从 React

到 ClojureScript

题叶

FP, GUI & Writing

ChenYong, 饿了么前端

CoffeeScript, ClojureScript

概要

- 回顾函数式编程
- 认识 ClojureScript
- React 的函数式特性和不足
- ClojureScript 写 Virtual DOM
- 思考, 关于数据流

▲ chenglou 845 days ago [-]

I contribute to React. I'm definitely a big fan of ClojureScript and I borrow ideas from it when I can. I currently have a few crazy ideas that are direct results of looking into the Clojure (or Haskell) ecosystem and finding that they've solved my problems, but better. In general, some of the problems we face, at language/library/ecosystem level, might simply not exist in these languages. It's a shame they're not more popular.

Regarding popularity, I feel that sometimes the web development community can be ironically dogmatic, considering that it's a free and open platform (see: HN/Reddit's initial reaction to React). Looking back, I still find it incredible that React actually gained so much traction when it defies so many of the pre-established "best practices". I think lots of people, including me, are learning more and more to take a step back before dismissing an idea.

So if you were one of those that dismissed Clojure/ClojureScript because "parentheses in front of my function call?", "snake case isn't for me", or "it's too different from idiomatic js" (I hope this one isn't too much of a strawman here), then I urge you to give it another try. Learning React doesn't have to be the only time you open your mind to new ideas and reconsider best practices.

“Reason is a new interface to OCaml - a highly expressive dialect of the ML language featuring type inference and static type checking.”

– https://facebook.github.io/reason/

- 1957 Fortron
- 1958 LISP
- 1962 Simula
- 1964 BASIC
- 1972 C
- 1972 Smalltalk
- 1972 ML
- 1975 Scheme
- 1980 C++
- 1984 Common Lisp
- 1986 Erlang
- 1987 Self
- 1990 Haskell
- 1995 Java
- 1995 JavaScript
- 2003 Scala
- 2007 Clojure
- 2009 Go
- 2009 CoffeeScript
- 2010 Rust
- 2011 ClojureScript
- 2014 Swift

https://en.wikipedia.org/wiki/History_of_programming_languages

- Lisp, a practical mathematical notation for programs, including tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, and the self-hosting compiler.

- Scheme, a minimalist design philosophy, lexical scope, first to require tail-call optimization, (one of first to) support first-class continuations

- ML, "LISP with types", Hindley–Milner type system, type inference, pattern matching

- Common Lisp, not an implementation, but rather a language specification
- Haskell, standardized, general-purpose purely functional programming language. By 1987, more than a dozen non-strict, purely functional programming languages existed
- Clojure, a general-purpose programming language with an emphasis on functional programming. treats code as data and has a macro system. support for lazy sequences and encourages the principle of immutability and persistent data structures

Clojure(Script)

```
515  (defn load-file
516    ([repl-env f] (load-file repl-env f *repl-opts*))
517    ([repl-env f opts]
518      (if (:output-dir opts)
519        (let [src (cond
520                    (util/url? f) f
521                    (.exists (io/file f)) (io/file f)
522                    :else (io/resource f))
523              compiled (binding [ana/*reload-macros* true]
524                         (cljsc/compile src
525                           (assoc opts
526                             :output-file (cljsc/src-file->target-file src)
527                             :force true
528                             :mode :interactive)))]
529          ;; copy over the original source file if source maps enabled
530          (when-let [ns (and (:source-map opts) (first (:provides compiled)))]
531            (spit
532              (io/file (io/file (util/output-directory opts))
533                (util/ns->relpath ns (util/ext (:source-url compiled))))
534              (slurp src)))
535          ;; need to load dependencies first
536          (load-dependencies repl-env (:requires compiled) opts)
537          (-evaluate repl-env f 1 (cljsc/add-dep-string opts compiled))
538          (-evaluate repl-env f 1
539            (cljsc/src-file->goog-require src
540              {:wrap true :reload true :macros-ns (:macros-ns compiled)})))
541        (binding [ana/*cljs-ns* ana/*cljs-ns*]
542          (let [res (if (= File/separatorChar (first f)) f (io/resource f))]
543            (assert res (str "Can't find " f " in classpath"))
544            (load-stream repl-env f res)))))
```

Clojure

- clojure.org, Rich Hickey
- Dynamic, interactive development
- Lisp, code as data, Macro
- Functional, persistent data structure
- Atoms to manage shared, synchronous, independent state
- JVM, CLR, JavaScript

"ClojureScript is a compiler for Clojure that targets JavaScript. It emits JavaScript code which is compatible with the advanced compilation mode of the Google Closure optimizing compiler."

– *clojurescript.org*

尝试 ClojureScript

使用 Lumo

```
=>>
=>>
=>>
=>>
=>> brew install lumo
Warning: lumo-1.0.0 already installed
=>> lumo
Lumo 1.0.0
ClojureScript 1.9.293
 Docs: (doc function-name-here)
 Exit: Control+D or :cljs/quit or exit

cljs.user=> (def a 1)
#'cljs.user/a
cljs.user=> (def b 2)
#'cljs.user/b
cljs.user=> (defn f [x y] (* x y))
#'cljs.user/f
cljs.user=> (f a b)
2
cljs.user=> (map inc [1 2 3 4 5])
(2 3 4 5 6)
cljs.user=> (get {:a 1 :b 2} :a)
1
cljs.user=> (conj (map inc [1 2 3 4 5]) 7 8 9)
(9 8 7 2 3 4 5 6)
cljs.user=>
```

学习 ClojureScript

- https://learnxinyminutes.com/docs/clojure/
- http://cljs.info/cheatsheet/
- https://github.com/clojure-china/cljs-book
- http://www.michielborkent.nl/fpamsclj/fpamsclj.pdf

React

seeing from ClojureScript side

纯函数

- 渲染函数没有副作用
- Virtual DOM 代码方便复用, 组件化, 服务端渲染
- Store updater 没有副作用

组件化

- 组件模拟纯函数, 输入一致, 输出一致
- 组件可以被缓存, 引用透明
- 组件可以进行复合, 形成高阶组件

不可变数据

- 用于快速判断组件是够改变, 减少 CPU 开销
- 结构复用, 减少内存申请, 从而减少资源开销
- 函数式语言, 特殊的优化内存的方案

全局可变状态

- Single Source of Truth
- 限制副作用造成的影响
- 嵌套的树形数据结构的大量使用
- 注意: 数据不可变, 引用(reference)可变

单向数据流

- 单一的数据源
- 更新需要冒泡, 流动, 形成数据流
- 构建复杂的应用
- 同时限制数据的逆向流动(如何限制)

React bad parts

seeing form ClojureScript side

this.

- this 指针需要绑定
- this.setState 是过程式写法, 存在问题
- ES6 class 强化的局部可变状态
- (ClojureScript 可以构造 React, 不需要 OOP)

JSX

- XHP, 静态检查, 容易上手
- 额外的编译器, 额外的字符, 推动了 ES6
- JSX 的过度使用(route, fetch)
- if/switch 不是表达式
- (ClojureScript 通过 S 表达式构造 DSL)

Babel

- ES2016, 实现新的语言特性
- 频繁更新(200+ 个版本), 配置复杂
- (ClojureScript, 通过 Macro 系统直接扩展语法)

Redux

- 意外地复杂...
- 什么才是准确的 Store 的抽象?
- (ClojureScript, 可变状态 + watcher)

Immutable.js

- API 相对复杂, 使用成本. 体积不小
- 与 JavaScript 原生数据结构经常转化和混用
- (ClojureScript 中是原生的数据类型)

热替换

- React 有组件私有状态, 不好处理
- JavaScript 有太多副作用需要处理
- (ClojureScript 副作用受到限制, 相对轻松)

React + ClojureScript

- Reagent, 2306+
- Om, 5796+
- Rum, 802+
- Quiescent, 563+
- Respo, 45+

Reagent

- http://reagent-project.github.io/
- Reagent provides a minimalistic interface between ClojureScript and React.

Source

```
(ns example
  (:require [reagent.core :as r]))

(defn atom-input [value]
  [:input {:type "text"
           :value @value
           :on-change #(reset! value (-> % .-target .-value))}])

(defn shared-state []
  (let [val (r/atom "foo")]
    (fn []
      [:div
       [:p "The value is now: " @val]
       [:p "Change it here: " [atom-input val]]])))
```

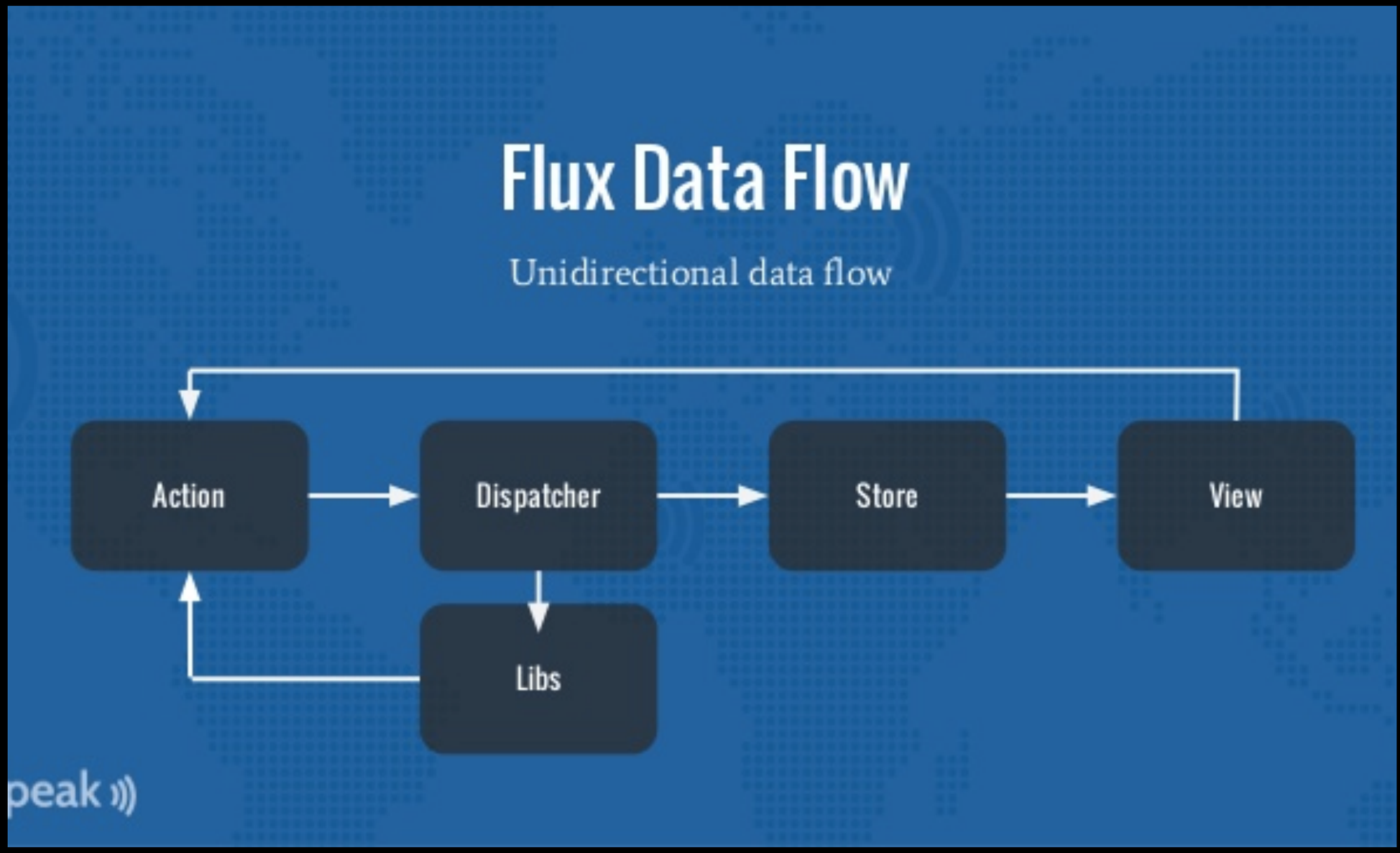
Respo

- http://respo.site
- 重新实现 Virtual DOM
- 将 Virtual DOM 作为数据用于渲染
- Patching first screen(优化首屏动画)
- 全局 state tree, 便于热替换
- 组件状态更加自由的数据结构
- core.async(对应 Rx)

ClojureScript 的不足

- 编译工具不够完善
- npm 生态跟进太慢, 没有大厂背书
- 国内人才储备, 文档储备
- 不够成熟, 等待 WebAssembly

思考…



单向数据流

http://www.slideshare.net/speak_io/electron-flux-react

“Streams everywhere!”

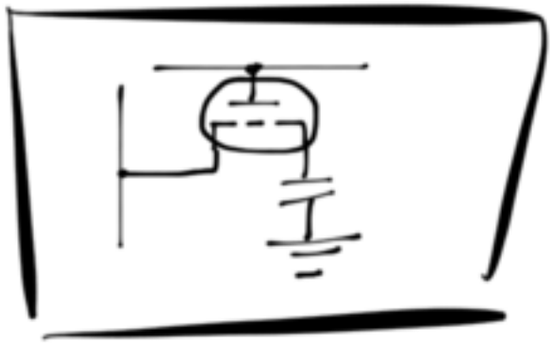
– https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/

DB

Cache

HTML DOM

Video mem



raw data as written

Business logic

eg. JSON over REST

UI logic

template rendering

Browser rendering

pixels

更多

- 用函数实现数据流, 抽象, 灵活, 严谨
- 相应提供了性能优化方案
- 服务端到客户端的数据流(Diff/Patch)
- Virtual DOM 到界面动画的数据流

Questions?

- clojure-china.org
- map.clj.im
- tiye.me