = customRange {
    print("第一个对象的起始值是：\(customRange.start)")
}

let customRange2 = CloseRange (startValue: 5, endValue: 3)
if let customRange = customRange2 {
    print("第二个对象的起始值是：\(customRange.start)")
}else{
    print("第二个对象是空对象");
}
```

输出结果是

第一个对象的起始值是：3

结束值 应大于 起始值

第二个对象是空对象

5.6. 检测 API 可用性

```
if #available(平台名称 版本号, ..., *) {
   APIs 可用，语句将执行
} else {
   APIs 不可用，语句将不执行
}
eg:
if #available(iOS 10, macOS 10.12, *) {
   // 在 iOS 使用 iOS 10 的 API, 在 macOS 使用 macOS 10.12 的 API
} else {
   // 使用先前版本的 iOS 和 macOS 的 API
}
```

6. 函数