WWDC 2010 Session 207 Run Loops Section

This thread has been locked by a moderator.

55

At WWDC 2010 I gave a two part presentation on iPhone OS (yes, it’s that old) networking:

- WWDC 2010 Session 207 *Network Apps for iPhone OS, Part 1*
- WWDC 2010 Session 208 *Network Apps for iPhone OS, Part 2*

The second part included an in-depth explanation of run loops, one that I like to reference when I help folks with that technology.

Sadly, this presentation is no longer available from Apple. While you may be able to find copies lurking in the darker corners of the Internet, I’d rather not send folks down that path and so I’m posting a transcript of that section, along with some *fabulous* ASCII art.

If it helps, imagine this being spoken, rather hesitatingly, in a mild Australian accent (-:

Oh, and there are a few bits where I’ve changed the text slightly because what I said on stage is a little confusing, especially when rendered as text. Those edits are denoted with ~~strike through~~ for the removal and *italics* for the addition.

Share and Enjoy
—
Quinn “The Eskimo!” @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

17:25

… and we’ll dive straight into run loops.

17:30

```
Run Loop Factoids

* One run loop per thread
* Event dispatch mechanism
* Event sources
  * Each with associated callback
* Run loop must be run explicitly
  * Monitors event sources
  * Call callbacks
* UIKit run main thread's run loop
```

And that’s definitions to start off with.

There’s one run loop per thread, always.

A run loop is a dispatch… an event dispatch mechanism… it monitors a set of event sources, and each event source has a callback associated with it. And when the event source fires, it calls the callback.

Now, the run loop has to be explicitly run by the thread associated with it. So the thread runs the run loop and while it’s running inside the run loop it monitors these event sources and calls the callbacks if they fire. And if no event sources fire, it blocks waiting for one of them to fire.

In general, you must explicitly run your run loops but as one special case the UI frameworks, like UIKit on iPhone OS, will automatically run the main thread’s run loop.

Now to look at this graphically, I have a series of less cheesy diagrams.

18:18

```
Run Loops

Main Thread         (Run Loop)

Secondary Thread    (Run Loop)

Secondary Thread    (Run Loop)

… and so on
```

Here’s the main thread… here’s a bunch of threads, the main thread and a couple of secondary threads, and each of them has an associated run loop, sorta… the run loop sorta owns that thread.

18:30

```
Thread     (Run Loop)
            /   |    \
           /    |     \
          /     |      \
[Source 1] [Source 2] [Source 3]
```

Now if we focus on one of these threads, we can zoom into it, and here we see the run loop, and the run loop is associated with all of the run loop sources. Now these run loop sources aren’t abstract notions; they are related to what you’ve done.

18:44

```
Thread           (Run Loop)
               /     |      \
              /      |       \
             /       |        \
[Timer Source] [Stream Source] [Conn Source]
      |               |              |
  [NSTimer]      [NSStream]     [NSURLConn]
```

So, for example, here on the left, if you start a timer, we create a timer event source that’s attached to the run loop. And on the right there we started an `NSURLConnection` and its created a connection source that’s attached to the run loop. So these run loop sources don’t come out from… you know… just out of thin air, they come because of operations that you’ve done.

18:44

```
Explicit Scheduling

NSInputStream * stream;

[netService getInputStream:&stream outputStream:NULL];
[stream scheduleInRunLoop:[NSRunLoop currentRunLoop]
  forMode:NSDefaultRunLoopMode];
[stream setDelegate:self];
[stream open];



-(void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)e
{
}
```

Here’s an example of actually scheduling something on the run loop. Here’s… we’ve started with a net service, which is a reference to a service that we’ve found on Bonjour. We’ve created an input stream for that service. Now that input stream by itself is not scheduled on the run loop, so we explicitly schedule it on the run loop with this `-scheduleInRunLoop:forMode:` method. You pass in the current run loop and you pass in the run loop mode. Now run loop modes are a source of some confusion, and I’m gonna cover those in detail in a few slides but, for the moment, we’re just going to ignore it and choose the default run loop mode.

Now the other thing you do is that you set the delegate and the delegate is effectively the callback. The real callback is internal to… ah… `NSInputStream` but for the moment… from your perspective the callback is the delegate. And once it’s all set up on the run loop you kick off the open and then the open proceeds asynchronously in the background.

At some point in the future, when the open is complete, you start getting events to your delegate and it’ll call this method, the `-…:handleEvent` method, on your delegate. And the question is “Well, what thread is that running on?” It’s running on the thread associated with the run loop that you passed in when you scheduled the stream on the run loop. Those two are tightly bound together, and this also means that in order for this delegate callback to be called you have to be running this run loop. Now if you’re in the main thread that’s really easy, the UIKit does it for you, but if you’re in other threads you have to go out of your way to make sure that runs.

Now this is explicit scheduling, where you explicitly tell the frameworks what run loop you want to schedule on.

20:44

```
Implicit Scheduling

NSURLConnection * conn;

conn = [NSURLConnection connectionWithRequest:req
  delegate:self];



- (void)connection:(NSURLConnection *)conn
  didReceiveResponse:(NSURLResponse *)resp
{
}
```

In addition to this you get implicit scheduling, here’s an example of this, where the frameworks sorta… decide for themselves what to schedule on. This is `NSURLConnection`… has a utility called `-connectionWithRequest:delegate:` and that automatically schedules on the current run loop in the default mode. And that’s the context that this callback will run in.

Every time we have an implicit scheduling we almost always have an equivalent method that’s explicit.

21:06

```
Making It Explicit

NSURLConnection * conn;

conn = [[NSURLConnection alloc] initWithRequest:req
  delegate:self
  startImmediately:NO];
[conn scheduleInRunLoop:[NSRunLoop currentRunLoop]
  forMode:NSDefaultRunLoopMode];
[conn start];



- (void)connection:(NSURLConnection *)conn
  didReceiveResponse:(NSURLResponse *)resp
{
}
```

So here’s the explicit version. You allocate a connection with the request and the callback, which is the delegate, and you pass `NO` to this `startImmediately` parameter. So it doesn’t start, it doesn’t schedule on the run loop automatically. And in the next step you schedule it on the run loop that you want to schedule it on, and then you call the `-start` method, and from then on, at some point in the future, you’ll get this… ur… the delegate callbacks associated with this operation.

So that’s pretty much how you schedule things on the run loops. What about these run loop modes?

21:38

```
Run Loop Modes

* Event sources added in a mode
* Run loop runs in a mode
  * Monitors event sources in that mode
  * Other event sources ignored
```

Whenever you add an event source to a run loop mode… to a run loop, you actually add it in a particular mode. And whenever a run loop runs, it always runs in ~~that~~ *a* mode. And when it runs in ~~that~~ *a* mode, and it only runs… it only monitors the event sources associated with that mode. All the other event sources are ignored. So this is just the basic facts.

22:01

```
Run Loop Modes

Thread           (Run Loop)
               /     |      \
              /      |       \
             /       |        \
[Timer Source] [Stream Source] [Conn Source]
      |               |              |
  [NSTimer]      [NSStream]     [NSURLConn]
```

Here it is graphically. Well, not quite yet. This is where we left off our run loop model. I’m going to insert the modes in there.

22:07

```
Run Loop Modes

Thread          (Run Loop)
                 /      \
                /        \
               /          \
        Default            Tracking
      [X] [X] [X]         [ ] [X] [X]

[Timer Source] [Stream Source] [Conn Source]
      |               |              |
  [NSTimer]      [NSStream]     [NSURLConn]
```

So now we have two run loop modes in this blue layer, default run loop mode and a tracking run loop mode. Some of them have all the event loop sources associated with them, the default mode, and the tracking mode only has a subset. So when you run the run loop in the default mode we monitor all of these event sources. In contrast, when you run the run loop in this tracking mode we only monitor a subset of the event sources. In this case we ignore the timer. So if the timer fires the callback won’t be… for that timer won’t be called.

And that’s useful in a variety of circumstances but the real question is “Why do we have this whole run loop mode mess?”

22:44

```
Why Run Loop Modes?

* It’s all about recursion
* Synthetic synchronous
  * To run async APIs sync
  * Schedule in custom mode
  * Run in custom mode
* UI tracking
```

And it’s associated with recursion.

Sometimes you’re in a run loop callback and you want to run the run loop again. You might want to call an API using this synthetic synchronous model; sometimes it’s important to do so. And an example of doing that is the user interface tracking that’s done by UIKit. And I’ll talk about in a little more… in depth in a few slides.

But I just want give an example of this synthetic synchronous model. You’re in a run loop callback and you want to run an async API synthetic synchronous. So the way you do that is that you set up the async call and you schedule it in a custom run loop mode. Run loop modes are just strings, you can pull them out of thin air. And generally we recommend that you use reverse DNS notation, just to keep away from other people’s run loop modes.

So you schedule your event source in this custom run loop mode and you run the run loop in that custom run loop mode and what the means is that only your event source will run. All other event sources are held off until the run loop starts… returns to the other modes. So here’s an example of this.

23:42

```
Run Loop Modes

Thread          (Run Loop)
                 /      \
                /        \
               /          \
        Default            Tracking
      [X] [X] [X]         [ ] [X] [X]

[Timer Source] [Stream Source] [Conn Source]
      |               |              |
  [NSTimer]      [NSStream]     [NSURLConn]
```

Here’s where we left our run loop off, with two run loop modes. What we do is we create a custom mode and we run the run loop in that custom mode.

23:51

```
Run Loop Modes

Thread                (Run Loop)
                 /        |          \
                /         |           \
               /          |            \
    My Mode            Default           Tracking
[X] [ ] [ ] [ ]    [ ] [X] [X] [X]    [ ] [ ] [X] [X]

[FD Source]  [Timer Source] [Stream Source] [Conn Source]
     |              |               |              |
[CFFileDesc]   [NSTimer]      [NSStream]     [NSURLConn]
```

And we add our file descriptor in this mode. In this case we’ve created a `CFFileDescriptor` just as an example. And we add that, its event source, to the run loop in that custom mode. And when we run the run loop in that custom mode, only that event source is looked at; all other event sources are ignored.

It’s really useful. In this case, if you’re doing it on the main thread this might be a bad idea because it’s effectively… if you block forever you’ll be killed by the watchdog. If you’re doing it on a secondary thread it’s perfectly reasonable. You can even do it on the main thread if you can limit the amount of time that you’ll spend… if there’s some upper bound to the amount of time you’ll spend running the run loop in the custom mode.

Now an example of where this is used in practice is user interface tracking.

24:33

```
UI Tracking

* Tracking scroll view
* Specific form of synthetic synchronous
  * Needs touch events, but not others
  * `UITrackingRunLoopMode`
* Common modes
  * `NSRunLoopCommonModes`
  Meta mode when scheduling
  Default, tracking mode, and so on
* Context issues
```

If you have have a scroll view on screen and the user ~~taps~~ *presses* on the scroll view and drags up and down, the UI scroll view wants to run the run loop in order to track the user’s finger and it doesn’t want to return to the main run loop to do that, to the top level. So it uses a form of synthetic synchronous. It gets all of the run loop sources… event sources that are associated with tracking touches, such as the input event sources, and the compositing sources required to composite out to the screen so that you can see things, and its adds those to a custom run loop mode, which is `UITrackingRunLoopMode`. And then it runs the run loop in that mode. And so all of the event sources required to track… run, and other event sources, such as maybe ones… you know… open URL source event sources, or ones related to push notifications, don’t run. This is a hugely important technique.

Now it’s a gotcha for you guys because if you take a run loop source… if you take an object and you schedule it in the default mode, then the run loop isn’t running in that mode… at this point. So you might have created an `NSURLConnection` and it’s receiving data, the user puts their finger down in a scroll view, and it stops receiving data, because its event source isn’t being monitored. And you might work around this by scheduling, not only in the default mode but also in the `UITrackingRunLoopMode`, but there’s actually a better solution to that, and that is to schedule in this common modes object… concept.

The common modes are a meta mode, you can’t run the run loop in the common modes, but you can schedule event sources in the common modes. And when you do so the run loop automatically schedules those event sources in all the likely places that you need to be run, which are these modes called common modes.

Now on iPhone OS the common modes consist of the default run loop mode and the `UITrackingRunLoopMode`, but that could be extended. For example, on Mac OS X there’s a run loop mode for tracking across the menu bar, when the user puts the mouse down in the menu bar. And so the key thing about using the common modes is that you run in all of these modes where you’re likely to need to run, and it’s a good abstraction layer for getting your code running even though the user is interacting with the user interface.

The gotcha with using the common modes is that, if the run loop is running in the default mode, then you can do all sorts of things. You can do pretty much anything. If you put up… if you get a network error and you put up an alert, that’s fine. But if the user is tracking their thumb… you know… their finger across the scroll view and they’re running in the `UITrackingRunLoopMode`, then if you get a network error, ’cause you’re running now, ’cause you’re scheduled in the common modes, if you get a network error and you put up an error alert, that’s going to do bad things, the user is going to be hopelessly confused. It may… it probably won’t crash the frameworks but it’s not going to work… it’s not going to look good.

So if you use this common mode concept, make sure you understand the context you’re running in, and for example you can use other mechanisms, like a short timer that’s scheduled only in the default mode, to defer these sorts of user interface operations.

27:28

```
Run Loop Tips

* No create or destroy
* Invalidate your sources
* Avoid cross thread scheduling
* No recursion in default mode on main thread
* Serialization implies latency
  * Single secondary networking thread
* Beware hidden threads
```

As you’re using run loops, keep in mind the following.

There’s never any need to create or destroy run loops. Run loops are created on demand per thread and they’re destroyed when the thread’s destroyed. So you don’t need to mess with the run loop itself.

In contrast, run loop sources, it’s vitally important that you invalidate them. If you think about those previous diagrams, they’re a mass of pointers with one object pointing to the next object which is pointing back to the other object and so on. So it produces a mass… massive amount of retain loops between all these objects. And if you fail to invalidate your run loop sources, then what happens is those retains loop are never broken and you just leak memory.

So you have a primary… whenever you schedule a run loop mode… an event source in a run loop… you typically sort of have an owning objects, which sort of owns that scheduling. And then when it… before it releases its last reference it’s vitally important that you invalidate the run loop source before you release the last reference to it. Otherwise you just leak. And in some cases… I had a developer today, who was leaking sockets because he was failing to invalidate his… ah… socket run loop sources.

Try to avoid scheduling… cross-thread scheduling, where you’re running on thread A and you’re trying to schedule an event source on thread B’s run loop. In general it’s meant to work… it works at least 99% of the time… so sometimes it just blows up, but worse than that, you know, we could fix those bugs, we know about them, we are fixing them, but the real issue here is that… inside your own code you can get into these race conditions, where the run loop sources are or aren’t schedule, and it just gets very confusing. So always try an schedule on the current thread’s run loop. And if you need to, use `-performSelector:onThread:…` to get over to the thread you want to be on and then schedule on the current run loop from there.

Don’t run the run loop recursively in the default mode on the main thread. The UI frameworks have run loop sources that are only meant to run in the default mode at the top level of the framework, where you’re nearest to main. If you run the framework… the run loop… the main thread’s run loop in the default mode, those sources will fire in the wrong context and bad things will happen. On both iPhone OS and Mac OS X.

Run loops are a serialisation mechanism. This is vitally useful in most cases. If you think about a run loop, it monitors event sources, and then calls the callback, and when the callback returns it returns to monitoring the ~~next event source~~ *event sources*. So these callbacks are inherently serialised, which makes your network programming… very… much easier. It radically reduces the amount of race conditions you have to deal with.

But the issue is, of course, that this serialisation can give you latency. If your main thread is off doing user interface compositing somewhere, or calculating Mandelbrot sets, or whatever its doing, then while it’s doing that, your network event sources aren’t firing because it’s in a run loop callback. And so you really want to, either keep the main thread doing very non-synchronous operations, always returning to the run loop quickly, or in some cases it’s a good idea to create a single secondary thread and put all of your network event sources on that thread. And so they’ll never be held off due to latency on the main thread.

And finally there’s this problem with hidden threads. I’m going to go into that in a little more detail.

30:55

```
Beware Hidden Threads

…
[self performSelectorInBackground:@selector(doStuff) withObject:nil];
…

- (void)doStuff
{
  … do stuff …
  (void) [NSTimer scheduledTimerWithTimeInterval:1.0
    target:self
    selector:@selector(doMoreStuff)
    userInfo:nil
    repeats:NO
  ];
}
```

Here you see me doing `-performSelectorInBackground:…` to call the `-doStuff` method.

Now when the `-doStuff` method runs, it’s running on a secondary thread, that’s the whole point. It does its stuff and, then when it’s finished, it wants to schedule a timer to… to continue doing more stuff, in about a second from now.

Now the thing here is that `-doMoreStuff` method that it’s trying to call can never possibly execute. And the reason is that the `-scheduledTimerWithTimeInterval:…` method always targets the current run loop, which is the run loop associated with the current thread, which is the secondary thread ’cause we run… we ran `-doStuff` with `-performSelectorInBackground:…`. Now when that secondary thread is created by `-performSelectorInBackground:…`, it’s created… it calls `-doStuff` and when `-doStuff` returns it’s destroy. And so any event source you schedule on it will never run because the secondary thread never runs the run loop.

It’s a real gotcha that confuses a lot of people, so watch out for this one. And this is why hidden threads are a danger if you’re mixing and matching threads and run loops.

And that wraps it up for run loops. It’s been a long haul…

Foundation

Reply Posted 3 days ago by eskimo

Add a Comment

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

Copyright © 2022 Apple Inc. All rights reserved. Terms of Use | Privacy Policy | License Agreements