

# Реактивные примитивы Warp9

@rystsov

2013

Warp9 поддерживает реактивные переменные (Cell) и реактивные списки (List).

## Cell: реактивные переменные

## Cell / ctor

Реактивная переменная в `war9` может содержать какое-либо значение или быть пустой, по умолчанию переменная создается пустой.

---

```
var a = new Cell();
```

---

## Cell / set

Для того чтобы положить в неё какое-либо значение  
используется метод “set”

---

```
var a = new Cell();  
a.set(42);
```

---

## Cell / ctor(...)

Альтернатива - передать начальное значение в конструкторе

---

```
var a = new Cell(42);
```

---

## Cell / unset

В любой момент переменную можно сделать пустой

---

```
var a = new Cell(42);  
a.unset();
```

---

# Cell / unwrap

У переменной можно вызвать метод `unwrap` для того, чтобы получить значение, которое она содержит

---

```
var a = new Cell(42);  
console.info(a.unwrap());  
//> 42
```

---



## Cell / unwrap(...)

Если переменная была пустой - вылетит исключение, но `unwrap` можно передать значение, которое следует вернуть, если переменная пустая

---

```
console.info(new Cell().unwrap(42));  
//> 42
```

---

## Cell / lift

К существующей переменной можно применить функцию и получить реактивную переменную, которая связана с первой этой функцией: при изменении первой - автоматически меняется вторая

---

```
var a = new Cell(); // a is unset
var b = a.lift(function(a) {
    return a+2;
}); // b is unset
a.set(1); // a contains 1, b contains 3
a.set(5); // a contains 1, b contains 7
a.unset(); // a is unset, b is unset
```

---

## Cell / lift

Можно сказать, что для метода lift выполняется следующий закон

---

```
forall (f, x):  
  new Cell(x).lift(f).unwrap() == f(x);
```

---

У переменных полученных лифтингом (а впрочем и любым другим способом, кроме как вызовом конструктора Cell) невозможно вызывать методы `set`, `unset`

# Cell / coalesce

Еще один способ создать переменную - вызвать метод `coalesce` и передать ему значение по умолчанию. Если исходная переменная содержит какое-либо значение, то и новая переменная будет его содержать, если исходная переменная пустая, то новая переменная будет содержать значение по умолчанию.

# Cell / coalesce

---

```
var a = new Cell();  
var b = a.coalesce(42); // b contains 42  
a.set(13); // a contains 13, b contains 13  
a.unset(); // a is unset, b contains 42
```

---

## Cell / isSet

Вызов `isSet` вернет реактивную переменную, которая содержит `true` если исходная не пустая, и `false` в ином случае.

---

```
var a = new Cell();  
var b = a.isSet(); // b has false  
a.set(13); // a has 13, b has true  
a.unset(); // a is unset, b has false
```

---

## Cell / when(...)

Другой способ создать переменную - вызвать метод `when`. Если метод вызван с одним параметром, то параметр рассматривается как фильтр. Если значение фильтра `true`, то переменная будет содержать значение, совпадающее с исходной, иначе переменная будет пустая.



## Cell / when(...)

---

```
var a = new Cell();  
var b = a.when(function(a) {  
    return a>3;  
}); // b is unset  
a.set(42); // a contains 42, b contains 42  
a.set(4);  // a contains 4, b contains 4  
a.set(1);  // a contains 1, b is unset
```

---

## Cell / when

Если в качестве фильтра передать не функцию, то значение исходной переменной будет сравниваться с “фильтром” на равенство.

## Cell / when(..., ...)

Так же методу `when` можно передать два параметра: фильтр и трансформер, в это случае трансформер будет применен к значению исходной переменной, если фильтр вернул `true`.

## Cell / when(..., ...)

---

```
var a = new Cell();  
var b = a.when(  
    function(a) { return a>3; },  
    function(x){ return x+1 }  
); // b is unset  
a.set(42); // a contains 42, b contains 43  
a.set(4); // a contains 4, b contains 5  
a.set(1); // a contains 1, b is unset
```

---

## Cell / when(..., ...)

На самом деле `when` с двумя аргументами практически эквивалентен комбинации `when` и `lift`

---

```
forall(f, t, cell) :  
    cell.when(f, t) == cell.when(f).lift(t);
```

---

Практически, потому что, как и в случае с одним параметром - вместо фильтра и трансформера можно передать константы - “фильтр” будет сравниваться с значением исходной переменной, а “трансформер” вернет себя

## Cell / when(..., ...)

---

```
var a = new Cell();  
var b = a.when(42,13); // b is unset  
a.set(42); // a contains 42, b contains 13  
a.set(4); // a contains 4, b is unset
```

---

## Cell / when(..., ..., ...)

Последняя форма when - три аргумента: фильтр, трансформер и альтернативный трансформер, работает точно так же как и предыдущий, но если фильтр вернул false - применяется альтернативный трансформер

## Cell / when(..., ..., ...)

---

```
var a = new Cell();  
var b = a.when(  
    function(a) { return a>=0; },  
    function(x) { return x+1; },  
    function(x) { return x-1; }  
); // b is unset  
a.set(0); // a contains 0, b contains 1  
a.set(-1); // a contains -1, b contains -2  
a.unset(); // a is unset, b is unset
```

---



## Cell / when(..., ..., ...)

Вместо функций можно передать значения, в этом случае when ведет себя как реактивный тернарный оператор

---

```
var a = new Cell();  
var b = a.when(true, 1, 0); // b is unset  
a.set(true); // a has true, b has 1  
a.set(false); // a has false, b has 0  
a.unset(); // a is unset, b is unset
```

---

## Cell / when(..., ..., ...)

За исключения случая с константами, для when с тремя аргументами выполняется закон

---

```
forall(c, f, t, a) :  
  c.when(f, t, a) == c.lift(function(c) {  
    return f(c) ? t(c) : a(c)  
  }) ;
```

---

# Cell / bind

Теперь рассмотрим последний метод Cell для создания новых переменных - bind. Если бы мне пришлось выбирать из lift, when и bind - я бы выбрал bind, так как остальные легко из него выводятся

# Cell / bind

Bind'у на вход нужно передавать функцию. Эта функция применяется к значению реактивной переменной и возвращает Cell, в свою очередь bind возвращает новую реактивную переменную, которая содержит то же значение, что и переменная, которую вернула функция. Звучит страшно, но надеюсь после примеров станет понятнее, но вначале...

# Cell / bind

...закон

---

```
forall (cell , f) :  
    cell . bind ( f ) . unwrap ()  
    ==  
    f ( cell . unwrap () ) . unwrap ()
```

---

Сложно сказать зачем именно нужен этот монстр, так как применяется он практически везде, но начнем по порядку, я сказал, что с него помощью можно создать lift...

# Cell / bind / lift

---

```
var cell = new Cell();  
cell.lift = function(f) {  
    return this.bind(function(x) {  
        return new Cell(f(x));  
    });  
}
```

---

# Cell / bind / when

Теперь реализуем самую первую форму when

---

```
var cell = new Cell();
cell.when = function(f) {
    return this.bind(function(x) {
        return f(x) ? new Cell(x) : new
            Cell();
    });
};
```

---

# Cell / bind / binary

Это далеко не все, например, с помощью `bind` можно взять любую бинарную функцию от обычных значений и возвращающую обычное значение, и применить её к двум реактивным переменным и получить реактивную переменную



# Cell / bind / binary

---

```
var a = new Cell(1);  
var b = new Cell(2);  
var sum = a.bind(function(a) {  
    return b.bind(function(b) {  
        return new Cell(a+b);  
    });  
});  
// sum has 3  
a.set(2); // a has 2, b has 2, sum has 4  
b.unset(); // a has 2, b & sum are unset
```

---

# Монады

Как только вы скажете “Ага, я все понял!” - поздравляю, вы только что познакомились с монадами

(черт, я же не хотел писать очередной tutorial по монадам)

Переходим к реактивным спискам (List)

# List / ctor

Создаем список

---

```
var list = new List();
```

---

## List / ctor(...)

Списку можно передать начальные значения в конструкторе.

---

```
var list = new List([1,2,3])
```

---

# List / add

Кладем элементы в список

---

```
var list = new List();  
list.add("Warp9");  
list.add("React");
```

---

# List / unwrap

Получить содержимое списка можно через unwrap

---

```
var list = new List();  
list.add("Warp9");  
console.info(list.unwrap());  
// ["Warp9"]
```

---

## List / remove

Метод add возвращает id элемента,  
по которому его можно удалить

---

```
var list = new List();  
var warpId = list.add("Warp9");  
var reactId = list.add("React");  
console.info(list.unwrap());  
//> ["Warp9", "React"]  
list.remove(reactId);  
console.info(list.unwrap());  
//> ["Warp9"]
```

---



## List / add( $\lambda$ )

Часто id элемента должен содержаться внутри элемента, поэтому методу add можно передать функцию, эта функция будет вызвана, аргументом будет id, а в список будет добавлен результат выполнения этой функции

---

```
var list = new List();  
list.add(function(id) {  
    return { id: id, name: "Warp" };  
});
```

---

## List / removeWhich

Кроме `remove`, есть еще один метод - `removeWhich`, он принимает предикат и удаляет из списка все элементы, которые удовлетворяют этому предикату

---

```
var list = new List([1,2,3]);  
list.removeWhich(function(x) {  
    return x < 2;  
}));  
// list contains 2,3
```

---

# List / forEach

Так же в списке есть метод `forEach`, который ведет себя подобно `forEach` массива

---

```
var list = new List([
    "Warp9", "React"
]);
list.forEach(function(x) {
    console.info(x);
});
//> Warp9
//> React
```

---

## List / remove, removeWhich и forEach

Методы `remove`, `removeWhich` и `forEach` не учитывают реактивную природу списка и выполняются один раз в момент вызова (не перевыполняются при добавлении новых элементов)

## List / lift

Так же как и Cell, в List есть метод lift, который создает новый список и связывает его функцией с исходным, получается реактивный map

---

```
var a = new List();
var b = a.lift(function(x){ return x+2; });
// b.unwrap()==[]
var id1 = a.add(1);
// a.unwrap()==[1], b.unwrap()==[3]
var id2 = a.add(2);
// a.unwrap()==[1,2], b.unwrap()==[3,4]
a.remove(id1);
// a.unwrap()==[2], b.unwrap()==[4]
```

---

# List / lifting

У списка полученный через лифтинг невозможно вызвать  
методы `add`, `remove`, `removeWhich`

## List / reduce

Все, что я пока рассказал про списки есть и в Knockout и ReactiveCoffee, но я обещал, что warp9 поддерживает больше, а именно агрегацию. Начнем с метода `reduce` и попробуем просуммировать элементы в списке и получить реактивное значение суммы

## List / reduce / sum

Первый параметр reduce - значение, которое соответствует пустому списку, второе значение - функция, которая занимается сверткой

---

```
var list = new List();  
var sum = list.reduce(0, function(a,b) {  
    return a+b;  
});  
list.add(41); // sum has 41  
list.add(1); // sum has 42
```

---

Как мы видим переменная sum получилась реактивной и изменяется при изменении списка.



# List / reduce / count

После того, как мы посчитали сумму списка, давайте узнаем  
кол-во элементов в нем

## List / reduce / count

---

```
var list = new List();
var count = list.lift(function(x) {
    return 1;
}).reduce(0, function(a,b) {
    return a+b;
});
var id41 = list.add(41); // count has 1
list.add(1); // count has 2
list.remove(id41); // count has 1
```

---

## List / reduce / count / wrap

Это работающий метод, правда требует создания нового списка; существует чуть более эффективный путь

---

```
var count = list.reduce(0, function(a,b) {  
    return a+b;  
}), {  
    wrap: function(x) {  
        return 1;  
    }  
});
```

---

# List / wrap

Функция `wrap` выполняется для каждого элемента списка и уже её результат агрегируется, очевидно, что выполняется закон

---

```
forall(list, f, id, fold):  
    list.reduce(id, fold, {wrap: f})  
    ==  
    list.lift(f).reduce(id, fold)
```

---

# List / reduce / эффективность

Раз мы заговорили об эффективности, какова сложность у функции `reduce`, при добавлении элемента в список размера  $n$ ? В `knockout` она равна  $O(n)$ , в `war9` она  $O(\ln n)$ . Но делать даже  $\ln n$  вычислений на каждое добавление элемента в список, чтобы инкрементировать одно значение (`sum` или `count`) странно, поэтому в `war9` есть другой путь

## List / reduceMonoid

Если посмотреть на реализацию функции `reduce`, видно, что  
вызов `list.reduce(id, fold)` раскрывается в

---

```
list.reduceMonoid({  
  identity: id,  
  add: fold  
});
```

---

Искушенный читатель догадается, что раз есть `Monoid`, то есть  
и `Group`, и будет прав

# List / reduceGroup

Пример с суммой элементов списка:

```
var sum = list.reduce(0, function(a,b) {  
    return a+b;  
});
```

можно переписать так:

```
var sum = list.reduceGroup({  
    identity: function() { return 0 },  
    add: function(a,b) { return a+b; },  
    invert: function(x) { return -x; }  
});
```

И сложность, магическим образом, упадет с  $O(\ln n)$  до  $O(1)$ .  
Аналогично можно посчитать и кол-во элементов в списке

## List / reduce\*

Понять, можно ли использовать `reduce*` и какой вариант использовать достаточно просто. Если агрегат списка не зависит от порядка элементов в списке и по посчитанному агрегату и вставляемому элементу вы можете посчитать значение нового агрегата, то агрегат можно выразить через `reduce*` функции.



## List / reduceGroup

Если зная агрегат и удаляемый элемент вы сможете посчитать значение, которое совпадет с агрегатом нового списка, то агрегат можно выразить через `reduceGroup`

Очевидно, что агрегаты `sum` и `count` попадают под оба определения, а значит мы можем для них использовать `reduceGroup`

# List / reduceMonoid

Может сложится впечатление, что любой коммутативный агрегат попадает под это определение, но на самом деле это не так.

Допустим, мы хотим посчитать логическое & от списка bool. Данная задача попадает под `reduceMonoid`, но не под `reduceGroup`

Зная, что агрегат false и зная, что удаляемый элемент тоже false, мы не обладаем информацией, есть ли в списке еще false, а от этого зависит будет ли агрегат false или true

# List / reduce(&)

Хотя я вру

На самом деле, задачу логического & можно свести к `reduceGroup`. Достаточно каждое `true` рассматривать как пару (1,1), каждое `false` как (0,1), при добавлении элемента - покомпонентно складывать пары, при удалении - удалять.

Если в каждой компоненте одинаковое число - значит значение агрегата `true`, иначе `false`. Единственная проблема - в качестве результата будет пара, а не `boolean`

# List / reduce / unwrap

Оказывается в warp9 это не проблема, так как есть опция  
unwrap

После того как reduce отработает, вызывается unwrap и  
преобразовывает результат в нужный тип..

## List / reduce / unwrap

```
list.reduceGroup({
  identity: function() { return [0,0]; },
  add: function(x,y) {
    return [x[0]+y[0],x[1]+y[1]];
  },
  invert: function(x) {
    return [-x[0],-x[1]];
  }
},{
  wrap: function(x) {
    return x ? [1,1] : [0,1];
  },
  unwrap: function(x) {
    return x[0]==x[1];
  }
});
```

# List / reduce\* / Cell

Особенностью работы `reduce*` является то, что он учитывает не только реактивность списка, но и реактивность переменных...

## List / reduce\* / Cell

---

```
var list = new List();
var it1 = new Cell(0);
var it2 = new Cell(1);
var sum = list.reduce(0, function(a,b) {
    return a + b;
}); // sum has 0
var itId1 = list.add(it1); // sum has 0
var itId2 = list.add(it2); // sum has 1
it1.set(5); // sum has 6
it2.set(2); // sum has 7
it1.unset(); // sum is unset
list.remove(itId1); // sum has 2
```

---

## List / reduce\* / Cell / ignoreUnset

Видно, что если внутри реактивный списка есть пустая переменная, то результат агрегации будет пустым. Это поведение можно изменить: достаточно в `reduce` передать опцию `ignoreUnset` - все пустые элементы будут заменяться на единичный элемент моноида или группы

```
var item1 = new Cell();
var item2 = new Cell(1);
var list = new List([item1, item2]);
var sum = list.reduce(0, function(a,b) {
    return a + b;
}, {
    ignoreUnset: true
}); // sum has 1
item1.set(3); // sum has 4
```



Спасибо

Wow! 65 слайдов, вы это  
выдержали!