Reactive Programming

Junghoo Cho

cho@cs.ucla.edu

Reactive Programming - Junghoo Cho - cho@cs.ucla.edu

Summing an Array

```
let sum = 0;
for (let i = 0; i < a.length; i++) {
    sum += a[i];
}
console.log(sum);
```

- Q: What if a is a list not an array?
- Code needs to be rewritten because a list does not support a

Summing an Array

```
let sum = 0;
for (let i = 0; i < a.length; i++) {
    sum += a[i];
}
console.log(sum);
```

- What we really want is
  1. "next" item in general, not `i++` in particular
  2. checking for the end, not `i < a.length`
- *Iterable*: generalization of array
  1. `next()`: returns the next item in the iterable
  2. `end()`: returns `true` if we reached the end

Iterable: Example

```
let sum = 0;
while (!a.end()) {
    sum += a.next();
}
console.log(sum);
```

- Doesn't matter if a is an array, list, table, …
- Q: What happens if a is coming over the network? We don't w
  stuck waiting for the next item!

Iterable: Example

```
let sum = 0;
while (!a.end()) {
    sum += a.next();
}
console.log(sum);
```

- What we really want is
  1. Sum the next item when it is available
  2. Print the sum when all is done
- Observable: generalization of iterable
  1. `onNext(e)`: called on every item `e`
  2. `onCompleted()`: called when done

Observable: Example

```
let sum = 0;
a.onNext = (x => { sum += x; });
a.onCompleted = (() => { console.log(sum)); });
```

Iterable vs Observable

Task	Iterable	Observable
Consume Item	`T next()`	`onNext(T)`
Encounter Error	`throw Error(e)`	`onError(e)`
Finish	`end()`	`onCompleted()`

- Observable is mostly assumed to be “push”
  - But it doesn’t have to be
  - We don’t really care how it is implemented

Key Terminology

- Observable
  - An object that produces a sequence of “events”
  - “Publisher”
- Observer
  - An object interested in the events from an observable
  - “Subscriber”

Everything is Observable!

- Array is observable
- Iterable is observable
- Events are observable
- Variable is observable
- John is observable
  - He gives lectures
  - He assigns projects
  - As long as we get notified, we don't care how they are done
- The world is full of observables!

Reactive Programming

- Write program as a a set of “operators” performed on observ
  - Everything is observable!
- Program consists of “reactions” to input events
  - Reactive programs “react to” input events

Operator

- Operator transforms input observables to output observable
  - Observable(s) → operator → Observable(s)
- Example: `filter(x > 0)`
  - 10, -2, 3, -1, … → `filter(x > 0)` → 10, 3, …
- Complex operators can be created by "piping together" simp operators
  - Obsv → op1 → Obsv' → op2 → Obsv" → …
- Final goal: Using simple operators, convert single-click strea double/triple/… clicks if there is less than 250ms pause in be clicks

Operator: Transform

- `filter()`: "filter" only those events that meets a condition
  - `filter(x => x > 0)`: 1, -3, 2, -1, 3, … → 1, 2, 3, …
- `map()`: "map" every input event to an output event
  - `map(x => 2*x)`: 1, 2, 3, … → 2, 4, 6, …
- `flatmap()`: one input event produces multiple output event the output to single event stream
  - `flatmap(x => [x, x+10])`: 1, 2, … → 1, 11, 2, 12, …

Operator: Aggregate

- `reduce()`: perform cumulative operation on (result-so-far, ne
  and produce one final output at the end
  - `reduce((a, b) => a+b)`: 1, 2, 3 → 6
- `scan()`: similar to reduce, but one output per every input
  - `scan((a, b) => a+b)`: 1, 2, 3, … → 1, 3, 6, …

Operator: Buffering

- `buffer(time)`: "buffer" input events for the specified period produce buffered inputs as output
  - `buffer(250ms)`: 1, 2, 3, 4, 5, … → [1, 2], [3, 4, 5], …
- `bufferTime(bufferTimeSpan, bufferCreationInterval)` for `bufferTimeSpan` every `bufferCreationInterval`
  - `bufferTime(50ms, 100ms)`: 1, 2, 3, 4, 5, … → [1, 2], [4, 5], …
- `bufferCount(m, n)`: "buffer" `m` events every `n` input events
  - `bufferCount(3, 2)`: 1, 2, 3, 4, 5, … → [1, 2, 3], [3, 4, 5], …

Operator: Throttling

- `debounce(time)`: produce an output after a specified period inactivity

Multi-Way Operators

- Operators so far: one input, one output
- Operators next: *multiple inputs*, one output

Operator: Example

- `merge()`: "merge" events from all input streams into a single stream

Operator: Example

- `zip()`: take one event from each input stream and generate from the pair

Operator: Example

- `A.buffer(B)`: buffer events from A until B emits a new event

Operator: Example

- `A.bufferToggle(openings, closings)`: buffer events from openings until closings

Operator: Example

- `join()`: produce one output per every input event pair with window

- See RxJS operators for more operators

Simple to Complex: Example

- Convert single-click stream into double/triple/… clicks if the than 250ms pause in between clicks
- Q: Can we do it using operators that we have seen so far?
- A: Operators to use
  - `buffer()`
  - `debounce(250ms)`
- `clickstream.buffer(clickstream.debounce(250ms)).mae.length);`

Consuming Observable: `subscribe`

- Once desired observable is created, *observer* can set `onNext`, `onCompleted` handlers to take appropriate actions
- `obsv.subscribe(onNext, onError, onCompleted)`
  - Bind `onNext`, `onError`, `onCompleted` handlers to the observable `obsv`

When is Observable Useful?

- Observable can be used for any type of programming
- But it is particularly useful when dealing with ***stream of even***
  - UI apps
  - Asynchronous programs
  - …

Declarative Programming

- Reactive program is declarative
  - We specify what to do, not how to do
  - Different from procedural or imperative programming
- *Declarative program* provides enormous optimization oppor
  - We don't care when, where, and how our program is executed
  - Push or pull, here or there, we don't care
  - System can decide the best way to execute the program
- SQL, Map/reduce, …, can be seen as reactive programs

Pure Function

- The same input, the same output
  - Output is determined only by input
  - Function can be understood on its own, nothing else
- No side effect
  - Function does not change outside “states”
  - “Immutable object”
    - Do not modify input parameter directly
    - Create a new separate output object
  - No need to worry about leaving “unexpected side effect”

Pure Function

- Pure functions make programs
  - Easy to understand
  - Easy to predict its behavior
  - Easy to debug and maintain
- Reactive programs are expected to use pure functions
  - Strongly encouraged, but not strictly enforced in most cases

Reactive Program: Properties

- Functional
- Declarative
- Asynchronous

→ leads to easy-to-understand, easy-to-optimize, easy-to-mair program

RxJS

- RxJS: JavaScript library for Reactive programming
- Many tutorials on reactive programming at http://reactivex.i
- Combine Framework: Swift API for reactive programming on iOS/macOS

What We Learned

- Iterator: `next()`, `end()`
- Observable: `onNext()`, `onCompleted()`
  - Observable and observer
- Reactive program
  - Operators on observables
  - Pure functions
  - Declarative (vs procedural)
- Reactive programming is a generalization of event-driven pr
  SQL, map/reduce, …