字符变成大写的方法：

```
var newString = "Hello".map { (c) -> String.Element in
    return c.uppercased().first!
}
print(String(newString))  //HELLO
```

filter 方法用来进行字符串中字符的过滤，其需要传入一个闭包参数，字符串依次遍历出的字符会作为闭包的参数，闭包需要返回一个布尔值，如果返回布尔值 false，则当前字符会被过滤掉，下面的代码会过滤掉字符串中所有的大写字母：

```
var newString = "Hello".filter { (c) -> Bool in
    if c.asciiValue! <= "Z".first!.asciiValue! &&  c.asciiValue! >=
"A".first!.asciiValue! {
        return false
    }
    return true
}
print(newString)  //ello
```

reduce 函数也被称为累加器，其第 1 个参数为累加前的初始结果，第 2 个参数为闭包，闭包中会将上一次执行累加后的结果和遍历出的字符作为参数传入。例如，下面的代码会在字符串的每个字符前插入感叹号：

```
var newString = "Hello".reduce("") { (result, c) -> String in
    return result + "!" + String(c)

}
print(newString)  //!H!e!l!l!o
```

其实，上面列举的方法并非是 String 所独有的。在 Swift 中，集合类型都可以调用这些迭代方法。与 map 方法类似的还有 flatMap 和 compactMap。flatMap 在调用时会将返回的二维集合进行降维，即可以将二维数组中的元素全部合并到一个数组中。compactMap 方法可以自动提出新集合中的 nil 值。

3.4.5　Swift 中的 Array 类型

Array 是 Swift 中非常强大的一种数据类型，只要将其声明为变量类型，就可以方便地调用方法对其增删。其实，Array 类型采用了动态扩容的方式实现可变性。示例代码如下：

```
var array:Array<Int> = [1, 2, 3]
print(array.capacity)   // 3
array.append(4)
print(array.capacity)   // 6
array.append(contentsOf: [5, 6, 7])
```

```
print(array.capacity)   // 12
array.append(contentsOf: [8, 9, 10, 11, 12, 13])
print(array.capacity)   // 24
```

数组对象的 capacity 属性用来获取数组的空间大小，也可以理解为数组中可以存放的元素的个数。与之对应的还有一个更常用的 count 属性，用来获取当前数组中元素的个数。从上面的打印信息可以看到，初创的数组分配的内存空间的大小刚好可以存放数组中已有的元素，如果进行元素的追加，则数组会扩容成元素组容量的两倍。之后，每当数组容量不够追加新的元素时，都会进行 2 倍的扩容。

Array 类型在 Swift 中是以结构体的方式实现的，因此其是值类型，通过前面的学习，我们知道值类型在赋值时会被复制，其实在实际操作中并非所有值类型数据的赋值都会产生深复制，否则将产生极大的内存浪费。Swift 采用了写时复制的技术解决资源的优化问题。例如，创建如下两个数组变量：

```
var array1 = [1, 2, 3]
var array2 = array1
print(array1, array2)
```

在 print 语句中添加一个断点，当程序中断时，在 lldb 控制台中执行如下两条指令来打印数组变量的内部数据：

```
frame variable -R array1
frame variable -R array2
```

控制台输出信息如下：

```
(lldb) frame variable -R array1
(Swift.Array<Swift.Int>) array1 = {
  _buffer = {
    _storage = {
      rawValue = 0x0000000102968d70 {
        Swift.__ContiguousArrayStorageBase = {
          Swift.__SwiftNativeNSArrayWithContiguousStorage = {
            Swift.__SwiftNativeNSArray = {}
          }
          countAndCapacity = {
            _storage = {
              count = {
                _value = 3
              }
              _capacityAndFlags = {
                _value = 6
              }
            }
```

```
            }
        }
    }
  }
}
(lldb) frame variable -R array2
(Swift.Array<Swift.Int>) array2 = {
_buffer = {
  _storage = {
    rawValue = 0x0000000102968d70 {
      Swift.__ContiguousArrayStorageBase = {
        Swift.__SwiftNativeNSArrayWithContiguousStorage = {
          Swift.__SwiftNativeNSArray = {}
        }
        countAndCapacity = {
          _storage = {
            count = {
              _value = 3
            }
            _capacityAndFlags = {
              _value = 6
            }
          }
        }
      }
    }
  }
}
```

在上面打印出的信息里，我们只需要关注 rawValue 的值即可，表示真实的数组数据所在地址，可以看到 array1 和 array2 变量的 rawValue 地址都是 0x0000000102968d70，即此时并没有产生复制操作，如果对其中一个数组进行了修改，则情况会发生变化，具体如下：

```
var array1 = [1, 2, 3]
var array2 = array1
print(array1, array2)
array2.append(4)
print(array1, array2)
```

在最后一行 print 语句处添加断点，重复上面的操作，通过控制台的输出可以看到此时 array1 和 array2 已经变成完全不同的数组对象，这就是写时复制技术的魅力所在。在 Swift 中，

其实所有值类型的传递都有这样的特性，因此虽然在 Swift 中 String、Array 和 Dictionary 等这类数据类型都是值类型，但是在使用时可以放心地进行传递，无须考虑额外的资源消耗，Swift 通过写时复制技术确保只有在真正需要时才会进行深复制。

3.4.6 Swift 中的 Dictionary 类型

Dictionary 在 Swift 中也是通过结构体实现的，通常在使用字典时都会用字符串作为字典中键的类型，例如：

```
var dic = ["1" : "one" , "2" : "two", "3" : "three"]
```

与 Objective-C 类似，如果需要让自定义的类型可以作为字典中的键，则此类型的示例必须可以进行哈希，在 Swift 中需要遵守 Hashable 协议，示例如下：

```
class Index: Hashable {
    var value:Int

    func hash(into hasher: inout Hasher) {
        hasher.combine(self.value)
    }

    static func == (lhs:Index, rhs:Index) -> Bool {
        return lhs.value == rhs.value
    }

    init(value:Int) {
        self.value = value
    }
}
var dic = [Index(value: 0) : "1", Index(value: 1) : "2", Index(value: 2) : "3"]
print(dic)
```

在上面的代码中，Hashable 协议继承于 Equatable 协议，这个协议中定义了重载等于运算符的方法。

在 Objective-C 中，我们知道字典类型内部实际上是一个哈希表，并且通过关联链表的方式来处理哈希冲突。Swift 中的 Dictionary 与之类似，只是在处理哈希冲突时采用的是开放寻址法，即寻找冲突位置的下一个位置是否空闲，如果空闲就将数据放入其中。Dictionary 的动态扩容机制与 Array 基本一致，当字典中元素存满时，如果需要存储新的元素，则会按照之前容量的 2 倍标准进行扩容。