ALL ABOUT SWIFT (/)

Jun 4

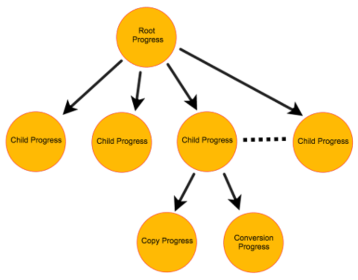
Working with NSProgress

During my research for my last week's blog about CoreData's asynchronous API, I stumbled upon *NSProgress*. I was kind of surprised the way NSProgress was used and how the asynchronous CoreData requests got aware of it since there was no explicit handover. So I decided to take a closer look at *NSProgress* for this week's blog post.

If you haven't read my last week's blog (http://www.allaboutswift.com/dev/2016/5/29/asynchronous-core-data-requests) (which you should right after you finished this one) you might ask yourself what *NSProgress* is about. Well, *NSProgress* is a class that had been introduced with iOS7 to assist in tracking progress of any kind. For this purpose, it's got three properties

- *totalUnitCount*: number of total units
- *completedUnitCount*: number of completed units.
  The operation is complete if *completedUnitCount* equals *totalUnitCount*
- *fractionCompleted*:
  the progress completed in percent ranging from 0 to 1 with 1 being 100%

A key feature of *NSProgress* is that it can have one or more child instances of *NSProgress*. Consider for example that you want to to copy a folder from location A to B. For the sake of this example let's assume that the folder has 10 images and that each image will be converted to a PNG once it's been copied. So you have one operation which consists of 10 sub operations which can be further split up into a copy operation and conversion operation. The resulting progress - tree can be seen in Picture 1 below.



Picture 1 NSProgress tree

There is one *NSProgress* with a total unit count of n which has has n child instances with a total unit count of 1 which again have two child instances with a total unit count of 1. The interesting part of it is that you can charge each child progress with a certain amount of units of the total progress. While the operation is progressing, the completed unit count of each child progress is propagated up to its parent progress which results in the root progress having the total progress of the whole operation at each given moment.

There are basically two ways to add a child progress, explicitly and implicitly, about which I want to talk about in the following two sections.

Explicitly adding a child progress

Explicitly adding a child is in my opinion the better approach. It is the more description variant since it tells any reader of your code what is going on. Let's take a look at some sample code:

```
func doMainTask(parentProgresss :
NSProgress,pendingUnitCount : Int64) {
   let childProgress = NSProgress(totalUnitCount:
Int64(100))
   parentProgresss.addChild(childProgress,
withPendingUnitCount: pendingUnitCount)
   for task in 1...100 {
      doSubTaskTask(childProgress,unitCount: Int64(task))
{
         childProgress.completedUnitCount = Int64(task)
      }
   }
}
```

```
let progress = NSProgress(totalUnitCount: 1000)
doMainTask(progress,pendingUnitCount: 200)
doMainTask(progress,pendingUnitCount: 200)
doMainTask(progress,pendingUnitCount: 100)
doMainTask(progress,pendingUnitCount: 500)
```

In this example a main *NSProgress* instance is created with a total unit count of 1000, which subsequently is distributed among its four child progresses. Each child progress is created via *NSProgress(totalUnitCount:)* with a total unit count of 100 and charged against the main progress accordingly via *addChild(child:withPendingUnitCount)*. Every time a subtask of a child progress completes, the completed unit counter is incremented.

This approach is straightforward and descriptive. It's main advantage in my opinion however compared to the implicit version is that there is no global variable. More about that later.

The capabability of explicitly adding a child has been made available with iOS9. So you have to fall back to the implicit version if you still need to support older versions of iOS.

Implicitly adding a child progress

You encounter the implicit version when you are working with the asynchronous CoreData API or when you want to track the progress of *NSData* related tasks. The implicit version relies on thread local storage and hence makes the current progress available to each method running in that thread. If you never heard of thread local storage before look at it as address space whose scope is restricted to one thread. Let's look at our previous example which has been adapted to add children to the root progress implicitly.

```
func doMainTask() {
   let childProgress = NSProgress(totalUnitCount:
Int64(subTasks))
   for task in 1...subTasks {
      doSubTaskTask(childProgress,unitCount: Int64(task)) {
         childProgress.completedUnitCount = Int64(task)
      }
   }
}

let progress = NSProgress(totalUnitCount: 1000)

progress.becomeCurrentWithPendingUnitCount(200)
doMainTask()
progress.resignCurrent()

progress.becomeCurrentWithPendingUnitCount(200)
doMainTask()
progress.resignCurrent()

progress.becomeCurrentWithPendingUnitCount(100)
doMainTask()
progress.resignCurrent()

progress.becomeCurrentWithPendingUnitCount(500)
doMainTask()
progress.resignCurrent()
```

First we create an instance of *NSProgress* to track the total progress. Then we add each child progress one by one. Before we can add a child progress we have to make our total progress the current Progress. This is done via *becomeCurrentWithPendingUnitCount(_:)*. The *pendinUnitCount* part of the method determines how many units will be assigned to the child progress. After that the child just needs to be created via *NSProgress(:totalUnitCount:)*. *NSProgress*' initializer checks if there is a current progress and adds the new progresss as one of its children. The child progress setup has to be concluded with a subsequent call to *resignCurrent()*.
Mind that the child progress has to be immediately setup on the main thread (after *becomeCurrentWithPendingUnitCount(_:))*. It has to be done immediately due to *the call to resignCurrent()* and needs to happen on the same thread to enable the child progress to assign itself as a child.

What's the advantage of this approach? Some argue it's a very loosely coupled implementation since the various *NSProgress* instances don't have to be made aware of each other. In my opinion it's a geek's version of how to setup child progresses and hence I don't see any advantage over the explicit version. I think it's the worse version. It's not just that there is more code involved which is not very descriptive since it lacks to communicate intent (*1). You even have to be aware of the implicit contract that is in place here (setup child progress a) immediately b) on the same thread) which makes it a terrible interface to use.

Do you remember what I said above about building a hierarchy of *NSProgress* instances? Well, using the implicit method you have to do this asynchronously. Since there is only one current progress at a time, the newly create progress in *doMainTask()* can only set itself up as the child (which happens in the initializer). If it wants to add other children implicitly, it needs to postpone this e.g. via *dispatch_async()*. If you think that is cumbersome then you are absolutely right.

Since we got this particular feature of *NSProgress* covered let's talk in the remaining two sections about how to monitor progress and how to cancel an operation in progress with NSProgress.

Monitoring progress

Monitoring progress is done via Key Value Observation (KVO). *completedUnitCount* and *fractionCompleted* are both key-value observable. Just setup an observer and handle any updates in observeValueForKeyPath().

```
let progress = NSProgress(totalUnitCount: 1000)
let observer = KeyValueObserver(target: progress, keyPath:
"fractionCompleted") { (keyPath, dict) in
    let value = dict!["new"] as! Double
    progressUI.progress = Float(value)
}
```

KeyValueObserver() is just a block based wrapper for KVO. Whenever a child progress gets updated, the change is propagated to the root progress instance which triggers a KVO call that eventually ends up in the block provided to KeyValueObserver(). Here we extract the current progress and update the UI (a UIProgressView) accordingly.

Cancellation

Cancellation with *NSProgress* is straightforward. If you have ever worked with *NSOperations* then you know how it works. In your child operation you have to check if it is cancelled via the *cancelled* property of its associated progress. Then when the main task gets cancelled via its associated progress' *cancel()* method, you need to act upon it and stop any further work on the current subtask.

```
func doMainTask(progresss :
NSProgress,pendingUnitCount : Int64) {
   let childProgress = NSProgress(totalUnitCount:
Int64(100))
   progress.addChild(childProgress,
withPendingUnitCount: pendingUnitCount)
   for task in 1...100 {
      guard !childProgress.cancelled else {
         break
      }
      doSubTaskTask(childProgress,unitCount: Int64(task))
{
         childProgress.completedUnitCount = Int64(task)
      }
   }
}
```

```
let progress = NSProgress(totalUnitCount: 1000)
```

```
cancelButton.touchHandler = { _ in
     progress.cancel()
}
```

I highlighted the relevant sections in bold. The *cancel()* method in the touch handler sets the *cancelled* flag which then each child working on a subtask acts upon.

Conclusion

If you need to track progress of any kind, there is no need to waste time implementing some custom class to do so. *NSProgress* does a very neat job here. It however has its shortcomings. Let me address a few of those:

1. hardly supported
   NSProgress is already around since iOS7 but Apple's own frameworks (appart from CoreData and NSData) take hardly advantage of it. In the two instances Apple does, you need to use the implicit version which I don't like at all. I am sure if Apple decided to give it more prominence by using it more often, we all ended up paying it some more attention.

2. problematic interface when adding children implicitly
   It's a terrible interface to use and I don't think it has any future. The future is Swift and this is no Swift way of doing things.
3. cumbersome progress observation
   Monitoring progress via KVO is again not the way things are done in Swift . If you take a look at the documentation for *NSProgress* you can see that it has block support. There is a cancellation handler, a resume handler and a pause handler. Why not add a progress handler and a completion handler and get rid of the need to use KVO? I am surprised to see this hasn't happened already. Well, iOS X is around the corner. Let's see ...

Before I conclude this blog, let me tell you that you can find the complete code of the samples I've shown here in a playground on github (https://github.com/fsaar/allAboutSwift/tree/master/NSProgress).

*1 Just look at the explicit version. Any method that exposes a *NSProgress* as a parameter 'parentProgress' gives you an idea of what's going to happen. The implicit version doesn't do this. It's not clear at all that something's going to happen here with the current progress between the two calls of *becomeCurrentWithPendingUnitCount(_:)* and *resignCurrent()*.

frank saar (/?author=56e4898ef699bb97173ad019)

NSProgress (/?tag=NSProgress), CoreData (/?tag=CoreData), NSData (/?tag=NSData)

(HTTPS://WWW.FACEBOOK.COM/SHARER/SHARER.PHP?U=HTTP%3A%2F%2FWWW.ALLABOUTSWIFT.COM%2FDEV%2F2016%2F6%

(HTTPS://TWITTER.COM/INTENT/TWEET?URL=HTTP%3A%2F%2FWWW.ALLABOUTSWIFT.COM%2FDEV%2F2016%2F6%2F4%2FWORK

(HTTPS://PLUS.GOOGLE.COM/SHARE?URL=HTTP%3A%2F%2FWWW.ALLABOUTSWIFT.COM%2FDEV%2F2016%2F6%2F4%2FWORKIN

Jun 12 Working with Time Profiler

(/dev/2016/6/12/qouw296w2sjpt74qea20rp1yiyhbjd)

❯

May 29 Asynchronous Core Data Requests

(/dev/2016/5/29/asynchronous-core-data-requests)