

# [译]PromiseKit入门

原文： [\*Getting Started with PromiseKit\*](#)

作者： [\*Michael Katz\*](#)

译者：kmyhy

异步编程真的让人头疼。无论你怎么小心，总是容易出现臃肿的委托、混乱的完成句柄以及长时间的代码调试！幸运的是，现在有一个更好的办法：promise。Promise 能够让你以基于事件的方式编写一连串的动作来实现异步。对于需要以确定顺序执行的动作尤其有用。在本教程中，你将学习如何使用第三方框架 PromiseKit 来让你的异步代码和头脑同时保持清晰。

通常，iOS 开发中都会有许多委托和回调。

你可能写过许多类似这样的代码：

- Y 负责管理 X。
- 告诉 Y 去抓取 X。
- 当 X 可用的时候，Y 通知它的委托对象。

Promise 将这种如同乱麻的关系理清成这个样子：

当 x 可用时，执行 y。

是不是优雅多了？Promise 还能够将错误处理和成功代码分离开来，导致代码在处理各种条件的时候更加清晰。它们能够很好滴解决复杂的、步骤繁多的工作流，比如web登录，执行 SDK 登录认证、处理和显示图片等。

Promise 是比较常见的，它的实现方式也很多，但在这篇教程中，我们将学习一个比较时髦的第三方 Swift 框架，叫做 PromiseKit。

## 开始

本文的示例项目是 WeatherOrNot，它是一个简单的实时天气应用。它的天气 API 用的是 OpenWeatherMap。访问这个 API 的模式和概念可以用

到任意 web 服务上。

从 [这里](#) 下载开始项目。PromiseKit 通过 cocoapods 发布，但开始项目中已经包括了 this pod。如果你以前没用过 CocoaPods，请参考 [这篇教程](#)。否则请直接在 CocoaPods 中安装 PromiseKit。除此之外，本教程不需要任何其他 CocoaPods 知识。

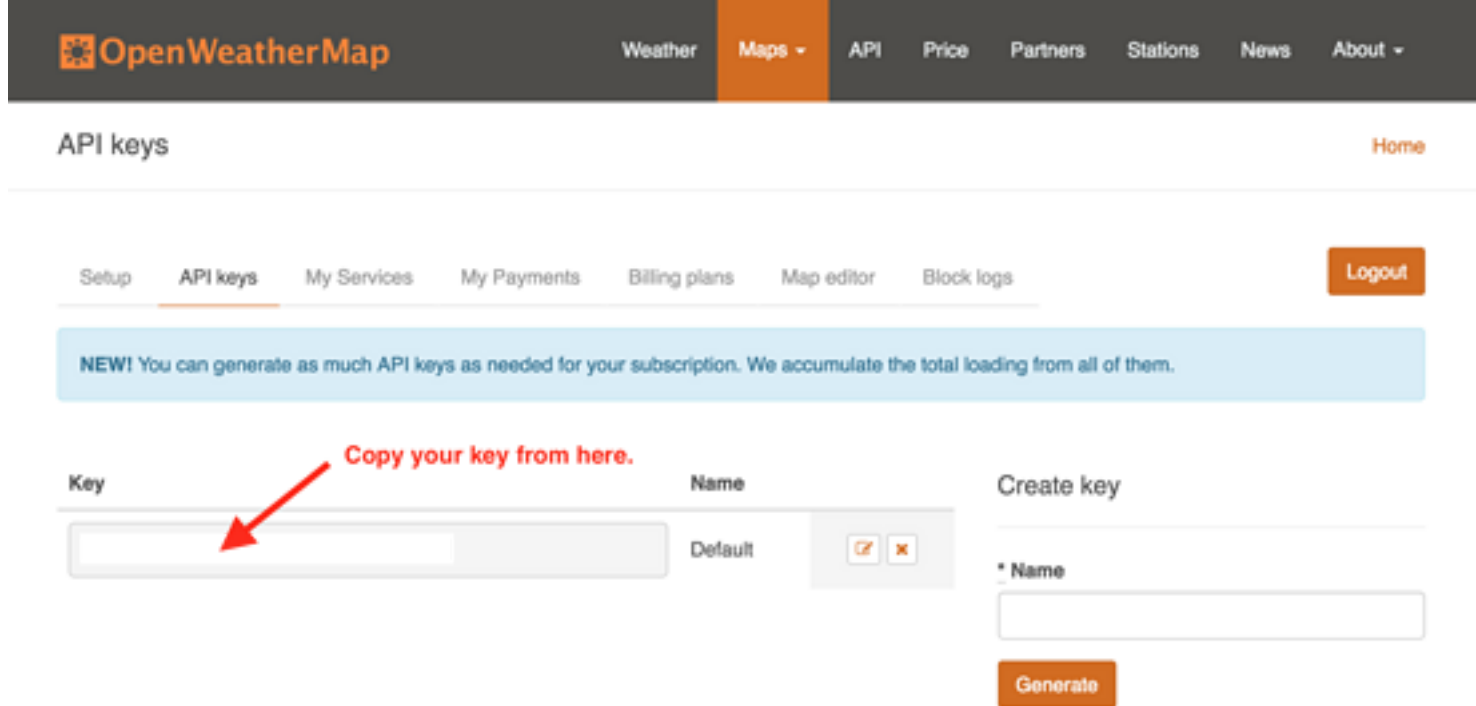
打开 PromiseKitTutorial.xcworkspace，你会看到项目结构非常简单。只有 5 个 swift 文件：

- AppDelegate.swift: 自动生成的 app delegate 文件。
- BrokenPromise.swift: 这个文件创建了一个空的 promise，用于暂时构成开始项目的一个部分。
- WeatherViewController.swift: 处理所有与用户交互的 view controller。这也是 Promise 的主要消费者。
- LocationHelper.swift: 一个辅助文件，用于实现 CoreLocation。
- WeatherHelper.swift: 一个辅助文件，用于包装天气数据提供者。

## OpenWeatherMap API

关于天气数据，这个 app 用 OpenWeatherMap 作为天气数据源。和大部分第三方 API 相同，要访问这个服务需要获取一个 API key。别担心，在本教程中使用的是它的免费套餐，完全够用了。

我们先获取一个 API key。访问 <http://openweathermap.org/appid>，先进行注册，注册后就可以在 [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys) 这里找到你的 API key。

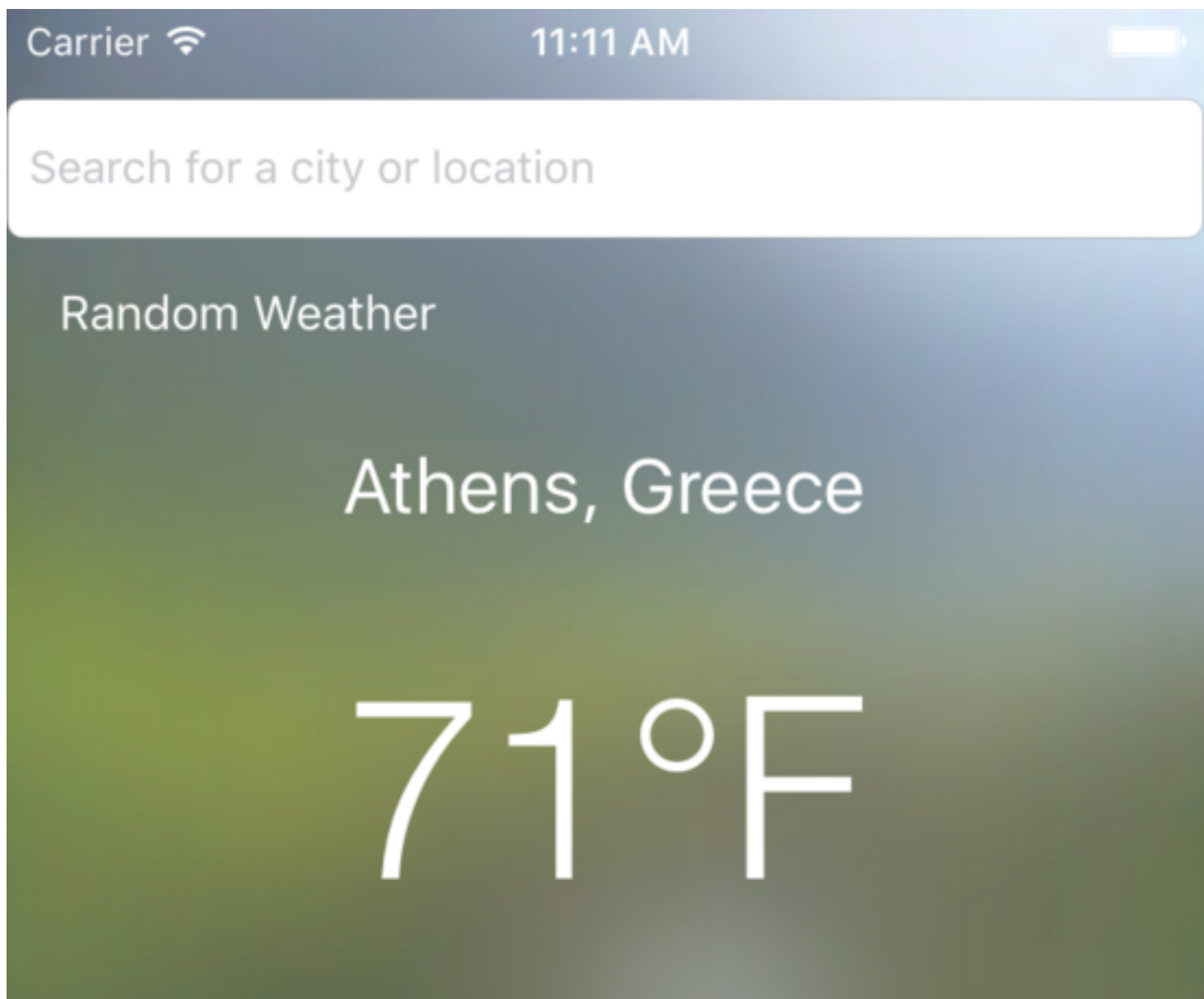


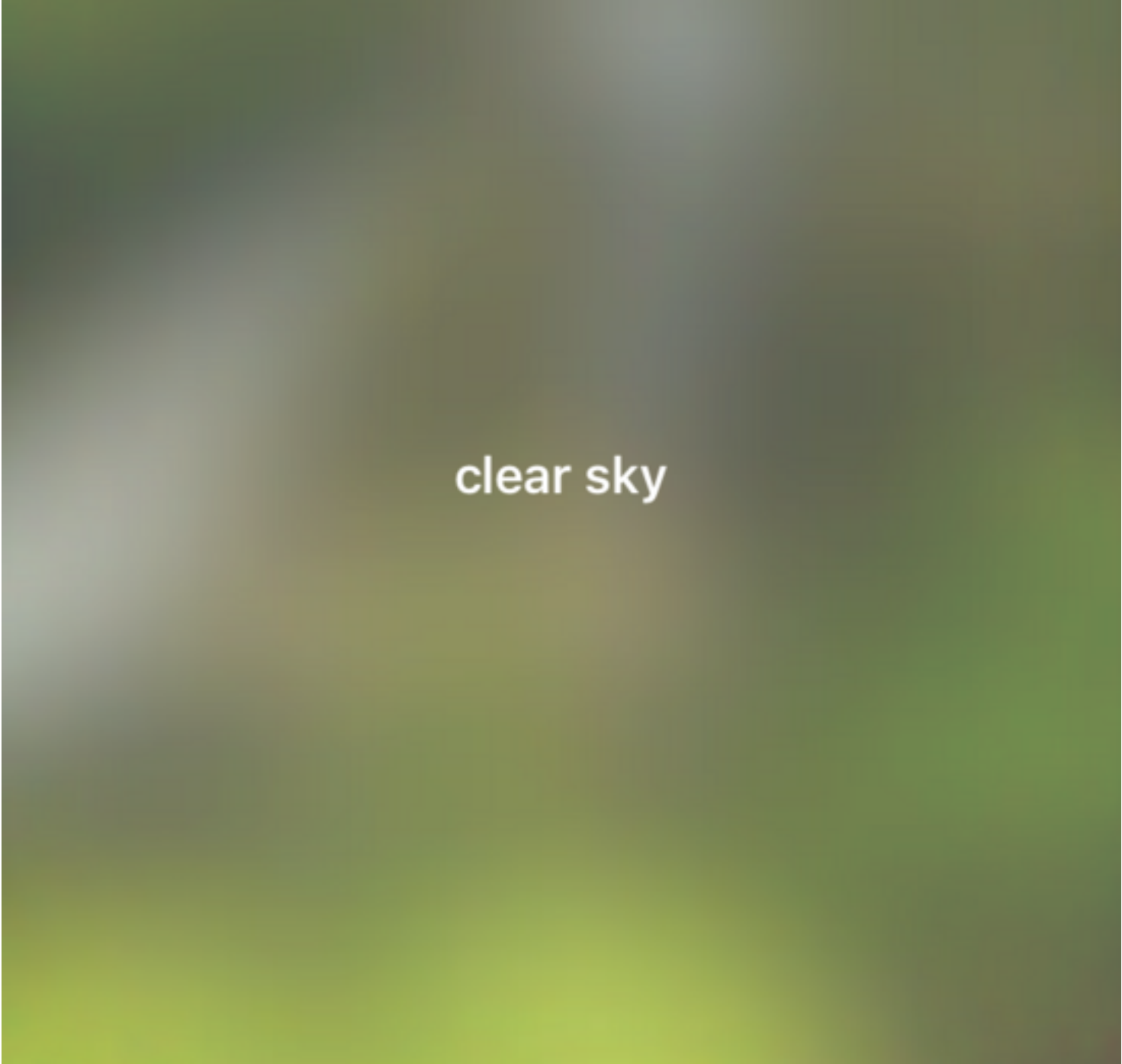
复制这个 key，将它粘贴到 WeatherHelper.swift 头部的 appId 常量中。

## 测试

运行 app，如果一切正常，你将看到雅典当前的天气。

呃……这个 app 现在有一个 bug（都会我们会解决它），所以 UI 显示可能有点慢。





clear sky

## 理解 Promise

在日常生活中的承诺 (promise) 你肯定知道。例如，你可以许诺自己在完成本程之后来一本冷饮。这个叙述中包含了一个动作（“来一杯饮料”），这个动作会在另一个动作完成（“完成这篇教程”）之后发生。变成中的承诺与此类似，即期望某些事情在未来当某些数据到达之后被执行。

承诺被用于实现异步。和传统方法，比如通过完成块或选择器进行回调不同，承诺可能被简单地进行链式连接，从而表达一连串异步动作。承诺和 Operation 有点像，也有一个执行生命周期并能被取消。

一个 PromiseKit 中的承诺会执行一个代码块，这个代码块应当用一个值来满足（或兑现）。如果这个值被兑现了，代码块就会被执行。如果这个块返回了一个承诺，则这个承诺也会执行（某个值被兑现），以此类推。如果在这个过程中发生错误，一个可选的 catch 块将替代这个块执行。

例如，一个 PromiseKit 承诺的口语化描述是这个样子：

```
doThisTutorial().then { haveAColdOne() }.catch { postToForum(error) }
```

## 关于 PromiseKit 中的承诺

PromiseKit 是承诺的 Swift 实现。它不是唯一实现，只是流行度最高而已。除了提供块式构造语法，PromiseKit 还提供了对许多常见 iOS SDK 类的封装和简单的错误处理机制。

要亲身体会 promise 是什么，请看一眼 BrokenPromise.swift 中的这个函数：

```
func BrokenPromise<T>(method: String = #function) -> Promise<T> {  
    return Promise<T>() { fulfill, reject in  
        let err = NSError(domain: "PromiseKitTutorial", code: 0, userInfo: [NSL  
            reject(err)  
        }  
    }  
}
```

方法返回了一个新的泛型化的 promise，它是 PromiseKit 中的核心类。它的构造函数使用一个块参数，这个块有两个参数：

- fulfill: 一个函数，当这个承诺所需的值被兑现时，调用这个函数。
- reject: 一个函数，当错误发生时，调用这个函数。

对于 BrokenPromise 来说，代码只会返回一个错误。这个辅助对象用于在你真正实现这个 app 时提示你仍然有功能需要实现。

## 创建 promise 对象

访问远程服务器是一种最常见的异步操作，因此我们从简单的网络调用开始。

看一眼 WeatherHelper.swift 中的

getWeatherTheOldFashionedWay(latitude:longitude:completion:) 方法。这个方法用指定的经纬度、完成块为参数，抓取天气数据。

但是，这个完成块无论是成功还是失败都会被调用。只会增加完成块的复杂性，因为你需要在代码中对成功和失败两种情况进行处理。

更过分的是，这个完成块是在后台线程中调用的，因此会导致 (accidentally :cough:) 在后台更新 UI !:[

这里用 promise 有用吗？答案是肯定的！

在  
getWeatherTheOldFashionedWay(latitude:longitude:completion:):  
后添加方法：

```
func getWeather(latitude: Double, longitude: Double) -> Promise<Weather> {
    return Promise { fulfill, reject in
        let urlString = "http://api.openweathermap.org/data/2.5/weather?lat=\(latitude)&lon=\(longitude)&appid=\(appID)"
        let url = URL(string: urlString)!
        let request = URLRequest(url: url)

        let session = URLSession.shared
        let dataTask = session.dataTask(with: request) { data, response, error
            if let data = data,
                let json = (try? JSONSerialization.jsonObject(with: data, options: .mutableLeaves)) as? [String: Any],
                let result = Weather(jsonDictionary: json) {
                fulfill(result)
            } else if let error = error {
                reject(error)
            } else {
                let error = NSError(domain: "PromiseKitTutorial", code: 0,
                                     userInfo: [NSLocalizedDescriptionKey: "Unknown error"])
                reject(error)
            }
        }
        dataTask.resume()
    }
}
```

这个方法和 getWeatherTheOldFashionedWay 方法一样也使用 URLSession，但没有使用完成块，而是将网络操作放在了一个 Promise 中。

当 `dataTask` 的 `completion` 处理回调中，如果成功返回数据，将 JSON 序列化后创建一个 `Weather` 对象。用这个对象调用 `fulfill` 函数，完成这个承诺。

如果发生错误，用 `error` 对象调用 `reject` 函数。

否则，表明既没有返回 JSON 数据也没有错误发生，则创建一个 `NSError` 传递给 `reject` 函数，因为调用 `reject` 函数必须要一个 `NSError` 参数。

然后，在 `WeatherViewController.swift` 中将 `handleLocation(city:state:latitude:longitude:)` 替换成：

```
func handleLocation(city: String?, state: String?,
                    latitude: CLLocationDegrees, longitude: CLLocationDegrees) {
    if let city = city, let state = state {
        self.placeLabel.text = "\(city), \(state)"
    }

    weatherAPI.getWeather(latitude: latitude, longitude: longitude).then { we
        self.updateUIWithWeather(weather: weather)

    }.catch { error in
        self.tempLabel.text = "--"
        self.conditionLabel.text = error.localizedDescription
        self.conditionLabel.textColor = errorColor
    }
}
```

太棒了，使用 `promise` 时只需要提供一个 `then` 块和一个 `catch` 块！

新的 `handleLocation` 方法和原来相比，好了许多。首先，单一的完成块被分为两个可读性更好的块：`then` 用于成功 `catch` 用于失败。其次，默认 `PromiseKit` 在主线程中执行这两个块，因此不会导致在后台线程中刷新 UI 的错误发生。

## PromiseKit Wrapper

`Promise` 很好，但 `PromiseKit` 并不仅仅是这些。除了 `Promise`，`PromiseKit` 还对常见的 iOS SDK 方法进行了扩展，让它们能够以承诺的方法表达。例如，`URLSession data task` 方法的完成块可以替换为 `promise`。

将 `getWeather(latitude:longitude:)` 方法替换为:

```
func getWeather(latitude: Double, longitude: Double) -> Promise<Weather> {  
    return Promise { fulfill, reject in  
        let urlString = "http://api.openweathermap.org/data/2.5/weather?lat=" +  
            "\ (latitude)&lon=\ (longitude)&appid=\ (appID)"  
        let url = URL(string: urlString)!  
        let request = URLRequest(url: url)  
  
        let session = URLSession.shared  
  
        // 1  
        let dataPromise: URLDataPromise = session.dataTask(with: request)  
  
        // 2  
        _ = dataPromise.asDictionary().then { dictionary -> Void in  
  
            // 3  
            guard let result = Weather(jsonDictionary: dictionary as! [String : A  
                let error = NSError(domain: "PromiseKitTutorial", code: 0,  
                    userInfo: [NSLocalizedStringKey: "Unknown  
                reject(error)  
                return  
            }  
  
            fulfill(result)  
  
            // 4  
        }.catch(execute: reject)  
    }  
}
```

看到了吗? PromiseKit 的 Wrapper 就这么简单! 解释一下上述代码:

1. PromiseKit 提供了一个 `URLSession.dataTask(with:)` 方法的重载, 让它返回一个 `URLDataPromise`, 这是一个类型化的 Promise。注意, data promise 自动启动它所包含的 data task。
2. 所返回的这个 dataPromise 有一个便利方法 `asDictionary()`, 这个方法会为你处理 JSON 的序列化, 可以大大节省你的代码!
3. 因为已经解析好 dictionary, 可以直接用它创建一个 result 对象。我们用 `guard let` 语句确保从 dictionary 创建 Weather 对象一定成功。如果不, 创建一个 NSError 并调用 reject 函数, 和前面一样。否则, 用 result 对象调用 fulfill 函数。



4. 在这个过程中，有可能网络请求失败，或者 JSON 序列化失败。在之前的方法中我们必须分别检查这两种情况。而这里，只需要一个 catch 块就能让所有错误进入失败块。

在这个方法中，两个 promise 被链接在一起。第一个 promise 是 dataPromise，它从 URL 请求中返回数据 data。第二个 promise 是 asDictionary()，它用 data 做参数并将它转换成字典返回。

## 添加地点

现在网络部分已经就绪，我们来看单位功能。无论你是否有幸去过希腊，这个 app 都不会给你真正想要的的数据。要解决这个，我们需要使用设备的定位功能。

在 WeatherViewController.swift 中，将 updateWithCurrentLocation() 替换为：

```
private func updateWithCurrentLocation() {
    // 1
    _ = locationHelper.getLocation().then { placemark in
        self.handleLocation(placemark: placemark)
    }.catch { error in
        self.tempLabel.text = "--"
        self.placeLabel.text = "--"
        switch error {

            // 2
            case is CLError where (error as! CLError).code == CLError.Code.denied:
                self.conditionLabel.text = "Enable Location Permissions in Settings"
                self.conditionLabel.textColor = UIColor.white
            default:
                self.conditionLabel.text = error.localizedDescription
                self.conditionLabel.textColor = errorColor
        }
    }
}
```

这里使用了辅助类来进行 Core Location 调用。待会再来实现这个类。getLocation() 返回一个 promise，这个 promise 会从当前位置获得一个地名 placemark。

这个 catch 块显示了各种错误并在单个 catch 块中对错误进行处理。用一个 switch 语句，根据用户是否授予位置访问权限还是其他类型的错误来给予不同的提示。

然后，在 LocationHelper.swift 中将 getLocation() 替换为：

```
func getLocation() -> Promise<CLPlacemark> {  
    // 1  
    return CLLocationManager.promise().then { location in  
  
        // 2  
        return self.coder.reverseGeocode(location: location)  
    }  
}
```

这里利用了前面介绍过的 PromiseKit 的两个概念：PromiseKit Wrapper 和 promise 链。

CLLocationManager.promise() 返回了一个当前位置的 promise。

一旦获取到用户当前位置，将位置传递给 CLGeocoder.reverseGeocode(location:) 方法，这也返回了一个 promise，返回反地理编码的位置。

通过 promise，两个异步动作被链接在 3 行代码里。因为所有的错误处理都由调用者的 catch 块处理，我们也不需要显式的异常处理。

运行 app。接受地理位置授权请求，你当前位置（模拟器）的温度显示了。成功了！

[https://koenig-media.raywenderlich.com/uploads/2016/10/2\\_build\\_and\\_run\\_with\\_location.png](https://koenig-media.raywenderlich.com/uploads/2016/10/2_build_and_run_with_location.png)

## 搜索其它位置

干得不错，但用户想知道其它地方的气温怎么办？

在 WeatherViewController.swift 中将 textFieldShouldReturn(\_) 替换为

(暂时不用管编译器报的“missing method”错误):

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {  
  
    textField.resignFirstResponder()  
  
    guard let text = textField.text else { return true }  
    _ = locationHelper.searchForPlacemark(text: text).then { placemark -> Voi  
        self.handleLocation(placemark: placemark)  
    }  
    return true  
}
```

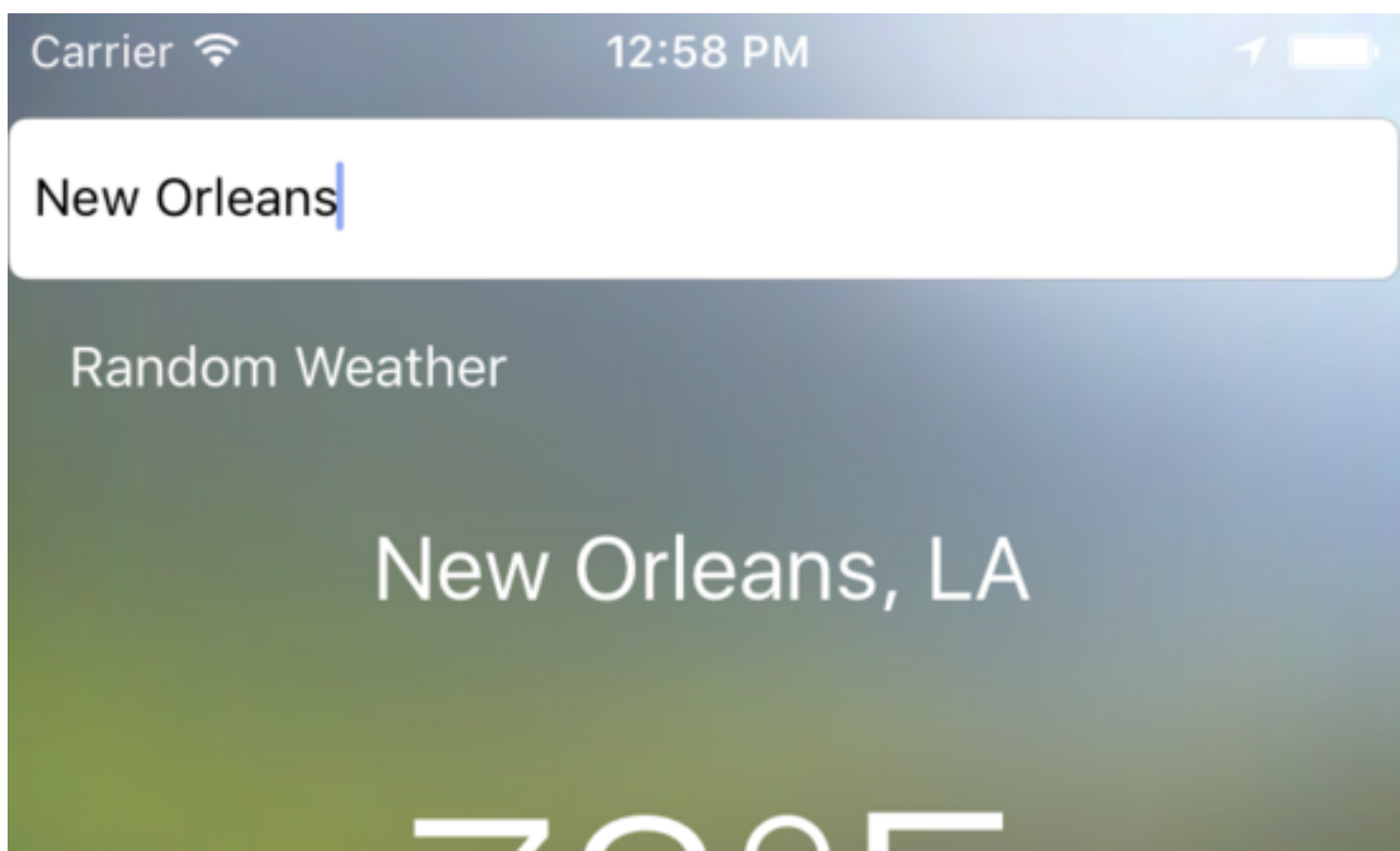
这里使用了和其它几个 promise 的相同模板：查找地名，找到后刷新 UI。

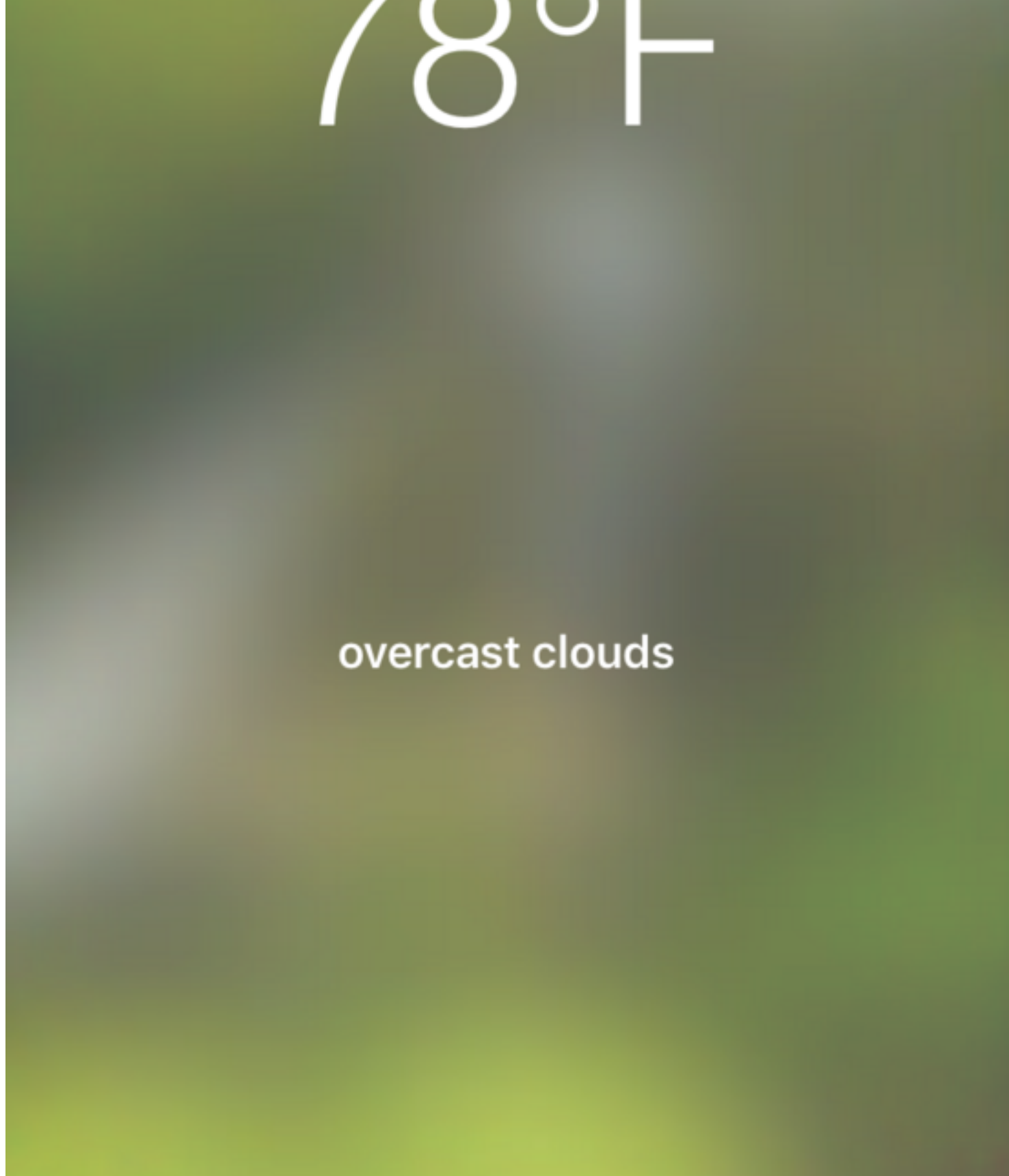
然后，在 LocationHelper.swift 添加方法：

```
func searchForPlacemark(text: String) -> Promise<CLPlacemark> {  
    return CLGeocoder().geocode(text)  
}
```

很简单！PromiseKit 已经对 CLGeocoder 进行了扩展，会查找匹配的 placemark 并用一个 promise 返回 placemark。

运行 app，这次在顶部搜索栏中输入一个城市名称后点击回车。这回找到一个最匹配的城市名并获取天气信息。





## 线程

我们已经习惯了将所有的块都放在主线程中执行。这是一个很好的特性，因为 view controller 中大部分工作都和刷新 UI 有关。但是，对于耗时任务，应当在后台线程中进行，这样不会阻塞 app。

我们接下来会从 OpenWeatherMap 加载一个图标，以表示当前的天气状况。

在 WeatherHelper 的 `getWeather(latitude:longitude:)` 方法后添加这个方法：

```

func getIcon(named iconName: String) -> Promise<UIImage> {
    return Promise { fulfill, fail in
        let urlString = "http://openweathermap.org/img/w/\(iconName).png"
        let url = URL(string: urlString)!
        let request = URLRequest(url: url)

        let session = URLSession.shared
        let dataPromise: URLDataPromise = session.dataTask(with: request)
        let backgroundQ = DispatchQueue.global(qos: .background)
        _ = dataPromise.then(on: backgroundQ) { data -> Void in
            let image = UIImage(data: data)!
            fulfill(image)
        }.catch(execute: fail)
    }
}

```

这里，我们在 `then(on:execute:)` 方法中用 `on` 参数指定图片的加载在后台队列中进行。PromiseKit 会将繁重任务放到指定的 `dispatch` 中进行。

现在，promise 在后台队列中被兑现，这样调用者就必须自己保证 UI 刷新在主队列中进行了。

回到 `WeatherViewController.swift`，在 `handleLocation(city:state:latitude:longitude:)` 方法中，将调用 `getWeather(latitude:longitude:)` 的语句修改为：

```

// 1
weatherAPI.getWeather(latitude: latitude, longitude: longitude).then { weat
    self.updateUIWithWeather(weather: weather)

// 2
return self.weatherAPI.getIcon(named: weather.iconName)

// 3
}.then(on: DispatchQueue.main) { icon -> Void in
    self.iconImageView.image = icon

}.catch { error in
    self.tempLabel.text = "--"
    self.conditionLabel.text = error.localizedDescription
    self.conditionLabel.textColor = errorColor
}

```

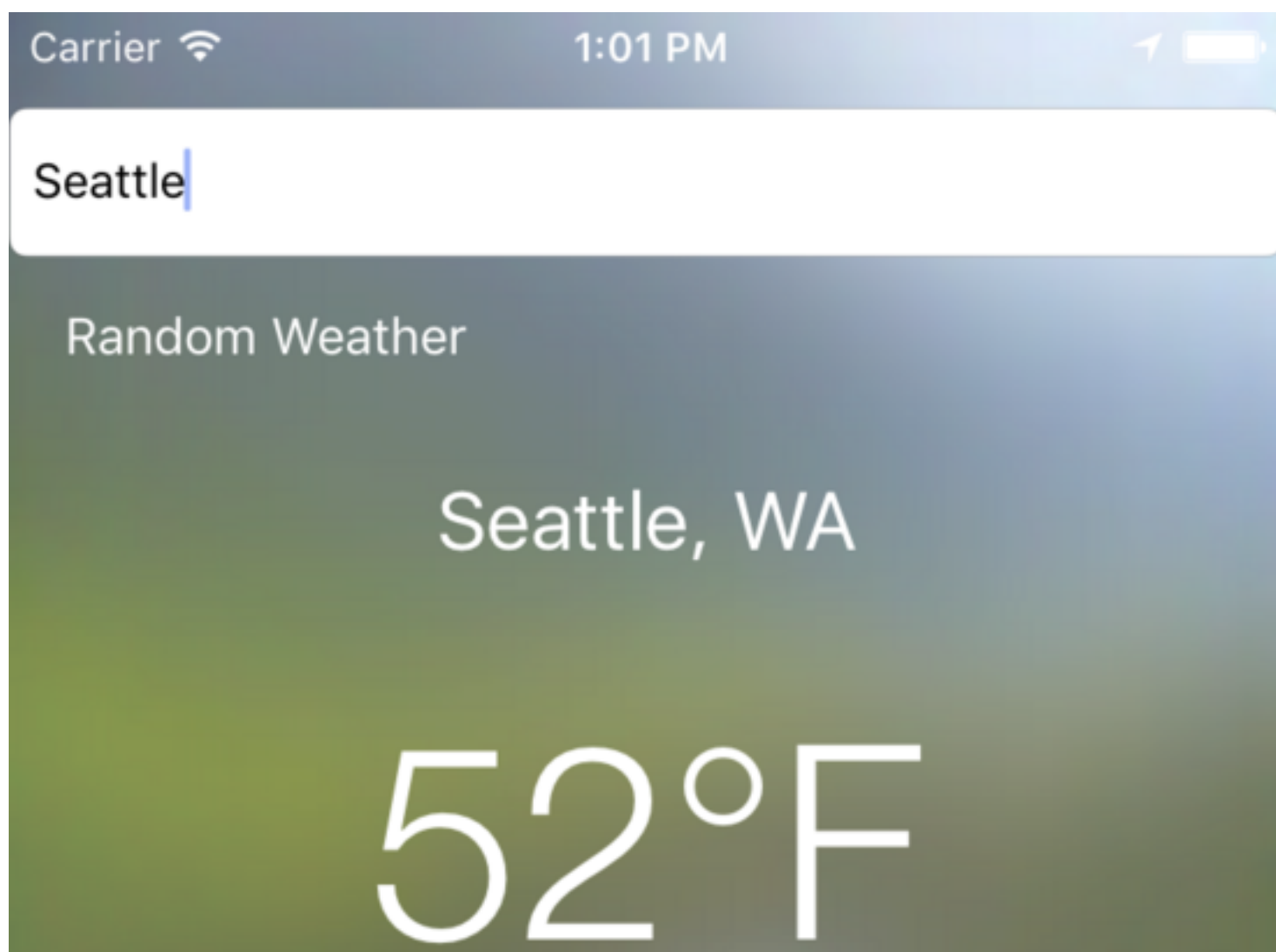
在这个调用中，有 3 个地方与之前有细微的区别：

1. 首先，`getWeather(latitude:longitude:)` 的 `then` 块的返回值由原来的 `Void` 修改 `promise`。这意味着当 `getWeather` 的承诺兑现时，又给出了另一个承诺。
2. 用 `getIco` 方法创建一个新的承诺…以得到一个图标。
3. 在承诺链中再加一个 `then` 块，当 `getIco` 被兑现时，这个块要在主线程中执行。

这样，承诺以一种顺序执行的步骤链接在一起。当一个承诺兑现，下一个承诺会被执行时，以此类推直到最后一个 `then` 或者有错误发生——即 `catch` 块被调用。这种方式比起嵌套完成块来说有两大好处：

1. 承诺以单链的形式构建，易于阅读和维护。每个 `then` 块都有单独的上下文，避免逻辑和状态相互污染。竖列的代码块不需要很多缩进，读起来更加轻松。
2. 所有错误代码都在一个地方进行处理。例如，在一个复杂的流程中，比如用户登录，只需要一个错误对话框就能够显示每个步骤所发生的错误。

运行 app，图片显示了！





moderate rain

## 封装承诺

如何调用不支持 PromiseKit 的老代码、SDK 或者第三方库？PromiseKit 有一个 promise wrapper。

以我们的 APP 为例。因为天气状况总是有限的，没有必要每次都从 web 抓取表示天气状况的图片，不但效率低下，而且会造成浪费。

在 WeatherHelper.swift 已经有一个辅助方法，将图片加载并保存到本地缓存中。这些函数在后台线程中进行文件 IO 操作，当操作完成时调用异步完成块。这是最普通的方法，PromiseKit 提供了一种替代方法。

将 WeatherHelper 中的 getIcon(named:) 替换为(同样, 暂时忽略编译器的报警):

```

func getIcon(named iconName: String) -> Promise<UIImage> {

    // 1
    return wrap {
        // 2
        getFile(named: iconName, completion: $0)

        } .then { image in
            if image == nil {
                // 3
                return self.getIconFromNetwork(named: iconName)

            } else {
                // 4
                return Promise(value: image!)
            }
        }
    }
}

```

代码解释如下：

1. wrap(body:) 能够将跟在多个完成块后面的某个函数封装成一个承诺。
2. getFile(named: completion:) 有一个完成块参数 @escaping (UIImage?) -> Void, 它会被转换成一个 Promise。在 wrap 的块中，调用了这个函数，将完成块参数传入。
3. 如果图片未缓存到本地，返回一个承诺，从网络抓取图片。
4. 如果图片缓存有效，返回一个值承诺 (value promise) 。

这是一种新的 promise 的用法。如果创建承诺时使用一个已经兑现的值，将立即调用 then 块。这样，如果图片已经在本地，它会立即返回。这种方式既能够创建一个承诺去异步执行某件事情（比如从网络加载），也能同步执行某件事情（比如使用一个内存中的值）。这在你有本地缓存时是有用的，比如这里的图片。

要让上述代码能够工作，我们必须在获取到图片时对它进行缓存。在前面的方法后面添加方法：

```

func getIconFromNetwork(named iconName: String) -> Promise<UIImage> {
    let urlString = "http://openweathermap.org/img/w/\(iconName).png"
    let url = URL(string: urlString)!

```



```

let request = URLRequest(url: url)

let session = URLSession.shared
let dataPromise: URLDataPromise = session.dataTask(with: request)
return dataPromise.then(on: DispatchQueue.global(qos: .background)) { data
    return firstly { Void in
        return wrap { self.saveFile(named: iconName, data: data, completion:
        }.then { Void -> Promise<UIImage> in
            let image = UIImage(data: data)!
            return Promise(value: image)
        }
    }
}
}

```

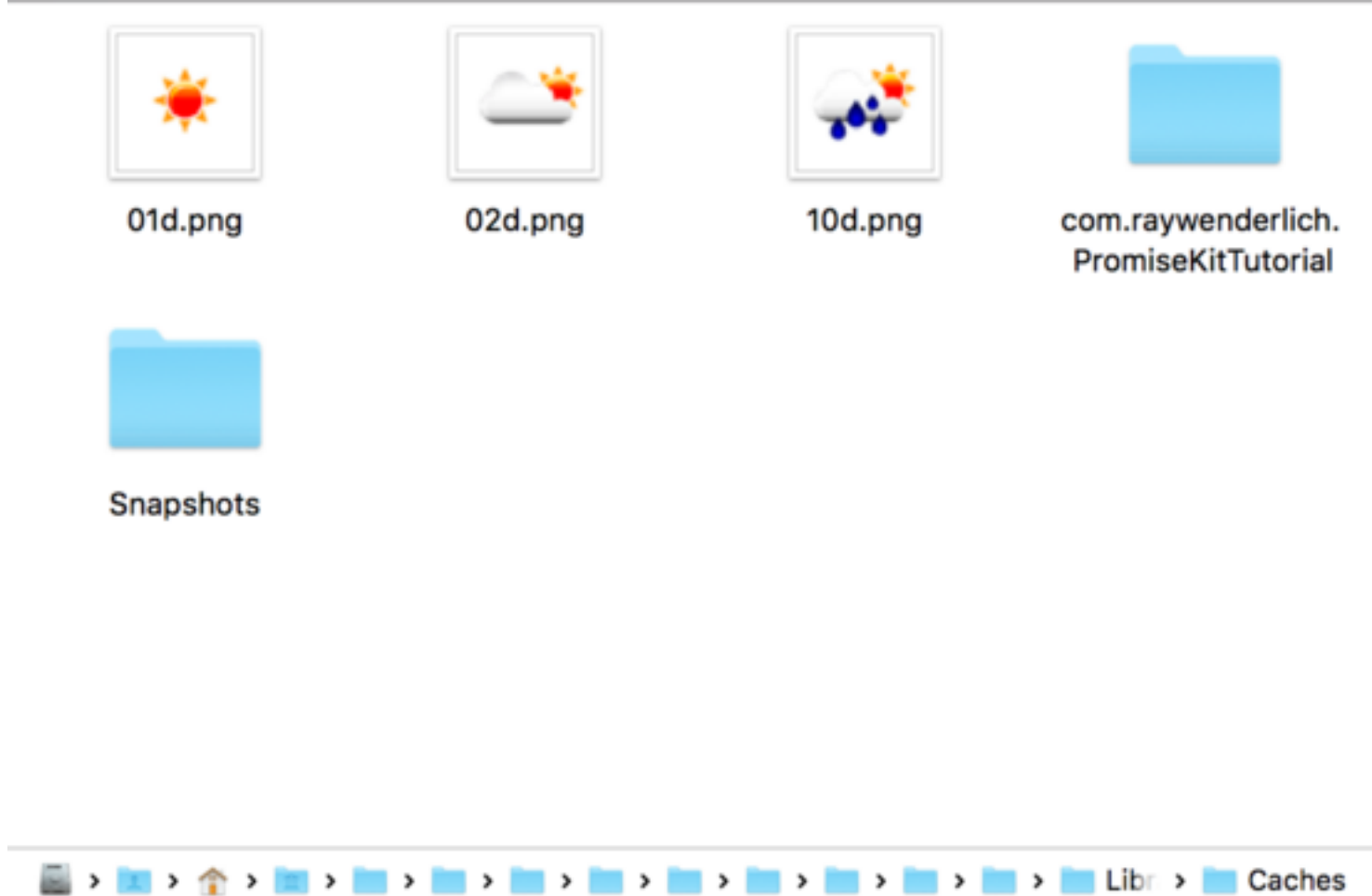
和先前的 `getIcon(named:)` 方法一样，但是在 `dataPromise` 的 `then` 块中调用了 `saveFile` 方法，这个方法进行了和 `getFile` 方法一样的封装。

这里用到了一个新结构，`firstly`。`firstly` 是一个语法糖，简单滴执行它的承诺。其实仅仅是添加了一层封装以便更易读。因为 `saveFile` 方法调用是加载图标后的一个附带功能，用 `firstly` 能够确保执行的顺序以便我们可以对这个承诺更有信心一点。

当你第一次请求图片时会是这个样子：

1. 首先，加载一个 `URLRequest`。
2. 加载成功后，数据保存到文件中。
3. 保存完后，将数据转换成图片传递给下个承诺链。

如果你运行 app，不会有什么不同，但通过文件系统你可以看到图片都被保存了。你可以在控制台中搜索 `Save iamge to:`，它会显示文件保存的 URL 地址，你可以在硬盘上找到这个文件：



## 确认动作

看过了 PromiseKit 的语法，你可能会问：既然有 then 和 catch，那么有 finally 吗（比如执行一些清理），确保某些动作总是会发生，而无论是否成功？答案是：always。

在 WeatherViewController.swift 中修改 `handleLocation(city:state:latitude:longitude:)`，当从服务器抓取天气数据时，在状态栏中显示一个小菊花。

在调用 `weatherAPI.getWeather...` 之前插入代码：

```
UIApplication.shared.isNetworkActivityIndicatorVisible = true
```

然后，在 catch 块后面添加：

```
.always {  
    UIApplication.shared.isNetworkActivityIndicatorVisible = false  
}
```

然后，为了让编译器不再报“unused result”警告，将整个表达式赋给一个 `_`。

这是 `always` 的一个常规用法。无论是加载成功还是出错，以及网络活动是否完成，网络活动状态都应当隐藏。类似的，可以用 `always` 关闭 `socket`，数据库连接或者断开硬件服务。

## 定时器

有一种例外情况，即承诺会在数据有效并经过某个固定时间周期之后才会兑现。当前，当天气信息加载后就不会刷新了。我们可以修改它，让它每隔一个小时就更新一次。

在 `updateWithCurrentLocation()` 方法最后添加代码：

```
_ = after(interval: oneHour).then {  
  self.updateWithCurrentLocation()  
}
```

`.after(interval:)` 创建一个承诺，以指定时间间隔兑现。不幸的是，这是一个一次性的定时器。要每小时刷新一次，需要在 `updateWithCurrentLocation()` 中递归。



## 并列承诺

当前的所有承诺都是孤立或者以某种顺序链式执行。`PromiseKit` 也提供了一个功能，将多个承诺同时兑现。有 3 个函数用于等待多个承诺。第一个 `race`，当一堆承诺兑现时返回第一个承诺。也就是，第一个完成的胜出。

另外两个函数是 `when` 和 `join`。它们都是在指定的承诺被兑现之后调用。只是 `rejected` 块有所不同。`join` 在拒绝之前总是等待所有的承诺完成，看它

们之中是否有被拒绝的。而 `when(fulfilled:)` 只要有任何一个承诺被拒绝它就拒绝。另外，`when(resolved:)` 会等待所有的承诺完成，但 `then` 块总会调用，`catch` 块永远不会调用。

注：对于所有的聚合函数，所有的单一承诺都会继续指导它们要么兑现要么拒绝，无论聚合函数的行为是什么。例如，如果三个承诺使用了 `race` 函数，当第一个承诺完成时，`then` 块被调用。但是其他两个未满足的承诺仍然会继续执行，一直到它们也被解决。

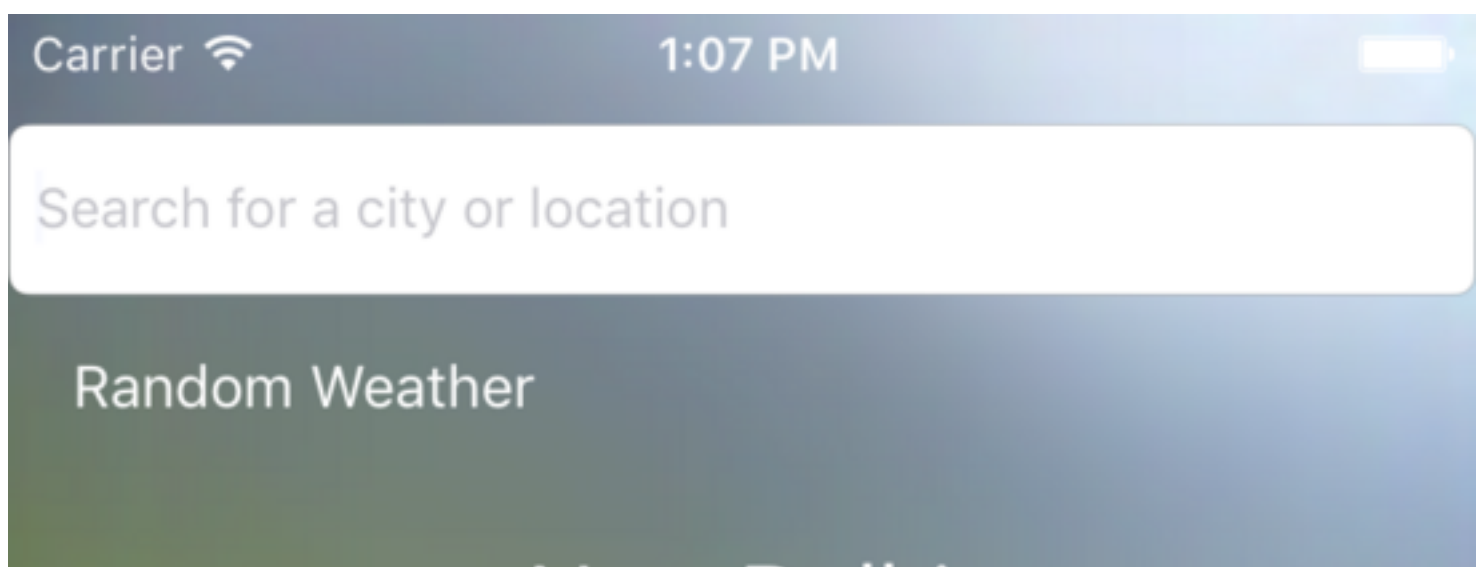
以随机显示任意城市的天气为例。因为用户不知道会显示什么城市，app 一次会抓取多个城市，但它只处理第一个城市。这会造成一种随机的假象。

将 `showRandomWeather(_:)` 替换为：

```
@IBAction func showRandomWeather(_ sender: AnyObject) {
    let weatherPromises = randomCities.map { weatherAPI.getWeather(latitude:
    _ = race(promises: weatherPromises).then { weather -> Void in
        self.placeLabel.text = weather.name
        self.updateUIWithWeather(weather: weather)
        self.iconImageView.image = nil
    }
}
```

这里我们创建了多个承诺去抓取城市列表中的天气。这些承诺用 `race(promises:)` 函数形成进行竞争关系。只有第一个被满足的承诺会调用 `then` 块。理论上，这是一种随机选择，因为服务器状况是不定的，但这个例子的说服力不是很强。注意所有的承诺都会继续执行，因此仍然会有 5 个网络调用，当然只有一个承诺会被关心。

运行 app。当 app 启动，点击 Random Weather。



New Delhi

63°F

clear sky

关于天气图标的刷新和错误处理就留给读者练练手了 :)

结束

在 [这里](#) 下载最后完成项目。

请在阅读 PromiseKit 文档: <http://promisekit.org/> , 尽管它看起来很

难。FAQ <http://promisekit.org/faq/> 对于调试信息很有帮助。

PromiseKit 是一个活跃的 pod，为了在自己的项目中安装 cocoapods，并保持它的更新，你可能需要 [研究一下 CocoaPods 的用法](#)。

最后说一句，Promise 还有其他 Swift 实现。其中一个流行的实现就是 BrightFutures。

如果你有任何建议、问题和评论，请在下面留言。