Cheatography

Bacon.js Cheat Sheet

by ProLoser via cheatography.com/1600/cs/1103/

Observable Interface

observable.map(f)

maps values using given function, returning a new EventStream. Instead of a function, you can also provide a constant value. Further, you can use a property extractor string like ".keyCode". So, if f is a string starting with a dot, the elements will be mapped to the corresponding field/function in the event value. For instance map(".keyCode") will pluck the keyCode field from the input values. If keyCode was a function, the result stream would contain the values returned by the function. The Function Construction rules below apply here.

stream.map(property)

maps the stream events to the current value of the given property. This is equivalent to *property.sampledBy(stream)*

observable.mapError(f)

maps errors using given function. More specifically, feeds the "error" field of the error event to the function and produces a "Next" event based on the return value. Function Construction rules apply. You can omit the argument to produce a Next event with *undefined* value.

observable.mapEnd(f)

Adds an extra Next event just before End. The value is created by calling the given function when the source stream ends. Instead of a function, a static value can be used. You can omit the argument to produce a Next event with *undefined* value.

Observable Interface (cont)

observable.filter(f)

filters values using given predicate function. Instead of a function, you can use a constant value (true/false) or a property extractor string (like ".isValuable") instead. Just like with *map*, indeed.

observable.filter(property)

filters values based on the value of a property. Event will be included in output iff the property holds *true* at the time of the event.

observable.takeWhile(f)

takes while given predicate function holds true

observable.take(n)

takes at most n elements from the stream. Equals to Bacon.never() if n <= 0.

observable.takeUntil(stream2)

takes elements from source until a Next event appears in the other stream. If other stream ends without value, it is ignored

observable.skip(n)

skips the first n elements from the stream

observable.delay(delay)

delays the stream/property by given amount of milliseconds. Does not delay the initial value of a Property.

observable.throttle(delay)

throttles stream/property by given amount of milliseconds. Events are emitted with the minimum interval of *delay*. The implementation is based on stream.bufferWithTime. Does not affect emitting the initial value of a Property.

Observable Interface (cont)

observable.debounce(delay)

throttles stream/property by given amount of milliseconds, but so that event is only emitted after the given "quiet period". Does not affect emitting the initial value of a Property. The difference of *throttle* and *debounce* is the same as it is in the same methods in jQuery.

observable.debounceImmediate(delay)

passes the first event in the stream through, but after that, only passes events after a given number of milliseconds have passed since previous output.

observable.doAction(f)

returns a stream/property where the function f is executed for each value, before dispatching to subscribers. This is useful for debugging, but also for stuff like calling the preventDefault() method for events. In fact, you can also use a property-extractor string instead of a function, as in ".preventDefault".

observable.not()

returns a stream/property that inverts boolean values

By **ProLoser**
cheatography.com/proloser/

Published 11th June, 2013.
Last updated 13th May, 2016.

Observable Interface (cont)

observable.flatMap(f)

for each element in the source stream, spawn a new stream using the function *f*. Collect events from each of the spawned streams into the result *EventStream*. This is very similar to selectMany in RxJs. Note that instead of a function, you can provide a stream/property too. Also, the return value of function *f* can be either an Observable (stream/property) or a constant value. The result of *flatMap* is always an *EventStream*. *stream.flatMap()* can be used conveniently with *Bacon.once()* and *Bacon.never()* for converting and filtering at the same time, including only some of the results.

observable.flatMapLatest(f)

like flatMap, but instead of including events from all spawned streams, only includes them from the latest spawned stream. You can think this as switching from stream to stream. The old name for this method is *switch*. Note that instead of a function, you can provide a stream/property too.

observable.flatMapFirst(f)

like flatMap, but doesn't spawns a new stream only if the previously spawned stream has ended.

observable.scan(seed, f)

scans stream/property with given seed value and accumulator function, resulting to a Property. For example, you might use zero as seed and a "plus" function as the accumulator to create an "integral" property. Instead of a function, you can also supply a method name such as ".concat", in which case this method is called on the accumulator value and the new stream value is used as argument.

Observable Interface (cont)

observable.fold(seed, f)

is like *scan* but only emits the final value, i.e. the value just before the observable ends. Returns a Property.

observable.reduce(seed,f)

synonym for *fold*.

observable.diff(start, f)

returns a Property that represents the result of a comparison between the previous and current value of the Observable. For the initial value of the Observable, the previous value will be the given start.

observable.zip(other, f)

return an EventStream with elements pair-wise lined up with events from this and the other stream. A zipped stream will publish only when it has a value from each stream and will only produce values up to when any single stream ends. Be careful not to have too much "-drift" between streams. If one stream produces many more values than some other excessive buffering will occur inside the zipped observable.

observable.slidingWindow(max[, min])

returns a Property that represents a "-sliding window" into the history of the values of the Observable. The result Property will have a value that is an array containing the last *n* values of the original observable, where *n* is at most the value of the *max* argument, and at least the value of the *min* argument. If the *min* argument is omitted, there's no lower limit of values.

Observable Interface (cont)

observable.log()

logs each value of the Observable to the console. It optionally takes arguments to pass to console.log() alongside each value. To assist with chaining, it returns the original Observable. Note that as a side-effect, the observable will have a constant listener and will not be garbage-collected. So, use this for debugging only and remove from production code.

observable.combine(property2, f)

combines the latest values of the two streams or properties using a two-arg function. Similarly to *scan*, you can use a method name instead, so you could do *a.combine(b, ".concat")* for two properties with array value. The result is a Property.

observable.withStateMachine(initState, f)

lets you run a state machine on an observable. Give it an initial state object and a state transformation function that processes each incoming event and returns and array containing the next state and an array of output events.

observable.decode(mapping)

decodes input using the given mapping. Is a bit like a switch-case or the decode function in Oracle SQL. For example, the following would map the value 1 into the the string "mike" and the value 2 into the value of the *who* property.

Both EventStream and Property share the Observable interface, and hence share a lot of methods. Common methods are listed below.
https://github.com/raimohanska/bacon.js-#common-methods-in-eventstreams-and--properties

Cheatography

Bacon.js Cheat Sheet

by ProLoser via cheatography.com/1600/cs/1103/

EventStream

Bacon.EventStream

a stream of events

stream.onValue(f)

subscribes a given handler function to event stream. Function will be called for each new value in the stream. This is the simplest way to assign a side-effect to a stream. The difference to the subscribe method is that the actual stream values are received, instead of Event objects. Function Construction rules below apply here.

stream.onValues(f)

like onValue, but splits the value (assuming its an array) as function arguments to f

stream.onEnd(f)

subscribes a callback to stream end. The function will be called when the stream ends.

stream.subscribe(f)

subscribes given handler function to event stream. Function will receive Event objects (see below). The subscribe() call returns a unsubscribe function that you can call to unsubscribe. You can also unsubscribe by returning Bacon.noMore from the handler function as a reply to an Event.

stream.skipDuplicates([isEqual])

drops consecutive equal elements. So, from [1, 2, 2, 1] you'd get [1, 2, 1]. Uses the === operator for equality checking by default. If the isEqual argument is supplied, checks by calling isEqual(oldValue, newValue). For instance, to do a deep comparison,you can use the isEqual function from underscore.js like stream.skipDuplicates(_.isEqual).

EventStream (cont)

stream1.concat(stream2)

concatenates two streams into one stream so that it will deliver events from stream1 until it ends and then deliver events from stream2. This means too that events from stream2, occurring before the end of stream1 will not be included in the result stream.

stream.merge(stream2)

merges two streams into one stream that delivers events from both

stream.bufferWithTime(delay)

buffers stream events with given delay. The buffer is flushed at most once in the given delay. So, if your input contains [1,2,3,4,5,6,7], then you might get two events containing [1,2,3,4] and [5,6,7] respectively, given that the flush occurs between numbers 4 and 5.

stream.bufferWithTime(f)

works with a given "defer-function" instead of a delay. Here's a simple example, which is equivalent to stream.bufferWithTime(10): stream.bufferWithTime(function(f) { setTimeout(f, 10) })

stream.bufferWithCount(count)

buffers stream events with given count. The buffer is flushed when it contains the given number of elements. So, if you buffer a stream of [1, 2, 3, 4, 5] with count 2, you'll get output events with values [1, 2], [3, 4] and [5].

stream.bufferWithTimeOrCount(delay, count)

buffers stream events and flushes when either the buffer contains the given number elements or the given amount of milliseconds has passed since last buffered event.

EventStream (cont)

stream.toProperty()

creates a Property based on the EventStream. Without arguments, you'll get a Property without an initial value. The Property will get its first actual value from the stream, and after that it'll always have a current value.

stream.toProperty(initialValue)

creates a Property based on the EventStream with the given initial value that will be used as the current value until the first value comes from the stream.

stream1.awaiting(stream2)

creates a Property that indicates whether stream1 is awaiting stream2, i.e. has produced a value after the latest value from stream2. This is handy for keeping track whether we are currently awaiting an AJAX response: var showAjaxIndicator = ajaxRequest.awaiting(ajaxResponse)

https://github.com/raimohanska/bacon.js?utm_source=javascriptweekly&utm_medium=email#eventstream

Bus

new Bacon.Bus()

returns a new Bus.

bus.push(x)

pushes the given value to the stream.

bus.end()

ends the stream. Sends an End event to all subscribers. After this call, there'll be no more events to the subscribers. Also, the Bus push and plug methods have no effect.

bus.error(e)

sends an Error with given message to all subscribers

Cheatography

Bacon.js Cheat Sheet

by ProLoser via cheatography.com/1600/cs/1103/

Bus (cont)

bus.plug(stream)

plugs the given stream to the Bus. All events from the given stream will be delivered to the subscribers of the Bus. Returns a function that can be used to unplug the same stream. The plug method practically allows you to merge in other streams after the creation of the Bus. I've found Bus quite useful as an event broadcast mechanism in the Worzone game, for instance.

Bus is an EventStream that allows you to push values into the stream. It also allows pluggin other streams into the Bus. The Bus practically merges all plugged-in streams and the values pushed using the push method.

https://github.com/raimohanska/bacon.js?utm_source=javascriptweekly&utm_medium=email#bus

Property

Bacon.Property

a reactive property. Has the concept of "current value". You can create a Property from an EventStream by using either toProperty or scan method. Note depending on how a Property is created, it may or may not have an initial value.

Bacon.constant(x)

creates a constant property with value x.

property.subscribe(f)

subscribes a handler function to property. If there's a current value, an Initial event will be pushed immediately. Next event will be pushed on updates and an End event in case the source EventStream ends.

Property (cont)

property.onValue(f)

similar to eventStream.onValue, except that also pushes the initial value of the property, in case there is one. See Function Construction rules below for different forms of calling this method.

property.onValues(f)

like onValue, but splits the value (assuming its an array) as function arguments to f

property.onEnd(f)

subscribes a callback to stream end. The function will be called when the source stream of the property ends.

property.assign(obj, method, *[param...]*)

calls the method of the given object with each value of this Property. You can optionally supply arguments which will be used as the first arguments of the method call. For instance, if you want to assign your Property to the "disabled" attribute of a JQuery object, you can do this: myProperty.assign($("#my-button"), "attr", "disabled") A simpler example would be to toggle the visibility of an element based on a Property: myProperty.assign($("#my-button"), "toggle") Note that the assign method is actually just a synonym for onValue and the function construction rules below apply to both.

property.sample(interval)

creates an EventStream by sampling the property value at given interval (in milliseconds)

property.sampledBy(stream)

creates an EventStream by sampling the property value at each event from the given stream. The result EventStream will contain the property value at each event in the source stream.

Property (cont)

property.sampledBy(property)

creates a Property by sampling the property value at each event from the given property. The result Property will contain the property value at each event in the source property.

property.sampledBy(streamOrProperty, f)

samples the property on stream events. The result values will be formed using the given function f(propertyValue, samplerValue). You can use a method name (such as ".concat") instead of a function too.

property.skipDuplicates(*[isEqual]*)

drops consecutive equal elements. So, from [1, 2, 2, 1] you'd get [1, 2, 1]. Uses the === operator for equality checking by default. If the isEqual argument is supplied, checks by calling isEqual(oldValue, newValue). The old name for this method was "distinctUntilChanged".

property.changes()

returns an EventStream of property value changes. Returns exactly the same events as the property itself, except any Initial events. Note that property.changes() does NOT skip duplicate values, use .skipDuplicates() for that.

property.and(other)

combines properties with the && operator.

property.or(other)

combines properties with the || operator.

https://github.com/raimohanska/bacon.js?utm_source=javascriptweekly&utm_medium=email#property

Event Types

Bacon.Event

has subclasses Next, End, Error and Initial

Event Types (cont)

Bacon.Next

next value in an EventStream or a Property. Call isNext() to distinguish a Next event from other events.

Bacon.End

an end-of-stream event of EventStream or Property. Call isEnd() to distinguish an End from other events.

Bacon.Error

an error event. Call isError() to distinguish these events in your subscriber, or use onError to react to error events only. errorEvent.error returns the associated error object (usually string).

Bacon.Initial

the initial (current) value of a Property. Call isInitial() to distinguish from other events. Only sent immediately after subscription to a Property.

https://github.com/raimohanska/bacon.js?utm_source=javascriptweekly&utm_medium=email#event

Event Methods

event.value()

returns the value associated with a Next or Initial event

event.hasValue()

returns true for events of type Initial and Next

event.isNext()

true for Next events

event.isInitial()

true for Initial events

event.isEnd()

true for End events

https://github.com/raimohanska/bacon.js?utm_source=javascriptweekly&utm_medium=email#event