Basilisk Documentation

Release 0.3.8-pre

Brad Shuttleworth

June 18, 2014

Contents

Basilisk is a **value** library for Javascript. It allows you to create **immutable** data in a familiar way, and to **derive new versions** from that data.

```
var Person = basilisk.makeStruct(['name', 'age']),
    example = new Person({ name: 'Joe', age: 32 }),
    older = example.with_('age', example.age + 2);
```

Making code which updates deeply nested structures **simple** and **clear** is library's main aim.

```
// we make a quick alias, to reduce clutter in the code.
var b$ = basilisk.query,

    Person = basilisk.makeStruct(['name', 'age', 'addresses']),
    Address = basilisk.makeStruct(['city', 'country']),

    example = new Person({
        name: 'Joe',
        age: 32,

        addresses = basilisk.Vector.from([
            new Address({ city: 'London', country: 'United Kingdom' }),
            new Address({ city: 'Cape Town', country: 'South Africa' })
        ])
    }),
    example2, example3;

// first we create a new object, with a US address added.

example2 = b$.swap(example, ['addresses'], function (current) {
    return current.push(new Address({ city: 'Boston', country: 'USA' }));
});

// and if we have to replace a part of that new address.

example3 = b$.replace(example2, ['addresses', b$.at(2), 'city'], 'New York');
```

CHAPTER 1

User Guide

1.1 Why Values?

We have a double-standard in programming: when we do simple arithmetic or work with strings we do not expect the values we're using to be changed by our work.

```
var a, b, c;

a = 5;
b = 7;
c = a + b;

// we would not expect that a or b were changed by adding them together.

a = 5;
b = a;
a += 10;

// we expect that b is still "5".

a = "Hello, "
b = "Gill"

c = a + b;

// Again, we believe that a and b would not be changed by this.
```

We have long known that having simple values makes understanding software a lot simpler: numbers, booleans, and strings are all commonly **immutable** now: If you pass one to a function you know that it will not be changed. You may well be returned a **different string** to use, but the one you started with is still there for comparison.

In some languages (notably C) strings are **mutable**: anyone can change that. While that makes in-place modification faster, it means that any code which interacts with strings needs to know whether it is allowed to change a particular instance, and it is easy for bugs to arise as a result.

1.1.1 Application data

In Javascript (as with most programming languages) the higher-level objects we use to do most of our programming are **mutable**: arrays, objects, and (if you have them in your language) Maps must be changed in-place to effect change, or they must be "deep-copied" to create a history if comparison is required.

The result of this is that any part of the code-base which can **see** an object can **change** that object: we mix up the idea of "identity" with the idea of "value". It thus becomes much harder to how your program moves from one state to another - a fact which looks great in demo code, but which is terrible for making changes or debugging.

1.1.2 Change and Events

The standard solution to the change problem is to use events: when a property is changed the host object (or the browser) fires an event to which other code can be listening, and which can then perform additional updates as required. That works reasonably well when you have very **flat** objects - a "User" object, say, with 10 properties. Change the `username` and an event is fired.

This gets harder to manage when you have several properties which must be updated at the same time: say `x` and `y` co-ordinates for an item in a scene. A naive approach would see you updating first the `x` position, redrawing and then updating again.

Now: `Object.observe` provides a better solution to this in the mutable case by delivering a stream of changes to handlers. However, there are still some hard to follow cases there.

1.1.3 Dancing together

Even these tools don't work fantastically once you are co-ordinating many objects. Consider a person with a list of addresses:

```
var gill = new Person({
    name: 'Gill Jones',
    addresses: {
        home: { country: 'United Kingdom', city: 'London' },
        holiday: { country: 'Ireland', city: 'Dublin' },
        birth: { country: 'South Africa', city: 'Cape Town' }
    },
    currentAddress: 'home'
});
```

Now: changing something about the 'home' address has an impact on anyone who is trying to list Gill's current address. We would need to

1. Add a listener to `gill` to watch for changes to the currentAddress property.
2. Add a listener to `gill.addresses` in case the `home` value was replaced
3. Add a listener to `gill.addresses['home']` in case `home` is changed in some way.

That's a huge amount of book keeping, just to track potential changes in one place.

The alternative approach (using values) is to

1. Be notified of all changes at the root.
2. Use === to quickly (pointer comparison, so very fast) check if the properties you care about (`.currentAddress`, `.addresses`, `.addresses.get(gill.currentAddress)`) might have changed.

Vitally, you never have to bind to current instances of the child values: you only ever need access to a single variable at the top of the tree.

1.1.4 In short

Using events to observe changes to properties is a very **easy** solution to knowing about changes, but it makes **composite** objects very hard to reason about.

Using values makes working with **composite** objects very simple: more composition doesn't lead to spiralling complexity.

1.2 Getting Started

1.2.1 Structs

The first important idea in Basilisk is the `struct`. A `struct` is an object that has a `with_` method.

```
var joe = new Person({ name: 'Joe' });

// Properties can be accessed directly.
console.log('basic value', joe.name);

// You can derive a new value using 'with_'
console.log('adjusted', joe.with_('name', 'Joe Bloggs').name);

// The original value is - of course - not modified by deriving a new value.
console.log('original value', joe.name);
```

An easy way to make constructors for structs is the `makeStruct` function.

```
var Person = basilisk.makeStruct(['name', 'age']),

    joe = new Person({ name: 'Joe' });
```

You can of course extend the prototype with any additional methods you like

```
Person.prototype.toString = function () {
    return this.name + ' (' + this.age + ')';
}

console.log('I am ' + joe);
```

One additional method is created for you automatically: `.equals()`. We'll come back to that in a little bit.

1.2.2 Collections

In a huge number of cases, you don't need to create your own structures to represent data: lists and maps will do just fine. Basilisk comes with these essential data structures in a persistent [1] and performant form.

```
// Vectors have fast append, and fast random access (including set)
var numbers = basilisk.Vector.from([]),
    numbers = numbers.push(5),
    numbers = numbers.push(6),
    numbers = numbers.push(7);

console.log(numbers.get(1));
numbers = numbers.set(1, 10);
```

[1] a persistent data structure is a data structure that always preserves the previous version of itself when it is modified.

```
// .length is O(1)
console.log(numbers.length);
```

The StringMap class gives a simple, safe map from strings to any value.

```
var students = basilisk.StringMap.from({});

students = students.set('Joe', 'Joe Bloggs');

console.log(students.get('Joe'));
// .get accepts a default (undefined is the default value)
console.log(students.get('Mary', 'not present'));

// Unlike normal javascript objects, StringMaps are safe for any string.
students.set('__proto__', 'Does not break the object');
```

The HashMap class is a more flexible mapping object - see its detailed documentation for more info.

1.2.3 Query

Creating immutable objects in Javascript is actually very easy: just use `Object.freeze`. However, deriving new versions of complex objects in a way which is both fast and easy to read is more of a challenge. Basilisk has a `query` module to make this easier. We often bind this to `b$` to make our code a little clearer.

```
var b$ = basilisk.query,
    example = make_a_complex_object();

console.log('Deep access:', example.deep.prop);

// changing 'prop' to be a new value would involve 'backward' reasoning
// in most environments:

example = example.with_('deep', example.deep.with_('prop', 5));

// with basilisk structs, this is clearer:
console.log(b$.replace(example, ['deep', 'prop'], 5));

// Where you are modifying more properties, or deriving a changed value
// use swap

console.log(b$.swap(example, ['deep', 'position'], function (current) {
    return current
        .with_('x', current.x + 5)
        .with_('y', current.y + 10);
}));

// the second parameter (the path) can include more intelligent matchers

console.log(b$.replace(example, ['deep', b$.at(5)], 'hello'));

// basilisk.query.at() will handle any collection which uses .get and .set
// - so Vector, HashMap, and StringMap at the very least.
```

1.2.4 Equality

Probably the most useful thing about working with value objects is that strict equality (===) means that the objects **and their children** are exactly the same - and the check is incredibly quick.

There are many situations, however, where you want to check if two objects are the **same**: for this, basilisk supports `.equals`.

```
var personA = new Person({ name: 'Joe' }),
    personB = new Person({ name: 'Mary' }).with_('name', 'Joe');

console.log('Not the same: ', personA === personB);

// however, it *is* valuable to know if they are identical:

basilisk.equals(personA, personB); // returns true.
```

1.3 Collections

Basilisk has good implementations of the most important data structures for application work. All collections are **immutable** - methods which would mutate them (in normal code) return new versions of the collections.

Where time complexity is stated, recall that log[32] of 1 billion is just less than 6.

1.3.1 Vector

class **`Vector`**

The Vector class is a random access data structure which can be accessed by numeric key. Push, pop and set are all O(log[32] n), which makes the time complexity very low for practical datasets.

Note that - as for all Basilisk collections - the constructor is private and the `from` static method should be used instead.

static `Vector.`**`from`**([*source : mixed*]) → Vector

Creates a new Vector from the specified source object.

Parameters **source** – An `array`, `Vector`, or object with a `.forEach` method. This will be iterated to fill the vector. Passing `null` will result in an empty Vector.

`Vector.length : number`

The number of elements in the Vector. This is a pre-computed property, so access is O(1)

`Vector.`**`get`**(*index : number*) → any

Retrieve the value at a particular position in the Vector. Note that (unlike Javascript arrays) retrieving a position which is outside the range of the collection is an Error. Time complexity: O(log[32] n)

Parameters **index** – The position in the vector to return. Must be in the range (-length ; length). Negative indexes are interpreted as being from the end of the vector (ie. `.length + index`).

`Vector.`**`push`**(*value : any*) → Vector

Creates a **new Vector** which has the specified value in its last position. The instance on which it is called is not modified. Time complexity: O(log[32] n)

`Vector.`**`set`**(*index : number*, *value : any*) → Vector

Creates a **new Vector** which has the specified position replaced with the specified value. If the value **`'==='`** the current value in that position, will return `this`. Time complexity: O(log[32] n)

Parameters **index** – a number in the range (-length ; length).

`Vector.`**`pop`**`()` → Vector
: Returns a **new Vector** which has the item in the final position removed. Time complexity: O(log[32] n)

`Vector.`**`peek`**`()` → any
: Returns the last element in the Vector.

`Vector.`**`forEach`**`(`*callback: function (item : any, index : number), context:any*`)`
: Iterates over the Vector in order, calling the `callback` for each element in turn. It is perfectly valid to pass a function which takes fewer arguments (ie. `function (item)` instead of `function (item, key)` - this is handled natively by Javascript).

`Vector.`**`equals`**`(`*other : any*`)` → boolean
: Checks whether the two Vectors are **equal**. Each element is checked in turn. If all elements are **equal** (see *equality-protocol*)

  Parameters other – Another object to check for equality. If this is **not** a Vector, this will never return true.

1.3.2 StringMap

class `StringMap`
: A `HashMap` of `strings` to any other object. In Typescript, this class is generic on type `T` of the stored objects.

  Note that - as for all Basilisk collections - the constructor is private and the `from` static method should be used instead.

static `StringMap.`**`from`**`(`[*source : mixed*]`)` → StringMap
: Create a new StringMap from the specified source object.

  If the object is a StringMap, then that object is returned directly.

  Finally, the object is iterated using `for in` and own properties are added to the map.

`StringMap.`**`get`**`(`*key : string*[*, default: any = undefined*]`)` → any
: Retrieve the value stored against the key. If it is not present, then the default will be returned (if none is provided, `undefined` is returned.)

`StringMap.`**`set`**`(`*key : string, value: any*`)` → StringMap
: Returns a new StringMap with the added relation. The original map is **not changed**.

`StringMap.`**`remove`**`(`*key : string*`)` → StringMap
: Returns a new StringMap with the relation removed, if it was ever present. The original map is **not changed**.

`StringMap.`**`has`**`(`*key : string*`)` → boolean
: Returns whether the specified key is set in the map. Note that `undefined` is a perfectly legitimate value, so "set" is not the same as "not undefined".

`StringMap.`**`forEach`**`(`*function (value : any, key : string)*[*, context: any = undefined*]`)` → any
: Iterate over the elements of the map in an undefined order. The function will be called with the value and key for each item in turn. Optionally, you can specify a context which will appear as `this` to the function.

1.3.3 HashMap

class `HashMap`
: A configurable HashMap of values. In Typescript, this class is generic on type `T` of the stored objects, type `K` of keys.

  Note that - as for all Basilisk collections - the constructor is private and the `from` static method should be used instead.

static `HashMap.`**`from`**(*hashFn: function (key : any) -> Number*[, *source : mixed*]) → Vector

Create a new HashMap from the specified source object. The `hashFn` will be called every time the hash of a key needs to be evaluated, and should handle any object you might use as a key. `basilisk.hashCode` is a standard implementation which should handle most important cases.

If the object is a HashMap and its hashFn === the provided function, then it will be returned directly. Otherwise it will be iterated and each key passed through the provided hashFunction.

Finally, the object is iterated using `for in` and own properties are added to the map.

`HashMap.`**`get`**(*key : any*[, *default: any = undefined*]) → any

Retrieve the value stored against the key. If it is not present, then the default will be returned (if none is provided, `undefined` is returned.)

`HashMap.`**`set`**(*key : any*, *value: any*) → HashMap

Returns a new HashMap with the added relation. The original map is **not changed**.

`HashMap.`**`remove`**(*key : any*) → HashMap

Returns a new HashMap with the relation removed, if it was ever present. The original map is **not changed**.

`HashMap.`**`has`**(*key : any*) → boolean

Returns whether the specified key is set in the map. Note that `undefined` is a perfectly legitimate value, so "set" is not the same as "not undefined".

`HashMap.`**`forEach`**(*function (value : any*, *key : any)*[, *context: any = undefined*])

Iterate over the elements of the map in an undefined order. The function will be called with the value and key for each item in turn. Optionally, you can specify a context which will appear as `this` to the function.

`hashCode`(*key:any*) → uint

Generate a hashCode for the provided object. If the object has a `hashCode` method, that will be called and the return returned. For strings, numbers, booleans, null and undefined, a default hash implementation is used.

If none of the above apply a TypeError is thrown.

Hash functions should be fast, deterministic, and well distributed over the integers.

1.4 Query

Creating an unchanging value is a very simple thing to do: just instantiate the things you want. Programming is about **change**, and that's where the most important features of Basilisk are targetted.

The `query` module (which we usually bind to `b$` to keep our code clutter-free) allows you to create new versions of values in a way which is simple to understand.

`basilisk.query.`**`replace`**(*initial : any*, *path : PathSegment*[ ], *changed: any*) → any

Given an `initial` value, this returns a new value which has `changed` substituted at the end of `path`.

For example:

```
var a = new Person({
    'name': 'joe',
    addresses: new basilisk.Vector.from([
        new Address({ country: 'RSA' }),
        new Address({ country: 'United Kingdom' })
    ])
}),
    b;

b = b$.replace(a, ['addresses', b$.at(0), 'country'], 'South Africa');
```

```
a.addresses.get(0).country; // 'RSA';
b.addresses.get(0).country; // 'South Africa';
```

Parameters

- **initial** – a value you wish to derive a new value from.
- **path** – `PathSegments` (or `strings` which will be converted into `prop` path segments.)
- **changed** – the value to substitute at the end of the `path` in the newly created return value.

`basilisk.query.`**`swap`**(*initial : any*, *path : PathSegment*[], *swapFn : function(value) -> any*) → any

Like `replace`, but calls `swapFn` with the current value at the end of the chain, and replaces the end value with the result.

For example:

```
var a = new Person({
    'name': 'joe',
    addresses: new basilisk.Vector.from([
        new Address({ country: 'RSA' })
    ])
}),
    b;

b = b$.swap(a, ['addresses', b$.at(0), 'country'], function (country) {
    // normalise common abbreviations of South Africa
    if (country === 'RSA' || country == 'ZA') {
        return 'South Africa';
    } else {
        return country;
    }
});

a.addresses.get(0).country; // 'RSA';
b.addresses.get(0).country; // 'South Africa';
```

Parameters

- **initial** – a value you wish to derive a new value from.
- **path** – `PathSegments` (or `strings` which will be converted into `prop` path segments.)
- **swapFn** – A function to be called with the value at the end of the path. The return value will be substituted at the end of the path, in the newly created result.

`basilisk.query.`**`value`**(*root : any*, *path: PathSegment*[]) → any

Applies the path to the specified `root` and returns the current value at the end of the chain.

`basilisk.query.`**`path`**(*...pathsegments*) → Path

Create a new Path object from the specified Path Segments. `strings` will be converted into `prop` segments.

Parameters **pathsegments** – `string`'s or `PathSegment`'s which will be stored and can be used to `swap` or `apply`

1.4.1 Path

A Path is an ordered list of Path Segments, which can be applied to many values to produce updated versions.

class Path
: (Interface) A Path which can be applied to many different values.

swap (*initial : any*, *swapFn : function(value) -> any*) → any
: Like the `query.swap` method, but with this path applied.

replace (*initial : any*, *changed : any*) → any
: Like the `query.replace` method, but with this path applied.

1.4.2 PathSegment

The `swap` and `replace` functions are wrappers around Path objects, which are made up of *path segments*. A path object allows you to

- find the next value in a chain.
- replace that value with a new one.

This is where the `Struct` interface becomes very important:

```
var b$ = basilisk.query,

    Person = basilisk.makeStruct(['name', 'age']),
    joe = new Person({ name: 'Joe Bloggs', age: 32 }),

    changed,

    propSegment;

// basilisk.query.prop is a path segment that looks at Struct properties.

propSegment = b$.prop('age');

propSegment.current(joe);   // returns '32'
changed = propSegment.replace(joe, 35);

/**

Changed will now be:

  {
    name: 'Joe Bloggs',
    age: 35
  }
*/
```

The path constructor (called by `swap` or `replace`) will convert any plain string to a prop segment.

The `basilisk.query.at` path segment will work with any collection or object which has both `.get` and `.set` methods. The `.set` method must produce a *new* value with the key replaced. Keys can be any type that the collection understands (and collections should throw an error if they aren't).

For example:

```
var b$ = basilisk.query,

    numbers = basilisk.Vector.from([10, 11, 12, 13]),

    segment;
```

```
segment = b$.at(3);

segment.current(numbers);    // returns 13
segment.replace(numbers, 9); // returns V([10, 11, 12, 9])
```

Any object can be used in a path, as long as is has all the methods on the PathSegment interface.

class `PathSegment`
(Interface) Any object which has all the methods on the PathSegment interface can be used in a Path. PathSegments **must** be immutable - they can be cached and re-used.

`current` (*from : any*) → any
Given an object, descend a step into it as appropriate for the segment.

For example, prop segments simply do `value[key]` where `key` is configured at creation time.

Returns the next value in the path.

`replace` (*from : any*, *changed : any*) → any
Perform the update appropriate for the path segment on the `from` parameters, using `changed` as the property.

1.4.3 Basic Path Segments

Basilisk comes with a small set of generic path segments, which

`basilisk.query.`**`prop`** (*propertyName : string*) → PathSegment
Creates a PathSegment which will descend and replace a single property in a Struct.

`basilisk.query.`**`at`** (*key : any*) → PathSegment
Creates a PathSegment which will apply the `key` to the `.get` and `.set` methods of a collection.

CHAPTER 2

License

Basilisk is licensed under the MIT License by MOO Print Ltd.