翻译：marsprince　校对：numbbbbb

声明

本页包含内容：

一条声明可以在你的程序里引入新的名字和构造。举例来说，你可以使用声明来引入函数和方法，变量和常量，或者来定义 新的命名好的枚举，结构，类和协议类型。你也可以使用一条声明来延长一个已经存在的命名好的类型的行为。或者在你的 程序里引入在其他地方声明的符号。

在swift中，大多数声明在某种意义上讲也是执行或同事声明它们的初始化定义。这意味着，因为协议和他们的成员不匹配， 大多数协议成员需要单独的声明。为了方便起见，也因为这些区别在swift里不是很重要，声明语句同时包含了声明和定义。

GRAMMAR OF A DECLARATION

declaration → import-declaration

declaration → constant-declaration

declaration → variable-declaration

declaration → typealias-declaration

declaration → function-declaration

declaration → enum-declaration

declaration → struct-declaration

declaration → class-declaration

declaration → protocol-declaration

declaration → initializer-declaration

declaration → deinitializer-declaration

declaration → extension-declaration

declaration → subscript-declaration

declaration → operator-declaration

declarations → declarationdeclarationsopt

declaration-specifiers → declaration-specifierdeclaration-specifiersopt

declaration-specifier → class | mutating | nonmutating | override | static | unowned |

模块范围

模块范围定义了对模块中其他源文件可见的代码。（注：待改进）在swift的源文件中，最高级别的代码由零个或多个语句， 声明和表达组成。变量，常量和其他的声明语句在一个源文件的最顶级被声明，使得他们对同

一模块中的每个源文件都是可见的。

GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → statements opt

代码块

代码块用来将一些声明和控制结构的语句组织在一起。它有如下的形式：

```
{
    `statements`
}
```

代码块中的语句包括声明，表达式和各种其他类型的语句，它们按照在源码中的出现顺序被依次执行。

GRAMMAR OF A CODE BLOCK

code-block → statements opt

引入声明

引入声明使你可以使用在其他文件中声明的内容。引入语句的基本形式是引入整个代码模块；它由import关键字开始，后面 紧跟一个模块名：

```
import  module
```

你可以提供更多的细节来限制引入的符号，如声明一个特殊的子模块或者在一个模块或子模块中做特殊的声明。（待改进） 当你使用了这些细节后，在当前的程序汇总只有引入的符号是可用的（并不是声明的整个模块）。

```
import import kind module.symbol name
import module.submodule
```

GRAMMAR OF AN IMPORT DECLARATION

import-declaration → attributes opt import import-kind opt import-path import-

kind → typealias | struct | class | enum | protocol | var | func import-path → import-path-identifier import-path-identifier.import-path import-path-identifier → identifier operator

常量声明

常量声明可以在你的程序里命名一个常量。常量以关键词let来声明，遵循如下的格式:

```
let constant name: type = expression
```

当常量的值被给定后，常量就将常量名称和表达式初始值不变的结合在了一起，而且不能更改。 这意味着如果常量以类的形式被初始化，类本身的内容是可以改变的，但是常量和类之间的结合关系是不能改变的。 当一个常量被声明为全局变量，它必须被给定一个初始值。当一个常量在类或者结构体中被声明时，他被认为是一个常量 属性。常量并不是可计算的属性，因此不包含getters和setters。（译者注：getters和setters不知道怎么翻译，待改进）

如果常量名是一个元祖形式，元祖中的每一项初始化表达式中都要有对应的值

```
let (firstNumber, secondNumber) = (10, 42)
```

在上例中，firstNumber是一个值为10的常量，secnodeName是一个值为42的常量。所有常量都可以独立的使用:

```
1  println("The first number is \(firstNumber).")
2  // prints "The first number is 10."
3  println("The second number is \(secondNumber).")
4  // prints "The second number is 42."
```

类型注释（:type）在常量声明中是一个可选项，它可以用来描述在类型接口（type inference）中找到的类型。

声明一个静态常量要使用关键字static。静态属性在类型属性（type propetries）中有介绍。

如果还想获得更多关于常量的信息或者想在使用中获得帮助，请查看常量

和变量（constants and variables）, 存储属性（stored properties）等节。

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → attributes opt declaration-specifiers opt letpattern-initializer-list pattern-initializer-list → pattern-initializer | pattern-initializer , pattern-initializer-list pattern-initializer → pattern initializer opt initializer → = expression

变量声明

变量声明可以在你的程序里声明一个变量，它以关键字var来声明。根据声明变量类型和值的不同，如存储和计算 变量和属性，存储变量和属性监视，和静态变量属性，有着不同的声明形式。（待改进） 所使用的声明形式取决于变量所声明的范围和你打算声明的变量类型。

注意:

你也可以在协议声明的上下文声明属性，详情参见类型属性声明。

存储型变量和存储型属性

下面的形式声明了一个存储型变量或存储型变量属性

```
var variable name: type = expression
```

你可以在全局，函数内，或者在类和结构体的声明(context)中使用这种形式来声明一个变量。当变量以这种形式 在全局或者一个函数内被声明时，它代表一个存储型变量。当他在类或者结构体中被声明时，他代表一个存储型变量属性。

构造器表达式可以被

和常量声明相比，如果变量名是一个元祖类型，元祖的每一项的名字都要和初始化表达式一致。

正如名字一样，存储型变量的值或存储型变量属性存储在内存中。

计算型变量和计算型属性

如下形式声明一个一个存储型变量或存储型属性:

```
var variable name: type {
get {
    statements
}
set(setter name) {
    statements
}
}
```

你可以在全局，函数体内或者类，结构体，枚举，扩展声明的上下文中使用这种形式的声明。 当变量以这种形式在全局或者一个函数内被声明时，它代表一个计算型变量。当他在类，结构体，枚举，扩展声明的上下文 中中被声明时，他代表一个计算型变量属性。

getter用来读取变量值，setter用来写入变量值。setter子句是可选择的，只有getter是必需的，你可以将这些语句 都省略，只是简单的直接返回请求值，正如在只读计算属性(read-only computed properites)中描述的那样。 但是如果你提供了一个setter语句，你也必需提供一个getter语句。

setter的名字和圆括号内的语句是可选的。如果你写了一个setter名，它就会作为setter的参数被使用。如果你不写setter名， setter的初始名为newValue，正如在seter声明速记(shorthand setter declaration)中提到的那样。

不像存储型变量和存储型属性那样，计算型属性和计算型变量的值不存储在内存中。

获得更多信息，查看更多关于计算型属性的例子，请查看计算型属性(computed properties)一节。

存储型变量监视器和属性监视器

你可以用willset和didset监视器来声明一个存储型变量或属性。一个包含监

视器的存储型变量或属性按如下的形式声明:

```
var variable name: type = expression {
willSet(setter name) {
    statements
}
didSet(setter name {
    statements
}
}
```

你可以在全局，函数体内或者类，结构体，枚举，扩展声明的上下文中使用这种形式的声明。 当变量以这种形式在全局或者一个函数内被声明时，监视器代表一个存储型变量监视器； 当他在类，结构体，枚举，扩展声明的上下文中被声明时，监视器代表属性监视器。

你可以为适合的监视器添加任何存储型属性。你也可以通过重写子类属性的方式为适合的监视器添加任何继承的属性 (无论是存储型还是计算型的)，参见重写属性监视器(overriding properyt observers)。

初始化表达式在类或者结构体的声明中是可选的，但是在其他地方是必需的。无论在什么地方声明， 所有包含监视器的变量声明都必须有类型注释(type annotation)。

当变量或属性的值被改变时，willset和didset监视器提供了一个监视方法（适当的回应）。 监视器不会在变量或属性第一次初始化时不会被运行，他们只有在值被外部初始化语句改变时才会被运行。

willset监视器只有在变量或属性值被改变之前运行。新的值作为一个常量经过过willset监视器，因此不可以在 willset语句中改变它。didset监视器在变量或属性值被改变后立即运行。和willset监视器相反，为了以防止你仍然 需要获得旧的数据，旧变量值或者属性会经过didset监视器。这意味着，如果你在变量或属性自身的didiset监视器语句 中设置了一个值，你设置的新值会取代刚刚在willset监视器中经过的那个值。

在willset和didset语句中，setter名和圆括号的语句是可选的。如果你写了一个setter名，它就会作为willset和didset的参数被使用。如果你不写setter

名， willset监视器初始名为newvalue，didset监视器初始名为oldvalue。

当你提供一个willset语句时，didset语句是可选的。同样的，在你提供了一个didset语句时，willset语句是可选的。

获得更多信息，查看如何使用属性监视器的例子，请查看属性监视器(prpperty observers)一节。

类和静态变量属性

class关键字用来声明类的计算型属性。static关键字用来声明类的静态变量属性。类和静态变量在类型属性(type properties)中有详细讨论。

GRAMMAR OF A VARIABLE DECLARATION

variable-declaration → variable-declaration-headpattern-initializer-list

variable-declaration → variable-declaration-head variable-name type-annotation code-block

variable-declaration → variable-declaration-head variable-name type-annotation getter-setter-block

variable-declaration → variable-declaration-head variable-name type-annotation getter-setter-keyword-block

variable-declaration → variable-declaration-head variable-name type-annotation initializer opt willSet-didSet-block

variable-declaration-head → attributes opt declaration-specifiers opt var
variable-name → identifier

getter-setter-block → {getter-clause setter-clause opt}

getter-setter-block → {setter-clause getter-clause}

getter-clause → attributes optgetcode-block

setter-clause → attributes opt set setter-name opt code-block

setter-name → (identifier)

getter-setter-keyword-block → {getter-keyword-clause setter-keyword-clause opt} getter-setter-keyword-block → {setter-keyword-clause getter-keyword-clause}

getter-keyword-clause → attributes opt get

setter-keyword-clause → attributes opt set

willSet-didSet-block → {willSet-clause didSet-clause opt}

willSet-didSet-block → {didSet-clause willSet-clause}

willSet-clause → attributes opt willSet setter-name opt code-block

didSet-clause → attributes opt didSet setter-name opt code-block

类型的别名声明

类型别名的声明可以在你的程序里为一个已存在的类型声明一个别名。类型的别名声明以关键字typealias开始，遵循如下的 形式：

```
typealias name = existing type
```

当一个类型被别名被声明后，你可以在你程序的任何地方使用别名来代替已存在的类型。已存在的类型可以是已经被命名的 类型或者是混合类型。类型的别名不产生新的类型，它只是简单的和已存在的类型做名称替换。

查看更多Protocol Associated Type Declaration.

GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → typealias-head typealias-assignment typealias-head → typealias typealias-name typealias-name → identifier typealias-assignment → =type

函数声明

你可以使用函数声明在你的程序里引入新的函数。函数可以在类的上下文，结构体，枚举，或者作为方法的协议中被声明。 函数声明使用关键字func，遵循如下的形式：

```
func function name(parameters) -> return type {
    statements
}
```

如果函数不返回任何值，返回类型可以被忽略，如下所示：

```
func function name(parameters) {
    statements
}
```

每个参数的类型都要标明，它们不能被推断出来。初始时函数的参数是常值。在这些参数前面添加var使它们成为变量， 作用域内任何对变量的改变只在函数体内有效，或者用inout使的这些改变可以在调用域内生效。更多关于in-out参数的讨论，参见in-out参数(in-out parameters)

函数可以使用元组类型作为返回值来返回多个变量。

函数定义可以出现在另一个函数声明内。这种函数被称作nested函数。更多关于nested函数的讨论，参见nestde functions。

参数名

函数的参数是一个以逗号分隔的列表 。函数调用是的变量顺序必须和函数声明时的参数顺序一致。 最简单的参数列表有着如下的形式：

```
parameter name: parameter type
```

对于函数参数来讲，参数名在函数体内被使用，而不是在函数调用时使用。对于方法参数，参数名在函数体内被使用， 同时也在方法被调用时作为标签被使用。该方法的第一个参数名仅仅在函数体内被使用，就像函数的参数一样，举例来讲：

```
func f(x: Int, y: String) -> String {
    return y + String(x)
```

```
}
f(7, "hello")  // x and y have no name

class C {
    func f(x: Int, y: String) -> String {
       return y + String(x)
    }
}
let c = C()
c.f(7, y: "hello")  // x没有名称，y有名称
```

你可以按如下的形式，重写参数名被使用的过程：

```
external parameter name local parameter name:
parameter type
#parameter name: parameter type
_ local parameter name: parameter type
```

在本地参数前命名的第二名称(second name)使得参数有一个扩展名。且不同于本地的参数名。 扩展参数名在函数被调用时必须被使用。对应的参数在方法或函数被调用时必须有扩展名 。

在参数名前所写的哈希符号(#)代表着这个参数名可以同时作为外部或本体参数名来使用。等同于书写两次本地参数名。 在函数或方法调用时，与其对应的语句必须包含这个名字。

本地参数名前的强调字符(_)使参数在函数被调用时没有名称。在函数或方法调用时，与其对应的语句必须没有名字。

特殊类型的参数

参数可以被忽略，值可以是变化的，并且提供一个初始值，这种方法有着如下的形式：

```
_ : <#parameter type#.
parameter name: parameter type...
parameter name: parameter type = default argument
value
```

以强调符(_)命名的参数明确的在函数体内不能被访问。

一个以基础类型名的参数，如果紧跟着三个点(...)，被理解为是可变参数。一个函数至多可以拥有一个可变参数， 且必须是最后一个参数。可变参数被作为该基本类型名的数组来看待。举例来讲，可变参数int...被看做是int[]。 查看可变参数的使用例子，详见可变参数(variadic parameters)一节。

在参数的类型后面有一个以等号(=)连接的表达式，这样的参数被看做有着给定表达式的初试值。如果参数在函数 调用时被省略了，就会使用初始值。如果参数没有胜率，那么它在函数调用是必须有自己的名字.举例来讲， f()和f(x:7)都是只有一个变量x的函数的有效调用，但是f(7)是非法的，因为它提供了一个值而不是名称。

特殊方法

以self修饰的枚举或结构体方法必须以mutating关键字作为函数声明头。

子类重写的方法必须以override关键字作为函数声明头。不用override关键字重写的方法，使用了override关键字 却并没有重写父类方法都会报错。

和类型相关而不是和类型实例相关的方法必须在static声明的结构以或枚举内，亦或是以class关键字定义的类内。

柯里化函数和方法

柯里化函数或方法有着如下的形式:

```
func function name(parameters)(parameters) -> return type {
    statements
}
```

以这种形式定义的函数的返回值是另一个函数。举例来说，下面的两个声明时等价的:

```
func addTwoNumbers(a: Int)(b: Int) -> Int {
    return a + b
}
func addTwoNumbers(a: Int) -> (Int -> Int) {
```

```
    func addTheSecondNumber(b: Int) -> Int {
        return a + b
    }
    return addTheSecondNumber
}

addTwoNumbers(4)(5) // Returns 9
```

多级柯里化应用如下

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → function-head function-name generic-parameter-clause optfunction-signature function-body function-head → attributes opt declaration-specifiers opt func function-name → identifier operator function-signature → parameter-clauses function-result opt function-result → ->attributes opt type function-body → code-block parameter-clauses → parameter-clause parameter-clauses opt parameter-clause → () (parameter-list...opt) parameter-list → parameter parameter,parameter-list parameter → inout opt let opt#optparameter-name local-parameter-name opt type-annotation default-argument-clause opt parameter → inoutoptvar#optparameter-namelocal-parameter-name opt type-annotationdefault-argument-clause opt parameter → attributes opt type parameter-name → identifier *local-parameter-name → identifier* default-argument-clause → =expression：

枚举声明

在你的程序里使用枚举声明来引入一个枚举类型。

枚举声明有两种基本的形式，使用关键字enum来声明。枚举声明体使用从零开始的变量——叫做枚举事件，和任意数量的 声明，包括计算型属性，实例方法，静态方法，构造器，类型别名，甚至其他枚举，结构体，和类。枚举声明不能 包含析构器或者协议声明。

不像类或者结构体。枚举类型并不提供隐式的初始构造器，所有构造器必须显式的声明。构造器可以委托枚举中的其他 构造器，但是构造过程仅当构造器将一个枚举时间完成后才全部完成。

和结构体类似但是和类不同，枚举是值类型：枚举实例在赋予变量或常量时，或者被函数调用时被复制。 更多关于值类型的信息，参见结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以扩展枚举类型，正如在扩展名声明(Extension Declaration)中讨论的一样。

任意事件类型的枚举

如下的形式声明了一个包含任意类型枚举时间的枚举变量

```
enum enumeration name {
    case enumeration case 1
    case enumeration case 2(associated value types)
}
```

这种形式的枚举声明在其他语言中有时被叫做可识别联合(discrinminated)。

这种形式中，每一个事件块由关键字case开始，后面紧接着一个或多个以逗号分隔的枚举事件。每一个事件名必须是 独一无二的。每一个事件也可以指定它所存储的指定类型的值，这些类型在关联值类型的元祖里被指定，立即书写在事件 名后。获得更多关于关联值类型的信息和例子，请查看关联值(associated values)一节。

使用原始事件值的枚举

以下的形式声明了一个包含相同基础类型的枚举事件的枚举:

```
enum enumeration name: raw value type {
    case enumeration case 1 = raw value 1
    case enumeration case 2 = raw value 2
}
```

在这种形式中，每一个事件块由case关键字开始，后面紧接着一个或多个以逗号分隔的枚举事件。和第一种形式的枚举 事件不同，这种形式的枚举事件包含一个同类型的基础值，叫做原始值(raw value)。这些值的类型在原始值类型(raw value type) 中被指定，必须是字面上的整数，浮点数，

字符或者字符串。

每一个事件必须有唯一的名字，必须有一个唯一的初始值。如果初始值类型被指定为int，则不必为事件显式的指定值， 它们会隐式的被标为值0,1,2等。每一个没有被赋值的Int类型时间会隐式的赋予一个初始值，它们是自动递增的。

```
num ExampleEnum: Int {
    case A, B, C = 5, D
}
```

在上面的例子中，ExampleEnum.A的值是0，ExampleEnum.B的值是。因为ExampleEnum.C的值被显式的设定为5，因此 ExampleEnum.D的值会自动增长为6.

枚举事件的初始值可以调用方法roRaw获得，如ExampleEnum.B.toRaw()。你也可以通过调用fromRaw方法来使用初始值找到 其对应的事件，并返回一个可选的事件。查看更多信息和获取初始值类型事件的信息，参阅初始值(raw values)。

获得枚举事件

使用点(.)来引用枚举类型的事件，如 EnumerationType.EnumerationCase。当枚举类型可以上下文推断出时，你可以 省略它(.仍然需要)，参照枚举语法(Enumeration Syntax)和显式成员表达(Implicit Member Expression).

使用switch语句来检验枚举事件的值，正如使用switch语句匹配枚举值（Matching Enumeration Values with a Switch Statement)一节描述的那样。

枚举类型是模式匹配(pattern-matched)的，和其相反的是switch语句case块中枚举事件匹配，在枚举事件类型(Enumeration Case Pattern)中有描述。

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → attributesoptunion-style-enum attributesoptraw-value-style-enum union-style-enum → enum-namegeneric-parameter-clauseopt{ union-style-enum-membersopt} union-style-enum-members → union-style-enum-memberunion-style-enum-membersopt union-style-enum-member → declaration union-style-enum-case-clause union-style-enum-case-clause →

attributesoptcaseunion-style-enum-case-list union-style-enum-case-list → union-style-enum-case union-style-enum-case,union-style-enum-case-list union-style-enum-case → enum-case-nametuple-typeopt enum-name → identifier enum-case-name → identifier raw-value-style-enum → enum-name generic-parameter-clauseopt:type-identifier{raw-value-style-enum-members opt} raw-value-style-enum-members → raw-value-style-enum-memberraw-value-style-enum-membersopt raw-value-style-enum-member → declaration raw-value-style-enum-case-clause raw-value-style-enum-case-clause → attributesoptcaseraw-value-style-enum-case-list raw-value-style-enum-case-list → raw-value-style-enum-case raw-value-style-enum-case,raw-value-style-enum-case-list raw-value-style-enum-case → enum-case-nameraw-value-assignmentopt raw-value-assignment → =literal

结构体声明

使用结构体声明可以在你的程序里引入一个结构体类型。结构体声明使用struct关键字，遵循如下的形式：

```
struct structure name: adopted protocols {
    declarations
}
```

结构体内包含零或多个声明。这些声明可以包括存储型和计算型属性，静态属性，实例方法，静态方法，构造器， 类型别名，甚至其他结构体，类，和枚举声明。结构体声明不能包含析构器或者协议声明。详细讨论和包含多种结构体 声明的实例，参见类和结构体一节。

结构体可以包含任意数量的协议，但是不能继承自类，枚举或者其他结构体。

有三种方法可以创建一个声明过的结构体实例：

-调用结构体内声明的构造器，参照构造器(initializers)一节。

—如果没有声明构造器，调用结构体的逐个构造器，详情参见 Memberwise Initializers for Structure Types.

—如果没有声明析构器，结构体的所有属性都有初始值，调用结构体的默

认构造器，详情参见默认构造器(Default Initializers).

结构体的构造过程参见初始化(initiaization)一节。

结构体实例属性可以用点(.)来获得，详情参见获得属性(Accessing Properties)一节。

结构体是值类型；结构体的实例在被赋予变量或常量，被函数调用时被复制。获得关于值类型更多信息，参见 结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以使用扩展声明来扩展结构体类型的行为，参见扩展声明(Extension Declaration).

GRAMMAR OF A STRUCTURE DECLARATION

struct-declaration → attributesoptstructstruct-namegeneric-parameter-clauseopt type-inheritance-clauseoptstruct-body struct-name → identifier struct-body → { declarationsopt}

类声明

你可以在你的程序中使用类声明来引入一个类。类声明使用关键字class，遵循如下的形式：

```
class class name: superclass, adopted protocols {
    declarations
}
```

一个类内包含零或多个声明。这些声明可以包括存储型和计算型属性，实例方法，类方法，构造器，单独的析构器方法， 类型别名，甚至其他结构体，类，和枚举声明。类声明不能包含协议声明。详细讨论和包含多种类声明的实例，参见类和 结构体一节。

一个类只能继承一个父类，超类，但是可以包含任意数量的协议。这些超类第一次在type-inheritance-clause出现，遵循任意协议。

正如在初始化声明(Initializer Declaration)谈及的那样，类可以有指定和方

便的构造器。当你声明任一中构造器时， 你可以使用requierd变量来标记构造器，要求任意子类来重写它。指定类的构造器必须初始化类所有的已声明的属性， 它必须在子类构造器调用前被执行。

类可以重写属性，方法和它的超类的构造器。重写的方法和属性必须以override标注。

虽然超类的属性和方法声明可以被当前类继承，但是超类声明的指定构造器却不能。这意味着，如果当前类重写了超类 的所有指定构造器，它就继承了超类的方便构造器。Swift的类并不是继承自一个全局基础类。

有两种方法来创建已声明的类的实例:

-调用类的一个构造器，参见构造器(initializers)。

-如果没有声明构造器，而且类的所有属性都被赋予了初始值，调用类的默认构造器，参见默认构造器(default initializers).

类实例属性可以用点(.)来获得，详情参见获得属性(Accessing Properties)一节。

类是引用类型；当被赋予常量或变量，函数调用时，类的实例是被引用，而不是复制。获得更多关于引用类型的信息， 结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以使用扩展声明来扩展类的行为，参见扩展声明(Extension Declaration).

GRAMMAR OF A CLASS DECLARATION

class-declaration → attributesoptclassclass-namegeneric-parameter-clauseopt type-inheritance-clauseoptclass-body class-name → identifier class-body → { declarationsopt}

协议声明(translated by 小一)

一个协议声明为你的程序引入一个命名了的协议类型。协议声明使用

`protocol` 关键词来进行声明并有下面这样的形式：

```
protocol protocol name: inherited protocols {
    protocol member declarations
}
```

协议的主体包含零或多个协议成员声明，这些成员描述了任何采用该协议必须满足的一致性要求。特别的，一个协议可以声明必须实现某些属性、方法、初始化程序及附属脚本的一致性类型。协议也可以声明专用种类的类型别名，叫做关联类型，它可以指定协议的不同声明之间的关系。协议成员声明会在下面的详情里进行讨论。

协议类型可以从很多其它协议那继承。当一个协议类型从其它协议那继承的时候，来自其它协议的所有要求就集合了，而且从当前协议继承的任何类型必须符合所有的这些要求。对于如何使用协议继承的例子，查看协议继承

注意：

你也可以使用协议合成类型集合多个协议的一致性要求，详情参见协议合成类型和协议合成

你可以通过采用在类型的扩展声明中的协议来为之前声明的类型添加协议一致性。在扩展中你必须实现所有采用协议的要求。如果该类型已经实现了所有的要求，你可以让这个扩展声明的主题留空。

默认地，符合某一个协议的类型必须实现所有声明在协议中的属性、方法和附属脚本。也就是说，你可以用`optional`属性标注这些协议成员声明以指定它们的一致性类型实现是可选的。`optional`属性仅仅可以用于使用`objc`属性标记过的协议。这样的结果就是仅仅类类型可以采用并符合包含可选成员要求的协议。更多关于如何使用`optional`属性的信息及如何访问可选协议成员的指导——比如当你不能肯定是否一致性的类型实现了它们——参见可选协议要求

为了限制协议的采用仅仅针对类类型，需要使用`class_protocol`属性标记整个协议声明。任意继承自标记有`class_protocol`属性协议的协议都可以智能地仅能被类类型采用。

注意：

如果协议已经用`object`属性标记了，`class_protocol`属性就隐性地应用于该协议；没有必要再明确地使用`class_protocol`属性来标记该协议了。

协议是命名的类型，因此它们可以以另一个命名类型出现在你代码的所有地方，就像协议类型里讨论的那样。然而你不能构造一个协议的实例，因为协议实际上不提供它们指定的要求的实现。

你可以使用协议来声明一个类的代理的方法或者应该实现的结构，就像委托(代理)模式描述的那样。

协议声明的语法 protocol-declaration → attributesoptprotocolprotocol-name type-inheritance-clauseoptprotocol-body protocol-name → identifier protocol-body → {protocol-member-declarationsopt} protocol-member-declaration → protocol-property-declaration protocol-member-declaration → protocol-method-declaration protocol-member-declaration → protocol-initializer-declaration protocol-member-declaration → protocol-subscript-declaration protocol-member-declaration → protocol-associated-type-declaration protocol-member-declarations → protocol-member-declarationprotocol-member-declarationsopt

协议属性声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议属性声明来实现一个属性。协议属性声明有一种特殊的类型声明形式：

```
var property name: type { get set }
```

同其它协议成员声明一样，这些属性声明仅仅针对符合该协议的类型声明了`getter`和`setter`要求。结果就是你不需要在协议里它被声明的地方实现`getter`和`setter`。

`getter`和`setter`要求可以通过一致性类型以各种方式满足。如果属性声明包含`get`和`set`关键词，一致性类型就可以用可读写（实现了`getter`和`setter`）的存储型变量属性或计算型属性，但是属性不能以常量属性或只读计算型属性实现。如果属性声明仅仅包含`get`关键词的话，它可以作为任意类型的属性被实现。比如说实现了协议的属性要求的一致性类型，

参见属性要求

更多参见变量声明

协议属性声明语法 protocol-property-declaration → variable-declaration-head variable-nametype-annotationgetter-setter-keyword-block

协议方法声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议方法声明来实现一个方法. 协议方法声明和函数方法声明有着相同的形式，包含如下两条规则：他们不包括函数体，你不能在类的声明内为他们的 参数提供初始值.举例来说，符合的类型执行协议必需的方法。参见必需方法一节。

使用关键字class可以在协议声明中声明一个类或必需的静态方法。执行这些方法的类也用关键字class声明。 相反的，执行这些方法的结构体必须以关键字static声明。如果你想使用扩展方法，在扩展类时使用class关键字，在扩展结构体时使用static关键字。

更多请参阅函数声明。

GRAMMAR OF A PROTOCOL METHOD DECLARATION

protocol-method-declaration → function-headfunction-namegeneric-parameter-clauseoptfunction-signature

协议构造器声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议构造器声明来实现一个构造器。协议构造器声明 除了不包含构造器体外，和构造器声明有着相同的形式，

更多请参阅构造器声明。

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → initializer-headgeneric-parameter-clauseopt parameter-clause

协议附属脚本声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议附属脚本声明来实现一个附属脚本。协议属性声明 对附属脚本声明有一个特殊的形式:

subscript (parameters) -> return type { get set }
附属脚本声明只为和协议一致的类型声明了必需的最小数量的的getter和setter。如果附属脚本申明包含get和set关键字， 一致的类型也必须有一个getter和setter语句。如果附属脚本声明值包含get关键字，一致的类型必须至少包含一个 getter语句，可以选择是否包含setter语句。

更多参阅附属脚本声明。

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

protocol-subscript-declaration → subscript-headsubscript-resultgetter-setter-keyword-block

协议相关类型声明

协议声明相关类型使用关键字typealias。相关类型为作为协议声明的一部分的类型提供了一个别名。相关类型和参数 语句中的类型参数很相似，但是它们在声明的协议中包含self关键字。在这些语句中，self指代和协议一致的可能的类型。 获得更多信息和例子，查看相关类型或类型别名声明。

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → typealias-headtype-inheritance-clause opttypealias-assignmentopt

构造器声明

构造器声明会为程序内的类，结构体或枚举引入构造器。构造器使用关键字Init来声明，遵循两条基本形式。

结构体，枚举，类可以有任意数量的构造器，但是类的构造器的规则和行

为是不一样的。不像结构体和枚举那样，类 有两种结构体，designed initializers 和convenience initializers，参见构造器一节。

如下的形式声明了结构体，枚举和类的指定构造器：

```
init(parameters) {
     statements
}
```

类的指定构造器将类的所有属性直接初始化。如果类有超类，它不能调用该类的其他构造器,它只能调用超类的一个 指定构造器。如果该类从它的超类处继承了任何属性，这些属性在当前类内被赋值或修饰时，必须带哦用一个超类的 指定构造器。

指定构造器可以在类声明的上下文中声明，因此它不能用扩展声明的方法加入一个类中。

结构体和枚举的构造器可以带哦用其他的已声明的构造器，来委托其中一个火全部进行初始化过程。

以关键字convenience来声明一个类的便利构造器：

```
convenience init(parameters) {
   statements
}
```

便利构造器可以将初始化过程委托给另一个便利构造器或类的一个指定构造器。这意味着，类的初始化过程必须 以一个将所有类属性完全初始化的指定构造器的调用作为结束。便利构造器不能调用超类的构造器。

你可以使用requierd关键字，将便利构造器和指定构造器标记为每个子类的构造器都必须拥有的。因为指定构造器 不被子类继承，他们必须被立即执行。当子类直接执行所有超类的指定构造器(或使用便利构造器重写指定构造器)时， 必需的便利构造器可以被隐式的执行，亦可以被继承。不像方法，附属脚本那样，你不需要为这些重写的构造器标注 overrride关键字。

查看更多关于不同声明方法的构造器的例子，参阅构造过程一节。

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → initializer-headgeneric-parameter-clauseoptparameter-clauseinitializer-body initializer-head → attributesoptconvenienceoptinit initializer-body → code-block

析构声明

析构声明为类声明了一个析构器。析构器没有参数，遵循如下的格式：

```
deinit {
   statements
}
```

当类没有任何语句时将要被释放时，析构器会自动的被调用。析构器在类的声明体内只能被声明一次——但是不能在 类的扩展声明内，每个类最多只能有一个。

子类继承了它的超类的析构器，在子类将要被释放时隐式的调用。子类在所有析构器被执行完毕前不会被释放。

析构器不会被直接调用。

查看例子和如何在类的声明中使用析构器，参见析构过程一节 。

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → attributesoptdeinitcode-block

扩展声明

扩展声明用于扩展一个现存的类，结构体，枚举的行为。扩展声明以关键字extension开始，遵循如下的规则：

```
extension type: adopted protocols {
   declarations
}
```

一个扩展声明体包括零个或多个声明。这些声明可以包括计算型属性，计算型静态属性，实例方法，静态和类方法，构造器， 附属脚本声明，甚至其他结构体，类，和枚举声明。扩展声明不能包含析构器，协议声明，存储型属性，属性监测器或其他 的扩展属性。详细讨论和查看包含多种扩展声明的实例，参见扩展一节。

扩展声明可以向现存的类，结构体，枚举内添加一致的协议。扩展声明不能向一个类中添加继承的类，因此 type-inheritance-clause是一个只包含协议列表的扩展声明。

属性，方法，现存类型的构造器不能被它们类型的扩展所重写。

扩展声明可以包含构造器声明，这意味着，如果你扩展的类型在其他模块中定义，构造器声明必须委托另一个在 那个模块里声明的构造器来恰当的初始化。

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → extensiontype-identifiertype-inheritance-clauseopt extension-body extension-body → {declarationsopt}

附属脚本声明(translated by 林)

附属脚本用于向特定类型添加附属脚本支持，通常为访问集合，列表和序列的元素时提供语法便利。附属脚本声明使用关键字`subscript`，声明形式如下：

subscript (`parameter`) -> (return type){ get{ `statements` } set(`setter name`){ `statements` } } 附属脚本声明只能在类，结构体，枚举，扩展和协议声明的上下文进行声明。
变量*(parameters)*指定一个或多个用于在相关类型的附属脚本中访问元素的索引（例如，表达式`object[i]`中的`i`）。尽管用于元素访问的索引可以是任意类型的，但是每个变量必须包含一个用于指定每种索引类型的类型标注。返回类型*(return type)*指定被访问的元素的类型。

和计算性属性一样，附属脚本声明支持对访问元素的读写操作。getter用

于读取值，setter用于写入值。setter子句是可选的，当仅需要一个getter子句时，可以将二者都忽略且直接返回请求的值即可。也就是说，如果使用了setter子句，就必须使用getter子句。

setter的名字和封闭的括号是可选的。如果使用了setter名称，它会被当做传给setter的变量的名称。如果不使用setter名称，那么传给setter的变量的名称默认是`value`。setter名称的类型必须与返回类型*(return type)*的类型相同。

可以在附属脚本声明的类型中，可以重载附属脚本，只要变量*(parameters)*或返回类型*(return type)*与先前的不同即可。此时，必须使用`override`关键字声明那个被覆盖的附属脚本。(注：好乱啊！到底是重载还是覆盖？！)

同样可以在协议声明的上下文中声明附属脚本，Protocol Subscript Declaration中有所描述。

更多关于附属脚本和附属脚本声明的例子，请参考Subscripts。

GRAMMAR OF A SUBSCRIPT DECLARATION

subscript-declaration → subscript-headsubscript-resultcode-block subscript-declaration → subscript-headsubscript-resultgetter-setter-block subscript-declaration → subscript-headsubscript-resultgetter-setter-keyword-block subscript-head → attributesoptsubscriptparameter-clause subscript-result → -> attributesopttype

运算符声明(translated by 林)

运算符声明会向程序中引入中缀、前缀或后缀运算符，它使用上下文关键字`operator`声明。 可以声明三种不同的缀性：中缀、前缀和后缀。操作符的缀性描述了操作符与它的操作数的相对位置。 运算符声明有三种基本形式，每种缀性各一种。运算符的缀性通过在`operator`和运算符之间添加上下文关键字`infix`，`prefix`或`postfix`来指定。每种形式中，运算符的名字只能包含Operators中定义的运算符字符。

下面的这种形式声明了一个新的中缀运算符：

operator infix `operator name`{ precedence `precedence level` associativity `associativity` }
中缀运算符是二元运算符，它可以被置于两个操作数之间，比如表达式`1 + 2`中的加法运算符(`+`)。

中缀运算符可以可选地指定优先级，结合性，或两者同时指定。

运算符的优先级可以指定在没有括号包围的情况下，运算符与它的操作数如何紧密绑定的。可以使用上下文关键字`precedence`并优先级*(precedence level)*一起来指定一个运算符的优先级。优先级可以是0到255之间的任何一个数字(十进制整数)；与十进制整数字面量不同的是，它不可以包含任何下划线字符。尽管优先级是一个特定的数字，但它仅用作与另一个运算符比较(大小)。也就是说，一个操作数可以同时被两个运算符使用时，例如`2 + 3 * 5`，优先级更高的运算符将优先与操作数绑定。

运算符的结合性可以指定在没有括号包围的情况下，优先级相同的运算符以何种顺序被分组的。可以使用上下文关键字`associativity`并结合性*(associativity)*一起来指定一个运算符的结合性，其中结合性可以说是上下文关键字`left`，`right`或`none`中的任何一个。左结合运算符以从左到右的形式分组。例如，减法运算符(`-`)具有左结合性，因此`4 - 5 - 6`被以`(4 - 5) - 6`的形式分组，其结果为`-7`。 右结合运算符以从右到左的形式分组，对于设置为`none`的非结合运算符，它们不以任何形式分组。具有相同优先级的非结合运算符，不可以互相邻接。例如，表达式`1 < 2 < 3`非法的。

声明时不指定任何优先级或结合性的中缀运算符，它们的优先级会被初始化为100，结合性被初始化为`none`。

下面的这种形式声明了一个新的前缀运算符：

operator prefix `operator name`{}
紧跟在操作数前边的前缀运算符*(prefix operator)*是一元运算符，例如表达式`++i`中的前缀递增运算符(`++`)。

前缀运算符的声明中不指定优先级。前缀运算符是非结合的。

下面的这种形式声明了一个新的后缀运算符：

operator postfix `operator name`{}
紧跟在操作数后边的后缀运算符*(postfix operator)*是一元运算符，例如表达式`i++`中的前缀递增运算符(`++`)。

和前缀运算符一样，后缀运算符的声明中不指定优先级。后缀运算符是非结合的。

声明了一个新的运算符以后，需要声明一个跟这个运算符同名的函数来实现这个运算符。如何实现一个新的运算符，请参考Custom Operators。

GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → prefix-operator-declaration postfix-operator-declaration >infix-operator-declaration prefix-operator-declaration → operator prefix operator{} postfix-operator-declaration → operator postfix operator{} infix-operator-declaration → operatorinfixoperator{infix-operator-attributesopt} infix-operator-attributes → precedence-clauseoptassociativity-clauseopt precedence-clause → precedenceprecedence-level precedence-level → Digit 0 through 255 associativity-clause → associativityassociativity associativity → left right none