类型转换

cnswift.org/type-casting

*类型转换*可以判断实例的类型，也可以将该实例在其所在的类层次中视为其父类或子类的实例。

Swift 中类型转换的实现为 is 和 as 操作符。这两个操作符使用了一种简单传神的方式来检查一个值的类型或将某个值转换为另一种类型。

如同协议实现的检查（此处应有链接）中描述的那样，你还可以使用类型转换来检查类型是否遵循某个协议。

为类型转换定义类层次

你可以在类及其子类层次中使用类型转换来判断特定类实例的类型并且在同一类层次中将该实例类型转换为另一个类。下面的三段代码定义了一个类层次以及一个包含了这些类实例的数组，作为类型转换的例子。

第一个代码片段定义了一个叫做 MediaItem 的新基类。这个类为出现在数字媒体库中的所有成员提供了基本的功能。它声明了一个 String 类型的 name 和一个叫做 init 的 name 初始化器。（这里假设所有的媒体项目，包括所有电影和音乐，都有一个名字。）

```
1  classMediaItem{
2  varname:String
3  init(name:String){
4  self.name=name
5  }
6  }
```

下一个片段定义了两个 MediaItem 的子类。第一个子类， Movie ，封装了额外的电影的信息。他在 MediaItem 的基础上添加了名为 director 的属性及其初始化器。第二个子类， Song ，增加了名为 artist 的属性及其初始化器。

```
1    classMovie: MediaItem{
2    vardirector:String
3    init(name:String,director:String){
4    self.director=director
5    super.init(name:name)
6    }
7    }
8    classSong: MediaItem{
9    varartist:String
10   init(name:String,artist:String){
11   self.artist=artist
12   super.init(name:name)
13   }
14   }
15
```

最后一个代码段创建了名为 library 的常量数组，它包含了两个 Movie 实例和三个 Song 实例。 library 数组的类型是在初始化时根据常量字面量推断出来的。 Swift 的类型检查器能够推断 Movie 和 Song 有一个共同的父类 MediaItem ，因此 library 的类型推断为

[MediaItem] ：

```
letlibrary=[
Movie(name:"Casablanca",director:"Michael Curtiz"),
Song(name:"Blue Suede Shoes",artist:"Elvis Presley"),
Movie(name:"Citizen Kane",director:"Orson Welles"),
Song(name:"The One And Only",artist:"Chesney Hawkes"),
Song(name:"Never Gonna Give You Up",artist:"Rick Astley")
]
// "library" 的类型被推断为[MediaItem]
```

事实上 library 储存的项目在后台依旧是 Movie 和 Song 实例。总之，如果你遍历这个数组的内容，你取出的项目将会是 MediaItem 类型而非 Movie 或 Song 类型。为了使用他们原生的类型，你需要*检查*他们的类型或将他们*向下转换*为不同的类型，如下所述。

类型检查

使用*类型检查操作符* （ is ）来检查一个实例是否属于一个特定的子类。如果实例是该子类类型，类型检查操作符返回 true ，否则返回 false 。

下面的例子定义了两个变量， movieCount 和 songCount ，用来计算数组 library 中 Movie 和 Song 实例的个数：

```
varmovieCount=0
varsongCount=0
foriteminlibrary{
ifitemisMovie{
movieCount+=1
}elseifitemisSong{
songCount+=1
}
}
print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"


```

这个例子遍历了 library 数组中的每个元素。每一轮中， for-in 的循环都将 item 常量设置为数组中的下一个 MediaItem 。

如果当前 MediaItem 是 Movie 类型的实例， item isMovie 返回 true ，反之返回 false 。同样的， item isSong 检查了该对象是否为 Song 类型的实例。在 for-in 循环的最后， movieCount 和 songCount 的值就是数组中对应类型实例的数量。

向下类型转换

某个类类型的常量或变量可能实际上在后台引用自一个子类的实例。当你遇到这种情况时你可以尝试使用*类型转换操作符*（ as? 或 as! ）将它*向下类型转换*至其子类类型。

由于向下类型转换能失败，类型转换操作符就有了两个不同形式。条件形式， as? ，返回了一个你将要向下类型转换的值的可选项。强制形式， as! ，则将向下类型转换和强制展开结合为一个步骤。

如果你不确定你向下转换类型是否能够成功，请使用条件形式的类型转换操作符 （ as? ）。使用条件形式的类型转换操作符总是返回一个可选项，如果向下转换失败，可选值为 nil 。

这允许你检查向下类型转换是否成功。

当你确信向下转换类型会成功时，使用强制形式的类型转换操作符（ as! ）。当你向下转换至一个错误的类型时，强制形式的类型转换操作符会触发一个运行错误。

下面的例子遍历了 library 中的每个 MediaItem ，并打印出相应的描述信息。要这样做的话，每个项目均需要被当做 Movie 或 Song 来访问，而不仅仅是 MediaItem 。为了在描述信息中访问 Movie 或 Song 的 director 和 artist 属性，这样做是必要的。

在这个例子中，数组中每一个项目的类型可能是 Movie 也可能是 Song 。你不知道遍历时项目的确切类型是什么，所以这时使用条件形式的类型转换符（ as? ）来检查遍历中每次向下类型转换：

```
foriteminlibrary{
ifletmovie=itemas?Movie{
print("Movie: '\(movie.name)', dir. \(movie.director)")
}elseifletsong=itemas?Song{
print("Song: '\(song.name)', by \(song.artist)")
}
}
// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley

```

例子开头尝试将当前 item 当做 Movie 向下类型转换。由于 item 是一个 MediaItem 的实例，它*有可能*是 Movie 类型；同样的，也有可能是 Song 或者仅仅是 MediaItem 基类。介于这种不确定性，类型转换符 as? 在向下类型转换到子类时返回了一个可选项。 item as?Movie 的结果是 Movie? 类型，也就是“可选 Movie 类型”。

当数组中的 Song 实例使用向下转换至 Movie 类型时会失败。为了处理这种情况，上面的例子使用了可选绑定来检查可选 Movie 类型是否包含了一个值（或者说检查向下类型转换是否成功）。这个可选绑定写作“ ifletmovie=item as?Movie ”，它可以被读作：

尝试以 Movie 类型访问 item 。如果成功，设置一个新的临时常量 movie 储存返回的可选 Movie 类型 。

如果向下类型转换成功， movie 的属性将用于输出 Movie 实例的描述信息，包括 director 的名字。同理，无论是否在数组中找到 Song ，均可以检查 Song 实例然后输出合适的描述（包括 artist 的名字）。

> 注意
>
> 类型转换实际上不会改变实例及修改其值。实例不会改变；它只是将它当做要转换的类型来访问。

Any 和 AnyObject 的类型转换

Swift 为不确定的类型提供了两种特殊的类型别名：
- AnyObject 可以表示任何类类型的实例。
- Any 可以表示任何类型，包括函数类型。

只有当你确切需要使用它们的功能和行为时再使用 Any 和 AnyObject 。在写代码时使用更加明确的类型表达总要好一些。

这里有一个使用 Any 类型来对不同类型进行操作的例子，包含了函数类型以及非类类型。这个例子定义了一个名为 things 的数组，它用于储存 Any 类型的值：

```
varthings=[Any]()
things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0,5.0))
things.append(Movie(name:"Ghostbusters",director:"Ivan Reitman"))
things.append({(name:String)->Stringin"Hello, \(name)"})

```

这个 things 数组包含了两个 Int 值、两个 Double 值、一个 String 值、一个 (Double,Double) 的元组、 Movie 实例“Ghostbusters”、以及一个接收 String 值并返回 String 值的闭包表达式。

你可以在 switch 结构的 case 中使用 is 和 as 操作符找出已知 Any 或 AnyObject 类型的常量或变量的具体类型。下面的例子使用 switch 语句遍历了 things 数组并查询每一项的类型。其中几个 switch 的 case 将确定的值和确定类型的常量绑定在一起，使其值可以被输出：

```
forthinginthings{
switchthing{
case0asInt:
print("zero as an Int")
case0asDouble:
print("zero as a Double")
caseletsomeInt asInt:
print("an integer value of \(someInt)")
caseletsomeDouble asDoublewheresomeDouble>0:
print("a positive double value of \(someDouble)")
caseisDouble:
print("some other double value that I don't want to print")
caseletsomeString asString:
print("a string value of \"\(someString)\"")
caselet(x,y)as(Double,Double):
print("an (x, y) point at \(x), \(y)")
caseletmovie asMovie:
print("a movie called \(movie.name), dir. \(movie.director)")
caseletstringConverter as(String)->String:
print(stringConverter("Michael"))
default:
print("something else")
}
}
// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael

```

> 注意
>
> Any类型表示了任意类型的值，包括可选类型。如果你给显式声明的Any类型使用可选项，Swift 就会发出警告。如果你真心需要在Any值中使用可选项，如下所示，你可以使用as运算符来显式地转换可选项为Any。

```
letoptionalNumber:Int?=3
things.append(optionalNumber)// Warning
things.append(optionalNumber asAny)// No warning
```