Change-Detection mental model in Ivy

author: misko@
March 2020

Overview

This document suggests a mental-model for change-detection (CD) in Ivy. The goal is to both simplify the mental model and improve the capabilities of the CD going forward.

What problem is this solving

This proposal is a follow-up for transplanted views. The transplanted views proposal solves a particular issue of transplanted views, whereas this proposal is a more generic solution that also solves transplanted views as a side-benefit. The proposal here is a mental model that can handle all of the use cases in a generic way.

The specific issues which this proposal tries to solve:

- The current mental model does not clearly specify how transplanted views should be handled.
- The current mental-model mixes the responsibility of marking a view dirty with knowing that we need to descend into that view. This is why `ChangeDetectorRef.markForCheck()` marks all ancestor views dirty as well. Without marking ancestors' views as dirty, the change detection would not know which views to descend into. This mixes the responsibility of descending with the responsibility of processing the view.
    - A check always views inside of the on-push component is shielded from change detection processing in the current implementation.
- There is no way to easily mark a view for re-processing. Developers run into this often and often use the `setTimeout` or `Promise.resolve` trick to get around it. This causes whole change detection to re-run, which is a lot more expensive than just processing a dirty view.
- This is a stepping stone for having Angular have explicit state management. Having explicit state management will allow angular to make zone.js optional. Having the `markDirty` method will allow future state management systems to communicate with Angular when change detection should run.
- Properly fix transplanted views by:
    - Having a CD always follows the insertion hierarchy.
    - Have `markDirty` always follow the declaration hierarchy.

- There is no easy way to mark other components as dirty. (The component can only ask for `ChangeDetectorRef` for itself, never for another component.)
    - There are also tree-shaking implications.

Developer documentation

> This section shows the proposed documentation for the developer to be added to angular.io. This section is written from the point of view of the application developer.

This document describes how the change detection is processed in Angular. Change detection is the process by which Angular detects changes in the template bindings and then updates the DOM to reflect the changes.

Mental model

Angular application is a tree of components. Each component has a view that is declared by the template of the component. A component's view can have embedded-views. (Embedded-views are usually defined inline in the component View as is the case with `*ngFor` for example. See Transplanted embedded views for non-inline embedded views.)

When change detection runs, it visits all views (including embedded-views) in depth-first order. Each view has a flag that marks if the view is "dirty" and should be processed. Processing a view means that its bindings are checked, and the corresponding DOM view is updated. When change detection runs and the view is not marked as "dirty," it is ignored.

There are two kinds of "dirty" flags:

1. Temporary "dirty" flag: This view will be checked as part of the next change-detection, and the "dirty" flag will be cleared on the processing of the view.
2. Permanent "dirty" flag (also known as a check-always flag): This behaves the same as the temporary-dirty flag above, but the flag is set on each new change detection pass. The result is that such a view will participate in change detection every time the change detection executes.

A view can be marked as dirty explicitly by the developer using `markDirty` API or implicitly by the framework. Angular implicitly marks a view as dirty on:

- Initial view creation => All initial views are created in a dirty state and will participate in the next change detection.
- A component input changes => automatically marks the component as dirty when the component input changes. (see next rule for more details)
- A component is marked dirty => Automatically mark all embedded views which were declared in the component dirty as well. (This is true even if the embedded view was

declared in this component but inserted in a different component. See Transplanted embedded views for more details)
    - This is needed because all embedded views which are associated with the current component share the same evaluation context. Since the component is the evaluation context, marking component dirty also needs to mark all embedded views which use the evaluation context dirty as well.

When a view is marked dirty (either implicitly or explicitly), it is guaranteed that the view will be processed after the `markDirty` API returns (but never as part of the `markDirty` API call). This means that the view will be processed either:

- as part of the current change detection, which is already on-going OR
- new change detection will be scheduled (The exact mechanism for scheduling TBD.)

Components declared with `ChangeDetectionStrategy`:

- `Default`: This will mark all views and embedded views that were declared in the component as permanently-dirty/check-always.
- `OnPush`: This will mark all views and embedded views that are declared in the component without a dirty flag. The implication is that the view will only be processed if they are implicitly or explicitly marked dirty.

NOTE:

- A special consideration exists for views that are marked dirty AND change detection is currently active AND that view has already been visited (views are visited in depth-first order). In such a situation, the change detection is effectively re-wound back to the first dirty view in depth-first order once the change detection processing ascends to the least common ancestor between the current change detection cursor and the first dirty view. (Views with permanent "dirty"/check-always flags will not be re-processed second time unless explicitly marked by `markDirty` API as dirty)
- A view that is marked dirty will be processed regardless of if the ancestor views are dirty or not. (A disconnected view will not participate in change detection and will not be processed)

Transplanted embedded views

There is a special case of transplanted views that requires clarification. A component can have an embedded view that is declared outside of the current component (for example using `*ngFor` with an explicitly passed template that comes from a different component.) This situation is referred to as a transplanted embedded view.

There are no special rules for transplanted embedded views other than those already stated above, but we think it is worth discussing the implications of the above rules with respect to the transplanted embedded views.

Implications:

- Change detection is always processed in depth-first traversal, which always follows insertion order (never declaration order.)
    - It does not matter where the view was declared; we should always reason about where it was inserted, which matters for the purposes of the change detection.
    - A disconnected view will not be reachable, and those will not execute as part of change detection.
- Whether or not a view is marked permanent "dirty"/check-always flags dependent on the component where it was declared (not where it was inserted.)
    - A transplanted embedded view may have different flags than the view in which it was inserted into.
- When a component is marked dirty, it also marks all declared views as dirty.
    - A transplanted embedded view may get marked dirty independently of the parent view into which it is inserted.

API

```
/**
 * Mark a view as dirty and schedule it for change detection.
 *
 * Mark the view as dirty. If the function was called outside of the existing
change
 * detection than schedule a new change detection (unless `option.scheduleCD` is
 * `false`)
 *
 * @param ref Reference to the view. This can either be a component instance or
 *        host element associated with the component.
 * @param options:
 *        - `parent` If set mark all ancestor views as dirty as well.
 *        - `scheduleCD` If set fall then prevent the scheduling of
 *           change detection
 *        - `afterCD` callback to execute after change detection processes the the
 *           view. This is inline with views vs the return promise which is after
 *           all views.
 * @returns A promise which resolves when change detection is completed. (this runs
 *          after all views are processed vs `afterCD` which is inlined with the
 *          view)
 */
function markDirty(ref: ComponentInstance|HTMLElement, options: {
  parent?: boolean,
  scheduleCD?: boolean,
  afterCD?: () => void
```

```
}): Promise<null>;
```

END of developer documentation

Glossary

`processing view`: Invoking the `LView` `template` function and also invoke any life-cycle hooks associated with the component if applicable.

`structural directive`: Directive which creates and inserts embedded views. This is done by directive injecting `ViewContainerRef` (and optionally `TemplateRef`).

`transplanted view`: An `LView` where the declaration `LView` and parent `LView` are different.

Basic proposed mental-model:

1. An Angular application consists of a tree of `LView`s
    a. A parent `LView` can have a child `LView`, as is the case for component views.
    b. A parent `LView` can have an `LContainer`, as is the case with any structural directive such as `*ngFor`, which in turn can contain zero or more `LView`s.
    c. Each `LView` keeps track if it is `DIRTY`/`CHECK_ALWAYS` (or if it contains children who are `DIRTY`/`CHECK_ALWAYS`.)
2. Root `LView` is just an `LView` that has no parents. (A disconnected `LView` is a root `LView`)
3. Marking an `LView` as dirty:
    a. `markDirty()` set current `LView`'s `DIRTY` flag (and schedules CD.)
        i. All invocations of `markDirty()` guarantee that the `LView` will be CDed either as part of:
            1. currently active CD; OR
            2. as part of a future CD, which `markDirty()` schedules through `scheduleCD()`.
        ii. Optionally we can include `markDirty({parents: true})` to set current `LView` as well as all ancestor `LView`s as `DIRTY`. This is similar to `ChangeDetectorRef.markForCheck()`, but unlike `ChangeDetectorRef.markForCheck()`, `markDirty` also schedules CD. This API would be optional and would be included only as a transition from `ChangeDetectorRef.markForCheck()`.
    b. `scheduleCD()` will schedule CD starting at the root `LView` of the current `LView`, which is being marked as dirty.
4. When CD runs, it starts at some `LView` (usually root) and traverses it in a depth-first

fashion.

a. **If** `LView` is marked as `CHECK_ALWAYS` or it contains `CHECK_ALWAYS` children:
    i. If `LView` is marked as `CHECK_ALWAYS` then the `LView`'s `template` function is processed so that the bindings are updated.
    ii. If `LView` contains `CHECK_ALWAYS` children, then we recurse into child `LView`s.
b. **While** `LView` is marked as `DIRTY` or it contains `DIRTY` children:
    i. If `LView` is marked as `DIRTY` then pre-clear the `DIRTY` flag and process the `LView`'s `template` function so that the bindings are updated.
    ii. If `LView` contains `DIRTY,` then we recurse into the child `LView`s.

Note: It may be helpful to ignore the fact that a `LView` can contain `DIRTY`/`CHECK_ALWAYS` children since that is just an optimization to prune the number of `LView`s which need to be scanned/visited while looking for `DIRTY`/`CHECK_ALWAYS`. A simpler mental model may be that a CD simply visits all `LView`s in depth-first order and processes the ones which are labeled as `DIRTY`/`CHECK_ALWAYS`.

Implications:

- CD always follows insertion locations (never declarations.)
- `markDirty` implies that it will be either: 1) CDed as part of this CD; OR 2) New CD is scheduled as a result of `markDirty`. No further scheduling or relying on `zone.js` is necessary.
- If, as part of the CD, an `LView` marks itself directly (or indirectly through intermediate `LView`s) as dirty, the result will be an infinite CD. We may choose to detect this in `ngDevMode` and limit the number of CDs, and throw an Error. (Although there is no easy algorithm to determine if a program will halt.)

Basic change detection

NOTE: assume that all components have no flags (equivalent to `ChangeDetectionStratogy.OnPush`.)

In the following example above, `a` is a root `LView` as it has no parents.

```
      a:LView
      /     \
b:LView    c:LVIew
```

If we `markDirty(c),` the resulting tree will be

```
      a:LView
      /     \
b:LView    c:LVIew [DIRTY]
```

There will also be an implied `scheduleCD(a)` call, which will schedule `a` for CD. (Most likely as part of `requestAnimationFrame`.) Once the CD begins following the mental model from above, the steps will be:

1. cursor at `a`:
    a. `a` is not `DIRTY`, or `CHECK_ALWAYS` => do not process `template`
    b. `b` does not contain `DIRTY`, or `CHECK_ALWAYS` => ignore
    c. `c` contains `DIRTY` or `CHECK_ALWAYS` => descend to `c`.
2. cursor at `c`:
    a. `c` is `DIRTY` => process the `template` to update the bindings.
    b. `c` does not contain `DIRTY`, or `CHECK_ALWAYS` => ascend to `a`
3. cursor at `a`:
    a. `a` does not contain `DIRTY`, or `CHECK_ALWAYS` => ascend (No parent CD => done)

`LContainer` Is really just a collection of `LView`s

The previous example showed only `LView`s, but the same things will occur in a slightly more complicated case that involves `LContainer`s. But the outcome is the same, so we will not discuss `LContainer`s after this example as `LContainer`s do not change the behavior in a material way.

```
          a:LView
         /       \
b:LContainer    c:LContainer
                  /      \
              d:LView    e:LView
```

If we `markDirty(d)` the resulting tree will be

```
          a:LView
         /       \
b:LContainer    c:LContainer
                  /      \
       d:LView[Dirty]    e:LView
```

Once the CD begins following the mental model from above the steps will be:

4. cursor at `a`:
    a. `a` is not `DIRTY` or `CHECK_ALWAYS` => do not process `template`
    b. `b` does not contain `DIRTY` or `CHECK_ALWAYS` => ignore
    c. `c` contains `DIRTY` or `CHECK_ALWAYS` => descend.
5. cursor at `c`:
    a. `c` is `LContainer` => there is no `template`.
    b. `c` is `LContainer` and therefore can't have `DIRTY` or `CHECK_ALWAYS` => descend to `d`
6. cursor at `d`:

    a. `d` is `DIRTY` => invokes a `template` to update the bindings.
    b. `d` does not contain `DIRTY`, or `CHECK_ALWAYS` => ascend `c`
7. cursor at `c`:
    a. `e` does not contain `DIRTY`, or `CHECK_ALWAYS` => ignore
    b. ascend to `a`
8. cursor at `a`:
    a. `a` does not contain `DIRTY` or `CHECK_ALWAYS` => ascend (No parent CD => done)

The implication of this is that `LContainer` will have all of the same counters as `LView`. The only difference is that `LContainer` does not have a `template` and thus has no `DIRTY` or `CHECK_ALWAYS` (or another way to think about it is that `DIRTY` or `CHECK_ALWAYS` are always set to false.)

Calling `markDirty` while in active CD

An important question that should be addressed is what should happen if an `LView` is `markDirty` while it is in the active CD. There are several options:

- **disallow**: Not a good choice since a component will often need to compute new data as a result of it receiving data which it often pushes to the child component. The idea that a parent component can change data which child component uses is well established within Angular and we don't throw `ExpressionChangedAfterItHasBeenChecked` in that case. So disallowing this does not sound like the right approach.
- **schedule new CD**: This is not a good choice as it will cause a lot of CDs to be scheduled. The side effect of that is that users may see intermediate values as the internal state of the application is updated, and as one update triggers another update.
- **process it as part of the current CD**: This seems like the only reasonable option because:
    - It will not show an intermediate state to the users. (Executes as a single transaction.)
    - It will be efficient as:
        - it does not need to schedule a new CD for each `markDirty`. Especially for initial rendering it is very common that the parent component would mark the child component as dirty.
        - CD from the first common ancestor will have a slight benefit as the algorithm will not have to ascend and descend as much.

When calling `markDirty` as part of the current CD, there are two possibilities:

1. `markDirty` is called on an `LView,` which has not yet been passed. (it is in front of the CD wavefront)
2. `markDirty` is called on an `LView,` which has already been processed. (it is behind the CD wavefront)

Let's discuss these implications separately.

Calling `markDirty` in front of CD wavefront

Let's start with the following use case

```
           a:LView
          /       \
b:LView[DIRTY]     c:LView
```

Let's assume that we are doing CD from the root as we have already descended to `b`:

1. cursor at `b`:
    - `b` is `DIRTY` or `CHECK_ALWAYS` => process `template`.
    - side effect of calling `b`'s `template` is that `b` marks `c` as dirty.

      ```
            a:LView
           /       \
      b:LView     c:LView[DIRTY]
      ```

    - ascend to `a`
2. cursor at `a`:
    - `a` contains `DIRTY` or `CHECK_ALWAYS` => descend to `c`.
3. cursor at `c`:
    - `b` is `DIRTY` or `CHECK_ALWAYS` => c process all `template`.

The above behavior should not be surprising and fits well within the current ViewEngine/Ivy behavior.

Calling `markDirty` behind the CD wavefront

The key observation of the previous example is that marking an `LView` as dirty was done in front of the CD wavefront (an `LView` was marked as dirty, which was on our way to visit anyway as part of the depth-first-traversal CD). The next question we need to answer is what should happen if we mark an already visited `LView` as `DIRTY`. Already visited implies that it is behind the current cursor location in the depth-first-traversal.

Here are some things to keep in mind:

- The current state of CD is not something which developer should care about:
    - It is not easy to know if calling `markDirty` is appending itself to the current CD or if it is scheduling a new CD. If we expect to have composable components, then one could argue that knowing/maintaining this information for the developer would be counterproductive to composability and ease of use.
    - Similarly, it is not easy to know when calling `markDirty` on `LView`, where the `LView` is located with respect to the current component (before or after CD wavefront).
- Outside of the context of the behind-CD-wavefront situation calling `markDirty` on any `LView` will result in that `LView` being updated without causing visual flicker for the user.

Because of the above reasons, I (misko@) would argue that `markDirty` should behave consistently no matter if you call it in an existing CD or outside an existing CD or if you call it on after, before, or parent `LView`. The implication of this is that there needs to be a way to retry CD as part of the current CD pass.

```
      a:LView
     /       \
b:LView     c:LVIew[DIRTY]
```

1. ...
2. cursor at `c`
    a. `c` is `DIRTY` => process `template`.
    b. side effect of calling `c`'s `template` is that `b` is marked dirty
    ```
                a:LView
               /       \
    b:LView[DIRTY]     c:LVIew
    ```
    c. ascend to `a`.
3. cursor at `a`
    a. `a` contains `DIRTY` => descend to `b`
4. cursor at `b`
    a. `b` is `DIRTY` => process `template`.

NOTE: Very similar behavior would occur if `markDirty` would be called on parent `LView`.

The implementation detail of the above-described behavior is that:

- Instead of descending to children in a depth-first-traversal => The traversal logic needs to have a `while`-loop which will retry descending to children `LView` if they get re-marked as `DIRTY`.

AngularJS and TTL

At first glance, this looks very similar to AngularJS running `$digest` multiple times until `TTL` is reached. Because AngularJS would re-run `$digest` it was not possible to throw `ExpressionChangedAfterItHasBeenCheckedError` instead the error AngularJS would throw is `Infinite $digest Loop`. It may seem that the two errors are the same, but they are not:

- AngularJS has no way of detecting when data flows backward. It only knows that the `$scope` was not able to stabilize.
- Angular, on the other hand, can detect backwards data flow and throw `ExpressionChangedAfterItHasBeenCheckedError`. This ability will remain even after implementing ideas in this document. Even with a `while`-loop on dirty, it is still possible

to get `ExpressionChangedAfterItHasBeenCheckedError`.

- The key difference between AngularJS and Angular is that in AngularJS, the back-propagation of data was implicit/normal/expected, whereas, with this proposal, back-propagation can only happen if the developer explicitly asks for it through `markDirty`. The difference may seem pedantic, but it is important. In one case, it is implicit and hence back-propagation can happen accidentally, whereas in this proposal, it is explicit and can only happen if the developer opts into it.
    - Additionally, we could allow enforcement of back-propagation by supporting API such as this: `markDirty({allowBackPropagation: false})`

Should `markDirty` schedule CD?

As currently proposed, `markDirty` would schedule CD at some later point through `requestAnimationFrame` **IF** not already scheduled. In essence, `markDirty` calls are coalesced. The coalescing is an important feature as it should not be the responsibility of the developer to keep track if an `LView` should be dirty and if CD has been scheduled. The argument here is that coalescing is a desirable property, and scheduling should not be the responsibility of the developer.

Having said that, it might be useful to have APIs such as:

- `markDirty({parents: true})`. Set all ancestors as well.
    - Useful for migration from `ChangeDetectionRef.markForCheck()` to `markDirty` to have an equivalent API.
- `markDirty({scheduleCD: false})`. Allows control of scheduling/coalescing.
    - Useful to mark a component as dirty but setting a low priority which should eventually be performed (rather than now). For example, low priority updates.
- `markDirty({afterCD: () => console.log('afterCD')})` to get notified when CD runs.
    - This is useful when code would like to read DOM, such as the width of the component but this can only be done after the DOM is updated. So having a callback would be useful.

Implementation Details

This section talks about possible implementation details, but it should not change the mental model presented above.

Contains `DIRTY`/`CHECK_ALWAYS` counts

As a mental model it is useful to think about the CD as visiting all `LView`s in depth-first-order, but only process `template` functions for `LView`s which are marked as `DIRTY` or `CHECK_ALWAYS`. While this is a useful mental model to reason about the algorithm, we can get better runtime performance by pruning the `depth-first-order` tree to only visit branches that contain `DIRTY`

or CHECK_ALWAYS. This pruning can be achieved by each LView storing DIRTY/CHECK_ALWAYS child counts in addition to whether the LView itself is DIRTY/CHECK_ALWAYS.

The algorithm talks about DIRTY and LView only, but the same rules will apply to CHECK_ALWAYS and LContainer as well. Additionally, both DIRTY and CHECK_ALWAYS counts can be stored in a single word by dedicating half of the bits to DIRTY and half to the CHECK_ALWAYS further improving efficiency:

1. LView count should reflect how many child LViews contain the flag.
    a. Marking LView as dirty should NOT increment its own count.
    b. If LView count transitions from 0->1 OR the LView is marked => the parent LView should be incremented.
    c. If LView count transitions from 1->0 AND LView is NOT marked => the parent LView should be decremented.
2. During CD
    a. If LView count is >0 this implies that there is at least one child view that has the flag.
        i. Iterate over all of the child LViews to determine if any of them have the flag set if they do process the template function.
        ii. If a child LView has the count set for any flag, then recurse into the LView.

NOTE: The count only keeps track of how many immediate children have the flag, not the total number of flags in the branch. This is done intentionally so that adding LViews would not have to update LView counts all the way to the root on each addition/removal. In essence, each LView level performs coalescing to minimize the number of count updates.

```
            a:LView[CHILD_DIRTY_COUNT:1]
           /       \
     b:LView        c:LVIew[CHILD_DIRTY_COUNT:2]
                   /       \
    d:LView[DIRTY]           e:LView[DIRTY]
```

The above example is for DIRTY but the same rules will apply for CHECK_ALWAYS:

- d and e are marked as DIRTY.
- c is not marked as DIRTY but CHILD_DIRTY_COUNT is set to 2 as there are 2 children d and e marked as DIRTY.
- a is not marked as DIRTY but CHILD_DIRTY_COUNT is set to 1 as there is 1 child c marked as DIRTY.

Pre-clearing flags before the template

As of the current implementation of CD, the DIRTY flag is cleared after the template function is processed. This will have to change to clear before template processing. The reason for this is

the processing of the `template` can mark the `LView` as `DIRTY`. Doing post-clear would not correctly handle this use case.

Retry CD strategy

NOTE: This section deals with when should `LView` be executed within the current CD, not whether the `LView` should be executed as part of this or some future CD. The latter was discussed in Calling markDirty while in active CD.

When an `LView` is marked as `DIRTY` behind the CD wavefront the question is when should the CD re-process the `DIRTY` flag in `LView`? There are two possibilities:

1. The retry CD logic should be performed at the root of the `LView` tree.
    a. PRO: Better coalescing.
    b. CON: It does not work when CD is started from place other than root `LView`.
2. The retry CD logic should be performed immediately before each ascends. *[preferred option]*
    a. PRO: Consistent behavior regardless if the CD was invoked from root or from an arbitrary location in `LView` tree.
    b. PRO: Compatible with Fractal `LView`s goal.
    c. CON: In pathological cases, we don't coalesce, and so we may end up descending to `LView`s more often than in the optimal case.

Fractal `LView`s

> NOTE: This section deserves its own design doc. It is included here only to point out that the above design takes this into account.

Fractal is a mathematical concept that can be summarized as having the same shape-ness at any zoom level. In this context, fractal means that `LView`s behave the same in isolation as they do in `LView`-forrest or as part of `ApplicationRef`.

Currently, `LView`s do not have fractal property because CD is tied to `ApplicationRef` which makes that root `LView` special. In Ivy this is reflected as the root `LView` having `RootContext`.

Problems with `RootContext`

- `RootContext` contains key pieces without which processing `LView` CD becomes problematic:
    - `scheduler`: Is needed to schedule CDs.
    - `playerHandler`: Needed to process animations correctly.
    - `clean` promise: Needed by protractor to know when to assert UI.
- Once `LView` gets disconnected from the special root `LView`, it loses its pointer to the `RootContext` which means it can't participate in proper CD, animation, or Protractor. The implication is that `LView`s can only be created in the `RootContext` and once disconnected, can't start acting on their own.

The main point of this section is that going forward, our goal should be to have fractal `LView`s. Therefore the `RootContext` should not be the place where the `DIRTY` flag gets retried, as disclosed above.

Benefits

So far, the document discussed how the CD should be working. This section answers what benefits we would gain if the CD worked as described above.

Transplanted views

A transplanted view problem has been discussed here. With the above CD mental model, the transplanted view mental model becomes straightforward as well.

Algorithm:

- When processing the `template` function on an `LView` (either because it is dirty or check-always)
    - If `LView` contains `<ng-template>` instances that are transplanted, mark all of the transplanted `LView`s as `Dirty`.

The above algorithm will have the property that:

- CD is always run at the insertion point only.
- If the parent `LView` of the transplanted view insertion is disconnected, it will not be part of CD.

In the above algorithm, transplanted views are not handled in a special way, rather they just naturally fall out of the proposed CD mental model. This simplifies Angular for our developers, which is a very desirable outcome.

Forms need to back-propagate from state

Forms often run into the issue that a validator can only run after the whole form has been CDed. Validators can produce errors that are often placed above the form. The result is that there is a backward data flow of information. (Validation error needs to be displayed above the form.)

The above problem can be solved by a validator simply marking the destination `LView` as dirty. (There is an open question as to how does the validator know what is the destination `LView` but that is outside the scope of this document)

Angular Elements

Angular Elements are Angular applications that need to be bootstrapped with a common `PlatformRef` (or `RootContext`). Having `LView`s be fractal would mean that the Angular Elements could be bootstrapped independently and composed into an `LView` tree if appropriate or broken apart into independent `LView`s and have each Angular Element have its own CD. This is a desirable and often requested feature from the community.

Zoneless

> NOTE: This section deserves its own design doc. It is included here only to point out that the above design takes this into account.

One of the things which we have been discussing is that `zone.js` will have to be retired in the near future as it is not compatible with ES2018 `async`/`await` statements. A replacement for `zone.js` will most likely require some sort of explicit state management. As these states get updated, they will need to be able to signal which `LView`s need to be updated—using `markDirty` fits nicely with this ability. It is possible that state management may result in backpropagation of information, as is an example with forms.

Partial DOM rehydration (SSR)

> NOTE: This section deserves its own design doc. It is included here only to point out that the above design takes this into account.

When performing SSR, one goal would be that the components can be rehydrated in any order. The implication of partial out-of-order hydration is that one can not rely on having `RootContext`, which further justifies the fractal view approach. If `RootContext` needs to be present for CD, then it is not possible to rehydrate a specific component without also rehydrating all of the parent components all the way to the `RootContext` which is not a desirable property.

Performance

The cost of tracking the counts should be about the same as the cost of tracking flags themselves. So from that point of view, we don't expect any change in performance.

However, because this proposal allows CDing just specific `LView`s (rather than all `LView`s between the mark dirty and root location) we do expect that to have significant performance benefits if `markDirty` API is used compared to `ChangeDetectorRef.markForCheck()`. Since this is a new API, it will not benefit existing applications.

Work Breakdown

The work can be broken down into these categories:

1. **Change detection**: Make changes to Change Detection to match the above proposal. This has already been prototyped in #35428.
    a. Add contains-child-dirty counters to `LView` and `LContainer`.
    b. Update the current rules about marking all declared views as dirty(including transplanted views)
2. Create `scheduleCD` API:
    a. Create a way to schedule CD. A trivial implementation is already in the repository.
    b. Requires some design so that the scheduling work can work in a fractal manner.
3. New public API `markDirty`:
    a. Expose existing API as public
    b. Add additional options to `markDirty` such as `{parents, etc...}`.
4. Documentation: Public documentation

For now, only #1 is needed to fix the transplanted views issue.

TODO:

- Google3 experiment: Run with `{parents:false}` to see what things break.