audreywelch / ios-astronomy
forked from LambdaSchool/ios-astronomy

No description, website, or topics provided. Edit

Manage topics

4 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is even with LambdaSchool:master. Pull request Compare

armadsen Move video to YouTube Latest commit 7d1cb5e on Sep 6, 2018

Astronomy.xcodeproj	Add README	5 months ago
Astronomy	Add README	5 months ago
.gitignore	Add project start	5 months ago
README.md	Move video to YouTube	5 months ago

README.md

Astronomy

For this project, you'll build an app that uses NASA's Mars Rover API to display images taken by the rovers on Mars.

This project will help you practice the concepts learned in the Concurrency III - iOS Concurrency module of Sprint 7. After completing the lesson material and this project, you should be able to:

- understand and explain the difference between a GCD queue and a thread
- use GCD to implement thread safe properties in Swift
- implement a custom subclass of NSOperation
- cancel an operation
- set up dependencies between operations

Watch this video to see what the finished app should do.

Part 0 - Fork and Clone Project Start

This is a large project with a lot of pieces. In order to allow you to focus on the concurrency concepts taught today, we're providing you with a project start. The project start includes networking and some UI code to get you started.

1. Fork this repository to your own GitHub account.
2. Clone your fork to your local machine.
3. Open Astronomy.xcodeproj

Part 1 - Implement Basic Collection View

A collection view and associated view controller have already been set up for you, and data is already being loaded from the API. However, loading images from the Mars Rover API is a multi-step process:

1. Load mission manifest (`MarsRover`) for a given Rover. (Done for you already)
2. Load photos metadata (`MarsPhotoReference` s) for a particular solar day. (Done for you)
3. For each photo reference, load the actual image data using the photo reference's `imageURL` property. (**You need to do this.**)

Fill out `PhotosCollectionViewController.loadImage(forCell:, forItemAt:)` . For now, just use `URLSession` directly.

1. Get the `MarsPhotoReference` instance for the passed in `indexPath` from the `photoReferences` array property.
2. Get the URL for the associated image using the `imageURL` property. Use `.usingHTTPS` (provided in `URL+Secure.swift`) to make sure the URL is an `https` URL. By default, the API returns `http` URLs.
3. Create and run a data task to load the image data from the `imageURL` .
4. In the data task's completion handler, check for an error, and return early if there is one. Otherwise, create a `UIImage` from the received data.
5. **Important:** Check to see if the current index path for `cell` is the same one you were asked to load. If not, this means that that item has scrolled off screen and the `UICollectionViewCell` instance has been reused for a different index path. If this happens, abort setting the image.
6. If the cell hasn't been reused, set its `imageView` 's image to the `UIImage` you just created.
7. Make sure you do all UIKit API calls on the main queue.

Test your app. It should load images and display them.

Note: The API itself can be somewhat slow. If you see a white screen, wait a while for data to load before assuming something is broken. You can change the solar day (sol) being loaded in the `roverInfo.didSet` property observer in `PhotosCollectionViewController` if you want. Some sols have many fewer images than others, causing the API to respond more quickly depending on the sol you pick. I've found sol 105 to be a good choice for testing where you don't need a lot of images. For the next section, however, you'll want to load as many images as you can. Sol 100 is a good choice.

Part 2 - Test Scrolling Performance

Your app should work at this point, however, you should evaulate its performance, especially with a slow network. Apple provides a tool called Network Link Conditioner that you can install on your Mac and use to simulate a slow or unreliable network for the purposes of testing your apps. You can download it (along with a number of other supplemental Apple developer tools) here: https://download.developer.apple.com/Developer_Tools/Additional_Tools_for_Xcode_9.3/Additional_Tools_for_Xcode_9.3.dmg . You can also read a good article on the Network Link Conditioner on NSHipster.

Install Network Link Conditioner, enable it, and select the DSL setting. This is a good choice for testing with a working but relatively slow network connection.

Test your app:

1. Run the app and wait for the first page of images to load. Note that they may load much more slowly than before.
2. Scroll as quickly as you can to the bottom of the list of images. Is the scrolling smooth or choppy?
3. When you get to the bottom, are the bottommost images loaded? Do they load relatively quickly?
4. If you quickly scroll back to the top, are the images at the top still there?

You will probably notice that the answers to these questions indicate poor performance. Images get loaded on demand as cells scroll on screen, and if a cell subsequently scrolls off screen, loading for it continues, preventing later images from loading until after now-offscreen images finish. Also, if you scroll already-loaded images off screen, then go back to them, you may notice that they've disappeared and have to load all over again.

All of these problems add up to provide a poor user experience, as well as overuse of the network and system resources.

Disable network link conditioner when you're done with it! It will slow all network activity on your Mac.

Part 3 - Implement Caching

You can aleviate problem #4 by implementing a caching system. Instead of reloading the same images over and over again every time they're requested, you can store them in a cache after loading them the first time. Then, on subsequent requests for a given image, you check the cache, and if the image is already there you can avoid doing another network request.

Implement a simple cache in your app:

1. Create a new file called `Cache.swift`.
2. Create a class called `Cache`. It should be generic with respect to the type it stores, and its keys. e.g. `Cache<Key, Value>`.
3. The generic `Key` type will need to be constrained to conform to `Hashable`.
4. Create a private property that is a dictionary to be used to actually store the cached items. The type of the dictionary should be `[Key : Value]. Make sure you initialize it with an empty dictionary.
5. Implement `cache(value:, for:)` and `value(for:)` methods to add items to the cache and remove them, respectively.
6. Add a `cache` proeprty to `PhotosCollectionViewController`. Its keys should be `Int`s as you'll use `MarsPhotoReference` `id`s for the keys. Its values should be `Data` objects, as you'll be caching image data. (You could also cache `UIImage`s directly.)
7. In your `PhotosCollectionViewController.loadImage(forCell:, forItemAt:)` method, before starting a data task, first check to see if the cache already contains data for the given photo reference's id. If it exists, set the cell's image immediately without doing a network request.
8. In your network request completion handler, save the just-received image data to the cache so it is available later.

Test for Thread Safety

Xcode includes a feature to help you find common threading problems, most notably race conditions. It's called the thread sanitizer, and you should enable it and run your app. To enable it, hold down the option key while clicking Xcode's run button. In the sheet that appears, select the Diagnostics tab, then check the box for "Thread Sanitizer" and the associated "Pause on Issues". This will enable additional checking and will automatically stop in the debugger if Xcode finds a race condition (e.g. unprotected access to a shared resource from multiple threads) while your app is running.

Run your app with the thread sanitizer enabled, wait for images to load, and scroll back and forth. Does the thread sanitizer catch any problems? What seems to be the issue?

Make `Cache` Thread-safe.

The problem is that your cache class is not thread-safe. Fundamentally it's a thin wrapper for a dictionary, and dictionaries themselves are not thread-safe. That means you can't use them as a resource shared between threads without implementing some kind of locking to prevent multiple threads from accessing them simultaneously.

You can make `Cache` thread-safe, that is safe to use from multiple threads simultaneously, by making sure that *it* ensures that all access to its underlying dictionary is done from one thread at a time.

You could use an instance of `NSLock` for this. However, Grand Central Dispatch serial queues are designed to only allow one closure or unit of code to run at a time. They make for a great way to implement synchronized access to a shared resource without using a lock.

1. Create a private `queue` property and initialize it with a serial `DispatchQueue`. Give it an appropriate label.
2. In `cache(value:, for:)`, dispatch the actual setting of the dictionary key/value pair so that it occurs on the `queue`.
3. In `value(for:)`, use a *synchronous* dispatch to retrieve the requested value from the dictionary before returning it. Note that `DispatchQueue.sync()`'s closure can return a value which will subsequently be returned from `DispatchQueue.sync()` itself. This means you don't need to create a temporary variable outside the dispatched closure.
4. Run your app again. Verify that thread sanitizer no longer flags any issues.

Part 4 - Implement Image Fetch Cancellation

The other major problem with your current implementation is that as you quickly scroll, every cell that scrolls on screen, no matter how briefly, triggers a network request. Even if a cell scrolls off screen, its network request continues, preventing later cells' images from being loaded as quickly as possible. This also wastes CPU time, battery power, and network bandwidth loading images that the user may not actually look at.

To fix this, you'll want to implement cancellation of in-flight network requests if they're no longer needed. This can be done directly with GCD, but `Operation` / `OperationQueue` are a natural fit for this kind of job.

Implement PhotoFetchOperation

1. Create a subclass of `ConcurrentOperation` * (provided for you in the project start) called `FetchPhotoOperation`.
2. Add properties to store an instance of `MarsPhotoReference` for which an image should be loaded as well as `imageData` that has been loaded. `imageData` should be optional since it won't be set until after data has been loaded from the network.
3. Implement an initializer that takes a `MarsPhotoReference`.
4. Override `start()`. You should begin by setting `state` to `.isExecuting`. This tells the operation queue machinery that the operation has started running.
5. Create a data task to load the image. You should store the task itself in a private property so you can cancel it if need be.
6. In the data task's completion handler, check for an error and bail out if one occurs. Otherwise, set `imageData` with the received data.
7. Make sure you set `state` to `.isFinished` before exiting the completion closure. This is a good use case for `defer`.
8. Override `cancel()`, which will be called if the operation is cancelled. In your implementation, call `cancel()` on the `dataTask`.

* `ConcurrentOperation` is a nice "boilerplate" class that makes it easier to implement concurrent/asynchonous `Operation` subclasses in Swift. Feel free to keep it in your personal code library so you can use it in other apps you write.

Update `PhotosCollectionViewController` to Use Operations

Now that you've created an operation to fetch images, you should rewrite `PhotosCollectionViewController` so that it uses operations to do image loading.

1. Add a private property called `photoFetchQueue`, which is an instance of `OperationQueue`.
2. In `PhotosCollectionViewController.loadImage(forCell:, forItemAt:)`, delete the code that creates a data task.
3. Create three operations:
    - One should be a `PhotoFetchOperation` to fetch the image data from the network.
    - One should be used to store received data in the cache.
    - The last should check if the cell has been reused, and if not, set its image view's image.
      The last two of these can be instances of `BlockOperation`.
4. Make the cache and completion operations both depend on completion of the fetch operation.
5. Add each operation to the appropriate queue. Note that the last operation above uses UIKit API and must run on the main queue.
6. Add a dictionary property that you'll use to store fetch operations by the associated photo reference id.
7. When you finish creating and starting the operations for a cell, add the fetch operation to your dictionary. This way you can retrieve it later to cancel it if need be.

Test

Your app should behave as it did before, loading images and scrolling. Because you haven't yet implemented cancellation, the problem with lower-down images loading slowly is *not* yet fixed.

Implement Cancellation

The last piece of the puzzle is to implement cancellation of operations that are in-flight but not yet complete when the associated cell scrolls ofscreen.

1. Implement the `UICollectionViewDelegate` method `collectionView(_:, didEndDisplaying:, forItemAt:)`, which is called when a given item scrolls off screen.
2. In your implementation, get the associated fetch operation and cancel it.

Test One Last Time

At this point, your application should perform well even on a (simulated) slow network. You should be able to scroll all the way to the bottom of a large page of images and see the bottom-most images load relatively quickly without waiting for all previous images to load first. You can add print statements to verify that operations are being cancelled as they should.

Go Farther

If you finish with time to spare or would like to push yourself, try the following bonus goals:

- Implement UI to allow the user to move between sols
- Implement UI to allow the user to see all available rovers and select them
- Modify your `Cache` class so that it can be configured to evict old entries if its size gets too big. This prevents the app from using too much memory due to a lot of cached data.
- Investigate and implement the use of the system-provided `URLCache` API for caching instead of your own cache class. What benefit does this provide? Downsides?