Change Detection

Status: Draft

Authors: misko@google.com

This document is published to the web as part of the public Angular Design Docs folder

Objective

The goals of change-detection are:

- Have the most efficient dirty checking implementation in JavaScript
    - Have an extremely simple check loop. (This is important so that VM vendors can better optimize it, and possibly implement in VM)
    - Looking for changes as well as detecting a change should produce no memory pressure. (Important to not cause GC pauses)
    - Coalesce checks in case of `a.b.c` and `a.b.e`. In this case `a.b` only need to be checked once.
- Provide design pattern for easy drop in of Object.Observe
    - Separate field change detection from function/method change detection
    - Process reaction functions asynchronously from change-detection to be consistent with Object.observe
- Allow bulk watch removes in efficient manner (important when destroying large areas of view)
- Provide insight into performance by making timing and internal counters part of the API.

Background

This work is based on these performance benchmarks:

- field: 20,000 field checks / 1 ms: http://jsperf.com/object-observe-polyfill-sandbox
- array: 300,000 array field checks / 1ms: http://jsperf.com/array-change-detection

These numbers are very encouraging since they demonstrate that even a large app with several thousand bindings and large array iterators can be dirty checked on the order of 1ms.

Prior Art

The current AngularJS performance is slower mainly due to GC pressure which is associated with processing of the changes. The current solution also suffers from interleaving change detection with change processing, making it hard to know how much time is spent in dirty

checking and how much is just to process the result. The new design processes the changes asynchronously which will allow clear measure of field check, array check, function invocation and cost of change processing.

Detailed Design

Layer 1: Object.observe Polyfill

The lower layer implements field, array and map change detection. It is structured this way so that it can be easily replaced with Object.observe (or other change detection algorithm).

Field change detection

The algorithm relies on traversing a linked list of change records which contain information about which object and field to read and what was the previous value of the read. The information is stored in this data structure:

```
var record = {
  obj,       // The object whose field needs to be checked.
  field,     // The field name to dereference.
  lastValue, // The last value of the field.
  next,      // Next record in the chain of records to check.
};
```

Here is the algorithm which does the checking

```
var current = head;
while(current !== null) {
  if (current.obj[current.field] !== current.lastValue) {
    // record change
  }
  current = current.next;
}
```

The key points here is that this loop is small and hence lends itself well to VM optimization.

The most expensive part of this loop is the field read `current.obj[current.field]`. This is known as slow field read in VM, since the VM can not optimize the field offset in the object because the objects share no common type. Instead the VM needs to perform full resolution each time. (All other field reads in the loop are on the same object type. The VM detects that and replaces it with fast field read, which consist of simply adding field offset to object address).

To allow the VM to optimize the field reads with a fast read, we can pre-generate a function like so:

```
record.getter = new Function('o', 'return o.' + record.field);
```

We can then change the slow field read from `current.obj[current.field]` to `current.getter(current.obj)` which triples the performance.

Array and Map Change Detection

The array change detection is essentially the same algorithm, but instead of field dereference we can simply dereference the array. Because no field reads are involved the algorithm is significantly faster than field dereference algorithm.

Map detection is essentially the same as Array except that we iterate over the map keys rather than array items. While map keys are not guaranteed to come in any particular order, the order is stable over multiple iterations as long as the map has not changed. For this reason we can treat the map the same way we treat arrays.

Computing Array Difference

Once change in array is detected we need to compute the difference (additions, moves and removals) This difference can be computed in linear cost to the size of the array without the need for additional traversals of the array. A special attention needs to be paid to duplicates in the array.

```
var maybeDirty = false;
for(int index = 0, length = list.length; index < length; index++) {
  var item = list[index];
  if (record === null || item !== record.item) {
    record = changeLog.mismatch(record, item, index);
    maybeDirty = true;
  } else if (maybeDirty) {
    record = changeLog.verifyReinsertion(record, item, index);
  }
  record = record._nextRec;
}
changeLog.truncate(record);
```

`changeLog` object keeps track of changes to the array. It contains a list of removals, insertions and moves as well as hashmap which can retrieve the `ItemRecord` for an item in the array.

If a mismatch is detected then:

- Place the mismatched record into the removal list
- See if the removal list contains the item. If yes, move it from removals into the moves list and record the positional information

- Otherwise it is a new item. Create a new `ItemRecord` for it and add it to additions list.

NOTE: If a change is detected in an array, the algorithm slows down due to an additional call to `verifyReinsertion`. This is needed since even if the array as well as `ItemRecord` have the same item, we still need to compute that it is the same item due to duplicates, which may result in moves and addition to moves list. For example going from `[a, a]` to `[x, a, a]` would assume that `'a'` at position 1 is same as in the second array. This is not true since the `'a'`s have shifted.

The resulting change log follows following rules.

- Assume that items are reused before new items are added. This implies that the log will never show '`x`' removed and '`x`' added at the same time.
- Duplicate items always keep internal order, and are shifted as a unit. `[a, a]` to `[x, a, a]` is insert '`x`', shift first '`a`', shift second '`a`'; not insert '`x`' and move first '`a`' after second '`a`'.

Layer 2: Pure function, Closure, and Method Checking

The second layer adds function, closure, method invocation and coalescing on top of Layer 1. It is unlikely that such functionality will be implemented by VM which is the reason for the separation.

Method invocation algorithm is essentially same as the field dereference algorithm, except that field read is replaced with function invocation.

In addition the `ItemRecord` contains additional pointers for forwarding of values. Example: an expression such as `fn(a.b, c)` (where '`fn`' is pure function) needs to invoke the function when either '`a`', '`b`', or '`c`' changes. This creates an internal dependency tree which is used for propagating the values to achieve the final result.

NOTE: a side effect of the current setup is that we can not short circuit expression evaluations (ie: `a ? b : c` will always check '`a`', '`b`' and '`c`'.)

Change Tracking and Memory Pressure

Internally the change detection algorithm keeps track of fields to check as `Record` data structure. When reporting changes we need to return a list of `Record`s which have changes. Creating such a list would create memory pressure. While small for field checks, it is quite large for array checks. Arrays need list of items, list of removals, list of additions, and list of moves and a hashmap. A change into array would require moving these records between different lists which would create memory pressure. For this reason the `Record`

contains multiple previous/next pointers so that the record can be part of multiple lists at once.

```
var record = {
  obj, field, lastValue, // Bookkeeping info
  prev, next,    // List of records. Defines order of checks.
  nextChange,    // List of changes
  // For arrays/maps we also have
  nextDuplicate, // List of duplicates
  nextAddition,  // List of additions
  nextRemoval,   // List of removals
  nextMove       // List of moves
};
```

The result is that every change `Record` has some fixed memory cost allocated to it at initialization, but once allocated all of the checking, and change reporting produce no memory allocations. This is important since change detection should not produce GC pressure.

Assuming that a `Record` contains about 20 pointers (many not listed here) times 4 bytes per pointer is 80 bytes. (VMs have additional internal pointers for each object) A reasonable estimate would be that each `Record` will consume about 100 bytes. Therefore a large application with 10,000 watches would consume about 1MB of book-keeping memory.

Caveats

None that I can think of.

Security Considerations

Since this is an internal algorithm, I don't believe there are security considerations.

Performance Considerations / Test Strategy

This work is based on these performance benchmarks:

- field: 20,000 field checks / 1 ms: http://jsperf.com/object-observe-polyfill-sandbox
- array: 300,000 array field checks / 1ms: http://jsperf.com/array-change-detection

Because the tight loop algorithms presented here are simple and short, they are great candidates for optimization by the VM teams or possible inclusion into the VM, which could produce even better performance gains in the future.

Work Breakdown

The above algorithm is already implemented in AngularDart: https://github.com/angular/angular.dart/pull/434.

Still need to:

- [DONE] Write code to hook up the Lexer/Parser to the change-detection
- Wrap in new scope.
- Port to JavaScript