I reverse-engineered Zones (zone.js) and here is what I've found

A gentle introduction into zones: what it is and how to use it

Max Koretskyi aka Wizard Follow
Sep 28, 2017 · 14 min read

> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

Zones is a new mechanism that helps developers work with multiple logically-connected async operations. Zones work by associating each async operation with a zone. A developer can take advantage of this binding to:

- Associate some data with the zone, analogous to thread-local storage in other languages, which is accessible to any async operation inside the zone.
- Automatically track outstanding async operations within a given zone to perform cleanup or rendering or test assertion steps
- Time the total time spent in a zone, for analytics or in-the-field profiling
- Handle all uncaught exceptions or unhandled promise rejections within a zone, instead of letting them propagate to the top level

Most articles on the web either describe outdated API or explain Zones using significantly simplified analogies. In this article I'm using the latest API and exploring the essential API in great details as close to the implementation as possible. I start by describing the API, then show async task association mechanism and continue with interception hooks that a developer can use to to perform the tasks listed above. In the end of the article I provide a short explanation of how Zones works under the hood.

Zones is currently a stage 0 proposal to the EcmaScript standard that is at the moment being blocked by Node. Zones are usually referred to as `Zone.js` and this is the name of the github repository and npm package. However, in this article I'll be using the name `Zone` as it's specified in the spec. **Please note that this article is not about NgZone, but about the mechanism NgZone builds upon—Zones (zone.js).** By knowing the material I present in this article you will be able to create your own NgZone or understand how existing NgZone works. To learn more about NgZone read Do you still think that NgZone (zone.js) is required for change detection in Angular?

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

• • •

Relevant Zone API

Let's first take a look at the most relevant methods when working with Zones. The class has the following interface:

```
class Zone {
  constructor(parent: Zone, zoneSpec: ZoneSpec);

  static get current();

  get name();
  get parent();

  fork(zoneSpec: ZoneSpec);
  run(callback, applyThis, applyArgs, source);
  runGuarded(callback, applyThis, applyArgs, source);
  wrap(callback, source);

}
```

Zones have a notion of a **current zone** which is crucial. The current zone is the async context that propagates with all async operations. It represents the zone associated with a stack frame/async task currently being executed. This current zone can be accessed using static getter `Zone.current`.

Each zone has `name`, which is mostly used for tooling and debugging purposes. It also defines methods designed to manipulate zones:

- `z.run(callback, ...)` synchronously invokes a function in a given zone. It sets the current zone to `z` when executing `callback` and resets it to its previous value once the callback has finished executing. Executing a callback in a zone is usually referred to as "entering" a zone.
- `z.runGuarded(callback, ...)` same as `run` but catches runtime errors and provides a mechanism to intercept them. If an error is not handled by any parent Zone it's re-thrown.
- `z.wrap(callback)` produces a new function that captures `z` in a closure and essentially performs `z.runGuarded(callback)` when executed. If a callback later is passed to `other.run(callback)` it is still executed in `z` zone, not `other`. The mechanism is similar in the idea to how `Function.prototype.bind` works in JavaScript.

In the next section we will talk about `fork` method at length. Zone also has a bunch of methods to run, schedule and cancel a task:

```
class Zone {
  runTask(...);
  scheduleTask(...);
  scheduleMicroTask(...);
  scheduleMacroTask(...);
  scheduleEventTask(...);
  cancelTask(...);
```

These are low-level methods which are rarely used by developers so I won't talk about them in details in this article. Scheduling a task is an internal operation of Zone and for a developer it usually means simply calling some async operation like `setTimeout`.

Persisting the zone across a call stack

JavaScript VM executes each function in it's own stack frame. So if you have a code like this:

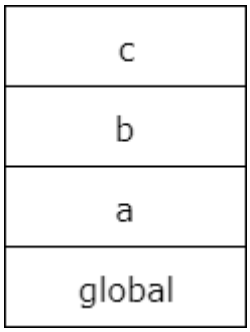
```
function c() {
    // capturing stack trace
    try {
        new Function('throw new Error()')();
    } catch (e) {
        console.log(e.stack);
    }
}

function b() { c() }
function a() { b() }

a();
```

Inside the `c` function it has the following call stack:

```
at c (index.js:3)
at b (index.js:10)
at a (index.js:14)
at index.js:17
```

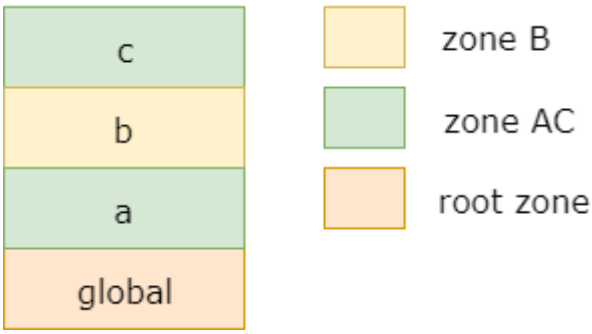
The approach for capturing stack trace I used in the `c` function is described at MDN website.

The callstack can be drawn like this:



So we have 3 stack frames for our function calls and one stack for global context.

In the regular JavaScript environment, the stack frame for the function `c` is not in any way associated with a stack frame for the function `a`. What Zone allows us to do is to associate each stack frame with a particular zone. For example, we can associate stack frames `a` and `c` with the same zone effectively linking them together. So we end up with the following:



We will see in a minute how this can be done.

Creating a child zone with zone.fork

One of most used features of Zones is creating a new zone using the `fork` method. Forking a zone creates a new child zone and sets its `parent` to the zone used for forking:

```
const c = z.fork({name: 'c'});
console.log(c.parent === z); // true
```

The `fork` method under the hood simply creates a new zone using the class:

```
new Zone(targetZone, zoneSpec);
```

So to accomplish our task of associating `a` and `c` functions with the same zone we first need to create that zone. To do that we will use the `fork` method I showed above:

```
const zoneAC = Zone.current.fork({name: 'AC'});
```

The object that we pass to the `fork` method is called zone specification (ZoneSpec) and has the following properties:

```
interface ZoneSpec {
    name: string;
    properties?: { [key: string]: any };

    onFork?: ( ... );
    onIntercept?: ( ... );
    onInvoke?: ( ... );
    onHandleError?: ( ... );
    onScheduleTask?: ( ... );
    onInvokeTask?: ( ... );
    onCancelTask?: ( ... );
    onHasTask?: ( ... );
```

`name` defines the name of a zone and `properties` is used to associate data with a zone. All other properties are interception hooks that allow parent zone intercept certain operations of child zones. It's important to understand that forking creates zones hierarchy and all methods on Zone class that manipulate zones can be intercepted by parent zones using hooks. Later in the article we will see how we can use `properties` to share data between async operations and hooks to implement tasks tracking.

Let's create one more child zone:

```
const zoneB = Zone.current.fork({name: 'B'});
```

Now that we have two zones we can use them to execute functions inside a particular zone. To do that we can use `zone.run()` method.

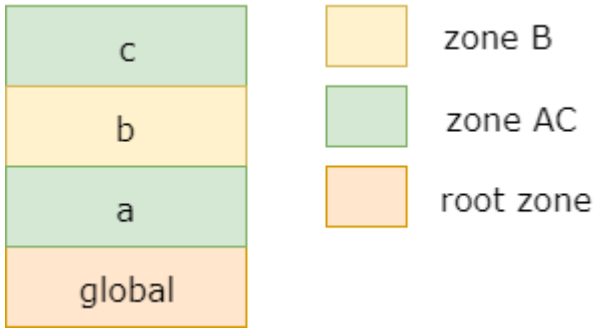
Switching zones with zone.run

And to make a particular stack frame associated with a zone we need to run the function in that zone using `run` method. As you know it synchronously runs a callback in a specified zone and after its completion it restores the zone.

So let's apply that knowledge and slightly modify our example:

```
function c() {
    console.log(Zone.current.name);  // AC
}

function b() {
    console.log(Zone.current.name);  // B
    zoneAC.run(c);
}

function a() {
    console.log(Zone.current.name);  // AC
    zoneB.run(b);
}

zoneAC.run(a);
```

And every call stack is now associated with a zone:



As you can see from above code we executed each function using the `run` method which directly specified which zone to use. You're

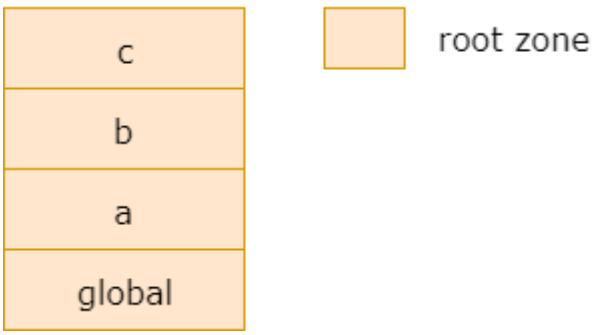
probably wondering now what happens when we don't use `run` method and simply execute the function inside the zone?

It's important to understand that all function calls and asynchronous tasks scheduled inside the function will be executed in the same zone as this function.

We know that zones environment always has a root zone. So if we don't switch zones with `zone.run` we expect all functions to be executed in `root` zone. Let's see if it's the case:

```
function c() {
    console.log(Zone.current.name);  // <root>
}
function b() {
    console.log(Zone.current.name);  // <root>
    c();
}
function a() {
    console.log(Zone.current.name);  // <root>
    b();
}
a();
```

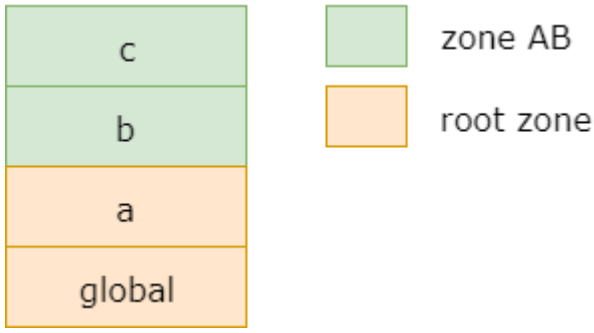
Yep, that's the case. Here is the diagram:



And if we only use `zoneAB.run` once in the `a` function, `b` and `c` will be executed in the `AB` zone:

```
const zoneAB = Zone.current.fork({name: 'AB'});

function c() {
    console.log(Zone.current.name);  // AB
}

function b() {
    console.log(Zone.current.name);  // AB
```

```
    c();
}

function a() {
    console.log(Zone.current.name);  // <root>
    zoneAB.run(b);
}

a();
```



You can see that we explicitly call `b` function in the `AB` zone. However, the `c` function is also executed in this zone.

. . .

Persisting the zone across async tasks

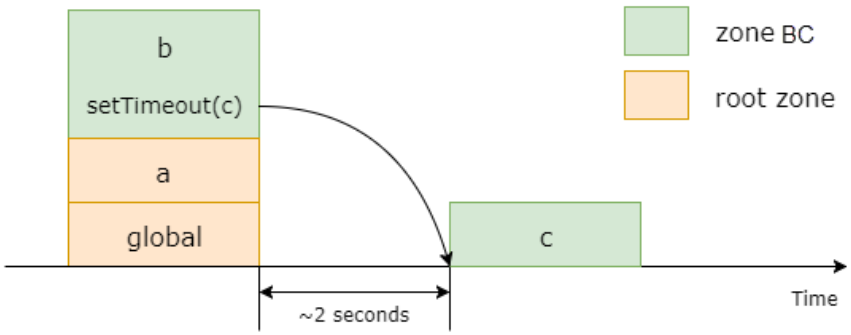
One of the distinct characteristics of JavaScript development is asynchronous programming. Probably most new JS developers become familiar with this paradigm using `setTimeout` method that allows postponing execution of a function. Zone calls `setTimeout` async operation a task. Specifically, a macrotask. Another category of tasks is a microtask, for example, a `promise.then` . This terminology is used internally by a browser and Jake Archibald explains it in depth in the Tasks, microtasks, queues and schedules.

So let's see now how Zone handles asynchronous tasks like `setTimeout` . To do that we will just use the code we used above but instead of immediately calling function `c` we will pass it as a callback to the `setTimeout` function. So this function will be executed **in the separate call stack** sometime in the future (approximately in 2 seconds):

```
const zoneBC = Zone.current.fork({name: 'BC'});

function c() {
    console.log(Zone.current.name);  // BC
}

function b() {
    console.log(Zone.current.name);  // BC
    setTimeout(c, 2000);
}

function a() {
    console.log(Zone.current.name);  // <root>
    zoneBC.run(b);
}

a();
```

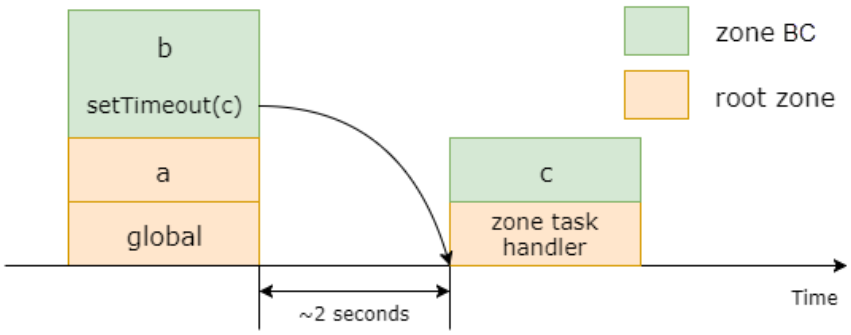
We learnt above that if we call a function inside a zone that function will be executed in the same zone. And this behavior applies to an asynchronous operation as well. If we schedule an asynchronous task and specify a callback function then this function will be executed in the same zone from which the task is scheduled.

So if we draw a history of calls we will have the following:



That's very nice, however, this diagram hides important implementation details. Under the hood, Zone has to **restore** the correct zone for each task it's about to execute. To do so it has to remember what zone this task should be executed and it does so by keeping a reference to associated zone on the task. This zone is then used to invoke a task from handler in the root zone.

It means that a callstack for every asynchronous task always starts with the root zone that uses the information associated with a task to restore correct zone and then invoke the task. So here is more accurate representation:



. . .

Propagating context across async tasks

Zone has several interesting capabilities that a developer can take advantage of. One of such capabilities is context propagation. It simply means that we can attach data to a zone and access this data inside any task that is executed inside that zone.

Let's use our last example and demonstrate how we can persist data across `setTimeout` async task. As you learnt earlier when forking a new zone we pass the zone spec object. This object can have an optional property `properties`. We can use this property to associate data with a zone like this:

```
const zoneBC = Zone.current.fork({
    name: 'BC',
    properties: {
        data: 'initial'
    }
});
```

And then it can be accessed using `zone.get` method:

```
function a() {
    console.log(Zone.current.get('data')); // 'initial'
}

function b() {
    console.log(Zone.current.get('data')); // 'initial'
    setTimeout(a, 2000);
}

zoneBC.run(b);
```

The object that `properties` property points at is shallow-immutable which means you can't add/remove this object properties. This is largely because Zone doesn't provide any methods to do so. So in the example above we can't set different value for `properties.data`.

However, we can pass an object to `properties.data` instead of a primitive and then we'll be able to modify the data:

```
const zoneBC = Zone.current.fork({
    name: 'BC',
    properties: {
        data: {
            value: 'initial'
        }
    }
});

function a() {
    console.log(Zone.current.get('data').value); //
'updated'
}

function b() {
    console.log(Zone.current.get('data').value); //
'initial'
    Zone.current.get('data').value = 'updated';
    setTimeout(a, 2000);
}

zoneBC.run(b);
```

It's also interesting that child zones created using `fork` method inherit properties from the parent zones:

```
const parent = Zone.current.fork({
    name: 'parent',
    properties: { data: 'data from parent' }
});

const child = parent.fork({name: 'child'});

child.run(() => {
    console.log(Zone.current.name); // 'child'
    console.log(Zone.current.get('data')); // 'data from
parent'
});
```

• • •

Tracking outstanding tasks

Another capability that is probably much more interesting and useful is the ability to track outstanding asynchronous macro and micro tasks. Zone keeps all outstanding tasks in the queue. To get notified whenever this queue status changes we can use `onHasTask` hook of the zone spec. Here's its signature:

```
onHasTask(delegate, currentZone, targetZone, hasTaskState);
```

Since parent zones can intercept child zones events Zone supplies `currentZone` and `targetZone` parameters to distinguish between a zone that has changes in the tasks queue and the zone that intercepts the event. So, for example, if you need to make sure that you're intercepting the event for the current zone just compare zones:

```
// We are only interested in event which originate from our
zone
if (currentZone === targetZone) { ... }
```

The last parameter passed to the hook is `hasTaskState` which describes the status of the task queue. Here's its signature:

```
type HasTaskState = {
    microTask: boolean;
    macroTask: boolean;
    eventTask: boolean;
    change: 'microTask'|'macroTask'|'eventTask';
};
```

So if you call `setTimeout` inside a zone you will get the `hasTaskState` object with the following values:

```
{
    microTask: false;
    macroTask: true;
    eventTask: false;
    change: 'macroTask';
}
```

which states that there's a pending macrotask in the queue and the change in the queue comes from the `macrotask`.

So let's this in action:

```
const z = Zone.current.fork({
    name: 'z',
    onHasTask(delegate, current, target, hasTaskState) {
        console.log(hasTaskState.change);          // "macroTask"
        console.log(hasTaskState.macroTask);       // true
        console.log(JSON.stringify(hasTaskState));
    }
});

function a() {}

function b() {
    // synchronously triggers `onHasTask` event with
    // change === "macroTask" since `setTimeout` is a macrotask
    setTimeout(a, 2000);
}

z.run(b);
```

And we get the following output:

```
macroTask
true
{
    "microTask": false,
    "macroTask": true,
    "eventTask": false,
    "change": "macroTask"
}
```

Whenever in two seconds the timeout is finished executing `onHasTask` is triggered again:

```
macroTask
false
{
    "microTask": false,
    "macroTask": false,
    "eventTask": false,
    "change": "macroTask"
}
```

There's however one caveat. You can use `onHasTask` hook only to track the `empty/non-empty` state **of the entire tasks queue**. You **can't use it to track individual tasks.** If you run the following code:

```
let timer;

const z = Zone.current.fork({
    name: 'z',
    onHasTask(delegate, current, target, hasTaskState) {
        console.log(Date.now() - timer);
        console.log(hasTaskState.change);
        console.log(hasTaskState.macroTask);
    }
});

function a1() {}
function a2() {}

function b() {
    timer = Date.now();
    setTimeout(a1, 2000);
    setTimeout(a2, 4000);
}

z.run(b);
```

you'll get the following output:

```
1
macroTask
true

4006
macroTask
false
```

You can see that there's no event for the `setTimeout` task that completed in `2` seconds. The `onHasTask` hook is triggered once when the the first `setTimeout` is scheduled and the tasks queue state is changed from `non-empty` to `empty` and it's triggered second time in `4` seconds when the last `setTimeout` callback has completed.

If you want to track individual tasks you need to use `onSheduleTask` and `onInvoke` hooks.

onSheduleTask and onInvokeTask

Zone spec defines two hooks that can be used to track individual tasks:

- onScheduleTask
  executed whenever an async operation like `setTimeout` is detected
- onInvokeTask
  executed when a callback passed to an async operation like `setTimeout(callback)` is executed

Here is how you can use these hooks to track individual tasks:

```
let timer;

const z = Zone.current.fork({
    name: 'z',

    onScheduleTask(delegate, currentZone, targetZone, task)
{
      const result = delegate.scheduleTask(targetZone,
task);
      const name = task.callback.name;

      console.log(
          Date.now() - timer,
         `task with callback '${name}' is added to the task
queue`
      );

      return result;
    },

    onInvokeTask(delegate, currentZone, targetZone, task,
...args) {
      const result = delegate.invokeTask(targetZone, task,
...args);
      const name = task.callback.name;

      console.log(
        Date.now() - timer,
       `task with callback '${name}' is removed from the
task queue`
     );

     return result;
    }
});

function a1() {}
function a2() {}

function b() {
    timer = Date.now();
```

```
    setTimeout(a1, 2000);
    setTimeout(a2, 4000);
}

z.run(b);
```

And here is the expected output:

```
1 “task with callback ‘a1’ is added to the task queue”
2 “task with callback ‘a2’ is added to the task queue”
2001 “task with callback ‘a1’ is removed from the task
queue”
4003 “task with callback ‘a2’ is removed from the task
queue”
```

Intercepting zone “enter” with onInvoke

A zone can be entered (switched) either explicitly by calling `z.run()` or implicitly by invoking a task. In the previous section I explained the `onInvokeTask` hook that can be used to intercept zone entering when Zone internally executes a callback associated with an asynchronous task. There’s also another hook `onInvoke` that you can use to get notified whenever the zone is entered by running `z.run()` .

Here is an example of how it can be used:

```
const z = Zone.current.fork({
    name: 'z',
    onInvoke(delegate, current, target, callback, ...args) {
        console.log(`entering zone '${target.name}'`);
        return delegate.invoke(target, callback, ...args);
    }
});

function b() {}

z.run(b);
```

And the output is:

```
entering zone ‘z’
```

How `Zone.current` works under the hood

Current zone is tracked using _currentZoneFrame variable that gets captured into a closure here and is returned by the `Zone.current` getter. So in order to switch the zone simply the `__currentZoneFrame` variable needs to be updated. And you now that a zone can be switched either by running `z.run()` or invoking a task.

So here is where `run` method updates the variable:

```
class Zone {
   ...
   run(callback, applyThis, applyArgs,source) {
      ...
      _currentZoneFrame = {parent: _currentZoneFrame, zone:
this};
```

And the `runTask` updates the variable here:

```
class Zone {
   ...
   runTask(task, applyThis, applyArgs) {
      ...
      _currentZoneFrame = { parent: _currentZoneFrame, zone:
this };
```

The `runTask` method is called by the `invokeTask` method that each task has:

```
class ZoneTask {
    invokeTask() {
         _numberOfNestedTaskFrames++;
      try {
          self.runCount++;
          return self.zone.runTask(self, this, arguments);
```

Every task when created saves its zone in the `zone` property. And this is exactly the zone that is used to `runTask` inside `invokeTask`

(`self` refers to the task instance here):

```
self.zone.runTask(self, this, arguments);
```

. . .

Additional resources

Here are some of the good resources if you want to get more information about Zones:

- A talk by Brian Ford
- Zone Primer google doc
- Github sources (for the brave ones)

. . .

Many thanks to JiaLi for answering my questions related to Zones on GitHub!

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.