ClojureScript 带给 React 的借鉴意义

提纲

- ClojureScript 语法
- 语言特点
- 运行和编译 cljs
- 热替换
- 不可变数据
- 树形数据
- 可变状态(Atom)
- Cursor

语法

```
(defn make-foreign-js-header
  "goog.provide/goog.require statements for foreign js files"
  [{:keys [provides require-order]}]
  (let [sb (StringBuilder.)]
    (doseq [provide provides]
      (doto sb
        (.append "goog.provide(\"")
        (.append (str (comp/munge provide)))
        (.append "\");\n")))
  (doseq [require require-order]
    (doto sb
      (.append "goog.require(\"")
      (.append (str (comp/munge require)))
      (.append "\");\n")))
  (.toString sb)
  ))
```

```clojure
(defn counting-component []
  [:div
   "The atom " [:code "click-count"] " has value: "
   @click-count ". "
   [:input {:type "button" :value "Click me!"
            :on-click #(swap! click-count inc)}]])
```

```clojure
(defn make-js-module-per-source
  [{:keys [compiler-env build-sources] :as state}]

  (let [base
        (doto (JSModule. "goog.base.js")
          (.add (SourceFile/fromCode "goog.base.js"
                  (str (output/closure-defines-and-base state)
                       goog-nodeGlobalRequire-fix))))

        js-mods
        (reduce
          (fn [js-mods src-name]
            (let [{:keys [ns require-order output js-name] :as src}
                  (get-in state [:sources src-name])

                  defs
                  (when ns
                    (->> (get-in compiler-env [::ana/namespaces ns :defs])
                         (vals)
                         (filter #(get-in % [:meta :export]))
                         (map :name)
                         (map (fn [def]
                                (let [export-name
                                      (-> def name str comp/munge pr-str)]
                                  (str export-name ":" (comp/munge def)))))
                         (str/join ",")))

                  code
                  (str (if (util/foreign? src)
                         (make-foreign-js-header src)
                         output)
                       ;; module.exports will become window.module.exports, rewritten later
                       (when (seq defs)
                         (str "\nmodule.exports={" defs "};")))

                  js-mod
                  (doto (JSModule. (util/flat-filename js-name))
                    (.add (SourceFile/fromCode js-name code)))]

              #_(let [file (io/file "target" "npm-bug" js-name)]
                  (io/make-parents file)
                  (spit file code))

              ;; everything depends on goog/base.js
              (.addDependency js-mod base)

              (doseq [dep
                      (->> require-order
                           (remove '#{goog})
                           (map #(get-in state [:provide->source %]))
                           (distinct)
                           (into []))]
                (let [other-mod (get js-mods dep)]
                  (.addDependency js-mod other-mod)))

              (assoc js-mods src-name js-mod)))
          {}
          build-sources)

        modules
        (->> build-sources
             (map (fn [src-name]
                    (let [{:keys [name js-name] :as src}
                          (get-in state [:sources src-name])]

                      {:name name
                       :js-name (util/flat-filename js-name)
                       :js-module (get js-mods src-name)
                       :sources [name]})))
             (into [{:name "goog/base.js"
                     :js-name "goog.base.js"
                     :js-module base
                     :sources ["goog/base.js"]}]))]

    (assoc state ::modules modules)))
```

```clojure
(+ 1 2 3)                ; => 6
(= 1 2)                  ; => false
(if true "y" "n")        ; => "y"

(if (= a b c)    ; <-- determines if a=b=c
    (foo 1)      ; <-- only evaluated if true
    (bar 2)      ; <-- only evaluated if false
    )

; define k as 3
(def k 3)            ; <-- notice that k is not evaluated here
                     ;     (def needs the symbol k, not its value)

; make a greeting function
(fn [username]                  ; <-- expected parameters vector
  (str "Hello " username))

; creating local bindings (constants)
(let [a (+ 1 2)
      b (* 2 3)]
  (js/console.log "The value of a is" a)
  (js/console.log "The value of b is" b))
```

特点

- Lisp 方言, 2007
- 编译到 JVM, JavaScript
- 不可变数据
- Macros
- 状态管理(并发…)
- Gradual Typing(code.typed)

執行

```
# ClojureScript environment based on V8
npm install -g lumo-cljs

# a friendly ClojureScript Compiler
npm install shadow-cljs
```

1. /Users/chen/repo/gist/highlight (node)

```
$ lumo
Lumo 1.5.0
ClojureScript 1.9.542
Node.js v7.10.0
 Docs: (doc function-name-here)
       (find-doc "part-of-name-here")
 Source: (source function-name-here)
 Exit: Control+D or :cljs/quit or exit

cljs.user=> (+ 1 2)
3
cljs.user=> (println "Clojure")
Clojure
nil
cljs.user=> (defn f [x y] (inc (+ x y)))
#'cljs.user/f
cljs.user=> (f 3 4)
8
cljs.user=>
```

shadow-cljs.edn

```
{:source-paths ["src"]
 :dependencies []
 :builds {:app {:output-dir "target/"
                :asset-path "."
                :target :browser
                :modules {:main {:entries [app.main]}}
                :devtools {:after-load app.main/on-reload!}}}}
```

```
shadow-cljs --build app --dev
```

Demo

shadow-cljs 热替换

不可变数据

2. /Users/chen/repo/gist/highlight (node)

```
=>> node
> var a = {a: {b: "c"}}
undefined
> a.b = 'd';
'd'
> a
{ a: { b: 'c' }, b: 'd' }
> b = [1, 2, 3]
[ 1, 2, 3 ]
> c = b.push(4)
4
> b
[ 1, 2, 3, 4 ]
> c
4
>
```

2. /Users/chen/repo/gist/highlight (node)

```
cljs.user=> (def a {:a {:b "c"}})
#'cljs.user/a
cljs.user=> (def a' (assoc-in a [:a :b] "d"))
#'cljs.user/a'
cljs.user=> a
{:a {:b "c"}}
cljs.user=> a'
{:a {:b "d"}}
cljs.user=> (def b [1 2 3])
#'cljs.user/b
cljs.user=> (def c (conj b 4))
#'cljs.user/c
cljs.user=> b
[1 2 3]
cljs.user=> c
[1 2 3 4]
cljs.user=>
```

2. /Users/chen/repo/gist/highlight (node)

```
coffee> a = Immutable.fromJS {a: {b: "c"}}
Map { "a": Map { "b": "c" } }
coffee> a2 = a.setIn ['a', 'b'], 'd'
Map { "a": Map { "b": "d" } }
coffee> a
Map { "a": Map { "b": "c" } }
coffee> a2
Map { "a": Map { "b": "d" } }
coffee> b = Immutable.fromJS [1, 2, 3]
List [ 1, 2, 3 ]
coffee> c = b.push 4
List [ 1, 2, 3, 4 ]
coffee> b
List [ 1, 2, 3 ]
coffee> c
List [ 1, 2, 3, 4 ]
coffee>
```

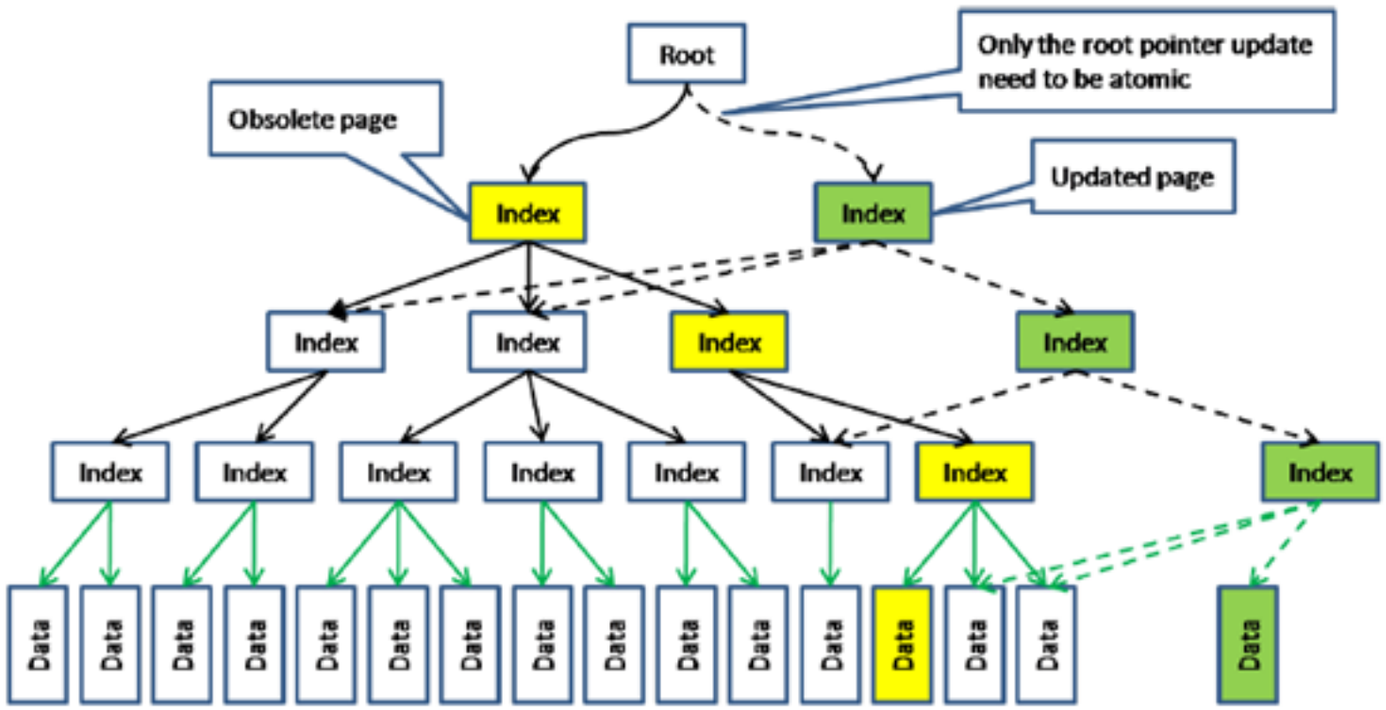
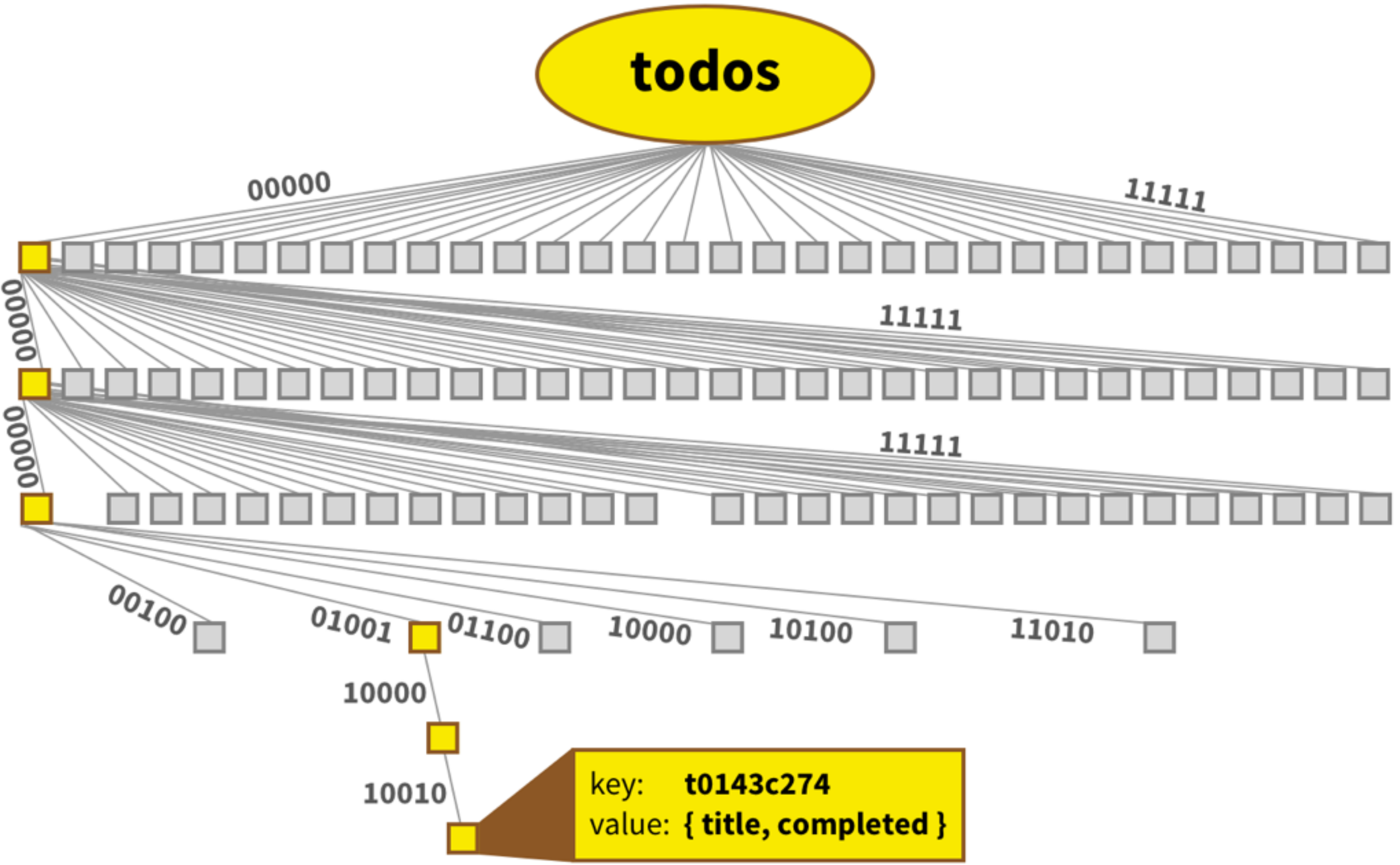
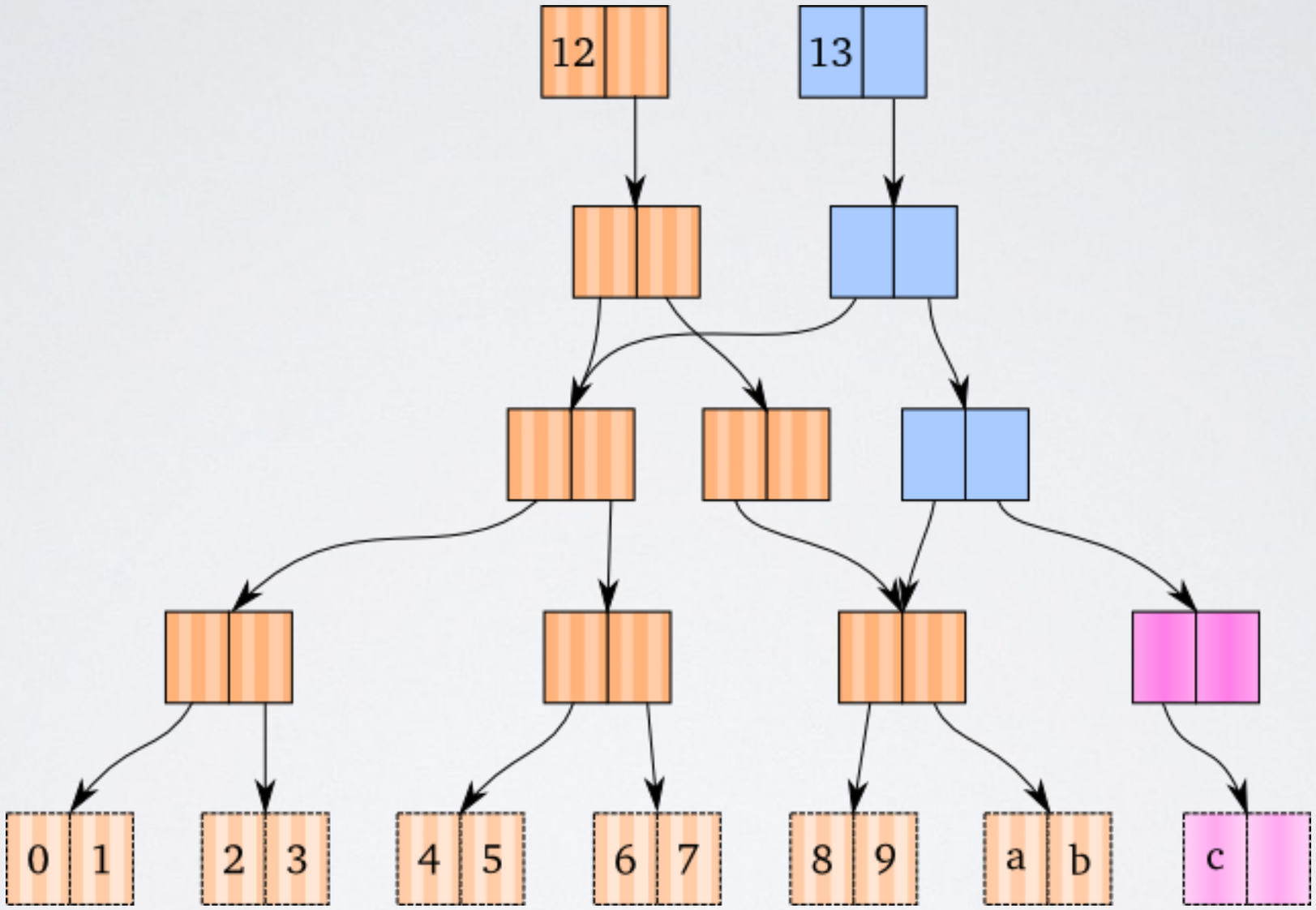
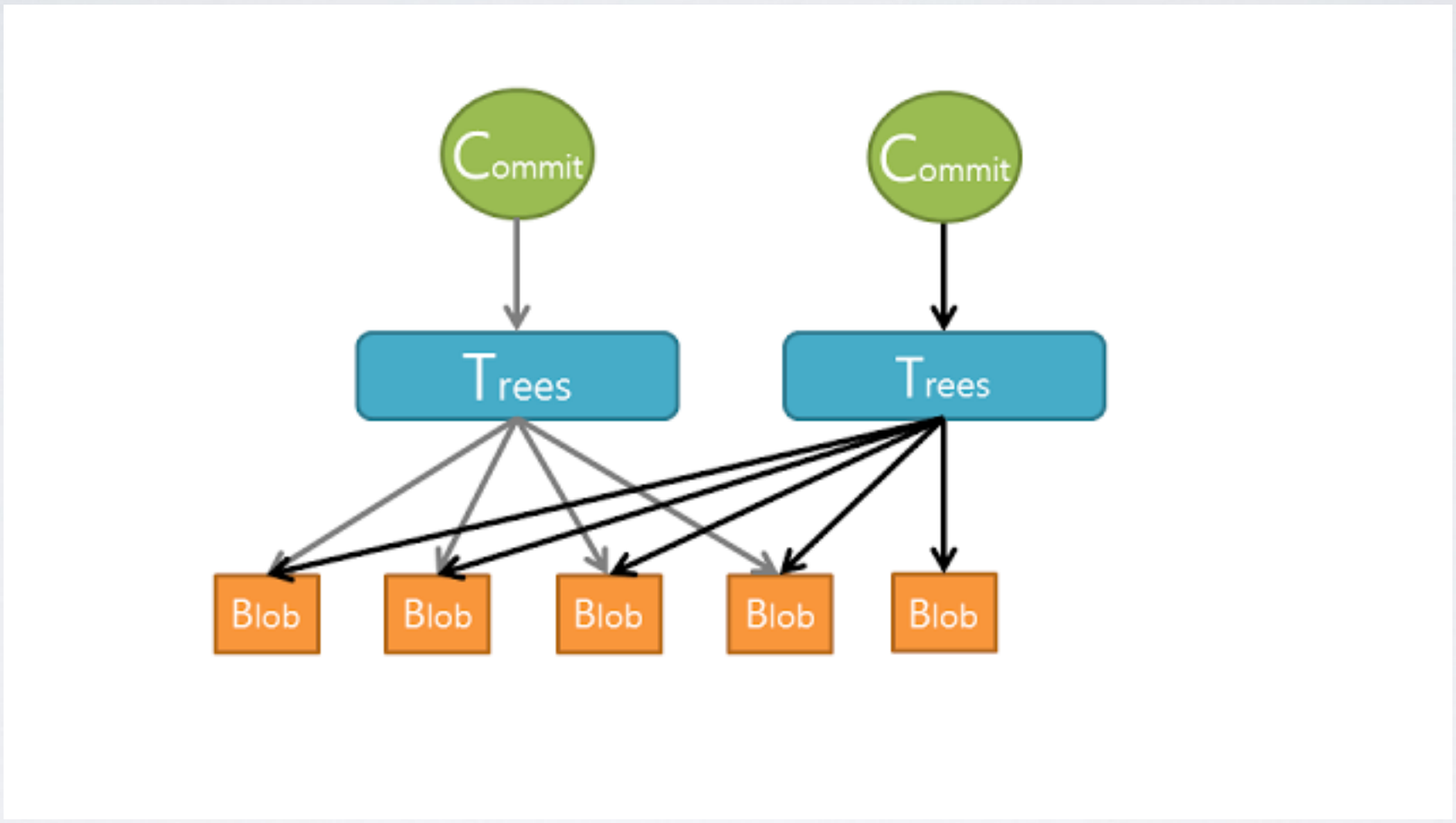
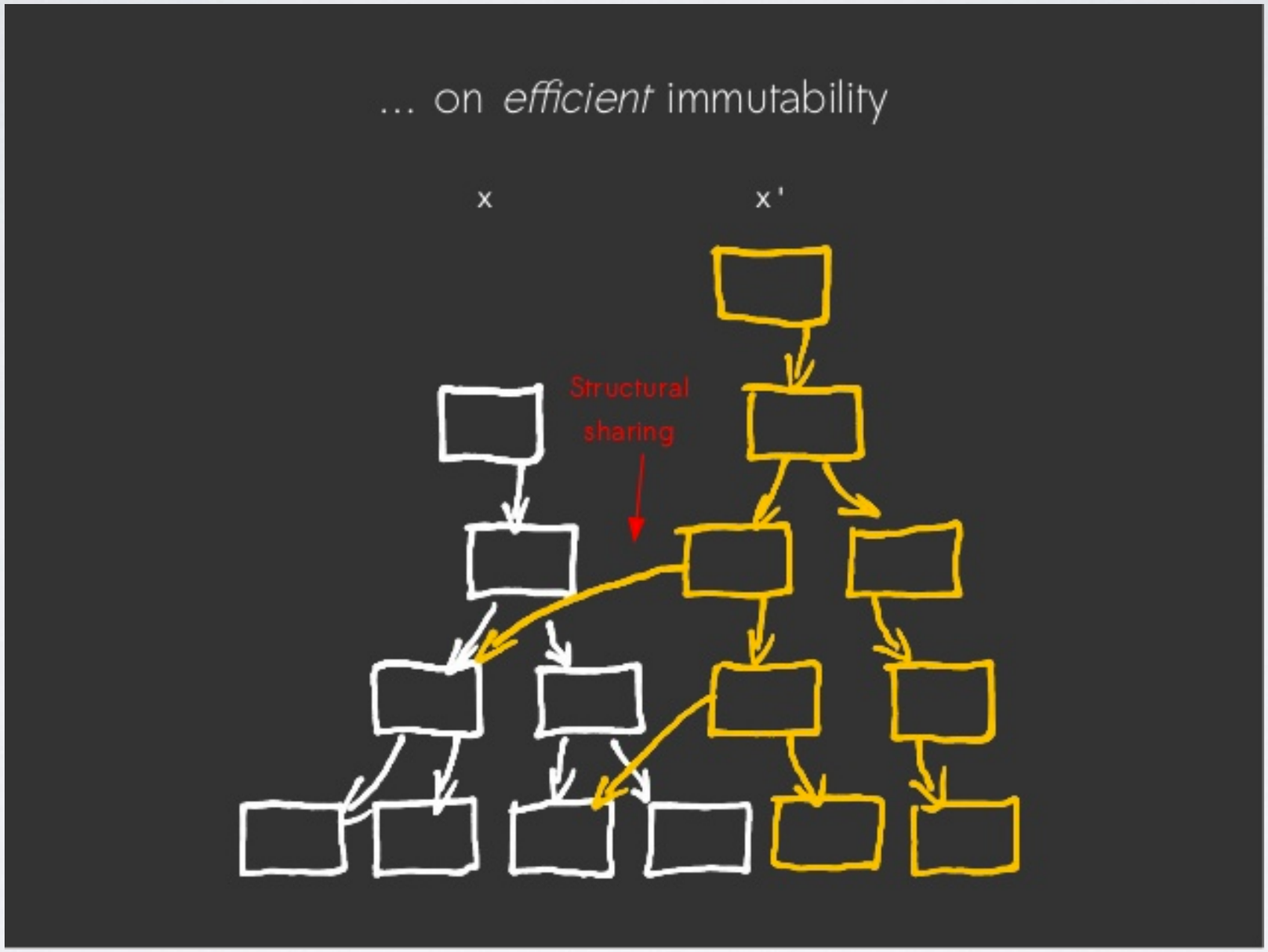


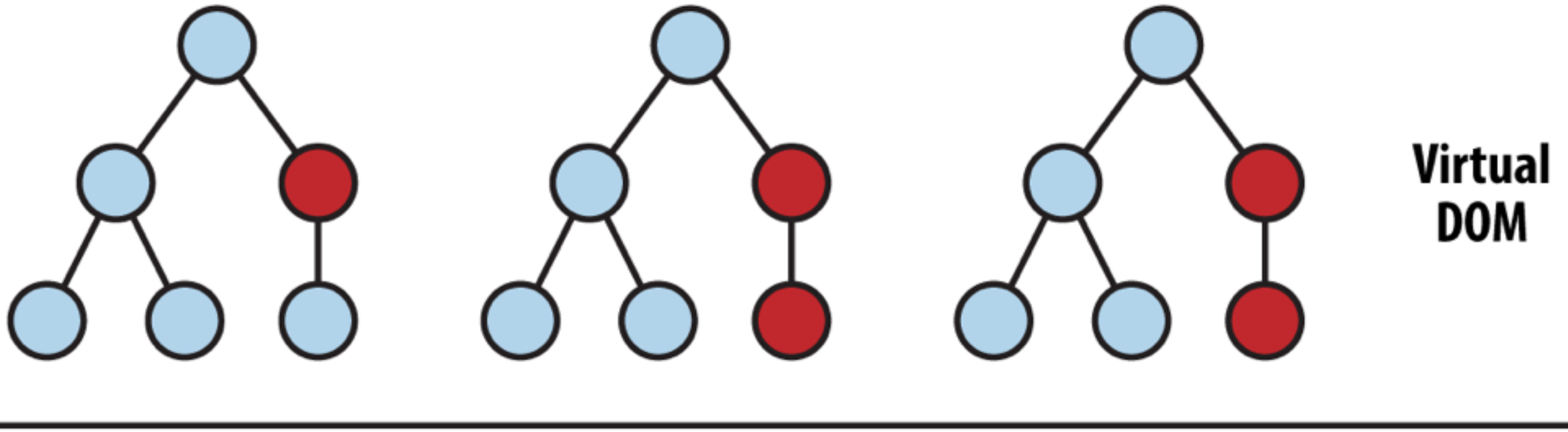
Figure 3.13.: Storage Layout – Copy-on-modfify in CouchDB (taken from [Ho09a])

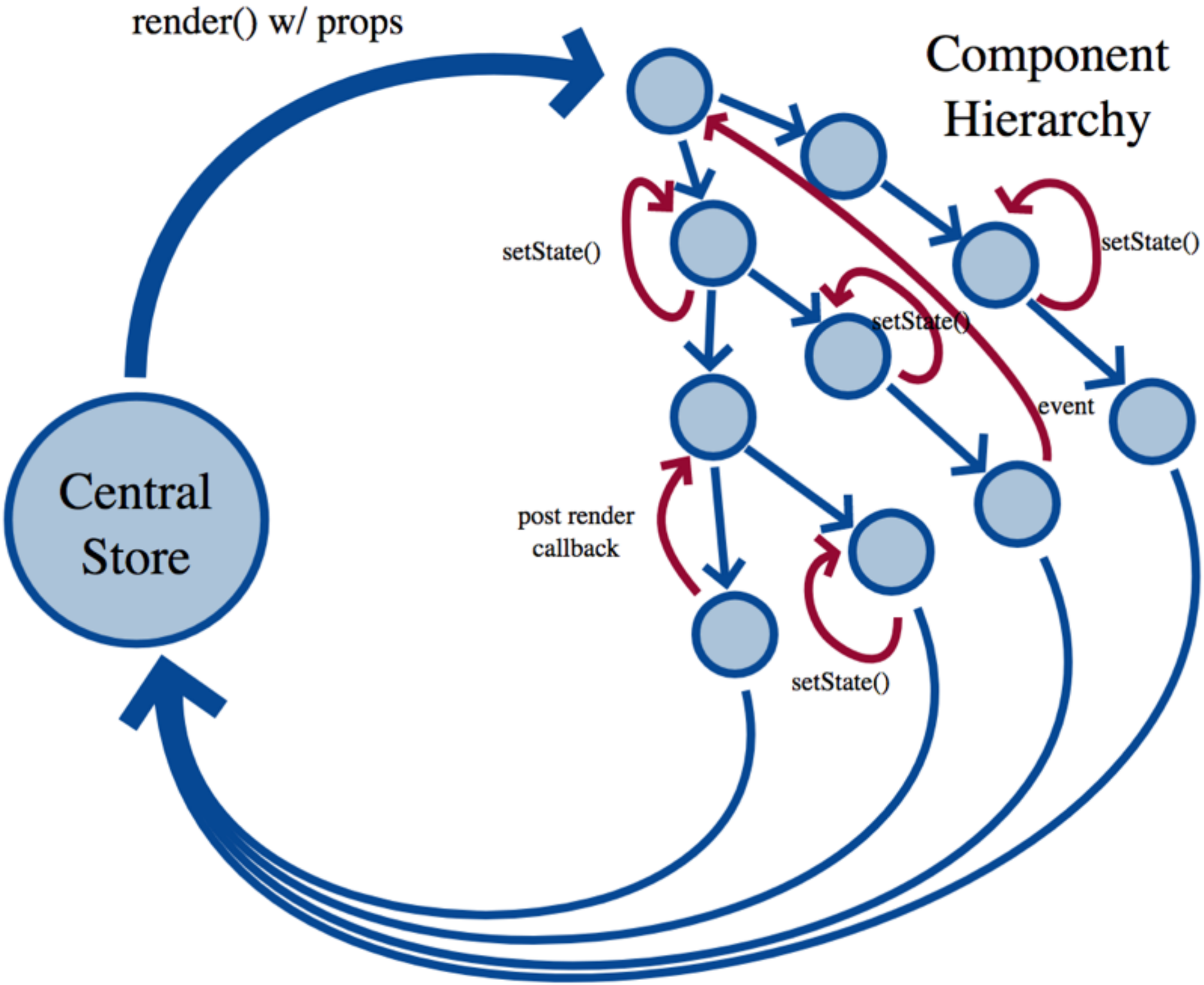

todos
00000
11111
00000
11111
00000
11111
00100
01001
01100
10000
10100
11010
10000
10010
key: t0143c274
value: { title, completed }


12
13
0
1
2
3
4
5
6
7
8
9
a
b
c


Commit
Commit
Trees
Trees
Blob
Blob
Blob
Blob
Blob


... on *efficient* immutability
x
x'
Structural
sharing


Virtual
DOM
State Change
Compute Diff
Re-render
Browser
DOM


render() w/ props
Component Hierarchy
Central Store
setState()
setState()
setState()
event
post render callback
setState()
messages/events/callbacks/etc.

树形数据

```clojure
; number
1.23

; string
"foo"

; keyword (like strings, but used as map keys)
:foo

; vector (array)
[:bar 3.14 "hello"]

; map (associative array)
{:msg "hello" :pi 3.14 :primes [2 3 5 7 11 13]}

; set (distinct elements)
#{:bar 3.14 "hello"}
```

```clojure
(def data
  [{"a" {"b" 1 "c" 2}
    "children" [{"a" {"b" 3 "c" 4} "children" []}]}
   {"a" {"b" 5 "c" 6}
    "children" []}
  {"a" {"b" 7 "c" 8}
   "children" [{"a" {"b" 9 "c" 10} "children" []}]}])
```

```
(defonce app-state
  (atom
    {:contacts
     [{:first "Ben" :last "Bitdiddle" :email "benb@mit.edu"}
      {:first "Alyssa" :middle-initial "P" :last "Hacker" :email "aphacker@mit.edu"}
      {:first "Eva" :middle "Lu" :last "Ator" :email "eval@mit.edu"}
      {:first "Louis" :last "Reasoner" :email "prolog@mit.edu"}
      {:first "Cy" :middle-initial "D" :last "Effect" :email "bugs@mit.edu"}
      {:first "Lem" :middle-initial "E" :last "Tweakit" :email
"morebugs@mit.edu"}]}))
```

```
(defn todo-item []
  (let [editing (r/atom false)]
    (fn [{:keys [id done title]}]
      [:li {:class (str (if done "completed ")
                        (if @editing "editing"))}
       [:div.view
        [:input.toggle {:type "checkbox" :checked done
                        :on-change #(toggle id)}]
        [:label {:on-double-click #(reset! editing true)} title]
        [:button.destroy {:on-click #(delete id)}]]
       (when @editing
         [todo-edit {:class "edit" :title title
                     :on-save #(save id %)
                     :on-stop #(reset! editing false)}])])))
```

```
<li className={classNames({
  completed: this.props.todo.completed,
  editing: this.props.editing
})}>
  <div className="view">
    <input
      className="toggle"
      type="checkbox"
      checked={this.props.todo.completed}
      onChange={this.props.onToggle}
    />
    <label onDoubleClick={this.handleEdit}>
      {this.props.todo.title}
    </label>
    <button className="destroy" onClick={this.props.onDestroy} />
  </div>
  <input
    ref="editField"
    className="edit"
    value={this.state.editText}
    onBlur={this.handleSubmit}
    onChange={this.handleChange}
    onKeyDown={this.handleKeyDown}
  />
</li>
```

可变状态(Atom)

```
user=> (def my-atom (atom 0))
#'user/my-atom

user=> @my-atom
0

user=> (swap! my-atom inc)
1

user=> @my-atom
1

user=> (swap! my-atom (fn [n] (* (+ n n) 2)))
4

user=> (reset! my-atom 0)
0

user=> @my-atom
0
```

```clojure
(def a (atom {}))

(add-watch a :watcher
  (fn [key atom old-state new-state]
    (prn "-- Atom Changed --")
    (prn "key" key)
    (prn "atom" atom)
    (prn "old-state" old-state)
    (prn "new-state" new-state)))

(reset! a {:foo "bar"})

;; "-- Atom Changed --"
;; "key" :watcher
;; "atom" #<Atom@4b020acf: {:foo "bar"}>
;; "old-state" {}
;; "new-state" {:foo "bar"}
;; {:foo "bar"}
```

Cursor

```clojure
(def state (atom { :color "#cc3333"
                   :user { :name "Ivan" } }))

(def user-name (rum/cursor-in state [:user :name]))

@user-name ;; => "Ivan"

(reset! user-name "Oleg") ;; => "Oleg"

@state ;; => { :color "#cc3333"
       ;;      :user  { :name "Oleg" } }
```

完结