# Chapter 27: Caching

By Tanner Nelson

无论您是在创建JSON API，构建iOS应用程序，还是设计CPU的电路系统，您最终都需要一个缓存。缓存（发音为cashes）是一种加速缓慢进程的方法，如果没有它们，互联网将是一个非常缓慢的地方。缓存背后的理念很简单：存储缓慢进程的结果，因此您只需运行一次。构建Web应用程序时可能遇到的一些缓慢进程的示例如下：

- 大型数据库查询。
- 对外部服务的请求，例如其他API。
- 复杂计算，例如，解析大文档。

通过缓存这些缓慢进程的结果，您可以使您的应用程序感觉更快，响应更快。

## Cache存储

作为DatabaseKit的一部分，Vapor定义了KeyedCache协议。该协议为不同的缓存存储方法创建了一个通用接口。协议本身很简单；看一看：

```
public protocol KeyedCache {
  // 1
  func get<D>(_ key: String, as decodable: D.Type)
    -> Future<D?> where D: Decodable

  // 2
  func set<E>(_ key: String, to encodable: E)
    -> Future<Void> where E: Encodable
```

```
    // 3
    func remove(_ key: String) -> Future<Void>
}
```

以下是每种方法的作用：

1. get(_:as:)从缓存中获取给定键的存储数据。如果没有数据，则返回nil。
2. set(_:to:)将数据存储在提供的键的缓存中。如果之前存在某个值，则将其替换。
3. remove(_:)从提供的键缓存中删除数据（如果有）。

每个方法都返回一个Future，因为与缓存的交互可能是异步发生的。

现在您已经了解了缓存和KeyedCache协议的概念，现在是时候看看Vapor可用的一些实际缓存实现了。

## In-memory caches

Vapor附带两个基于内存的缓存：MemoryKeyedCache和DictionaryKeyedCache。这些缓存将其数据存储在程序的运行内存中。这使得这两个缓存都非常适合开发和测试，因为它们没有外部依赖性。但是，它们可能不适合所有用途，因为在应用程序重新启动时会清除存储，并且无法在应用程序的多个实例之间共享。但最有可能的是，这种内存波动不会影响经过深思熟虑的缓存设计。

MemoryKeyedCache和DictionaryKeyedCache之间的区别很细微，但很重要。下面是更深入的介绍。

### Memory cache

MemoryKeyedCache的内容在所有应用程序的事件循环中共享。这意味着一旦某些内容存储在缓存中，以后的所有请求都将看到相同的内容，无论它们分配给哪个事件循环。这非常适合测试和开发，因为它模拟了外部缓存的运作方式。但是，MemoryKeyedCache存储仍然是进程本地的，这意味着在水平扩展时，它不能在应用程序的多个实例之间共享。

更重要的是，此缓存的实现不是线程安全的，因此需要同步访问。这使得MemoryKeyedCache不适合在生产系统中使用。

### Dictionary cache

DictionaryKeyedCache的内容是每个事件循环的本地内容。这意味着分配给不同事件循环的后续请求可能会看到不同的缓存数据。应用程序的单独实例也可以缓存不同的数据。对于纯粹基于性能的缓存来说，这种行为是很好的，例如缓存慢查询的结果，其中逻辑不依赖于正在同步的缓存存储。但是，对于像session存储这样必须同步缓存数据的用途，DictionaryKeyedCache将不起作用。

由于DictionaryKeyedCache不在事件循环之间共享内存，因此它适用于生产系 统。

## 数据库缓存

所有基于DatabaseKit的缓存都支持使用已配置的数据库作为缓存存储。这包括所有Vapor的Fluent映射（FluentPostgreSQL，FluentMySQL等）和数据库驱动程序（PostgreSQL，MySQL，Redis等）。

如果希望缓存数据在重新启动之间保持不变并且可以在应用程序的多个实例之间共享，则将其存储在数据库中是一个很好的选择。如果您已经为应用程序配置了数据库，则可以轻松进行设置。

您可以使用应用程序的主数据库进行缓存，也可以使用单独的专用数据库。例如，将Redis数据库用于缓存是很常见的。

### Redis

Redis是一种开源的缓存存储服务。它通常用作Web应用程序的缓存数据库，并受大多数部署服务（如Vapor Cloud，Heroku等）的支持。 Redis数据库通常非常容易配置，它们允许您在应用程序重新启动之间保留缓存数据，并在应用程序的多个实例之间共享缓存。 Redis是内存缓存的一个伟大，快速且功能丰富的替代品，它只需要更多的工作来配置。

现在您已了解Vapor中可用的缓存实现，现在是时候将缓存添加到应用程序了。

# Example: Pokédex(精灵宝可梦)

构建Web应用程序时，向其他API发出请求可能会导致延迟。如果您正在与之通信的API速度很慢，则可能会使您的API感觉变慢。此外，外部API可能会对您在给定时间段内可以对其发出的请求数量实施速率限制。幸运的是，通过缓存，您可以在本地存储这些外部API查询的结果，并使您的API感觉更快。

您将使用缓存来提高“Pokédex”的性能，这是一个用于存储和列出您捕获的所有神奇宝贝的API。

您已经学习了如何创建基本的CRUD API以及如何创建外部HTTP请求。因此，本章的入门项目已经实现了基础知识。在Terminal中，切换到starter项目的目录并使用以下命令生成并打开一个Xcode项目：

```
vapor xcode -y
```

## 概述

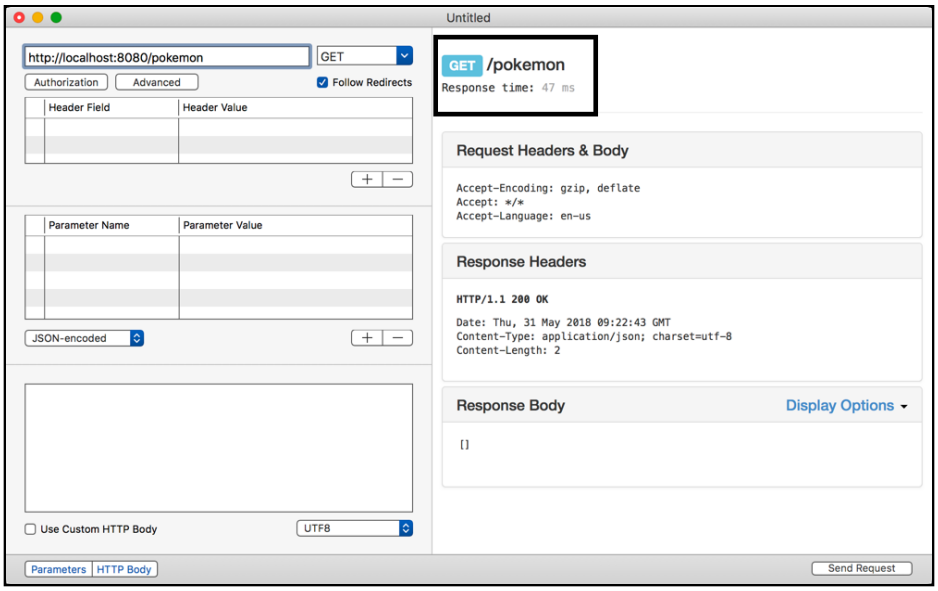
这个简单的Pokédex API有两个路由：

- **GET /pokemon**：返回所有捕获的神奇宝贝的列表。
- **POST /pokemon**：在Pokédex中存储一个被捕获的神奇宝贝。

当您存储新的神奇宝贝时，Pokédex API会调用外部API pokeapi.co来验证您输入的神奇宝贝名称是否真实。虽然这个验证可以正常工作，但pokeapi.co API响应速度非常慢，从而使您的应用感觉变慢。

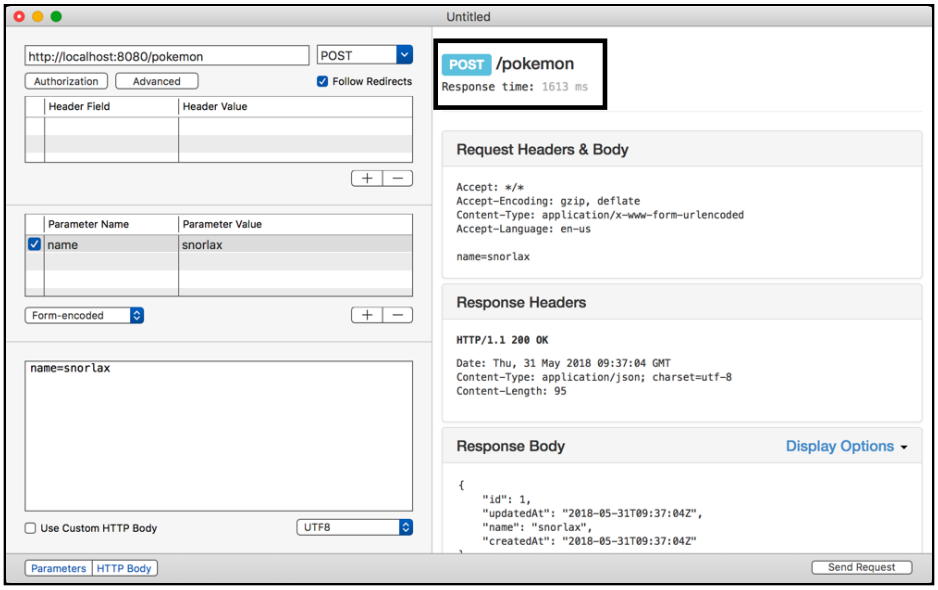
### Normal request

在本地工作时，典型的Vapor请求只需几毫秒即可响应。在随后的屏幕截图中，您可以看到**GET /pokemon**路由的总响应时间约为40毫秒。




## PokeAPI dependent request

在下面的屏幕截图中，您可以看到**POST /pokemon**路由慢25倍，大约1,500ms。这是因为pokeapi.co API对查询的响应速度很慢引起的。




现在，您已准备好查看代码，以便更好地了解导致此路由变慢的原因，以及如何用缓存来解决这个问题。

## 验证名称

在Xcode中，打开**PokeAPI.swift**并查看verifyName(_:on:)。

这个类是HTTP客户端的简单包装类，可以更方便地查询PokeAPI。它验证了提供的神奇宝贝名称的合法性。如果名称是真实的，则该方法返回true，包装在Future中。

现在看一下fetchPokemon(named:)。此方法将请求发送到外部pokeapi.co接口并返回神奇宝贝的数据。如果提供的神奇宝贝名称不存在，则API（此方法）将返回**404 Not Found**响应。

fetchPokemon(named:)是**POST /pokemon**路由响应时间慢的原因。 KeyedCache正是正式解决问题的办法！

## 创建KeyedCache

第一项任务是为PokeAPI包装类创建KeyedCache。在**PokeAPI.swift**中，在let client: Client下面添加一个新属性来存储缓存：

```
let cache: KeyedCache
```

接下来，替换init实现以支持新属性：

```
public init(client: Client, cache: KeyedCache) {
  self.client = client
  self.cache = cache
}
```

最后，解决剩余的编译器错误，通过将makeService(for:)中的return语句替换为：

```
return try PokeAPI(
  client: container.make(),
  cache: container.make())
```

构建并运行，然后在RESTed中创建一个新请求。配置请求如下：

- **URL**: http://localhost:8080/pokemon
- **method**: POST
- **Parameter encoding**: JSON-encoded

添加一个名称和值参数：

- **name**: Test

您将看到以下错误：

```
[ ERROR ] ServiceError.ambiguity: Please choose which KeyedCache you
prefer, multiple are available: MemoryKeyedCache, DictionaryKeyedCache,
DatabaseKeyedCache<ConfiguredDatabase<SQLiteDatabase>>. (Config.swift:72)
[ DEBUG ] Suggested fixes for ServiceError.ambiguity:
`config.prefer(MemoryKeyedCache.self, for: KeyedCache.self)`.
`config.prefer(DictionaryKeyedCache.self, for: KeyedCache.self)`.
`config.prefer(DatabaseKeyedCache<ConfiguredDatabase<SQLiteDatabase>>.sel
f, for: KeyedCache.self)`. (Logger+LogError.swift:20)
```

这可能一开始看起来令人生畏，但不要担心，这是预料之中的。由于此应用程序配置为使用FluentSQLite作为其数据库，因此可以使用多个KeyedCache实现。由于已配置Fluent，因此您将使用SQLiteCache (DatabaseKeyedCache <ConfiguredDatabase <SQLiteDatabase >>)。

打开**configure.swift**，在return migrations之前并添加以下行：

```
migrations.prepareCache(for: .sqlite)
```

正如您必须运行migration以在数据库中设置模型一样，您必须允许Fluent配置用于存储缓存数据的基础数据库模式。

接下来，在configure(_:_:_:)的末尾添加以下内容：

```
config.prefer(SQLiteCache.self, for: KeyedCache.self)
```

这告诉Vapor使用SQLite作为应用程序的KeyedCache。这解决了歧义错误。

> 注意：Fluent使用**fluentcaches**表来存储缓存数据。

构建并运行。使用RESTed将相同的请求发送到**POST /pokemon**。现在在**Response Body**中您将看到以下内容：

```
{
    "error": true,
    "reason": "Invalid Pokemon Test."
}
```

太好了！您已经创建了KeyedCache。是时候把它付诸实践了。

## 获取和存储

现在PokeAPI包装类可以访问工作中的KeyedCache，您可以使用缓存来存储pokeapi.co API的响应，以后就可以更快地获取它们了。

打开**PokeAPI.swift**并用以下代码替换verifyName(_:on:)的实现：

```
public func verifyName(_ name: String, on worker: Worker)
    -> Future<Bool> {
  // 1
  let key = name.lowercased()
  // 2
  return cache.get(key, as: Bool.self).flatMap { result in
    if let exists = result {
      // 3
      return worker.eventLoop.newSucceededFuture(result: exists)
    }

    // 4
    return self.fetchPokemon(named: name).flatMap { res in
      switch res.http.status.code {
      case 200..<300:
        // 5
        return self.cache.set(key, to: true).transform(to: true)
      case 404:
        return self.cache.set(key, to: false)
          .transform(to: false)
      default:
        let reason =
          "Unexpected PokeAPI response: \(res.http.status)"
        throw Abort(.internalServerError, reason: reason)
      }
    }
  }
}
```

1. 通过小写名称来创建一致的缓存键。这确保了“Pikachu”和“pikachu”共享相同的缓存结果。

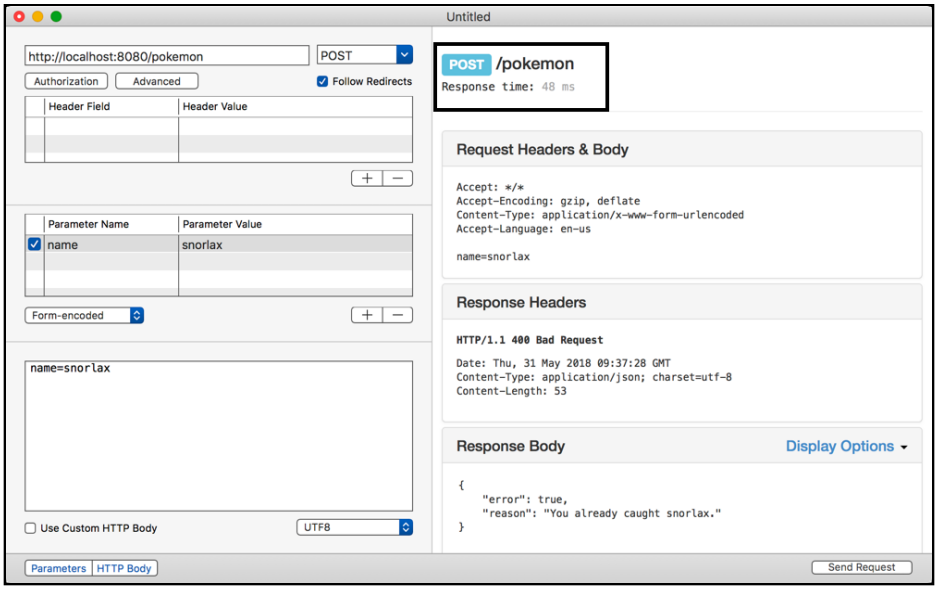
2. 查询缓存以查看它是否包含所需的结果。

3. 如果存在缓存结果，则返回该结果。这意味着对verifyName(_:on:)的调用将永远不会对给定名称再次调用fetchPokemon(named:)。这是提高性能的关键步骤。

4. 当fetchPokemon(named:)完成时，将API查询的结果存储在缓存中。

构建并运行。

再次使用RESTed将相同的请求发送到**POST /pokemon**。记下第一个请求的响应时间。这可能是几秒钟。现在，发送第二个请求并记下时间；它应该快了得多！




# 然后去哪儿?

缓存是计算机科学中的一个重要概念，了解如何使用它将有助于使您的Web应用程序感觉快速响应。有几种方法可以存储Web应用程序的缓存数据：内存，Fluent数据库，Redis等。每个都有明显不同的好处。

您可以检出可用于缓存的不同类型的算法，例如最近最少使用（LRU），随机替换（RR）或后进先出（LIFO）。根据您正在编写的应用程序类型以及您在其中缓存的数据类型，每种方法都有优缺点。

在本章中，您学习了如何配置Fluent数据库缓存。使用缓存保存外部API请求的结果，可以显着提高应用的响应速度。

如果您想要挑战，请尝试将您的应用配置为使用Redis或内存缓存而不是SQLiteCache。但请记住，你必须全部缓存它们！