Tips for Reading Coffee Code

Function

JavaScript

```
var square = function(x) {
  return x * x;
};
```

CoffeeScript

```
square = (x) -> x * x
```

-> 表示函数声明

执行路径中的最后一个表达式的返回值就是函数返回值

Lexical Scoping and Variable Safety

JavaScript

```
var changeNumbers, inner, outer;

outer = 1;

changeNumbers = function() {
  var inner;
  inner = -1;
  return outer = 10;
};

inner = changeNumbers();
```

CoffeeScript

```
outer = 1
changeNumbers = ->
  inner = -1
  outer = 10
inner = changeNumbers()
```

缩进表示 scope，而不是 {}

无需 var，当前 scope 中出现的新的变量将自动在 scope 中被声明

对象和数组

JavaScript

```
var bitlist, kids;

bitlist = [1, 0, 1, 0, 0, 1, 1, 1, 0];

kids = {
  brother: {
    name: "Max",
    age: 11
  },
  sister: {
    name: "Ida",
    age: 9
  }
};
```

CoffeeScript

```
bitlist = [
  1, 0, 1
  0, 0, 1
  1, 1, 0
]

kids =
  brother:
    name: "Max"
    age:  11
  sister:
    name: "Ida"
    age:  9
```

ES6 包含的 Coffee 特性

- Arrow Function
- Object Literal Property Value Shorthand
- Spread Operator
- Rest Parameter
- Default Argument
- Destructuring Assignment
- Multiple return values

Functional Reactive Programming

Rong Shen

FRP = Functional +
Reactive

请保持耐心，集中精力

养成 FRP 的思维方式需要时间

Bacon.JS

https://github.com/baconjs/bacon.js

Stream & Property

- Stream: 时间线一系列分离的事件，例如: mouse click
- Property: 相当于时间的函数 f(t)，任何时间都可以获得确定的值。有些 FRP 框架中称为 Behavior，例如: mouse position
- 它们可以相互转换:
  - stream.toProperty(initValue)
  - prop.changes()

Streamed Events

```
----a---b----c---d---X---|->
```

1. a, b, c, d are emitted values
2. X is an error
3. | is the ‘completed’ signal
4. ---> is the timeline

Subscribe

```
$('.this').asEventStream('click').onValue ->
    $('h2').text 'Clicked'
```

- Bacon 扩展了 jQuery (jQuery.fn.prototype)
- onValue: 订阅发生的事件
- onError: 订阅发生的错误
- onEnd: stream 结束时触发

产生 stream 的方式很多

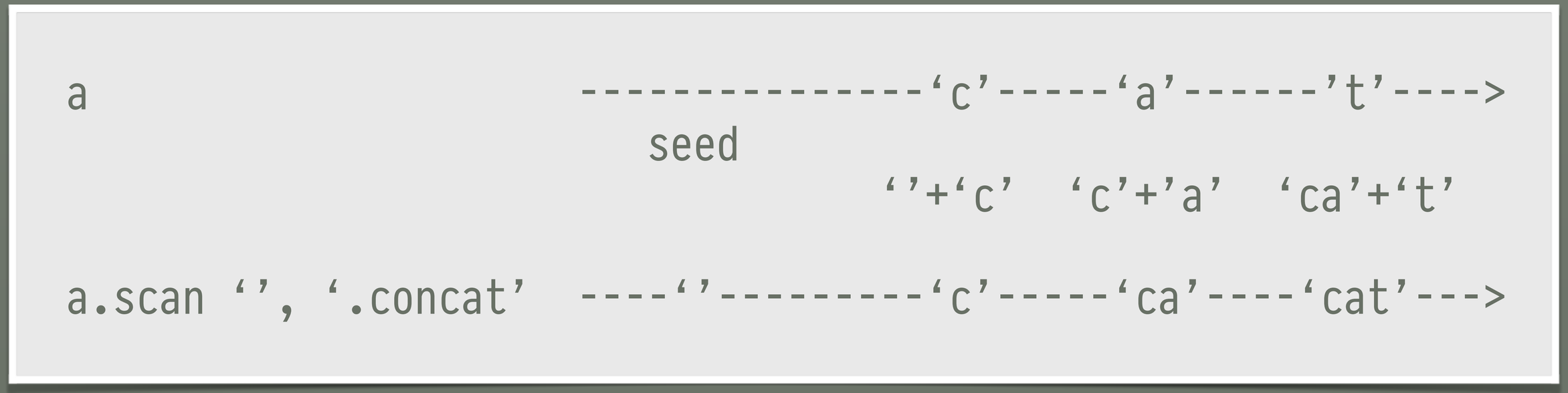
- Bacon.fromArray [1,2,3,4]
- Bacon.constant 'bacon': 一个输出常量的
- Bacon.fromCallback (cb) -> setTimeout -> cb('Bacon!'), 1000
  - .fromNodeCallback, .fromPromise
- Bacon.fromEventTarget document.body, 'click' (也支持 Node.JS stream)
- Bacon.sequentially 1000, 'c', 'a', 't': cat
- Bacon.repeatedly 1000, 'c', 'a', 't': catcatcat...
- ...

Map

```
a                          ----1----2----3---->

a.map (x) -> x % 2         ----1----0----1---->
```

- 将 source streams 的 events 一一映射为其他 events

Scan



- result stream 的第一个值就是scan制定的初始值。随后就是处理函数就会收到最近的汇总值和新的 source event，由处理函数生成新的汇总值

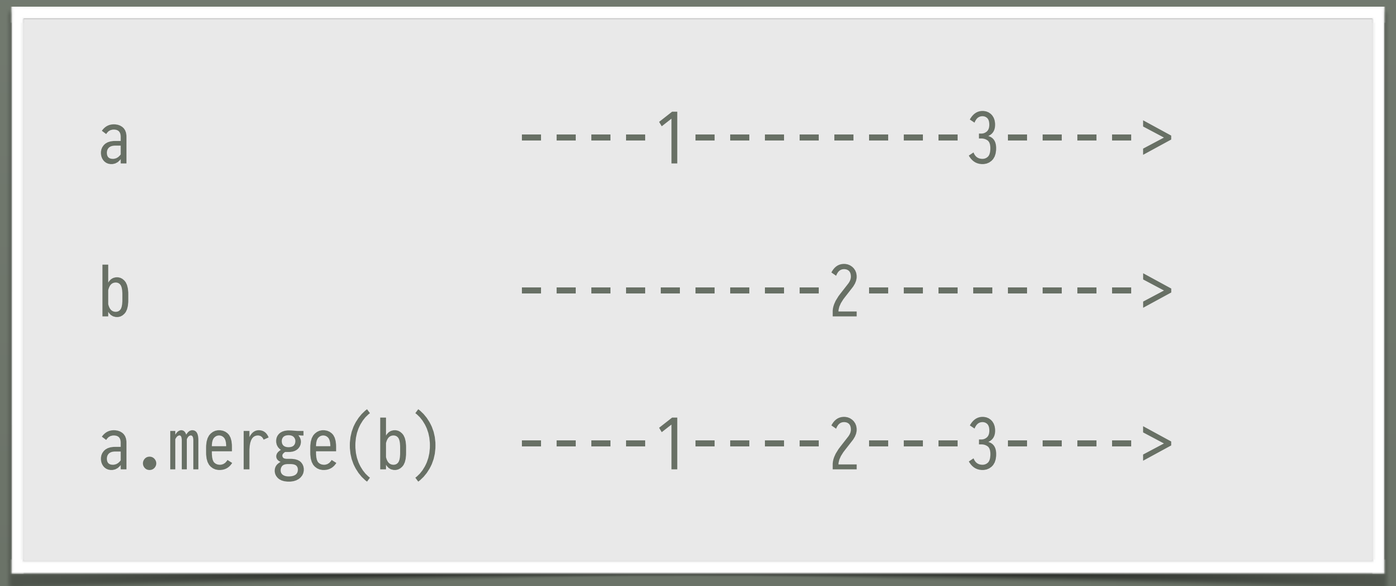
Demo(scan.coffee)

```coffee
chalk = require 'chalk'
Bacon = require 'baconjs'

Bacon.fromArray ['c', 'a', 't']
    .scan '', (seed, curr) -> seed + curr
    .onValue console.log.bind console, chalk.red 'Joined string:'

Bacon.fromArray [1, 2, 3]
    .scan 0, (seed, curr) -> seed + curr
    .onValue console.log.bind console, chalk.green 'Summed numbers:'

Bacon.fromArray [1, 2, 3]
    .reduce 0, (seed, curr) -> seed + curr
    .onValue console.log.bind console, chalk.blue 'Reduced:'
```

```
Joined string:
Joined string: c
Joined string: ca
Joined string: cat
Summed numbers: 0
Summed numbers: 1
Summed numbers: 3
Summed numbers: 6
Reduced: 6
```

- Bacon.fromArray: 从数组生成一个 stream
- .reduce: 当所有 source stream 结束时才输出结果。.fold 是 .reduce 作用完全一样的别名

Merge



- 将 a,b 两个 streams 的 events 按照时间先后合并到一个新的 stream

Demo (plusMinus)

```
plus = $('#plus').asEventStream('click').map -> 1
minus = $('#minus').asEventStream('click').map -> -1

total = Bacon.mergeAll(plus, minus).scan 0, (a, b) -> a + b
total.assign $('#total'), 'text'
```

- BaconJS 会在 jQuery 的 fn.prototype 上加 Helper，便于直接将 DOM 元素的事件转化为 stream
- Bacon.mergeAll 和 plus.merge minus 效果是一样的
- .assign 相当于 .onValue (value) -> $('#total').text value

Filter

```
onlyEven = (x) -> x % 2 is 0


a                       ----1----2----3----4--->

a.filter onlyEven       ---------2---------4--->
```

- 仅有满足过滤条件的 events 才会出现在目标 stream 中

bufferWithTimeOrCount

```
a                                      ----1-------2--3--4------>

                                       |   100ms   |   100ms   |

.bufferWithTimeOrCount(100, 2)   ----[1]----------[2,3]---->
```

- 设定一个时间窗口和最大缓存数量。哪一个限制先到达，就把缓存中的 events 一次性发出。如果缓存为空就什么都不发。
- .bufferWithTime, .bufferWithCount

Demo(doubleClick)

```
clicked = $('.this').asEventStream 'click'

clicked.onValue ->
    $('h3').text "You clicked"

multiClicked = clicked.bufferWithTimeOrCount(300, 5).map (list) ->
    list.length
.filter (count) ->
    count >= 2

multiClicked.onValue (count) ->
    $('h2').text "#{count} clicked"
```

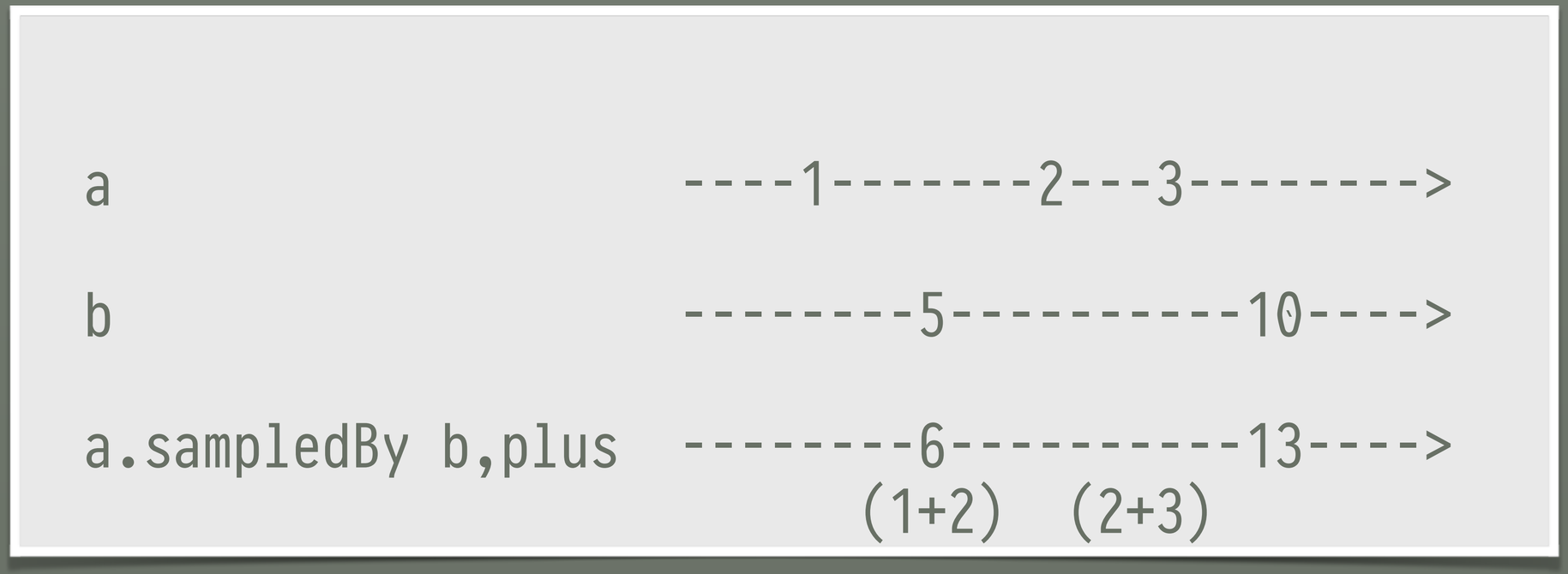
- bufferWithTimeOrCount: 缓存 m 毫秒之内的 n 个 events，然后将缓存的 events 一次 emit (数组形式)

Combine

```
plus = (a,b) -> a+b

a                    ----1-----------3---->

b                    ---------2----------->

a.combine(b,plus)    ---------3------5---->
                                (1+2)  (2+3)
```

- 当a或b有事件发生时，通过组合函数产生新的事件值。第一次必须等a,b都有events发生时才会发生

sampledBy




```
a                       ----1-------2---3-------->

b                       --------5----------10---->

a.sampledBy b,plus      --------6----------13---->
                                (1+2)  (2+3)
```

- a 必须是一个 property，根据 b 中事件的发生频率对 a 进行采样，交给传入的函数进行处理

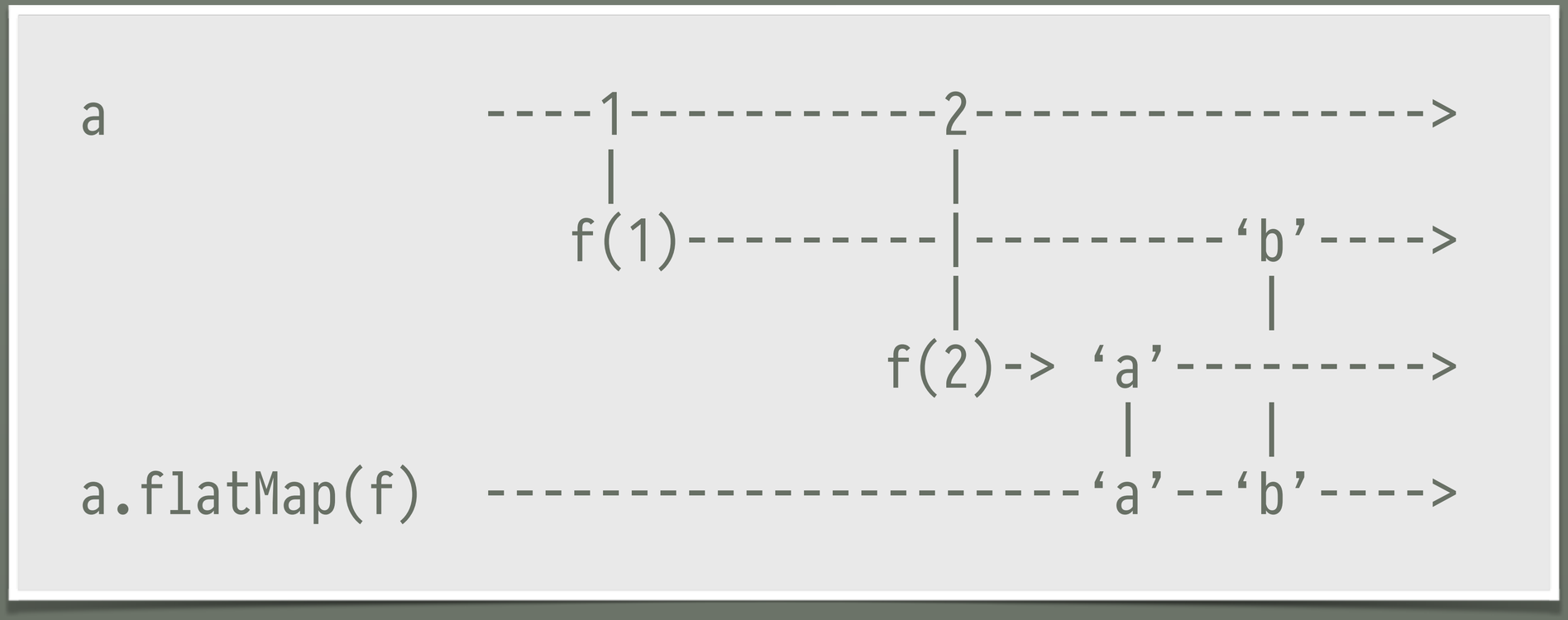
Demo(sampledBy.coffee)

```coffee
Bacon = require 'baconjs'

tick = Bacon.interval 1000, 'tick'
sampled = Bacon.interval 100, 1
    .scan 0, (seed, curr) -> seed + curr

sampled.sampledBy(tick).log 'Sampled counter:'
```

```
Sampled counter: 9
Sampled counter: 19
Sampled counter: 29
Sampled counter: 39
Sampled counter: 48
^C
```

- Bacon.interval: 根据设定的时间，周期性的发出预设的 event

flatMap



- 对 source stream 的每一个 event，创建一个独立的新的 stream，然后将各个 stream 的返回的 event 组合成结果 stream
- 逻辑为：陆续不断地约妹子，约到的每个妹子生的孩子都收着
- .flatMapWithConcurrencyLimit(limit, f)

flatMap Example

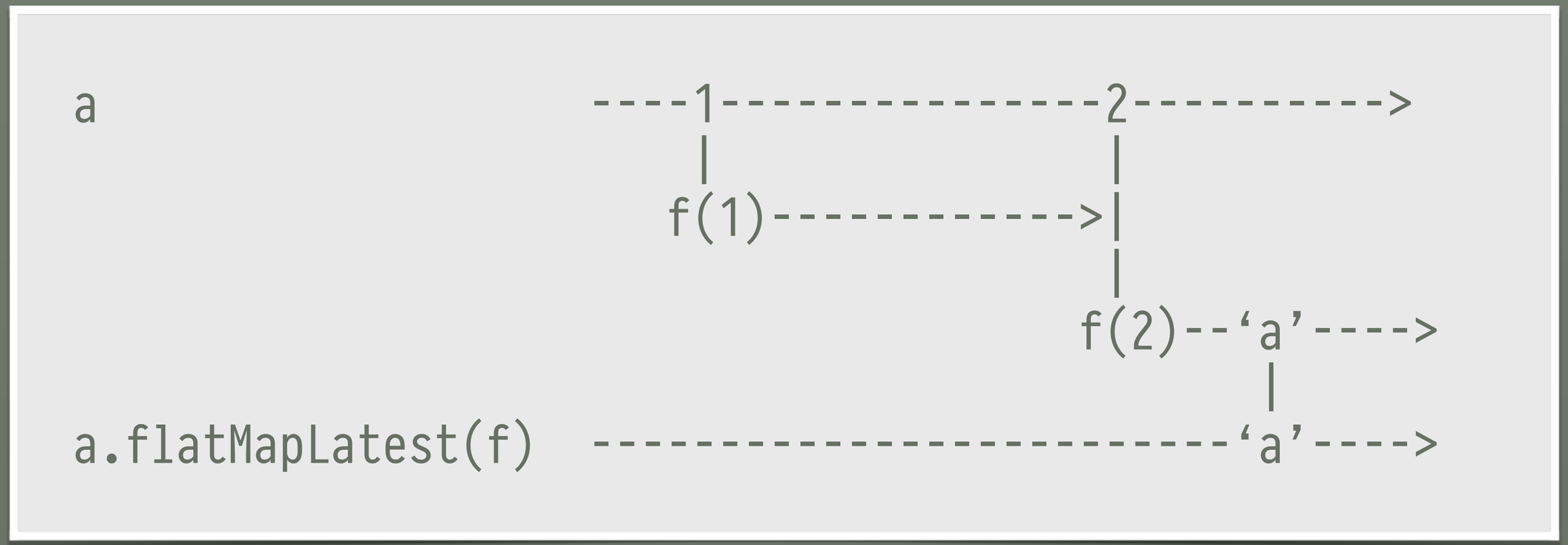
```
asyncServer = (name, cb) ->
    setTimeout ->
        if Math.random() > 0.5
            cb name + ' errors'
        else
            cb null, name
    , Math.random() * 100

batchCall = Bacon.fromArray [1..10].map (n) -> 'server' + n
    .flatMap (server) ->
        Bacon.fromNodeCallback asyncServer.bind null, server

results = batchCall.reduce [], (seed, value) ->
        seed.concat value
results.onValue logResult

errors = batchCall.errors()
    .mapError( (err) -> err )
    .reduce([],  '.concat')
errors.onValue logError
```

Demo

```
shenrong@localhost ~/Dropbox/github/FRP/baconjs coffee flatMap.coffee
Aggregated Results: [ 'server10',
  'server7',
  'server5',
  'server8',
  'server2',
  'server6' ]
Errors: [ 'server1 errors',
  'server4 errors',
  'server9 errors',
  'server3 errors' ]
shenrong@localhost ~/Dropbox/github/FRP/baconjs
```

- Error 不会(不一定)要终止 stream
- Promise, callback, 相当于是一个 event 的 stream
- generator 可以直接对应到 stream

flatMapLatest



- 类似 flatMap，为每一个 source event 生成 stream，但是先建立的 stream 在 emit event 之前，如果有新的 stream 生成，那么之前的 stream 就将被丢弃。
- 逻辑为：约到一号妹子，如果在约到二号妹子之前一号生娃了，那么留下这个孩子；一号未生而二号约到了，就等二号生，而踢掉一号，无论是否怀孕；以此类推

flatMapLatest Example

```
asyncServer = (name, cb) ->
    setTimeout ->
        cb null, name
    , Math.random() * 100 // 1

counter.flatMapLatest (c) ->
    Bacon.fromNodeCallback asyncServer.bind null, c
.onValue emitLog
```

.flatMapFirst: 仅当之前产生的 stream 产生了结果，才会根据之后的 source event 产生新的 stream，之前可能会跳过一些 source events

slidingWindow

```
a                      ------1------2------3------4------->

a.slidingWindow 2      ------[]-[1]-[1,2]--[2,3]--[3,4]--->
```

- a.slidingWindow max, min: 设定滑动窗口的最大和最小容量。如果 {max, min} = {2,2}，那么上例的结果 stream 是 [1,2], [2,3], [3,4]

Bacon.combineTemplate(template)

```
# properties or streams
# password, username, firstname, lastname

loginInfo = Bacon.combineTemplate {
    magicNumber: 3                  # constant
    userid: username
    passwd: password
    name: { first: firstname, last: lastname }
}
```

- loginInfo 是一个新的 property，当组成它的任何一个 stream 的值变化，loginInfo 都会发出新值

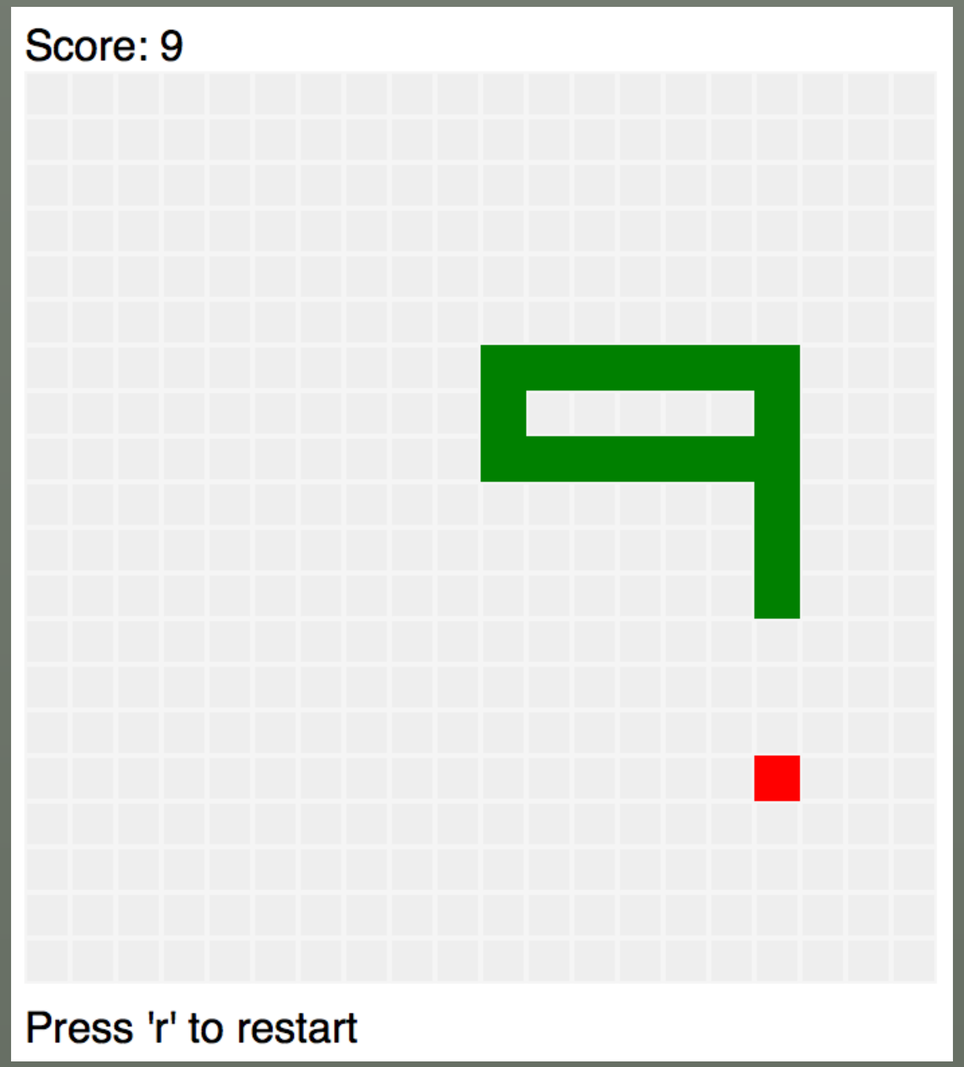
loginInfo

```
{
    magicNumber: 3
    userid: "juha"
    passwd: "easy"
    name : { first: "juha", last: "paananen" }
}
```

Real Game: Snake

http://philipnilsson.github.io/badness/

Demo



换做你，怎么下手?

基本输入

```
bindInputs = ->
    keys = $(document).asEventStream('keydown').map '.keyCode'
    lefts   = keys.filter (x) -> return x is 37
    rights  = keys.filter (x) -> return x is 39
    tick    = Bacon.interval 100
    { left: lefts, right: rights, tick: tick }
```

- tick 是 100ms 周期的时钟发生器

方向定义和变换

```
rotateRight = (vec) ->
    new Vector2 -vec.y, vec.x

rotateLeft = (vec) ->
    new Vector2 vec.y, -vec.x
```

- Vector2 表示方向:
  - (0,1) 向 Y 轴正方向运动(下)，(0,-1) 向 Y轴负方向(上)
  - (1,0) 向 X 轴正方向(右)，(-1，0) 向 X轴负方向(左)

如何获得最新的方向?

```
actions =
    input.left.map(-> rotateLeft).merge(
        input.right.map -> rotateRight
    )

startDirection = new Vector2 0, 1
direction = actions.scan startDirection, (seed, f) ->
    newSeed = f(seed)
```

- WoW, 方向计算函数也可以是 stream event
- 最后的方向就是之前所有方向量计算函数的计算的最终结果

那么最后的位置就是

```
startPosition = new Pos 0, 0

latestPosition = direction
    .sampledBy input.tick
    .scan startPosition, (seedPos,currVec) ->
        newPos = seedPos.add currVec
```

- 那么最后的位置就是初始位置加上一系列的“转向”
- 当然 100ms转一次

消化一下...

还有蛇身...

```
snakeHead = latestPosition
snake  = snakeHead.slidingWindowBy length
```

- 如果蛇头是一个位置流，蛇身就是包括蛇头和前 n 个时刻的蛇头位置
- slidingWindowBy 之前我们介绍过(固定 window 大小)，我们需要构造一个变长的 slidingWindow

slidingWindowBy

```
Bacon.Observable.prototype.slidingWindowBy = (lengthObs) ->
    self = @
    new Bacon.EventStream (sink) ->
        buf = []
        length = 0
        lengthObs.onValue (n) -> length = n
        self.onValue (x) ->
            buf.unshift x
            buf = buf[0...length]
            sink new Bacon.Next buf
        ->
```

- buf[0...length] 相当于 buff.slice(0, length)

Apple

```
apple = (snakeHead) ->
    applePos = randomPos()
    snakeHead
        .filter (p) -> p.equals applePos
        .take 1
        .flatMapLatest apple.bind null, snakeHead
        .toProperty applePos

snakeLength = apple.map(1).scan 10, (x, y) -> x + y
```

- Apple 是一个 stream，每当和蛇头碰撞，就产生一个新的位置
- 每次 Apple 产生，就让蛇身加长

take 系列

- .takeWhile(f): 当 f 返回 false 时停止 stream
- .takeWhile(property): 当 property 产生的 event 的值为 false 停止
- .take(n): 获取 n 个 events 之后停止
- .takeUntil(stream): 如果 stream 出现了 event，则停止目标 stream 的输出，否则继续输出

Score

```
score  = apple.map(1).scan  0, (x, y) -> x + y
```

- Easy now, huh...

Dead

```
_ = Bacon._

contains = (arr, x) ->
    for a in arr
        return true if a.equals x
    false

snake = snakeHead.slidingWindowBy length
dead = snake.filter (body) ->
    contains _.tail(body), _.head(body)
```

- Bacon._ 提供了一组函数式编程的帮主函数(head, tail, fold, without...)

Game

```
game = (position) ->
    snakeHead = position()
    appl = apple snakeHead

    length = appl.map(1).scan 10, (x, y) -> x + y
    score  = appl.map(1).scan  0, (x, y) -> x + y
    snake  = snakeHead.slidingWindowBy length

    dead   = snake.filter (body) ->
        contains _.tail(body), _.head(body)

    game = Bacon.combineTemplate {
        snake: snake
        apple: appl
        score: score
    }
    game.takeUntil dead
```

FRP 的优点

- 我们一直在写"转换函数"，将 stream 转换来转换去
- 不用枚举和处理复杂的状态机，每一个函数就解决一个局部问题
- 不会更改一个既有的 stream，因此也不会因为修改造成“副作用”
- Laziness - 惰性求值，有了订阅者才计算；没有订阅者则只是装配好了管道
- 重用的是"转换"的方法

FRP 的“缺点”

- 命令式的思维习惯
- Know How - 你需要了解你用的 FRP 框架的实现原理
- 额外的性能和堆栈消耗

Make Run

Make Right

Make Fast

Take Aways

- 静态结构和关系，仍然用 OO 方法
- 复杂的状态，FRP

The End

Demo: Tween

```
raf = Bacon.fromBinder (sink) ->
    window = window ? global

    request =
        window.requestAnimationFrame         ?
        window.webkitRequestAnimationFrame   ?
        window.mozRequestAnimationFrame      ?
        window.oRequestAnimationFrame        ?
        window.msRequestAnimationFrame       ?
        (f) -> window.setTimeout f, 1e3 / 60
```

```
tween 1 -100
tween 1 -96.2
tween 1 -91
tween 1 -87.2
tween 1 -83.2
tween 1 -79.4
tween 1 -76
tween 1 -72.6
tween 1 -68.6
tween 1 -65.19999999999999
tween 1 -61.59999999999999
tween 1 -57.999999999999986
tween 1 -53.999999999999986
```