Internals Of Zone.js

This document explores the internals of the Zone.js library and how it is used from within a sample large framework, Angular 8.

All Zone.js source extracts are from v0.9 (the version used by Angular 8)
All Angular 8 source extracts are from 8.0.0-rc.3

Overview

The Zone.js project provides multiple asynchronous execution contexts running within a single JavaScript thread (i.e. within the main browser UI thread or within a single web worker). Zones are a way of sub-dividing a single thread using the JavaScript event loop. A single zone does not cross thread boundaries. A nice way to think about zones is that they sub-divide the stack within a JavaScript thread into multiple mini-stacks, and sub-divide the JavaScript event loop into multiple mini event loops that seemingly run concurrently (but really share the same VM event loop). In effect, Zone.js helps you write multithreaded code within a single thread.

When Zone.js is loaded by your app, it monkey-patches certain asynchronous calls (e.g. `setTimer`, `addEventListener`) to implement zone functionality. Zone.js does this by adding wrappers to the callbacks the application supplies, and when a timeout occurs or an event is detected, it runs the wrapper first and then the application callback. Chunks of executing application code form tasks and each task executes in the context of a zone. Zones are arranged in a hierarchy and provide useful features in areas such as error handling, performance measuring and executing configured work items upon entering and leaving a zone (all of which might be of great interest to implementors of change detection in a modern web UI framework, like Angular 8).

Zone.js is mostly transparent to application code. Zone.js runs in the background and for the most part "just works". For example, Angular 8 uses Zone.js and Angular application code usually runs inside a zone. Tens of thousands of Angular application developers have their code execute within zones without evening knowing these exist. More advanced applications can have custom code to make zone API calls if needed and become more actively involved in zone management. Some application developers may wish to take certain steps to move some non-UI performance-critical code outside of the Angular zone – using the `NgZone` class).

Project Information

The homepage and root of the source tree for Zone.js is at:

- https://github.com/angular/zone.js

Below we assume you have got the Zone.js source tree downloaded locally under a

directory we will call <ZONE-MASTER> and any pathnames we use will be relative to that. We also will look at the Angular code that calls Zone.js and we use the <ANGULAR-MASTER> as the root of that source tree.

Zone.js is written in TypeScript. It has no package dependencies (its package.json has this entry: `"dependencies": {}`), though it has many `devDependencies`. It is quite a small source tree, whose size (uncompressed) is about 3 MB.

Using Zone.js

Before looking in detail at how Zone.js itself is implemented, we will first look at how it is used in production, using the example of the very popular Angular 8 project.

The use of Zone.js with Angular is optional, but used by default. Use of zones is mostly a good idea but there are some scenarios (e.g. using Angular Elements to build Web Compoenents) when this is not the case.

To use Zone.js in your applications, you need to load it. Your package.json file will need (if creating a project using Angular CLI, this is added automatically for you):

```
"dependencies": {
    ..
    "zone.js": "<version>"
  },
```

You should load Zone.js after loading core.js (if using that). For example, if using an Angular application generated via Angular CLI (as most production apps will be), Angular CLI will generate a file called <project-name>/src/polyfills.ts and it will contain:

```
/***************************************************************************
 * Zone JS is required by default for Angular itself.
 */
import 'zone.js/dist/zone';  // Included with Angular CLI.
```

Angular CLI also generates an angular.json configuration file, with this line that sets up polyfills:

```
"build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            ..
            "polyfills": "src/polyfills.ts",
```

If writing your application in TypeScript (recommended), you also need to get access to the ambient declarations. These define the Zone.js API and are supplied in:

- <ZONE-MASTER>/dist/zone.js.d.ts

(IMPORTANT: This file is particularly well documented and well worth some careful study by those learning Zone.js). Unlike declarations for most other libraries, zone.js.d.ts does not use `import` or `export` at all (those constructs do not appear even once in that file). That means application code wishing to use zones cannot simply import its .d.ts file, as is normally the case. Instead, the `///reference` construct needs to be used. This includes the referenced file at the site of the `///reference` in the containing file. The benefit of this approach is that the containing

file itself does not have to (but may) use `import`, and thus may be a script, rather than having to be a module. The use of zones is not forcing the application to use modules (however, most larger applications, including all Angular applications - will). How this works is best examined with an example, so lets look at how Angular includes zone.d.ts. Angular contains a file, types.d.ts under its packages directory (and a similar one under its modules directory and tools directory):

- <ANGULAR-MASTER>/packages/types.d.ts

and it has the following contents:

```
/// <reference types="hammerjs" />
/// <reference types="jasmine" />
/// <reference types="jasminewd2" />
/// <reference types="node" />
/// <reference types="zone.js" />
/// <reference lib="es2015" />
/// <reference path="./system.d.ts" />
```

Use Within Angular

When writing Angular applications, all your application code runs within a zone, unless you take specific steps to ensure some of it does not. Also, most of the Angular framework code itself runs in a zone. When beginning Angular application development, you can get by simply ignoring zones, since they are set up correctly by default for you and applications do not have to do anything in particular to take advantage of them. The end of the file <ZONE-MASTER>/blob/master/MODULE.md explains where Angular uses zones:

"Angular uses zone.js to manage async operations and decide when to perform change detection. Thus, in Angular, the following APIs should be patched, otherwise Angular may not work as expected.

- ZoneAwarePromise
- timer
- on_property
- EventTarget
- XHR"

Zones are how Angular initiates change detection – when the zone's mini-stack is empty, change detection occurs. Also, zones are how Angular configures global exception handlers. When an error occurs in a task, its zone's configured error handler is called. A default implementation is provided and applications can supply a custom implementation via dependency injection. For details, see here:

- https://angular.io/api/core/ErrorHandler

On that page note the code sample about setting up your own error handler:

```
class MyErrorHandler implements ErrorHandler {
  handleError(error) {
    // do something with the exception
  }
}
@NgModule({
  providers: [{provide: ErrorHandler, useClass: MyErrorHandler}]
})
class MyModule {}
```

Angular provide a class, `NgZone`, which builds on zones:

- https://angular.io/api/core/NgZone

As you begin to create more advanced Angular applications, specifically those involving computationally intensive code that does not change the UI midway through the computation (but may at the end), you will see it is desirable to place such CPU-intensive work in a separate zone, and you would use a custom `NgZone` for that.

Elsewhere we will be looking in detail at `NgZone` and the use of zones within Angular in general when we explore the source tree for the main Angular project later, but for now, note the source for `NgZone` is in:

- <ANGULAR-MASTER>/packages/core/src/zone

and the zone setup during bootstrap for an application is in:

- <ANGULAR-MASTER>/packages/core/src/application_ref.ts

When we bootstrap our Angular applications, we either use `bootstrapModule<M>` (using the dynamic compiler) or `bootstrapModuleFactory<M>` (using the offline compiler). Both these functions are in application_ref.ts. `bootstrapModule<M>` calls the Angular compiler **1** and then calls `bootstrapModuleFactory<M>` **2**.

```
  bootstrapModule<M>(
      moduleType: Type<M>, compilerOptions:
(CompilerOptions&BootstrapOptions)|
      Array<CompilerOptions&BootstrapOptions> = []): Promise<NgModuleRef<M>>
{
    const options = optionsReducer({}, compilerOptions);
1   return compileNgModuleFactory(this.injector, options, moduleType)
        .then(moduleFactory
2                   => this.bootstrapModuleFactory(moduleFactory, options));
}
```

It is in `bootstrapModuleFactory` we see how zones are initialized for Angular:

```
bootstrapModuleFactory<M>(moduleFactory: NgModuleFactory<M>, options?:
BootstrapOptions):
      Promise<NgModuleRef<M>> {
    // Note: We need to create the NgZone _before_ we instantiate the module,
    // as instantiating the module creates some providers eagerly.
    // So we create a mini parent injector that just contains the new NgZone
    // and pass that as parent to the NgModuleFactory.
    const ngZoneOption = options ? options.ngZone : undefined;
    1 const ngZone = getNgZone(ngZoneOption);
    const providers: StaticProvider[]=[{provide: NgZone, useValue: ngZone}];
    // Attention: Don't use ApplicationRef.run here,
    // as we want to be sure that all possible constructor calls
    // are inside `ngZone.run`!
    2 return ngZone.run(() => {
      const ngZoneInjector = Injector.create(
          {providers: providers, parent: this.injector,
                          name: moduleFactory.moduleType.name});
      const moduleRef =
            <InternalNgModuleRef<M>>moduleFactory.create(ngZoneInjector);
      const exceptionHandler: ErrorHandler =
      3               moduleRef.injector.get(ErrorHandler, null);
      if (!exceptionHandler) {
```

```
          throw new Error(
              'No ErrorHandler. Is platform module (BrowserModule) included?');
        }
        moduleRef.onDestroy(() => remove(this._modules, moduleRef));
        ngZone !.runOutsideAngular(
        4    () => ngZone !.onError.subscribe(
                 {next: (error: any) =>
                              { exceptionHandler.handleError(error); }}));
        return _callAndReportToErrorHandler(exceptionHandler, ngZone !,
        () => {
          const initStatus: ApplicationInitStatus =
                              moduleRef.injector.get(ApplicationInitStatus);
          initStatus.runInitializers();
          return initStatus.donePromise.then(() => {
        5   this._moduleDoBootstrap(moduleRef);
            return moduleRef;
          });
        });
      });
    }
```

At 1 we see a new `NgZone` being created and at 2 its `run()` method being called, at 3 we see an error handler implementation being requested from dependency injection (a default implementation will be returned unless the application supplies a custom one) and at 4, we see that error handler being used to configure error handling for the newly created `NgZone`. Finally at 5, we see the call to the actual bootstrapping.

So in summary, Angular application developers should clearly learn about zones, since that is the execution context within which their application code will run.

API Model

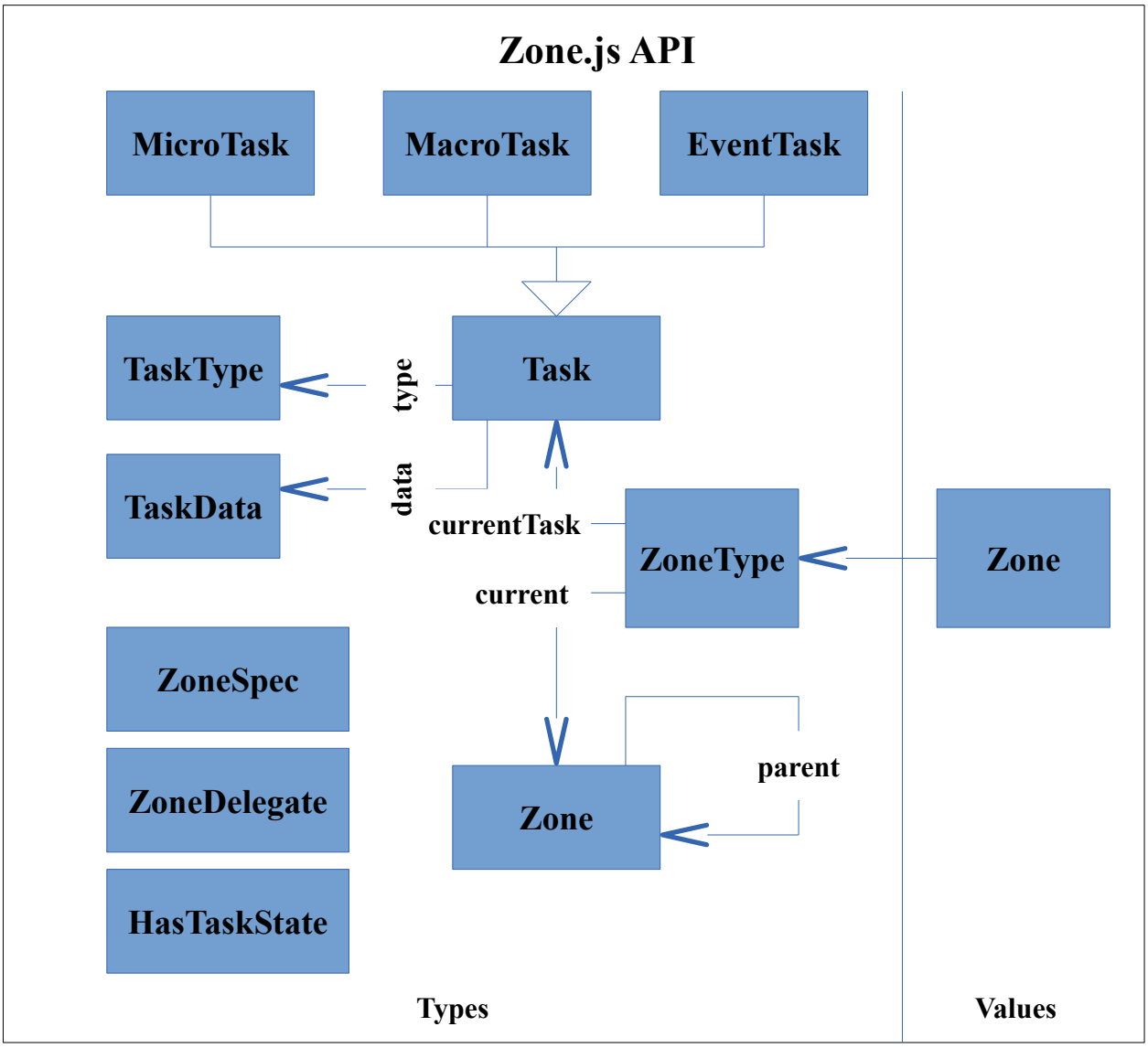
Zone.js exposes an API for applications to use in the <ZONE-MASTER>/dist/zone.js.d.ts file.

The two main types it offers are for tasks and zones, along with some helper types. A zone is a (usually named) asynchronous execution context; a task is a block of functionality (may also be named). Tasks run in the context of a zone.

Zone.js also supplies a const value, also called `Zone`, of type `ZoneType`:

```
interface ZoneType {
    /**
     * @returns {Zone} Returns the current [Zone]. The only way to change
     * the current zone is by invoking a run() method, which will update the
     * current zone for the duration of the run method callback.
     */
    current: Zone;
    /**
     * @returns {Task} The task associated with the current execution.
     */
    currentTask: Task | null;
    /**
     * Verify that Zone has been correctly patched.
     * Specifically that Promise is zone aware.
     */
```

```
    assertZonePatched(): void;
    /**
     *  Return the root zone.
     */
    root: Zone;
}
declare const Zone: ZoneType;
```



Recall that TypeScript has distinct declaration spaces for values and types, so the `Zone` value is distinct from the `Zone` type. For further details, see the TypeScript Language Specification – Section 2.3 – Declarations:

- https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#2.3

Apart from being used to define the `Zone` value, `ZoneType` is not used further.

When your application code wishes to find out the current zone it simply uses `Zone.current`, and when it wants to discover the current task within that zone, it

uses `Zone.currentTask`. If you need to figure out whether Zone.js is available to your application (it will be for Angular applications), then just make sure `Zone` is not undefined. If we examine:

- <ANGULAR-MASTER>/packages/core/src/zone/ng_zone.ts

– we see that is exactly what Angular's NgZone.ts does:

```
constructor({enableLongStackTrace = false}) {
   if (typeof Zone == 'undefined') {
     throw new Error(`In this configuration Angular requires Zone.js`);
   }
```

Two simple helper types used to define tasks are `TaskType` and `TaskData`. `TaskType` is just a human-friendly string to associate with a task. It is usually set to one of the three task types as noted in the comment:

```
/**
 * Task type: `microTask`, `macroTask`, `eventTask`.
 */
declare type TaskType = 'microTask' | 'macroTask' | 'eventTask';
```

`TaskData` contains a boolean (is this task periodic, i.e. is to be repeated) and two numbers - delay before executing this task and a handler id from `setTimout`.

```
interface TaskData {
    /**
     * A periodic [MacroTask] is such which get automatically
     * rescheduled after it is executed.
     */
    isPeriodic?: boolean;
    /**
     * Delay in milliseconds when the Task will run.
     */
    delay?: number;
    /**
     * identifier returned by the native setTimeout.
     */
    handleId?: number;
}
```

A task is an interface declared as:

```
interface Task {
    type: TaskType;
    state: TaskState;
    source: string;
    invoke: Function;
    callback: Function;
    data?: TaskData;
    scheduleFn?: (task: Task) => void;
    cancelFn?: (task: Task) => void;
    readonly zone: Zone;
    runCount: number;
    cancelScheduleRequest(): void;
}
```

There are three marker interfaces derived from `Task`:

```
interface MicroTask extends Task { type: 'microTask'; }
```

```
interface MacroTask extends Task { type: 'macroTask'; }
interface EventTask extends Task { type: 'eventTask'; }
```

The comments for Task nicely explains their purpose:

```
* - [MicroTask] queue represents a set of tasks which are executing right
*   after the current stack frame becomes clean and before a VM yield. All
*   [MicroTask]s execute in order of insertion before VM yield and the next
*   [MacroTask] is executed.
* - [MacroTask] queue represents a set of tasks which are executed one at a
*   time after each VM yield. The queue is ordered by time, and insertions
*   can happen in any location.
* - [EventTask] is a set of tasks which can at any time be inserted to the
*   end of the [MacroTask] queue. This happens when the event fires.
```

There are three helper types used to define `Zone`. `HasTaskState` just contains booleans for each of the task types and a string:

```
declare type HasTaskState = {
    microTask: boolean;
    macroTask: boolean;
    eventTask: boolean;
    change: TaskType;
};
```

`ZoneDelegate` is used when one zone wishes to delegate to another how certain operations should be performed. So for forcking (creating new tasks), scheduling, intercepting, invoking and error handling, the delegate may be called upon to carry out the action.

```
interface ZoneDelegate {
    zone: Zone;
    fork(targetZone: Zone, zoneSpec: ZoneSpec): Zone;

    intercept(targetZone: Zone, callback: Function, source: string)
                                            : Function;
    invoke(targetZone: Zone, callback: Function, applyThis?: any, applyArgs?
                                            : any[], source?: string): any;
    handleError(targetZone: Zone, error: any): boolean;
    scheduleTask(targetZone: Zone, task: Task): Task;
    invokeTask(targetZone: Zone, task: Task, applyThis?: any, applyArgs?
                                            : any[]): any;
    cancelTask(targetZone: Zone, task: Task): any;
    hasTask(targetZone: Zone, isEmpty: HasTaskState): void;
}
```

`ZoneSpec` is an interface that allows implementations to state what should have when certain actions are needed. It uses `ZoneDelegate` and the current zone:

```
interface ZoneSpec {
    name: string;
    properties?: {
        [key: string]: any;
    };
    onFork?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
                          targetZone: Zone, zoneSpec: ZoneSpec) => Zone;
    onIntercept?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
          targetZone: Zone, delegate: Function, source: string) => Function;
    onInvoke?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
              targetZone: Zone, delegate: Function, applyThis: any,
              applyArgs?: any[], source?: string) => any;
```

```
    onHandleError?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
              targetZone: Zone, error: any) => boolean;
    onScheduleTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
               targetZone: Zone, task: Task) => Task;
    onInvokeTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
     targetZone: Zone, task: Task, applyThis: any, applyArgs?: any[]) => any;
    onCancelTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
          targetZone: Zone, task: Task) => any;
    onHasTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
             targetZone: Zone, hasTaskState: HasTaskState) => void;
}
```

The definition of the `Zone` type is:

```
interface Zone {
    parent: Zone | null;
    name: string;
    get(key: string): any;
    getZoneWith(key: string): Zone | null;
    fork(zoneSpec: ZoneSpec): Zone;
    wrap<F extends Function>(callback: F, source: string): F;
    run<T>(callback: Function, applyThis?: any, applyArgs?: any[], source?
       : string): T;
    runGuarded<T>(callback: Function, applyThis?: any, applyArgs?: any[],
        source?: string): T;
    runTask(task: Task, applyThis?: any, applyArgs?: any): any;
    scheduleMicroTask(source: string, callback: Function, data?: TaskData,
        customSchedule?: (task: Task) => void): MicroTask;
    scheduleMacroTask(source: string, callback: Function, data?: TaskData,
        customSchedule?: (task: Task) => void,
         customCancel?: (task: Task) => void): MacroTask;
    scheduleEventTask(source: string, callback: Function, data?: TaskData,
       customSchedule?: (task: Task) => void,
       customCancel?: (task: Task) => void): EventTask;
    scheduleTask<T extends Task>(task: T): T;
    cancelTask(task: Task): any;
}
```

Relationship between Zone/ZoneSpec/ZoneDelegate interfaces

Think of `ZoneSpec` as the processing engine that controls how a zone works. It is a required parameter to the `Zone.fork()` method:

```
// Used to create a child zone.
// @param zoneSpec A set of rules which the child zone should follow.
// @returns {Zone} A new child zone.
fork(zoneSpec: ZoneSpec): Zone;
```

Often when a zone needs to perform an action, it uses the supplied `ZoneSpec`. Do you want to record a long stack trace, keep track of tasks, work with WTF (discussed later) or run async test well? For each a these a different `ZoneSpec` is supplied, and each offers different features and comes with different processing costs. Zone.js supplies one implementation of the `Zone` interface, and multiple implementations of the `ZoneSpec` interface (in <ZONE-MASTER>/lib/zone-spec). Application code with specialist needs could create a custom `ZoneSpec`.

An application can build up a hierarchy of zones and sometimes a zone needs to make a call into another zone further up the hierarchy, and for this a `ZoneDelegate` is used.

Source Tree Layout

The Zone.js source tree consists of a root directory with a number of files and the following immediate sub-directories:

- dist
- doc
- example
- scripts
- tests

The main source is in lib:

- lib

During compilation the source gets built into a newly created build directory.

Root directory

The root directory contains these markdown documentation files:

- CHANGELOG.md
- DEVELOPER.md
- MODULE.md
- NON-STANDARD-APIS.md
- README.md
- SAMPLE.md
- STANDARD-APIS.md

When we examine <ZONE-MASTER>/dist/zone.js.d.ts we see it is actually very well documented and contains plenty of detail to get us up and running writing applications that use Zone.js. From the DEVELOPER.md document we see the contents of dist is auto-generated (we need to explore how).

The root directory contains these JSON files:

- tslint.json
- tsconfig[-node|-esm-node|esm|].json
- package.json

There are multiple files starting with tsconfig – here are the compilerOptionsfrom the main one:

```
"compilerOptions": {
  "module": "commonjs",
  "target": "es5",
  "noImplicitAny": true,
  "noImplicitReturns": false,
  "noImplicitThis": false,
  "outDir": "build",
  "inlineSourceMap": true,
  "inlineSources": true,
  "declaration": false,
  "noEmitOnError": false,
  "stripInternal": false,
  "strict": true,
  "lib": [
```

```
      "es5",
      "dom",
      "es2015.promise",
      "es2015.symbol",
      "es2015.symbol.wellknown"
    ]
```

The package.json file contains metadata (including main and browser, which provide alternative entry points depending on whether this package 1 is loaded into a server [node] or a 2 browser app):

```
{
  "name": "zone.js",
  "version": "0.9.0",
  "description": "Zones for JavaScript",
1 "main": "dist/zone-node.js",
2 "browser": "dist/zone.js",
  "unpkg": "dist/zone.js",
  "typings": "dist/zone.js.d.ts",
  "files": [
    "lib",
    "dist"
  ],
  "directories": {
    "lib": "lib",
    "test": "test"
  },
```

and a list of scripts:

```
  "scripts": {
    "changelog": "gulp changelog",
    "ci": "npm run lint && npm run format && npm run promisetest
                          && npm run test:single && npm run test-node",
    "closure:test": "scripts/closure/closure_compiler.sh",
    "format": "gulp format:enforce",
    "karma-jasmine": "karma start karma-build-jasmine.conf.js",
    "karma-jasmine:es2015": "karma start karma-build-jasmine.es2015.conf.js",
    "karma-jasmine:phantomjs": "karma start
                    karma-build-jasmine-phantomjs.conf.js --single-run",
    "karma-jasmine:single": "karma start karma-build-jasmine.conf.js
                    --single-run",
    "karma-jasmine:autoclose": "npm run karma-jasmine:single
                    && npm run ws-client",
    "karma-jasmine-phantomjs:autoclose":
                    "npm run karma-jasmine:phantomjs && npm run ws-client",
    "lint": "gulp lint",
    "prepublish": "tsc && gulp build",
    "promisetest": "gulp promisetest",
    "promisefinallytest": "mocha promise.finally.spec.js",
    "ws-client": "node ./test/ws-client.js",
    "ws-server": "node ./test/ws-server.js",
    "tsc": "tsc -p .",
    "tsc:w": "tsc -w -p .",
    "tsc:esm2015": "tsc -p tsconfig-esm-2015.json",
    "tslint": "tslint -c tslint.json 'lib/**/*.ts'",
    // many test scripts
    ...
  },
```

It has no dependencies:

```
"dependencies": {},
```

It has many devDependencies.

The root directory also contains the MIT license in a file called LICENSE, along with the same within a comment in a file called LICENSE.wrapped.

It contains this file concerning bundling:

- webpack.config.js

This has the following content:

```
module.exports = {
  entry: './test/source_map_test.ts',
  output: {
    path: __dirname + '/build',
    filename: 'source_map_test_webpack.js'
  },
  devtool: 'inline-source-map',
  module: {
    loaders: [
      {test: /\.ts/, loaders: ['ts-loader'], exclude: /node_modules/}
    ]
  },
  resolve: {
    extensions: ["", ".js", ".ts"]
  }
}
```

Webpack is quite a popular bundler and ts-loader is a TypeScript loader for webpack. Details on both projects can be found here:

- https://webpack.github.io/
- https://github.com/TypeStrong/ts-loader

The root directory contains this file related to GIT:

- .gitignore

It contains this task runner configuration:

- gulpfile.js

It supplies a gulp task called “test/node” to run tests against the node version of Zone.js, and a gulp task “compile” which runs the TypeScript tsc compiler in a child process. It supplies many gulp tasks to build individual components and run tests.

All of these tasks result in a call to a local method `generateScript` which minifies (if required) and calls webpack and places the result in the dist sub-directory.

dist

This single directory contains all the output from the build tasks. The zone.d.ts file is the ambient declarations, which TypeScript application developers will want to use. This is surprisingly well documented, so a new application developer getting up to speed with zone.js should give it a careful read. A number of implementations of Zone

are provided, such as for the browser, for the server and for Jasmine testing:

- zone.js / zone.min.js
- zone-node.js
- jasmine-patch.js / jasmine-patch.min.js

Minified versions are supplied for the browser and jasmine builds, but not node. If you are using Angular in the web browser, then zone.js (or zone.min.js) is all you need Assuming you have created your Angular project using Angular CLI, it will automatically have set everything up correctly (no manual steps are needed to use Zone.js).

The remaining files in the dist directory are builds of different zone specs, which for specialist reasons you may wish to include – for example:

- async-test.js
- fake-async-test.js
- long-stack-trace-zone.js / long-stack-trace-zone.min.js
- proxy.js / proxy.min.js
- task-tracking.js / task-tracking.min.js
- wtf.js / wtf.min.js

We will look in detail at what each of these does later when examining the <ZONE-MASTER>/lib/zone-spec source directory.

example

The example directory contains a range of simple examples showing how to use Zone.js.

scripts

The script directory (and its sub-directories) contains these scripts:

- grab-blink-idl.sh
- closure/clousure_compiler.sh
- closure/closure_flagfile
- sauce/sauce_connect_setup.sh
- sauce/sauce_connect_block.sh

Source Model

The main source for Zone.js is in:

- <ZONE-MASTER>/lib

It contains a number of sub-directories:

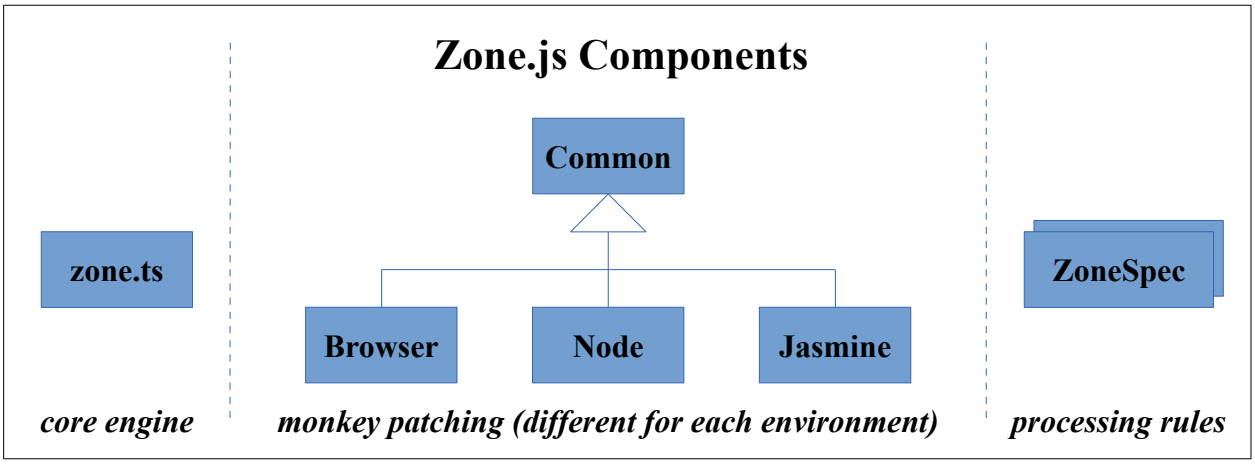
- browser
- closure
- common
- extra
- jasmine
- mix (a mix of browser and node)
- mocha
- node

- rxjs
- testing
- zone-spec

along with one source file:

- zone.ts

It is best to think of them arranged as follows:



To enable Zone.js to function, any JavaScript APIs related to asynchronous code execution must be patched – a Zone.js specific implementation of the API is needed. So for calls such as `setTimeout` or `addEventListener` and similar, Zone.js needs its own handling, so that when timeouts and events and promises get triggered, zone code runs first.

There are a number of environments supported by Zone.js that need monkey patching, such as the browser, the server (node) and Jasmine and each of these has a sub-direcory with patching code. The common patching code reside in the common directory. The core implementation of the Zone.js API (excluding `ZoneSpec`) is in zone.ts file. The additional directory is for zone specs, which are the configurable logic one can add to a zone to change its behavior. There are multiple implementations of these, and applications could create their own.

zone.ts

The first six hundred lines of the zone.ts file is the well-commented definition of the Zone.js API, that will end up in zone.d.ts. The slightly larger remainder of the file is an implementation of the `Zone` const:

```
const Zone: ZoneType = (function(global: any) {
...
  return global['Zone'] = Zone;
...
```

The `_ZonePrivate` interface defines private information that is managed per zone:

```
interface _ZonePrivate {
  currentZoneFrame: () => _ZoneFrame;
  symbol: (name: string) => string;
  scheduleMicroTask: (task?: MicroTask) => void;
```

```
    onUnhandledError: (error: Error) => void;
    microtaskDrainDone: () => void;
    showUncaughtError: () => boolean;
    patchEventTarget: (global: any, apis: any[], options?: any) => boolean[];
    patchOnProperties: (obj: any, properties: string[]|null,
            prototype?: any) => void;
    patchThen: (ctro: Function) => void;
    setNativePromise: (nativePromise: any) => void;
    patchMethod: ..
    bindArguments: (args: any[], source: string) => any[];
    patchMacroTask:
        (obj: any, funcName: string,
         metaCreator: (self: any, args: any[]) => any) => void;
    patchEventPrototype: (_global: any, api: _ZonePrivate) => void;
    isIEOrEdge: () => boolean;
    ObjectDefineProperty:
        (o: any, p: PropertyKey,
         attributes: PropertyDescriptor&ThisType<any>) => any;
    ObjectGetOwnPropertyDescriptor:
        (o: any, p: PropertyKey) => PropertyDescriptor | undefined;
    ObjectCreate(o: object|null, properties?
        : PropertyDescriptorMap&ThisType<any>): any;
  ..
}
```

When using the client API you will not see this, but when debugging through the Zone.js implementation, it will crop up from time to time.

A microtask queue is managed, which requries these variables:

```
let _microTaskQueue: Task[] = [];
let _isDrainingMicrotaskQueue: boolean = false;
let nativeMicroTaskQueuePromise: any;
```

`_microTaskQueue` is an array of microtasks, that must be executed before we give up our VM turn. `_isDrainingMicrotaskQueue` is a boolean that tracks if we are in the process of emptying the microtask queue. When a task is run within an existing task, they are nested and `_nativeMicroTaskQueuePromise` is used to access a native microtask queue. Which is stored as a global is not set. Two functions manage a microtask queue:

- scheduleMicroTask
- drainMicroTaskQueue

It also implements three classes:

- Zone
- ZoneDelegate
- ZoneTask

There are no implementations of `ZoneSpec` in this file. They are in the separate zone-spec sub-directory.

`ZoneTask` is the simplest of these classes:

```
class ZoneTask<T extends TaskType> implements Task {
    public type: T;
    public source: string;
```

```
public invoke: Function;
public callback: Function;
public data: TaskData|undefined;
public scheduleFn: ((task: Task) => void)|undefined;
public cancelFn: ((task: Task) => void)|undefined;
_zone: Zone|null = null;
public runCount: number = 0;
_zoneDelegates: ZoneDelegate[]|null = null;
_state: TaskState = 'notScheduled';
```

The constructor just records the supplied parameters and sets up `invoke`:

```
constructor(
    type: T, source: string, callback: Function,
    options: TaskData|undefined,
    scheduleFn: ((task: Task) => void)|undefined,
    cancelFn: ((task: Task) => void)|undefined) {
  this.type = type;
  this.source = source;
  this.data = options;
  this.scheduleFn = scheduleFn;
  this.cancelFn = cancelFn;
  this.callback = callback;
  const self = this;
  // TODO: @JiaLiPassion options should have interface
  if (type === eventTask && options && (options as any).useG) {
    this.invoke = ZoneTask.invokeTask;
  } else {
    this.invoke = function() {
      return ZoneTask.invokeTask.call(
                global, self, this, <any>arguments);
    };
  }
}
```

The interesting activity in here is setting up the `invoke` function. It increments the `_numberOfNestedTaskFrames` counter, calls `zone.runTask()`, and in a `finally` block, checks if `_numberOfNestedTaskFrames` is 1, and if so, calls the standalone function `drainMicroTaskQueue()`, and then decrements `_numberOfNestedTaskFrames`.

```
static invokeTask(task: any, target: any, args: any): any {
    if (!task) {
      task = this;
    }
    _numberOfNestedTaskFrames++;
    try {
      task.runCount++;
      return task.zone.runTask(task, target, args);
    } finally {
      if (_numberOfNestedTaskFrames == 1) {
        drainMicroTaskQueue();
      }
      _numberOfNestedTaskFrames--;
    }
}
```

A custom `toString()` implementation returns `data.handleId` (if available) or else the object's `toString()` result:

```
public toString() {
```

```
        if (this.data && typeof this.data.handleId !== 'undefined') {
          return this.data.handleId.toString();
        } else {
          return Object.prototype.toString.call(this);
        }
  }
```

`drainMicroTaskQueue()` is defined as:

```
  function drainMicroTaskQueue() {
      if (!_isDrainingMicrotaskQueue) {
        _isDrainingMicrotaskQueue = true;
        while (_microTaskQueue.length) {
          const queue = _microTaskQueue;
          _microTaskQueue = [];
          for (let i = 0; i < queue.length; i++) {
            const task = queue[i];
            try {
              task.zone.runTask(task, null, null);
            } catch (error) {
              _api.onUnhandledError(error);
            }
          }
        }
        _api.microtaskDrainDone();
        _isDrainingMicrotaskQueue = false;
      }
    }
```

The `_microTaskQueue` gets populated via a call to `scheduleMicroTask`:

```
  function scheduleMicroTask(task?: MicroTask) {
      // if we are not running in any task, and there has not been anything
      // scheduled we must bootstrap the initial task creation by manually
      // scheduling the drain
      if (_numberOfNestedTaskFrames === 0 && _microTaskQueue.length === 0) {
        // We are not running in Task, so we need to kickstart the
        // microtask queue.
        if (!nativeMicroTaskQueuePromise) {
          if (global[symbolPromise]) {
            nativeMicroTaskQueuePromise = global[symbolPromise].resolve(0);
          }
        }
        if (nativeMicroTaskQueuePromise) {
          let nativeThen = nativeMicroTaskQueuePromise[symbolThen];
          if (!nativeThen) {
            // native Promise is not patchable, we need to use `then` directly
            // issue 1078
            nativeThen = nativeMicroTaskQueuePromise['then'];
          }
          nativeThen.call(nativeMicroTaskQueuePromise, drainMicroTaskQueue);
        } else {
          global[symbolSetTimeout](drainMicroTaskQueue, 0);
        }
      }
      task && _microTaskQueue.push(task);
    }
```

If needed (not running in a task), this calls `setTimeout` with timeout set to 0, to enqueue a request to drain the microtask queue. Even though the timeout is 0, this

does not mean that the `drainMicroTaskQueue()` call will execute immediately. Instead, this puts an event in the JavaScript's event queue, which after the already scheduled events have been handled (there may be one or move already in the queue), will itself be handled. The currently executing function will first run to completion before any event is removed from the event queue. Hence in the above code, where `scheduleQueueDrain()` is called before `_microTaskQueue.push()`, is not a problem. `_microTaskQueue.push()` will execute first, and then sometime in future, the `drainMicroTaskQueue()` function will be called via the timeout.

The `ZoneDelegate` class has to handle eight scenarios:

- fork
- intercept
- invoke
- handleError
- scheduleTask
- invokeTask
- cancelTask
- hasTask

It defines variables to store values for a `ZoneDelegate` and `ZoneSpec` for each of these, which are initialized in the constructor.

```
private _interceptDlgt: ZoneDelegate|null;
private _interceptZS: ZoneSpec|null;
private _interceptCurrZone: Zone|null;
```

`ZoneDelegate` also declares three variables, to store the delegates zone and parent delegate, and to represent task counts (for each kind of task):

```
public zone: Zone;

private _taskCounts: {microTask: number,
                      macroTask: number,
                      eventTask: number}
                     = {'microTask': 0, 'macroTask': 0, 'eventTask': 0};

private _parentDelegate: ZoneDelegate|null;
```

In `ZoneDelegate`'s constructor, the `zone` and `parentDelegate` fields are initialized to the supplied parameters, and the `ZoneDelegate` and `ZoneSpec` fields for the eight scenarios are set (using TypeScript type guards), either to the supplied `ZoneSpec` (if not null), or the parent delegate's:

```
constructor(
  zone: Zone,
  parentDelegate: ZoneDelegate|null, zoneSpec: ZoneSpec|null
){
  this.zone = zone;
  this._parentDelegate = parentDelegate;

  this._forkZS = zoneSpec
  && (zoneSpec && zoneSpec.onFork ? zoneSpec : parentDelegate!._forkZS);
  this._forkDlgt = zoneSpec
  && (zoneSpec.onFork ? parentDelegate : parentDelegate!._forkDlgt);
  this._forkCurrZone = zoneSpec
  && (zoneSpec.onFork ? this.zone : parentDelegate!.zone);
```

The `ZoneDelegate` methods for the eight scenarios just forward the calls to the selected `ZoneSpec` (pr parent delegate) and does some house keeping. For example, the invoke method checks if `_invokeZS` is defined, and if so, calls its `onInvoke`, otherwise it calls the supplied callback directly:

```
invoke(
     targetZone: Zone, callback: Function,
     applyThis: any, applyArgs?: any[],
     source?: string): any {
   return this._invokeZS ? this._invokeZS.onInvoke!
                  (this._invokeDlgt!,
                   this._invokeCurrZone!,
                   targetZone, callback,
                   applyThis, applyArgs, source) :
                           callback.apply(applyThis, applyArgs);
 }
```

The `scheduleTask` method is a bit different, in that it first **1** tries to use the `_scheduleTaskZS` (if defined), otherwise **2** tries to use the supplied task's `scheduleFn` (if defined), otherwise **3** if a microtask calls `scheduleMicroTask()`, otherwise **4** it is an error:

```
scheduleTask(targetZone: Zone, task: Task): Task {
      let returnTask: ZoneTask<any> = task as ZoneTask<any>;
      if (this._scheduleTaskZS) {
1       if (this._hasTaskZS) {
          returnTask._zoneDelegates!.push(this._hasTaskDlgtOwner!);
        }
        returnTask = this._scheduleTaskZS.onScheduleTask!
                     (this._scheduleTaskDlgt!,
                      this._scheduleTaskCurrZone!, targetZone, task)
                      as ZoneTask<any>;
        if (!returnTask) returnTask = task as ZoneTask<any>;
      } else {
2       if (task.scheduleFn) {
          task.scheduleFn(task);
3       } else if (task.type == microTask) {
          scheduleMicroTask(<MicroTask>task);
4       } else {
          throw new Error('Task is missing scheduleFn.');  }
      }
      return returnTask;
 }
```

The fork method is where new zones get created. If `_forkZS` is defined, it is used, otherwise a new zone is created with the supplied `targetZone` and `zoneSpec`:

```
 fork(targetZone: Zone, zoneSpec: ZoneSpec): AmbientZone {
   return this._forkZS ? this._forkZS.onFork!(this._forkDlgt!,
                  this.zone, targetZone, zoneSpec) :
                          new Zone(targetZone, zoneSpec);      }
```

The internal variable `_currentZoneFrame` is initialized to the root zone and `_currentTask` to null:

```
let _currentZoneFrame: _ZoneFrame
                    = {parent: null, zone: new Zone(null, null)};
let _currentTask: Task|null = null;
let _numberOfNestedTaskFrames = 0;
```