

## Clojure 1.2 Handbook

感谢 `jameqiu`, 虽然我还不认识您, 但我对您敬仰犹如滔滔江水连绵不绝, 又如黄河泛滥一发不可收拾.  
保留下来作为传播.

`jamesqiu@sina.com`  
`http://qiujj.com`

2012-01-01 修订版

发音：可鲁耶! λ☺

“和别人分享你的知识，那才是永恒之道”——somebody

## 前言：

书籍浩如烟海，技术方面的书也是汗牛充栋，可惜我们的阅读速度和理解力、记忆力太有限，好不容易学懂的知识转眼就变得非常陌生，“博闻强志、过目不忘”者毕竟罕见。对于大部分人来说，即便昔日信手拈来的东西，时间久了也会毫无头绪。所以知识不在于学过多少，而在于你能记住和使用多少。

“好记性不如烂笔头”——近年来我慢慢习惯把费力学习的东西都做一个笔记，一是在学习的过程中加深印象，毕竟技术学习不同于欣赏娱乐大片和浏览娱乐新闻看个过眼烟云；二是便于学而“时习之”，书上的东西一般是针对不同技术背景的读者，有很多作者费力用墨之处对你来说纯属废话，而他一笔带过的地方恰恰让你困惑不已。一本读书笔记相当于你对书的“注解”。

Lisp 是一门神秘的语言，有无数的 geek, hacker 对他推崇备至，也有众多的程序员对它嗤之以鼻，他和我们小学、中学、大学最先接触的 Basic、Pascal、Fortran、C/C++ 是如此的风格迥异，以至于第一眼看上去就令人排斥。其根本原因在于，大部分人有天生的排异反应，对于和自己理念相左的事物，第一反应就是：切！胡扯！但人类能弄出来的任何新玩意儿，没有那个是和原有知识完全脱钩的。终于有一天，我看到一篇文章（Lisp 的本质(The Nature of Lisp)），分析 Lisp 就是 XML++：Lisp 表达式既表示数据也表示代码，代码即数据、数据即代码，`<f>...</f>` 简化成 `(f..)`。Clojure 是 Lisp 出生数十年后的新的实现（200x 年出品），或者 JLisp（Lisp on JVM）。

对于从 C/Java 或者其他 FP 走过来的人，Lisp/Clojure 有很多“别扭”的用法，很难记清楚用正确。学任何语言，包括 Clojure，最佳的做法是把它用到日常的应用开发中，不断加深记忆。但即便你准备这么做了，手头没有一份方便的备查材料，刚开始也会步履艰难。我在使用的过程中也有这个体会，所以才不厌其烦地把一些学来并尝试过的东西记在本文档中备查，以便之后能行云流水地“玩转”它。

个人认为，对于一门编程语言使用中的查阅，大致有几个阶段：查教程（tutorial）——查手册（handbook）——查自己写的库（把个人所有的编程语言经验写成类或函数）。这个材料，不是严格的教程，或手册，而是介于这两者间。Clojure 目前已经出版了几本书，这些书从各自的角度解读 Clojure，大部分是英文的，不利于以母语速度快速浏览。数学问题用公式表达最清楚，编程问题用图表和代码表示最清楚，这二者也是本文用的最多的表达方式，我尽量采用简短的代码来说明问题（简短代码也能说明很多事情，广受赞誉的 Effective Java 基本没有超过一页的程序代码）。能够熟练使用 Java 的程序员，参考本笔记，应该可以自如地开始着手写 Clojure 程序。

希望本材料能给同样对 Clojure 感兴趣的人一些帮助。

——JamesQiu

jamesqiu@sina.com

<http://qiujj.com>

## 目录

1、 Why Clojure.....	9
1.1 美观方便、DSL .....	9
1.2 易用的数据结构 .....	10
1.3 STM 模型 .....	11
1.4 基于 JVM .....	11
1.5 Clojure 是 Lisp reload .....	12
1.6 代码==数据 .....	13
1.7 开发社区 .....	13
1.8 几点期待 .....	13
2、 lang.....	14
2.1 REPL.....	14
2.2 定义变量 def let binding .....	17
2.3 内部变量 .....	18
2.4 基本类型 .....	18
2.5 类型判断 .....	25
2.6 条件语句 if when cond condp case .....	26

2.7	循环语句	28
2.8	正则表达式 regex	34
2.9	命名空间	34
2.10	结构 defstruct	37
2.11	类 defrecord	38
2.12	接口 protocol	39
2.13	->	40
2.14	编码规范	41
3、	coll 数据结构	42
3.1	List	42
3.2	Vector	43
3.3	Set	45
3.4	Map	46
3.5	操作	51
3.6	序列 seq	62
4、	函数	62
4.1	函数帮助	62

4.2	调用函数 .....	63
4.3	以“函数名”调用 .....	63
4.4	运行时动态创建函数 .....	64
4.5	meta.....	64
4.6	定义函数 defn .....	65
4.7	defmutil 函数名重载 .....	67
4.8	匿名函数 fn #() .....	69
4.9	偏函数 partial .....	70
4.10	组合函数 .....	71
4.11	递归函数 .....	72
5、	宏 macro.....	73
5.1	概念 .....	73
5.2	设计方法 .....	73
5.3	调试宏 .....	74
5.4	~ ~' '~ ~@ .....	74
6、	调用 Java 的类和方法.....	75

6.1	基本用法 .....	75
6.2	得到所有 Java 类方法.....	76
6.3	Java 数组 .....	76
6.4	reflect 调用 Java 方法.....	77
6.5	Java 方法作为函数参数 .....	78
6.6	设置属性值 .....	78
6.7	JavaBean .....	78
6.8	提升性能 .....	78
6.9	proxy 实现接口.....	79
6.10	Exception .....	80
6.11	Java 调 Clojure .....	80
6.12	编译 .....	81
6.13	调用 OS 系统功能.....	81
7、	正则表达式 regex.....	82
8、	并发 STM.....	82
8.1	基本概念 .....	83

8.2	ref.....	83
8.3	atom.....	85
8.4	agent.....	85
8.5	var.....	86
8.6	状态更新对比 .....	86
8.7	多线程 .....	87
8.8	pmap.....	87
9、	GUI.....	87
10、	IO JDBC .....	89
10.1	文件 IO .....	89
10.2	网络 IO .....	90
10.3	配置、数据文件 .....	91
10.4	JDBC 数据库 IO.....	93
10.5	ClojureQL .....	94
10.6	MongoDB 操作.....	94
11、	Unit Test.....	96
12、	web 开发.....	97
12.1	Ring.....	97
12.2	Compojure .....	99

12.3 Conjure .....	100
12.4 Weld.....	100
12.5 hiccup.....	100
12.6 Enlive 模板引擎 .....	103
13、 网络资源.....	103
14、 临时.....	104
14.1 InfoQ 采访 Clojure 实用 .....	104
14.2 Lisp 用户的问题 .....	105
14.3 Rich Hickey 访谈.....	107
14.4 语言结构决定人类思维方式及行动方法 .....	108
14.5 The Joy of Clojure 笔记.....	108
14.6 On Lisp 笔记.....	108
14.7 Clojure on Heroku .....	108
14.8 Clojure 1.3 changelog .....	108
15、 对照表.....	109
15.1 和 clisp、scheme 的区别 .....	109
15.2 和 newlisp 的区别 .....	112



# 1、Why Clojure

“Clojure is computer reload and Lisp reload”

“Rooted in 50 years of Lisp, as well as 15 years of Java best practices”

“不识篆字不好意思说懂中文，不熟 Lisp 不好意思说懂编程”

“方便有用酷是我判断一个东西好坏的标准”

方便	<ul style="list-style-type: none"><li>● 简化，消除附加复杂性</li><li>● REPL</li><li>● 小写-空格而非骆驼命名法_,;</li><li>● 内置的数据结构及统一处理方式</li></ul>
有用	<ul style="list-style-type: none"><li>● STM 模型</li><li>● 无缝接入 JVM，直接使用现有资源</li><li>● 友好的开发社区</li></ul>
酷	<ul style="list-style-type: none"><li>● Lisp reload</li><li>● FP、STM</li><li>● 微内核，比 Scala 简单得多</li></ul>

## 1.1 美观方便、DSL

- 美观方便

C/Java 中用得最多的就是

, ; { }

而 Clojure 中用得最多的是

( ) 空格

无论敲起来还是看起来，Clojure 都更方便更美观，小写字母和连字符也比骆驼命名法不断切换大小写快得多，也好看得多（请对比：strJoin 和 str-join, writeFileWithName 和 write-file-with-name）。事实上有人宣称 Lisp 是“世界上最美丽的编程语言”，除了设计思想，估计命名也是个原因。

Lisp 相对 Java 有其方便之处，如：

1 + 2 + 3 + 4 + 5	(+ 1 2 3 4 5)
int s=0; for(i : arr) s+=i; //数据元素求和	(apply + arr)
if (n>3 && n<5)	(if (< 3 n 5))
"hello-world".substring(0,5).toUpperCase()	(.. "hello-world" (substring 0 5) toUpperCase) 或 (.. (subs "hello-world" 0

	5) toUpperCase)
简单函数也必须包装在 class 中 <pre>class foo {   void f() {...}   public static void main(String[] args) {...} }</pre>	函数随手可用: <pre>(fn ..) #(..) (defn f [] ..)</pre>

### ● DSL 易于表达业务

XML 可以表示任意 DSL, Lisp 是 xml++, 当然更面向 DSL 了。

例如对比:

xml	s 表达式
<pre>&lt;todo des="写 clojure 笔记"&gt;   &lt;item priority="high"&gt;定框架&lt;/item&gt;   &lt;item priority="high"&gt;写内容&lt;/item&gt;   &lt;item priority="low"&gt;调整格式&lt;/item&gt; &lt;/todo&gt;</pre>	<pre>(todo "写 clojure 笔记"   (item (priority high) "定框架")   (item (priority high) "写内容")   (item (priority low) "调整格式"))</pre>

## 1.2 易用的数据结构

list/vector/map 及其丰富特性支持让你解决数据结构问题时游刃有余。

```
'(1 2 3 4 5)
```

```
[1 2 3 4 5]
```

```
{:name "qh" :age 30}
```

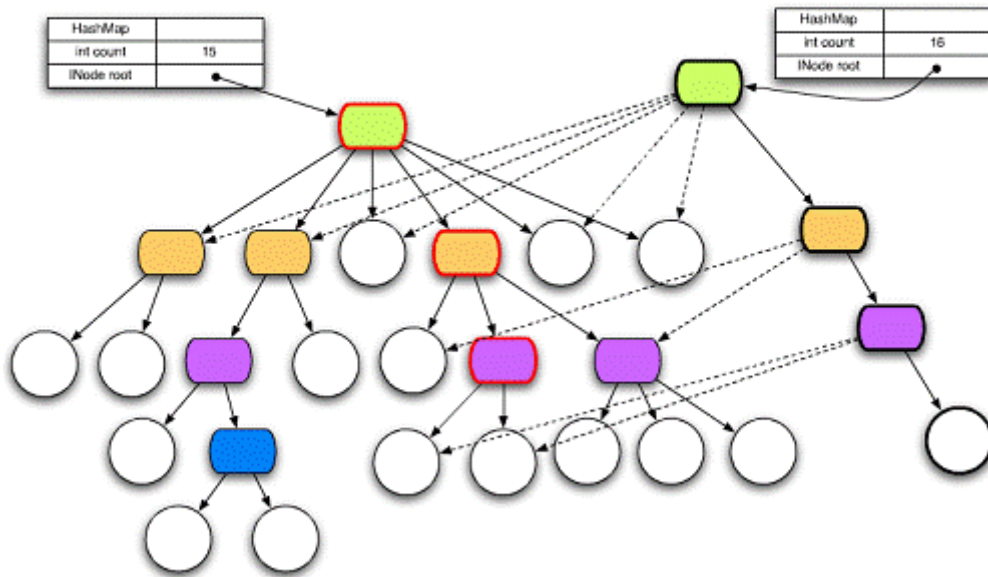
```
(defn f [n] (and (> n 10) (< n 100)))
```

```
(filter f '(1 31 4 3 53 4 234)) ; (31 53)
```

Clojure 中的数据结构具备 FP 的 immutable, 线程安全, 易于测试, 便于从小程序逐步进化到大型应用。

为了避免大量的数据结构复制会出现性能和内存瓶颈, Clojure 采用共享元素的方式仅进行部分 copy:

# Path Copying



可用 `identical?` 简单验证是否同一对象:

```
(def a [1 3 5 [2 4]])  
(def b (cons 7 (next a))) ; b 复用了 a 的部分对象  
(identical? (last a) (last b)) ; true
```

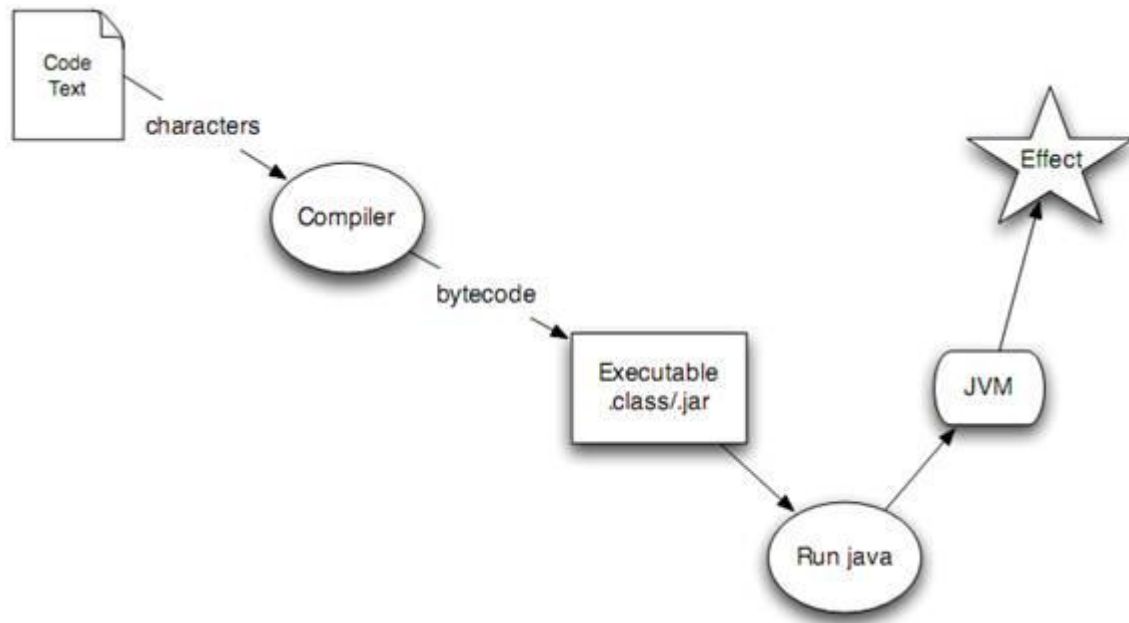
## 1.3 STM 模型

Clojure 采用 STM 模型、`agent`、`atom`、动态变量简化了并发编程。  
STM 模型和 RDB 的 OLTP 交易概念类似，易于程序员理解。

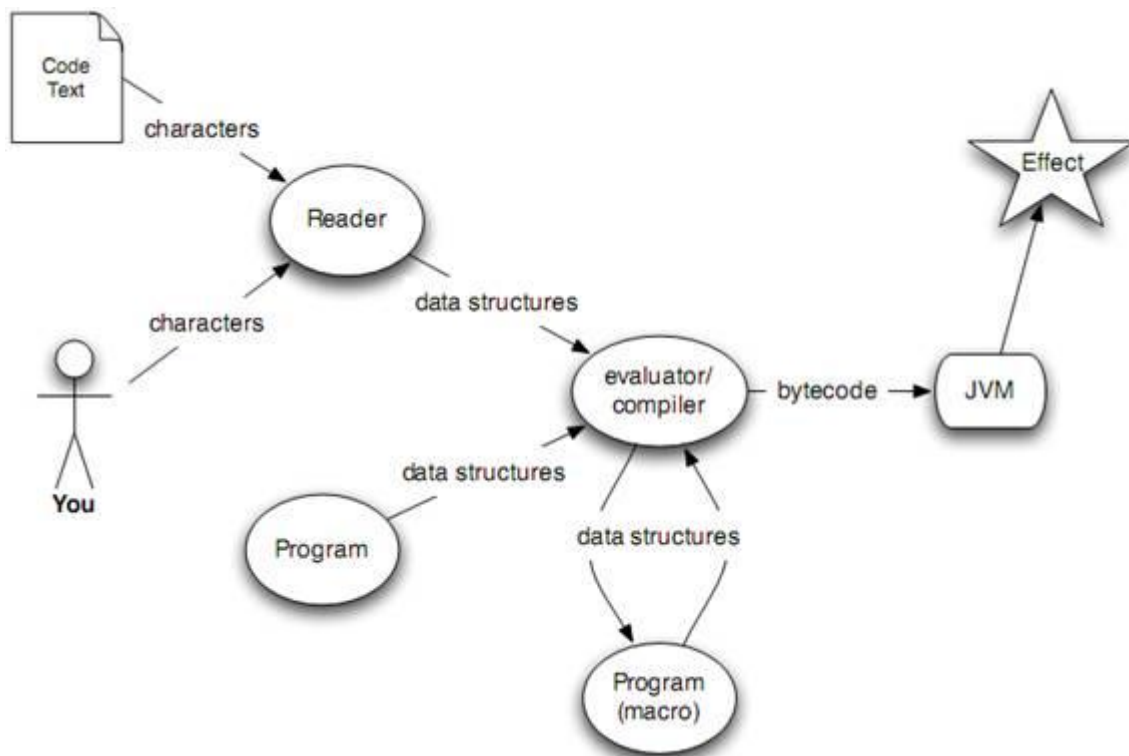
## 1.4 基于 JVM

——使用 ASM 在内存中进行编译后运行，比大部分动态语言要快得多。(Clojure 将 `org.objectweb.asm` 包下最必要的源代码摘取到 `clojure.asm` 包中，和 ASM 依赖包解耦。)  
——调用 `java` 的对象、方法直接而快速，没有太多额外开销。  
——有更多的内建库和数据结构，编程就更快，Clojure 在完全继承 Java 和 .NET 的标准库的基础上，还扩展了更丰富有用的函数库。看看 C++、D、Go 等语言库的发展情况（不是匮乏就是混乱），就知道从头创建如 Java、.NET 这般庞大全面的类库并非易事；  
类库和运行速度有关系吗？——很大程度上有，众多专家已经在类库中准备了充分优化的稳定算法，Clojure 对 Java Collection 算法进行直接包装或者直接调用，如果没有丰富的类库，你在项目周期内免不了摘抄一些不一定靠谱的算法和功能代码，这些代码极有可能在运行时给你带来麻烦。使用类库算法，不用担忧自造轮子的运行效率。

Java/Scala 运行模式如下：



Clojure 运行模式如下:



## 1.5 Clojure 是 Lisp reload

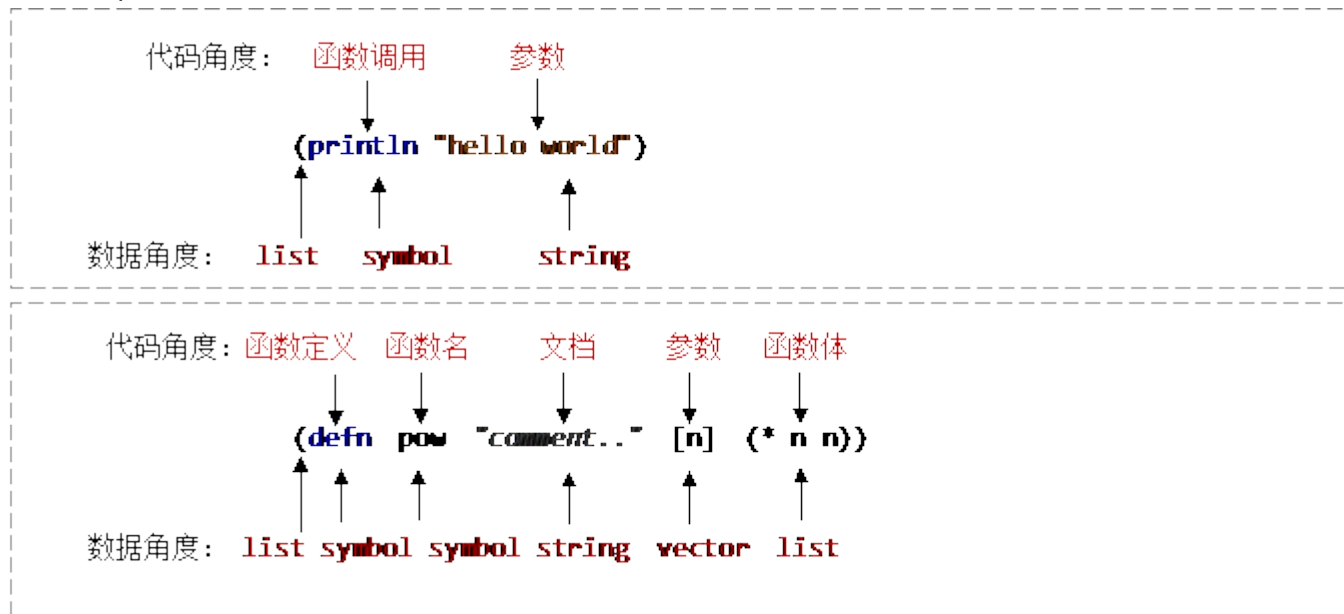
语言是大事——语言影响使用者的思维方式。语言和语言不一样。

- **Lisp** 微内核，几乎没有多余的关键字和语法，只有 `(f (g ..) (h..) ..) '() [] {} #{}` `(def )` `(defn )` 等少数，其他都是函数库，学习起来比 **Scala** 和 **Java** 都简单。
- **Lisp** 基于数学逻辑 **s-表达式**、支持 **GC**
- **Lisp** 年代久远（1958 by John McCarthy），健康长寿。

- Lisp 有 metaprogramming 能力
  - ^开头的类名或者任意 map, 如: `^String ^{:doc ".." :author "qh"}`
- Clojure 增加了对非 Lisp 程序员的友好性（语法比传统 Lisp 丰富一些）。
- Clojure 比 Lisp 更纯函数式，意味着拥有不可变性和不易错性。
- Lisp/Clojure 是动态类型，动态编译，动态转载的
- Read-Eval-Print Loop 交互解释器，编程更愉快
- 代码即数据——我的理解: (1) Lisp 代码足以表述数据, 不需单独的 .properties, .xml 等辅之 (2) Lisp 代码也是以 list 的数据结构显式表示的, 在编译时、运行时都可以生成执行新代码。
- 所有数据结构通用的列表处理方式
- 语言内核非常精炼但扩展性极强

## 1.6 代码==数据

这是 Lisp 语言最与众不同之处:



## 1.7 开发社区

Clojure 开发/用户社区气氛良好，基本都是资深开发者、顶级 Hacker 以及有追求的程序员，层次很高。

## 1.8 几点期待

- 希望 Clojure 加入类似 python 的 `""" .... """` 多行注释，其中可以包含除 `;;;` 之外的任意字符，如：
 

```
;;; 11111111111
```

```
2222222222222222
```

```
3333333333333333 ;;;
```

目前可以用如下写法，但必须注意包含 "(" " " ";" 等特殊符号：

```
(comment 111111111
```

```
2222222222222222
```

```
3333333333333333 )
```

- 更好的错误提示，别来个：

```
NullPointerException at line 0
```

- 更快的启动速度，至少达到 bsh 和 scala 这种级别（注：推荐 newlisp 来写 shell 和进行系统调用）。

## 2、 lang

### 2.1 REPL

#### 2.1.1 使用

启动：java -cp clojure.jar;clojure-contrib.jar clojure.main

退出：Ctrl-Z

手动转载文件中的程序：

```
(load-file "temp.clj")
```

自动转载 clj 文件：

在 classpath 如 e:\clojure-1.2.0\jamesqiu 中建名为 user.clj 的文件即可。

#### 2.1.2 自动完成

通过 jLine (<http://jline.sourceforge.net/>) 的 Completor 类（放在 classpath 如 jamesqiu 目录下）：

```
@java
```

```
-cp clojure.jar;clojure-contrib.jar;.\jamesqiu\jline.jar;.\jamesqiu  
-Djline.ConsoleRunner.completors=ClojureCompletor jline.ConsoleRunner c  
lojure.main %*
```

ClojureCompletor.java 内容如下（sorry，忘了来自哪里，但自行编译即可）：

```
-----  
import java.util.List;  
import jline.Completor;  
import clojure.lang.MapEntry;
```

```

import clojure.lang.Namespace;
import clojure.lang.RT;
import clojure.lang.Symbol;

public class ClojureCompletor implements Completor{
    @Override
    public int complete(String buffer, int cursor, List candidates) {
        if(buffer == null)
            buffer = "";
        Namespace ns = (Namespace) RT.CURRENT_NS.get();
        String split[] = split(buffer);
        String head = split[0];
        String tail = split[1];
        boolean exist = false;
        for(Object it: ns.getMappings()){
            MapEntry entry = (MapEntry) it;
            if(entry.getKey() instanceof Symbol){
                Symbol symbol = (Symbol) entry.getKey();
                if(symbol.getName().startsWith(tail)){
                    candidates.add(symbol.getName());
                    exist = true;
                }
            }
        }
        return exist ? head.length() : -1;
    }

    String[] split(String buffer){
        int end = buffer.length() - 1;
        for(; end >= 0; end--){
            char ch = buffer.charAt(end);
            if(" \t,(".indexOf(ch)>0){
                break;
            }
        }
        String result[] = new String[2];
        if(end >=0){
            result[0] = buffer.substring(0, end+1);
            result[1] = buffer.substring(end+1);
        }
        else {
            result[0] = "";
            result[1] = buffer;
        }
        return result;
    }
}

```

```

    }
}
-----

```

### 2.1.3 print

```

(print "hello world")
(println "hello" "world" 1) ; "hello world 1"
(sprintf "hello %s (%.3f)" "world" 3.0) ; hello world (3.000)
(format "hello %s (%.3f)" "world" 3.0) ; "hello world (3.000)"
(str "hello " "world " 3) ; "hello world 3"
(println-str "hello" "world" 3) ; "hello world 3\n"

```

注意: pr,prn,pr-str,prn-str 与 print,println,print-str,println-str 略有不同

### 2.1.4 注释

单行: ; ;; ;;; Lisper 习惯于用越多;表示越重要或者越概要的注释

```

;      单行注释
;;     函数注释
;;;    macro 或者 defmulti 的注释
;;;;   ns 注释

```

多行:

```

(comment "
...1...
...2...
")

```

### 2.1.5 设置

```

(set! *print-length* 103)
(iterate inc 1)

```

输出就不会太多而停不下来了:

```

(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 ...)

```

要改回无限制:



```
(set! *print-length* false)
```

另: `*command-line-args*` 获得命令行参数

## 2.2 定义变量 `def let binding`

变量名可以下面字符开始

(<https://github.com/laurentpetit/ccw/blob/master/clojure-antlr-grammar/src/Clojure.g>) :

```
'a'..'z' | 'A'..'Z' | '*' | '+' | '!' | '-' | '_' | '?' | '>' | '<' | '='  
| '$'
```

### 2.2.1 `def`

`def` 定义值或名称, 相当于 `Scala` 的 `val/def`

定义:

```
(def v1 10)  
(def v2 (int 10)) ; 定义成基本类型, 在循环计算中可以提高效率  
(def v3 (fn [n] (* n n))) ; v3 是函数别名
```

### 2.2.2 `let`

局部临时变量定义:

```
(let [x 10] (println x))
```

定义多个变量 (`def` 好像不能定义多个):

```
(let [[x y] [3 4]] (println (* x y))) ; 12  
(let [x 3 y 4] (println (* x y)))  
(let [[x y] [3 4 5]] [x y]) ; [3 4] 多余的 5 被忽略  
(let [[_ _ z] [3 4 5]] z) ; 5  
(let [[a b &c] [1 2 3 4 5]] [a b c]) ; [1 2 (3 4 5)]
```

多个变量之间可以依赖 (后面的依赖前面的), 这点\*非常\*非常\*有用:

```
(let [x 10 y (* x x) z (* 2 y)] (println z)) ; 200
```

`let` 的执行体内可以调用多个函数:

```
(let [x 10] (println x) (println (* x x)))
```

### 2.2.3 `binding`

`binding` 可以临时改变 `def` 定义的变量:

```
(def v1 10)  
(def v2 20)  
(binding [v1 1 v2 2] (+ v1 v2)) ; 3  
(+ v1 v2) ; 30  
(defn f [] (+ v1 v2))  
(binding [v1 1 v2 2] (f)) ; 3
```

```
(f) ; 30
```

binding 内部还可以使用 set!来设置变量:

```
(def v 10)
(declare x)
(binding [x 20] (inc x)) ; 1 x 必须先用 def 或者 declare 定义
```

使用 declare 定义变量但不绑定初始值:

```
(declare v1)
(defn f [] (println v1))
(let [v1 10] (f)) ; 报错
(binding [v1 100] (f)) ; 100
```

## 2.2.4 let binding 区别

```
(def v 0)
(defn f [] (println v))
(let [v 10] (f)) ; 0 没有影响 f 函数, 只会影响(let..)内显式出现的 v 值
(binding [v 100] (f)) ; 100 影响 f 函数, 是 ThreadLocal 级别的
```

## 2.3 内部变量

函数或者过程内的变量可以使用 let 或者 def 来定义:

定义:  $(x+y)*(x+y)$

使用 let (推荐):

```
(defn f1 [x y]
  (let [xy (+ x y)]
    (* xy xy)))
```

使用内部 def (慎用,不推荐):

```
(defn f2 [x y]
  (def xy (+ x y))
  (* xy xy))
```

## 2.4 基本类型

### 2.4.1 一览表

类型	例子	说明
boolean	true false	(class true) -> java.lang.Boolean

char	\a	(class \a) -> java.lang.Character
string	"hello"	(class "hi") -> java.lang.String
keyword	:tag :doc	(class :tag) -> clojure.lang.Keyword
symbol (指针或内部标识)	'a + map java.lang.String	(class 'a) ; clojure.lang.Symbol (class +) ; clojure.core\$_PLUS_ (class map) ; clojure.core\$map (class java.lang.String) -> java.lang.Class
list	'(1 2 3) (println "hello")	一组数据, 或者函数调用 (class '(2 3)) -> clojure.lang.PersistentList
vector	[1 2 3]	(class [1 2 3]) -> clojure.lang.PersistentVector
set	#{:red :green :blue}	(class #{:r :g :b}) -> clojure.lang.PersistentHashSet
map	{:name "qh" :age 30}	(class {:k1 "v1"}) -> clojure.lang.PersistentArrayMap
空	nil	(class nil) -> nil
number	1 4.2	(class 1) -> java.lang.Integer (class 1.2) -> java.lang.Double (class 1.2M) -> java.math.BigDecimal

注: 取类型用 (`class foo`) 或者 (`type foo`)

## 2.4.2 区分#

样式	说明
#{1 3 5}	set
#(.toUpperCase %)	匿名函数
#"a?b"	regex 正则表达式

## 2.4.3 Number

### 2.4.3.1 Int

```
(int 10)
(int 10.9) ; 10
(int \a) ; 97
(Integer/parseInt "12") ; 12
```

数字转换为字符串:

```
(str 10) ; "10"
```

数字运算符:

```
(+) ; 0
(+ 3) ; 3
(+ 3 4) ; 7
(+ 3 4 5) ; 12
(-) ; 错误
(- 3) ; -3
(- 3 2) ; 1
(- 3 2 2) ; -1
(*) ; 1
(/) ; 错误
(/ 3) ; 1/3
(/ 3 2) ; 3/2
(/ 3 2 2) ; 3/4
```

判断奇偶:

```
(even? 10) ; true
(odd? 3) ; true
```

判断正负:

```
(neg? -1.2) ; true
(pos? 3) ; true
(zero? 0.0) ; true
```

增减 1:

```
(inc 3) ; 4
(dec 4) ; 3
```

愿意用++, --的, 可以

```
(def ++ inc)
(def -- dec)
```

随机数:

```
(rand-int 100) ; 得到一个[0 100) 的随机数
(for [i (range 20)] (rand-int 10)) ; (4 0 5 9 9 7 5 9 3 1 0 8 7 1 9 8
9 4 7 1)
(take 10 (repeatedly #(rand-int 100))) ; (50 12 86 73 42 90 84 96 71 33)
```

取一个集合中的随机位置元素:

```
(rand-nth [11 22 33 44 55]) ;
(rand-nth (range 100)) ; 每次运行结果会变
```

float 随机数:

```
(rand) ; [0 1.0) 的 float, 如: 0.43488316932510274
(rand 10) ; [0 10.0), 如: 6.26826853733003
```

#### 2.4.3.2 bigint、BigInt、BigInteger

注意:

Clojure1.3 对 `bigint` 的处理发生了变化，原来的 `bigint` 就直接用 `java.math.BigInteger`，1.3 为了统一并提高数值较小时的运算速度，自己造了 `clojure.lang.BigInt`，但没有实现所有 `BigInteger` 的方法。

Clojure 1.2	(type (bigint 3)) ; java.math.BigInteger 有.pow .isProbablePrime 等方法可用
Clojure 1.3	(type (bigint 3)) ; clojure.lang.BigInt 可用方法少, 但数小时运算更快, 要用 BigInteger 的方法的话, 可先 转换: (BigInteger/valueOf 3) ; 3 (BigInteger/valueOf 3N) ; 3 (bigint 3) ; 3N

==只能用于 Number

例如:

#### 2.4.3.4 其他进制的数

### 10 进制数——>其他进制字符串:

```
(Integer/toString 1024 2) ; "10000000000"
```

```
(Integer/toString 196 16) ; "c4"
```

### 其他进制字符串——> 10 进制数:

```
(Integer.parseInt "1001" 2) ; 9
```

```
(Integer/parseInt "ce" 16) ; 206
```

### 2.4.3.5 Ratio

```
(def x (/ 1 3)) ; 1/3
```

```
(class x) ; clojure.lang.Ratio
```

$$(* \times 3) ; 1$$

### 小数转换为分数:

```
(rationalize 1.33) ; 133/100
```

```
(rationalize 1.4M) ; 7/5
```

```
(rationalize 5.0) ; 5
```

取分子分母:

```
(numerator (/ 2 3)) ; 2
```

```
(denominator (/ 2 3)) ; 3
```

要整除:

(quot 10 3) ; 3 相当于 java 的 10/3 quotient: 除法的商

(rem 10 3) ; 1 相当于 java 的 10 % 3 remainder: 余数

#### 2.4.3.6 BigDecimal

表示: 1.20M

比较:

```
(+ 1 (/ 0.00001 10000000000000000000)) ; 1.0
```

```
(+ 1 (/ 0.00001M 100000000000000000000)) ; 1.000000000000000000000000001M
```

```
(class (* 1000 1000 1000)) ; java.lang.Integer
```

```
(class (* 1000 1000 1000 1000 1000 1000 1000 1000)) ; 自动提升为
java.math.BigInteger
```

```
(with-precision 10 (/ 1M 3)) ; 0.333333333M
```

#### 2.4.4 Boolean

原则：除了 false 和 nil，其他都为 true

```
(> 5 2) ; true
```

```
(>= 5 2) ; false
```

```
(>= 5 5) ; true
```

```
(= 5 2) ; false
```

```
(= 5 5) ; true
```

可以不止 2 个数比较，非常方便：

```
(> 5 3 1 -1) ; true
```

```
(> 5 3 6) ; false
```

```
(= 2 2 2 2 2) ; true
```

```
(not (= 5 3)) ; true
```

```
(not= 5 3) ; true
```

```
(and true false) ; false
```

```
(or true false) ; true
```

n 为 0 或者 1，可以写成：

```
(<= 0 n 1)
```

返回值为 true/false 的函数一般名字最后是"?"，如：

```
(string? "hello") ; true
```

```
(defn isodd? [n] (= 0 (mod n 2)))
```

判断奇偶正负：

```
(odd? 3) ; true
```

```
(even? 2) ; true
```

```
(pos? 1.2) ; true
```

```
(zero? 0) ; true
```

```
(neg? -3) ; true
```

#### 2.4.4.1 and/or 的应用

or	从左到右，碰到非“nil 和 false”马上返回，否则返回最后一个
and	从左到右，碰到“nil 或 false”马上返回，否则返回最后一个

例子：操作不成功时提供缺省值：

```
(or (first nil) "default") ; "default"
```

```
(or (first [1 2]) "default") ; 1
```

```
(and nil "default") ; nil
```

```
(and 1 2) ; 2
```

#### 2.4.5 String

```
(str 10) ; "10"
```

```
(str \a) ; "a"
```

```
(str \a nil "bc") ; "abc"
```

```

(str "hello" "world") ; "hello world"
(str '(\a \b \c)) ; "(\a \b \c)"
(apply str '(\a \b \c)) ; "abc" 相当于 (str \a \b \c)
(seq "abc") ; (\a \b \c)
(format "%s %s" "hello" "world") ; "hello world"
(subs "12345" 1) ; "2345" 相当于 (.substring "12345" 1)
(subs "12345" 1 3) ; "23" 相当于 (.substring "12345" 1 3)
(apply str (reverse "12345")) ; "54321"

```

例子: Stringx.join 的 Clojure 实现

```

(apply str (interpose sep seq))
(apply str (interpose " + " [1 2 3 4 5])) ; "1 + 2 + 3 + 4 + 5"

```

注意: wr3.clj.s 有所有常用的 String 处理函数, clojure.string 也有部分重要的 String 函数, 如:

```

blank? escape join split split-lines
capitalize lower-case/upper-case
replace replace-first reverse

```

#### 2.4.6 Char

```

\a \n \t \space \\
(char 97) ; \a
(first "hello") ; \h
(ffirst ["hello" "world"]) ; \h
(nth "hello" 4) ; \o
(Character/toUpperCase \s) ; \S
(interleave "abc" "123") ; (\a \1 \b \2 \c \3)
(apply str (interleave "abc" "123")) ; "a1b2c3"
(seq "hello") ; (\h \e \l \l \o)
(vec "hello") ; [\h \e \l \l \o]
(vector "hello") ; ["hello"]

```

#### 2.4.7 fnil

针对 nil, Clojure 提供了 fnil 函数, 可把 f 函数的第 1 个或者 2 个参数如下处理:

```

(defn f [a b] (+ a b))
(f 10 nil) ; java.lang.NullPointerException
((fnil f 0 0) 10 nil) ; 10
((fnil f 0 0) nil 10) ; 10
((fnil f 0 0) nil nil) ; 0
((fnil f 0) nil 10) ; 10
((fnil f 0) 10 nil) ; java.lang.NullPointerException

```



## 2.5 类型判断

取类型:

```
(class foo)
```

判断类型:

```
(instance? Integer 10)
```

```
(instance? String "hi")
```

特定类型判断的单目表达式:

```
(true? true) ; 只有 true 为 true, 其他皆为 false
```

```
(true? false) ; false
```

```
(true? nil) ; false
```

```
(true? "") ; false
```

```
(true? "abc") ; false
```

```
(false? false) ; 只有 false 为 true, 其他皆为 false
```

```
(false? true) ; false
```

```
(false? nil) ; false
```

```
(nil? nil) ; true
```

```
(nil? false) ; false
```

```
(zero? 0) ; true 参数只能是数字
```

```
(zero? 0.0) ; true
```

```
(zero? 0.0M) ; true
```

```
(zero? (/ 1 3)) ; false
```

```
(char? \a) ; true
```

```
(string? "hello") ; true
```

```
(number? 10) ; true
```

```
(number? 10.3) ; true
```

```
(number? 10.3M) ; true
```

```
(number? (/ 1 3)) ; true
```

```
(integer? 10) ; true
```

```
(ratio? (/ 1 3)) ; true
```

```
(every? rational? [3 3.14M (/ 1 3)]) ; true 有理数
```

```
(float? 1.3) ; true
```

```
(float? (double 1.3)) ; true *没有* (double? 1.3)
```

```
(keyword? :k1) ; true
```

```
(symbol? 's1) ; true
```

```
(every? coll? (list [] '() {} #{})) ; true
```

```
(seq? '()) ; true
```

```
(vector? []) ; true
(list? '()) ; true
(map? {}) ; true
(set? #{}) ; true
```

### 2.5.1 使用 seq 做判断

利用 seq 的特性:

```
(seq nil) ; nil
(seq {}) ; nil
(seq []) ; nil
(seq #{}) ; nil
(seq '()) ; nil
```

可统一判断处理空的基本类型、sequence 类型

## 2.6 条件语句 if when cond condp case

if 条件中除了 false 和 nil, 其他都为 true:

```
(if true "true") ; "true"
(if 0 "true") ; "true"
(if "" "true") ; "true"
(if nil "true") ; nil
(if false "true") ; nil
(if-not false "true") ; true
(if-not true "true") ; nil
```

```
(if (not (= a b)))
```

用 not= 可以写成:

```
(if (not= a b))
```

用 if-not 可以写成

```
(if-not (= a b))
```

第三个参数就是 else 子句, 但 "else" 不用写:

```
(if true "true" "false") ; "true"
(if false "true" "false") ; "false"
```

when 没有 else 子句, 执行条件后的所有语句:

```
(when true "1" "2" "3") ; "3"
(when false "1" "2" "3") ; nil
(when-not false "1") ; "1"
(when-not true "1") ; nil
```

例子: 区别 if 和 when, 打印小于 5 的正整数

`(loop [i 1] (if (< i 5) (println i) (recur (inc i)))) ; 错误 仅打印 1`  
`(loop [i 1] (if (< i 5) (do (println i) (recur (inc i))))) ; 正确`  
`(loop [i 1] (when (< i 5) (println i) (recur (inc i)))) ; 正确 when 把条件判断后的所有都执行`

cond 类似于 switch..case..default:

```
(cond (= 5 2) "5==2" (= 5 5) "5==5" (= 5 6) "5==6") ; "5==5"
(defn f [n] (cond (< n 0) "<0" (< n 10) "<10" :else ">=10"))
(f -1) ; "<0"
(f 1) ; "<10"
(f 10) ; ">=10"
```

注: :else 只是个习惯用法, 变成其他东西, 不影响 cond 的功能。

例子: 猜数字游戏

```
(def ans (rand-int 100))
(defn guess [n] (cond (= n ans) "got it!" (< n ans) "too small" (> n ans)
"too large"))
(guess 62)
```

condp:

condp	对应的 cond 写法
(def n 3)	(def n 3)
(condp = n 2 "2=n" 3 "3=n" "no match") ; "3=n"	(cond (= 2 n) "2=n" (= 3 n) "3=n" :else "no match")
(condp > n 0 "0>n" 5 "5>n" "others") ; "5>n"	(cond (> 0 n) "0>n" (> 5 n) "5>n" :else "other")

注意: 可见后面的值如 0, 5 是作为>函数的第一个参数。

case 类似于 Java 的 switch:

```
(defn f [x] (case x
1 10
2 20
3 30
0))
(f 1) ; 10
(f 3) ; 30
(f 4) ; 0
```

case 可以匹配不同类型:

```
(defn jumper [x]
(case x
"JoC" :a-book
"Fogus" :a-boy
:eggs :breakfast))
```

```
42      (+ x 100)
[42]    :a-vector-of-42
"The default"))
```

### 2.6.1 when-first

另有 `when-first` (“但集合不为空或者 `nil` 时，绑定为第一个元素”：

```
(when-first [a [1 3 5]] a) ; 1 如果
(when-first [a [1 3 5]] "hello") ; "hello"
(when-first [a []] a)      ; nil
(when-first [a nil] a)     ; nil
```

### 2.6.2 if-let when-let

`if-let` 在值不为 `nil`、`false` 时执行前者，否则后者：

```
(= (if-let [a nil] a 10) 10)
(= (if-let [a false] "yes" "no") "no")
(= (if-let [a 4] (* a 10) 10) 40)
```

`when-let` 在值为 `nil`、`false` 时返回 `nil`

```
(= (when-let [a 4] (* a 10)) 40)
(= (when-let [a false] (* a 10)) nil)
(when-let [a 9] (println a) (+ a 4))
9
13
```

## 2.7 循环语句

### 2.7.1 loop

```
(loop [a [1 3 5] b [2 4 6]] (interleave a b)) ; (1 2 3 4 5 6)
```

### 2.7.2 if recur, when recur

```
((fn [s ii] (if ii (recur (subs s (first ii)) (next ii)) s))
 "hello world" [1 2 3]) ; "world"
```

对比 `loop..if..recur` 版本：

```
(loop [s "hello world" ii [1 2 3]] (if ii (recur (subs s (first ii)) (next
ii)) s)) ; "world"
```

仅使用 `recur`：

```
(defn f [n] (when (pos? n) (println n) (recur (dec n))))
```

相当于：

```
(defn f [n] (when (pos? n) (do (println n) (recur (dec n))))))
```

相当于如下的简写：

```
(defn f [n] (loop [i n] (when (pos? i) (do (println i) (recur (dec i))))))
```

例子：求下一个素数

```
(defn pnext [n] (let [n2 (inc n)] (if (prime? n2) n2 (recur n2))))
```

如何找集合中符合条件的第一个元素（模拟 Scala 的 find）

方法 1:	可以用 if..recur: (defn find-first [pred coll] (loop [c coll] (if (or (empty? c) (pred (first c))) (first c) (recur (rest c)))))
方法 2:	其实由于 filter 是 lazy 的，如下即可： (first (filter pred coll))
方法 3:	或者直接用 contrib 中的函数： (use '[clojure.contrib.seq-utils :only (find-first)]) (find-first even? [1 3 5 2 4]) ; 2
方法 4:	巧用 some: (some #(when (pred %) %) coll)
方法 5:	用补函数： (first (drop-while (complement pred) coll))

这样 pnext 也可以写成

```
(defn pnext [n] (find-first prime? (range (inc n) (* 2 n))))
```

也可以用：

```
(defn pnext [n] (first (filter prime? (iterate inc (inc n)))))  
(take 10 (iterate pnext 2))
```

### 2.7.3 loop if recur

Clojure 是 FP，没有 C/Java 的循环变量，要使用递归（recursion）。

形式：

```
(loop [v...] (if (...) rt (recur v'...)))
```

说明：

- recur 表示递归表达式；if 内的条件是循环退出条件，如果多个条件也可以用 cond；rt 是递归出口值；
- v... 是循环变量初始值，v'... 是 v... 对应的变化（位置必须一致，变量个数必须一致）；

例子：变量 x 从 1 到 7，变量 r 存值

```
(loop [r [] x 1] (if (> x 7) r (recur (conj r x) (inc x)))) ; [1 2 3 4 5 6 7]
```

使用 loop 就不用定义函数了，对比：

```
(defn f [r x] (if (> x 7) r (recur (conj r x) (inc x))))
(f [] 1) ; [1 2 3 4 5 6 7]
```

例子：1+2+……+100

```
(loop [i 1 r 0] (if (> i 100) r (recur (inc i) (+ r i)))) ; 5050
```



例子：fib 数列 (1 2 3 5 8 13 ……)

```
(defn fib [n] (loop [a 1 b 2 i n] (if (zero? i) a (recur b (+ a b) (dec i)))))
(map fib (range 10)) ; (1 2 3 5 8 13 21 34 55 89)
```

对比不使用 loop 的自递归写法：

```
(defn fib [a b i] (if (zero? i) a (recur b (+ a b) (dec i))))
(map (partial fib 1 2) (range 10))
```

例子：fib 数列的另一种 loop 写法，直接得到前 n 个 fib 数

```
(defn fib2 [n] (loop [i 2 r [1 2]]
  (if (>= i n) r (recur (inc i) (conj r (+ (last r) (last (butlast r)))))))
```

例子：阶乘 5! =120

```
(loop [i 1 r 1] (if (> i 5) r (recur (inc i) (* i r)))) ; 120
```

参照上面，可定义阶乘函数如下：

```
(defn fac [n] (loop [i 1 r 1] (if (> i n) r (recur (inc i) (* i r)))))
```

对比不用 loop 的：

```
(defn fac0 [i r] (if (zero? i) r (recur (dec i) (* i r))))
(defn fac [n] (fac0 n 1))
```

例子：牛顿法求 n 的平方根（说明：任取一个数 a (1 或者 n/2)，另一个因子就是 b=n/a；如果 a 和 b 不够接近，取平均 a=(a+b)/2，而 b=n/a 直到 a、b 足够接近）

```
(defn sqrt2 [n]
  (loop [a (/ n 2.0) b (/ n a)]
    (if (>= 0.00001 (Math/abs (- a b))) (double a)
      (recur (/ (+ a b) 2) (/ n (/ (+ a b) 2)))))
(sqrt2 2)
```

## 2.7.4 dotimes dorun doall

`(dotimes [i 5] (println i))` ; 打印 0 1 2 3 4  
相当于 Scala 的

`0 until 5 foreach println // Scala`

一般也可以写成:

`(dotimes [ 5] (println ))`

注: 可以用 `doseq` 来完成 `dotimes` 的工作

`(doseq [i (range 5)] (println i))`

例子: 打印 `seq` 中的所有元素:

`(def s [2 4 6 8])`

`(dotimes [i (count s)] (println (nth s i)))`

输出:

2

4

6

8

或者:

`(dorun (map println [2 4 6 8]))`

相当于 Scala 的

`List(2,4,6,8) foreach println // Scala`

也可以用 `doseq`:

`(doseq [x [2 4 6 8]] (println x))`

`dorun` 只逐个处理 `seq` 元素而不在内存中生成 `seq`; 与 `doall` 有区别:

`(doall (map println [2 4 6 8]))`

## 2.7.5 doseq

`(doseq [i (range 10)] (println i))`

相当于 Scala 的

`for (i<-Range(0,10)) println(i) // Scala`

注意: 而 Clojure 的 `for` 相当于 Scala 的 `for..yield`

变量可以是多个:

`(doseq [i [1 2 3] j [10 20]] (println i "-" j))`

输出:

1 - 10

1 - 20

2 - 10

2 - 20

3 - 10

3 - 20

相当于 Scala

`for (i<- List(1,2,3); j<- List(10,20)) println(i + " - " + j) // Scala`

结合 destructure:

```
(def v (map vector (iterate inc 0) ["one" "two" "three"])) ;([0 "one"]  
[1 "two"] [2 "three"])  
(doseq [[i w] v] (println i w))
```

输出:

```
0 one  
1 two  
2 three
```

深入分析:

其实 `doseq` 和 `for` 的语法是一样的, 只不过 `for` 返回 lazy seq 而 `doseq` 是 side effect 的, 对比:

<pre>(doseq [x (range 7) y (range x) :while (odd? x)]   (print [x y]))</pre>	<pre>(for [x (range 7) y (range x) :while (odd? x)]   [x y])</pre>
<pre>[1 0][3 0][3 1][3 2][5 0][5 1][5 2][5 3][5 4]nil</pre>	<pre>([1 0] [3 0] [3 1] [3 2] [5 0] [5 1] [5 2] [5 3] [5 4])</pre>

## 2.7.6 repeat replicate

```
(repeat 10 \a) ; (\a \a \a \a \a \a \a \a \a \a)  
(apply str (repeat 10 \a)) ; "aaaaaaaaaa"
```

上两个例子中, 也可以用 `replicate`

无次数限制:

```
(repeat \a) ; 注意会无限循环
```

需要配合 `take` 来限制次数:

```
(take 10 (repeat \a))
```

## 2.7.7 iterate

var 变量迭代:

```
(iterate func var)
```

相当于:

```
(for i=0; ; i++) { fun }
```

所以一般要配合 `(take n sequence)` 来中止:

```
(take 10 (iterate inc 5)) ; (5 6 7 8 9 10 11 12 13 14)  
(take 10 (iterate #(+ % 5) 5)) ; (5 10 15 20 25 30 35 40 45 50)  
(take 10 (iterate #(* % 2) 2)) ; (2 4 8 16 32 64 128 256 512 1024)
```

例子: fib 数列 (1 2 3 5 8 13 .....)

```
(defn fib [[a b]] [b (+ a b)]) ; 得到下一组 fib 数, 如[1 2]->[2 3]->[3 5]  
(take 5 (iterate fib [1 2])) ; ([1 2] [2 3] [3 5] [5 8] [8 13])  
(take 5 (map first (iterate fib [1 2]))) ; 取头部 (1 2 3 5 8)
```



例子：质数表

```
(defn pnext [n] (first (filter prime? (iterate inc (inc n)))))  
(take 10 (iterate pnext 2)) ; (2 3 5 7 11 13 17 19 23 29)
```

### 2.7.8 cycle

类似于(repeat var)，不过(cycle seq)针对 seq，也必须用 take 来限制次数：

```
(take 5 (cycle [1 0])) ; (1 0 1 0 1)  
(take 10 (cycle (range 3))) ; (0 1 2 0 1 2 0 1 2 0)
```

### 2.7.9 take, take-while, drop-while

repeat, iterate, cycle 等无限循环都涉及 (take n coll)

```
(take-while pred? coll)
```

相当于：

```
while (pred?) { take element from coll }
```

如：

```
(take-while even? [2 4 6 1 8]) ; [2 4 6]  
(take-while #( > % 0) [3 2 1 0 -1 -2]) ; (3 2 1)
```

或者使用 pos? 代替匿名函数：

```
(take-while pos? [3 2 1 0 -1 -2]) ; (3 2 1)
```

drop-while: 一直 drop 直到碰到不符合条件后马上停止，并返回剩下的部分。

```
(drop-while even? [2 4 6 1 3 5]) ; [1 3 5]  
(drop-while even? [1 3 2 4 5]) ; [1 3 2 4 5]
```

### 2.7.10 for

类似于 scala 的 for..yield 和 Python 的 list comps。用于简化 map、filter、#()、fn:

用 map filter #()	用 for
(map #(* % %) (range 10))	(for [i (range 10)] (* i i))
(map #(* % %) (filter #(< (* % %) 20) (range 10)))	(for [i (range 10) :when (> 20 (* i i))] (* i i))

多个变量：

```
(for [x (range 10) y (range 3) :while (< y x)] [x y])  
(for [a [1 3 5] b [2 4]] [a b]) ; ([1 2] [1 4] [3 2] [3 4] [5 2] [5 4])
```

区别:when 和:while

```
(for [x [1 2 0 3 4] :when (pos? x)] x) ; (1 2 3 4) 全部做完为止  
(for [x [1 2 0 3 4] :while (pos? x)] x) ; (1 2) 碰到不满足就停
```

例子：乘法口诀表

```
(for [a (range 1 10) b (range 1 10) :while (<= b a)] (str b "*" a "="
(* a b)))
```

例子：找杨辉三角形

```
(defn tri-yang? [a b] (def c (Math/sqrt (+ (* a a) (* b b)))) (= (int c)
c))
(defn f [n] (for [a (range 1 n) b (range (inc a) n) :when (tri-yang? a b)]
(list a b)))
(f 21) ; ((3 4) (5 12) (6 8) (8 15) (9 12) (12 16) (15 20))
```

### 2.7.11 interleave , interpose

interleave	“混合”两个序列:无限序列和定长序列	(interleave (iterate inc 1) ["a" "b" "c"]) ; (1 "a" 2 "b" 3 "c")
interpose	混合分隔元素和 1 个定长序列	(interpose "-" ["a" "b" "c"]) ; ("a" "-" "b" "-" "c")

### 2.7.12 while

通过 atom 明确使用循环变量:

```
(def a (atom 10))
(while (pos? @a) (do (println @a) (swap! a dec)))
```

## 2.8 正则表达式 regex

#"foo" => 表示一个 java.util.regex.Pattern  
#"a?c" ; 匹配 abc aBc aac acc 等

## 2.9 命名空间

### 2.9.1 namespace

参见: <http://clojure.org/namespaces>

Clojure 和 Lisp 一样没有类名和包层次的概念，可能设计函数名不够用的情况。例如要定义一个函数名为 map:

```
(defn map [] "hello")
```

clojure 自带的 map 函数就会被 'user/map 覆盖（用 refer 恢复）。

Clojure 自带 6 个命名空间:

clojure	Clojure 基础命名空间
---------	----------------

clojure.inspector	Swing 监视器
clojure.parallel	并发处理库（实现 JSR166 ForkJoin）
clojure.set	处理 set 集合
clojure.xml	处理 xml
clojure.zip	压缩

启动 REPL 时，创建一个 user 命名空间

函数	区别
ns	自动引入 java.lang 和 clojure.core
in-ns	自动引入 java.lang 但不引入 clojure.core

**\*ns\*** 取得当前 namespace

切换命名空间：

```
(in-ns 'my)
(def v1 10) ; 在 my 名字空间中，v1=10
(in-ns 'user) ; 切换回缺省的 user 名字空间
(def v1 "hello") ; 在 user 名字空间中，v1="hello"
```

在自己开发的函数库文件中定义 namespace：

```
(ns wr3)
(ns wr3.util)
```

同时也引入其他 namespace 和 Java 类空间

```
(ns wr3 (:use clojure.contrib.str-utils) (:import (java.io File)))
```

覆盖 clojure.core 中的函数名：

```
(def vec 0)
(vec [1 2 3]) ; 报错
```

但可以这样用：

```
(clojure.core/vec [1 2 3])
```

恢复 clojure.core 的 namespace：

```
(refer 'clojure.core)
(vec [1 2 3]) ; 恢复可用
```

在 ns 内取消 "+" 函数：

```
(ns-unmap 'user '+)
(+ 3 4) ; 报错
(clojure.core/+ 3 4) ; 7
```

可以部分使用 clojure.core 的函数，部分使用覆盖的：

```
(refer 'clojure.core :exclude '(map set)) ; 恢复这两个外的其他所有 core
```

函数

```
(refer 'clojure.core :only '(println prn)) ; 只恢复两个 core 函数
```

或者其别名：

```
(refer 'clojure.core :rename {'map 'core-map, 'set 'core-set})  
(alias 'set 'clojure.set)
```

## 2.9.2 函数和 macro 的别名

函数直接用 `def` 就可以，如：

```
(def sys-map map)
```

macro 宏必须用如下的方法：

方法 1：

```
(def #^{:macro true} sys-loop #'loop)
```

方法 2：

```
(use 'clojure.contrib.def)  
(defalias sys-loop loop)
```

## 2.9.3 require, use (Clojure 函数空间)

使用全路径名（如果名字有冲突）：

```
(require 'clojure.contrib.math) ; '不能少  
(clojure.contrib.math/round 1.7) ; 2
```

合并到当前命名空间（如果名字没有冲突，相当于 `import static ns1.*`）：

```
(use 'clojure.contrib.math) //  
(round 1.4) ; 1
```

只引入制定的函数：

```
(use '[clojure.contrib.math :only (round floor ceil)]) ; '[]不能少  
(round 1.4) ; 1  
(floor 1.9) ; 1.0  
(ceil 1.1) ; 2.0
```

如果开发过程中，函数库文件更改了，可以不重启就重新装载：

```
(use :reload 'examples.exploring) ; 仅重新装载本函数库  
(use :reload-all 'examples.exploring) ; 把 examples.exploring 用到的函数库也重新装载
```

例子：

注：需要在 Clojure 的启动脚本中包含 `clojure-contrib-1.2.0.jar`。

```
(use '[clojure.contrib.str-utils :only (str-join)])  
(str-join "-" ["hello", "clojure"])
```

注：了解其他

```
(find-doc "ns-")
```

## 2.9.4 import (Java 类空间)

```
(import java.io.File)
```

可以 `import` 同一包内多个 Java 类:

```
(import '(java.io File InputStream)) ; '可以省略  
(File/seperator) ; 注意不能只用 (seperator)
```

或者不同包下的类:

```
(import '(java.io File InputStream) '(java.util Random))
```

## 2.9.5 load-file

在 `classpath` 类路径下的用 `use` 和 `import` 即可。

不在类路径下的可以使用:

```
(load-file "path/to/file.clj") ; unix
```

```
(load-file "f:\\lib\\Clojure\\file.clj") ; win
```

## 2.9.6 列出 ns 下的所有函数

<code>(ns-publics 'clojure.core)</code>	所有 <code>public</code> 的函数
<code>(ns-interns 'clojure.core)</code>	所有函数 (含 <code>private</code> 的)
<code>(ns-imports 'clojure.core)</code>	所有从 <code>java</code> <code>import</code> 的东西
<code>(ns-map 'clojure.core)</code>	含 <code>ns-interns</code> 和 <code>ns-imports</code> 的结果

一般:

```
(require 'wr3.clj.s)
```

```
(map first (ns-publics 'wr3.clj.s))
```

## 2.10 结构 `defstruct`

提示: 推荐使用 `defrecord` 替代 `defstruct`

<code>struct</code>	<code>record</code>
说明: 实质是一个 <code>key</code> 固定的 <code>map</code> , 不能用 <code>dissoc</code> 来删除已有的 <code>key</code>	说明: 实质是生成和 <code>import</code> 一个含参数构造函数的新 <code>class</code>
定义: <code>(defstruct person :name :age)</code>	定义: <code>(defrecord person [name age])</code>
初始化: <code>(struct person "qh" 30)</code>	初始化: <code>(person. "qh" 30)</code>
使用: <code>(:name (struct person "qh" 30))</code>	使用: <code>(:name (person. "qh" 30))</code>

C 用结构体来自定义数据结构，Java 用类来自定义数据结构，Clojure 用 defstruct:

```
(defstruct person :first-name :last-name) ; 声明 struct
(def qh (struct person "qiu" "iuh")) ; 初始化 struct
(def jm (struct person "james" "qiu"))
(:first-name qh) ; "qiu" ; 访问 struct
(:last-name jm) ; "qiu"
(struct-map person :age 30 :first-name "qiu") ; 初始化并新增加属性
```

## 2.11 类 defrecord

对比 map 和 record:

map	record
<ul style="list-style-type: none"><li>● 原生数据结构，不用定义，read-string 可直接读</li><li>● 没有类型和结构保证，需要自己保证正确性。</li></ul>	<ul style="list-style-type: none"><li>● 生成并 import 一个新 java 类，有带参数的构造函数。(defrecord r [name age]) 相当于: class r { public r(String name, int age) {...} }</li><li>● 需要先定义再使用</li><li>● 可使用原始类型，性能更好</li></ul>
(def stu {:fname "james" :lname "qiu" :address {:city "beijing" :zip 1001}})	(defrecord Stu [fname lname address]) (defrecord Address [city zip]) (def stu (Stu. "james" "qiu" (Address. "beijing" 1001)))
(:lname stu) (-> stu :address :zip)	(:lname stu) (-> stu :address :city)
(assoc stu :fname "iuh") (update-in stu [:address :zip] inc)	(assoc stu :fname "iuh") (update-in stu [:address :zip] inc)

map 可以自由定义格式，record 给数据结构限定了一个格式，避免不一致引起的难以发现的 bug。

用途:

- defrecord 定义 Clojure 中类似 java 的类，和 protocol 结合便于被 java 调用。
- 写法比 struct 更好一些。

```
(defrecord person [name age])
(def p1 (new person "qh" 20)) ; 形式 1
(def p2 (person. "james" 30)) ; 形式 2
(:name p1) ; "qh"
(:age p2) ; 30
```

(assoc p1 :name "qiu") ; #:user.person{:name "qiu", :age 20} 但 p1 没有改变  
(dissoc p1 :name) ; {:age 20} p1 也没有改变, 返回新 person

## 2.12 接口 protocol

Clojure1.2 新增加的功能, protocol 定义 Clojure 中类似 java 的接口, 命名一组函数名:

```
(defprotocol father (upgrade [o]))
(defprotocol employer (info [o]))
给 person 类增加 father 和 employer 接口:
(defrecord person [name age y c]
  father (upgrade [o] (str "become father at " (:y o))))
  (extend-type person employer (info [o] (str "work at " (:c o)))))
```

类 person 的实例化对象具备实现的接口函数:

```
(def qh (person. "qh" 35 2010 "nasoft"))
(upgrade qh) ; "become father at 2010"
(info qh) ; "work at nasoft"
```

### 2.12.1 给现有类型增加接口 protocol

可以扩展 protocol 来支持多种类型; 也可以扩展一个类型实现多个 protocol。

例子:

```
(defprotocol P1 (f1 [a] ".."))
(defrecord R1 [s] P1 (f1 [o] (str "R1: " (:s o))))
(f1 (R1. "hello"))
(f1 "hello") ; 报错
```

为 f1 函数参数增加 nil 类型和 string 类型 (注意: P1 放在前面):

```
(extend-protocol P1
  nil (f1 [s] nil)
  String (f1 [s] (str "f1: " s)))
(f1 "hello") ; "f1: hello"
```

也可以使用 extend-type 实现同样的效果 (注意: P1 放在后面):

```
(extend-type String P1
  (f1 [s] s))
(f1 "hello") ; "hello"
```

换个说法, 同一个函数名可以用于不同类型, 例如 cat 函数用于 string、list 和 Number:

```
(defprotocol p1 (cat [a b] nil))
(extend-protocol p1
  String (cat [a b] (apply str (concat a b)))
  java.util.List (cat [a b] (concat a b)))
```

```

        Number (cat [a b] (BigInteger. (str a b)))
    )
使用:
(cat "hello" "world") ; "helloworld"
(cat [1 3 5] [7 9]) ; (1 3 5 7 9)
(cat 10 24) ; 1024

```

### 2.12.2 具体化 reify

```

(defprotocol P1 (f1 [a] ".."))
(let [v 10 o (reify P1 (f1 [this] v))] (f1 o))

```

## 2.13 ->

->	后面的函数迭代使用之前的函数结果作为 <u>第一个</u> 参数，返回最后一次函数调用的值
->>	后面的函数迭代使用之前的函数结果作为 <u>最后一个</u> 参数，返回最后一次函数调用的值
doto	所有的函数始终用最初的那个对象值，最后还是返回最初的那个对象

```

(-> (Math/sqrt 25) int list)

```

相当于:

```

(list (int (Math/sqrt 25)))

```

好处在于:

- 接近于(Math/sqrt 25).int.list 的 Scala 调用习惯
- 更少的(((( )))

对比:

```

(doto (Math/sqrt 25) int list)

```

相当于:

```

(let [o (Math/sqrt 25)] (int o) (list o) o)

```

所以 doto 适用于 o 是可变对象的情况:

```

(doto (java.util.HashMap.) (.put 1 100) (.put 2 200))

```

相当于:

```

(let [o (java.util.HashMap.)] (.put o 1 100) (.put o 2 200) o)

```

当然也可以用:

```

(java.util.HashMap. {1 100 2 200})

```

例子:

```

(-> (/ 144 12) (/ 2 3) str keyword list) ; (:2)

```

相当于:



```
(-> (/ 144 12) (/ ,,, 2 3) str keyword list)
```

相当于:

```
(list (keyword (str (/ (/ 144 12) 2 3)))) ; 第一个结果(/ 144 12)是作为后面函数的第一个参数
```

例子:

```
(def m {:address {:city 'beijing :state 'china}})
(-> m :address :state) ; 'china
```

例子: ->和->>的区别

```
(-> 10 (/ 3)) ; 10/3 10 作为/函数第一个参数
(->> 10 (/ 3)) ; 3/10 10 作为/函数最后一个参数
```

## 2.14 编码规范

<http://dev.clojure.org/display/design/Library+Coding+Standards>

命名	包名: 和文件名相同的小写 <code>wr3/clj/util.clj</code> 函数名: -分隔的小写 <code>make-array</code> 变量名: -分隔的小写
常用名	<code>f,g,h</code> 如果传入的参数是函数 <code>n</code> 一般表示 <code>size</code> <code>index</code> 序号 <code>x,y</code> 数字 <code>s</code> 字符串 <code>coll</code> 集合 <code>pred</code> 条件表达式 <code>&amp; more</code> 不定长参数 <code>expr</code> 宏中的表达式 <code>body</code> 宏的执行体 <code>binding</code> 宏的绑定表达式
粒度	尽量细颗粒化, ( <code>source doseq</code> )看到的是极限了, 当然, 配置文件和 <code>hiccup</code> 生成 <code>html</code> 的代码
私有	如下写法可以存取到私有函数, 例如在测试中 <code>@#'some.ns/var</code> <code>@#'clojure.core/assert-args</code>
hint	只有经测试 <code>type hint</code> 确实有用再去使用
命名冲突	如果名字够好, 尽量去用, 不要怕冲突, <code>namespace</code> 以及 <code>:only</code> 等可以解决问题 在 <code>use</code> 和 <code>require</code> 中尽量多使用 <code>:only</code> 可以避免名字冲突, 实在不行还有 <code>alias</code>
boolean	返回 <code>true/false</code> 的函数或者 <code>boolean</code> 变量都用形如 <code>map?</code> , <code>has?</code> 的名字

### 3、 coll 数据结构

List	单向链表，在头部增加新元素，压栈式	'(1 2 3) (list 1 2 3)
Vect	数组，下标存取，在尾部增加新元素	[1 2 3]
Map	-	{:a 1 :b 2 :c 3} {1 "aa" 2 "bb"}
Set	-	#{1 2 3}

#### 3.1 List

```
(list 1 2 3)


可简写成：



```
'(1 2 3)
```



如果是 symbol 而不是数字或者字符串等，必须用 '



```
'(a b c d)

```


```

用 range 函数生成 list:

```
(range 10) ; (0 1 2 3 4 5 6 7 8 9)
(range 1 10) ; (1 2 3 4 5 6 7 8 9)
(range 1 10 2) ; (1 3 5 7 9)
```

用 repeat 函数生成相同元素的 list:

```
(repeat 5 1) ; (1 1 1 1 1)
(repeat 0 1) ; ()
(apply str (repeat 10 "*")) ; "*****"
```

##### 3.1.1 List 基本操作

CRUD 操作	代码例子
R 读取	<pre>(first '(2 4 6 8)) ; 2 (second '(2 4 6 8)) ; 4 (last '(2 4 6 8)) ; 8</pre> <p>更通用的函数：</p> <pre>(nth '(2 4 6 8) 0) ; 2 (rest '(2 4 6 8)) ; '(4 6 8) (butlast '(2 4 6 8)) ; (2 4 6) (next '(2 4 6 8)) ; '(4 6 8)</pre>

	<pre>(nnext '(2 4 6 8)) ; '(6 8) 相当于(next (next ..)) (nthnext [2 4 6 8 10] 3) ; (8 10) 取第3个及以后元素 rest 和 next 大致相同，不同如下： (rest '(3)) ; () (next '(3)) ; nil</pre>
CU 增加或者 修改	<p>加入元素（只能在前面，与 vector 不用）：</p> <pre>(cons 9 1) ; '(9 2 4 6 8) 结果同下 (conj 1 9) ; '(9 2 4 6 8) 结果同上，conjoin</pre> <p>如果一定要加一个元素到最后，就先把这个元素也变成单个元素的 list 后用 concat</p> <pre>(concat '(1 2 3) (list 4)) ; (1 2 3 4)</pre> <p>在前面加入多个元素：</p> <pre>(list* 1 2 3 [4 5 6]) ; (1 2 3 4 5 6) 把前面的元素都当成元素，最后 一个当成 List</pre> <p>相当于：</p> <pre>(apply list 1 2 3 [4 5 6]) ;</pre> <p>合并 2 个 list：</p> <pre>(concat '(9 7) 1) ; (9 7 2 4 6 8) 可 2 至多个 (into '(9 7) '(2 4 6 8)) ; 把后一个 list 的元素逐个压栈到另一个 list: (8 6 4 2 9 7)</pre> <p>对比[]：</p> <pre>(concat [9 7] [2 4 6 8]) ; [9 7 2 4 6 8] (into [9 7] [2 4 6 8]) ; [9 7 2 4 6 8]</pre>
D 删除	<p>用 remove, filter 或者 take-while, drop-while</p> <pre>(remove #(&gt;= % 7) '(9 7 2 4 6 8)) (filter #(&lt; % 7) '(9 7 2 4 6 8)) ; (2 4 6)</pre>

### 3.1.2 和 java.util.List 互转

java.util.List -> Clojure list	<pre>(def l (doto (java.util.ArrayList.) (.add 1) (.add 3) (.add 5)) (apply list l) 或者 (concat () l)</pre>
Clojure list -> java.util.List	<pre>(java.util.ArrayList. '(1 3 5))</pre>

## 3.2 Vector

### 3.2.1 区别 vec 和 vector 函数

vec: 把其他 seq 转为 vector	vector: 用不定长参数构建新 vector
(vec '(1 2 3)) (vec "hello") ; [\h \e \l \l \o]	(vector 1 2 3) (vector "hello") ; ["hello"]

(vector 1 3 5 7)

可简写成:

[1 3 5 7]

如果是 symbol, 必须加'

'[a b c d]

(quote [a b c d])

### 3.2.2 Vector 基本操作

CRUD 操作	代码例子
R 读取	<p>vector 可以定位元素:</p> <pre>(def v [1 3 5 7]) (nth v 0)</pre> <p>因为 vector <u>是数组</u>, 也可以轻松使用下标存取, 可简写如下  (v 0) ; 从 0 开始第一个元素, 1  (count v) ; 4 相当于 Scala 的 size 或者 length  (v (dec (count v))) ; 最后一个元素, 7  first, rest, next 用法与 list 相同, 没有 second, 但可用 nth 取任何位置元素。</p>
C 增加	<p>concat 得到 list 而不是 vector:</p> <pre>(concat [-3 -1] [1 3 5 7]) ; (-3 -1 1 3 5 7)而非[-3 -1 1 3 5 7]</pre> <p>(into [-3 -1] [1 3 5 7]) ; 这个才返回: [-3 -1 1 3 5 7]  增加元素, conj 在后, cons 在前 (list 调用 conj, cons 都插入在前):  (conj v 0) ; [1 3 5 7 0]  (cons 0 v) ; [0 1 3 5 7]</p>
U 修改	<p>更改指定位置元素:</p> <pre>(assoc [1 3 5 7] 2 0) ; [1 3 0 7] 注意: list*不能*用 assoc 更改</pre> <p>assoc 可连续更改, 如: 交换第一个和最后一个元素  (let [v [1 3 5 7] n (dec (count v))]  (assoc v 0 (v n) n (v 0)))</p> <p>assoc-in 和 update-in  更改[[1 2] [3 4] [5 6]]列表的第[0 1]个元素为 20:  (assoc-in [[1 2] [3 4] [5 6]] [0 1] 20) ; [[1 20] [3 4] [5 6]]</p> <p>通过函数更新:  (update-in [[1 2] [3 4] [5 6]] [0 1] * 10) ; [[1 20] [3 4] [5 6]]</p>

	<pre>(update-in [[1 2] [3 4] [5 6]] [0 1] (fn [i] (* i i i))) ; [[1 8] [3 4] [5 6]]</pre>
D 删除	<p>取片段：</p> <pre>(take 3 [1 3 5 7]) ; [1 3 5] (drop 2 [1 3 5 7]) ; [5 7] (subvec [1 3 5 7] 1) ; [3 5 7] (subvec [1 3 5 7 9] 1 3) ; [3 5]</pre> <p>相当于：</p> <pre>(take 2 (drop 1 [1 3 5 7 9])) ; [3 5]</pre>

例子：把 Map 转为 Vector

```
(reduce into {1 10 2 20 3 30 4 40}) ; [1 10 2 20 3 30 4 40]
```

或者：

```
(vec (apply concat {1 10 2 20 3 30 4 40}))
```

也可以把 Vector 转为 Map：

```
(def v [1 10 2 20 3 30 4 40])
(into {}) (for [i (range 0 (count v) 2)] [(v i) (v (inc i))])) ; {1 10,
2 20, 3 30, 4 40}
```

### 3.2.3 和 list 互转

例子：list<->vector

<b>list -&gt; vector</b>	<pre>(vec '(1 2 3))</pre>
<b>vector -&gt; list</b>	<pre>(lazy-seq [1 2 3]) (seq [1 2 3]) (list* [1 2 3])</pre>

### 3.2.4 和 java 数组互转

java 数组 -> vector	<pre>(vec (into-array [1 3 5]))</pre>
vector -> java 数组	<pre>(into-array [1 3 5])</pre>

## 3.3 Set

```
(set [1 3 5])
```

可简写成：

```
#{1 3 5}
```

```
(sorted-set 3 5 1) ; #{1 3 5}
```

```
(set [1 3 5 3]) ; #{1 3 5}
```

set 合集（加法）：

```
(clojure.set/union #{1 2 3} #{1 2 4}) ; #{1 2 3 4}
```

set 交集:

```
(clojure.set/intersection #{1 2 3} #{1 2 4}) ; #{1 2}
```

set 差集 (减法):

```
(clojure.set/difference #{1 2 3} #{1 2 4}) ; #{3}
```

增加元素:

```
(conj #{1 3} 1 5 7) ; #{1 3 5 7}
```

删除元素:

```
(disj #{1 3 5 7} 3 7) ; #{1 5} disjoin
```

筛选:

```
(clojure.set/select even? #{1 2 3 4 5}) ; #{2 4}
```

注意与 filter 的区别, 返回值类型不同

```
(filter even? #{1 2 3 4 5}) ; (2 4) 返回 list 而不是 set
```

## 3.4 Map

```
(hash-map 1 10 2 20 3 30)
```

可简写为:

```
{1 10, 2 20, 3 30}
```

```
{1 10 2 20 3 30}
```

CRUD 操作	代码例子
R 读取	用 keyword 类型则可前可后, 其他类型做 key 只能放后面用: ( <b>:name</b> {:name "qh" :age 10}) ({:name "qh" :age 10} <b>:name</b> ) ({1 100 2 200} <b>2</b> ) ( <b>get-in</b> {:n "qh", :addr {:cn {:bj {:hd "tsinghua"}}}} [:addr :cn :bj :hd])
CU 增加 或者 修改	没有则增加, 有则修改: ( <b>conj</b> {:name "qh" :age 20} {:age 30} {:gender 'male}) ( <b>merge</b> {:name "qh" :age 20} {:age 30} {:gender 'male}) (reduce <b>into</b> {} [{:name "qh" :age 20} {:age 30} {:gender 'male}]) ( <b>assoc</b> {:name "qh" :age 20} :age 30 :gender 'male)
D 删除	( <b>dissoc</b> {:name "qh" :age 30} :name)

### 3.4.1 读取、选取

{1 100, 2 200, 3 300} ; 注意不是(map 1 100 2 200 3 300), 因为 map 是列表操作符

可省略分隔符", "简写成:

```
{1 100 2 200 3 300}
```

例如:

```
(def m {:name "qh" :age 30})
(get m :name) ; "qh" 方式 1
(:name m)      ; "qh" 方式 2
(m :name)      ; "qh" 方式 3
(:age m) ; 30
(count m) ; 2
(m :mail) ; nil
(get m :mail "anonymous@mail") ; "anonymous@mail" 返回缺省值, 相当于
```

scala 的 getOrElse

或者不用":", 用"'"

```
(def m '{name "qh" age 30})
(m 'name) ; "qh"
(m 'age) ; 30
```

注: 通过 get 函数获取 Map 的元素更安全, 因为能处理 nil 的情况

```
(nil :a) ; 报错
(get nil :a) ; nil
```

层层 get 可以用 get-in

不用 get-in	<pre>((({:n "qh", :addr {:cn :bj :hd "tsinghua"}}} :addr) :cn) :bj) :hd) 或者: (-&gt; {:n "qh", :addr {:cn :bj :hd "tsinghua"}}} :addr :cn :bj :hd)</pre>
用 get-in	<pre>(get-in {:n "qh", :addr {:cn :bj :hd "tsinghua"}}} [:addr :cn :bj :hd])</pre>

参见: assoc-in

例子: destructure

```
(defn m [{a :age}] (println "age is:" a))
(m {:name "qh" :age 30}) ; "age is: 30"
```

例子: 根据 key 取 value:

```
(map1 k1) 或者 (map1 k1 k1-default)
({1 100 2 200} 3 -1) ; 有 key 为 3 的就取其值, 否则返回缺省值
```

例子: 取部分 keys-vals 对:

```
(select-keys {1 100 2 200 3 300} [1 3]) ; {3 300, 1 100}
(select-keys {1 100 2 200 3 300} (reverse [1 3])) ; {1 100, 3 300}
```

组合函数:

```
((comp vals select-keys) {1 100 2 200 3 300} [3 1]) ; {100, 300}
```

例子: 按指定 key 的顺序取出 map 的部分

```
(conj {} (select-keys {1 100 2 200 3 300} [3 1])) ; {3 300, 1 100}
```

例子: 条件选择

```
(def m {1 100 2 200 3 300})  
(select-keys m (for [[k v] m :when (even? k)] k))
```

或者:

```
(into {} (filter #(even? (key %)) m))
```

例子: 取 keys, vals:

```
(keys {:name "qh" :age 30}) ; (:name :age)  
(vals {:name "qh" :age 30}) ; ("qh" 30)  
(keys {1 100 2 200 3 300})
```

相当于:

```
(map key {1 100 2 200 3 300})  
(vals {1 100 2 200 3 300})
```

注意, key 和 val 函数的参数不是一个 map, 而是 map 的一个 Entry<k,v>:

```
(key {1 100}) ; 报错  
(key (first {1 100 2 200})) ; 1  
(first (keys {1 100})) ; 1
```

相当于:

```
(map val {1 100 2 200 3 300})
```

例子: 判断 key 是否已经存在:

```
(contains? {:name "qh" :age 30} :age) ; true
```

**特别注意:** contains? 对 list、vector 也有效, 但 key 是下标而不是值。

```
(contains? [10 20 30 40] 3) ; true 表明存在下标 index=3  
(contains? [10 20 30 40] 40) ; false 没有下标 40, 只有 0,1,2,3
```

### 3.4.2 修改

合并多个 map:

```
(conj {:name "qh" :age 30} {:gender 'm :mail "qh@mail"})  
; {:mail "qh@mail", :gender m, :name "qh", :age 30}
```

或者增加 1 个 [key value] 新元素:

```
(conj {:name "qh" :age 30} [:gender 'm]) ; [] 不能用 assoc
```

也可以用

```
(merge {:name "qh" :age 30} {:gender 'm :mail "qh@mail"})
```



相同 key 可以合并 value:

```
(merge-with + {:name "qh" :age 30} {:gender 'm :age 5})  
; {:gender m, :name "qh", :age 35}
```

增加多个新元素, 返回新 map: (assoc 联合)

```
(assoc {:name "qh" :age 30} :gender 'm :mail "qh@mail")  
; {:mail "qh@mail", :gender m, :name "qh", :age 30}
```

删除多个元素, 返回新 map: (dissoc 分离)

```
(dissoc {:name "qh" :age 30} :name) ; {:age 30}
```

更改值 (用函数):

```
(update-in {:name "qh" :age 30} [:age] #(inc %)) ; {:name "qh", :age 31}
```

简单设置值:

```
(assoc {:name "qh" :age 30} :age 31) ; {:name "qh", :age 31}
```

增加或者修改层次沈的元素:

```
(assoc-in {:name "qh", :addr {:cn {:bj {:hd "tsinghua"}}}} [:addr :cn :bj :hd] "pku")  
; {:name "qh", :addr {:cn {:bj {:hd "pku"}}}}
```

参见: get-in

例子: 使用->进行一系列修改操作

```
(> {:name "qh" :age 10} (dissoc :age) (assoc :name "james"))  
; {:name "james"}
```

例子: 把所有的 value 变大写

```
(def m {:k1 "hello" :k2 "world" :k3 "clojure"})  
(zipmap (keys m) (map #(.toUpperCase %) (vals m)))
```

例子: 按 key 排序:

```
(into (sorted-map) { 1 10 3 30 2 20})
```

例子: 修改 key

```
(into {} (for [[k v] {1 10 2 20 3 30}] [(str "k" k) v])) ; {"k1" 10, "k2"  
20, "k3" 30}
```

### 3.4.3 生成 map

通过 keys 和 vals 来构造 map:

```
(zipmap [1 2 3] [100 200 300]) ; {3 300, 2 200, 1 100}
```

例子: list->map:

```
(into {} (for [k ["Susan" "Barbara" "Ian" "Vicki"]] [k (count k)]))
```

结果: {"Susan" 5, "Barbara" 7, "Ian" 3, "Vicki" 5}

比使用 `zipmap` 好:

```
(binding [k ["Susan" "Barbara" "Ian" "Vicki"]] (zipmap k (map count k)))
```

结果: {"Vicki" 5, "Ian" 3, "Barbara" 7, "Susan" 5}

例子: 特定 `list->map`

```
(def s [1 10 2 20 3 30 4 40])
(into {} (for [i (range (/ (count s) 2))] [(nth s i) (nth s (+ (* 2 i) 1))]))
; {1 10, 10 20, 2 30, 20 40}
```

### 3.4.4 和 vector 互转

把长度为偶数的 `vector` 转为 `map`, 例如: `[1 10 2 20 3 30 4 40 5 50] <-> {1 10 2 20 3 30 4 40}`

```
(def v [1 10 2 20 3 30 4 40])
(def m {1 10 2 20 3 30 4 40})
(defn v2m [v] (into {} (for [i (range 0 (count v) 2)] [(v i) (v (inc i))])) )
(v2m v) ; {1 10, 2 20, 3 30, 4 40}
(defn m2v [m] (reduce into m))
(m2v m) ; [1 10 2 20 3 30 4 40]
```

### 3.4.5 和 java.util.HashMap 互转

`java.util.HashMap` 转换成 Clojure `map`:

```
(def m (doto (java.util.HashMap.) (.put 1 10) (.put 2 20) (.put 3 30)))
(into {} m) ; {1 10, 2 20, 3 30}
```

或者

```
(zipmap (keys m) (vals m)) ; {3 30, 2 20, 1 10}
```

Clojure `map` 转换成 `java.util.HashMap`

```
(java.util.HashMap. {1 10 2 20 3 30}) ; #<HashMap {1=10, 2=20, 3=30}>
```

### 3.4.6 destructure

`destructure`——“结构拆分”在函数定义的参数中（或者变量初始化中）把数据结构拆出部分或者全部，方便调用，避免在函数体中拆分。

例如我们只对 `{:name "qh" :age 30}` 中的 `age` 感兴趣:

例子 1, 一般方法:

```
(defn f1 [m] (:age m))
(println (f1 {:name "qh" :age 30}))
```

使用 destructure:

```
(defn f2 [{age :age}] age)
(println (f2 {:name "qh" :age 30}))
```

例子 2, 再如:

```
(defn f1 [m] (+ (:x m) (:y m)))
(f1 {:x 3 :y 5})
```

使用 destructure:

```
(defn f1 [{x :x y :y}] (+ x y))
(f1 {:x 3 :y 5})
```

或者

```
(defn f2 [{:keys [x y]}] (+ x y))
(f2 {:x 3 :y 5})
```

例子 3, 求勾股弦

```
(defn f3 [xy] (Math/sqrt (+ (* (nth xy 0) (nth xy 0)) (* (nth xy 1) (nth
xy 1)))))
(f3 [3 4]) ; 5.0
```

使用 destructure:

```
(defn f4 [{x y}] (Math/sqrt (+ (* x x) (* y y))))
(f4 [3 4]) ; 5.0
```

也可以在 Map 之外的其他数据结构中进行 destructure, 例如:

```
(let [[x y] [1 2 3]] (+ x y)) ; 4
```

## 3.5 操作

对集合 coll 进行操作最有意思的两个操作是 MapReduce (map/mapcat 和 reduce/reductions)

表 1: 常用

操作	结果	Scala 对应函数
map	$n \rightarrow n$ 或者 $n*m \rightarrow n$	map
mapcat	$n \rightarrow n*m$	flatMap
reduce	$n \rightarrow 1$	reduce          foldLeft foldRight
reductions	$n \rightarrow n$	scanLeft scanRight
for	$n \rightarrow n$	for..yield
for :when	$n \rightarrow m$	for..if..yield
filter remove	$n \rightarrow m$	filter filterNot
split split-with	$n \rightarrow 2$	span partition
partition	$n \rightarrow m*k$	
group-by	$n \rightarrow 2 \sim m$	没有
dotimes doseq dorun	$n \rightarrow \text{nil}$	foreach

juxt	n->m1+m2+...+mi	
------	-----------------	--

表 2: 分类

减少长度 m -> m-	distinct filter/remove for :while/:when keep/keep-indexed
增加长度 m -> m+	cons concat lazy-cat mapcat cycle interpose interleave
取一个 m -> 1	first ffirst nfirst second last nth rand-nth when-first
掐头 m -> 1~m-	rest next fnext nnext nthnext drop/drop-while
去尾 m -> 1~m-	take take-nth take-while butlast drop-last
整理 m -> m~m+	flatten reverse sort sort-by shuffle
分组 m -> 2+	split-at split-with partition partition-all partition-by
映射 m -> m   n*m	map pmap mapcat replace reductions map-indexed seque

### 3.5.1 apply

```
(apply f [e1 e2 e3])
```

```
(apply f e1 [e2 e3]) ; 这个很有用!
```

都是把 sequence 转换函数参数来用，相当于：

```
(f e1 e2 e3)
```

例子：

```
(apply str [1 2 3 4 5]) ; "12345", 对比 (str [1 2 3 4 5]) ; "[1 2 3 4 5]"
```

```
(apply str 1 2 [3 4 5]) ; "12345",对比 (str 1 2 [3 4 5]) ; "12[3 4 5]"
```

```
(apply str "hello:" [1 3 5]) ; "hello:135"
```

```
(apply + 1 [2 3]) ; (+ 1 [2 3])是不行的
```

```
(max [1 3 2]) ; 错误 [1 3 2]
```

```
(apply max [1 3 2]) ; 3 相当于 (max 1 3 2)
```

### 3.5.2 map

模式 1: 操作一个集合	模式 2: 操作多个集合
<code>(map f [a1 a2..an])</code> 相当于: <code>((f a1) (f a2) .. (f an))</code>	<code>(map f [a1 a2..an] [b1 b2..bn] [c1 c2..cn])</code> 相当于: <code>((f a1 b1 c1) (f a2 b2 c2) .. (f an bn cn))</code>

例子:

```
(def v ["aa" "bbb" "cccc"])
(map count v) ; (2 3 4)
```

可以通过 map 把 symbol 转换为 string, 免得敲无数的""

```
(map str '[aa bbb cccc]) ; ("aa" "bbb" "cccc")
```

或者

```
(map name '[aa bbb cccc])
```

上例就变成

```
(map (comp count str) '[aa bbb cccc])
```

可使用匿名函数:

```
(map #(* % %) (range 6)) ; (0 1 4 9 16 25)
```

例子: 多个 coll 做 map

```
(map + [1 2 3] [10 20 30] [100 200 300]) ; (111 222 333)
```

例子:

```
(map vector [1 2 3] [100 200 300]) ; ([1 100] [2 200] [3 300])
```

例子: fib 数列 (1 2 3 5 8 .....

```
(def fib (lazy-cat [1 2] (map + fib (rest fib))))
(take 10 fib) ; (1 2 3 5 8 13 21 34 55 89)
```

原理说明(from SICP):

fib	1	2	3	5	8	13	21	..	..
(rest fib)	2	3	5	8	13	21	34	..	..
[1 2]	3	5	8	13	21	34	55		

keep 和操作一个集合的 map 相似, 但不保留为 nil 的元素, 对比:

```
(map first [[1 2] [] [3 4]]) ; (1 nil 3)
```

```
(keep first [[1 2] [] [3 4]]) ; (1 3)
```

例子: 一种有趣的写法

```
(map '[a b c d e] [0 3]) ; (a d)
```

解析: vector 数组可以直接使用下标来存取, ('[a b c d e] 0)是合法的 (返回'a), 比如下面的写法简单:

```
(map #(nth '[a b c d e] %) [0 3])
```

也体现了 Clojure 中数据 (vector) 即代码 (函数) 的说法。

例子: 数据分段

```
(def c [0 60 80 100])
(map vector (butlast c) (next c)) ; ([0 60] [60 80] [80 100])
```

### 3.5.3 map-indexed

```
(map-indexed f coll)
```

其中 `f` 有两个参数 `[i e]`, `i` 为元素 index, 从 0 开始; `e` 为元素本身。

例子:

```
(map-indexed (fn [i e] (str i "-" e)) [10 20 30])
; ("0-10" "1-20" "2-30")
```

方法 2:

```
(map (fn [i e] (str i "-" e)) (iterate inc 0) [10 20 30])
```

例子(只取 map 中的 val, 带上 index):

```
(def m [{:name "qh" :age 20} {:name "james" :age 30}])
(map-indexed (fn [i r] (conj (map val r) i)) m)
; ((0 "qh" 20) (1 "james" 30))
```

另有 `keep-indexed`, 和 `map-indexed` 类似, 但不保留为 nil 的元素。

### 3.5.4 lazy-cat lazy-cons

`lazy-cat` 是 `concat` 的 lazy 版本, 有时必须使用 `lazy-cat`。

例如: fib 数列 (1 1 2 3 5 8 .....)

```
(defn fib1 [n1 n2] (concat [n1] (fib1 n2 (+ n1 n2))))
(take 10 (fib1 1 2)) ; 出错 java.lang.StackOverflowError
(defn fib2 [a b] (lazy-cat [a] (fib2 b (+ a b))))
(take 10 (fib2 1 2)) ; 正确 (1 2 3 5 8 13 21 34 55 89)
```

这是 fib 最自然易读的写法, 和 Scala 对应的 `Stream` 版本一样:

```
defn fib2(a:Int,b:Int):Stream[Int] = a #:: fib2(b,a+b) // Scala
```

fib 数列的前后项之比接近黄金分割比:

```
(defn fibn [n] (nth (fib2 1 2) n))
(for [i (range 10)] (double (/ (fibr i) (fibr (inc i))))) ; (0.5 0.666 ...
0.618)
```

### 3.5.5 mapcat

用于把 sequence 中的 1 个元素 map 成集合的情况。

map + concat:

```
(map #(repeat 3 %) [1 2 3]) ; ((1 1 1) (2 2 2) (3 3 3))
(mapcat #(repeat 3 %) [1 2 3]) ; (1 1 1 2 2 2 3 3 3)
```

相当于:

```
(reduce concat (map #(repeat 3 %) [1 2 3]))
```

在 Scala 中对应的是 flatMap, 如:

```
1 to 3 flatMap (x=>List.fill(3)(x)) // Scala: Vector(1, 1, 1, 2, 2, 2, 3, 3, 3)
```

例子: 得到多个 zip 文件的所有文件列表

```
(mapcat #(enumeration-seq (.entries (ZipFile. %))) ["1.zip" "2.zip"])
```

例子: 产生  $n \geq i > j \geq 1$  的  $(i\ j)$  序列如  $n=4$  时  $((2\ 1)\ (3\ 1)\ (3\ 2)\ (4\ 1)\ (4\ 2)\ (4\ 3))$

```
(defn f1 [i] (map #(list i %) (range 1 i)))
(f1 4) ; ((4 1) (4 2) (4 3))
;----- *不用 mapcat
(defn f2 [n] (map f1 (range 2 (inc n))))
(f2 4) ; ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
(reduce concat (f2 4)) ; ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
;----- 使用 mapcat
(defn f3 [n] (mapcat f1 (range 2 (inc n))))
(f3 4) ; ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
;----- 使用 mapcat, f1 f3 合在一起
(defn f4 [n] (mapcat (fn [i] (map #(list i %) (range 1 i))) (range 2 (inc n))))
(f4 4) ; ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
;----- 使用 for, 最清楚明白
(defn f5 [n] (for [i (range 2 (inc n)) j (range 1 i)] (list i j)))
(f5 4) ; ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
```

例子: 求集合元素的全排列

```
(defn perm [s]
  (defn rm [e s] (filter #(not= e %) s))
  (if (empty? s) (list ())
    (mapcat (fn [x] (map (fn [p] (cons x p)) (perm (rm x s)))) s)))
```

### 3.5.6 filter

```
(filter (fn [x] (= 0 (mod x 2))) (range 10)) ; (0 2 4 6 8)
```

可简化为:

```
(filter even? (range 10))
```

### 3.5.7 remove

```
(remove zero? [1 2 0 2 4 0 3 5 0]) ; [1 2 2 4 3 5]
```

remove 的操作 filter 都可以完成, 但更直接, 对比:

```
(filter (complement zero?) [1 2 0 2 4 0 3 5 0]) ;
```

### 3.5.8 replace

用法 1: 替换 coll 中的匹配元素,

```
(replace {0 'zero -1 'neg} [1 0 3 0 -1]) ; [1 zero 3 zero neg]
```

替换

源

用法 2: 按照 index 重新排列 (第二个参数是下标)

```
(replace ['a 'b 'c 'd] [3 2 1 0]) ; [d c b a]
```

```
(replace ['a 'b 'c 'd] [0 0 3 3]) ; [a a d d]
```

和如下写法效果一样 (推荐):

```
(map ['a 'b 'c 'd] [3 2 1 0]) ; [d c b a]
```

```
(map ['a 'b 'c 'd] [0 0 3 3]) ; [a a d d]
```

### 3.5.9 peek pop

FIFO 的 sequence 取第一个; FILO 的 sequence 取最后一个。

'(1 2 3)列表是压栈在前面, 所以:

```
(peek '(1 2 3)) ; 1
```

```
(pop '(1 2 3)) ; (2 3)
```

[1 2 3]数组是压栈在后面, 所以:

```
(peek [1 2 3]) ; 3
```

```
(pop [1 2 3]) ; [1 2]
```

### 3.5.10 drop drop-last take take-last

```
(drop 2 [1 2 3 4 5]) ; (3 4 5)
```

```
(drop-last [1 2 3 4 5]) ; (1 2 3 4)
```

```
(drop-last 2 [1 2 3 4 5]) ; (1 2 3)
```

```
(take 2 [1 2 3 4 5]) ; (1 2)
```

```
(take-last 2 [1 2 3 4 5]) ; (4 5)
```

### 3.5.11 reduce

```
(reduce f [a b c d ... z])
```

```
(reduce f a [b c d ... z])
```

就是:

```
(f (f .. (f (f (f a b) c) d) ... y) z)
```

即先把 seq 的头 2 个元素作为 f 函数的 2 个参数运行, 所得结果和第 3 个元素再作为 f 函数的 2 个参数运行, 一直到最后一个元素。



```
(reduce + [1 2 4 5]) ; 12
```

展开来就是：

```
(+ (+ (+ 1 2) 4) 5)
```

对比 reduce 和 apply:

```
(apply + [1 2 4 5])
```

展开就是

```
(+ 1 2 4 5)
```

可见有时 reduce 和 apply 结果一样，但过程不同。

又如：

```
(reduce * (range 1 6)) ; 120
```

展开为：

```
(* (* (* (* 1 2) 3) 4) 5)
```

对比 apply:

```
(apply * (range 1 6))
```

展开为：

```
(* 1 2 3 4 5)
```

例子：求 1000 以下是 3 或 5 的倍数的所有数的 sum

```
(defn f [n] (or (= 0 (mod n 3)) (= 0 (mod n 5))))  
(reduce + (filter f (range 1000))) ; 用 apply 结果一样
```

例子：只能用 reduce，不能用 apply

```
(reduce subs ["hello world" 1 2 3])
```

或者

```
(reduce subs "hello world" [1 2 3])
```

展开为：

```
(subs (subs (subs "hello world" 1) 2) 3) ; "world"
```

例子：如果 f 函数的参数不止 2 个，可用 apply 展开

```
(reduce #(apply subs %1 %2) ["hello world" [0 10] [0 8] [0 6]])  
(reduce #(apply subs %1 %2) "hello world" [[0 10] [0 8] [0 6]])
```

展开为：

```
(apply subs (apply subs (apply subs "hello world" [0 10]) [0 8]) [0 6])
```

进一步展开：

```
(subs (subs (subs "hello world" 0 10) 0 8) 0 6)
```

例子：

```
(reverse [1 2 3])
```

就是：

```
(reduce conj () [1 2 3])
```

展开为：

```
(conj (conj (conj () 1) 2) 3)
```

reverse 可以针对所有 sequence，但效率不高；为了提高效率，[]和 sorted-map 可以使用 rseq

```
(rseq [1 2 3])
```

但(list 1 2 3)等不行：

```
(rseq '(1 2 3)) ; 报错，必须是 clojure.lang.Reversible
```

### 3.5.12 fold-left fold-right

Scala 的 foldLeft、foldRight 可以由 reduce 完成：

- (reduce f x coll)直接对应 coll.foldLeft(x)(f)

如：

```
(reduce / 1 [2 4 8]) ; 1/64= 0.015625
```

```
List(2,4,8).foldLeft(1.0)(_/_) ; Scala
```

- (reduce #(f %2 %1) (reverse [x coll])) 对应 coll.foldRight(x)(f)

如：

```
(reduce #(/ %2 %1) (reverse [1 2 4 8]) ; (/ 1 (/ 2 (/ 4 8)))=1/4=0.25
```

```
List(2,4,8).foldRight(1.0)(_/_) ; Scala 1.0/(2/(4/8.0))=0.25
```

所以可以定义 fold-left, fold-right 如下

```
(defn fold-left [f x coll] (reduce f x coll))
```

```
(defn fold-right [f x coll] (reduce #(f %2 %1) (reverse (cons x coll))))
```

使用：

```
(fold-left / 1 [2 4 8]) ; 1/64
```

```
(fold-right / 1 [2 4 8]) ; 1/4
```

### 3.5.13 reductions

```
(reductions f [a b c d .. z])
```

```
(reductions f a [b c d ...z])
```

就是：

```
[a (reduce f [a b]) (reduce f [a b c]) (reduce f [a b c d]) ... (reduce f [a b c d .. z])
```

相当于 Scala 的 scanLeft

例如：

```
(reductions * [1 2 3 4 5]) ; [1 2 6 24 120]
```

对应 Scala 版本：

```
List(2,3,4,5).scanLeft(1)(_*_) ; List(1, 2, 6, 24, 120)
```

也可以用 loop..recur 完成：

```
(defn scan-left [f i0 seq]
```

```
  (loop [s seq r [i0]] (if (empty? s) r (recur (rest s) (conj r (f (last r) (first s)))))))
```

```
(scan-left + 0 [1 2 3 4 5]) ; [0 1 3 6 10 15] 相当于
```

```
List(1,2,3,4,5).scanLeft(0)(_+_)
```

`(scan-left * 1 [1 2 3 4 5]) ; [1 1 2 6 24 120]` 相当于  
`List(1,2,3,4,5).scanLeft(1)(*_)`

### 3.5.14 split

例子:

```
(split-at 3 (range 10)) ; [(0 1 2) (3 4 5 6 7 8 9)]  
(split-with neg? (range -3 3)) ; [(-3 -2 -1) (0 1 2)]
```

### 3.5.15 partition

`([n coll] [n step coll] [n step pad coll])`

**n:** 每组元素的个数

**step:** 上下组第一个元素之间的距离, 缺省 `step=n`

**pad:** 最后一组元素不够时补足的元素

按个数分组:

```
(partition 2 [1 2 3 4 5 6 7]) ; ((1 2) (3 4) (5 6))  
(partition 2 1 [1 2 3 4 5 6 7]) ; ((1 2) (2 3) (3 4) (4 5) (5 6) (6 7)) ; 步  
长 1  
(partition 3 3 [0 0] [1 2 3 4]) ; ((1 2 3) (4 0 0))
```

### 3.5.16 group-by

```
(group-by neg? [0 1 -2 3 -4 5]) ; {false [0 1 3 5], true [-2 -4]}  
(group-by #( < % 3) (range 10)) ; {true [0 1 2], false [3 4 5 6 7 8 9]}
```

可分多组

例子:

```
(group-by #(cond (> % 0) "+" (= % 0) "0" (< % 0) "-") [1 2 -3 -2 0 4 -1  
0])  
; {"+" [1 2 4], "-" [-3 -2 -1], "0" [0 0]}
```

例子:

```
(group-by count ["aa" "bbb" "cccc" "ddd" "ee"]) ; {2 ["aa" "ee"], 3  
["bbb" "ddd"], 4 ["cccc"]}
```

### 3.5.17 juxt

**juxtaposition** n.并排

`((juxt a b c) x)` 相当于 `[(a x) (b x) (c x)]`

```
((juxt filter remove) even? (range 10)) ; [(0 2 4 6 8) (1 3 5 7 9)]
```

相当于:

```
[(filter even? (range 10)) (remove even? (range 10))]
```

方法 2: 也可以用

```
(map val (group-by even? (range 10))) ; ([0 2 4 6 8] [1 3 5 7 9])
```

方法 3: 或者

```
(use '[clojure.contrib.seq-utils :only [separate]])  
(separate even? (range 10))
```

### 3.5.18 sort

```
(sort [1 3 6 2 5]) ; (1 2 3 5 6)  
(sort > (range 10)) ; (9 8 7 6 5 4 3 2 1 0)
```

或者写成

```
(reverse (range 10)) ; (9 8 7 6 5 4 3 2 1 0)
```

可以结合 comparator(实现 java.util.Comparator)来用:

```
(sort (comparator >) [1 3 6 2 5]) ; [6 5 3 2 1]  
(sort (comparator <) [1 3 6 2 5]) ; [1 2 3 5 6]
```

自定义排序函数:

```
(sort-by #(.toString %) (range 1 11)) ; (1 10 2 3 4 5 6 7 8 9)  
(sort-by #(Math/abs %) [1 -3 2 9 8 -6]) ; (1 2 -3 -6 8 9)  
(sort-by #(- %) [1 -3 2 9 8 -6]) ; (9 8 2 1 -3 -6) 倒序
```

排序 map:

```
(sort-by :age [{:age 10} {:age 3} {:age 5}])  
(sort-by :age > [{:age 10} {:age 3} {:age 5}])
```

例子: 按照指定顺序 sort

```
(let [m '({:k a :v 10} {:k c :v 30} {:k b :v 20} {:k d :v 40})  
      order '(c b a)]  
  (sort-by  
    #((into {} (map-indexed (fn [i e] [e i]) order))) (:k %))  
    m))
```

结果:

```
({:k c, :v 30} {:k b, :v 20} {:k a, :v 10})
```

### 3.5.19 min max min-key max-key

函数	用法
min	(min 3 1 4 1 5 9 2 7) ; 1
max	(apply max [3 1 4 1 5 9 2 7]) ; 9
min-key	(min-key #(Math/abs %) -3 -1 4 -1 5 -9 2 7) ; -1
max-key	(apply max-key #(Math/abs %) [-3 -1 4 -1 5 -9 2 7]) ; -9

### 3.5.20 for

类似于 scala 的 `for..yield`:

```
(for [i (range 10)] (* i i)) ; (0 1 4 9 16 25 36 49 64 81)
(for [i (range 10) :when (> 20 (* i i))] (* i i)) ; (0 1 4 9 16)
```

注: 上例中`(* i i)`算了两次没有必要, 可用`:let`来解决

```
(for [i (range 10) :let [ii (* i i)] :when (> 20 ii)] ii)
```

`map` 对单个集合的操作基本都可以用 `for` 来代替:

例子:

```
(map (comp count str) '[aa bbb cccc]) ; (2 3 4)
(for [i '[aa bbb cccc]] (count (str i))) ; (2 3 4)
```

例子

```
(map #(* % %) [1 2 3 4 5])) ; (1 4 9 16 25)
(for [i [1 2 3 4 5]] (* i i))
```

`for` 比 `map` 少的功能: 不能操作多个集合如`(map + [1 2] [10 20] [100 200])`

`for` 比 `map` 多出来的功能: 可用多个变量, 还可加 `when`。

例子: 乘法口诀表

```
(for [i (range 1 10) j (range 1 10) :when (>= i j)] (str j "x" i "=" (* i j)))
(for [i (range 1 10) j (range 1 (inc i))] (str j "x" i "=" (* i j)))
```

### 3.5.21 every? some

条件	说明
<code>(every? cond seq)</code>	每个都符合 <code>cond</code>
<code>(some cond seq)</code>	返回第一个符合 <code>cond</code> 的结果, 或者 <code>nil</code> 和 Scala 的 <code>find</code> 不一样
<code>(not-every? cond seq)</code>	不是每个都符合 <code>cond</code>
<code>(not-any? cond seq)</code>	没有任何一个符合 <code>cond</code>

注意: 没有 `exists?`, 可用 `some` 来模拟

例子: 判断字符串是否仅仅包含空格字符(" ", "\n", "\t")

```
(defn blank? [s] (every? #(Character/isWhitespace %) s))
```

例子: 判断是否质数

```
(defn prime? [n] (not-any? #(zero? (rem n %)) (range 2 n)))
(filter prime? (range 2 100))
```

注: `contrib` 中的惰性质数序列

```
(use '[clojure.contrib.lazy-seqs :only (primes)])
(take 10 primes)
(nth primes 1000)
```

## 3.6 序列 seq

Clojure 中几乎所有东西都可以抽象成序列 seq。

```
(seq '(1 2 3 4)) ; (1 2 3 4)
(seq [1 2 3 4]) ; (1 2 3 4)
(seq #{1 2 3 4}) ; (1 2 3 4)
(seq {1 2 3 4}) ; ([1 2] [3 4])
```

所有可序列化的数据结构都可以使用相同的函数来处理：

```
(first seq1)
(rest seq1)
(cons e seq1) ; 插前 construct, 构造 List, 也说明该数据结构的不可变性
(conj seq1 e) ; 加后 join
(concat seq1 seq2) ; 连接前后
(into seq1 seq2) ; 压栈 seq2 元素到 seq1
(distinct seq1) ; 去重复
(distinct? 1 2 3) ; true
(distinct? 1 2 1 3) ; false
```

注：原始的 Lisp 使用 cons、car、cdr 作为 List 的基本操作函数

cons	用 2 个元素构造一个 list
car	相当于 first
cdr	相当于 rest

常用函数：

```
(defn p "打印一个 sequence" [seq] (dorun (map println seq)))
或者
(defn p "打印一个 sequence" [seq] (doseq [i seq] (println i)))
```

## 4、函数

Clojure 中函数也是 list 列表：第 1 个元素是函数名，后面是函数参数的列表

### 4.1 函数帮助

```
(doc method-name)
(find-doc "foo")
```

如：

```
(doc print) ; 准确查找
(find-doc "print") ; 模糊 (regex) 查找
(source print) ; 查看源码
```

## 4.2 调用函数

```
(method-name param1 param2 ...)
```

如:

```
(count "hello") ; 5
(println "hello" "world")
```

返回值为 true/false 的函数名一般带一个"?":

```
(string? "hello") ; true
(string? 10) ; false
(keyword? :foo)
(symbol? 'boo)
```

## 4.3 以“函数名”调用

( <b>ns-resolve</b> *ns* 'f)	可指定 ns; 只查找, 不创建, 使用中安全些
( <b>intern</b> *ns* 'f)	可指定 ns; 有则查找, 没有则创建
( <b>eval</b> 'f)	使用当前 ns
( <b>resolve</b> 'f)	使用当前 ns

```
(defn f [n] (* n n n))
((ns-resolve *ns* (symbol "f"))) 10) ; 1000
```

注: \*ns\* 取得当前 namespace

或者使用 intern:

```
(intern *ns* (symbol "f2")) ; 没有 f2 函数就创建一个, 但不绑定
(intern *ns* (symbol "f2") f) ; 绑定到 f2 到 f, 相当于别名
```

也可以使用 eval

```
((eval (symbol "f"))) 10) ; 1000
```

还可以使用 resolve (相当于 ns-resolve \*ns\*):

```
((resolve (symbol "f"))) 10)
```

通过字符串 use 包:

```
(use (symbol "wr3.clj.stringx"))
```

等同于:

```
(use 'wr3.clj.stringx)
```

```
((ns-resolve (the-ns (symbol "wr3.clj.stringx")) (symbol "left"))
"hello-world" "-")
```

或者:

```
(let [n (symbol "wr3.clj.stringx") f (symbol "Left")]
  (require n)
  ((ns-resolve (the-ns n) f) "hello-world" "-"))
```

## 4.4 运行时动态创建函数

```
(defn gen-fn
  [n as b]
  (let [n (symbol n)
        as (vec (map symbol as))
        fn-value (eval `(fn ~n ~as ~b))]
    (intern *ns* n fn-value)))

(gen-fn "foo" ["x"] '(do (println x) (println (inc x))))
(foo 5)
(gen-fn "f1" ["x" "y"] '(* x y) )
(f1 3 5)
```

也可以直接使用 read-string 和 eval:

```
(def f1 (eval (read-string "(fn [x y] (* x y)) (f1 3 5)")))
(f1 3 5)
```

## 4.5 meta

得到函数的 meta

```
(meta map)
```

得到函数参数列表:

```
((meta map) :arglists)
```

得到函数缺省参数个数:

```
(-> map var meta :arglists first count)
```

```
(use 'wr3.clj.s)
(meta (var left))
```

设置和获取对象（基本对象不行）的 meta（类型为 map）:

```
(def v1 (with-meta [1 3 5] {:type "int"}))
(meta v1) ; {:type "int"}
```



## 4.6 定义函数 `defn`

### 4.6.1 基本用法

函数名也可以是未使用的字符，如：`**` ！

不带参数：

```
(defn m1 [] "hello")  
(m1)
```

带参数：

```
(defn m2 [x] (format "hello %s" x))  
(m2 "world")  
(defn m3 [x y] (format "hello %s (%d)" x y))  
(m3 "world" 108)
```

带函数 doc 注释：

```
(defn m "这是 doc 注释" [] ("函数体"))  
(doc m) ; 查看
```

注意：函数注释只能是一个 `string`，而`*不能*`是多个：

```
(defn m  
  "comment 111"  
  "comment 222"  
  [] (..))
```

应该是：

```
(defn m  
  "comment 111  
  comment 222"  
  [] (..))
```

`defn-` 用来定义 `private` 的函数

### 4.6.2 子函数

函数内部定义和使用的函数

例如，定义  $f(x\ y) = x*x + y*y$

```
(defn x2y2 [x y]  
  (defn f [x] (* x x))  
  (+ (f x) (f y)))  
(x2y2 3 4) ; 25
```

### 4.6.3 函数重载

类似于构造方法重载:

```
(defn m1
  ([ ] (m1 "anonymous"))
  ([name] (str "my name is " name)))
```

调用:

```
(m1) ; "my name is anonymous"
(m1 "qh") ; "my name is qh"
```

#### 4.6.4 变长参数

```
(defn f [a b & c] (list a b c))
(= (f 1 2 3 4) '(1 2 (3 4)))
```

```
(defn f2 [a b & c] (list* a b c))
(defn f3 [a b & c] (apply list a b c))
(= (f2 1 2 3 4) (f3 1 2 3 4) '(1 2 3 4))
```

注意:

- &不能紧挨参数, &c 的写法是**错误**的;
- & c 是作为一个 list 出现的, 一般要用 apply 展开

例子 1: 全变长

```
(defn m [& arg] (str arg ", size=" (count arg)))
(m 2 3 4 5) ; "(2 3 4 5), size=4"
```

例子 2: 固定+变长

```
(defn team [leader & persons]
  (format "%s has %d persons: %s" leader (count persons) persons))
(team "qh" '张三 '李四 '王二) ; "qh has 3 persons: (张三 李四 王二)"
```

#### 4.6.5 函数作为参数

定义 C/Java 类型的运算:

```
(defn exp [a f1 b f2 c] (f2 (f1 a b) c))
(exp 5 - 2 + 3) ; 6
```

#### 4.6.6 函数作为返回值

使用匿名函数:

```
(defn f [a] (fn [b] (- a b)))
((f 7) 4) ; 3
```

## 4.7 defmulti 函数名重载

### 4.7.1 by 参数个数

根据参数个数来决定调用的函数，函数重载的加强版：

```
(defmulti f (fn [& args] (count args)))
(defmethod f 0 [& args] "000")
(defmethod f 1 [& args] "111")
(defmethod f 2 [& args] "222")
(defmethod f :default [& args] (str "argn=" (count args) ">3"))
(f) ; "000"
(f 'foo) ; "111"
(f 5 8) ; "222"
(f 'a 'b 'c 'd) ; "argn=4>3"
```

### 4.7.2 by 参数类型

根据参数类型来决定调用的函数：

```
(defmulti f1 class)
(defmethod f1 Integer [x] (double x))
(defmethod f1 Double [x] (int x))
(defmethod f1 :default [x] "unknown")
(f1 2) ; 2.0
(f1 3.14) ; 3
(f1 "abc") ; "unknown"
```

如果是多个参数的类型：

```
(defmulti f2 (fn [x y] [(class x) (class y)]))
(defmethod f2 [Integer Integer] [x y] (* x y))
(defmethod f2 [Double Double] [x y] (+ x y))
(defmethod f2 :default [x y] "others")
(f2 2 3) ; 6
(f2 2.0 3.0) ; 5.0
(f2 2 3.0) ; "others"
```

### 4.7.3 by 参数值

由单个参数的值来判断:

```
(defmulti f3 (fn [x] x))
(defmethod f3 0 [x] (repeat 3 x))
(defmethod f3 1 [x] (repeat 5 x))
(defmethod f3 :default [x] "not 0 or 1")
(f3 0) ; (0 0 0)
(f3 1) ; (1 1 1 1 1)
(f3 7) ; "not 0 or 1"
```

单个参数的值条件:

```
(defmulti f5 (fn [x] (<= 0 x 5)))
(defmethod f5 true [x] (str x " in [0,5]"))
(defmethod f5 false [x] (rem x 10))
(f5 3) ; "3 in [0,5]"
(f5 107) ; 7
```

如果是多个参数的值:

```
(defmulti f4 (fn [x y] (and (> x 0) (> y 0))))
(defmethod f4 true [x y] (- (+ x y)))
(defmethod f4 false [x y] (+ (Math/abs x) (Math/abs y)))
(f4 2 3) ; -5
(f4 -2 3) ; 5
```

由参数值 destructure:

```
(defmulti f6 :name)
(defmethod f6 "qh" [arg] (str "qh: " arg))
(defmethod f6 "james" [arg] (str "james: " arg))
(defmethod f6 :default [arg] (str "unknown: " arg))
(f6 {:name "qh" :age 20})
(f6 {:name "james" :age 30})
(f6 {:name "rich" :age 40})
```

例子: fib 数列的 multi-methods 版本

```
(defmulti fib int)
(defmethod fib 0 [n] 1)
(defmethod fib 1 [n] 2)
(defmethod fib :default [n] (+ (fib (- n 2)) (fib (- n 1))))
(map fib (range 10))
; (1 2 3 5 8 13 21 34 55 89)
```

## 4.8 匿名函数 fn #()

```
(fn [] "hello") ; 仅定义
((fn [] "hello")) ; 定义并调用
((fn [x] x) "hello") ; 带参数
```

很简短的函数可以使用匿名#()

```
(def add #(+ %1 %2))
(= 7 (add 3 4))
```

例子 1: 1 个参数

```
((fn [x] (* 2 x)) 12) ; 24
```

可以简写成:

```
(#(* 2 %) 12)
```

改成偏函数

```
((partial * 2) 12)
```

例子 2: 多个参数

```
((fn [x y] (* x y)) 3 4) ; 12
```

可以简写成:

```
(#(* %1 %2) 3 4)
```

改写成偏函数:

```
((partial * 1) 3 4)
```

例子 3:

```
(map #(.toUpperCase %) ["ab" "cd" "ef"]); ("AB" "CD" "EF")
```

也可:

```
(def ma #(.toUpperCase %))
(map ma ["ab" "cd" "ef"])
```

没有匿名函数之前有个函数叫 memfn, 功能差不多, 但有个匿名函数后基本就被替代不用了:

```
(map (memfn toUpperCase) ["hello" "world"]) ; ("HELLO" "WORLD")
(map #(.toUpperCase %) ["hello" "world"]) ; 对应的匿名函数版本
```

又如:

```
(map (memfn charAt i) ["hello" "world"] [1 2]) ; (\e \r)
(map #(.charAt %1 %2) ["hello" "world"] [1 2]) ; 对应的匿名函数版本
```

注: 使用 memfn 可以把 Java 的对象方法临时变成 Clojure 的函数

例子 3 的 memfn 版本:

```
(def mb (memfn toUpperCase))
(map mb ["ab" "cd" "ef"])
```

例子 4:

```
(filter #(> % 5) (range 10)) ; (6 7 8 9)
```

改成偏函数形式:

```
(filter (partial < 5) (range 10)) ; (6 7 8 9)
```

例子 5:

```
(filter #(> (count %) 2) ["a" "bb" "ccc" "dddd"])
```

对比 Scala:

```
List("a", "bb", "ccc", "dddd") filter (_.size>2) // Scala
```

改写成 comp + partial 版本:

```
(filter (comp (partial < 2) count) ["a" "bb" "ccc" "dddd"])
```

例子 6: 使用 fn 和 AtomicInteger 来跟踪状态变化

```
(def amount (let [a (java.util.concurrent.atomic.AtomicInteger. 100)]  
  (fn [x] (.addAndGet a x))))
```

```
(amount 50) ; 150
```

```
(amount -80) ; 70
```

匿名函数的使用场合:

- 函数体言简意赅, 要起个名字都难于下笔
- 在函数内部创建和使用的函数

## 4.9 偏函数 partial

形如:

```
((partial f arg1 arg2 .. argn) arga argb .. argz)
```

就是执行:

```
(f arg1 arg2 .. argn arga argb .. argz)
```

从名字以及上面可以看出, **partial** 就是整个完整函数执行的前面一部分。

注意: 偏函数的第一个参数是一个函数, 后面至少有 1 个其他参数

部分形式的匿名函数可以改写成偏函数。

**partial** 函数称为“偏函数”或者“部分完整函数”, 因为它是不完整的, 定义也用 **def** 而不是 **defn**。

```
(defn f [n] (* n 10)) ; 正常函数
```

```
(def fp (partial * 10)) ; 偏函数
```

也可以直接调用:

```
((partial * 10) 5) ; 50
```

相当于:

```
(* 10 5)
```

多个参数:

```
((partial * 10) 2 3 4) ; 240
```

相当于:

```
(* 10 2 3 4)
```

例子:

```
(map #(apply str "=" %) [1 2 3] ["+" "+" "+"] [10 20 30])
; ("=1+10" "=2+20" "=3+30")
```

使用 `partial` 的版本:

```
(map (partial str "=") [1 2 3] (repeat "+" [10 20 30]))
```

对比:

fn	(map (fn [x] (* 10 x)) [1 3 5])	(10 30 50)
#()	(map #(* 10 %) [1 3 5])	(10 30 50)
partial	(map (partial * 10) [1 3 5])	(10 30 50)
comp	(map (comp - *) [1 3 5] [2 4 6])	(-2 -12 -30)
complement	(map (complement odd?) [1 2 3]) ; complement 后面的函数需返回 bool 值	(false true false) 逻辑补函数, 取反, 相当于: (map (comp not odd?) [1 2 3])
constantly	(map (constantly 0) [1 2 3])	(0 0 0) ; 常数函数

补函数例子:

```
(filter (complement zero?) [-1 1 2 0 3 4 0 0 5 6 0]) ; (-1 1 2 3 4 5 6)
```

同下:

```
(remove zero? [[-1 1 2 0 3 4 0 0 5 6 0]])
```

## 4.10 组合函数

形如:

```
((comp f1 f2 .. fn) arg1 arg2 .. argn) ; composite
```

就是对参数从右到左组合执行所有函数:

```
(f1 (f2 (... (fn arg1 arg2 .. argn))))
```

```
(defn f [x y] (- (* x y)))
```

可以用组合函数:

```
(def fc (comp - *))
```

```
(fc 3 5) ; -15
```

组合函数按照从右到左的顺序执行。

也可以直接调用:

```
((comp - *) 2 4 6) ; -48
```

这个不用 `comp`, 需要定义变长参数如下:

```
(defn f [& x] (- (apply * x))) ; (f 2 4 6)=-48
```

例子: 得到长度的 10 倍

```
((comp (partial * 10) count) "hello") ; 50
```

例子: 函数当数据来计算

```
((apply comp (repeat 3 rest)) [1 2 3 4 5 6]) ; (4 5 6)
```

相当于:

```
(rest (rest (rest [1 2 3 4 5 6])))
```

但调用 `rest` 的次数不事先确定的时候，只能用 `comp` 来处理。

## 4.11 递归函数

例子 1: 阶乘

```
(defn fac [n] (if (= n 0) 1 (* n (fac (dec n))))) ; (fac 5) = 120
```

自递归写法:

```
(defn fac [n] (defn f [i r] (if (zero? i) r (recur (dec i) (* r i)))) (f n 1))
```

非递归写法:

```
(defn fac1 [n] (reduce * (range 1 (inc n))))
```

或者:

```
(defn ! [n] (reduce * (range 1 (inc n)))) ; (! 5)=120
```

例子 2: fib 数列(1 1 2 3 5 8 13 ...)

```
(defn fib [n] (if (< n 3) 1 (+ (fib (- n 1)) (fib (- n 2)))))
```

自递归的 fib 数列:

```
(defn fib2 [n1 n2 n] (if (= 1 n) n1 (recur n2 (+ n1 n2) (dec n))))  
(fib2 1 1 100) ; 354224848179261915075
```

例子 3: 定义 pow

```
(defn pow0 [n m r] (if (zero? m) r (p n (dec m) (* r n)))) ; 尾递归
```

```
(defn pow1 [n m] (if (zero? m) 1 (* n (pow1 n (dec m))))) ; 递归
```

```
(defn pow2 [n m] (reduce * (repeat m n))) ; reduce
```

```
(defn pow3 [n m] (Math/pow n m)) ; 利用库
```

```
(defn pow4 [n m] (.pow (bigint n) m)) ; 首选，最快 (1.3 要用  
(BigInteger/valueOf n) )
```

```
(= (pow0 2 10 1) (pow1 2 10) (pow2 2 10) (pow3 2 10))
```

例子: 快速排序

```
(defn qsort [s]  
  (if (<= (count s) 1) s  
      (let [m (nth s (int (/ (count s) 2)))]  
        (concat  
          (qsort (filter #(< % m) s))  
          (filter #(= % m) s)  
          (qsort (filter #(> % m) s))))))  
(qsort [1 5 2 1 4 3 7 2]) ; (1 1 2 2 3 4 5 7)
```



## 5、宏 macro

### 5.1 概念

macro 宏在运行之前机械展开；定义宏相当于给语言增加新特性。

写宏的\*原则\*：

- 能写成函数就不要用宏（因为写宏没有写函数简单直观，容易写错，需要先在 REPL 中测试一番）
- 只有不得不用时才用宏（性能要求高时比函数调用快，或者需要“代码<->数据”相互转换）
- 精心设计的宏调用比函数调用更 DSL（如实现控制结构、传递 Java 方法）

例子：

使用宏	不使用宏
<pre>(defmacro op [x f1 y f2 z]   (list f2 z (list f1 x y))) (op 3 + 2 * 10) ; 50</pre>	<pre>(defn op [x f1 y f2 z]   (f2 z (f1 x y))) (op 3 + 2 * 10) ; 50</pre>

注：

- 1、可用 (`macroexpand` '(op 3 + 2 \* 10))来检测展开情况为：(\* 10 (+ 3 2))
- 2、若错误地写成

```
(defmacro op [x f1 y f2 z] (f2 z (f1 x y)))
```

用 REPL 测试一下

```
user=> ('f2 'z ('f1 'x 'y))
```

结果为 y，所以这时

```
(op 3 + 2 * 10) ; 输出 2
```

### 5.2 设计方法

例子：实现 (do f f f)

**X** 不正确写法：

在 REPL 中写：

```
user=> (do 'fun 'fun 'fun)
```

**fun**

所以一旦套上(defmacro macro-name [args] ...)的外衣如下：

```
(defmacro m [fun] (do fun fun fun))
```

那就只会执行一次 fun，例如：

```
(m (println "hello")) ; 仅仅打印 1 次"hello"
```

## V 正确写法:

如果你写:

```
user=> (list 'do 'f 'f 'f)
(do f f f)
```

如果套上(`defmacro macro-name [& args] ...`)的外衣如下:

```
(defmacro m [f] (list 'do f f f))
```

那就执行多次了, 例如:

```
(m (println "hello")) ; 如愿打印 3 次"hello"
```

## 5.3 调试宏

`macroexpand`

`macroexpand-1`

`clojure.walk/macroexpand-all`

## 5.4 ``` `~'` `'~` `~@`

例子:

```
(defmacro with-dict
  "连接到 meta 库的 dict 表进行操作"
  [& body]
  `(let [~'dbname "meta"
        ~'tname :dict
        ~'conn (make-connection ~'dbname)]
    with-mongo ~'conn
    ~@body)))
```

说明:

<code>`</code>	使用就会原原本本地直译过去, 不用 <code>`</code> , <code>let</code> 语句不被翻译
<code>~'</code>	使用则后面的变量被直接翻译过去, 否则翻译成 <code>user/dbname</code> 等
<code>~@</code>	表示多条语句
<code>'~</code>	变量名本身而非值

例子:

```
(defmacro debug [x] `(println "-----" '~x ":" ~x))
(let [a 10] (debug a)) ; "----- a : 10"
```

## 6、调用 Java 的类和方法

和 Java 的直接对应更甚于 Scala:

- String、Number 直接对应
- Collection 实现 java Collection 接口
- 函数实现 java Runnable 和 Callable 接口
- 可以继承 Java 类，实现 Java 接口
- sequence 函数可操作 java 的 String、Array、Iterable

### 6.1 基本用法

内容	写法	简写
构造新对象	(new java.util.Date) (new StringBuffer "hi")	(java.util.Date.) (StringBuffer. "hi") ; 构造方法带参数
对象方法调用	(. "Hello" toUpperCase)	(.toUpperCase "Hello") (.toLocaleString (java.util.Date.))
带参数方法调用	(. "hello" replace "l" "L")	(.replace "hello" "l" "L")
静态类方法调用	(. System currentTimeMillis) (. Math PI) (. Math pow 2 10) ; 1024.0	(System/currentTimeMillis) (Math/PI) ; 注意不是(Math.PI) (Math/pow 2 10) ; 注意不是(Math.pow 2 10)
连续方法调用	(. (. (. System currentTimeMillis) getClass) toString)	(. System currentTimeMillis getClass toString) (. Runtime availableProcessors) (. "hello" toUpperCase (substring 1 4))
多次方法调用 (无返回值的 方法)	(let [m (java.util.HashMap.)] (.put m 1 100) (.put m 2 200) m)	(doto (java.util.HashMap.) (.put 1 100) (.put 2 200)) ; doto 的意思: do something to it (m)

```
(import java.io.File) ; 单个
```

```
(import '(java.io File InputStream)) ; '可以省略
```

```
(import '(java.io File InputStream) '(java.util Random))
```

例子 1:

```
(def rdm (java.util.Random.))  
(.nextInt rdm)
```

```
(.nextInt rdm 10)
```

例子：3 种形式，结果一样

```
(Math/PI)
(. Math PI)
(.. Math PI)
```

使用 Math/PI 的形式更普遍：

```
(= (Math/toRadians 180) Math/PI) ; true
(= (Math/toDegrees Math/PI) 180) ; true
```

## 6.2 得到所有 Java 类方法

```
(map #(.getName %) (.getMethods java.util.Date))
(map #(.getName %) (.. "hello" getClass getMethods))
```

## 6.3 Java 数组

可查阅：

```
(find-doc "-array")
```

按类型有：

函数	用法
object-array	(object-array [10.3 "hello" 10])
boolean-array	(boolean-array 5)
byte-array	(boolean-array [false true])
char-array	(boolean-array 5 [false true])
double-array	二维数组：
float-array	(into-array (map int-array [[1 2 3] [4 5 6]]))
int-array	
long-array	
short-array	
into-array	
make-array	其他非基本类型，如 String、Integer、Double
to-array	
to-array2d	
byte-array-type	
char-array-type	

例如：

```
(seq (object-array [10.3 "hello" 10])) ; (10.3 "hello" 10)
```

不够数目的用缺省值填充:

```
(seq (boolean-array 5)) ; (false false false false false)
(seq (boolean-array [false true false])) ; (false true false)
(seq (boolean-array 5 [false true])) ; (false true false false false)
```

```
(seq (int-array 5)) ; (0 0 0 0 0)
(seq (int-array [2 4 6])) ; (2 4 6)
(seq (int-array 5 [2 4])) ; (2 4 0 0 0)
```

创建 Java 类型的数组:

```
(make-array String 5) ; 创建 new String[5]
(make-array String 5 3) ; 创建 new String[5][3]
(seq (make-array String 5)) ; (nil nil nil nil nil)
```

set/get 数组元素:

```
(def a (make-array String 3))
(aset a 0 "hello") ; aset: array setting
(aset a 2 "world")
(seq a) ; ("hello" nil "world")
(aget a 0) ; "hello"
(alength a) ; 3, 和 (count a) 一样
```

Clojure 的 list 转换为 Object 数组:

```
(to-array [1 2 3 4]) ; java.lang.Object[] 注意不会转为 int[] 或者 Integer[]
```

主要用于调用 Java 的方法:

```
(String/format "%s*%s=%d" (to-array [2 3 6])) ; "2*3=6",
```

当然, 上例使用 Clojure 的 format 函数更方便:

```
(format "%s*%s=%d" 2 3 6)
```

如果需要明确指定类型, 可以使用 into-array:

```
(into-array [1 2 3]) ; java.lang.Integer[]
(import '(wr3.util Stringx))
(Stringx/join (into-array String ["1" "2" "3"]) "+")
```

如果是字符串数组等, 会智能转换类型, 不需要 into-array:

```
(Stringx/join [1 2 3] "+") ; "1+2+3"
```

## 6.4 reflect 调用 Java 方法

```
(clojure.lang.Reflector/invokeInstanceMethod "hello-world"
"substring" (to-array [1 4]))
```

相当于:

```
(.substring "hello-world" 1 4) ; "ell"
```

## 6.5 Java 方法作为函数参数

```
(defmacro f [g] `(.. "Hello" ~g))
(f toUpperCase) ; "HELLO"
(f toLowerCase) ; "hello"
```

例子：更复杂的情况，可变参数

```
(defmacro f [g & args] `(.. "Hello" (~g ~@args)))
(f substring 0 3) ; "Hel"
(f substring 3)   ; "lo"
```

## 6.6 设置属性值

```
(set! (. obj f1) v1)
```

## 6.7 JavaBean

bean 得到 Java 对象的所有 JavaBean properties:

```
(bean (java.util.Date.))
```

例子:

```
(import '(java.security MessageDigest))
(bean (MessageDigest/getInstance "SHA"))
```

返回:

```
{:provider #<Sun SUN version 1.6>,
 :digestLength 20,
 :class java.security.MessageDigest$Delegate,
 :algorithm "SHA"}
```

## 6.8 提升性能

### 6.8.1 类型提示

正常版本:

```
(defn f0 [n] (reduce + (range (inc n)))) ; 1 to n sum
```

性能版本:

```
(defn f1 [n]
  (let [n (int n)]
    (loop [i (int 1) s (int 0)]
      (if (<= i n) (recur (inc i) (+ i s)) s))))
```

测试:



```

        pid0 (.toString (.get row 1))
        name (.toString (.get row 2))
        pid (IDUtil/to18 pid0)]
      (do
        (when pid (do (swap! i inc) (println (line id pid
name))))))
      true) ]))]]

(doto dbs
  (.process sql row-filter)
  (.close))
(println "i=" @i) )

```

## 6.10 Exception

形如: (try (throw ..) (finally ..))

例如:

```

(try (throw (Exception.
"--error--")) (finally (println
"--final--")))

```

输出:

```

--final--
java.lang.Exception: --error-- (NO_SOURCE_FILE:0)

```

例子 2:

```

(defn f [cls]
  (try (Class/forName cls) true (catch ClassNotFoundException e
false)))

```

例子 3:

```

(try (/ 3 0) (catch Exception e (println e)))

```

## 6.11 Java 调 Clojure

```

import clojure.lang.RT;
import clojure.lang.Var;

```

```

public class Foo {
  public static void main(String[] args) throws Exception {
    // Load the Clojure script -- as a side effect this initializes the
runtime.
    RT.loadResourceScript("foo.clj");
    // Get a reference to the foo function.

```



```

    Var foo = RT.var("user", "foo");
    // Call it!
    Object result = foo.invoke("Hi", "there");
    System.out.println(result);
  }
}

```

## 6.12 编译

如果不发布.clj 代码，则需要编译，如果是可执行类，需要如下的函数：

test1.clj:

```

(ns test1 (:gen-class))
(defn -main [& args] (println "hello main"))

```

在 REPL 中编译：

```

(binding [clojure.core/*compile-path* "."]
(clojure.core/compile 'test1))

```

运行：

```
java -cp clojure.jar;. test1
```

使用命令行编译：

```

java -Dclojure.compile.path=.
-cp .:clojure.jar:clojure-contrib-1.2.0.jar clojure.lang.Compile test1

```

## 6.13 调用 OS 系统功能

从 Clojure 中调用系统 shell 命令：

```

(use '[clojure.java.shell :only [sh]])
(sh "java")
(sh "java" "-version")

```

从 Clojure 中调用 web 浏览器访问指定 url：

```

(use 'clojure.java.browse)
(browse-url "http://clojuredocs.org")

```

调用 web 浏览器查看在线 java 类或者对象的 javadoc：

```

(javadoc String)
(def d (java.util.Date.))
(javadoc d)

```

## 7、正则表达式 regex

```
(re-pattern "\\d+")
```

可简写成:

```
#"\\d+"
```

进行完全匹配:

```
(re-matches #"\\d+" "1920") ; "1920" 注意不是返回 true, 不为 nil 就是匹配  
(re-matches #"\\d+" "19-20") ; nil
```

逐步列出所有匹配项:

```
(def m (re-matcher #"\\d+" "from 1900 to 2010"))  
(re-find m) ; "1900"  
(re-find m) ; "2010"  
(re-find m) ; nil
```

或者

```
(def m (re-matcher #"\\d+" "from 1900 to 2010"))  
(loop [e (re-find m)] (if e (do (println e) (recur (re-find m)))))
```

用 when 更好:

```
(def m (re-matcher #"\\d+" "from 1900 to 2010"))  
(loop [e (re-find m)] (when e (println e) (recur (re-find m))))
```

找到第一个匹配项:

```
(re-find #"\\d+" "from 1900 to 2010") ; "1900"
```

一次列出所有匹配项:

```
(re-seq #"\\d+" "from 1900 to 2010") ; ("1900" "2010")  
(re-seq #"\\w+" "get all words list") ; ("get" "all" "words" "list")
```

## 8、并发 STM

Erlang	解决多台机器之间的并发控制
Clojure	解决单台机器上多线程之间的并发控制

## 8.1 基本概念

ref	引用	管理对一个状态或多个状态的协同、同步更新。 采用 STM 的交易管理。
atom	原子	管理对单个状态的非协同同步更新
agent	代理	管理对状态的异步更新 可采用 STM 的交易管理。
var	变量	本地线程变量，参看 def let binding

STM 的基本原理：

Clojure 的 STM 采用 MVCC (Multiversion Concurrency Control) ,和一般 RDB 用的概念是相同的。

交易 A 以 Clojure 唯一的时间戳来标识，交易 A 通过该时间戳标识。

## 8.2 ref

定义：

```
(def v1 (ref 10))
```

引用：

```
(deref v1)
```

可简化为：

```
@v1
```

改变：

```
(dosync (ref-set v1 0))  
(dosync (ref-set v1 (inc @v1))) ; 这个写法
```

使用函数改变值，可简化为：

```
(dosync (alter v1 inc))  
(dosync (alter v1 + 10))
```

或者用：

```
(dosync (commute v1 + 100)) ; commutative 交替的
```

改变多次：

```
(def cu (ref 10))  
(dotimes [i 5] (dosync (alter cu inc))) ; @cu = 15
```

Clojure 中 STM 的特性和 RDB 类似：

- 更新操作是原子操作
- ref 可以指定 validation 函数，失败可完全回滚
- 多个更新操作之间是无关的，一个更新操作不能读取另一个更新操作的中间结果

定义一组原子操作：

```
(def v1 (ref 10))  
(def v2 (ref 100))
```

```
(dosync
  (ref-set v1 20)
  (ref-set v2 200)) ; @v1=20 @v2=200
```

例子:

```
(def c (ref []))
(defn next1 [] (dosync (alter c conj (System/nanoTime))))
(defn next2 [] (dosync (commute c conj (System/nanoTime))))
(dotimes [i 10] (next1))
(dotimes [i 10] (next2))
```

例子: 转账

```
(def a1 (ref 100))
(def a2 (ref 0))
(defn transfer [from n to] (dosync (when (>= @from n) (alter from - n) (alter
to + n))))
(transfer a1 60 a2) ; @a1=40 @a2=60
(transfer a1 60 a2) ; @a1=40 @a2=60
(dotimes [_ 10] (transfer a1 10 a2)) ; @a1=0 @a2=100
(dotimes [_ 10] (transfer a2 10 a1)) ; @a1=100 @a2=0
; 并发转
(use 'wr3.clj.numberx)
(dotimes [i 110] (future (do (Thread/sleep (* 10 (random))) (transfer a1
1 a2))))) ;
```

### 8.2.1 validator

例子: 聊天室

(def **nonnull?** (partial every? #(and (:from %) (:text %)))) ; 判断是否都有值的 partial 函数

(def board (ref [] **:validator** **nonnull?**)) ; 定义黑板

或者用:

```
(def board (ref []))
(set-validator! board nonnull?)
(defn add2 [m] (dosync (alter board conj m)))
```

(defstruct msg :from :text) ; 定义消息

(def m1 (struct msg "qh" "qh here"))

(def m2 (struct msg "james" "james here"))

(do (add2 m1) (add2 m2))

(add2 (struct msg "xx" "hello world"))

(add2 "some unknow") ; 失败, 找不到非 nil 的 :from 和 :text

(add2 (struct msg nil "hello world")) ; 失败

(add2 (struct msg "xx" nil)) ; 失败

### 8.2.2 watch

`watch` 是在 `ref`、`atom`、`agent` 状态改变时会被调用的函数，可以设置多个 `watch`：

```
(def v (ref 10))
(defn w0 [key id old new] (println "key =" key "id =" id))
(defn w1 [key id old new] (println "old =" old "new =" new))
(add-watch v "watch1" w1)
(add-watch v "watch2" w0)
(dosync (alter v inc)) ; old = 10 new = 11 \n key = watch2 id =
#<Ref@560932fe: 11>
@v ; 11
(remove-watch v "watch2")
(dosync (alter v inc)) ; old = 11 new = 12
```

## 8.3 atom

`atom` 基于 `java.util.concurrent.atomic.*` 实现。

`atom` 比 `ref` 更轻量级，`atom` 不能同时改变多个变量的值，但存取速度更快。

```
(def v1 (atom 10))
@v1 ; 10
(reset! v1 20) ; @v1=20
(swap! v1 + 3) ; @v1=23
```

如果 `atom` 要同时改变多个变量的值，可以把多个变量组合成一个数据结构，如：

```
(def v2 (atom {:name "qh" :age 30}))
@v2
(reset! v2 {:name "james" :age 20}) ; @v2={:name "james", :age 20}
(swap! v2 assoc :age 25) ; @v2={:name "james" :age 25}
```

## 8.4 agent

```
(def v (agent 3))
@v ; 3
(send v inc) ; 发出指令后马上返回，让 agent 把指令排入队列中在合适的时候执行
@v ; 4
(send v * 10)
@v ; 40
```

等待 `agent` 执行结果：

```
(await v)
(await-for 100 v)
```

使用 validator:

```
(def v1 (agent 3 :validator number?))
(send v1 (fn [_] "abc")) ; #<Agent@786c1a82: 3>
@v1 ; 3
(send v1 inc) ; java.lang.RuntimeException: Agent is failed, needs
restart (NO_SOURCE_FILE:0)
(agent-errors v1) ; (#<IllegalStateException java.lang.IllegalStateException: Invalid
reference state>)
(clear-agent-errors v1) ; 3
(send v1 inc) ; @v1=4
```

## 8.5 var

```
(def v 0)
(binding [v 10] (println v) (set! v 100) (println v))
(println v) ; 0
```

lazy 变量

```
(def v1 (delay (System/nanoTime))) ; 定义但不求值
(System/nanoTime) ; 14870352 36144105
(force v1) ; 14870352 39977365 第一次使用时求值
@v1 ; 14870352 39977365 之后保持不变
(System/nanoTime) ; 14870352 46999304
```

## 8.6 状态更新对比

更新机制	更新手段	ref	atom	agent
常用方式	通过一个函数	(dosync (alter v1 inc))	(swap! v1 inc)	(send-off v inc)
通信方式	通过一个函数	(dosync (commute v1 inc))		
非阻塞方式	通过一个函数			(send v inc)
简单设值方式	通过一个值	(dosync (ref-set v1 0))	(reset! v1 10)	

## 8.7 多线程

```
(.start (Thread. #(do (Thread/sleep 3000) (println 10))))
```

；注：上述语句马上返回，3s 后在控制台打印 10

使用 Clojure 的 future 版本如下：

```
(future (do (Thread/sleep 3000) (println 10)))
```

例子：

```
(dotimes [i 5] (future (do (Thread/sleep 3000) (println 10))))
```

例子：

```
(doseq [msg ["one" "two" "three" "four"]] (future (println "Thread" msg "says Hello World!")))
```

## 8.8 pmap

map 的多线程版本，对比：

```
(defn f [n] (when (pos? n) (recur (dec n)))) ; 一个啥也不做的耗时操作
(time f 1000000000) ; 1150ms
(time (dotimes [i 2] (f 1000000000))) ; 单线程版本 2321ms
(time (doall (pmap f (repeat 2 1000000000)))) ; 多线程版本 2091ms
```

例子：查看文件夹大小（字节数）

```
(defn psize [f]
  (if (.isDirectory f)
    (apply + (pmap psize (.listFiles f)))
    (.length f)))
(import java.io.File)
(psize (File. "f:/lib/clojure"))
```

## 9、GUI

```
(import 'javax.swing.JFrame)
(doto (JFrame. "hello world") (.setSize 200 200) (.setVisible true))

(doto (javax.swing.JFrame.)
  (.setLayout (java.awt.GridLayout. 2 2 3 3))
  (.add (javax.swing.JTextField.))
  (.add (javax.swing.JLabel. "Enter some text")))
```

```
(.setSize 300 80)
(.setVisible true))
```

例子:

```
(ns gui-demo (:import [javax.swing JPanel JFrame] [java.awt Dimension]))
```

```
(defn panel2 []
  (let [panel (proxy [JPanel] [] (paintComponent [g] (.drawLine g 0 0 100
100)))]
    (doto panel (.setPreferredSize (Dimension. 400 400)))))
```

```
(defn frame2 [panel]
  (doto (JFrame.) (.add panel) .pack .show))
```

```
(frame2 (panel2)) ; 调用
```

例子: 摄氏度转换为华氏度

```
(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener)
        '(java.awt GridLayout))

(let [frame (new JFrame "Celsius Converter (摄氏度转华氏度)")
      temp-text (new JTextField)
      celsius-label (new JLabel "Celsius (摄氏度)")
      convert-button (new JButton "Convert (进行转换)")
      fahrenheit-label (new JLabel "Fahrenheit (华氏度)")]
  (. convert-button
    (addActionListener
      (proxy [ActionListener] []
        (actionPerformed [evt]
          (let [c (Double/parseDouble (. temp-text (getText)))]
            (. fahrenheit-label
              (setText (str (+ 32 (* 1.8 c)) " Fahrenheit (华
氏度)"))))))))
  (doto frame
    ;(.setDefaultCloseOperation (JFrame/EXIT_ON_CLOSE)) ; uncomment this to
quit app on frame close
    (.setLayout (new GridLayout 2 2 3 3))
    (.add temp-text)
    (.add celsius-label)
    (.add convert-button)
    (.add fahrenheit-label)
    (.setSize 300 80)
    (.setVisible true)))
```



## 10、 IO JDBC

### 10.1 文件 IO

#### 10.1.1 全部读

一次读入全部内容:

```
(use 'clojure.contrib.duck-streams)
(slurp "somefile.txt") ; 把文件内容读入字符串 slurp: 咔吃
(def s (slurp "somefile.txt"))
```

可以指明 encoding:

```
(slurp "gbk.txt" :encoding "gbk")
```

写字符串到文件:

```
(spit "output.txt" "some output text") ; 写字符串内容入文件。spit 原意:
吐口水
```

```
(append-spit "output.txt" "\nmore text with spit append") ; 追加文件内
容
```

参数:

```
:append true
:encoding "UTF-8"
```

#### 10.1.2 逐行读

每次读入一行:

(0) 使用 java.io.\*

```
(with-open [r (java.io.BufferedReader. (java.io.FileReader.
"gbk.txt"))]
  (let [seq (line-seq r)]
    (count seq)))
```

(1) 使用 clojure.java.io/reader

例子: 打印短行

```
(with-open [rdr (clojure.java.io/reader "gbk.txt" :encoding "gbk")]
  (doseq [line (line-seq rdr)] (when (< (count line) 10) (println
line))))
```

例子：所有行放入 vector

```
(with-open [rdr (clojure.java.io/reader "gbk.txt" :encoding "gbk")]
  (reduce conj [] (line-seq rdr)))
```

例子：计算特定行的数目

```
(def n (atom 0))
(with-open [rdr (clojure.java.io/reader "gbk.txt" :encoding "gbk")]
  (doseq [line (line-seq rdr)] (when (< (count line) 10) (swap! n inc))))
@n
```

例子：简单实现 wc -l cust.txt

```
(with-open [rdr (clojure.java.io/reader "cust.txt" :encoding "gbk")]
  (count (line-seq rdr)))
```

例子：标记行号

```
(use '[clojure.contrib.seq-utils :only (indexed)])
(with-open [r (clojure.java.io/reader "gbk.txt" :encoding "gbk")]
  (doseq [[i line] (indexed (line-seq r))] (println i ":" line)))
```

## (2)使用 clojure.contrib.duck-streams

```
(use '[clojure.contrib.duck-streams :only (reader)])
(line-seq (reader "output.txt"))
```

或：

```
(with-open [l (reader "output.txt")] (vec (line-seq l)))
```

或：

```
(with-open [r (reader "output.txt")] (doseq [l (line-seq r)] (println
l)))
```

注：with-open 会自动关闭[]内的每个值。

### 10.1.3 文件树

得到文件树：

```
(file-seq (java.io.File. "."))
(count (file-seq (java.io.File. ".")))
```

## 10.2 网络 IO

### 10.2.1 读文本

```
(with-open [s (.openStream (java.net.URL. "http://g.cn"))]
  (let [buf (java.io.BufferedReader. (java.io.InputStreamReader. s))
        seq (line-seq buf)])
```

```
(apply str seq)))
```

### 10.2.2 读二进制

```
(let [con (-> "http://g.cn" java.net.URL. .openConnection)
      fields (reduce (fn [h v] (assoc h (.getKey v) (into [] (.getValue
v))))
      {} (.getHeaderFields con))
      in (java.io.BufferedInputStream. (.getInputStream con))
      out (java.io.BufferedOutputStream. (java.io.FileOutputStream.
"out.file"))
      buffer (make-array Byte/TYPE 1024)]
(loop [g (.read in buffer) r 0]
(if-not (= g -1)
(do
(println (String. buffer "UTF-8"))
(.write out buffer 0 g)
(recur (.read in buffer) (+ r g))))))
(.close in)
(.close out)
(.disconnect con))
```

## 10.3 配置、数据文件

Clojure 不用象其他语言一样使用 xml、json、yaml 之类的外部格式来存储数据，使用 Clojure 代码即数据就能很地完成。

把代码结构写入文件：

```
(def dbs '[oracle mssql sybase db2 mysql informix postgresql])
(spit "conf3.clj" dbs)
```

文件内容为：

```
[oracle mssql sybase db2 mysql informix postgresql]
```

读出使用：

```
(def c (read-string (slurp "conf3.clj")))
```

凡是 Clojure 表达出来的数据结构都可以进行存储，如：

```
(def m {:name "qiuji" :gender 'male :age 20 :mail ["qh@mail"
"james@msn.com"]})
```

；注意：在 1.conf 中 'male 是没有 ' 的

```
(spit "1.conf" m) ; spit 写入可能会乱码，自行编辑改成 utf8 即可
(def c (read-string (slurp "1.conf")))
```

如果 2.conf 内容为：

```
(* 2 3 4)
需要配合 eval
(eval (read-string (slurp "2.conf")))
```

### 10.3.1 读写 json

需要使用 json 与 js 或者其他语言交换数据时:

```
(use 'clojure.contrib.json)
(json-str [1 2 3 4]) ; "[1,2,3,4]"
(read-json "[1,2,3,4]") ; [1 2 3 4]
```

其他:

pprint-json	打印 pretty 格式的 json
print-json	打印 json
(write-json o out)	输出到 out

### 10.3.2 读写 xml

使用 clojure.xml 来解析 xml

例如 a.xml:

```
-----
<?xml version="1.0" encoding="UTF-8" ?>
<persons meta="测试 clojure.xml">
  <count>3</count>
  <person id="qh">
    <name>qh</name>
    <age>16</age>
    <email>qh@mail</email>
  </person>
  <person id="james">
    <name>james</name>
    <age>26</age>
    <email>james@mail</email>
  </person>
  <person id="qiuiuh">
    <name>qiuiuh</name>
    <age>36</age>
    <email>qiuiuh@mail</email>
  </person>
</persons>
-----
```

代码:

```
(use 'clojure.xml)
(def p (parse (java.io.File. "a.xml")))
```

每个节点会被解释为含{:tag .. :attr .. :content ..}三部分的 map, 例如 persons 根节点

每部分标识	值	说明
:tag	:persons	总是一个 key 类型变量
:attr	{:meta "测试 clojure.xml"}	一个 map, 属性名为 key 类型
:content	[{:tag :name, :attrs nil, :content ["qh"]} {:tag :age, :attrs nil, :content ["16"]} ...]	一个节点/字符串数组, 每个节点又被递归解析。

使用 xml-seq 来处理每个节点 (本例是对有属性的节点打印其属性):

```
(doseq [n (xml-seq p) :when (:attrs n)] (println (:attrs n)))
```

结果:

```
{:meta 测试 clojure.xml}
{:id qh}
{:id james}
{:id qiuiuh}
```

## 10.4 JDBC 数据库 IO

参考: [http://en.wikibooks.org/wiki/Clojure\\_Programming/Examples/JDBC\\_Examples](http://en.wikibooks.org/wiki/Clojure_Programming/Examples/JDBC_Examples)

; 引用 contrib 包

```
(use 'clojure.contrib.sql) ; open "test_sql.clj" for examples.
```

; 定义 DataSource

```
(def db {
  :classname "org.h2.Driver",
  :subprotocol "h2",
  :subname "mem:testdb",
  :user "sa",
  :password ""})
```

; 查询 sql

```
(with-connection db
  (with-query-results rs ["select 'qh' name,20 age"]
    (doseq [r rs] (println r))))
```

; 创建, 删除 Table

```
(with-connection db (transaction (create-table :t1 [:name1 :varchar]
[:age1 :int])))
```

```

(with-connection db (transaction (try (drop-table :t1) (catch Exception
_))))
; 增加插入数据
(with-connection db (transaction (insert-values :t1 [:name :age] ["qh"
20])))
; 查询数据
(with-connection db (with-query-results rs ["select * from t1"] (dorun (map
#(println %) rs))))
; update 数据
(with-connection db (transaction (update-values :t1 ["name=" "qh"]
{:age 35})))
; 删除数据
(with-connection db (transaction (delete-rows :t1 ["name=" "qh"])))

```

## 10.5 ClojureQL

<http://www.clojureql.org/>

用更 DSL 的方式书写 SQL 语句，自动转化为 SQL92，支持 MySQL 和 Postgresql

## 10.6 MongoDB 操作

<https://github.com/somnium/congomongo>

[MongoDB 文档](#)

驱动源码: [congomongo.clj](#)

测试源码: [congomongo\\_test.clj](#)

### 10.6.1 创建连接

初始化:

```

(use 'somnium.congomongo)
(mongo! :db "cbs400")
(set-database! "test")

```

或者更正规:

```

(with-mongo (make-connection "cbs400"))
...)

```

可以连接多个 mongodb:

```

(def db (make-connection :example
  {:host "server1"}
  {:host "server2" :port 25017}))

```

## 10.6.2 CRUD

CRUD	代码示例	MongoDB 命令
Create 增	( <b>insert!</b> :cust {:name "qh" :age 30})	db.cust.save( {name:"qh",age:30})
Read 查	( <b>fetch-one</b> :cust) ; p1 ( <b>fetch</b> :cust :limit 10)	db.cust.findOne() db.cust.find().limit(10)
Update 改	( <b>update!</b> :cust p1 (merge p1 {:name "james"})) ( <b>update!</b> :cust {:_id "01" {:name "james"}}) ( <b>update!</b> :cust {:_id "01" {"\$set" :name "james"}}) ; 更新单个字段 存在则更新, 没有则增加 (update! :coll {: _id x} {: _id x :name "Joe" :age 20}))	db.cust.update({name:"qh"}, {\$set:{name:"james"}}, true) 存在则更新, 没有则增加 coll.update( { _id: X }, { _id: X, name: "Joe", age: 20 }, true);
Delete 删	( <b>destroy!</b> :cust p1) ( <b>destroy!</b> :cust {:name 'QH' :age 30})	db.cust.remove({name:"qh"})

高级操作	代码示例	MongoDB 命令
插入多条	( <b>mass-insert!</b> :cust [{:name "a"} {:name "b"} {:name "c"}])	
条件查询 :where	(fetch :cust <b>:where</b> {:name "a" :age {">" 30}}) (fetch :cust <b>:where</b> {:age {"\$exists" true}}) (fetch :cust <b>:where</b> {:name {"\$in" ["qh" "qiu"]}) (fetch :cust <b>:where</b> {:age {"\$gt" 30 "\$lt" 50}}) (fetch :cust <b>:where</b> {"\$or" [{:name "qh"} {:age 30}]})) (fetch :cust <b>:where</b> {:name #"^q"}) ; 用正则表达式相当于 sql 中的 like	db.cust.find( {name:"a",age:{\$gt:30}}) db.cust.find({age:{\$exists:true}}) db.cust.find( {name:{\$in:['qh','qiu']}) db.cust.find( {age:{\$gt:30,\$lt:50}}) db.cust.find({"\$or" [{:name:'qh'},{:age:30}]})) db.cust.find({name: /^q/}) db.cust.find().min()
查询范围	(fetch :cust <b>:skip</b> 10 <b>:limit</b> 3)	db.cust.find.skip(10).limit(3)
选择指定列 :only	(fetch :cust :only ["name" "age"])	db.cust.find({}, {name:1, age:1})
排序	(fetch :cust <b>:sort</b> {:name 1})	db.cust.find().sort({name:1})
count	( <b>fetch-count</b> :cust)	db.cust.find().count()

distinct	(distinct-values :cust "ptype") ; 注意列名用 str 类型	db.cust.distinct("ptype")
index	(add-index! :cust [:org :sex] :name "myIndex2") (get-indexes :cust) (drop-index! :cust [:org [:sex -1]])	db.cust.ensureIndex( {org:1,age:1})  db.cust.dropIndexes()
插入文件	(fetch-files :cust) (insert-file! :cust (.getBytes "blah..") :filename "f1" :contentType "dir1/file1") (fetch-one-file :cust :where {:filename "f1"}) (destroy-file! :cust {:filename "f1"}) (insert-file! :cust (.getBytes "blah.."))	
map-reduce		
server-eval		
删除库	(drop-database! "test1")	
删除集合	(drop-coll! :cust)	

### 10.6.3 object-id

每条记录都有内置 mongodb 自动生成的 `:_id` 字段，用户可以自行制定但需要自行保证唯一性。

objectId <--> 字符串:

(-> p :id str)

(object-id "4de706a71da3906ba3240b83") ; 必须是 24 位

## 11、 Unit Test

```
(use 'clojure.contrib.test-is)
(deftest f1 [] (is (= 10 (+ 7 3))))
(deftest f2 [] (is (= 10 (+ 7 13))))
(deftest f3 []
  (are (= _1 _2)
    2 (+ 1 1)
    4 (* 2 2)))
(f1) ; nil
(f2) ; 打印出错信息
```

注:

```
(are (= _1 _2)
```



```
2 (+ 1 1)
4 (* 2 2))
```

模板会按照变量`_1`, `_2` 展开为:

```
(do
  (is (= 2 (+ 1 1)))
  (is (= 4 (* 2 2))))
```

## 12、 web 开发

一些新的趋势:

- 精通 JavaScript, 使用 jQuery 和 Google Closure
- Mobile First
- Html5 应用: 可离线运行
- 使用函数式编程和 key-value
  - Server 端 Clojure: 函数式, key-value (map)
  - Client 端 JavaScript: 函数式, key-value (json)
  - DB 端 NoSQL: 函数式, key-value

### 12.1 Ring

类似于 Python's WSGI and Ruby's Rack

<https://github.com/mmcgrana/ring> (主页)

<https://github.com/laurentpetit/ring-java-servlet> (Servlet)

<http://mmcgrana.github.com/2010/03/clojure-web-development-ring.html> (教程)

步骤:

- 下载项目工具 lein.bat (<https://github.com/technomancy/leiningen>)
- 运行 (下载 jar 包):

```
lein self-install
```
- 从下载 Ring 解压 (版本 0.3.5)
- 在 Ring 解压后的目录下运行 (下载 jar 包到 lib 目录及 maven 目录下):

```
lein deps
```
- 创建应用 hello\_world.clj:

```
-----
(use 'ring.adapter.jetty)
(defn app [req]
  {:status 200
   :headers {"Content-Type" "text/html; charset=utf-8"}
   :body    "Hello World from Ring (cn 中文)"})
(run-jetty app {:port 8080})
```

- 运行:

```
java -cp "lib/*" clojure.main hello_world.clj
```

- 例子 2: wrapping.clj:

```
-----
(use 'ring.adapter.jetty)
(defn app [req]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body "Hello World from Ring"})
(defn wrap-upcase [app]
  (fn [req]
    (let [orig-resp (app req)]
      (assoc orig-resp :body (.toUpperCase (:body orig-resp))))))
(def upcase-app (wrap-upcase app))
(run-jetty upcase-app {:port 8080})
-----
```

- 运行:

```
java -cp "lib/*" clojure.main wrapping.clj
```

Request Map 中有如下内容:

:server-port	8080
:server-name	"localhost"
:remote-addr	"0:0:0:0:0:0:0:1"
:uri	"/"
:query-string	nil
:scheme	:http
:request-method	:get (可以是:head, :options, :put, :post, :delete)
:content-type	nil
:content-length	nil
:character-encoding	nil
:headers	{       "accept-charset" "GBK,utf-8;q=0.7,*;q=0.3",       "accept-language" "zh-CN,zh;q=0.8",       "accept-encoding" "gzip,deflate,sdch",       "user-agent" "Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) ..",       "accept" "application/xml, text/html;q=0.9,text/plain;..",       "connection" "keep-alive",       "host" "localhost:8080"     }

	}
:body	#<Input org.mortbay.jetty.HttpParser\$Input@87b7b11>

Response Map 中的内容

:status	>=100
:headers	{"Content-Type" "text/html; charset=utf-8"}
:body	String ISeq File InputStream

## 12.2 Compojure

对 Ring 的 DSL 化包装

### 12.2.1 session

```
(ns pkg1 (:use compojure))

(defroutes helloworld-routes
  (GET "/login/:user"
    [(session-assoc :loggedin (:user params))
     (html [:html [:body [:p "You are now logged in as " (:user params)]]]])
  (GET "/logout"
    (if (contains? session :loggedin)
      [(session-dissoc :loggedin)
       (html [:html [:body [:p "Logged out"]]])
       (html [:html [:body [:p "You are not logged in"]]])
      ])
    (GET "/*"
      (if (contains? session :loggedin)
        (html [:html [:body [:p "Welcome back " [:b (:loggedin session)]]]])
        (html [:html [:body [:p "Hello World"]]])))
    ))

(decorate helloworld-routes (with-session :memory))

(run-server {:port 8080} "/*" (servlet helloworld-routes))
```

### 12.3 Conjure

### 12.4 Weld

### 12.5 hiccup

<https://github.com/weavejester/hiccup>

cljs-html 的加强版，用法相同：vector 来对应 tag，map 来对应 tag 的属性

例子：

```
(use 'hiccup.core)
(html [:span {:class "foo"} "bar"])
```

输出：

```
<span class="foo">bar</span>
```

例子：

```
(html [:html [:head [:style "h1 {color:red}"]] [:body [:h1 "hello world"]]])
```

输出：

```
"<html><head><style>h1          {color:red}</style></head><body><h1>hello
world</h1></body></html>"
```

例子：用#表示 id，用.foo 表示 class

```
(html [:div#id1.cls2 "..."])
```

输出：

```
<div class="cls2" id="id1">...</div>
```

例子：循环

```
(html [:table (for [tr (range 3)] [:tr (for [td (range 2)] [:td (str tr
td)])])] )
```

输出：

```
<table>
<tr><td>00</td><td>01</td></tr>
<tr><td>10</td><td>11</td></tr>
<tr><td>20</td><td>21</td></tr>
</table>
```

#### hiccup.core

简单 tags	[:br]	 
带 content 的 tag	[:span "baz"]	<span>baz</span>

带 attr 的 tag	<code>[ :span { :id "foo", :class "bar" } "baz" ]</code>	<code>&lt;span class="bar" id="foo"&gt;baz&lt;/span&gt;</code>
id 和 class	<code>[ :span#foo.bar "baz" ]</code>	<code>&lt;span class="bar" id="foo"&gt;baz&lt;/span&gt;</code>
html 转码	<code>(escape-html "&lt;\\"foo\\"&amp;\\"bar\\"" )</code>	<code>&amp;lt;&amp;quot;foo&amp;quot;&amp;amp;&amp;quot;bar&amp;quot;</code>
以上简写	<code>(h "&lt;\\"foo\\"&amp;\\"bar\\"" )</code>	<code>&amp;lt;&amp;quot;foo&amp;quot;&amp;amp;&amp;quot;bar&amp;quot;</code>

### hiccup.page-helpers

Doctypes: html4/5, xhtml-strict, xhtml-transiti onal	<code>(doctype :html4)</code>	<code>&lt;!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.d td"&gt;</code>
XHTML tag	<code>(xhtml-tag "en" "foo")</code>	<code>&lt;html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml" &gt;foo&lt;/html&gt;</code>
JavaScript 包含	<code>(include-js "foo.js" "bar.js")</code>	<code>&lt;script src="foo.js" type="text/javascript"&gt; &lt;/script&gt; &lt;script src="bar.js" type="text/javascript"&gt; &lt;/script&gt;</code>
CSS 包含	<code>(include-js "foo.css" "bar.css")</code>	<code>&lt;script src="foo.css" type="text/javascript"&gt; &lt;/script&gt; &lt;script src="bar.css" type="text/javascript"&gt; &lt;/script&gt;</code>
JavaScript 嵌入	<code>(javascript-tag "document.write('' hello world');")</code>	<code>&lt;script type="text/javascript"&gt; //&lt;![CDATA[ document.write('hello world'); //]]&gt;&lt;/script&gt;</code>
超连接	<code>(link-to "http://foo.bar" "baz")</code>	<code>&lt;a href="http://foo.bar"&gt;baz&lt;/a&gt;</code>
<ul>	<code>(unordered-list ["foo" "bar"]</code>	<code>&lt;ul&gt; &lt;li&gt;foo&lt;/li&gt;&lt;li&gt;bar&lt;/li&gt;&lt;li&gt;baz&lt;/li&gt;</code>

	"baz"])	> </ul>
<ol>	(ordered-list ["foo" "bar" "baz"])	<ol> <li>foo</li><li>bar</li><li>baz</li> > </ol>

### hiccup.form-helpers

<form>	(form-to [:POST "/foo"] "...")	<form action="/foo" method="POST"> <input id="_method" name="_method" type="hidden" value="POST" />...</form>
<label>	(label :foo "bar")	<label for="foo">bar</label>
<hidden>	(hidden-field :foo "bar")	<input id="foo" name="foo" type="hidden" value="bar" />
<text>	(text-field :foo "bar")	<input id="foo" name="foo" type="text" value="bar" />
<password>	(password-field :foo "bar")	<input id="foo" name="foo" type="password" value="bar" />
<checkbox>	(check-box :foo true "bar")	<input checked="checked" id="foo" name="foo" type="checkbox" value="bar" />
<radio>	(radio-button :foo true "bar")	<input checked="checked" id="foo-bar" name="foo" type="radio" value="bar" />
<option>	(select-options [:foo :bar :baz] :bar)	<option>foo</option> <option selected="selected">bar</option> <option>baz</option>
<select>	(drop-down :foo [:bar :baz] :baz)	<select id="foo" name="foo"> <option>bar</option> <option selected="selected">baz</option> </select>
<text>	(text-area :foo "bar")	<textarea id="foo" name="foo">bar</textarea>

<file>	(file-upload :foo)	<input id="foo" name="foo" type="file" />
<submit>	(submit-button "submit")	<input type="submit" value="submit" />
<reset>	(submit-button "reset")	<input type="submit" value="reset" />

## 12.6 Enlive 模板引擎

<https://github.com/cgrand/enlive>

<https://github.com/swannodette/enlive-tutorial/>

## 13、网络资源

- <http://clojuredocs.org/>
- <http://www.clojure-toolbox.com/>  
常用 lib 分类列表
- <http://clojure-examples.appspot.com/>
- <http://java.ociweb.com/mark/clojure/article.html>  
在线教程: Clojure - Functional Programming for the JVM
- <http://clojure-euler.wikispaces.com/>  
ProjectEuler 的 Clojure 解答
- [http://en.wikibooks.org/wiki/Clojure\\_Programming/Examples/Cookbook](http://en.wikibooks.org/wiki/Clojure_Programming/Examples/Cookbook)  
Clojure Programming/Examples/Cookbook
- <http://www.gettingclojure.com/cookbook:clojure-cookbook>  
Clojure cookbook
- <http://alexott.net/en/clojure/video.html>  
Video lectures & presentations about Clojure
- <http://langref.org/>  
几个语言的常用功能代码对比
- <http://clojars.org/>  
Clojure 的库 repo
- <http://www.glenstampoultzis.net/blog/clojure-web-infrastructure/>  
Clojure Web Infrastructure
- <http://thegeektalk.com/>  
Geek Talks, 使用 Clojure、Scala 的 geek 挺多
- <http://nakkaya.com/>

个人 blog, 使用 Clojure 处理摄像头、图像等

- [http://www.ic.unicamp.br/~meidanis/courses/mc336/2006s2/funcional/L-9\\_9\\_Ninety-Nine\\_Lisp\\_Problems.html](http://www.ic.unicamp.br/~meidanis/courses/mc336/2006s2/funcional/L-9_9_Ninety-Nine_Lisp_Problems.html)

99 个 Lisp 练习题, 对应的 Scala 网站: <http://aperiodic.net/phil/scala/s-99/>

## 14、临时

### 14.1 InfoQ 采访 Clojure 实用

**InfoQ:** 你能简要介绍一下你们的 Clojure 代码是如何组织的吗? 比如, 如何使用命名空间、muti-methods (多重分派)、macros (宏) 等等。

过往函数式编程的经验使我们能够将代码组织成非常“函数”的形式。我们使用命名空间的方式与在其它语言中使用命名空间的方式相同。我们尽量少用 macros 和 muti-methods, 就算用也只是在适合的地方使用。人们对 Lisp 的所有印象就是其有大量的 macros 和 meta-sauce。尽管从某些方面来说这是对的, 但是用 FP 基础构建块, 你能走得更远; 如 lambdas、HOFs、currying、partial application 等等。

我不能算是精通 monad, 但是我觉得 monads 和 multi-methods 及 macros 都差不多。这些抽象概念不但功能强大, 而且也很酷, 吸引了不少眼球。当这些抽象概念有助于简化开发时, 我就会设法使用这些工具。但是这要求思维缜密, 使用原始的函数编程方式及数据结构你也能做得很好。

最后谈一下 Clojure 中的 monads, 有些人认为 monads 是用于状态的, 可是 Clojure 已经有了许多有效方法来处理状态, 为什么还要用 monads。我们没有使用过 state monad, 但是用过许多其他很有用的 monads。我很欣赏 Brian Beckman 对 monads 的描述: 他们只是伪装的函数合成物而已。使用这一概念的最大好处是, 能够安全构造出中间可能失败或碰到空值的多阶段计算。

虽然 Destructuring bind 看起来并不像 macros 和 monads 那样引人注目, 但在实践中它实际上是一个非常强大的抽象概念。Rich 选择从模式匹配中解耦 destructuring bind 的方法非常聪明。我相信很快在 Clojure 里就会有 ML 风格的模式匹配, 这一切正在实现中。

**InfoQ:** 你提到过用于将数据格式输入到 Clojure 数据结构的 Clojure reader: 你在使用 Reader macros 吗?

Clojure 没有 reader macros, 我们也压根不需要。要读写 Clojure 数据结构, 我们只需使用 print-dup 语句即可, 它可以让我们定义 multimethods 来分派 printers。只有在你需要读写没有内建的类型时, 才真正需要实现 Printer。例如, 我们有一个针对 joda-time 日期的特殊 printer, 那么我们也将以一种特殊的方式将他们读回。

**InfoQ:** 你提到过编写 Clojure 数据结构——你是用其来序列化传输、存储或其它用途的数据吗?



我们将 **Clojure 数据结构用作通信和存储的中间表示**。例如，我们所有数据转换工作的输出都是 Clojure 数据结构形式，这种中间表示遍布于我们所有的 Hadoop 工作中。这是我们置于 Cascading 和 Hadoop 之上的那一层东西的关键，可以让我们免于处理 Hadoop 输入格式。

**InfoQ:** 你们有什么想加到 Clojure 或 Clojure 生态系统中的东西么（类库、工具……）？有人提议加个高质量的 **destructuring 模式匹配工具**，这个东西挺有用。但我对局部改进更感兴趣。我们没有用模式匹配，而且 Clojure 有许多很好的抽象概念，因而并不会缺失太多模式匹配，但是我认为它还是让很多地方的代码更加整洁了。

如果 Rich 开放该 reader，我想一定很酷，或许这将帮助为模式匹配和 monad 实现创建良好的语法。其次，Clojure 是我第一次使用 Lisp，因此我并没有使用 reader macros 的经验，因此，在这方面我并不在行。

如果 Rich 不保留竖线也挺好，这样我们可以将它作为我们核心 DSL 的一部分，用来作为条件概率记号。:-)

**InfoQ:** 对于你所使用的 Clojure 类库，你有什么建议？

使用 Clojure 的第一个建议是：**Clojure-core 和 Clojure-contrib 都很小，因此最好通读全部代码**。从中你将发现非常好的东西。留意所有神奇的数据结构及数据结构处理函数。例如，Clojure 有一个友好的 Set 及 Set 操作实现，以及一些准关系代数实现。

我把 Clojure 看作是一种面向函数编程的数据结构。

Clojure 拥有一套神奇的数据结构。此外，所有这些数据结构都有书面陈述，因此 reader 和 destructuring 都很自然。所有这些结合在一起让人非常愉快。

传承自 ML 的函数语言中，函数是类型签名。而在 Clojure 中，函数是数据结构拓扑签名。

## 14.2 Lisp 用户的问题

<http://www.lambdassociates.org/blog/bipolar.htm>

进行 Lisp 相关编程 20 年，并参看了 Lisp 程序员在 Usenet 和 blog 上的诸多发言，我常想获知是否存在形如人种特征的某种“Lisp 特征”。

经过思量，我觉得 Lisper 身上确实有一些反应 Lisp 趣史及其优缺点的特征存在。

毫无疑问，本文会引发持不同意见者的恼怒和争论：

### **Lisper 的两面性**

每位大学老师在教学生涯中都带毕业过数百数千名学生，大部分印象模糊与你见面打招呼聊半天也记不起来这是那届的大哥大姐。但有些你一定记住：在你手下做过项目混了脸熟的，班上特别优秀的和特别差劲的。最好的和最差的都能印象深刻地留在脑子里，这确实挺奇妙的，这也是本文写“两面性”这个标题的原因。

中学生中，处于中间的，能力差的，聪明但不优秀的那些学生我另文讨论。我现在想讨论那部分以聪明且成绩优异的中学生，这些娃有两特征：聪明敏感、但有点二。他们有点二对事不严肃是因为事情对他们来说太过简单乏味。这些人的另一个特征是接触一项工作之初

很有热情，但短时间之后便厌烦，并在事情完全完工之前撒手不管，之后他/她会无所事事地躺在床上弹吉它。这种人一般有人格的两面性：有时很忧郁，有时又很激昂。

这些中学生上了大学之后，可能有两种境遇：一种是在所选专业上大有所为，证实了自己的天才；一种是发现大学和中学一样无趣，并非随处是大师以及和他一样聪明的同学。和中学不同的是，大学没人来管束，他就从此不振作了，但大学没有中学那么容易念，仅靠聪明是得不了 A 的，不努力的后果是只能得 B+ 甚至 B、C。

这种聪明但成绩垫底的学生在大学里比比皆是。他们如果能在最后学期呈上一份优秀的毕设还好，否则大部分教授不会给他们很好的毕业评价。

当然这类学生是不会无所事事消停的，他们的时间基本都花在阅读和学习新东西上，因为天生聪明的脑袋瓜如饥似渴，需要不断往里塞东西。

现在回到 Lisp 上来，我很早就发现 Lisp 真的对这类学生很有吸引力。理解这点就能理解是些什么样思维的人对 Lisp 文化做了贡献，也就了解了 Lisp 和他的拥趸一样，是个聪明的失败者。Lisp 和聪明的两面性格者（brilliant bipolar mind——BBM）都有独特的优点和缺点。

BBM 们喜欢做抱负远大的事情，并认为这些事情需要巨大的资源，而资源总是不够。

Lisp 作为工具，就像是手中的杠杆，他比更低级的语言如 C 语言等更能放大使用者的构想和设计，使用 Lisp 的人喜欢嘲笑使用其他语言的程序员。

BBM 们热爱 Lisp。Lisp 极富魅力的独创性反映了 BBM 的创造力。Lisper 们有很多先驱性的原创：GC 垃圾回收、list 列表处理、PC、Window 等。

BBM 们作为原创者，在拥有巨大创造力的同时也有其弱点，那就是不能把一个事做到最后完工，不能把先驱概念做到完善可用。“可抛弃的设计”一词绝对来自 Lisp 社区的 BBM。因为用 Lisp 构建新东西很容易，所以摒弃已有系统自造轮子就成了常事。10 年前当我需要在 Lisp 中找一个 GUI 库来用的时候，一下子发现 9 个不同的，但没有一个提供合适的文档和完善无 bug，每个作者都有自己的实现，在他自己那里运行良好。这就是 BBM 的态度：在我这里没问题，我清楚这东西。这是不需要也不想要他人来协助的产品。

C/C++ 的方式与此大相径庭。由于做出点东西来不容易，就会敝帚自珍仔细文档化。在大型 C 项目中要依赖社区他人的帮助，要和他人协同工作。

从雇主的角度来看，后者更有吸引力。10 人共同工作交流，有完善的文档，这比雇佣一个玩 Lisp 的 BBM 黑客更好，后者走人了只能再找另一个 BBM（如果有幸能找到的话），并且他一般不会同意接手别人的工作，而更愿意重头再来一套。

BBM 们的另一方面的特点是对技巧的敏感性，他们能一针见血地看到现有事物的荒谬性，并知道该如何弄好它。而且他们也不喜欢折中方案。

Lisp 机器是这种思维的产物，他们拒绝为市场而进行妥协。

BBM 们的最后一个特点是：在逆境中容易悲观。如果你读过大量关于 Lisp 的讨论话题（包括 comp.lang.lisp 上那些说“Common Lisp Sucks”的话题），就会发现此言不虚。编程老鸟也对他们所钟情的 Lisp 语言的不足之处倍感忧伤。而这些问题是可解决的（Lisp 方言 Qi 就是一个例子），但你仅沉沦于悲观就啥也解决不了。

佐证以上所述的最佳材料是经典文章：Lisp: Good News, Bad News, How to Win Big。如果读了这篇文章，你就能感觉到 BBM 的性格。这边文章的不同之处在于作者同时把 BBM 的两面都真实展现了出来。

基本上存在 2 个问题：Lisper 的问题和 Lisp 语言的问题。Lisper 的问题就是 BBM 的问题。Lisp 的问题是 Lisper 引起的。

问题的答案是：Lisp 本身没有问题，因为 Lisp 就和 Life 一样，看你怎么过。

## 14.3 Rich Hickey 访谈

<http://www.simple-talk.com/opinion/geek-of-the-week/rich-hickey-geek-of-the-week/>

**RM:** Rich, plenty of people must have been tired of programming in C++/Java/C# as you did but simply put up with it. What was the trigger to develop Clojure? Was it a matter of waking up in the middle of the night and realising what the problem was?

**RH:** One watershed moment for me was when I designed and implemented a new system from scratch in Common Lisp, which was a new language for me at the time, only to have the client require it be rewritten in C++, with which I was quite proficient. The port to C++ took three times as long as the original work, was much more code, and wasn't significantly faster. I knew then that I wanted to work in a Lisp, but realized that it had to be client-compatible. I worked on bridging Common Lisp and Java, while doing functional-style programming for clients in C#, for a couple of years, then bit the bullet and wrote Clojure.

对我来说，这事的分水岭是：当时，有一个我用刚了解的 **Common Lisp** 语言来设计和实现的新系统，由于用户的需要改用我精通的 **C++** 来改写，整个改写耗费了 3 倍的时间以及多得多的代码量，但性能却没有明显的提升。通过此事我就觉得必须用 **Lisp** 来干活，但我又得保证客户兼容性，很多时候这需要桥接 **Common Lisp** 和 **Java**，还需要为客户用 **C#** 写函数式风格的代码，几年后，当我有了足够的储备，开发 **Clojure** 就水到渠成了。



## 14.4 语言结构决定人类思维方式及行动方法

<http://nklein.com/2009/02/sapir-whorf-wit-programming-languages/>

作者对比了 Objective-C 和 Lisp，认为 Lisp 为进行功能的函数化分离提供了更多的便利，使编程者更愿意进行此类重构，而 Objective-C/C++/Java/Perl 的使用者维护几处相同的代码的可能性更大。

用 Lisp 构建新函数，不用为起函数名发愁，不用为参数列表太多而发愁，不用为 .h 和 .c 中重复定义而发愁，不用为区分函数和方法而发愁，不用为返回多个值发愁。

## 14.5 The Joy of Clojure 笔记

Clojure: beautiful and practical design principles

## 14.6 On Lisp 笔记

Lisp-1 和 Lisp-2 的区别：

Lisp-1 的变量和函数名在同一个空间

Lisp-2 的变量和函数名在 2 个空间，可以同时有一个变量 `f` 和一个函数 `f`

Clojure 和 Scheme 是 Lisp-1，CLisp 是 Lisp-2

## 14.7 Clojure on Heroku

[http://blog.heroku.com/archives/2011/7/5/clojure\\_on\\_heroku/](http://blog.heroku.com/archives/2011/7/5/clojure_on_heroku/)

## 14.8 Clojure 1.3 changelog

<https://github.com/clojure/clojure/blob/1.3.x/changes.txt>

要点：

- Clojure1.2 编译出来的 .class 不能用 Clojure1.3 的 jar 来执行
- clojure.set, clojure.xml, clojure.zip 需要手动引入才能用
- replicate 函数和 repeat 重复，被去除
- 增加 clojure.data/diff，比较两个 map

```
(use 'clojure.data)
(diff {:a 10 :b 20} {:b 20 :c 30}) ; ({:a 10} {:c 30} {:b 20})
得到[things-only-in-param1 things-only-in-param2 things-in-both]
```

- 增加 `every-pred`、`some-fn` 函数
 

```
((every-pred even?) 2 4 6) ; true
((some-fn even?) 2 4 5)
```
- 增加 `realized?` 函数，判断是否已经由一个 `lazy`、`delay` 的变量被真正生成了
 

```
(def r (range 10)) ; 仅定义，lazy，尚未真正生成
(realized? r) ; false
(count r) ; 随使用一下，在此过程中被真正生成了
(realized? r) ; true
```
- 增加 `with-redefs-fn` 和 `with-redefs` 临时改变函数定义
 

```
(println nil?) ; #<core$nil_QMARK_ clojure.core$nil_QMARK_@7b283052>
(with-redefs [nil? "hello"] (println nil?)) ; "hello"
```
- 增加 `find-keyword` 看函数名和 `namespace` 是否已经被用了
 

```
(find-keyword map)
```
- `clojure-contrib.jar` 已经被分成小块 `module`，不再打包发布，需要的各自下载

## 15、对照表

### 15.1 和 clisp、scheme 的区别

Lisp	Clojure
<code>(defn m1 (args) args)</code>	<code>(defn m1 [args] args)</code> []使函数参数看上去一目了然
<code>[1, 2, 3, 4]</code>	<code>[1 2 3 4]</code> 减少,
<code>(cond ((&lt; x 10) "less")       ((&gt; x 10) "more"))</code>	<code>(cond (&lt; x 10) "less"       (&gt; x 10) "more")</code> 减少不必要的()
<code>car cdr</code>	<code>first rest</code> 函数名更人性化

clisp scheme clojure 语法对比 (<http://hyperpolyglot.org/lisp>)

Clojure (2007)	clisp (1984)	Scheme (1975)
<code>(def v 10)</code>	<code>(defvar v) (setf v 10) (setq v 10) (set 'v 10)</code>	<code>(define v 10)</code>
<code>(defn f [x y] (+ x y))</code>	<code>(defun f (x y) (+ x y))</code>	<code>(define (f x y) (+ x y))</code>
<code>(vector 2 4 6)</code>	<code>(vector 2 4 6)</code>	<code>(vector 2 4 6)</code>

[2 4 6]	#(2 4 6)	#(2 4 6)
(fn [x y] (+ x y)) #(+ %1 %2) (#(+ %1 %2) 3 4)	(lambda (x y) (+ x y)) ( <b>funcall</b> (lambda (x y) (+ x y)) 3 4)	(lambda (x y) (+ x y))
first rest	first car rest cdr	car cdr
true false	T NIL	#t #f
(map (fn [x] (* x x)) '(2 3))	(mapcar (lambda (x) (* x x)) '(2 3))	(map (lambda (x) (* x x)) '(2 3))
(reduce + 0 '(1 2 3))	(reduce <b>+</b> '(1 2 3) :initial-value 0)	(reduce + 0 '(1 2 3))
(let [x 3 y 4] (+ x y)) (let [x 3 y (inc x)] (* x y))	(let ((x 3) (y 4)) (+ x y)) (let* ((x 3) (y (incf x))) (* x y))	(let ((x 3) (y 4)) (+ x y))
(str "hello" "world")	(concatenate 'string "hello" "world")	
(nth "hello" 4)	(aref "hello" 4)	
(printf "hello")	(write-line "hello")	
(type "hello")	(type-of "hello")	
<b>Clojure (2007)</b>	<b>clisp (1984)</b>	<b>Scheme (1975)</b>
(cond (int? i) 0 :else -1)	(cond ((int? i) 0) (:else -1))	
number? integer? float?	numberp integerp floatp	
Math/pow Math/log	expt log	
(Integer/parseInt "107") (Double/parseDouble "3.14")	(parse-integer "10") (read-from-string "3.14")	
(quot 10 3) ; 3 (rem 10 3) ; 余数	(truncate 10 3) ; 整除 (rem 10 3) ; 余数	
(rand 10.0) (int (rand 10))	(random 10.0) (random 10)	
\e (.charAt "hello" 1) (.indexOf "hello" "e") (.substring "hello" 1) (count "hello") (.equals "hello" "heLlo") (.compareTo "ab" "ac") (.toLowerCase "Hello") (.trim " foo ")	#\e (char "hello" 1) (search "e" "hello") (subseq "hello" 1) (length "hello") (string= "hello" "heLlo") (string> "ab" "ac") (string-downcase "Hello") (string-trim '#\Space	

<pre>(str "hello " 10)  (seq      (.split      "a b" "[ \t]+"))  (format "%s %d %.1f" "a" 7 3.14)</pre>	<pre>#\Tab #\NewLine) " foo ") (concatenate 'string "hello " (princ-to-string 10)) (regex:regex-split "[ \t]+" "a b") (format nil "~a ~a ~,1F" "a" 7 3.14)</pre>	
Clojure (2007)	clisp (1984)	Scheme (1975)
(concat [1 2] [3 4])	(append '(1 2) '(3 4))	
(nth '(10 20 30) 1)	(nth 1 '(10 20 30))	
(position 3 '(5 3 7))	(wr3.clj.s/position 3 '(5 3 7))	
(last '(1 3 5))	(first (last '(1 3 5)))	
(def a '(1 3 5)) (cons 10 (rest a)) 或者 (assoc [1 3 5] 0 10)	(setq a '(1 3 5)) (setf (first a) 10); (10 3 5)	
(sort '(2 3 1) '<)	(sort < '(2 3 1))	
(find '{a 10 b 20} 'a)	(assoc 'a '((a 3) (b 5)))	
('{a 10 b 20 c 30} 'a)	(getf '(a 10 b 20 c 30) 'a)	
(def  ^{:macro true} lambda #'fn)	(defmacro fn (&rest args) ` (lambda ,@args))	
(map #(* % %) '(1 2 3))	(mapcar (lambda (x) (* x x)) '(1 2 3))	
(filter #(> % 2) '(1 2 3))	(remove-if-not (lambda (x) (> x 2)) '(1 2 3))	
(reduce - 0 '(1 2 3 4))	(reduce '- '(1 2 3 4) :initial-value 0)	
(doseq [x '(1 2 3)] (println x))	(dolist (x '(1 2 3)) (print x))	
(nth [1 2 3] 0)	(aref #(1 2 3) 0)	
(replace {2 20} [1 2 3])	(setq v #(1 2 3)) (setf (aref v 1) 20)	
(seq [1 2 3]) (vec '(1 2 3))	(coerce #(1 2 3) 'list) (coerce '(1 2 3) 'vector)	
(seq? '(1 2 3))	(typep '(1 2 3) 'list)	
(load-file "a.clj")	(load "a.lisp")	

## 15.2 和 newlisp 的区别

newLISP for little jobs and Clojure for the heavy stuff  
newlisp 也是一个现代的 lisp 方言，特别适合用于快速启动的脚本以及调用 C 函数库，和 Clojure 优势互补。

Clojure	newlisp
(def v 10)	(setq v 10) (define v 10)
(defn f [x y] (+ x y)) (defn f [] nil)	(define (f x y) (+ x y)) (define (f) nil)
(vector 2 4 6) [2 4 6]	
(fn [x y] (+ x y)) #(+ %1 %2) (#(+ %1 %2) 3 4)	(fn (x y) (+ x y))  ((fn (x y) (+ x y)) 3 4)
(range 5)	(sequence 0 4)
	(import "msvcr71.dll" "printf") (printf "hello\n")
(slice '(a b c d e f) 2 -1) (slice '(a b c d e f) 2 4)	(2 '(a b c d e f)) ; (c d e f) (2 3 '(a b c d e f)) ; (c d e)
	(copy coll) ; 一个新的 clone
(doseq [[i e] (map-indexed (fn [i e] [i e]) '(a b c d e f))] (printf "%s: %s\n" i e))	(dolist (x '(a b c d e f)) (println <u>\$idx</u> ": " x))

### 15.2.1 newlisp 常用模式

功能	代码示例
得到命令行参数	newlisp -m 10 (main-args) ; ("newlisp" "-m" "10")
执行脚本	注意：在最后必须写(exit)，如 (println (main-args)) (exit)
不装载 init.lsp	newlisp -n test1.lsp
处理每行	\$newlisp -n t1.lsp < t1.lsp ;----- t1.lsp (while (read-line) (println (current-line))) (exit)





----- The End  
-----