Why does Clojure distinguish between symbols and vars?

I saw this question already, but it doesn't explain what I am wondering about.

When I first came to Clojure from Common Lisp, I was puzzled why it treats symbols and keywords as separate types, but later I figured it out, and now I think it is a wonderful idea. Now I am trying to puzzle out why symbols and vars are separate objects.

As far I know, Common Lisp implementations generally represent a "symbol" using a structure which has 1) a string for the name, 2) a pointer to the symbol's value when evaluated in function call position, 3) a pointer to its value when evaluated outside call position, and 4) property list, etc.

Ignoring the Lisp-1/Lisp-2 distinction, the fact remains that in CL, a "symbol" object points directly to its value. In other words, CL combines what Clojure calls a "symbol" and a "var" in a single object.

In Clojure, to evaluate a symbol, first the corresponding var must be looked up, *then* the var must be dereferenced. Why does Clojure work this way? What benefit could there possibly be from such a design? I understand that vars have certain special properties (they can be private, or const, or dynamic...), but couldn't those properties simply be applied to the symbol itself?

clojure lisp symbols

asked Jul 26 '12 at 3:49
Alex D
18.3k 3 33 74

1 Not all symbols are evaluated to a var. For example the symbol `String` evaluates to a class. – Alex Taggart Jul 26 '12 at 4:08

2 Note also that symbols resolve at compile-time, and vars exist at runtime. – amalloy Jul 26 '12 at 4:13

@AlexTaggart, excellent point -- I wonder if that might be the correct answer. – Alex D Jul 26 '12 at 5:19

@amalloy, that may be true -- but it could just as easily be the other way. That doesn't explain *why* the designer of Clojure chose to make a break with previous Lisps. – Alex D Jul 26 '12 at 5:20

@amalloy With the compile-time vs runtime comment, is the thought behind it that symbols would exist for the sake of flexible and readable compile-time manipulations and that vars would exist for the sake of runtime speed? Or am I simply misinterpreting? – Omri Bernstein Jul 26 '12 at 7:07

7 Answers

Other questions have touched on many true aspects of symbols, but I'll try explaining it from another angle.

Symbols are names

Unlike most programming languages, Clojure makes a distinction between *things* and the *names* of things. In most languages, if I say something like `var x = 1`, then it is correct and complete to say "x is 1" or "the value of x is 1". But in Clojure, if I say `(def x 1)`, I've done *two* things: I've created a Var (a value-holding entity), and I've *named* it with the symbol `x`. Saying "the value of x is 1" doesn't quite tell the whole story in Clojure. A more accurate (although cumbersome) statement would be "the value of the var named by the symbol x is 1".

Symbols themselves are just names, while vars are the value-carrying entities and don't themselves have names. If extend the earlier example and say `(def y x)` , I haven't created a new var, I've just given my existing var a second name. The two symbols `x` and `y` are both names for the same var, which has the value of 1.

An analogy: my name is "Luke", but that isn't identical with me, with who I am as a person. It's just a word. It's not impossible that at some point I could change my name, and there are many other people that share my name. But in the context of my circle of friends (in my namespace, if you will), the word "Luke" means me. And in a fantasy Clojure-land, I could be a var carrying around a value for you.

But why?

So why this extra concept of names as distinct from variables, rather than conflating the two as most languages do?

For one thing, not all symbols are bound to vars. In local contexts, such as function arguments or let bindings, the value referred to by a symbol in your code isn't actually a var at all - it's just a local binding that will be optimized away and transformed to raw bytecode when it hits the compiler.

Most importantly, though, it's part of Clojure's "code is data" philosophy. The line of code `(def x 1)` isn't just an expression, it's also data, specifically a list consisting of the values `def` , `x` , and `1` . This is very important, particularly for macros, which manipulate code as data.

But *if* `(def x 1)` is a list, than what are the values in the list? And particularly, what are the *types* of those values? Obviously `1` is a number. But what about `def` and `x` ? What is their *type*, when I'm manipulating them as data? The answer, of course, symbols.

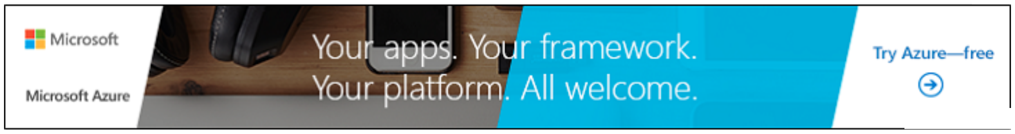
And that is the main reason symbols are a distinct entity in Clojure. In some contexts, such as macros, you want to take names and manipulate them, divorced from any particular meaning or binding granted by the runtime or the compiler. And the names must be some sort of thing, and the sort of thing they are is symbols.

answered Jul 27 '12 at 2:34
levand
4,608 1 20 45

Very nicely put. I'm divided on whether to accept this or my own answer. – Alex D Jul 28 '12 at 4:31

1 This is a great answer, but doesn't clarify, for me, why Clojure chose a different route than Common Lisp. – Mars Mar 8 '14 at 19:20

Amazingly clear and educational answer ! Thanks ! – jhegedus Feb 19 '15 at 8:19



After giving this question a lot of thought, I can think of several reasons to differentiate between symbols and vars, or as Omri well put it, to use "two levels of indirection for mapping symbols to their underlying values". I will save the best one for last...

1: By separating the concepts of "a variable" and "an identifier which can refer to a variable", Clojure makes things a bit cleaner conceptually. In CL, when the reader sees `a` , it returns a symbol object which carries pointers to top-level bindings, *even if* `a` is locally bound in the current scope. (In which case the evaluator will not make use of those top-level bindings.) In Clojure, a symbol is just an identifier, nothing more.

This connects to the point some posters made, that symbols can also refer to Java classes in Clojure. If symbols carried bindings with them, those bindings could just be ignored in contexts where the symbol refers to a Java class, but it would be messy conceptually.

2: In some cases, people might want to use symbols as map keys and such. If symbols were mutable objects (as they are in CL), they wouldn't fit well with Clojure's immutable data structures.

3: In (probably rare) cases where symbols are used as map keys, etc., and perhaps even returned by an API, the equality semantics of Clojure's symbols are more intuitive than CL's symbols. (See @amalloy's answer.)

4: Since Clojure emphasizes functional programming, a lot of work is done using higher-order functions like `partial` , `comp` , `juxt` , and so on. Even if you're not using these, you may still take functions as arguments to your own functions, etc.

Now, when you pass `my-func` to a higher-order function, it does *not* retain any reference to the variable which is called "my-func". It just captures the *value* as it is right now. If you redefine `my-func` later, the change will not "propagate" to other entities which were defined using the value of `my-func`.

Even in such situations, by using `#'my-func`, you can explicitly request that the current value of `my-func` should be looked up *every time* the derived function is called. (Presumably at the cost of a small performance hit.)

In CL or Scheme, if I needed this kind of indirection, I can imagine storing a function object in a cons or vector or struct, and retrieving it from there every time it was to be called. Actually, any time I needed a "mutable reference" object which could be shared between different parts of the code, I could just use a cons or other mutable structure. But in Clojure, lists/vectors/etc. are *all* immutable, so you need some way to refer explicitly to "something which is mutable".

answered Jul 27 '12 at 1:58
Alex D
18.3k 3 33 74

```
(ns a)

(defn foo [] 'foo)
(prn (foo))


(ns b)

(defn foo [] 'foo))
(prn (foo))
```

The symbol `foo` is the exact same symbol in both contexts (ie, `(= 'foo (a/foo) (b/foo))` is true), but in the two contexts it needs to carry a different value (in this case, a pointer to one of two functions).

answered Jul 26 '12 at 4:04
amalloy
37.4k 2 70 110

1 So symbols are essentially keys into a per-namespace hashtable of vars? +1 for explaining this. My question, though, is: *what* are the benefits of this design? – Alex D Jul 26 '12 at 5:15

I have surmised the following question from your post (tell me if I'm off base):
Why are there two levels of indirection for mapping symbols to their underlying values?

When I first went to answer this question, after a while I came up with two possible reasons: "re-defing" on the fly, and the related notion of dynamic scope. However, the following has convinced me that neither of these is a reason for having this double indirection:

```
=> (identical? (def a 0) (def a 10))
=> true

=> (declare ^:dynamic bar)
=> (binding [bar "bar1"]
     (identical? (var bar)
                 (binding [bar "bar2"]
                   (var bar))))
=> true
```

To me, this shows that neither "re-defing" nor dynamic scope produce any alteration to the relationship between a namespace qualified symbol and the var it points to.

At this point, I'm going to ask a new question:
Is a namespace-qualified symbol always synonymous with the var it refers to?

If the answer to this question is yes, then I simply don't understand why there should ever be another level of indirection.

If the answer is no, then I would like to know under what circumstances a namespace qualified symbol would point to different vars during a single run of the same program.

I suppose, in summary, great question :P

answered Jul 26 '12 at 6:30
Omri Bernstein
1,485 1 5 13

1 You understood my question exactly. I think I *may* have figured out the answer myself... but I'll have to do some more research to see if I am right before I post it here. – Alex D Jul 27 '12 at 1:01

The main benefit is that it is an extra layer of abstraction that is useful in various instances.

As a specific example, symbols can happily exist before creation of the var that they reference:

```
(def my-code `(foo 1 2))     ;; create a list containing symbol user/foo
=> #'user/my-code

my-code                      ;; confirm my-code contains the symbol user/foo
=> (user/foo 1 2)

(eval my-code)               ;; fails because user/foo not bound to a var
=> CompilerException java.lang.RuntimeException: No such var: user/foo...

(def foo +)                  ;; define user/foo
=> #'user/foo

(eval my-code)               ;; now it works!
=> 3
```

The benefit in terms of metaprogramming should be clear - you can construct and manipulate code before you need instantiate and run it in a fully populated namespace.

answered Jul 26 '12 at 7:06
mikera
73.6k 11 169 326

3 There's no difference from CL Here: CL-USER> (defvar my-code `(foo 1 2)) MY-CODE CL-USER> my-code (FOO 1 2) CL-USER> (eval my-code) ; in: FOO 1 ; (FOO 1 2) ; ; caught STYLE-WARNING: ; undefined function: FOO ; ; compilation unit finished ; Undefined function: ; FOO ; caught 1 STYLE-WARNING condition ; Evaluation aborted on #<UNDEFINED-FUNCTION FOO {10174E4A93}>. CL-USER> (defun foo (a b) (+ a b)) FOO CL-USER> (eval my-code) 3 – Vsevolod Dyomkin Jul 26 '12 at 10:52

for Common Lisp or other lisp, please check out: `Differences with other Lisps` from http://clojure.org/lisps

answered Jul 26 '12 at 4:03
number23_cn
2,790 12 28

2 My question is not *what* is the difference between Clojure and CL's idea of "symbols", but *why* Clojure is designed in this way. What does this design enable, which would not be possible if symbols and vars were combined into a single type? – Alex D Jul 26 '12 at 5:16

Clojure is my first (and only) lisp to date so this answer is something of a guess. That said, the following discussion from the clojure website seems pertinent (emphasis mine):

> Clojure is a practical language that recognizes the occasional need to maintain a persistent reference to a changing value and provides 4 distinct mechanisms for doing so in a controlled manner - Vars, Refs, Agents and Atoms. Vars provide a mechanism to refer to a mutable storage location that can be dynamically rebound (to a new storage location) on a ***per-thread basis***. Every Var can (but needn't) have a root binding, which is a binding that is shared by all threads that do not have a per-thread binding. Thus, the value of a Var is the value of its per-thread binding, or, if it is not bound in the thread requesting the value, the value of the root binding, if any.

So indirecting symbols to Vars permits *thread safe* dynamic re-binding (maybe this can be done in other ways, but I don't know). I take this to be part of the core clojure philosophy of rigorously and pervasively distinguishing between identity and state to enable robust concurrency.

I suspect this facility gives true benefit only rarely, if ever, compared to rethinking a problem to not require thread specific dynamic binding, but it is there if you need it.

answered Jul 26 '12 at 14:50
Alex Stoddard
6,026 3 27 55

Thanks for your input, but this still doesn't answer the question. Why? Because the per-thread data which is attached to a Var, which allows it to be rebound on a per-thread basis, *could* in theory be attached directly to a symbol. – Alex D Jul 27 '12 at 0:59