类型铸造

*类型转换*是一种检查实例类型的方法，或者将该实例视为其自己类层次结构中其他位置的不同超类或子类。

Swift中的类型转换是通过is和as运算符来实现的。这两个运算符提供了一种简单而富有表现力的方式来检查值的类型或将值转换为其他类型。

您还可以使用类型转换来检查类型是否符合协议，如检查协议一致性中所述。

定义类型铸造的类层次结构

您可以使用具有类和子类层次结构的类型转换来检查特定类实例的类型，并将该实例转换为同一层次结构中的另一个类。下面的三个代码片断定义了一个类的层次结构和一个包含这些类的实例的数组，以用于类型转换的示例。

第一个片段定义了一个新的基类MediaItem。该课程为出现在数字媒体库中的任何类型的项目提供基本功能。具体来说，它声明了一个name类型的属性String和一个init name初始化器。（假设所有媒体项目，包括所有电影和歌曲，都会有一个名称。）

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

下一个片段定义了两个子类MediaItem。第一个小类Movie囊括了关于电影或电影的附加信息。它director在基MediaItem类之上添加一个属性，并带有相应的初始化程序。第二个子类在基类的顶部Song添加一个artist属性和初始化程序：

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

最后的片段创建一个名为的常量数组library，其中包含两个Movie实例和三个Song实例。library数组的类型通过用数组文本的内容初始化来推断。Swift的类型检查器能够推断出Movie并Song具有一个公共的超类MediaItem，因此它推断出[MediaItem]该library数组的一种类型：

```
let library = [
    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")
]
// the type of "library" is inferred to be [MediaItem]
```

在本页

存储在项目`library`仍然`Movie`和`Song`幕后的情况。但是，如果您迭代该数组的内容，则收到的项目将被键入为`MediaItem`，而不是`Movie`或`Song`。为了和他们一起工作，你需要*检查*他们的类型，或者将它们*向下转换*为不同的类型，如下所述。

检查类型

使用*类型检查运算符*（`is`）来检查实例是否属于某个子类类型。类型检查运算符返回`true`实例是否属于该子类类型，`false`如果不是。

下面的例子定义两个变量，`movieCount`并且`songCount`，其计数的数量`Movie`和`Song`实例中`library`阵列：

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

本示例遍历`library`数组中的所有项目。在每次传递时，`for- in`loop将`item`常量设置`MediaItem`为数组中的下一个。

`item is Movie`true如果当前`MediaItem`是`Movie`实例并且`false`不是，则返回。同样，`item is Song`检查项目是否是`Song`实例。在`for- in`循环结束时`movieCount`，`songCount`包含`MediaItem`每个类型找到多少个实例的计数值。

溯造型

某种类型的常量或变量可能实际上指的是幕后的子类实例。如果您认为是这种情况，您可以尝试使用*类型转换运算符*（`as?`或`as!`）*向下转换*为子类型。

因为向下转换可能失败，所以类型转换运算符有两种不同的形式。条件表单`as?`将返回您要向下转换的类型的可选值。强制形式`as!`试图将压倒性结果作为一个单一的复合动作进行强制解开。

`as?`如果不确定downcast是否成功， 请使用类型转换运算符（）的条件形式。这种形式的操作符将始终返回一个可选值，`nil`如果downcast不可能，该值将为。这使您能够检查成功的downcast。

`as!`只有当你确定downcast将总是成功时，才 使用类型转换运算符（）的强制形式。如果尝试向下转换为不正确的类类型，此操作符形式将触发运行时错误。

下面在每个迭代的例子`MediaItem`中`library`，并打印每个项目的相应说明。要做到这一点，它需要访问每个项目作为一个真正的`Movie`或`Song`，而不是仅仅作为一个`MediaItem`。这是必要的，以便它能够访问或用于描述中的`director`或`artist`属性。 `MovieSong`

在这个例子中，数组中的每个项目可能是a `Movie`，或者它可能是a `Song`。您事先不知道每个项目使用哪个实际类，因此使用类型cast操作符（`as?`）的条件形式来检查每次遍历循环时的downcast 是合适的：

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? 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
```

这个例子首先尝试将当前项item向下转换为Movie。因为item是一个MediaItem类型的实例，这是可能的，它可能是一个Movie; 同样，它也有可能是一个Song，甚至只是一个基地MediaItem。由于这种不确定性，as?类型转换运算符的形式在尝试向下转换为子类类型时返回*可选*值。item as? Movie类型的结果Movie?，或“可选Movie”。

向下广播Movie应用于Song库数组中的实例时失败。为了解决这个问题，上面的示例使用可选绑定来检查可选Movie实际是否包含值（也就是说，查明downcast是否成功。）此可选绑定被写为“ if let movie = item as? Movie”，可以将其读作：

“尝试访问item作为一个Movie。如果这是成功的，请设置一个新的临时常量，调用movie存储在返回的可选参数中的值Movie。“

如果向下转换成功，movie则会使用该属性打印该Movie实例的描述，包括其名称director。使用类似的原理检查Song实例，并在库中找到一个合适的描述（包括artist名称）Song。

> 注意
>
> Casting实际上并不修改实例或更改其值。底层实例保持不变; 它只是作为它所投的类型的一个实例来处理和访问。

为Any和AnyObject类型强制转换

Swift提供了两种用于处理非特定类型的特殊类型：

- Any 可以表示任何类型的实例，包括函数类型。
- AnyObject 可以表示任何类类型的实例。

使用Any和AnyObject只有当你明确需要的行为以及它们提供的功能。对代码中期望使用的类型进行具体说明会更好。

下面是一个使用Any混合不同类型的例子，包括函数类型和非类类型。该示例创建一个名为的数组things，该数组可以存储类型的值Any：

```
var things = [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) -> String in "Hello, \(name)" })
```

该things数组包含两个Int值，两个Double值，一个String值，一个元组类型(Double, Double)，电影“捉鬼敢死队”，以及一个闭包表达式，它接受一个String值并返回另一个String值。

要发现特定类型的只是已知类型的固定或可变的Any或者AnyObject，你可以使用一个is或as一个模式switch语句的情况。下面的例子迭代things数组中的项目，并用switch语句查询每个项目的类型。几个switch语句的例子将它们的匹配值绑定到指定类型的常量，以使其值能够被打印：

```
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
```

```
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    cas
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter 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

该Any类型表示任何类型的值，包括可选类型。如果在Any预期类型值时使用可选值，Swift会给出警告。如果您确实需要将可选值用作Any值，则可以使用该as运算符将可选项显式转换为Any，如下所示。

```
let optionalNumber: Int? = 3
things.append(optionalNumber)        // Warning
things.append(optionalNumber as Any) // No warning
```