本文改编自我在公司内部分享的《Clojure 简介与应用》的讲稿。

1. OOP 的本质？

面向对象编程（OOP）中最终要的是什么？抽象、封装、集成、多态？实现模式？设计模式？还有更重要的么？

下面引用两段业内名言：

“如果我们现在回头看一下面向对象这个思想是从哪来的，如果以基于消息传递机制的 Smalltalk-80 的特性来衡量现在的状态继承和面向对象的使用方式，我们不禁要问，我们是不是已经走错路了？” ——2010 伦敦 QCon 大会采访

只关注状态，在类和基于映像的语言里缺乏良好的并发模型和消息机制。 —— Dave Thomas 博士

到底什么被忽视了？从这两段话中我们可以看出：是 OOP 的并发模型和消息机制被现代 OO 编程语言忽视了，尤其是 Java。

在当代 OO 语言中，可变状态让并发编程变得非常复杂，只能依靠悲观锁来进行并发的控制。

至于消息传递机制，大都 OO 语言本身并没有提供有效的机制，而是运用设计模式来达到目的的，但这又会使编程的过程复杂化，也会在一定程度上影响代码的可读性。

至今，业界已经承认 OOP 并不是万能的。而 OOP 的真正优势在于对现实世界的建模，而不是数据处理。我们应该辩证的看待不同范式的编程语言，死磕一个必然会使思想禁锢，甚至编程灵感尽失。

2. FP 是什么？

现在我们来看看在函数式编程（FP）中是怎样解决这些问题的。

2.1 函数式编程概览

- 一种编程范式
- 程序运算即为数学上的函数计算
- 以 λ 演算（lambda calculus）为基础
- 函数为 first-class，可以很方便的运用闭包创造出高阶函数
- 避免状态、变量和副作用
- 支持懒惰计算（lazy evaluation）和引用透明性

2.2 函数式编程详解

2.2.1 不可变数据

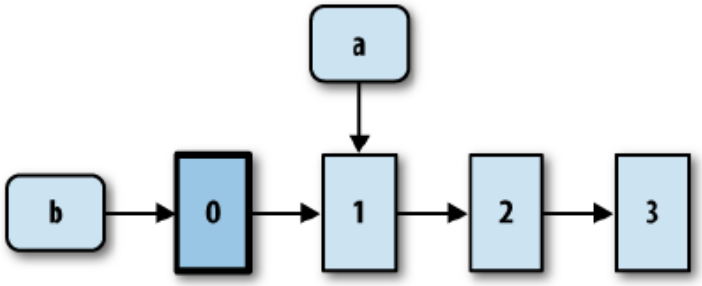
```
;; Immutable data
(def a '(1 2 3))
(def b (cons 0 a))
;; b -> '(1 2 3)
(println "The 'a' is " a)
;; b -> 0 append a
(println "The 'b' is " b)
```

运行结果：

The ’a’ is (1 2 3)

The ’b’ is (0 1 2 3)

引用 a 和 b 在定义时被赋值，并且在之后的任何时刻都不可能被改变。并且，在底层，a 和 b 的值会重用一部分数据存储的。如下图：



从图中可以看出，a 和 b 引用的其实是同一个序列，只是起点不同而已。

Clojure 使用的这种技术叫做 PDS（Persistent Data Structures）。Clojure 的数据结构是不可变的，也是持久的。持久数据结构的好处包括：

1) 可以大幅提高程序效率。
2) 为并发编程提供有力支持。
3) 更容易进行数据版本控制。

另外，与不可变数据相关的另一个函数式编程概念是——引用透明。引用透明意味着相同的输入一定会返回相同的输出，即：一个函数的计算过程不会因任何外部环境的改变而发生变化。相信我们真正理解不可变数据之后，引用透明这个概念也会非常好理解的。

2.2.2 一级类型——函数

“把函数作为语言的一级类型”的意思是说，语言本身支持把一个函数作为另一个函数的输入和输出。

首先，我们来看一个把函数作为另一个函数的输入的例子：

```
;; the function as a 'first-class' 1
(defn my-func1
  "A demo of first-class"
  [d f]
  (f d))
(my-func1 "It's first-class!" println)
```

运行结果：
It’s first-class!

上面的代码做了这些事：

1. 定义了一个名字叫“my-func1”的函数。
2. 为这个函数写了一段内容为“A demo of first-class”的注释。
3. 声明了这个函数的两个形参：d 和 f。
4. 调用 f，而 d 作为参数传给 f。因此形参 f 必须代表一个需要传入一个参数的函数，这样才能被正确调用。
5. 我们调用了函数“my-func1”，并将内容为“It’s first-class”的字符串绑定到了形参 d 上、将 println 这个函数绑定到了形参 f 上。
6. 根据“my-func1”函数中的定义，它本质上执行了这段代码：

```
(println "It's first-class!")  ;; 还记得“my-func1”函数体中的(f d)么？
```

初读 Clojure 的代码需要注意几点：

1. Clojure 是基于 JVM 的 Lisp 方言，所以会有很多的括号（但是与 Lisp 相比已经简化了很多），这一点需要习惯。
2. 读 Lisp 家族的代码需要从最里面的括号开始读，一直读到最外面的括号。这是一种嵌套结构。
3. Clojure 中的一对括号（即，“(”和“)”）叫一个 form，每一个 from 中可以有一个或多个元素。并且，form 中的第一个元素应该是一个函数的标识名（Symbol），后面的元素应该是传给这个函数的参数。所有的 Clojure 代码都会遵守 form 的这种格式。
4. 每一个 form 都是一个表达式，每一个表达式的返回值都是对这个表达式的求值结果。
5. 函数体的返回值不需要显示标明，而是在函数定义中最后一个 form 的求值结果。

现在，我们来看看怎样把函数作为另一个函数的输出：

```
;; the function as a 'first-class' 2
```

```
(defn func-a [s] (str "Func A: " s))
(defn func-b [s] (str "Func B: " s))
(defn my-func2
  "Another demo of first-class"
  [n]
  (cond
    (> n 0) func-a
    :else func-b))
(println ((my-func2 0) "my-first-class"))
```

运行结果：
Func B: my-first-class

上面的代码做了这些事：

1. 定义了一个名字叫“func-a”的函数,这个函数有一个形参 s，执行该函数之后会获得一个返回值，返回值的内容是‘"Func A: " s’。
2. 定义了一个名字叫“func-b”的函数,这个函数有一个形参 s，执行该函数之后会获得一个返回值，返回值的内容是‘"Func B: " s’。
3. 定义了一个名字叫“my-func2”的函数。这个函数有一个形参 n。函数体是一个 cond 的函数调用（cond 函数相当于 Java 中的 switch 语句）。这个函数调用表示：当 n 大于 0 时，返回函数“func-a”，否则返回“func-b”。
4. 之前已经提到过，一个函数的返回值即是其函数体中最后一个 form 的求值结果。在函数“my-func2”中，这最后一个 form 就是那个 cond 调用。也就是说，函数“my-func2”会根据 n 的值来返回函数“func-a”或“func-b”。
5. 在上面代码的最后一行，我们首先调用了函数“my-func2”，并传入了实参 0。根据上面的代码说明我们可以知道这次函数调用的返回值——函数“func-b”。接着我们调用了这个被返回的函数，并得到了结果。

上面这两段代码可以充分展现出了函数式编程的强大威力。函数可以当做代码块或算法单元传入其他函数或者被其他函数返回。这一特性极大的增强了代码的灵活性和扩展性。

2.2.3 懒惰计算

懒惰计算意味着对表达式的求值是按需进行的。当真正需要表达式的值（或值中的某部分）时，求值（或部分求值）的操作才会被执行。这种计算方式的意义在于最小化某一个时刻的计算量，从而达到最大化节省空间和时间的目的。

下面我们来看一个例子：

```
;; lazy and infinite sequences
;; (iterate inc 1) ;; Don't do that!!
```

```
(println (take 10 (iterate inc 1)))
```

运行结果：

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

上面的代码做了这些事：

1. 首先看一下第二行代码，这是一行注释。iterate 函数会返回一个无限迭代的序列。我们调用这个函数，并传入了两个实参：inc 和 1。inc 也是一个函数，在这里传入 inc 意味着返回的序列的每一个元素（整数）都是前一个元素加 1 的结果。第二个参数 1 表示返回序列的第一个元素为 1。**注意！iterate 函数返回的序列是无限迭代的。直接调用 iterate 会使程序一直计算这个无穷序列的下一个元素，直到内存溢出。**
2. 最后一行代码我们将调用 iterate 函数的 form 作为第二个参数传入了 take 函数中。这个 take 函数调用的第一个实参为 10。这意味着我们只想获取这个无穷序列的前 10 个元素。对这个调用 take 函数的 form 求值并不会造成内存溢出，因为我们只需要前 10 个元素。程序会很快计算完成并返回结果。这个无穷序列的其余元素并不会被计算出来，直到真正有程序需要它们的时候。

上面的例子虽然很简单，但是却展示出了懒惰计算的强大威力。懒惰计算可以大大提高程序的性能，并且使我们能够非常方便的按需取用数据。

2.2.4 闭包

闭包其实是建立在将函数作为一级类型的这个特性之上的。闭包使我们根据需要可以动态的生成函数。我们可以先定义一个不完整的函数，也就是说函数体中的算法是有缺失的。而后，在其他代码中将缺失的部分算法传入，生成这个函数的一个完整版本并返回。这其中用到了前文提到的将函数作为另一个函数的输入和输出的特性。

下面我们来看一个例子：

```
;; closure
(defn double-op
  [f]
  (fn [& args]
    (* 2 (apply f args))))
(def double-add (double-op +))
(println (double-add 1 2 3))
```

运行结果：

```
12
```

上面的代码做了这些事：

1. 我们定义了一个名为“double-op”的函数。这个函数用一个形参 f。这个形参 f 应该

是一个函数，因为我们的函数体是一个用 fn（fn 是一个宏，可以理解为宏也是一种能够动态生成函数的方式，且功能上强大很多）定义的匿名函数。这个匿名函数可以接受一或者多个参数（形参名字 args 前的“&”表明了这一点）。这个匿名函数会通过传入的实参（也就是 f 的值）而完整化，并作为函数“double-op”的返回值。

2. 函数 apply 会将第一个实参（一般为一个函数）作用于其余的实参之上，也就是说调用第一个实参代表的函数，并将其余的实参作为其参数传入。使用 apply 的好处在于不必立刻在代码中填入传入“其余的实参”，而可以用引用名代替。这时，这些“其余的实参”可以被叫做预参数。
3. 倒数第二行代码定义了一个名为“double-add”的引用，这个引用返回一个函数。这个返回的函数是通过向函数“double-op”传入函数“+”而完整化后得出的。换句话说，我们在这里定义了一个名为“double-add”的函数。
4. 之后我们调用了函数“double-add”，并得到了预期的结果（把所有“其余的参数”相加并乘以 2）。

闭包是函数式编程中非常重要的特性，并且在一些非函数式语言中也有闭包的身影。另外，还有两个与闭包有关联的两个函数式编程概念：偏函数和柯里化。大家有兴趣的话可以去 google 一下。

2.3 函数式编程（Clojure）的优势

2.3.1 处理数据？用管道的方式会更加简洁

```
;; Focus on results, not steps.
(println (reduce + (map #(* 2 %) (filter odd? (range 1 20)))))
```

运行结果：
200

我们从内向外读代码（从嵌套在最里面的括号开始读），可以清除的明白这段代码做了这些事：

1. 获取一个 1 到 19 的整数序列。（调用 range 函数后结果是“(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)”）
2. 将这个序列的中的奇数提取出来形成另外一个序列。（调用 filter 函数后结果是“(1 3 5 7 9 11 13 15 17 19)”）
3. 将这个奇数序列中的每个元素乘以 2。（调用 map 函数后结果是“(2 6 10 14 18 22 26 30 34 38)”）（其中，“#(* 2 %)”是用简化方式定义的一个匿名函数，也可以用 fn 来定义）
4. 将序列中的所有元素相加。（调用 reduce 函数后结果是“200”）

这种管道流的代码变现方式使得我们读起来非常顺畅。我们几乎在读代码的同时就能明确代码的含义。这种管道流代码也非常只管，数据从内层开始经过中间函数的逐一处理，到了最外层时就生成了我们最终想要的结果。

想象一下，如果用 Java 写的话需要多少行代码？需要多少次循环？需要声明多少个中间变量？

2.3.2 请描述一下我们要做的事情

```
;; Focus more on what the code does rather than how it does it.
(println (for [n (range 1 101) :when (= 0 (rem n 3))] n))
```

运行结果：

```
(3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99)
```

现在我们想找出 1 到 100 中能被 3 整除的数，组成序列并打印，看看上面这段代码是怎么做的：

1. 用一个 for 宏就能搞定！for 宏在 Clojure 里并不是用来做循环和迭代，而是用来对序列进行过滤等复杂操作的。
2. 调用 range 函数返回一个 1 到 100 的序列。
3. 通过“:when”这个关键字和后跟的函数来对上述序列进行过滤。
4. 在 Clojure 中，用方括号包裹的代码块一般定时来声明形参的，并且如果形参名称的下一个形参位置上是一个函数或引用名的话，那么就把它的值赋给这个形参。在这段代码中我们将过滤后的序列赋给了形参 n。
5. 我们调用 for 宏的最后一个表达式为 n，这就意味这调用后的返回值为 n 所代表的那个序列。
6. for 宏很强大，详情请看 Clojure 的文档。

Clojure 里有很多的内建宏和函数。它们为语言使用者提供了很大的便利。它们使得我们在编程时可以更多的关注我们要做什么，而不是怎么去做。换句话说，这我们可以更多的去关注我们想要实现的功能和业务，而不是纠缠在那些不重要的处理细节上。

2.3.3 要亲自管理可变状态？敬而远之吧

```
;; Allow the runtime to manage state.
(def counter
  (let [tick (atom 0)]
    #(swap! tick inc)))
(println (take 10 (repeatedly counter)))
```

运行结果：

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

这段代码做了这些事：

1. 定义了函数“counter”。
2. 在函数体中调用了 let 函数。

3. 声明了一个支持原子操作的变量 0，并将这个变量赋给了引用 tick。tick 作为这 let 函数的内部绑定。
4. 在 let 函数调用的主体中声明了一个匿名函数，这个匿名函数利用 swap 函数来改变 tick 多代表的值。
5. 无限调用函数“counter”并生成一个无穷序列（通过 repeatedly 函数），然后只取序列的前 10 个元素（注意，这里用到了懒惰计算）并返回。

在 Clojure 中，用方括号包裹的代码块一般是来声明形参的。我们可以把用方括号包裹的语句块看成一个 vector（实际上，在 Clojure 中数据结构 vector 的表示法就是用方括号包裹一到多个元素）。如果形参名称的下一个元素时一个 form 或者引用名或者字面量值，那么这下一个元素就是给这个形参的赋值。比如：

```
;; sum of x and y
(let [x 1 y 2] (+ x y))
```

在上面这段代码中，我们调用了 let 函数，将 1 赋值给形参 x、将 2 赋值给了形参 2，并以 x 和 y 的和作为这次 let 函数调用的返回值。

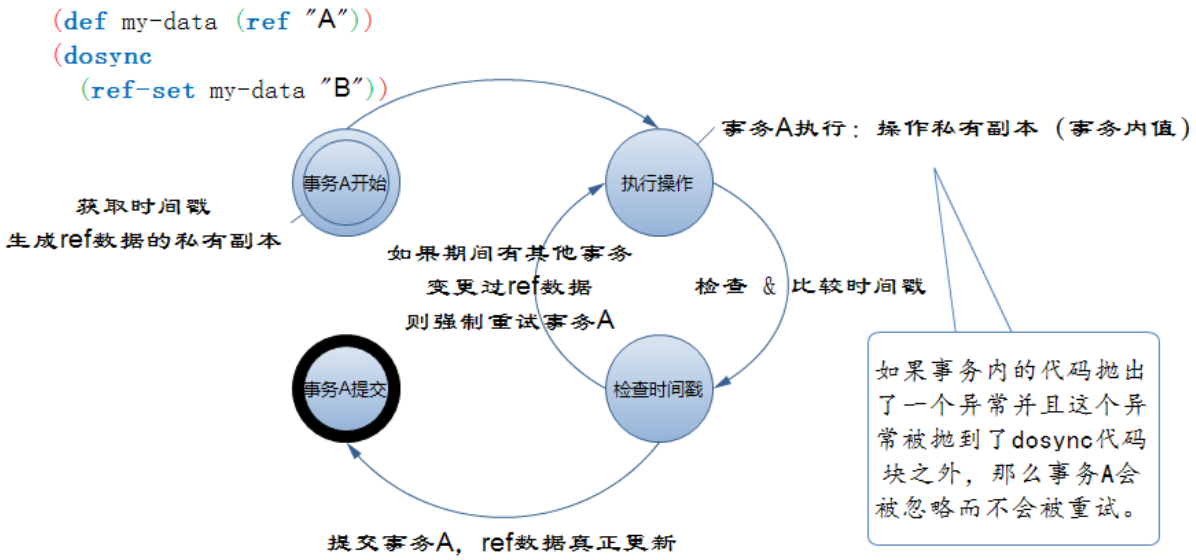
从本节开始的代码可知，在 Clojure 中是可以有可变状态的。但是，对可变状态的管理和状态是完全的由语言来控制的。

Clojure 使用 STM（Software Transactional Memory）技术来对可变状态及其并发操作进行控制。

Clojure 的 STM 使用了一种叫做多版本并发控制（MVCC）的技术。 这一技术也在被一些主流数据库使用。

STM 的事务与数据库的事务类似，保证三点：更新是原子的（A）、更新是一致的（C）和更新是隔离的（I）。 数据库的事务还可以保证更新是牢固的。因为 Clojure 的事务是内存事务，所以并不能保证更新的牢固性（D）。

下面我们用一张状态图来描绘 STM 的并发控制行为：



图中左上角的代码做了这些事：

1. 用 ref 定义了一个支持并发的引用（这个引用指向了一个不可变的值“A”）。这个 ref 引用是可变的。然后，我们将这个 ref 引用赋给引用 my-data。
2. 用 dosync 宏定义了一个事务。在这个事务中，我们使用 ref-set 函数将 my-data 所代表的 ref 引用的值由“A”改为了“B”。
3. 这个事务的执行过程是协作和并发的。“可协作的”意味着我们可以在 dosync 宏中执行一到多个并发操作。dosync 宏保证这些操作永远是按顺序执行的。至于并发控制方面已在图中说明了。

Clojure 的 STM 有四种操作模式，如下表：

名称	协作的/独立的	同步/异步	说明
Ref	协作的	同步	Ref 会为一个不可变的对象创建一个可变的引用。
Atomic	独立的	同步	Atom 是一种比 ref 更轻量级的机制。多个 ref 更新操作能够在事务被协调的执行，而 atom 允许非协调的单一值的更新操作。
Agent	独立的	异步	send 函数被调用后会立即返回，更新操作会稍后在另一个线程被执行。
Vars	线程本地的	同步	Var 是用 defn 或 def 定义，并用 ^:dynamic 修饰的。它可以用 binding 在本地线程将某个引用重新绑定为其他值。

注：

1) 协作/独立：状态是否与其他状态共同作用。
2) 同步/异步：状态的更新是同步还是异步 。

下面是 Ref、Atom 和 Agent 的更新模型：

Update Mechanism	Ref Function	Atom Function	Agent Function
Function application	alter	swap!	send-off
Function (commutative)	commute	N/A	N/A
Function (nonblocking)	N/A	N/A	send
Simple setter	ref-set	reset!	N/A

这个更新模型中描绘出了 Clojure 对并发更新的操作方式。更新机制包含了普通应用的、可交换的、非阻塞的和简易的。

关于 Clojure 的 STM 方面的知识已经超出了本文的主题范围，故在此就不再赘述了。对此感兴趣的读者可以参看 Clojure 的官方文档和书籍。

2.3.4 更自然的使用“组合”来解耦代码

下面的这段代码来自于 O’Reilly 出版的《Clojure Programming》一书的第 2 章的一个例子。这个例子很好的展示了函数式编程在组合/累加功能方面的独特优势。

```
 (defn print-logger
  [writer]
  #(binding [*out* writer]
     (println %)))
((print-logger *out*) "hello")

(require 'clojure.java.io)
(defn file-logger
  [file]
  #(with-open [f (clojure.java.io/writer file :append true)]
     ((print-logger f) %)))
((file-logger "messages.log") "hello, log file.")

(defn multi-logger
  [& logger-fns]
  #(doseq [f logger-fns]
     (f %)))
((multi-logger
   (print-logger *out*)
   (file-logger "messages.log")) "hello again")
```

```
(defn timestamped-logger
  [logger]
  #(logger (format "[%1$tY-%1$tm-%1$te %1$tH:%1$tM:%1$tS] %2$s"
(java.util.Date.) %)))
((timestamped-logger
   (multi-logger
     (print-logger *out*)
     (file-logger "messages.log"))) "Hello, timestamped logger~")
```

这段代码做了这些事：

1. 首先，例子定义了 print-logger 函数。这个函数的函数体是一个匿名函数。这个匿名函数通过调用 binding 宏将标准输出（用“*out*”表示）重新绑定为形参 writer 所代表的值（也就是说重新定义了用 println 函数打印内容的输出目的地），而后打印出调用这个匿名函数时所传入的实参。最后，我们将这个匿名函数作为 print-logger 函数的返回值。这个注意，这里用到了 闭包，通过形参 writer 所代表的值的传入，我们完整化了这个匿名函数，使它真正可以工作。
2. 例子中定义的第二个函数是 file-logger 函数。这个函数有一个形参 file，它的值应该是一个文件的路径。这个函数的函数体也是一个匿名函数。在这个匿名函数中，通过调用 with-open 宏打开了这个文件路径多代表的文件，并创建了一个相应的 Writer 实例并赋值给了内部绑定 f。:append 关键字是用来指定写入方式是否为追加的。在最后，通过调用之前写好的 print-logger 函数并传入 f。这就意味着我们把标准输出与一个指定文件的 Writer 绑定了。这就意味着，我们调用 file-logger 函数并传入文件的路径，就会得到一个可以把内容打印到指定文件的 log 记录函数了。这里同样是一个闭包应用。我们通过将“messages.log”作为参数传给函数 file-logger，完整化了 file-logger 函数体中多定义的匿名函数。待 file-logger 函数将这个匿名函数作为返回值返回之后我们就可以直接使用了。
3. multi-logger 函数可以把 print-logger 函数和 file-logger 函数的功能合并起来。参数 vector 中的“& logger-fns”表明我们可以传入多个函数。函数中的匿名函数会作为返回值返回。这个匿名函数会依次调用之前传入多个函数，并将调用这个匿名函数时传入的参数传递给这几个函数。我们调用 multi-logger 函数并将前面定义好的两个 log 记录函数传入，就可以得到一个可以同时将内容打印到屏幕和文件的多向日志记录函数了。我们得到的这个函数同样是通过闭包方式生成的。
4. 在理解了前面几个函数后，timestamped-logger 函数就很好解释了。我们首先将时间戳字符串和要打印的内容拼接（通过调用 format 函数）并作为参数传给了形参 logger 代表的函数。

这个例子稍显复杂一些，但是它是像搭积木一样一步步将功能堆叠起来的，读起来是非常直观的。当然这要在你理解了相关函数式编程概念之后。

2.4 一起来 FP 吧

上面说了这么多，只希望能够激发起你对 FP 的兴趣。如果你已经对 FP 有了一丝兴趣，那就说明我的文章没白写。如果你会 Java，那我强烈建议你看看 Clojure 这个函数式语言。怎么？你对 Clojure 一无所知？好吧，我在后面补上一些 Clojure 编程语言的基本信息。

3. 这就是 Clojure ['klәuʒә]

3.1 Clojure 是什么？

- 一种 Lisp 方言（最初只基于 JVM 构建，现在也有 CLR 和 JS 的版本）
- 开源语言（使用 Eclipse Public License v 1.0 协议）
- 动态类型语言（标识类型是可选操作）
- 函数式语言（但提供了安全的可变状态操作方法）
- 作者： Rich Hickey
- 2007 年 10 月第一次发布
- 官方网站： http://www.clojure.org/

3.2 Clojure 的亮点

- Clojure 里的每个操作都被实现成以下三种形式中的一种：special form, function, macro.
- Clojure 仅提供了很少的数据结构（但操作它们的方法众多）：regular expressions, list, maps, sets, vectors, metadata.
- 序列（sequence）——集合的统一逻辑视图
- 数据默认不可修改，但提供了保证并发安全的修改方式
- 大量使用了懒惰计算，大大提高程序效率
- 核心数据结构可以扩展（Common Lisp 和 Scheme 的核心数据结构可修改，但不可扩展）
- 所有数据结构是不可修改的、持久的并且支持递归的（在传统 Lisp 里，只有 list 是结构可递归的）
- 使用 PDS（Persistent Data Structures）技术解决了不可变数据造成的内存空间浪费和数据创建低效率问题
- STM（Software Transactional Memory）机制使得她内置的支持了并发编程

3.3 Clojure 的开发环境

- 构建工具——Leiningen，兼容 Maven 仓库。
- 轻量级 IDE——Clooj，集成了项目浏览器、支持语法高亮 Clojure 源码文件查看器、

输出查看器和 REPL。

- 更高级的 IDE——推荐 IDEA + La Clojure 插件。
- 手边的 Clojure 书籍。
- Clojure 文档站点。

3.4 Clojure 相关网站

- 官网：http://clojure.org
- 文档站点：http://clojuredocs.org
- 题库站点：http://www.4clojure.com
- Clojure 构件仓库： https://clojars.org
- 中文用户组：http://cnlojure.org

3.5 Clojure 相关书籍

- Programming Clojure, Second Edition （易入门，基于 Clojure 1.3）
- Clojure Programming（O’Reilly 出品，基于 Clojure 1.3）
- Clojure in Action（实践手册）
- The Joy of Clojure（比较深入）
- Clojure - Functional Programming for the JVM（易入门，有中文版）

3.6 Clojure 的一些应用场景

- Text Search：Clucy，Snowball Stemmer
- Asynchronous HTTP：Aleph
- HTTP Clients：clj-http，http.async.client
- GUI：Clarity，Seesaw
- Web Server：Ring
- Web Frameworks：Compojure，Conjure
- Databases：FleetDB，Jiraph，clj-record
- Redis Clients：clj-redis，redis-clojure
- **Twitter Storm——开源实时 Hadoop**