Error handling concepts

in Swift 2 & 3

04.10.2016

CocoaHeads LE

Motivation

- majority of applications depend on external systems (web servers, file system, databases, hardware accessories)
- high demand for solid error handling
- looking at the concepts helps us to gain a deeper understanding of the issues related to error handling (hopefully!)

Error handling in Swift

- swift/ErrorHandling.rst at master · apple/swift
- The Swift Programming Language (Swift 3): Error Handling

Demo

finally!

Basic rules

- conform to `Error` protocol
- always handle the errors of a throwing method by using
  - a `do-catch` statement
  - `try? or try!`
  - continue propagation

`Optional` vs. `Error`

- use Optionals to represent absence of a value (more on that later)
- use Errors to provide the cause of a failed operation or computation

`rethrows`

- e.g. used in map, filter, reduce, forEach
- means: "it throws if any of a set of named arguments do"

Performance

with regard to Apple:

- error handling in Swift does not involve unwinding the call stack
- performance characteristics of a throw statement are comparable to those of a return statement

concepts

swift/ErrorHandlingRationale.rst

typed propagation

- "Whether the language allows functions to be designated as producing errors or not"
  - Swift: `throws`

default propagation rule

- "Whether, in a language with typed propagation, the default rule is that a function can produce an error or that it can't"
- Swift: a function can't produce an error

statically-enforced typed propagation

- "Whether, in a language with typed propagation, the language enforces this statically, so that a function which cannot produce an error cannot call a function which can without handling it"
- Swift: ✔

marked propagation

- "Whether the language requires all potential error sites to be identifiable as potential error sites"
  - Swift: `try`

manual propagation

- "Whether propagation is done explicitly with the normal data-flow and control-flow tools of the language"
- Swift: `enum` with error property

automatic propagation

- "a language where control implicitly jumps from the original error site to the proper handler"
  - Swift: `throw` & `catch`

Kinds of error

- Simple domain errors
- Recoverable errors
- Universal errors
- Logic failures

Simple domain errors

- calling `String.toInt()` on a string that isn't an integer
- client will often handle the error immediately
- use optional return value

Recoverable errors

- file-not-found, network timeouts
- operation has a variety of possible error conditions
- use `do-try-catch`

Universal errors

- "An error is universal if it could arise from such a wealth of different circumstances that it becomes nearly impracticable for the programmer to directly deal with all the sources of the error."
  - out of memory, stack overflow
  - use Objective-C exceptions (at the moment)

Logic failures

- out of bounds array accesses
- forced unwrap of `nil` optionals
- don't recover, just fix it?
- correct handling is an open question

Why did they …?

specificity of typed automatic propagation

- "should a function be able to state the specific classes of errors it produces"
- lessons learned from Java (arguments from the proposal)
  - libraries generally want to reserve flexibility about the exact kind of error they produce
  - exceptions list end up just restating the library's own dependencies
  - or wrapping the underlying errors in ways that loses critical information

Async error handling

- Swift lacks language constructs for async execution and error handling

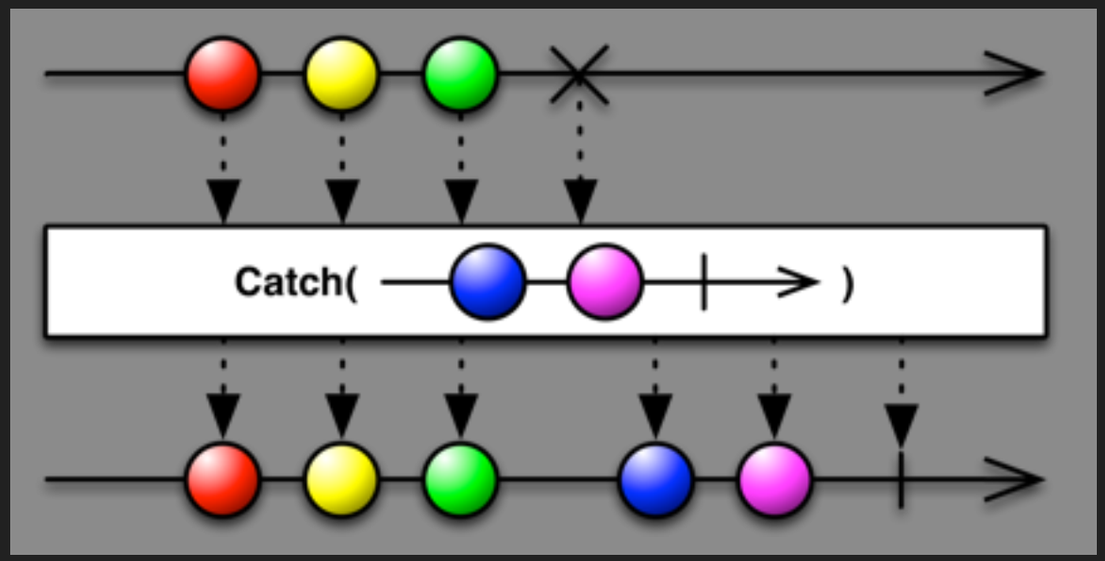
Option 1:

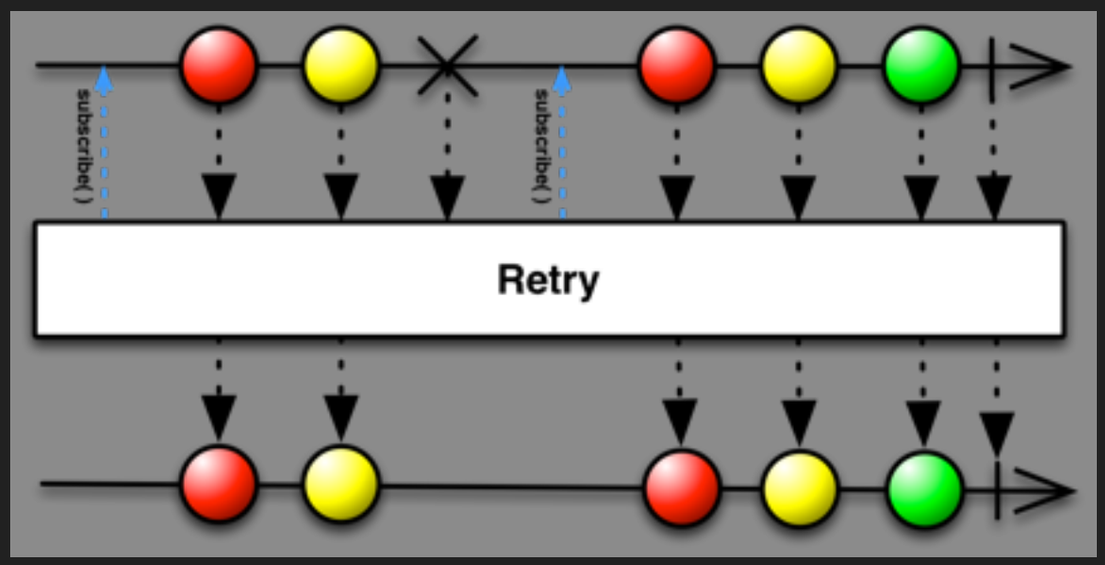
```
enum Result<T> {
    case success: T
    case error: Error
}

func readFile() -> Result<Data>
```

Async error handling

Option 2: Reactive Library (RxSwift or Reactive Cocoa)





reactivex.io/documentation/operators.html#error

Modeling tips

Ambiguity of Optionals

```
struct Dog {
    let dogID: DogID
    let master: String?
}

// load from a web server (this can fail)
func loadRemoteDogMaster(dogID: DogID) -> String? {
    // some mysterious closed-source implementation
}

// execute
let name = loadRemoteDogMaster(dogID: 43)
```

What do we know if name == nil?

a) dog has no master

b) OR operation failed

Ambiguity of Optionals

Reason: Using Optionals for app and domain modeling interchangeably

Possible solution: Use Optionals for domain modeling and errors or enums for operation results.

Explicit success/error

```
func writeFileToDisk() -> Bool

// or even worse
func writeFileToDisk() -> Int
```

- a `Bool`/`Int` is not quite descriptive in this case
- return value can be easily ignored
- use enums (can be ignored too) or throw

Don't do that (at work)

```
// this produces potentially vague defined state
func loadSomeReallyEssentialValue() -> String? {
    var result: String? = nil
    if let remoteValue = loadFromServer() {
        result = remoteValue.convert()
    } else {
        result = ???
    }
    return result
}

// better: fail and cancel/retry the operation
func loadSomeReallyEssentialValue() throws -> String {
    guard let remoteValue = loadFromServer() else {
        throw RequestError.couldNotSatisfy
    }
    let remoteValue = try loadFromServer()
    return remoteValue.convert()
}
```

In general

- avoid undefined state
- use at least a general error for all cases you don't want to think of (with identifier/error code)

try `try`

thanks :-)