Clojure, Concurrency, And You

Abhinav Sarkar

```
(((((IN/Clojure Bengaluru 2019)))))
```

Time

```clojure
(defn pass-time [person]
   (-> person
      (assoc :hair-color :gray)
      (update :age inc)))
```

Values and Identities

```
user=> (def abhinav {:hair-color :black :age 33})
#'user/abhinav
```

Values and Identities

```
user=> (def abhinav {:hair-color :black :age 33})
#'user/abhinav
```

```
user=> (def abhinav
  #_=>   (atom {:hair-color :black :age 33}))
#'user/abhinav
user=> (swap! abhinav pass-time)
{:hair-color :gray, :age 34}
```

Concurrency

> Concurrency is a program-structuring technique in which there are multiple threads of control which execute "at the same time".

— Simon Marlow, Parallel and Concurrent Programming in Haskell

Threads

- Thread is a sequence of instructions along with a context.
- In case of Clojure on JVM, the threading model is provided by the JVM which only supports OS threads.

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err);
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename);
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err);
        } else {
          console.log(filename + ' : ' + values);
          aspect = (values.width / values.height);
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect);
            console.log('resizing ' + filename + 'to ' + height + 'x' + height);
            this.resize(width, height).write(dest + 'w' + width + '_' + filename,
              function(err) {
                if (err)
                  console.log('Error writing file: ' + err);
              });
          }.bind(this));
        }
      });
    });
  }
});
```

```
user=> (def t (Thread. #(println "hello")))
#'user/t
user=> (.start t)
hello
nil
```

```
user=> (import java.util.concurrent.Executors)
java.util.concurrent.Executors
user=> (def tp (Executors/newSingleThreadExecutor))
#'user/tp
user=> (.submit tp #(println "hello"))
hello
#object[java.util.concurrent.FutureTask ...]
```

```
user=> (future (println "hello"))
hello
#object[clojure.core$future_call$reify__6962 ...]
```

```
user=> (future (println "hello"))
hello
#object[clojure.core$future_call$reify__6962 ...]
```

```
user=> (def f (future (do
  #_=>                  (println "hello")
  #_=>                  12345)))
hello
#'user/f
user=> (deref f)
12345
```

Synchronization

Synchronization

- The process by which multiple threads agree on *some things* at *some time*.
- For example:
  - timing: forking and joining threads
  - value of a variable
  - a sequence of steps to execute
  - access to a shared resource

˅ Time / Things >	One value	Multiple values
Synchronous	Lock, Atom	Multiple locks, Ref
Asynchronous	Agent	CRDTs, Raft/Paxos

Locks

Locks

- An easy way of synchronization.
- Prevent concurrent access to critical sections/memory.
- Do not compose.

```
user=> (def lock (Object.))
#'user/lock
user=> (locking lock
  #_=>   (println "locked hello"))
locked hello
nil
```

```
user=> (import java.util.concurrent.locks.ReentrantLock)
java.util.concurrent.locks.ReentrantLock
user=> (def lock (ReentrantLock.))
#'user/lock
user=> (try
  #_=>   (.lock lock)
  #_=>   (println "locked hello")
  #_=>   (finally
  #_=>     (.unlock lock)))
locked hello
nil
```

Atoms

Atoms

- Atoms are references which change atomically and immediately.
- Simplest of all reference types.
- Do not compose.

```
user=> (def abhinav
  #_=>   (atom {:hair-color :black :age 33}))
#'user/abhinav
user=> (swap! abhinav pass-time)
{:hair-color :gray, :age 34}
user=> (reset! abhinav {:hair-color :none :age 33})
{:hair-color :none, :age 33}
user=> @abhinav
{:hair-color :none, :age 33}
```

```
package clojure.lang;
import java.util.concurrent.atomic.AtomicReference;

final public class Atom {
  private final AtomicReference state;

  public Atom(Object o) { state = new AtomicReference(o); }

  public Object deref() { return state.get(); }

  public Object swap(IFn f) {
    for (;;) {
      Object v = deref();
      Object newv = f.invoke(v);
      if (state.compareAndSet(v, newv)) {
        notifyWatches(v, newv);
        return newv;
      }
    }
  }
}
```

Atom

- Most ubiquitous concurrency feature used in Clojure.
- Use cases: dynamic configs, database connections, simple caches.
- Do not call swap with long running or non-idempotent functions.

Agents

Agents

- Agents, like Atoms, are references which support atomic changes.
- But the changes are made in an asynchronous fashion.
- Do not compose.

```
user=> (def counter (agent 0))
#'user/counter
user=> (dotimes [i 10]
  #_=>   (send counter inc))
nil
user=> (await counter)
nil
user=> (println @counter)
10
nil
```

Agents are not Actors

```
-module(counter).
-export([loop/1]).

loop(N) -> receive
  {inc} -> loop(N+1);
  {get, Sender} -> Sender ! N, loop(N)
end.

> Pid = spawn(counter, loop, [0]).
> Pid ! {inc}.
> Pid ! {get, self()}.
> receive Value -> io:fwrite("~p~n", [Value]) end.
1
```

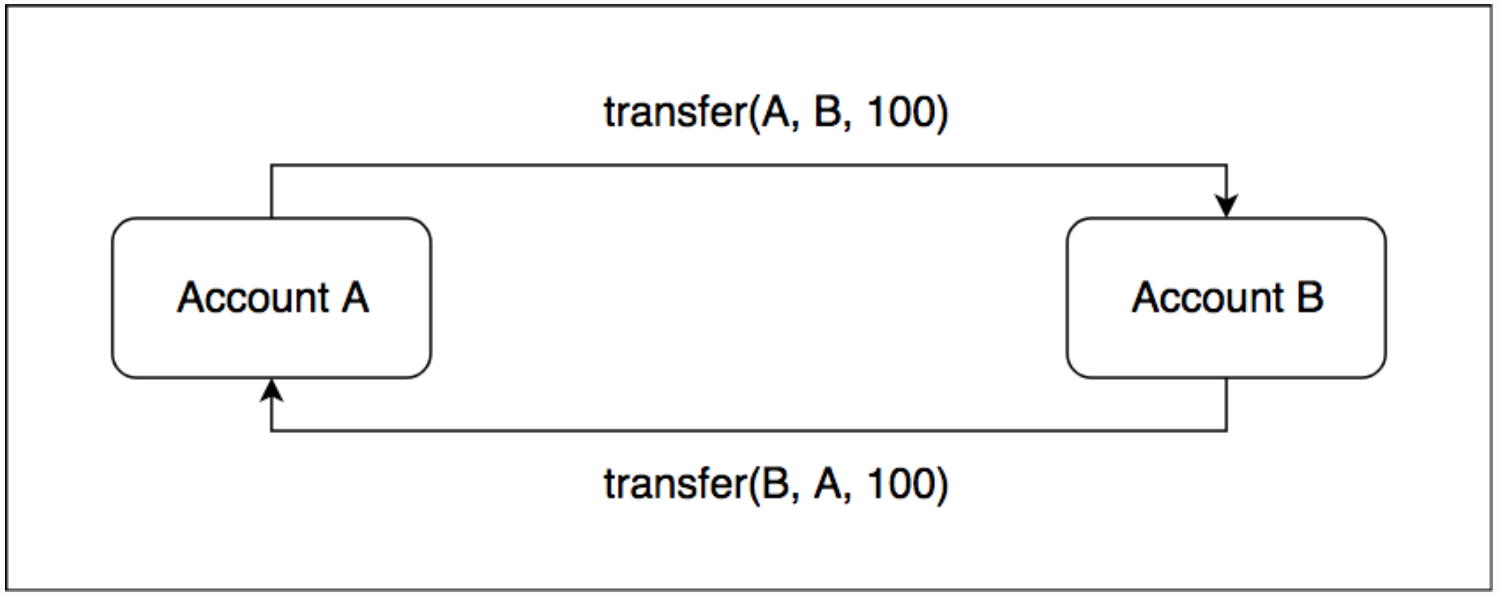
Agents

- Can be used for any state that does not require strict consistency for reads:
  - Counters (e.g. message rates in event processing)
  - Collections (e.g. recently processed events)
- Can be used for offloading arbitrary computations to a thread pool using `send-via`.
- Uses an unbounded queue, so too many functions enqueued in it may cause OOM.

Refs

Refs and Software Transactional Memory

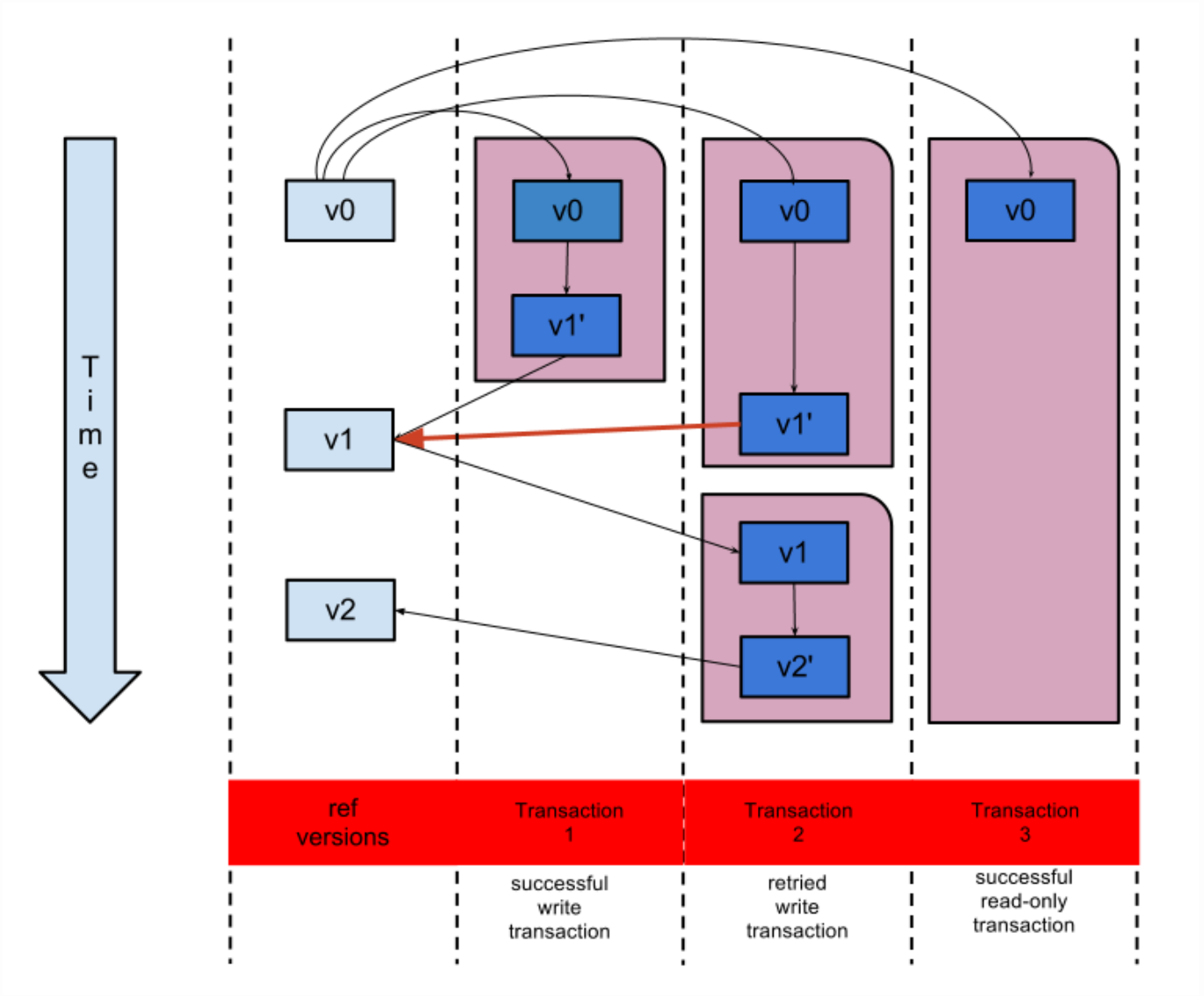
- Refs allows changing multiple references together in a single atomic operation.
- **Atomicity**: All the state changes become visible to all the threads at once.
- **Consistency**: All the state changes can be validated before allowing the transaction to commit.
- **Isolation**: The atomic operation is completely unaffected by whatever other threads are doing.


transfer(A, B, 100)
Account A
Account B
transfer(B, A, 100)

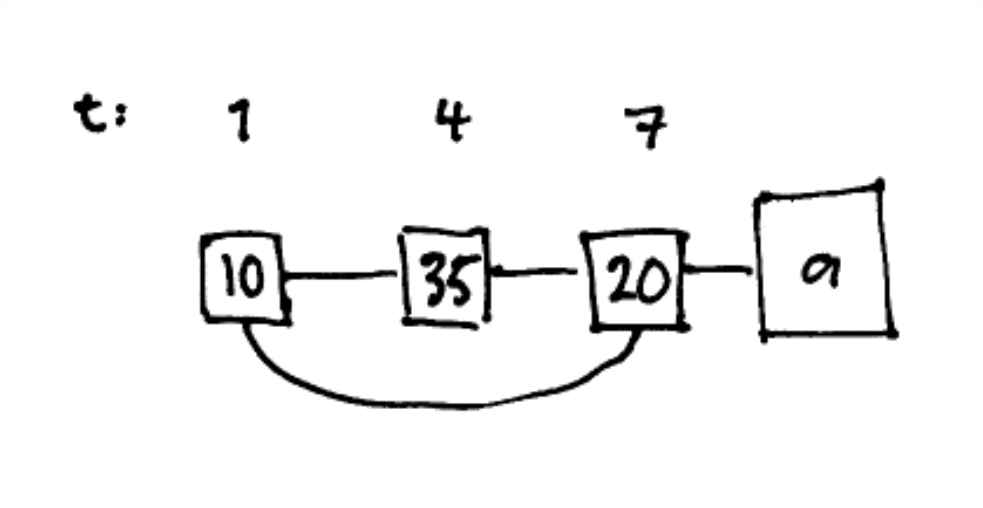
```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Account {
    private int id, amount;
    private Lock lock = new ReentrantLock();

    Account(int id, int initialAmount) {
        this.id = id;
        this.amount = initialAmount;
    }

    public void withdraw(int n) {
        this.lock.lock();
        try { this.amount -= n; }
        finally { this.lock.unlock(); }
    }

    public void deposit(int n) { this.withdraw(-n); }

    public void transfer(Account other, int n) {
        this.withdraw(n);
        other.deposit(n);
    }
}
```

```
public void transfer(Account other, int n) {
    if (this.id < other.id) {
        this.lock.lock();
        other.lock.lock();
    } else {
        other.lock.lock();
        this.lock.lock();
    }
    try {
        this.amount -= n;
        other.amount += n;
    } finally {
        if (this.id < other.id) {
            this.lock.unlock();
            other.lock.unlock();
        } else {
            other.lock.unlock();
            this.lock.unlock();
        }
    }
}
```

```clojure
(def account1 (ref 100))
(def account2 (ref 100))

(defn withdraw [account amount]
  (alter account - amount))

(defn deposit [account amount]
  (withdraw account (- amount)))

(defn transfer [from to amount]
  (dosync
    (withdraw from amount)
    (deposit to amount)))
```



Source: www.sw1nn.com/blog/2012/04/11/clojure-stm-what-why-how/



Source: mattias.niklewski.com/2014/04/stm.html

Clojure STM vs. Haskell STM

```haskell
import System.IO
import Control.Concurrent.STM

type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount = do
  bal <- readTVar acc
  writeTVar acc (bal - amount)

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically $ do
  deposit to amount
  withdraw from amount
```

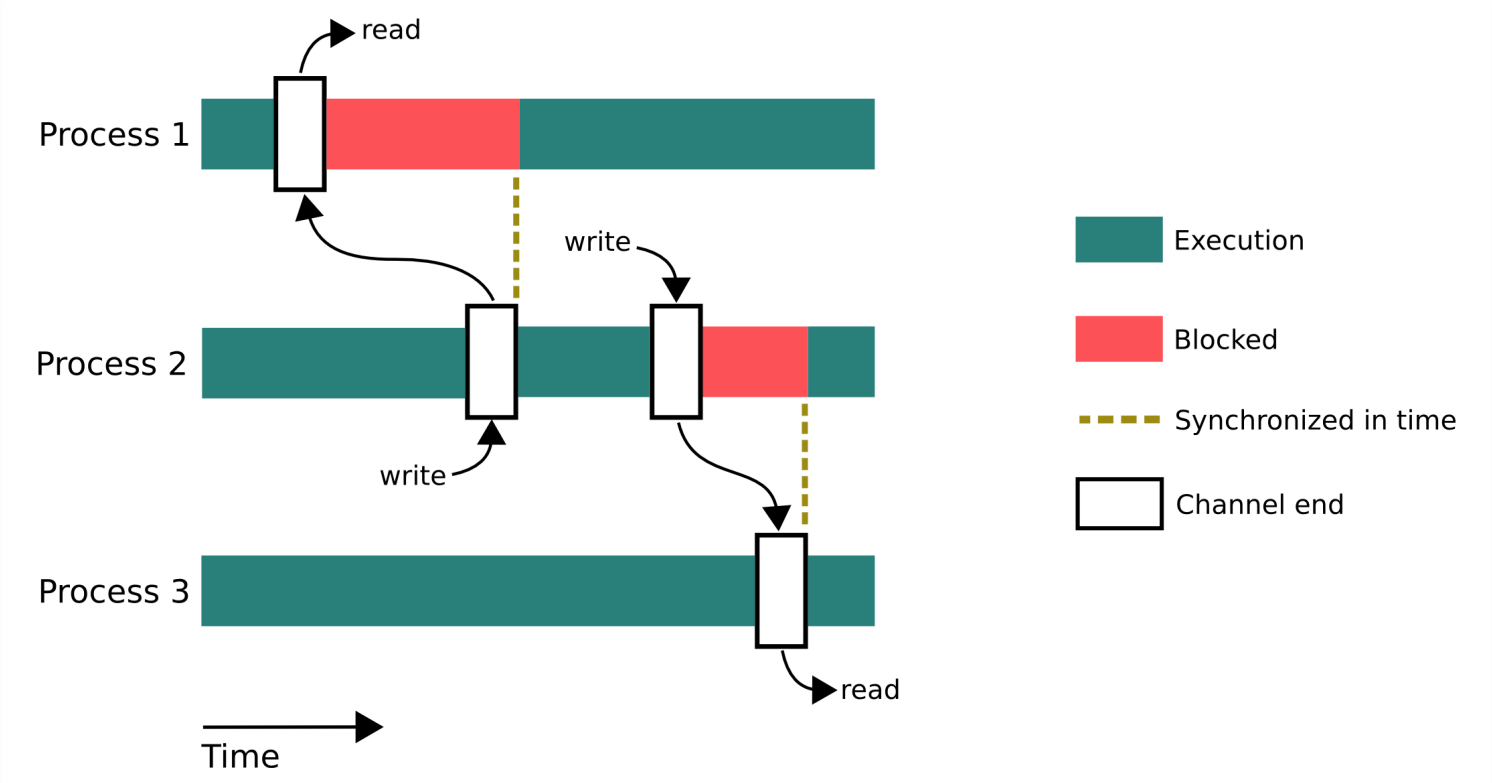
Refs

- Best option for implementing in-memory transaction data stores
  - Chat servers
  - Multiplayer games
- In-memory stream computation solutions.
- A long-running transaction may re-execute many times because it may be repeatedly aborted by shorter transactions.
- Keeping history of values is expensive. Even more so when the values are not persistent collections.

core.async

core.async and Communicating Sequential Processes

- Independent threads of activity.
- Synchronous communication through channels.
- Multiplexing of channels with alternation.



Source: arild.github.io/csp-presentation

```
(go (println "hi"))

(def echo-chan (chan))
(go (println (<! echo-chan)))
(go (>! echo-chan "hello"))
; => hello

(def echo-chan (chan 10))
```

```clojure
(let [c1 (chan)
      c2 (chan)
      c3 (chan)]
  (dotimes [n 3]
    (go
      (let [[v ch] (alts! [c1 c2 c3])]
        (println "Read" v "from" ch))))
  (go (>! c1 "hello"))
  (go (>! c2 "allo")
  (go (>! c2 "hola")))
; => Read allo from #<ManyToManyChannel ...>
; => Read hola from #<ManyToManyChannel ...>
; => Read hello from #<ManyToManyChannel ...>
```

Multiplexing

```
user=> (def input (chan 1))
#'user/input
user=> (def broadcast (mult input))
#'user/broadcast
user=> (dotimes [i 3]
  #_=>   (let [output (chan 1)]
  #_=>     (tap broadcast output)
  #_=>     (go-loop []
  #_=>       (if-let [v (<! output)]
  #_=>         (do (println i "Got!" v)
  #_=>             (recur))
  #_=>         (println "Exiting")))))
nil
```

Multiplexing

```
user=> (>!! input 42)
true0 Got! 421

2 Got! Got!42
42
user=> (>!! input 43)
true
01  Got!Got!  4343
2
Got! 43
user=> (close! input)
Exitingnil

Exiting
Exiting
```

Publish-Subscribe

```
user=> (def input (chan 1))
#'user/input
user=> (def p (pub input :tag))
#'user/p
user=> (let [c (chan 1)]
  #_=>   (sub p :cats c)
  #_=>   (go-loop []
  #_=>     (if-let [v (<! c)]
  #_=>       (println "Cat guy got:" v)
  #_=>       (println "Cat guy exiting"))))
#object[clojure.core.async.impl.channels.ManyToManyChannel ...]
user=> (let [c (chan 1)]
  #_=>   (sub p :dogs c)
  #_=>   (go-loop []
  #_=>     (if-let [v (<! c)]
  #_=>       (println "Dog guy got:" v)
  #_=>       (println "Dog guy exiting"))))
#object[clojure.core.async.impl.channels.ManyToManyChannel ...]
```

Publish-Subscribe

```
user=> (defn send-with-tags [msg]
  #_=>   (doseq [tag (:tags msg)]
  #_=>     (println "sending... " tag)
  #_=>     (>!! input {:tag tag :msg (:msg msg)})))
#'user/send-with-tags
user=> (send-with-tags {:msg "New Cat Story" :tags [:cats]})
sending...  :cats
nil
Cat guy got: {:tag :cats, :msg New Cat Story}
user=> (send-with-tags {:msg "New Dog Story" :tags [:dogs]})
sending...  :dogs
nil
Dog guy got: {:tag :dogs, :msg New Dog Story}
```

Publish-Subscribe

```
user=> (let [c (chan 1)]
  #_=>   (sub p :dogs c)
  #_=>   (sub p :cats c)
  #_=>   (go-loop []
  #_=>     (if-let [v (<! c)]
  #_=>       (println "Cat/Dog guy got:" v)
  #_=>       (println "Cat/Dog guy exiting"))))
#object[clojure.core.async.impl.channels.ManyToManyChannel]
```

Pipelines

```
(pipeline n to xf from close? ex-handler)
(pipeline-async n to af from)
(pipeline-blocking n to xf from close? ex-handler)
```

core.async vs Goroutines

```
func fibonacci(c, q chan int) {
  x, y := 0, 1
  for {
    select {
    case c <- x:
      x, y = y, x+y
    case <-q:
      fmt.Println("quit")
      return
    }
  }
}
```

```
func main() {
  c := make(chan int)
  quit := make(chan int)
  go func() {
    for i := 0; i < 10; i++ {
      fmt.Println(<-c)
    }
    quit <- 0
  }()
  fibonacci(c, quit)
}
```

core.async

- Data transformation pipelines like ETL.
- Multi-user chat servers, game servers.
- More broadly, Staged Event Driven Architecture programs.
- With Clojurescript, it is a great replacement for callback for UI interactions.

core.async

- Data transformation pipelines like ETL.
- Multi-user chat servers, game servers.
- More broadly, Staged Event Driven Architecture programs.
- With Clojurescript, it is a great replacement for callback for UI interactions.
- Doing blocking IO in go-threads blocks them.
- Error handling is complicated.
- Too many pending puts or take may throw errors.

References

- Clojure reference documentation
- Clojure STM: What, why, how?
- Thoughts on STM
- Implementation details of core.async Channels
- Core Async Go Macro Internals
- Publish and Subscribe with core.async

Thanks

@abhin4v

abnv.me/ccy