Swift Concurrency

2025. 7. 18.

Swift Concurrency

- 여러 작업을 동시에 처리할 수 있도록 하는 기능
- 앱은 모든 코드를 메인 스레드에서 실행하는 것으로 시작
- 동시성 코드는 단일 스레드 코드보다 복잡하므로 필요할 때만 도입

single-threaded to concurrent

- Single-threaded code
- Asynchronous tasks
- Introducing concurrency
- Sharing data
- Actors

single-threaded code

- @MainActor
  - 코드가 메인 스레드 에서만 실행되고 접근되도록 보장
  - **main actor에 격리**되었다고 표현
- Swift 6.2
  - MainActor를 default actor isolation로 설정 가능
  - Xcode 26으로 생성하는 프로젝트는 기본적으로 Enabled

single-threaded code

- Main Thread
  1. Load file
  2. Decode image
  3. Display image
- remote url인 경우 load file 작업의 main thread lock 지속 시간이 길어짐
- async 필요성

```
func fetchAndDisplayImage(url: URL) throws {
  let data = try Data(contentsOf: url)
  let image = decodeImage(data)
  view.displayImage(image)
}
```

Asynchronous tasks

- Background thread
  1. fetch image
- Main Thread
  2. decode image
  3. display image

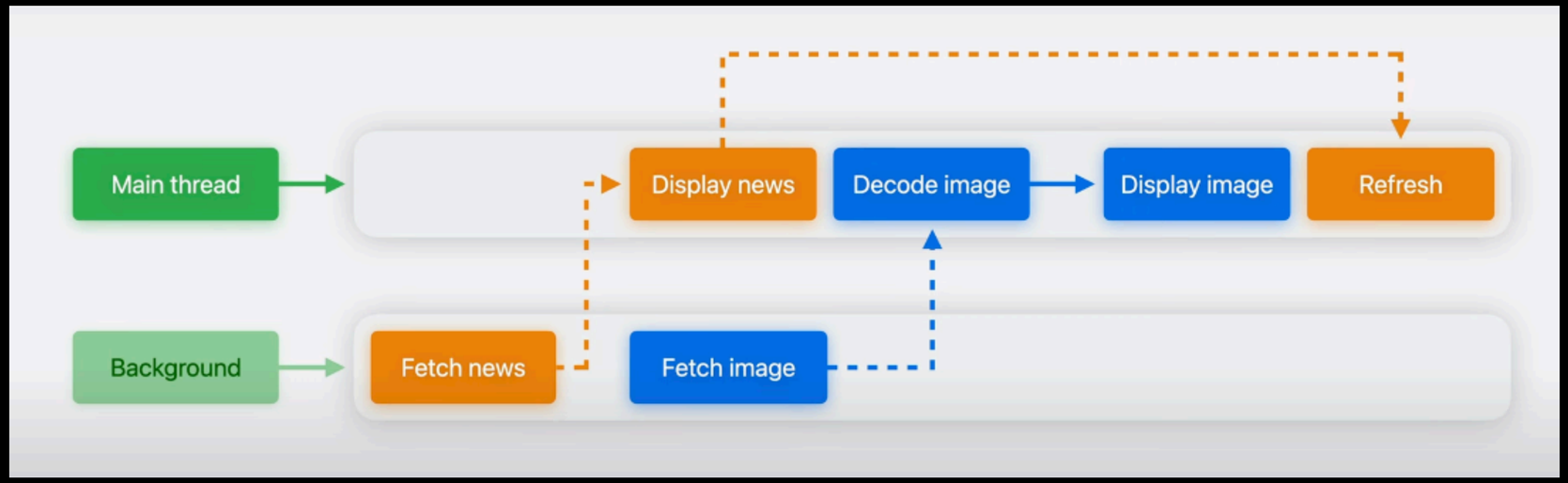
```
func fetchAndDisplayImage(url: URL) async throws {
  let (data, _) = try await URLSession.shared.data(from: url)
  let image = decodeImage(data)
  view.displayImage(image)
}
```

Asynchronous tasks

- 네트워크 요청과 같이 대기 시간이 긴 작업 시 스레드를 묶어두지 않고 **비동기 태스크**를 사용할 수 있음
  - await : 함수가 현재 스레드에서 일시 중단될 수 있음을 나타내며, 대기 중인 이벤트가 발생하면 실행을 재개함
  - 이러한 방식으로 함수를 나누어 다른 작업이 그 사이에 실행될 수 있도록 하여 UI 반응성을 유지함

Asynchronous tasks
interleaving

- 여러 개의 비동기 태스크는 메인 스레드에서 교대로 실행될 수 있음



Introducing concurrency

- 앱이 응답하지 않는다면, 메인 스레드에서 너무 많은 작업이 발생하고 있다는 신호
  - Instruments와 같은 프로파일링 도구를 사용하여 시간이 많이 소요되는 부분 식별
- 동시성 사용하지 않아도 해결할 수 있는 문제라면 그 방법으로 해결하자
  - 동시성의 과한 사용은 앱의 복잡성을 올릴 수 있으므로
- 동시성은 코드의 일부를 메인 스레드와 병렬로 백그라운드 스레드에서 실행하여 UI를 차단하지 않도록 함
  - 이는 시스템의 더 많은 CPU 코어를 사용하여 작업을 더 빠르게 완료하는 데도 사용될 수 있음

Introducing concurrency

@concurrent

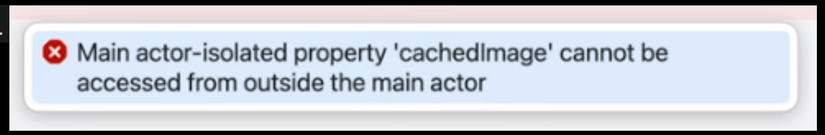
- background thread에서 실행되도록 보장

```
@concurrent
func decodeImage(_ data: Data, at url: URL) async -> Image {
  if let image = cachedImage[url] {
    return image
  }

  // decode image
  let image = Image()
  cachedImage[url] = image
  return image
}
```

Introducing concurrency
@concurrent

- @concurrent 함수에서 main actor 격리된 데이터에 접근하는 경우

```
@concurrent
func decodeImage(_ data: Data, at url
  if let image = cachedImage[url] {
    return image
  }

  // decode image
  let image = Image()
  cachedImage[url] = image
  return image
}
```



Introducing concurrency

@concurrent

- @concurrent 함수에서 main actor 격리된 데이터에 접근하는 경우
  - main actor 데이터에 접근하는 코드를 @MainActor 함수로 옮기기
  - await 사용하기
  - nonisolated 사용하기

Introducing concurrency

@concurrent

- await 사용하기
  - you can use await to access the main actor from concurrent code asynchronously

```
func fetchAndDisplayImage(url: URL) async throws {
  if let image = cachedImage[url] {
    view.displayImage(image)
    return
  }

  let (data, _) = try await URLSession.shared.data(from: url)
  let image = await decodeImage(data)
  view.displayImage(image)
}

@concurrent
func decodeImage(_ data: Data) async -> Image {
  // decode image
  Image()
}
```

Introducing concurrency

@concurrent vs nonisolated

- @concurrent
  - 항상 background 스레드에서 실행됨
- nonisolated
  - client 코드에서 어디서 실행될지 결정할 수 있게함

Introducing concurrency
nonisolated

- nonisolated
  - client 코드에서 어디서 실행될지 결정할 수 있게함

JSONDecoder 는 디코딩하는 Data의 사이즈에 따라서 작업이 클 수도 있고 작을 수도 있다. 따라서 사용하는 쪽에서 MainActor 또는 Background 결정할 수 있게 할 수 있는 nonisolated가 유용한 옵션이 될 수 있다.

```
nonisolated
public class JSONDecoder {
  public func decode<T: Decodable>(_ type: T.Type, from data: Data) -> T {
    fatalError("not implemented")
  }
}
```

Introducing concurrency
nonisolated

- nonisolated
  - MainActor와 마찬가지로 default actor isolation로 설정 가능

  모듈별로 설정이 가능하기 때문에 모듈의 특성에 따라 기본 값을 다르게 지정할 수 있다.

  예를 들어 UI 구현이 많은 모듈은 MainActor로 지정.

Sharing code

- 동시성 코드에서 스레드간 데이터 공유 문제
  - 값 타입의 경우 기본적으로 복사된 데이터가 전달되므로 괜찮
- Sendable 프로토콜은 데이터를 스레드간 안전하게 공유할 수 있게함
  - 구조체와 열거형은 인스턴스 데이터가 모두 Sendable인 경우 Sendable 채택 가능
  - @MainActor 타입은 명시하지 않아도 암시적으로 Sendable

Sharing code
non-Sendable

- 값이 actor에 들어오고 나갈 때 Sendable 타입이 체크된다
- non sendable의 경우에 그 값이 다시 사용되지 않는 경우에만 이동이 허용된다.

Sharing code
non-Sendable

- self : Sendable (@MainActor)
- data : Sendable (valueType)
- url : Sendable(valueType)
- image: non-sendable
  - decodeImage 내에서 참조하는 코드가 없으므로 안전
  - 컴파일에러 발생하지 않는다

```
final class ImageModel {
  var imageCache: [URL: Image] = [:]
  let view = View()

  func fetchAndDisplayImage(url: URL) async throws {
    let (data, _) = try await URLSession.shared.data(from: url)
    let image = await self.decodeImage(data, at: url)
    view.displayImage(image)
  }

  @concurrent
  func decodeImage(_ data: Data, at url: URL) async -> Image {
    Image()
  }
}
```

Sharing code
non-Sendable

- main thread
  - loadImage (use image)
  - displayImage (use image)
- background thread
  - scaleImage (use image)

-> data race

-> ui glitch or crash

컴파일 타임에 체크되어 알려준다

```
func scaleAndDisplay(imageName: String) {
  let image = loadImage(imageName)
  Task { @concurrent in
    image.scaleImage(by: 0.5)
  }

  view.displayImage(image)
}
```

Sharing code

Safe한 코드는?

```
func scaleAndDisplay(imageName: String) {
  let image = loadImage(imageName)
  Task { @concurrent in
    image.scaleImage(by: 0.5)
  }

  view.displayImage(image)
}
```

```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  await view.displayImage(image)
}
```

```
func scaleAndDisplay(imageName: String) {
  Task { @concurrent in
    let image = loadImage(imageName)
    image.scaleImage(by: 0.5)
    await view.displayImage(image)
  }
}
```

```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  await view.displayImage(image)
  image.applyAnotherEffect()
}
```

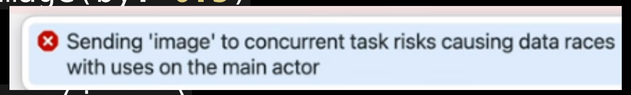
```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  image.applyAnotherEffect()
  await view.displayImage(image)
}
```

Sharing code

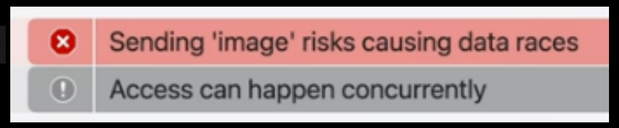
Safe한 코드는?

```
func scaleAndDisplay(imageName: String) {
  Task { @concurrent in
    let image = loadImage(imageName)
    image.scaleImage(by: 0.5)
    await view.displayImage(image)
  }
}
```

```
func scaleAndDisplay(imageName: String) {
  let image = loadImage(imageName)
  Task { @concurrent in
    image.scaleImage(by: 0.5)
  }

  view.displayImage(image)
}
```




```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  await view.displayImage(image)
  image.applyAnotherEffect()
}
```




```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  await view.displayImage(image)
}
```

```
@concurrent
func scaleAndDisplay(imageName: String) async {
  let image = loadImage(imageName)
  image.scaleImage(by: 0.5)
  image.applyAnotherEffect()
  await view.displayImage(image)
}
```

actor

- 앱이 성장함에 따라 main actor에서 다루는 상태가 많아질 수 있으며 UI 지연으로 이어질 수 있음
  - 코드의 non UI 부분 (e.g. 네트워크 관리 코드)을 새로운 actor로 분리할 수 있음
- actor는 main actor와 유사하게 데이터를 격리하여 한 번에 하나의 태스크만 데이터에 접근할 수 있도록 보장
- actor 타입도 Sendable 이므로 actor 객체를 공유할 수 있음
- actor 는 background thread에서 실행될 수 있음

Swift Concurrency 권장 설정

- approachable concurrency : 동시성 작업을 더 쉽게 만드는 향후 기능들을 활성화
- default main actor : main actor
- strict concurrency checking : complete