Spec

— Cloujre & ClojureScript —

- Why Need It ?
- Defining Specs
- Generators
- Using Specs

Beibei Kang
2019.9.6

Why Need It ?

1. Docs are not enough
2. Dynamic feature, no type checking
3. Generative testing and robustness
4. A standard approach is needed

Reference: https://www.clojure.org/about/spec#_problems

Defining Specs

- **Get Start**

  - Clojure

```clojure
(require '[clojure.spec.alpha :as s])
```

```clojure
(ns spec-test.basic
  (:require [clojure.spec.alpha :as s]))
```

  - ClojureScript

```clojure
(require '[cljs.spec.alpha :as s])
```

```clojure
(ns spec-test.basic
  (:require [cljs.spec.alpha :as s]))
```

Defining Specs

➢ Basic

- Simple Usage

Format: `(validate-func spec value)`

```
(s/conform even? 1000)
 ;;=> 1000
(s/valid? even? 1000)
;;=> true
(s/explain even? 1000)
;;=> Success
```

Note:

- s/conform, s/valid, s/explain, some functions to validate a value whether meets a spec.
- even? A simple predicate(boolean function), will be convert to spec when used.

```
(s/valid? #(> % 5) 10) ;; true
(s/valid? #(> % 5) 0) ;; false
```

Note:

- Use some customize boolean functions

```
(s/valid? #{:club :diamond :heart :spade} :club) ;; true
(s/valid? #{:club :diamond :heart :spade} 42) ;; false
```

Note:

- Use function object.

Defining Specs

- **Basic**

  - Registry – define reusable spec

Format: `(s/def keyword spec)`

```
(s/def ::date inst?)
(s/def ::suit #{:club :diamond :heart :spade})
```

Note:
- The keyword must use format such as ”::key-name”, which with “::”

```
;; unqualified namespace
(s/def ::date inst?)
(s/def ::suit #{:club :diamond :heart :spade})

;; qualified namespace
(s/def :animal/dog #{:name :age})
```

Note:
- Unqualified namespace – resolved to present namspace
- Qualified namespace – resolved to qualified namespace

See slide 8

Defining Specs

➢ **Basic**

- Validate/Check

Format:
```
(s/valid? spec value)
(s/conform spec value)
(s/explain spec value)
(s/explain-date spec value)
```

```
(s/valid? ::date (java.util.Date.))
;;=> true
(s/valid? ::date 42)
;;=> false

(s/conform ::suit :club)
;;=> :club
(s/conform ::suit "like")
;;=> :clojure.spec.alpha/invalid

(s/explain ::date (java.util.Date.))
;;=> Success!
(s/explain ::date 42)
;;=> 42 - failed: inst? spec: :user/date
```

Reference quickly:
https://clojure.org/api/cheatsheet
http://cljs.github.io/api/cljs.spec.alpha/
https://www.clojure.org/guides/spec#_explain

Defining Specs

➢ **Basic**

- Unqualified Namespace and Qualified Namepsace

`Use in the same namespace`

```
(ns spect-test.basic)

;; unqualified namespace
(s/def ::suit #{:club :diamond :heart :spade})

;; qualified namespace
(s/def :animal/dog #{:name :age})

;;; Use namespace qualified and unqualified
(s/explain ::suit :like)
(s/explain :spec-test.basic/suit :like)

(s/explain :animal/dog :apple)
```

Defining Specs

- **Basic**

  - Unqualified Namespace and Qualified Namepsace

```
Use in the different namespace
```

```
;;; Define
(ns spect-test.basic)
;; unqualified namespace
(s/def ::suit #{:club :diamond :heart :spade})
;; qualified namespace
(s/def :animal/dog #{:name :age})

;;; Use
(ns spec-test.core
  (:require [spec-test.basic])) ; must require
;;; Use namespace qualified and unqualified
(s/explain :spec-test.basic/suit :like)

(s/explain :animal/dog :apple)
```

```
(ns spec-test.core
  (:require [spec-test.basic :as sb]))
(s/explain ::sb/suit :like)

(s/explain :animal/dog :apple)
```

Note:
  Some dependencies don’t require.
  Plead add them.

Defining Specs

- **Composing predicates**

```
Format: (s/and predicate1 predicate2 ...)
        (s/or predicate1 predicate2 ...)
```

```
(s/def ::big-even (s/and int? even? #(> % 1000)))
(s/conform ::big-even :foo) ;; :clojure.spec.alpha/invalid
(s/conform ::big-even 10) ;; :clojure.spec.alpha/invalid
(s/conform ::big-even 100000) ;; true

;;; return a destruct data structure - a map entry.
(s/def ::name-or-id (s/or :name string?
                          :id   int?))
(s/conform ::name-or-id "abc") ;; [:name "abc"]
(s/conform ::name-or-id 100) ;; [:id 100]
(s/conform ::name-or-id :foo) ;; :clojure.spec.alpha/invalid
```

Defining Specs

- **Composing spec**

```
Format: (s/def keyword (s/keys :req [spec1 spec2 spec3]
                               :opt [spec1 spec2]))
        (s/def keyword (s/keys :req-un [spec1 spec2 spec3]
                               :opt-un [spec1 spec2]))
```

→ Qualified spec namespace

→ Unqulified spec namespace

```
(def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/def ::email-type (s/and string? #(re-matches email-regex %)))

(s/def ::phone int?)
(s/def ::first-name string?)
(s/def ::last-name string?)
(s/def ::email ::email-type)

(s/def ::person (s/keys :req [::first-name ::last-name ::email]
                        :opt [::phone]))

(s/def :unq/person
  (s/keys :req-un [::first-name ::last-name ::email]
          :opt-un [::phone]))
```

Rerence: http://clojuredocs.org/clojure.spec.alpha/keys

s/keys only validate the value that belongs to some kewords in :req and :opt

Defining Specs

➢ Composing spec

```
Format: (s/def keyword (s/keys :req [spec1 spec2 spec3]
                               :opt [spec1 spec2]))
        (s/def keyword (s/keys :req-run [spec1 spec2 spec3]
                               :opt-run [spec1 spec2]))
```

→ Qualified spec namespace

→ Unqulified spec namespace

```
;;; test in spec-test.compose-spec namespace
(s/explain ::person
          {::first-name :name
           ::last-name "Bunny"
           ::email "bugs@example.com"})
;; :name -
;; failed: string?
;; in: [:spec-test.compose-spec/first-name]
;; at: [:spec-test.compose-spec/first-name]
;; spec: :spec-test.compose-spec/first-name
```

```
;;; test in spec-test.compose-spec namespace
(s/explain :unq/person
          {:first-name :name
           :last-name "Bunny"
           :email "bugs@example.com"})
;; :name -
;; failed: string?
;; in: [:first-name]
;; at: [:first-name]
;; spec: :spec-test.compose-spec/first-name
```

Note: Must use "::keyword" format as value key
Error spec have a namesapce

Note: Must use ":keyword" format as value key
Error spec have no namesapce

Defining Specs

- **Mutil-spec**

Polymorphism method of spec.

```
;; Some functions

(defmulti name docstring? attr-map? dispatch-fn & options)
;; name:  mutil-fn name
;; docstring and attr-map: optional.
;; dispatch-fn: get dispatch val and return it.
;;              Then the corresponding function will be called
;; options: key-vals, such as
;;   - :default     default dispatch-val to handler args. If haven’t, then use “:default”.
;;   - :hierarchy   use it to create a hierarchy.

(defmethod multifn dispatch-val & fn-tail)
;; multifn:  the `name` in defmulti
;; dispatch-val: the return-val of dispatch-fn. If no, use :default
;; fn-tail: handler function。
```

Defining Specs

- **Mutil-spec**

Polymorphism method of spec.

```
;; Some functions

(multi-spec mm retag)

;; mm: multimethod fn name
;; retag: as `disaptch-val` keyword to use `assoc` to merge in generator generated
smaples.
```

Defining Specs

- **Mutil-spec**

  Polymorphism method of spec.

```clojure
;; register some spec
(s/def :event/type keyword?)
(s/def :event/timestamp int?)
(s/def :search/url string?)
(s/def :error/message string?)
(s/def :error/code int?)

;; implement the multimethod of spec
;; we use :type,
(defmulti event-type :type)
(defmethod event-type :search [_]
  (s/keys :req [:event/type :event/timestamp :search/url]))
(defmethod event-type :error [_]
  (s/keys :req [:event/type :event/timestamp :error/message :error/code]))

;; define mutil-spec
(s/def :event/event (s/multi-spec event-type :type))

;; generator some sample for it
(gen/sample (s/gen :event/event))
```

Defining Specs

- **Mutil-spec**

Polymorphism method of spec.

```
;; do some test

(s/valid? :event/event
          {:type :search
           :event/timestamp 1463970123000
           :search/url "https://clojure.org"})
;=> true
(s/valid? :event/event
          {:type :error
           :event/timestamp 1463970123000
           :error/message "Invalid host"
           :error/code 500})
;=> true
```

Defining Specs

➢ **Specs for collection**

There are some functions to define spec for collections directly.
Other than `map-of` and `tuple`, all have :into option.

```
;; map-of and coll-of work for whole coll.
(s/conform (s/coll-of keyword?) [:a :b :c])
;;=> [:a :b :c]
(s/def ::scores (s/map-of (s/or :name string? :like int?) int? :conform-keys true))
(s/conform ::scores {10 1000, "Joe" 500})
;;=> {[:like 10] 1000, [:name "Joe"] 500}

;; tuple is one to one to validate. Every entry has its own spec.
(s/def ::point (s/tuple double? double? string?))
(s/conform ::point [1.5 2.5 -0.5])
;;=> :clojure.spec.alpha/invalid

;;every-of is the same as map
(s/def ::person (s/every-kv keyword? string? ))
(gen/sample (s/gen ::person))

;; every do *coll-check-limit*
(s/def ::animal (s/every keyword?))
(gen/sample (s/gen ::animal))
```

Defining Specs

➢ **Specs for range**

`Ofen use `int-in`, `inst-in`, and `double-in`. The `int-in`, `inst-in` start inclusive and end exclusive; the `double-in` all inclusive`

```
(s/def ::roll (s/int-in 0 11))
(s/valid? ::roll 3)
;;=> true

(s/def ::the-aughts (s/inst-in #inst "2000" #inst "2010"))
(s/valid? ::the-aughts #inst"2005-03-03T08:40:05.393-00:00")
;; => true

(s/def ::dubs (s/double-in :min -100.0 :max 100.0 :NaN? false :infinite? false))
(s/valid? ::dubs 2.9)
;;=> true
(s/valid? ::dubs Double/POSITIVE_INFINITY)
;;=> false

;; More usage to see generator
```

Reference: https://clojure.org/api/cheatsheet

Defining Specs

- **Regx-spec**

```
This is for sequential data type.(list, vector, range, and (seq ...))
```

- cat - concatenation of predicates/patterns
- alt - choice among alternative predicates/patterns
- * - 0 or more of a predicate/pattern
- + - 1 or more of a predicate/pattern
- ? - 0 or 1 of a predicate/pattern

Defining Specs

- **Regx-spec**

  This is for sequential data type.(list, vector, range, and (seq ...))

```
;; desturct the args
(s/def ::config (s/*
                  (s/cat :prop string?
                         :val  (s/alt :s string? :b boolean?))))
(s/conform ::config ["-server" "foo" "-verbose" true "-user" "joe"])
;;=> [{:prop "-server", :val [:s "foo"]}
;;    {:prop "-verbose", :val [:b true]}
;;    {:prop "-user", :val [:s "joe"]}]


;; s/& which is the same as and
(s/def ::even-strings (s/& (s/* string?) #(even? (count %))))
(s/valid? ::even-strings ["a"])  ;; false
(s/valid? ::even-strings ["a" "b"])  ;; true
(s/valid? ::even-strings ["a" "b" "c"])  ;; false
(s/valid? ::even-strings ["a" "b" "c" "d"])  ;; true
```

Generator

- **Start**

In Leiningen, add it to project.clj

```
:profiles {:dev {:dependencies [[org.clojure/test.check "0.9.0"]]}}
;; we should use it in development mode.
```

And then, refer it in your namespace:

```
(require '[clojure.spec.gen.alpha :as gen])
```

Next, add `-main` for your project:

```
(defn -main [& args]
  (println (gen/sample (s/gen int?))))
```

Finally, run it with leingen:

```
lein with-profile dev run
```

Reference: use `lein help profiles` to see more

Generator

- **Start**

`Attention:`

Only use it in a project evironment,

- fist way, create a new project, and then add dependency, run

- use leiningen, `lein repl` in the project, then it will inject the evironment of the project.

- `clojure.spec.gen.alpha` is a simple wrap of `clojure.test.check.generators`

- just use `clojure.test.check.generators` is okay.

Note:
can’t require nonnative clojure dependency in `repl` cli directly.

Generator

- **Basic Usage**

Note:(All default generate 10 group)
gen/sample : generate samples
s/exercise: generate samples and `s/conform` result

Format: `(gen/sample (s/gen spec))`
`(s/exercise spec)`

```
;; use predicate
(gen/sample (s/gen int?))
;; (0 0 1 0 -2 -16 -5 -37 1 -1)

;; use spec
(gen/sample (s/gen (s/cat :k keyword? :ns (s/+ number?))) 5)
;;=> ((:D -2.0)
;;=>  (:q4/c 0.75 -1)
;;=>  (:*!3/? 0)
;;=>  (:+k_?.p*K.*o!d/*V -3)
;;=>  (:i -1 -1 0.5 -0.5 -4))

;; Use exercise
(s/exercise (s/or :k keyword? :s string? :n number?) 5)
;;=> ([:H [:k :H]]
;;     [:ka [:k :ka]]
;;     [-1 [:n -1]]
;;     ["" [:s ""]]
;;     [-3.0 [:n -3.0]])
```

Generator

- **Basic Usage**

Note:(Two usages)
- First, avoid predicate no mapping to generator
- Second, refine data.

Format: `(gen/sample (s/gen (s/and spec1 spec2 ...))`

```
;; No mapping generator for even?. Maybe have more
(gen/generate (s/gen even?))
;; Execution error (ExceptionInfo) at user/eval1281 (REPL:1).
;; Unable to construct gen at: [] for: clojure.core$even_QMARK_@73ab3aac

;; Use s/and. First spec used by generator; rest used as filter.

(gen/generate (s/gen (s/and int? Even?)))
;;=> -15161796


;; Use to refine generated data
(gen/sample (s/gen (s/and int?
                          #(> % 0)
                          (divisible-by 3))))
;;=> (3 9 1524 3 1836 6 3 3 927 15027)
```

Note:
- gen/generate: return 1 group smaple
- gen/sample: return a lot of samples

Generator

➢ **Custom Generators**

Three way to define custom generators:
(preference decreased)

- **Let spec create a generator based on a predicate/spec**
- **Create your own generator from the tools in clojure.spec.gen.alpha**
- **Use test.check or other test.check compatible libraries (like test.chuck)**

Generator

- **Custom Generators**

First way, we have seen a lot of examples.

```
;; examples
;; s/gen will return a generator.
(s/gen even?)

(s/gen (s/and int? Even?))

(s/gen (s/or int? String?))

…
```

Generator

- **Custom Generators**

Second way, need to see more API about clojure.spec.gen.alpha.

```
;; A example generator a qualified namspace keyword
;; namespace with `my.domain`
;; simple but invalid
(s/def ::kws (s/and keyword? #(= (namespace %) "my.domain")))
(s/valid? ::kws :my.domain/name) ;; true
(gen/sample (s/gen ::kws)) ;; unlikely we'll generate useful keywords this way

;; improve it
(def kw-gen-3 (gen/fmap #(keyword "my.domain" %)
                        (gen/such-that #(not= % "")
                                       (gen/string-alphanumeric))))
(gen/sample kw-gen-3 5)
;;=> (:my.domain/O :my.domain/b :my.domain/ZH :my.domain/31 :my.domain/U)
```

Reference: https://clojure.github.io/test.check/clojure.test.check.generators.html
https://clojure.org/api/cheatsheet (spec part)

Generator

- **Custom Generators**

Second way, need to see more API about clojure.spec.gen.alpha.

```
;; A example generator generate string include “hello”
;; namespace with `my.domain`
(s/def ::hello
  (s/with-gen #(clojure.string/includes? % "hello")
              #(gen/fmap (fn [[s1 s2]] (str s1 "hello" s2))
                         (gen/tuple (gen/string-alphanumeric) (gen/string-alphanumeric)))))
(gen/sample (s/gen ::hello))
;;=> ("hello" "ehello3" "eShello01" "vhello31p" "hello" "1Xhellow" "S5bhello"
"aRejhellorAJ7Yj" "3hellowPMDOgv7" "UhelloIx9E")
```

Third way, You may see an example lib in https://github.com/gfredericks/test.chuck.

Reference: https://clojure.github.io/test.check/clojure.test.check.generators.html
https://clojure.org/api/cheatsheet (spec part)

Usage

- **Outline View**

Some application aspects of spec:

- **Validate/Check the correctness of data type, data structure**
- **Validate/Check the correctness of function args, return value, and logical handling.**
- **Destruct the data**

Usage

- **Validate Data**

We have seen a lot of them, such as checking for map, coll, set, and other nested data.

```
;; validate a vector
(s/conform (s/coll-of keyword?) [:a :b :c])

;; validate a map
(s/def ::scores (s/map-of (s/or :name string? :like int?) int? :conform-keys
true))
(s/conform ::scores {10 1000, "Joe" 500})
;; => {"Sally" 1000, "Joe" 500}

;; validate range data
(s/def ::dubs (s/double-in :min -100.0 :max 100.0 :NaN? false :infinite?
false))
(s/valid? ::dubs 2.9)
;; => true
```

Usage

- **Validate function**

Format:
```
(fn name [& args]
      {:pre [expr]   ;expr is false, then throw an exception; if not, exec fn
       :post [expr]} ;expr is false, then throw an exception; if not, return
                            value or end the fn.
```

```
(defn person-name
  [person]
  {:pre [(s/valid? :unq/person person)]
   :post [(s/valid? string? %)]}
  (str (:first-name person) " " (:last-name person)))

;; can’t capture exception out of function. You may need:
(defn person-name
  [person]
  ;; is [expr]  is expr is false, println fail info and return false; else
return true.
  {:pre [(t/is (s/valid? :unq/person person))]
   :post [(t/is (s/valid? string? %))]}
  (str (:first-name person) " " (:last-name person)))
```

Usage

- **Validate function**

Format: `(s/assert spec value)`

```
(defn person-name-assert
  [person]
  ;; s/assert fail, throw an Exception; success, return the value.
  (let [p (s/assert ::person person)]
    (str (::first-name p) " " (::last-name p))))

;; open the assert functionality
(s/check-asserts true)
(try (person-name-assert 42)
     (catch Exception e (println (.getMessage e))))
```

Usage

- **Validate function**

  `Use spec tool to define fn spec validate`

```
;; Normal function
(defn ranged-rand
  "Returns random int in range start <= rand < end"
  [start end]
  (+ start (long (rand (- end start)))))
;; :args   spec for args
;; :ret    spec for return val
;; :fn     validate the relation between args and ret-val
(s/fdef ranged-rand
        :args (s/and (s/cat :start int? :end int?)
                     #(< (:start %) (:end %)))
        :ret int?
        :fn (s/and #(>= (:ret %) (-> % :args :start))
                   #(< (:ret %) (-> % :args :end))))
```

Usage

- **Validate function**

    Use spec tool to define fn spec validate

```
;; High order function, use fspec
(defn adder [x] #(+ x %))
(s/fdef adder
        :args (s/cat :x number?)
        :ret (s/fspec :args (s/cat :y number?)
                      :ret number?)
        :fn #(= (-> % :args :x) ((:ret %) 0)))

;; open test for args
(stest/instrument `ranged-rand)
(try (ranged-rand 8 7)
     (catch Exception e (println (.getMessage e))))
```

Usage

- **Validate function**

Use generator to do checking

```
; (stest/check `ranged-rand) ; return a lazy coll
(println (stest/check `ranged-rand))
;;=> ({:spec #object[clojure.spec.alpha$fspec_impl$reify__13728 ...],
;;     :clojure.spec.test.check/ret {:result true, :num-tests 1000, :seed
1466805740290},
;;     :sym spec.examples.guide/ranged-rand,
;;     :result true})

;; to get an abbreviated result
(println (stest/abbrev-result (first (stest/check `ranged-rand))))
;; to get a summarize result
(println (stest/summarize-results (stest/check `ranged-rand)))

; (s/exercise-fn `ranged-rand)
(println (s/exercise-fn `ranged-rand))
;;=> ([(-1 0) -1] [(-1 0) -1] [(-22 6) -15]
;;    [(-22 -1) -9] [(-3 -1) -3] [(-12 -10) -12]
;;    [(-8 12) -5] [(-226 -37) -100] [(-85 50) -45]
;;    [(-1 53) 6]])
```

Usage

- **Validate function**

Use spec generative value alter function

```
(defn invoke-service [service request]
  ;; invokes remote service
  )

(defn run-query [service query]
  (let [{::keys [result error]} (invoke-service service {::query query})]
    (or result error)))

(s/def ::query string?)
(s/def ::request (s/keys :req [::query]))
(s/def ::result (s/coll-of string? :gen-max 3))
(s/def ::error int?)
(s/def ::response (s/or :ok (s/keys :req [::result])
                        :err (s/keys :req [::error])))
```

Usage

- **Validate function**

`Use spec generative value alter function`

```
(s/fdef invoke-service
        :args (s/cat :service any? :request ::request)
        :ret ::response)

(s/fdef run-query
        :args (s/cat :service any? :query string?)
        :ret (s/or :ok ::result :err ::error))

(stest/instrument `invoke-service {:stub #{`invoke-service}})
(stest/summarize-results (stest/check `run-query))
;;=> {:total 1, :check-passed 1}
```

Usage

- **Validate macro**

```
Use `fdef` as a function, but unnecessary for `instrument`
```

```
(s/fdef clojure.core/declare
        :args (s/cat :names (s/* simple-symbol?))
        :ret any?)

(declare 100)
;; Syntax error macroexpanding clojure.core/declare at (REPL:1:1).
;; 100 - failed: simple-symbol? at: [:names]
```

Learn More

- **For spec**

  Spec API: https://clojure.github.io/spec.alpha/

  Spec cheatsheet:https://clojure.org/api/cheatsheet(search “spec” get)

- **For spec librarise**

  Schema: https://github.com/plumatic/schema

  Herbert : https://github.com/miner/herbert

  Test.check: https://github.com/clojure/test.check

  Test.check Guide: https://www.clojure.org/guides/test_check_beginner

- **For test**

  Clojure.test: https://clojure.github.io/clojure/clojure.test-api.html

- **For blogs**

  Spec: https://blog.taylorwood.io/2017/10/15/fspec.html

Learn More

- **Examples**

  PPT examples: https://github.com/kangbb/clojure-spec-examples