Chapter 4: Async

By Tim Condon

在本章中，您将了解异步和非阻塞架构。您将发现Vapor对这些架构的方法以及如何使用它。最后，本章简要概述了SwaporNIO，这是Vapor使用的核心技术。

Async

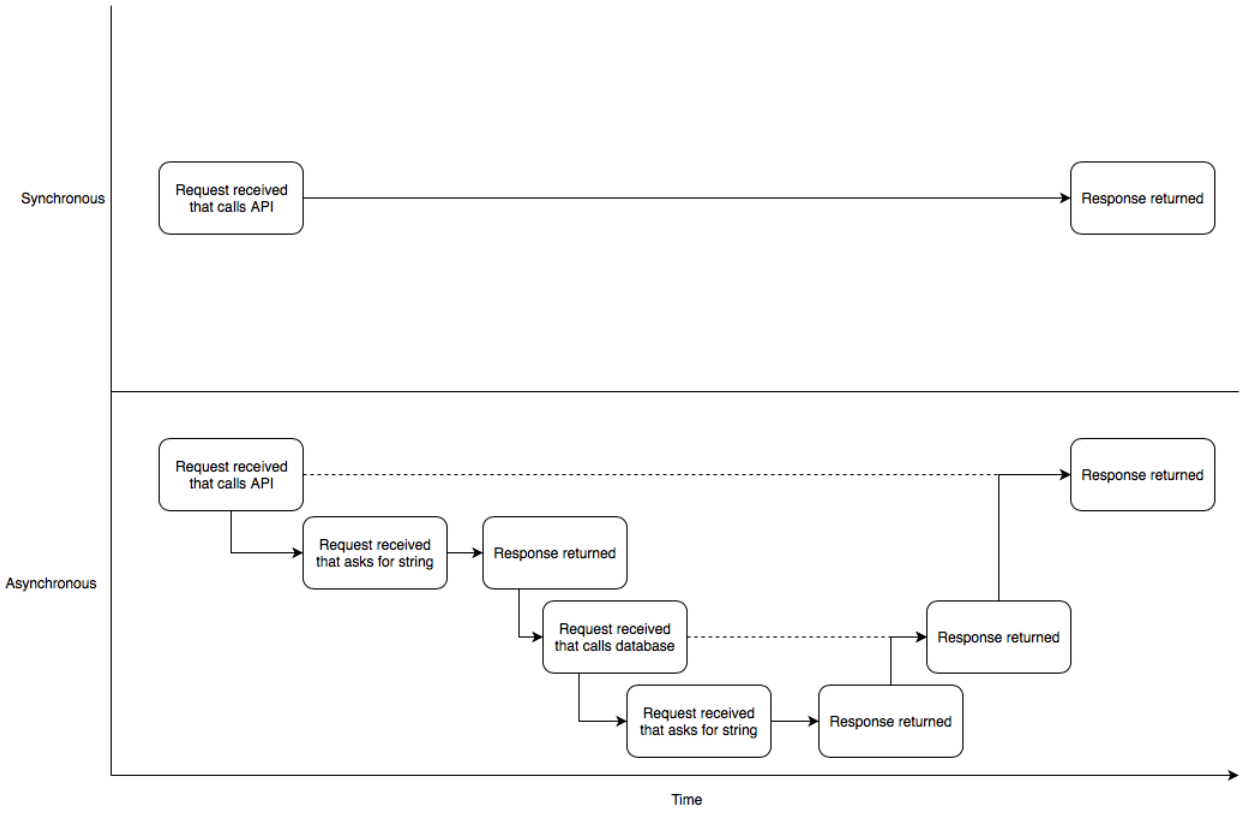
Vapor 3最重要的新功能之一是Async。它也可能是最令人困惑的一个。它为什么如此重要呢?

考虑一下您的服务器只有一个线程和四个客户端请求的情况，顺序如下:

1. 股票报价请求。这导致在另一台服务器上调用API。
2. 对静态CSS样式表的请求。 CSS无需查找即可立即使用。
3. 对用户个人图像的请求。必须从数据库中提取个人图像。
4. 请求一些静态HTML。 HTML无需查找即可立即使用

在同步服务器中，服务器的唯一线程将阻塞，直到返回股票报价。然后它返回股票报价和CSS样式表。它在数据库获取个人图像完成期间再次阻塞。只有在用户的个人图像被发送之后，服务器才会将静态HTML返回给客户端。

另一方面，在异步服务器中，线程启动调用以获取股票报价并将请求放在一边直到完成。然后它返回CSS样式表，启动数据库获取图像并返回静态HTML。当放在一边的请求完成时，线程将继续处理它们并将结果返回给客户端。



“但是，等等！”，你说，“服务器有多个线程。”你是对的。
但是，服务器可以拥有多少个线程是有限制的。创建线程需要使用资源。在线程之间切换上下文的代价是很高的，并且确保所有数据访问都是线程安全的，这非常耗时且容易出错。因此，尝试仅通过添加线程来解决问题是一种糟糕，低效的解决方案

Futures and promises

为了在等待响应时“搁置”请求，您必须将其包装在**promise**中，以便在收到响应时恢复其工作。实际上，这意味着您必须更改可以搁置一边的函数的返回类型。在同步环境中，您可能具有以下函数：

```
func getAllUsers() -> [User] {
  // do some database queries
}
```

在异步环境中，这将无法工作，因为在getAllUsers()必须返回时，您的数据库调用可能尚未完成。你知道你将来能够返回[User]但现在不能这样做。在Vapor中，您将返回被包装在**Future**中的结果。你可以编写你的函数，如下所示：

```
func getAllUsers() -> Future<[User]> {
  // do some database queries
}
```

返回Future<[User]>允许您将某些内容返回给函数调用者，即使此时可能没有任何内容可以返回。但是调用者知道该函数在将来的某个时刻返回[User]。

与Futures合作

与Futures合作起初可能令人困惑，但由于Vapor广泛使用它们，它们很快就会成为第二天性。在大多数情况下，当您从函数接收Future时，您希望对Future中的实际结果执行某些操作。由于函数的结果尚未实际返回，所以您提供了一个回调，以便在Future完成时执行。

在上面的示例中，当您的程序到达getAllUsers()时，它会在EventLoop上发出数据库请求。EventLoop进程工作，简单的话来说可以被认为是一个线程。 getAllUsers()不会立即返回实际数据，而是返回Future。这意味着EventLoop暂停执行该代码并处理排队的任何其他代码。例如，代码的另一部分，其中返回了Future结果。数据库调用返回后，EventLoop然后执行回调。

如果回调调用另一个返回Future的函数，则在原始回调中提供另一个回调，以便在第二个Future完成时执行。这就是为什么你最终会链接或嵌套许多不同的回调。这是与futures合作的难点。异步函数需要完全转变如何考虑代码。

解析 futures

Vapor为futures提供了许多便利功能，以避免直接处理它们的必要性。但是，有很多场景需要等待未来的结果。为了演示，假设您有一条返回HTTP状态代码204 No Content的路由。此路由使用类似上述功能从数据库中提取用户列表，并在返回之前修改列表中的第一个用户。

为了使用该调用的结果，您必须提供一个闭包，以便在Future解析时执行。您将使用两个主要函数来执行此操作：

- **flatMap(to:)**：在一个future上执行并返回另一个future。 回调接收已解析的future并返回另一个Future。
- **map(to:)**：在一个future上执行并返回另一个future。回调接收到已解析的future并返回除 Future之外的类型，然后map(to:)将其包装在Future中。

这两种选择都需要Future，并产生不同的Future，通常是不同的类型。重申一下，不同之处在于，如果处理Future结果的回调还将返回Future，请使用flatMap(to:)。如果回调返回Future以外的类型，请使用`map(to:)`。例如：

```
// 1
return database
  .getAllUsers()
  .flatMap(to: HTTPStatus.self) { users in
    // 2
    let user = users[0]
    user.name = "Bob"
    // 3
    return user.save(on: req)
      .map(to: HTTPStatus.self) { user in
        //4
        return .noContent
    }
}
```

这是它的作用：

1. 从数据库中获取所有用户。如上所述，getAllUsers()返回Future <[User]>。由于完成此Future的结果是另一个Future（参见步骤3），因此使用flatMap(to :)来解析结果。

flatMap(to:)的闭包接收已完成的future用户 - 数据库中所有用户的数组，类型[User] - 作为其参数。这个.flatMap(to:)返回Future<HTTPStatus>。

2. 更新第一个用户的名称。

3. 将更新的用户保存到数据库。这将返回Future <User>，但您需要返回的 HTTPStatus 值还不是Future，因此请使用map(to:)。

4. 返回适当的HTTPStatus值。

正如您所看到的，对于顶层promise，您使用flatMap(to:)，因为您提供的闭包返回Future。内部promise，闭包返回一个非future的HTTPStatus，使用map(to:)。

Transform

有时你不关心future的结果，只关心它成功完成。在上面的示例中，您不使用save(on:)的已解析结果并返回其他类型。对于此场景，您可以使用transform(to:)简化步骤3：

```
return database
  .getAllUsers()
  .flatMap(to: HTTPStatus.self) { users in
    let user = users[0]
    user.name = "Bob"
    return user.save(on:req)
      .transform(to: .noContent)
}
```

这有助于减少嵌套量，并使代码更易于阅读和维护。你会在整本书中看到这个用法。

Flatten

有些时候你必须等待一些futures完成。在数据库中保存多个模型时会出现这样的示例。在这种情况下，您使用flatten(on:)。例如：

```
static func save(_ users: [User], request: Request)
    -> Future<HTTPStatus> {
  // 1
  var userSaveResults: [Future<User>] = []
  // 2
  for user in users {
    userSaveResults.append(user.save(on: request))
  }
  // 3
  return userSaveResults.flatten(on: request)
    //4
    .transform(to: .created)
}
```

这是它的作用：

1. 定义一个Future <User>的数组，即步骤2中save(on:)的返回类型。
2. 遍历用户数组中的每个用户，并将user.save(on:)的返回值附加到数组。
3. 使用flatten(on:)等待所有futures完成。这需要一个Worker，实际执行工作的线程。这通常是Vapor中的Request，但您稍后会了解这一点。如果需要，闭包flatten(on:)，将返回集合作为参数。
4. 返回**201 Created**状态。

flatten(on:)等待所有futures返回，因为它们是由同一个Worker异步执行的。

多种 futures

有时候，你需要等待一些不相互依赖的不同类型的futures。例如，在解码请求数据并从数据库中获取URL中指定的用户时，您可能会遇到这种情况。 Vapor提供了许多全局便利方法，可以等待多达五种不同的futures。这有助于避免深层嵌套代码或混淆链。

如果您有两个futures - 从数据库中获取所有用户并从请求中解码一些数据 - 您可以使用flatMap(to:_:_)，如下所示：

```
// 1
flatMap(
  to: HTTPStatus.self,
  getAllUsers(),
  // 2
  request.content.decode(UserData.self)) { allUsers, userData in
    // 3
    return allUsers[0]
      .addData(userData)
      .transform(to: .noContent)
}
```

这是它的作用：

1. 使用全局flatMap(to:_:_ :)等待两个futures完成。
2. 闭包用两个futures的解析结果作为参数。
3. 调用addData(_:)，返回某个future的结果并将返回转换为.noContent。

如果闭包返回非future结果，则可以使用全局map(to:_:_:)代替：

```
// 1
map(
  to: HTTPStatus.self,
  database.getAllUsers(),
  // 2
  request.content.decode(UserData.self)) { allUsers, userData in
    // 3
    allUsers[0].syncAddData(userData)
    // 4
    return .noContent
}
```

这是它的作用：

1. 使用全局map(to:_:_:)等待两个futures完成。
2. 闭包用两个futures的解析结果作为参数。
3. 调用同步 `syncAddData(_:)`
4. 返回 `.noContent`.

创建 futures

有时你需要创造自己的futures。如果if语句返回非future，而else块返回Future，编译器将抱怨这些类型必须是同一类型。要解决此问题，您必须使用request.future(_:)将非future转换为Future。例如：

```
// 1
func createTrackingSession(for request: Request)
    -> Future<TrackingSession> {
  return request.makeNewSession()
}

// 2
func getTrackingSession(for request: Request)
    -> Future<TrackingSession> {
  // 3
  let session: TrackingSession? =
    TrackingSession(id: request.getKey())
  // 4
  guard let createdSession = session else {
    return createTrackingSession(for: request)
  }
  // 5
  return request.future(createdSession)
}
```

这是它的作用：

1. 定义一个从请求中创建TrackingSession的函数。这将返回Future <TrackingSession>。

2. 定义一个从请求中获取tracking session的函数。

3. 尝试使用请求的密钥创建tracking session。如果无法创建tracking session，则返回nil。

4. 确保会话已成功创建，否则创建新的tracking session。

5. 使用request.future(_:)从createdSession创建Future <TrackingSession>。这将返回请求使用的同一个Worker的future。

由于createTrackingSession(for:)返回Future<TrackingSession>，您必须使用request.future(_:)将createdSession转换为Future<TrackingSession>以使编译器满意。

错误处理

Vapor在整个框架中大量使用Swift的错误处理。许多函数抛出，允许您处理不同级别的错误。您可以选择处理路由处理程序中的错误，或者使用中间件来捕获更高级别的错误，或两者兼而有之。

但是，在异步世界中处理错误有点不同。你不能使用Swift的do/catch，因为你不知道什么时候会执行promise。Vapor提供了许多功能来帮助处理这些情况。在基本层面上，Vapor有自己的do/catch回调函数与Futures一起使用：

```
let futureResult = user.save(on: req)
futureResult.do { user in
  print("User was saved")
}.catch { error in
  print("There was an error saving the user: \(error)")
}
```

如果save(on:)成功，则do块将以future的已解析值作为其参数执行。如果future失败，它将执行.catch块，传入错误。

在Vapor中，您必须在处理请求时返回一些内容，即使它是future。使用上面的do / catch方法不会停止发生错误，但它会让你看到错误是什么。如果save(on :)失败并返回futureResult，则失败仍会传播到链中。但是，在大多数情况下，您希望尝试纠正此问题。

Vapor提供了catchMap(_ :)和catchFlatMap(_ :)来处理这种类型的失败。这允许您处理错误并修复它或抛出不同的错误。例如：

```
// 1
return user.save(on: req).catchMap { error -> User in
  // 2
  print("Error saving the user: \(error)")
  // 3
  return User(name: "Default User")
}
```

这是它的作用：

1. 尝试保存用户。如果出现错误，提供catchMap(_:)来处理错误。闭包将错误作为参数，并且必须返回已解析的future的类型 - 在本例中为User。

2. 记录收到的错误。

3. 创建要返回的默认用户。

当关联闭包返回future时，Vapor还提供相关的catchFlatMap(_:)：

```
return user.save(on: req)
  .catchFlatMap { error -> Future<User> in
    print("Error saving the user: \(error)")
    return User(name: "Default User").save(on: req)
}
```

由于save(on:)返回一个future，你必须调用catchFlatMap(_:)代替。

catchMap和catchFlatMap只在失败时执行它们的闭包。但是如果你既想要处理错误并又想处理成功的情况呢？简单！只需链接到适当的方法！

链接 futures

处理futures有时看起来势不可挡。很容易得到嵌套多层深度的代码。

Vapor允许您将futures链接在一起而不是嵌套它们。例如，考虑一个如下所示的代码段：

```
return database
  .getAllUsers()
  .flatMap(to: HTTPStatus.self) { users in
    let user = users[0]
    user.name = "Bob"
    return user.save(on: req)
      .map(to: HTTPStatus.self) { user in
        return .noContent
  }
}
```

map(to:)和flatMap(to:)可以链接在一起以避免嵌套，如下所示：

```
return database
  .getAllUsers()
  // 1
  .flatMap(to: User.self) { users in
    let user = users[0]
    user.name = "Bob"
    return user.save(on: req)
  // 2
  }.map(to: HTTPStatus.self) { user in
    return .noContent
  }
```

更改flatMap(to:)的返回类型允许您链接map(to:)，它接收Future <User>。最终的map(to:)然后返回你最初返回的类型。链接futures允许您减少代码中的嵌套并且可以使其更容易推理，这在异步世界中尤其有用。但是，无论是嵌套还是链，都完全是个人偏好。

Always

有时你想要执行一些事情，无论future的结果如何。您可能需要关闭连接，触发通知或只记录future已执行。为此，请使用always回调。

例如：

```
// 1
let userResult: Future<User> = user.save(on: req)
// 2
```

```
userResult.always {
  // 3
  print("User save has been attempted")
}
```

这是它的作用：

1. 保存用户并将结果设置为userResult。这是Future <User>类型。
2. 链接always到结果。
3. 应用程序执行future时打印字符串。

不管future结果如何，无论是失败还是成功，always闭包都会被执行。它对future也没有影响。您也可以将它与其他链相结合。

Waiting

在某些情况下，您可能希望确切等待结果返回。为此，请使用wait()。

> 注意：严重警告：你不能在主事件循环上使用wait()，这意味着所有请求处理程序和大多数其他情况。

但是，正如您将在第11章“Testing”中看到的，这在测试中尤其有用，因为编写异步测试很困难。例如：

```
let savedUser = try user.save(on: connection).wait()
```

savedUser不是Future<User>，因为您使用wait()，savedUser是User对象。请注意， 如果执行promise失败，则wait()会抛出错误。值得一提的是：这只能用于主事件循环之外！

SwiftNIO

Vapor 3构建于Apple的 SwiftNIO库之上。 SwiftNIO是一个跨平台的异步网络库，就像Java的Netty一样。它是开源的，就像Swift本身一样！

SwiftNIO处理Vapor的所有HTTP通信。这是允许Vapor接收请求并发送响应的管道。SwiftNIO管理数据的连接和传输。

它还管理您的futures的所有**EventLoops**，执行work并执行您的promises。每个EventLoop都有自己的线程。

Vapor管理与NIO的所有交互，并提供干净的Swifty API供使用。 Vapor负责服务器的更高级别方面，例如路由请求。它提供了构建出色的服务器端Swift应用程序的功能。SwiftNIO为构建提供了坚实的基础。

然后去哪儿?

虽然没有必要了解Futures和EventLoops如何在幕后工作的所有细节，但您可以在Vapor's API 文档 or SwiftNIO's API 文档中找到更多信息。Vapor的文档站点还有一个关于异步和futures的 大章节。